

Для тех, кто хочет стать профессионалом

Измайлов К. К.

БАЗЫ ДАННЫХ на Python и ИИ

Статистика, аналитика,
большие данные и машинное обучение



Измайлов К. К.

Базы данных на Python и ИИ

*Статистика, аналитика, большие данные
и машинное обучение*



"Издательство Наука и Техника"

Санкт-Петербург

УДК 004.42 ББК 32.973

ISBN 978-5-6049658-8-7

Измайлов К. К.

Базы данных на Python и ИИ: статистика, аналитика, большие данные и машинное обучение — СПб.: Издательство Наука и Техника, 2026 г. — 400 с., ил.

Серия "Для тех, кто хочет стать профессионалом"

Эта книга – практическое руководство по работе с данными в Python, написанное для тех, кто хочет не просто «знать теорию», а реально работать с данными в повседневных задачах. Шаг за шагом мы пройдем полный жизненный цикл данных: от хранения в базах и написания запросов до анализа, оптимизации и осмысленной интерпретации результатов.

Главный акцент сделан на практике, здесь нет перегруженной теории и абстрактных рассуждений – только то, что действительно нужно в реальности. Почти все примеры можно сразу копировать, запускать и видеть результат, благодаря чему обучение становится наглядным и максимально прикладным уже с первых глав, а практические проекты и кейсы из реального опыта покажут, как отдельные инструменты складываются в полноценные аналитические пайплайны и как принимать инженерные решения в условиях больших таблиц, ограниченных ресурсов и бизнес-задач.

Примерно 70% каждой главы – это код с подробными комментариями и пояснениями. Читатель не просто узнаёт, что такое SQL, SQLite, PostgreSQL, MongoDB или pandas, а поймет, как и зачем эти инструменты используются в реальных проектах. Благодаря системному подходу, обучение начинается с простых и доступных решений (SQLite и базовый SQL), затем постепенно переходит к промышленным СУБД (PostgreSQL и MySQL), NoSQL-подходу на примере MongoDB и современным Python-инструментам — SQLAlchemy и pandas. Такой маршрут помогает увидеть экосистему данных целиком, а не как набор разрозненных технологий.

Особое внимание уделено работе с большими данными, ИИ, машинному обучению: научимся создавать эффективные аналитические ИИ-модели, оптимизировать большие объёмы данных, разберем приёмы, которые редко встречаются во вводных курсах (обработка данных порциями, стриминговые выборки, снижение потребления памяти, ускорение pandas-кода, использование эффективных форматов хранения и т.д.).

При этом книга не требует глубоких знаний или предварительного опыта (все темы вводятся постепенно, простым языком, с упором на практику) и будет полезна широкой аудитории:

- новичкам, которые хотят понять, как устроена аналитика данных;
- разработчикам, которым нужно лучше понимать SQL и хранение данных;
- аналитикам, желающим укрепить фундамент;
- студентам, изучающим базы данных, Python, статистику;
- всем, кто хочет мыслить не только «кодом», но и данными.

ISBN 978-5-6049658-8-7



9 785604 965887

Контактные телефоны издательства:

(812) 412 70 26

Официальный сайт: www.nit.com.ru

© Измайлов К. К.

© Издательство Наука и Техника

Содержание

Введение	19
Содержание книги	20
Для кого эта книга	22
 ЧАСТЬ I. Быстрый старт работы с базами данных.....	25
1.1. SQLite для первого шага	26
1.1.1. Создание базы и таблиц	27
Подключение к БД	27
Объект <i>cursor</i>	27
Создание первой таблицы	28
Заполняем таблицу	29
1.1.2. Первые запросы: выборка, фильтрация, сортировка.....	30
Выбор всех записей из таблицы.....	30
Выбор отдельных колонок	31
Фильтрация: выбираем только нужное.....	32
Сортировка.....	33
Комбинация фильтрации и сортировки	33
1.1.3. Итоги раздела	34
1.2. Подключение к PostgreSQL и MySQL.....	34
1.2.1. Подключение из Python.....	35
Подключение к PostgreSQL	35
Подключение к MySQL	37
1.2.2. Импорт CSV в базу	39
Чтение CSV в Python.....	40
Загрузка CSV в PostgreSQL	40
Загрузка CSV в MySQL.....	40
1.2.3. Итоги раздела	41
1.3. NoSQL (MongoDB)	41

1.3.1. Подключение к MongoDB через PyMongo	42
Установка и запуск MongoDB.....	42
Подготовка каталогов для данных	43
Запуск сервера	43
Работа с PyMongo	43
1.3.2. Простые аналитические операции.....	45
Получение всех документов	45
Фильтрация	45
Сортировка.....	45
Агрегация	46
Подсчет записей	46
1.3.3. Итоги раздела	46

ЧАСТЬ II. Python-инструменты для работы с данными 49

2.1. Работа с модулем SQLAlchemy	50
2.1.1. Быстрое создание моделей.....	50
Настройка базы	51
Определение модели	51
2.1.2. Запросы в стиле Python	52
Добавление данных	52
Получение всех товаров	52
Фильтрация	53
Сортировка.....	53
Комбинация условий	53
2.1.3. Итоги раздела	55
2.2. Pandas + SQL	56
2.2.1. Загрузка данных из базы в DataFrame.....	56
Подключение через SQLAlchemy	57
2.2.2. Аналитика и группировки.....	57
Фильтрация	57
Группировка и агрегаты.....	58
Несколько метрик сразу	58
Сортировка результатов.....	59

2.2.3. Объединение таблиц (аналог JOIN).....	59
2.2.4. Преобразование и новые поля.....	59
2.2.5. Выгрузка результатов обратно	60
2.2.6. Итоги раздела	61
2.3. Работа с большими данными.....	61
Создание тестовой базы данных	62
2.3.1. Стриминговая выборка из PostgreSQL	64
Чтение по одной строке	65
Чтение блоками (fetchmany)	65
Итерация по курсору	65
2.3.2. Обработка миллионов строк в <i>pandas</i>	66
Решение: <i>chunksize</i>	66
Агрегации	66
Группировки по частям.....	67
Когда <i>pandas</i> уже не хватает	67
2.3.3. Итоги раздела	69
2.4. Оптимизация и ускорение работы с данными.....	69
2.4.1. Пример таблицы и замеры времени.....	70
2.4.2. Оптимизация памяти: правильные типы данных.....	71
2.4.3. Векторизация вместо циклов.....	72
2.4.4. Ускорение фильтрации: <i>query()</i> и <i>eval()</i>	73
2.4.5. Оптимизация объединений и группировок.....	73
2.4.6. Параллельная обработка: <i>swifter</i> и <i>modin</i>	74
Swifter — для ускорения <i>apply()</i>	74
Modin — <i>pandas</i> на всех ядрах процессора	74
2.4.7. Ускорение ввода-вывода: Parquet и Feather	74
2.4.8. Использование индексов	75
2.4.9. Сравнение производительности	75
2.4.10. Мини-кейс: ускоряем отчёт по выручке	76
2.4.11. Итоги раздела	77
2.5. Хранение и форматы данных: CSV, Parquet, Feather.....	77
2.5.1. CSV — универсальный, но медленный	78
Пример записи и чтения CSV	78

Пример сжатия CSV	79
2.5.2. Parquet — формат для аналитики.....	79
Пример записи и чтения Parquet.....	79
Сравнение CSV и Parquet по скорости.....	80
2.5.3. Feather — формат для скорости	81
Пример записи и чтения Feather	81
2.5.4. Сравнение форматов	82
2.5.5. Практика: сравнение форматов на одной таблице.....	82
2.5.6. Практический пример: ежедневный отчёт.....	84
2.5.7. Итоги раздела	84
 ЧАСТЬ III. Статистика и аналитика	87
 3.1. Что такое статистика в аналитике	88
 3.2. Описательная статистика	89
3.2.1. Генеральная и выборочная совокупности.....	90
3.2.2. Центральные тенденции.....	90
Среднее арифметическое	90
Среднее выборочное и среднее генеральное	91
Медиана	91
Мода	92
Квантили.....	93
Практический пример: анализ среднего чека	93
3.2.3. Измерение разброса данных	95
Минимум и максимум	95
Размах (range).....	96
Дисперсия и стандартное отклонение	96
Почему делим на $(n - 1)$?.....	97
Квантили и квартильный размах	97
3.2.4. Работа с сегментами и группами	98
Средние значения по категориям товаров	99
Медианы по сегментам пользователей	99
Несколько метрик для каждой группы	100
Срезы по времени	100
Комбинированные группировки.....	101

3.2.5. Итоги раздела	102
3.3. Инструменты статистического анализа	102
3.3.1. Зачем нужны статистические тесты.....	103
Ошибки первого и второго рода	104
Роль уровня значимости (α).....	105
3.3.2. Проверка статистических гипотез	106
Тесты для сравнения средних	106
Одновыборочный t-тест	106
Двухвыборочный t-тест (независимые выборки)	107
Парный t-тест (зависимые выборки)	108
Манна – Уитни (U-тест)	108
Тесты для сравнения долей	109
Z-тест для пропорций.....	109
χ^2 (хи-квадрат) тест.....	109
Проверка нормальности распределения.....	110
Когда и какой тест использовать	111
3.3.3. Корреляционный анализ	111
Коэффициент корреляции Пирсона.....	112
Корреляция Спирмена	112
3.3.4. Итоги раздела	113
3.4. Регрессия и прогнозы.....	113
3.4.1. Линейная и логистическая регрессия (StatsModels, sklearn) .	114
Логистическая регрессия	116
3.4.2. Предсказание конверсии пользователей	117
Постановка задачи	117
Загрузка и первичный анализ.....	118
Построение модели (scikit-learn)	118
Качество модели.....	119
Интерпретация коэффициентов.....	120
Применение модели для новых пользователей	121
Бизнес-применение	121
3.4.3. Итоги раздела	121
3.5. Временные ряды	122
3.5.1. SQL + Pandas для анализа временных данных.....	123

Подготовка данных	123
Первичный анализ в SQL	124
Переход к Pandas	125
Агрегации в Pandas	125
Ежедневная динамика	126
Визуализация	126
3.5.2. Скользящие средние и сезонность	127
Скользящие средние (Moving Average)	128
Реализация в Pandas	128
Взвешенные скользящие средние	130
Сезонность	131
Сезонность по дням недели	132
Сезонность по месяцам	133
Детализация: сочетание SQL и Pandas	134
3.5.3. Предсказание временных рядов	135
Готовим временной ряд	135
Разделяем на train/test по времени	135
Метрики качества прогноза	136
Очень простой базовый прогноз (бенчмарк)	136
Экспоненциальное сглаживание	137
Простое экспоненциальное сглаживание (без тренда и сезонности)	137
Holt (уровень + тренд)	138
Holt – Winters (уровень + тренд + сезонность)	138
ARIMA/ SARIMA	139
SARIMA: пример подбора параметров вручную	139
Логарифмирование и стабилизация дисперсии	140
Как выбрать модель	140
Полный рабочий пример: сравнение моделей и выбор лучшей	141
3.5.4. Итоги раздела	142
3.6. Визуализация отчетов	143
3.6.1. Matplotlib	143
Добавляем стиль и оформление	145
3.6.2. Seaborn	150
3.6.3. Plotly	157
3.6.4. Итоги раздела	163

ЧАСТЬ IV. Практические проекты	165
4.1. Аналитика интернет-магазина	166
Генерация данных	167
4.1.1. RFM-анализ и сегментация клиентов	170
Присвоение RFM-оценок и сегментация	172
4.1.2. Анализ удержания клиентов (Retention)	177
4.1.3. Lifetime Value (LTV) и ценность клиента	182
4.1.4. Итоги раздела	185
4.2. A/B-тестирование продукта	186
4.2.1. Хранение результатов эксперимента	188
4.2.2. Статистический анализ	191
4.2.3. Итоги раздела	195
4.3. Анализ социальных сетей	196
4.3.1. Импорт и генерация данных	196
4.3.2. Графовый анализ пользователей	200
4.3.3. Поиск лидеров мнений	204
4.3.4. Итоги раздела	210
4.4. Анализ и прогнозирование финансовых данных	211
4.4.1. Генерация и загрузка котировок	212
4.4.2. Анализ временного ряда	214
4.4.3. Прогнозирование котировок	216
4.4.4. Итоги раздела	218
4.5. Справочник: классические продуктовые метрики	218
4.5.1. Метрики вовлеченности и аудитории	218
Stickiness — "липкость" продукта	221
4.5.2. Метрики активации и конверсий	222
Activation Rate — коэффициент активации	223
Conversion Rate — коэффициент конверсии	224
Воронка конверсий (Conversion Funnel)	224
4.5.3. Метрики удержания	225
Когортный анализ удержания	226
SQL: когортная таблица удержания по неделям	226

Python: расчет когортного удержания	227
Churn Rate — показатель оттока	228
4.5.4. Метрики монетизации	229
ARPU — Average Revenue Per User	229
ARPPU — Average Revenue Per Paying User	230
LTV — Lifetime Value (пожизненная ценность пользователя) ..	231
Revenue Retention — удержание выручки	233
4.5.5. Метрики роста	234
Growth Rate — темп роста пользователей	234
Net New Users — чистый прирост	235
DAU / MAU Ratio — "липкость" продукта на уровне роста ...	236
Revenue Growth Rate	238

ЧАСТЬ V. Автоматизация и аналитические решения 239

5.1. Автоматизация аналитики 240

5.1.1. Планирование и автоматизация SQL/Python задач	240
Генерация данных и подготовка отчета	240
Создание скрипта <i>daily_report.py</i>	242
Cron	243
Airflow	244
Prefect	245
5.1.2. Как собирать отчеты автоматически (экспорт из Pandas в Excel, PDF, Google Sheets)	246
Экспорт в Excel	246
Экспорт в PDF	248
Экспорт отчета в Google Sheets	249
Настройка Google API	249
Автоматизация сборки отчетов	250
5.1.3. Отправка отчетов на email и в Telegram-бот	252
Отправка отчета по электронной почте	252
Отправка отчета в Telegram	253
5.1.4. Итоги раздела	258

5.2. Дашборды и визуальная аналитика 258

5.2.1. Основы построения дашбордов	258
--	-----

Принципы хорошего дашборда	259
Типы дашбордов	261
Какие графики использовать	261
Цвета и визуальные акценты	261
Ошибки при создании дашбордов	262
Инструменты для создания дашбордов	262
5.2.2. Создание интерактивного дашборда на Python: Plotly + Dash/ Streamlit	262
Подготовка данных	263
Интерактивный дашборд в Streamlit	264
Интерактивный дашборд в Dash (Plotly)	267
5.2.3. Итоги раздела	269
5.3. Очистка и подготовка данных	269
Типичные проблемы реальных данных	269
5.3.1. Обнаружение пропусков	270
Проверка наличия пропусков	271
Диагностическая информация о датафрейме	271
Обнаружение пропусков в отдельных строках	272
5.3.2. Работа с пропущенными значениями	272
Удаление пропусков	272
Заполнение числовых признаков	272
Заполнение категориальных признаков	273
Интерполяция (для временных рядов)	273
Заполнение на основе модели	274
Проверка результата	274
5.3.3. Обнаружение выбросов	275
Проверка распределения визуально	275
Обнаружение выбросов с помощью IQR (межквартильный размах)	276
Обнаружение выбросов с помощью Z-Score	276
Обнаружение выбросов в группах	277
Быстрая оценка распределения после удаления/исправления	277
5.3.4. Обработка выбросов	277
Удаление выбросов	278
Ограничение выбросов (Winsorization)	278

Логарифмирование	278
Преобразование степенями (корневые и степенные функции).....	279
Замена выбросов на групповые статистики	279
Моделирование выбросов	279
Быстрая проверка результата	280
5.3.5. Преобразование масштаба признаков	280
Стандартизация (Standardization, Z-score scaling)	281
Нормализация Min-Max.....	282
Когда использовать какой метод	282
Пример: сочетание логарифмирования и масштабирования	282
5.3.6. Итоги раздела	283
5.4. Инженерия признаков	284
5.4.1. Бакетирование (дискретизация) числовых признаков.....	286
Бакетирование фиксированными интервалами (pd.cut)	287
Квантильное бакетирование (pd.qcut).....	287
Бакетирование для бизнес-сегментации	288
Бакетирование по бизнес-логике	288
5.4.2. Линеаризация зависимостей и преобразование распределений.....	289
Логарифмирование	289
Корневые преобразования	290
Степенные преобразования	290
Box-Cox.....	290
Yeo-Johnson.....	291
Когда применять какое преобразование	291
5.4.3. Работа с категориальными признаками	292
One-Hot Encoding (OHE).....	292
Label Encoding.....	293
Target Encoding (Mean Encoding)	293
Риск утечки данных (data leakage)	294
Когда использовать какой метод	294
5.4.4. Создание новых признаков (Feature Construction).....	294
Арифметические комбинации признаков	295
Взаимодействующие признаки (interaction features).....	295
Агрегаты по группам.....	295

Работа с датами и временем.....	296
Складные признаки (rolling/moving window).....	296
Отношения и нормы	296
Бинаризация событий	297
5.4.5. Отбор и оценка качества признаков	297
Корреляционный анализ	298
Корреляция между признаком и целевой переменной.....	298
Feature Importance из деревьев решений и бустингов	298
Permutation Importance (перестановочная важность)	299
SHAP — интерпретируемая важность признаков	299
Удаление нерелевантных признаков	299
Итеративный отбор признаков (RFE).....	300
5.4.6. Итоги раздела	300
5.5. Основы машинного обучения	301
5.5.1. Обучение с учителем и без учителя	302
Обучение с учителем (Supervised Learning)	302
Обучение без учителя (Unsupervised Learning).....	303
5.5.2. Пример: прогноз оттока клиентов (Churn Prediction)	304
Подготовка данных	305
Разделение данных на обучение и тест	306
Обучение модели логистической регрессии	306
Интерпретация результатов.....	308
Проверка модели.....	308
Что можно улучшить.....	309
5.5.3. Оценка качества модели: accuracy, precision, recall, ROC AUC ..	309
Почему одной метрики недостаточно	309
Матрица ошибок (Confusion Matrix).....	310
Accuracy — общая точность	310
Precision и Recall — когда важна цена ошибки	311
ROC AUC — баланс чувствительности и точности.....	311
Расчет метрик на нашей модели оттока	312
Как выбирать метрику в реальной задаче.....	313
5.5.4. Итоги раздела	313
5.6. Инженерия данных и хранение аналитических систем.....	314
Установка Docker на Windows	314

Установка Docker на macOS	315
Запуск ClickHouse через Docker	315
Проверка работы ClickHouse	316
Проверка в браузере	316
Подключение из Python	316
Управление контейнером	316
5.6.1. Что такое ETL / ELT (извлечение, трансформация, загрузка) ..	317
Extract — извлечение данных	318
Transform — преобразование данных	319
Load — загрузка данных	320
ETL vs ELT — в чем разница	321
Автоматизация ETL	321
5.6.2. Пример пайплайна: выгрузка данных из PostgreSQL, очистка в Pandas, загрузка в ClickHouse	321
Подготовка окружения	322
Extract — извлекаем данные из PostgreSQL	323
Transform — очистка и подготовка данных в Pandas	324
Load — загрузка данных в ClickHouse	325
Проверка в ClickHouse	326
Запуск ETL по расписанию	326
5.6.3. Основы хранилищ данных: OLAP, кубы и партиции	327
OLTP и OLAP	327
Почему аналитика требует OLAP	327
Кубы данных	328
Партиционирование	329
Пример в ClickHouse	330
Партиции и хранение	330
5.6.4. Инструменты для больших объемов данных: ClickHouse, DuckDB, BigQuery	331
ClickHouse — локальный и производительный OLAP	331
Пример — анализ данных по регионам	332
DuckDB — легкий OLAP для локальной аналитики	332
Пример — анализ CSV без Pandas	333
Пример — использование с Pandas и Parquet	333
BigQuery — аналитика без серверов (serverless)	334
BigQuery ML	335
Как эти инструменты работают вместе	336

5.6.5. Итоги раздела	337
5.7. Практический проект: автоматизированная аналитическая система..	337
5.7.1. Архитектура проекта	338
Подготовка данных	338
Сбор данных (Extract).....	339
Обработка и расчет метрик (Transform).....	340
Загрузка в ClickHouse (Load)	341
Визуализация.....	342
Уведомления в Telegram	343
Автоматизация процесса	343
Полный листинг главы	343
5.7.2. Итоги раздела	350
5.8. Базы данных в эпоху искусственного интеллекта	350
5.8.1. Почему классических баз данных больше недостаточно.....	352
5.8.2. Новые типы данных.....	354
5.8.3. Векторные базы данных	357
5.8.4. Трансформация SQL.....	359
Первое важное изменение — SQL + ML.....	359
Следующий шаг — SQL + embeddings.....	360
Дальше появляется связка SQL + LLM.....	360
Изменение роли аналитика	361
5.8.5. Архитектура аналитики с AI	361
Следующий слой — Feature Store.....	362
Параллельно появляется Vector DB	362
Дальше — слой моделей: ML и LLM	363
И наконец, пользовательский слой — BI и чат-интерфейсы.....	363
5.8.6. Feature Store	364
5.8.7. RAG	366
5.8.8. Итоги раздела	368
ЧАСТЬ VI. Заключение и дальнейшие шаги	369
6.1. Аналитик данных: роли, специализации и путь роста.....	370
6.1.1. Аналитик данных как профессия	370
6.1.2. Data analyst: универсальный аналитик.....	371

6.1.3. Product analyst: аналитик, близкий к продукту	371
6.1.4. BI-аналитик: архитектор отчётности	372
6.1.5. Data engineer: инженерный путь из аналитики	372
6.1.6. Как “читать” вакансии.....	373
6.1.7. Траектории обучения: как двигаться осознанно	373
6.1.8. Итоги по выбору	374
6.2. Заключение.....	374
6.3. Дальнейшие шаги	375
6.4. Финальные слова.....	376

ПРИЛОЖЕНИЕ I. Подготовка окружения для работы с книгой 377

Установка Miniconda.....	378
Создание окружения для книги.....	379
Устанавливаем нужные пакеты	379
Регистрируем ядро в Jupyter.....	379
Проверка окружения	380
Мини-тест SQLite	380
Краткая памятка команд.....	380

ПРИЛОЖЕНИЕ II. Краткий справочник по SQL 383

Быстрый старт: выборка, фильтрация, сортировка	384
Агрегации и условные подсчёты.....	385
JOIN-шаблоны	386
Оконные функции (window functions).....	387
Топ-N388	
Дедупликация и выбор «последней версии»	389
Даты и время.....	390

Когортный анализ (retention - месяц регистрации × месяц жизни)	390
RFM-сегментация (Recency / Frequency / Monetary).....	391
«Gaps and Islands» (непрерывные последовательности)	392
Сессии: разбиение по таймауту (30 минут)	393
Воронка	393
Работа со строками/JSON (PostgreSQL)	394
Временные таблицы и CTE (common table expressions).....	395
Список использованных источников информации	397

Об авторе книги

Константин Измайлов – практикующий специалист с многолетним опытом работы в технологических IT-компаниях, прошедший полный путь аналитической карьеры: от прикладной разработки и анализа данных до руководящих и стратегических ролей. В разные годы он работал в крупных компаниях, включая VK, Yandex и T-Bank, а также в международных IT-стартапах, где занимался построением аналитических систем, разработкой и управлением командами, сочетая управленческий опыт с глубоким техническим бэкграундом.

Значительную часть профессионального пути автора составляет преподавательская деятельность: более пяти лет он преподавал в ведущих технических вузах Москвы, основным местом работы был факультет ВМК МГУ. В рамках сотрудничества с вузами были разработаны и выпущены образовательные программы и онлайн-курсы по Python, аналитике данных и работе с базами данных, ориентированные на практическую подготовку студентов и специалистов.

Профессиональные интересы автора охватывают аналитику данных, статистику и машинное обучение. Он участвовал в соревнованиях по спортивному машинному обучению на платформе Kaggle, что сформировало прикладной и инженерный подход к анализу данных: внимание к качеству моделей, воспроизводимости решений и работе с реальными, несовершенными данными.

Эта книга отражает многолетний опыт работы с данными и ориентирована на формирование профессионального взгляда на аналитику и базы данных в Python.

Введение

Данные стали главной движущей силой современной цифровой эпохи. Если раньше информация рассматривалась лишь как побочный продукт процессов: бухгалтерии, торговли, экспериментов, то сегодня она стала основным активом. Компании принимают решения, анализируя огромные массивы чисел, алгоритмы машинного обучения подстраивают сервисы под каждого пользователя, государства прогнозируют поведение экономик через статистические модели.

Мир работает на данных, и способность понимать их, систематизировать, анализировать и извлекать смысл — это не просто навык, а фундаментальная компетенция XXI века.

Эта книга — практическое и пошаговое путешествие в мир данных: от баз данных и SQL до современной аналитики, Python-инструментов, статистики и больших данных. Мы будем идти от простого к сложному, выстраивая системное понимание того, как информация живёт в реальных проектах: как она хранится, как обрабатывается, как превращается в знания и решения.

Чтобы данные стали полезными, их нужно уметь хранить, извлекать, комбинировать, очищать, анализировать, агрегировать, формализовать. Именно этому и посвящена книга.

Содержание книги

Книга выстроена в виде последовательного, логичного и практически ориентированного путешествия по миру данных. Она начинается с реляционных баз и SQL, потому что именно они формируют фундамент аналитики. В первой части читатель знакомится со SQLite — простой, лёгкой системой, которая идеально подходит для первого контакта с таблицами, строками, схемами и запросами. На её примере создаются структуры данных, форми-

руются выборки, выполняются сортировки, фильтрации и аналитические запросы. Даже если SQL раньше казался сложным, уже после первых шагов становится понятно, что работа с базой данных — это логичный и очень понятный процесс.

Развивая этот фундамент, книга переходит к промышленным СУБД — PostgreSQL и MySQL. Это те системы, которые лежат в основе интернет-магазинов, банковских сервисов, CRM-платформ и сотен тысяч корпоративных решений. Читатель учится подключаться к ним через Python, создавать базы и таблицы, загружать реальные данные из CSV и понимать архитектуру клиент-сервер. Таким образом, появляется важное инженерное понимание того, как устроено хранение данных в настоящих проектах, и чем «большие» базы отличаются от встроенных решений.

Следующий шаг — переход к NoSQL и MongoDB. Здесь читатель узнаёт, что данные могут храниться не только в жёстких таблицах, но и в гибких документах, которые позволяют моделировать более живые, меняющиеся структуры. MongoDB демонстрирует другой подход к работе с информацией: вместо схем — коллекции, вместо строк — JSON-подобные объекты. Это расширяет профессиональный кругозор и показывает, что выбор базы всегда зависит от того, какие задачи решает система.

Вторая часть книги посвящена современным Python-инструментам. Здесь появляется SQLAlchemy — библиотека, которая позволяет работать с базами в стиле Python, описывать таблицы как классы и формировать запросы без ручного SQL. Такой подход делает код чище, понятнее и проще в сопровождении. Затем в центре внимания оказывается pandas — главный инструмент аналитика. На его основе читатель учится загружать данные из SQL, выполнять фильтрации, группировки, агрегации, объединять таблицы, строить метрики и отчёты. Pandas становится тем мостом, который соединяет мир данных и мир анализа.

Постепенно книга подводит к важному препятствию: данные редко бывают маленькими. Когда таблицы начинают содержать миллионы строк, привычные методы перестают работать. Поэтому отдельная глава посвящена техникам работы с большими объёмами: стриминговой выборке, порционной загрузке, агрегациям по частям и продуманному использованию памяти. Читатель учится не только анализировать данные, но и контролировать масштаб, оптимизировать выполнение и избегать перегрузки ресурсов.

После этого логично следует глава об оптимизации. Даже обычные `DataFrame` могут работать медленно, если использовать их неэффективно. Здесь подробно разбирается, как ускорить вычисления в разы: правильным выбором типов данных, векторизацией вместо циклов, использованием ускоренных вычислительных движков, параллельных библиотек и продвинутых возможностей `pandas`. Это формирует инженерное мышление и позволяет работать быстрее, экономя время и ресурсы.

Ещё один важный блок посвящён форматам хранения данных. `CSV`, хотя и привычен, далеко не всегда подходит для серьёзной аналитики. Книга показывает преимущества современных форматов вроде `Parquet` и `Feather`, которые позволяют читать данные в разы быстрее, экономят память и идеально подходят для аналитических пайплайнов. Благодаря этому читатель понимает, как правильно организовывать хранение данных в реальных проектах, а также чем руководствоваться при выборе формата.

Завершается книга большим разделом по статистике — инструменту, который превращает данные из наборов чисел в источник знаний. Здесь подробно и доступно объясняется, что такое распределения, средние значения, разбросы, корреляции, доверительные интервалы и статистическая значимость. Читатель учится формулировать гипотезы, проверять их, правильно интерпретировать результаты и избегать типичных ловушек анализа. Этот раздел превращает технические навыки в аналитическое мышление, без которого невозможно принимать осознанные решения.

Таким образом, структура книги охватывает весь путь аналитика: от понимания того, где и как живут данные, через инструменты обработки и оптимизации, до статистики и интерпретации. Каждая часть выстраивает у читателя целостное видение профессии и учит не просто работать с данными, а понимать их глубинную логику.

Для кого эта книга

Она подойдёт:

- новичкам, которые хотят понять, как устроена аналитика данных;
- разработчикам, которым нужно лучше понимать `SQL` и хранение данных;
- аналитикам, желающим укрепить фундамент;

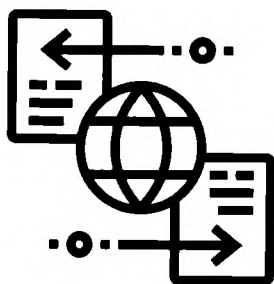
- студентам, изучающим базы данных, Python, статистику;
- всем, кто хочет мыслить не только «кодом», но и данными.

При этом книга не требует глубоких знаний математики или предварительного опыта: все темы вводятся постепенно, простым языком, с опорой на практику.

Эта книга — не просто сборник технологий или справочник по функциям. Это навигационная карта для путешествия в аналитический мир. Вы будете учиться не только инструментам, но и подходу к данным, стилю мышления, умению задавать вопросы и искать ответы так, как это делают профессиональные аналитики.

Открывая следующую страницу, вы начинаете путь, который изменит то, как вы видите данные, — а значит, изменит и то, как вы принимаете решения.

Добро пожаловать в мир, где каждая цифра имеет значение!



Часть I.

Быстрый старт работы с базами данных

1.1. SQLite для первого шага

Базы данных часто ассоциируются с чем-то сложным: мощные серверы, множество настроек, системные администраторы, гигабайты информации и тысячи пользователей. В крупных проектах действительно применяются такие системы, как PostgreSQL или MySQL. Они позволяют хранить и обрабатывать огромные массивы данных, обеспечивать отказоустойчивость и работу в многопользовательской среде.

Но для начала изучения SQL и баз данных вовсе не требуется поднимать сервер или осваивать сложные конфигурации. Нам достаточно простой и лёгкой системы, которая позволит сосредоточиться на самом главном — на работе с таблицами, строками и запросами.

Такой системой является SQLite. Это лёгкая реляционная база данных, где все данные хранятся в одном файле на диске.

Главные особенности SQLite:

- не требует отдельного сервера — достаточно одного файла;
- встроена в Python "из коробки" через модуль `sqlite3`;
- готова к работе сразу после подключения — можно писать код и выполнять запросы без дополнительных установок.

SQLite удобна для изучения SQL, для небольших проектов и прототипов, а также для приложений, где не предполагается высокая нагрузка. Например, мобильные и настольные программы часто используют SQLite как встроенное хранилище данных.

При этом все навыки работы с SQL, которые мы получим на SQLite, будут полностью применимы и в "больших" базах данных вроде PostgreSQL или MySQL. Синтаксис языка везде одинаков: те же SELECT, WHERE, JOIN и другие конструкции.

1.1.1. Создание базы и таблиц

Подключение к БД

Для начала работы с SQLite нужно установить соединение с базой данных. В Python это делается с помощью встроенного модуля `sqlite3`:

```
import sqlite3
# Создаем (или открываем, если он уже существует) файл базы данных
conn = sqlite3.connect("shop.db")
```

Команда `connect()` возвращает объект `conn` — соединение с базой. Его можно воспринимать как "канал связи" между вашей программой и файлом базы данных. Всё взаимодействие с таблицами и запросами будет происходить именно через это соединение.

Объект *cursor*

Нам нужно как-то передавать команды. Для этого далее мы создаем объект `cursor`:

```
cursor = conn.cursor()
```

Курсор — это интерфейс между программой и базой данных. Все SQL-команды отправляются в базу именно через него, а результаты запросов возвращаются обратно тем же путём.

Можно думать о курсоре как об "указателе" внутри базы: он задаёт, с какими данными вы сейчас работаете и какие операции выполняете.

Создание первой таблицы

Теперь начнём строить таблицу. Пусть у нас будет база данных для простого интернет-магазина.

```
cursor.execute("""
CREATE TABLE IF NOT EXISTS products (
    id INTEGER PRIMARY KEY AUTOINCREMENT,
    name TEXT NOT NULL,
    category TEXT,
    price REAL,
    stock INTEGER
)
""")
conn.commit()
```

Смотрим на первую строку: `CREATE TABLE IF NOT EXISTS products` — мы говорим базе: "Создай мне таблицу с именем `products`, но только если такой ещё нет". Это защита от ошибок: если таблица уже существует, новый дубль не появится.

Дальше идут колонки:

- `id INTEGER PRIMARY KEY AUTOINCREMENT` — это уникальный номер для каждой строки. База сама увеличивает значение, вам не придётся его указывать при создании новой записи.
- `name TEXT NOT NULL` — название товара. Оно обязательно для заполнения: без имени строка просто не запишется.
- `category TEXT` — категория, например "Одежда" или "Электроника".
- `price REAL` — цена. Тип `REAL` значит, что число может быть с десятичной частью.
- `stock INTEGER` — остаток на складе, простое целое число.

И наконец, строка `conn.commit()`. Это как кнопка "Сохранить". Все изменения, которые мы сделали (создание таблицы, добавление данных), сохраняются в файл базы. Если забыть вызвать `commit()`, таблица исчезнет сразу после закрытия программы.

И вот у нас уже есть структура, куда можно добавлять данные.

Заполняем таблицу

```
cursor.executemany("""
INSERT INTO products (name, category, price, stock)
VALUES (?, ?, ?, ?)
""", [
    ("Ноутбук", "Электроника", 75000, 15),
    ("Смартфон", "Электроника", 45000, 30),
    ("Кроссовки", "Одежда", 5000, 50),
    ("Книга", "Книги", 800, 100),
    ("Футболка", "Одежда", 1200, 70)
])
conn.commit()
```

Мы добавили пять записей. Согласитесь, уже похоже на настоящий каталог товаров!

Полный листинг программы приведен ниже.

Листинг 1.1. Создание базы данных и таблиц

```
# Подключаем встроенный модуль для работы с SQLite
import sqlite3
# Создаем (или открываем) файл базы данных "shop.db".
# Это обычный файл на диске, в котором будет храниться таблица
conn = sqlite3.connect("shop.db")
# Создаем объект cursor
cursor = conn.cursor()
# Создаем таблицу products (если её ещё нет).
# В ней будут храниться товары интернет-магазина
cursor.execute("""
CREATE TABLE IF NOT EXISTS products (
-- Уникальный номер (ключ), инкрементируется автоматически
    id INTEGER PRIMARY KEY AUTOINCREMENT,
-- Название товара, строка, не может быть пустым
    name TEXT NOT NULL,
```

```

-- Категория (например, "Электроника" или "Одежда")
    category TEXT,
-- Цена товара, вещественное число
    price REAL,
-- Остаток на складе, целое число
    stock INTEGER
)
"""
# Сохраняем изменения (без commit таблица не появится в файле базы)
conn.commit()
# Добавляем несколько товаров сразу
# executemany выполняет запрос для каждой строки списка
cursor.executemany("""
INSERT INTO products (name, category, price, stock)
VALUES (?, ?, ?, ?)
""", [
    ("Ноутбук", "Электроника", 75000, 15),
    ("Смартфон", "Электроника", 45000, 30),
    ("Кроссовки", "Одежда", 5000, 50),
    ("Книга", "Книги", 800, 100),
    ("Футболка", "Одежда", 1200, 70)
])
# Снова сохраняем изменения
conn.commit()

```

1.1.2. Первые запросы: выборка, фильтрация, сортировка

Выбор всех записей из таблицы

У нас есть данные, и пора сделать несколько запросов SQL.

```

# Простейший запрос: "Покажи всё"
cursor.execute("SELECT * FROM products")
rows = cursor.fetchall()
for row in rows:
    print(row)

```


Результат будет такой:

```
(1, 'Ноутбук', 'Электроника', 75000.0, 15)
(2, 'Смартфон', 'Электроника', 45000.0, 30)
(3, 'Кроссовки', 'Одежда', 5000.0, 50)
(4, 'Книга', 'Книги', 800.0, 100)
(5, 'Футболка', 'Одежда', 1200.0, 70)
```

Мы видим все строки из таблицы.

Теперь хотелось бы сказать пару слов про новый метод, который мы использовали в коде, — `fetchall()`.

Когда мы пишем `cursor.execute("SELECT * FROM products")`, база данных действительно возвращает результат, но он пока остаётся внутри курсора. Чтобы получить его в Python, мы вызываем метод `fetchall()`. Этот метод берет все строки результата и складывает их в список кортежей. Каждый кортеж — это одна строка таблицы: сначала `id`, затем название, категория, цена и остаток.

Например, если в таблице пять товаров, то `fetchall()` вернёт список из пяти кортежей. Если будет сто строк — вернёт сто. И это очень удобно, когда данных немного: можно сразу перебрать их в цикле, вывести на печать или отправить на дальнейший анализ.

Нужно помнить только одно: `fetchall()` загружает в память все строки сразу. Для наших учебных примеров это прекрасно подходит. Но если в базе окажется миллион записей, стоит использовать другие методы (`fetchone()`, `fetchmany(n)` или просто цикл по курсору), чтобы забирать данные постепенно. Об этом мы поговорим немного позднее.

Выбор отдельных колонок

Допустим, нам не нужно всё сразу. Мы хотим лишь посмотреть названия и цены:

```
cursor.execute("SELECT name, price FROM products")
for row in cursor.fetchall():
    print(row)
```

Результат:

```
('Ноутбук', 75000.0)
('Смартфон', 45000.0)
('Кроссовки', 5000.0)
('Книга', 800.0)
('Футболка', 1200.0)
```

Уже удобнее: таблица превратилась в простой прайс-лист.

Фильтрация: выбираем только нужное

SQL позволяет нам делать выборку "по условию". Например, пусть нас интересуют товары дороже 10 000 рублей:

```
cursor.execute("""
    SELECT name, price
    FROM products
    WHERE price > 10000
""")
for row in cursor.fetchall():
    print(row)
```

Результат:

```
('Ноутбук', 75000.0)
('Смартфон', 45000.0)
```

А вот запрос по категории "Одежда":

```
cursor.execute("""
    SELECT *
    FROM products
    WHERE category = 'Одежда'
""")
for row in cursor.fetchall():
    print(row)
```

Результат:

```
(3, 'Кроссовки', 'Одежда', 5000.0, 50)
(5, 'Футболка', 'Одежда', 1200.0, 70)
```

Уже можно анализировать ассортимент!

Сортировка

Иногда важен не сам факт наличия товара, а его расположение по цене.

Сортируем по возрастанию:

```
cursor.execute("""
    SELECT name, price
    FROM products
    ORDER BY price ASC
""")
for row in cursor.fetchall():
    print(row)
```

Результат:

```
('Книга', 800.0)
('Футболка', 1200.0)
('Кроссовки', 5000.0)
('Смартфон', 45000.0)
('Ноутбук', 75000.0)
```

А если нужно от дорогих к дешёвым — используем DESC.

Комбинация фильтрации и сортировки

Давайте выберем все товары категории "Одежда" и отсортируем их по цене от самых дорогих к дешёвым:

```
cursor.execute("""
    SELECT name, price
    FROM products
    WHERE category = 'Одежда'
""")
```

```
ORDER BY price DESC
""")
for row in cursor.fetchall():
    print(row)
```

Результат:

```
('Кроссовки', 5000.0)
('Футболка', 1200.0)
```

Вот уже и простейшая аналитика нашего набора данных: видим, что кроссовки в этом сегменте дороже футболки.

1.1.3. Итоги раздела

- ◆ Мы подключились к базе SQLite и создали свой первый файл-базу.
- ◆ Разобрались с `cursor`.
- ◆ Создали таблицу, добавили в неё данные.
- ◆ Научились выполнять базовые запросы: выбирать все данные, вытаскивать только нужные колонки, фильтровать по условиям, сортировать результаты, комбинировать условия и сортировку.

1.2. Подключение к PostgreSQL и MySQL

SQLite хорош для первых шагов: всё хранится в одном файле, установка не нужна. Но в реальной работе часто применяются более серьезные системы управления базами данных (СУБД).

Среди них самые популярные — PostgreSQL и MySQL. Оба варианта используют модель клиент-сервер: база работает как отдельный процесс (часто даже на отдельном сервере), а приложение подключается к ней по сети.

- **PostgreSQL** — это современная и очень надёжная система управления базами данных. Она называется *объектно-реляционной*, потому что кроме классических таблиц и связей (как в любой реляционной базе), в ней можно работать со сложными типами данных: массивами, JSON-документами, гео Данными (через PostGIS) и даже собственными функциями. PostgreSQL умеет выполнять сложные аналитические запросы, поддерживает транзакции и расширяется с помощью плагинов и расширений. Именно поэтому её часто выбирают для проектов, где данные — это ключевая ценность: аналитические платформы, финансовые системы, государственные сервисы.
- **MySQL** — это тоже реляционная СУБД, но с другим фокусом. Её главное достоинство — простота установки и использования. Она отлично подходит для веб-приложений: блогов, интернет-магазинов, CRM-систем. MySQL поддерживается большинством хостингов "из коробки" (особенно в связке с PHP и популярными CMS вроде WordPress или Joomla). У неё есть большое сообщество, много документации и инструментов, поэтому начать работу с MySQL может даже новичок.

1.2.1. Подключение из Python

Для работы с этими СУБД в Python используются специальные драйверы (библиотеки).

- Для PostgreSQL — это пакет `psycopg2`.
- Для MySQL — часто используют `mysql-connector-python`.

Перед началом убедитесь, что они установлены:

```
pip install psycopg2-binary mysql-connector-python
```

Установку всех компонентов и программ мы будем рассматривать для macOS. Аналогичную инструкцию установки PostgreSQL на Windows можно легко найти в интернете.

Подключение к PostgreSQL

Перед началом работы нам непосредственно необходимо установить и запустить сервер PostgreSQL.

1. Загрузите дистрибутив Postgres.app: скачайте Postgres.app с официального сайта: <https://postgresapp.com/> и перетащите приложение в папку Applications.



Изображение 1.1. Официальный сайт Postgres.app

2. Запустите приложение: откройте Postgres.app — сервер PostgreSQL запустится автоматически в пользовательском окружении. В окне приложения вы увидите список текущих **кластеров** (баз данных), порт и версию PostgreSQL. По умолчанию создаётся база `postgres`, пользователь будет совпадать с вашим macOS-именем.

Теперь можно подключаться к базе данных через Python (библиотека `psycopg2`), pgAdmin или другие клиенты.

Листинг 1.2. Подключение к БД

```
import psycopg2
conn = psycopg2.connect(
    dbname="",           # Имя базы, если есть
    user="postgres",     # Имя пользователя
    password="",         # Пароль, если есть
    host="localhost",    # Адрес сервера базы
    port=5432            # Порт PostgreSQL по умолчанию
)
```

```

cursor = conn.cursor()
cursor.execute("""
CREATE TABLE IF NOT EXISTS products (
    id SERIAL PRIMARY KEY,
    name TEXT NOT NULL,
    category TEXT,
    price REAL,
    stock INTEGER
)
""")
conn.commit()
print("Таблица products успешно создана в PostgreSQL!")

```

Подключение к MySQL

Также нам необходимо установить и запустить сервер MySQL (аналогично тому, как мы запускали сервер для PostgreSQL).

Загрузите дистрибутив MySQL Community Server: перейдите на официальный сайт <https://dev.mysql.com/downloads/mysql/> и скачайте версию macOS DMG Archive (обычно это последняя стабильная Community Server).

1. Откройте загруженный .dmg-файл и установите сервер MySQL через мастер установки. Во время установки мастер предложит задать пароль для пользователя root. Сохраните его — он понадобится для подключения.
2. После установки в **System Preferences** (Системные настройки) появится панель MySQL. Откройте её и нажмите **Start MySQL Server** — сервер запустится автоматически. По умолчанию сервер слушает на порту 3306.

Листинг 1.3. Подключение к БД MySQL

```

import mysql.connector
# Подключаемся к MySQL (без указания базы)
conn = mysql.connector.connect(
    host="localhost",
    user="root",          # Измените на своего пользователя или оставьте root
    root

```



```
password="PASS"      # Пароль от MySQL, который вы указали во время
установки
)
cursor = conn.cursor()
# Создаём базу данных, если её ещё нет
cursor.execute(
    "CREATE DATABASE IF NOT EXISTS shopdb CHARACTER SET utf8mb4
COLLATE utf8mb4_unicode_ci;"
)
print("База данных shopdb создана или уже существует.")
cursor.close()
conn.close()
# Подключаемся уже к новой базе shopdb
conn = mysql.connector.connect(
    host="localhost",
    user="root",      # Измените на своего пользователя или оставьте
root
    password="PASS", # Пароль от MySQL
    database="shopdb"
)
cursor = conn.cursor()
# Создаем таблицу products
cursor.execute("""
CREATE TABLE IF NOT EXISTS products (
    id INT AUTO_INCREMENT PRIMARY KEY,
    name VARCHAR(255) NOT NULL,
    category VARCHAR(100),
    price FLOAT,
    stock INT
)
""")
conn.commit()
print("Таблица products успешно создана в MySQL!")
cursor.close()
conn.close()
```


1.2.2. Импорт CSV в базу

Обычно таблицы не заполняются вручную, как мы делали в примерах с SQLite. В реальной работе данные приходят из внешних источников — например, в формате CSV (*comma-separated values*, "значения, разделённые запятыми").

Создадим файл, который затем мы будем загружать в нашу базу данных.

```
import csv
# Данные для записи
rows = [
    ["name", "category", "price", "stock"],
    ["Ноутбук", "Электроника", 75000, 15],
    ["Смартфон", "Электроника", 45000, 30],
    ["Кроссовки", "Одежда", 5000, 50],
    ["Книга", "Книги", 800, 100],
    ["Футболка", "Одежда", 1200, 70]
]
# Создаем CSV-файл
with open("products.csv", "w", newline="", encoding="utf-8") as f:
    writer = csv.writer(f)
    writer.writerows(rows)
```

Также есть более популярный вариант с использованием библиотеки **pandas**.

```
import pandas as pd
# Создаем DataFrame
data = {
    "name": ["Ноутбук", "Смартфон", "Кроссовки", "Книга", "Футболка"],
    "category": ["Электроника", "Электроника", "Одежда", "Книги",
    "Одежда"],
    "price": [75000, 45000, 5000, 800, 1200],
    "stock": [15, 30, 50, 100, 70]
}
df = pd.DataFrame(data)
```

```
# сохраняем в CSV
df.to_csv("products.csv", index=False, encoding="utf-8")
```

Чтение CSV в Python

Для чтения CSV можно использовать стандартный модуль `csv` или более удобную библиотеку `pandas`.

Мы рассмотрим `pandas`, так как она позволяет сразу загрузить файл в `DataFrame`.

```
import pandas as pd
# Читаем CSV с товарами
df = pd.read_csv("products.csv")
print(df.head())
```

Загрузка CSV в PostgreSQL

Используя `psycopg2`, мы можем пройти по строкам `DataFrame` и вставить их в таблицу:

```
for _, row in df.iterrows():
    cursor.execute("""
        INSERT INTO products (name, category, price, stock)
        VALUES (%s, %s, %s, %s)
        """,
        (row['name'], row['category'], row['price'], row['stock'])
    )
conn.commit()
print("CSV данные загружены в PostgreSQL!")
```

Обратите внимание: в `psycopg2` для параметров запроса используются знаки `%s`.

Загрузка CSV в MySQL

В MySQL всё выглядит почти так же:

```
for _, row in df.iterrows():
    cursor.execute("""
        INSERT INTO products (name, category, price, stock)
        VALUES (%s, %s, %s, %s)
        """, (row['name'], row['category'], row['price'], row['stock']))
conn.commit()
print("CSV данные загружены в MySQL!")
```

У `mysql-connector` синтаксис подстановки параметров совпадает с PostgreSQL.

1.2.3. Итоги раздела

- ◆ Мы научились подключаться к PostgreSQL и MySQL из Python с помощью библиотек `psycopg2` и `mysql-connector-python`.
- ◆ Создали простую таблицу товаров в обеих СУБД.
- ◆ Разобрались, как загружать реальные CSV-данные в базу через `pandas` и циклы вставки.

Теперь мы умеем работать не только с маленькой встроенной SQLite, но и с "настоящими" промышленными СУБД.

1.3. NoSQL (MongoDB)

До этого мы работали с реляционными базами данных — SQLite, PostgreSQL, MySQL. Там все данные хранятся в таблицах, каждая строка жёстко подчиняется схеме: набор колонок фиксирован, у каждой есть свой тип. Это удобно, но не всегда гибко.

А что если у нас данные "живые", постоянно меняются по структуре? Например, у одних пользователей есть номер телефона, у других нет; у товара может быть набор характеристик, уникальный для каждой категории; у поста в соцсети могут быть лайки, репосты, комментарии — и всё в разных комбинациях.

Здесь решением становится NoSQL, и самая популярная система в этом семействе — MongoDB.

- MongoDB хранит данные не в таблицах, а в коллекциях документов.
- Документ — это, по сути, JSON-объект, то есть словарь с произвольным набором ключей и значений.
- Схема гибкая — в одной коллекции могут лежать документы с разными полями.

Это делает MongoDB удобной для быстрых прототипов, для работы с полуструктурированными данными и аналитики в реальном времени.

1.3.1. Подключение к MongoDB через PyMongo

Установка и запуск MongoDB

Прежде чем мы напишем первую строчку кода с PyMongo, нужно установить сам сервер базы данных на локальный компьютер. Без этого MongoDB попросту некуда будет подключаться.

1. Загрузите дистрибутив MongoDB с официального сайта:

<https://www.mongodb.com/try/download/community>

2. В параметрах выберите:

- » Edition: Community Server;
- » Version: последнюю доступную (например, 7.0);
- » OS: macOS;
- » Package: tgz (tarball-архив).

3. После скачивания у вас будет архив вида: `mongodb-macos-x86_64-7.0.0.tgz` (имя может отличаться по версии и архитектуре).

4. Откройте терминал и перейдите в папку с загрузкой:

```
cd ~/Downloads  
tar -xvf mongodb-macos-x86_64-7.0.0.tgz
```

5. Создайте системный каталог для MongoDB и копируйте туда файлы:

```
sudo mkdir -p /usr/local/mongodb
sudo cp -R -n mongodb-macos-x86_64-7.0.0/ /usr/local/mongodb
```

6. Чтобы запускать MongoDB из любого места, нужно прописать путь в переменную окружения. Для этого открываем файл ~/.zshrc и добавляем строку:

```
export PATH="/usr/local/mongodb/bin:$PATH"
source ~/.zshrc
mongod -version
```

Подготовка каталогов для данных

MongoDB хранит данные и логи в отдельных директориях. Создадим их вручную:

```
sudo mkdir -p /usr/local/var/mongodb
sudo mkdir -p /usr/local/var/log/mongodb
sudo chown -R $(whoami) /usr/local/var/mongodb
sudo chown -R $(whoami) /usr/local/var/log/mongodb
```

Запуск сервера

Теперь можно стартовать MongoDB:

```
mongod --dbpath /usr/local/var/mongodb --logpath /usr/local/var/log/
mongodb/mongo.log -fork
```

- --dbpath указывает путь для хранения данных;
- --logpath — файл для логов;
- --fork означает запуск в фоновом режиме.

После этого сервер работает в фоне, и мы можем подключаться к нему из Python через PyMongo.

Работа с PyMongo

Для работы с MongoDB из Python используют библиотеку PyMongo.

```
pip install pymongo
```

Теперь подключаемся:

```
from pymongo import MongoClient
# Подключаемся к локальному серверу MongoDB
client = MongoClient("mongodb://localhost:27017/")
# Выбираем базу данных
db = client["shopdb"]
# Создаем коллекцию (аналог таблицы)
products = db["products"]
```

Теперь у нас есть база `shopdb` и коллекция `products`.

Документы в MongoDB выглядят как Python-словарь. Добавим несколько товаров (те же, что и в наших примерах с SQL):

```
products.insert_many([
    {"name": "Ноутбук",
     "category": "Электроника",
     "price": 75000,
     "stock": 15},
    {"name": "Смартфон",
     "category": "Электроника",
     "price": 45000,
     "stock": 30},
    {"name": "Кроссовки",
     "category": "Одежда",
     "price": 5000,
     "stock": 50},
    {"name": "Книга",
     "category": "Книги",
     "price": 800,
     "stock": 100},
    {"name": "Футболка",
     "category": "Одежда",
     "price": 1200,
     "stock": 70}
])
```

MongoDB автоматически добавит к каждому документу уникальное поле `id`, которое будет использоваться как идентификатор.

1.3.2. Простые аналитические операции

Получение всех документов

Теперь посмотрим, как извлекать и анализировать данные.

```
# Получение всех документов
for doc in products.find():
    print(doc)
```

Результат будет напоминать JSON:

```
{"_id": ObjectId("..."), "name": "Ноутбук", "category": "Электроника",
"price": 75000, "stock": 15}
...
```

Фильтрация

Выбираем только товары категории "Одежда":

```
for doc in products.find({"category": "Одежда"}):
    print(doc["name"], doc["price"])
```

Результат:

```
Кроссовки 5000
Футболка 1200
```

Сортировка

Выбираем товары, отсортированные по цене (от дешевых к дорогим):

```
for doc in products.find().sort("price", 1):
    print(doc["name"], doc["price"])
```

Здесь 1 — сортировка по возрастанию, -1 — по убыванию.

Агрегация

MongoDB поддерживает мощный pipeline агрегации. Например, средняя цена товаров по категориям:

```
pipeline = [  
    {"$group": {"_id": "$category", "avg_price": {"$avg": "$price"}}}  
]  
for result in products.aggregate(pipeline):  
    print(result)
```

Пример результата:

```
{'_id': 'Электроника', 'avg_price': 60000.0}  
{'_id': 'Одежда', 'avg_price': 3100.0}  
{'_id': 'Книги', 'avg_price': 800.0}
```

Подсчет записей

Сколько всего товаров в базе:

```
count = products.count_documents({})  
print("Всего товаров:", count)
```

А сколько из них дороже 10000:

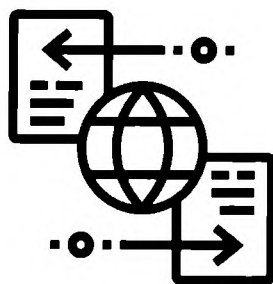
```
count = products.count_documents({"price": {"$gt": 10000}})  
print("Товаров дороже 10000:", count)
```

1.3.3. Итоги раздела

- ◆ Мы познакомились с NoSQL и MongoDB: здесь данные хранятся не в таблицах, а в коллекциях документов (JSON-подобных объектов).
- ◆ Подключились к базе через PyMongo, создали коллекцию и добавили документы.

- ♦ Научились выполнять базовые операции: выборка, фильтрация, сортировка.
- ♦ Познакомились с агрегирующими запросами: подсчёты и вычисление среднего значения.

MongoDB идеально подходит, когда данные слишком разнообразны или быстро меняются, и жёсткая схема только мешает. Она особенно полезна в проектах с пользовательским контентом (соцсети, логи, события) или когда вы работаете с "живыми" потоками данных.



Часть II.

Python-инструменты для работы с данными

2.1. Работа с модулем SQLAlchemy

До этого мы работали с базами напрямую через курсор: писали SQL-запросы в виде строк и отправляли их базе. Это полезно для понимания основ, но в реальных проектах такой подход быстро становится громоздким: запросы трудно поддерживать, много дублирования кода, легко сделать опечатку.

Здесь на помощь приходит SQLAlchemy — мощная библиотека, которая позволяет работать с базой данных в стиле Python-кода, почти не используя "сырой" SQL.

Главная идея SQLAlchemy:

- мы описываем таблицы как Python-классы (это называется ORM — Object Relational Mapping);
- потом работаем с объектами, а не с запросами в строках;
- библиотека сама переводит наш код в SQL и выполняет его в базе.

Это не значит, что SQL забывается насовсем: всегда можно выполнить "сырой" запрос. Но для большинства задач ORM намного удобнее и чище.

2.1.1. Быстрое создание моделей

Начнём с простого примера. Установим SQLAlchemy:

```
pip install sqlalchemy
```

Настройка базы

```
from sqlalchemy import create_engine
from sqlalchemy.orm import declarative_base, sessionmaker
# Создаём подключение к базе (для примера SQLite)
engine = create_engine("sqlite:///shop.db", echo=True)
# Базовый класс для моделей
Base = declarative_base()
# Создаём сессию
Session = sessionmaker(bind=engine)
session = Session()
```

Здесь:

- **engine** — объект, который знает, как подключаться к базе;
- **Base** — базовый класс для всех моделей;
- **Session** — "контекст работы" с базой: все запросы проходят через него.

Определение модели

Теперь опишем нашу таблицу **products** как Python-класс:

```
from sqlalchemy import Column, Integer, String, Float
class Product(Base):
    __tablename__ = "products" # Имя таблицы в базе
    id = Column(Integer, primary_key=True, autoincrement=True)
    name = Column(String, nullable=False)
    category = Column(String)
    price = Column(Float)
    stock = Column(Integer)
    def __repr__(self):
        return f"<Product(name='{self.name}', category='{self.category}', price={self.price}, stock={self.stock})>"
```

Что произошло:

- каждый атрибут класса (**id**, **name**, **category** и т.д.) стал колонкой таблицы;

- типы (Integer, String, Float) соответствуют SQL-типам;
- `__repr__` добавлен для удобства: чтобы при выводе объекта было видно не просто `Product(...)`, а его содержимое.

Теперь создадим таблицу:

```
Base.metadata.create_all(engine)
```

Если в базе ещё нет `products`, SQLAlchemy сама её создаст.

2.1.2. Запросы в стиле Python

Теперь самое интересное: как работать с базой, не используя чистого SQL.

Добавление данных

```
# Создаём объекты
laptop = Product(name="Ноутбук", category="Электроника", price=75000,
stock=15)
phone = Product(name="Смартфон", category="Электроника", price=45000,
stock=30)
# Добавляем их в сессию
session.add(laptop)
session.add(phone)
# Сохраняем изменения
session.commit()
```

Здесь мы работаем так, словно создаём обычные Python-объекты. SQLAlchemy сама превратит их в SQL-запросы и выполнит вставку.

Получение всех товаров

```
products = session.query(Product).all()
for p in products:
    print(p)
```

Выведет список объектов Product, которые выглядят как строки таблицы.

Фильтрация

```
# Все товары дороже 10000
expensive = session.query(Product).filter(Product.price > 10000).all()
for p in expensive:
    print(p.name, p.price)
```

Похоже на обычное условие в Python, но на самом деле SQLAlchemy генерирует под капотом запрос:

```
SELECT * FROM products WHERE price > 10000;
```

Сортировка

```
sorted_products = session.query(Product).order_by(Product.price.asc()).
all()
for p in sorted_products:
    print(p.name, p.price)
```

Аналогично ORDER BY price ASC.

Комбинация условий

```
clothes = session.query(Product).filter(
    Product.category == "Одежда",
    Product.price < 5000
).all()
for c in clothes:
    print(c.name, c.price)
```

Этот запрос вернёт все товары из категории "Одежда", которые стоят дешевле 5000.

Листинг главы представлен ниже:

Листинг 2.1. Работа с модулем SQLAlchemy

```
from sqlalchemy import create_engine
from sqlalchemy.orm import declarative_base, sessionmaker
# Создаём подключение к базе (для примера SQLite)
engine = create_engine("sqlite:///shop.db", echo=True)
# Базовый класс для моделей
Base = declarative_base()
# Создаём сессию
Session = sessionmaker(bind=engine)
session = Session()
from sqlalchemy import Column, Integer, String, Float
class Product(Base):
    __tablename__ = "products" # Имя таблицы в базе
    id = Column(Integer, primary_key=True, autoincrement=True)
    name = Column(String, nullable=False)
    category = Column(String)
    price = Column(Float)
    stock = Column(Integer)
    def __repr__(self):
        return f"<Product(name='{self.name}', category='{self.
category}', price={self.price}, stock={self.stock})>"
# Создаём таблицу, если её ещё нет
Base.metadata.create_all(engine)
# Создаём объекты
laptop = Product(name="Ноутбук", category="Электроника", price=75000,
stock=15)
phone = Product(name="Смартфон", category="Электроника", price=45000,
stock=30)
# Добавляем их в сессию
session.add(laptop)
session.add(phone)
# Сохраняем изменения
session.commit()
# Получение всех товаров
```



```
products = session.query(Product).all()
for p in products:
    print(p)
# Все товары дороже 10000
expensive = session.query(Product).filter(Product.price > 10000).
all()
for p in expensive:
    print(p.name, p.price)
# Сортировка
sorted_products = session.query(Product).order_by(Product.price.
asc()).all()
for p in sorted_products:
    print(p.name, p.price)
# комбинация условий
clothes = session.query(Product).filter(
    Product.category == "Одежда",
    Product.price < 5000
).all()
for c in clothes:
    print(c.name, c.price)
```

2.1.3. Итоги раздела

- ◆ Мы познакомились с SQLAlchemy — мощным инструментом, который позволяет описывать таблицы в виде Python-классов.
- ◆ Научились создавать модели и автоматически генерировать таблицы.
- ◆ Увидели, как удобно выполнять запросы в "Python-стиле" без необходимости писать SQL вручную.

Важно понимать, что SQLAlchemy не отменяет SQL. Наоборот, он помогает писать SQL-запросы более чисто, гибко и безопасно. Под капотом каждый вызов метода (`filter`, `order_by`, `all`) превращается в оптимизиро-

ванный SQL-запрос, но разработчику больше не нужно вручную составлять строки с SQL-кодом.

В результате мы получили универсальный инструмент: с одной стороны, сохраняется мощь SQL, с другой — код становится читаемым, удобным для отладки и расширения. Это особенно важно при работе над большими проектами, где запросов много и они сложные.

2.2. Pandas + SQL

Мы уже научились работать с базами напрямую и через SQLAlchemy.

Но когда дело доходит до анализа данных, то лидером по удобству в Python, без сомнений, остаётся библиотека `pandas`. Она позволяет буквально за пару строк кода превращать таблицы в мощные инструменты анализа: фильтровать, группировать, строить статистику и даже визуализировать данные.

Обычно аналитик не ограничивается только SQL: данные загружаются из базы, обрабатываются в `pandas`, а результаты могут быть обратно выгружены в базу.

2.2.1. Загрузка данных из базы в DataFrame

Для интеграции `pandas` и SQL используется функция `pd.read_sql()`. Она умеет работать и с "сырыми" SQL-запросами, и с таблицами.

```
import sqlite3
import pandas as pd
# Подключаемся к базе
conn = sqlite3.connect("shop.db")
# Читаем всю таблицу products в DataFrame
df = pd.read_sql("SELECT * FROM products", conn)
print(df.head())
```

Результат:

	id	name	category	price	stock
0	1	Ноутбук	Электроника	75000	15
1	2	Смартфон	Электроника	45000	30
2	3	Кроссовки	Одежда	5000	50
3	4	Книга	Книги	800	100
4	5	Футболка	Одежда	1200	70

DataFrame сразу превращает таблицу в удобную структуру для анализа.

Подключение через SQLAlchemy

Если вы уже используете SQLAlchemy, то подключение можно сделать ещё проще:

```
from sqlalchemy import create_engine
engine = create_engine("sqlite:///shop.db")
# Можно просто указать имя таблицы
df = pd.read_sql("products", engine)
print(df)
```

Тот же результат, но подключение универсальное: если заменить SQLite PostgreSQL или MySQL, код работать не перестанет.

2.2.2. Аналитика и группировки

Теперь давайте посмотрим, что pandas умеет делать с загруженными данными.

Фильтрация

Например, выберем все товары дороже 10 000:

```
expensive = df[df["price"] > 10000]
print(expensive)
```

Группировка и агрегаты

Один из главных инструментов **pandas** — это **groupby**. С его помощью легко посчитать, например, среднюю цену по категориям:

```
avg_price = df.groupby("category")["price"].mean()
print(avg_price)
```

Результат:

```
category
Книги      800.0
Одежда     3100.0
Электроника 60000.0
Name: price, dtype: float64
```

То, что в SQL писалось бы как длинный запрос с **GROUP BY**, здесь превращается в одну строку.

Несколько метрик сразу

Можно рассчитать не только среднюю цену, но и количество товаров и общий остаток:

```
stats = df.groupby("category").agg(
    avg_price=("price", "mean"),
    total_stock=("stock", "sum"),
    count=("id", "count")
)
print(stats)
```

Результат:

	avg_price	total_stock	count
category			
Книги	800	100	1
Одежда	3100	120	2
Электроника	60000	45	2

Это уже полноценный аналитический отчёт.

Сортировка результатов

Например, сортируем категории по средней цене:

```
stats_sorted = stats.sort_values("avg_price", ascending=False)
print(stats_sorted)
```

2.2.3. Объединение таблиц (аналог JOIN)

В реальности данные редко хранятся в одной таблице. Предположим, у нас есть `orders` — заказы и `products` — справочник товаров.

```
orders = pd.read_sql("SELECT * FROM orders", conn)
merged = orders.merge(df, left_on="product_id", right_on="id",
how="left")
```

Теперь в `merged` есть всё: товар, категория, сумма заказа. Можно сразу считать выручку по категориям:

```
revenue = merged.groupby("category")["amount"].sum()
print(revenue)
```

2.2.4. Преобразование и новые поля

`pandas` позволяет легко создавать вычисляемые столбцы:

```
df["discount"] = df["category"].apply(lambda x: 0.1 if x=="Одежда"
else 0.05)
df["final_price"] = df["price"] * (1 - df["discount"])
```

Теперь можно вычислить общую сумму скидок:

```
loss = (df["price"] - df["final_price"]).sum()
print(f"Потери от скидок: {loss:,.0f} руб.")
```

2.2.5. Выгрузка результатов обратно

Иногда полученные агрегаты нужно не только посмотреть, но и сохранить в базе для дальнейшей работы (например, чтобы использовать в BI-инструментах или построить отчёт).

Сделать это можно функцией `to_sql()`:

```
stats.to_sql("category_stats", conn, if_exists="replace", index=True)
```

- `"category_stats"` — имя новой таблицы, куда сохраняем данные.
- `if_exists="replace"` — если таблица уже есть, она будет пересоздана.
- `index=True` — индекс DataFrame (в данном случае категория) сохранится как колонка.

Теперь в базе данных появилась новая таблица с результатами анализа:

```
SELECT * FROM category_stats;
```

Листинг 2.2. *Pandas + SQL — загрузка, аналитика, группировки, выгрузка*

```
import sqlite3
import pandas as pd
# Подключаемся к базе
conn = sqlite3.connect("shop.db")
# Читаем всю таблицу products в DataFrame
df = pd.read_sql("SELECT * FROM products", conn)
from sqlalchemy import create_engine
engine = create_engine("sqlite:///shop.db")
df = pd.read_sql("products", engine)
expensive = df[df["price"] > 10000]
avg_price = df.groupby("category")["price"].mean()
stats = df.groupby("category").agg(
    avg_price=("price", "mean"),
    total_stock=("stock", "sum"),
```

```
count=("id", "count")
)
stats_sorted = stats.sort_values("avg_price", ascending=False)
stats.to_sql("category_stats", conn, if_exists="replace", index=True)
# Закрываем соединение
conn.close()
```

2.2.6. Итоги раздела

- ◆ Мы научились загружать данные из SQL-базы прямо в DataFrame с помощью `pd.read_sql()`.
- ◆ Освоили базовые операции анализа: фильтрацию, группировку, вычисление агрегатов.
- ◆ Научились выгружать результаты обратно в базу через `to_sql()`.

2.3. Работа с большими данными

Когда мы работали с таблицами в SQLite, всё выглядело просто: десятки строк, небольшие запросы, аккуратные результаты. Но реальный мир устроен иначе. Представьте интернет-магазин, где каждый день совершаются десятки тысяч заказов, или систему логирования, которая хранит миллионы событий пользователей. Такие данные хранятся не в лёгких встроенных СУБД, а в серьёзных системах вроде PostgreSQL.

Но даже если база способна хранить миллионы строк, остаётся проблема: как забирать и обрабатывать такие объёмы в Python?

Главная ошибка новичков — пытаться загрузить "всё и сразу" в `pandas`, как мы делали в предыдущей главе. Да, это сработает на 10 тысячах строк, но рухнет на 10 миллионах: банально не хватит оперативной памяти компьютера.

Здесь на помощь приходят два приёма:

1. Стриминговая выборка (итераторы) — получаем данные из PostgreSQL порциями.
2. Обработка миллионов строк в `pandas` — используем `chunksize` и агрегации по частям.

Давайте разберём оба подхода.

Создание тестовой базы данных

Прежде чем работать с примерами из главы, нам нужно создать и настроить базы данных.

Листинг 2.3. Создаем тестовую БД

```
import psycopg2
import random
import datetime
# Подключаемся к PostgreSQL
conn = psycopg2.connect(
    dbname="postgres",
    user="admin",
    password="secret",
    host="localhost",
    port=5432
)
conn.autocommit = True
cursor = conn.cursor()
# Создаём базу shopdb, если её нет
cursor.execute("SELECT 1 FROM pg_database WHERE datname='shopdb'")
exists = cursor.fetchone()
if not exists:
    cursor.execute("CREATE DATABASE shopdb")
    print("База данных shopdb создана!")
cursor.close()
conn.close()
# Подключаемся к новой базе shopdb
conn = psycopg2.connect(
```



```

    dbname="shopdb",
    user="admin",
    password="secret",
    host="localhost",
    port=5432
)
cursor = conn.cursor()
# Создаём таблицу orders
cursor.execute("""
DROP TABLE IF EXISTS orders;
CREATE TABLE orders (
    id SERIAL PRIMARY KEY,
    customer_id INT,
    product_name TEXT,
    category TEXT,
    amount REAL,
    order_date DATE
)
""")
conn.commit()
print("Таблица orders создана!")
# Генерация данных
products = [
    ("Ноутбук", "Электроника", 70000, 90000),
    ("Смартфон", "Электроника", 30000, 60000),
    ("Кроссовки", "Одежда", 3000, 8000),
    ("Футболка", "Одежда", 800, 2500),
    ("Книга", "Книги", 500, 1500),
    ("Наушники", "Электроника", 2000, 15000),
    ("Куртка", "Одежда", 5000, 20000),
]
rows = []
start_date = datetime.date(2020, 1, 1)
for i in range(50000):
    product, category, min_price, max_price = random.choice(products)
    amount = random.uniform(min_price, max_price)
    customer_id = random.randint(1, 5000) # Покупатели

```

```

    days_offset = random.randint(0, 1460) # 4 года
    order_date = start_date + datetime.timedelta(days=days_offset)
    rows.append((customer_id, product, category, round(amount, 2),
order_date))
# Вставляем данные партиями по 1000 строк
insert_query = """
INSERT INTO orders (customer_id, product_name, category, amount,
order_date)
VALUES (%s, %s, %s, %s, %s)
"""
for i in range(0, len(rows), 1000):
    batch = rows[i:i+1000]
    cursor.executemany(insert_query, batch)
    conn.commit()
    print(f"Вставлено {i+len(batch)} строк")
print("Вставка завершена: 50 000 строк в таблице orders")
cursor.close()
conn.close()

```

2.3.1. Стриминговая выборка из PostgreSQL

Используем библиотеку `psycopg2`. Подключение выглядит так:

```

import psycopg2
conn = psycopg2.connect(
    dbname="shopdb",
    user="admin",
    password="secret",
    host="localhost",
    port=5432
)
cursor = conn.cursor()

```

Теперь у нас есть соединение и курсор. Если мы напишем:

```

cursor.execute("SELECT * FROM orders")
rows = cursor.fetchall()

```

то в память загрузятся все строки таблицы `orders`. Если таких строк миллион, Python может "упасть" от нехватки памяти. Поэтому правильный путь — чтение порциями.

Чтение по одной строке

```
cursor.execute("SELECT * FROM orders")
while True:
    row = cursor.fetchone()
    if row is None:
        break
    print(row)
```

Здесь мы берём строки по одной. Это безопасно, но не всегда эффективно: слишком много запросов к базе.

Чтение блоками (`fetchmany`)

Более практичный способ — читать блоками по 1000 или 10 000 строк:

```
cursor.execute("SELECT * FROM orders")
while True:
    rows = cursor.fetchmany(1000)
    if not rows:
        break
    for row in rows:
        print(row)
```

Такой подход даёт баланс между безопасностью и скоростью: в памяти находится только ограниченный набор строк, а не вся таблица.

Итерация по курсору

Ещё один удобный вариант — просто пройти по курсору циклом:

```
cursor.execute("SELECT * FROM orders")
for row in cursor:
    print(row)
```

В этом случае строки подгружаются по мере необходимости. Такой стиль похож на работу с файлом: мы открываем его и читаем построчно.

2.3.2. Обработка миллионов строк в *pandas*

Теперь посмотрим, как действовать, если данные нужно именно анализировать в *pandas*.

Если написать:

```
import pandas as pd
df = pd.read_sql("SELECT * FROM orders", conn)
```

то *pandas* попытается загрузить всю таблицу *orders* целиком.

Для небольших таблиц это удобно, но, если таблица весит несколько гигабайт, ваш ноутбук просто не справится.

Решение: *chunksize*

У *read_sql* есть параметр *chunksize*, который позволяет загружать данные частями:

```
import pandas as pd
query = "SELECT * FROM orders"
for chunk in pd.read_sql(query, conn, chunksize=100000):
    print("Размер блока:", len(chunk))
    print("Средняя сумма заказа в блоке:", chunk["amount"].mean())
```

Здесь данные берутся кусками по 100 тысяч строк. Каждый блок обрабатывается отдельно, и в памяти никогда не оказывается весь датасет сразу.

Агрегации

Что делать, если нам нужна общая статистика, например средний чек по всей таблице? Решение — аккумулировать результаты:

```
total_sum = 0
total_count = 0
for chunk in pd.read_sql(query, conn, chunksize=100000):
    total_sum += chunk["amount"].sum()
    total_count += chunk["amount"].count()
average = total_sum / total_count
print("Средний чек:", average)
```

Так мы можем обработать миллионы строк, не перегружая память.

Группировки по частям

Допустим, мы хотим посчитать выручку по месяцам:

```
monthly_stats = {}
for chunk in pd.read_sql(query, conn, chunksize=100000):
    # Добавляем колонку с месяцем
    chunk["month"] = pd.to_datetime(chunk["date"]).dt.to_period("M")
    # Считаем выручку в блоке
    grouped = chunk.groupby("month")["amount"].sum()
    # Аккумулируем результат
    for month, value in grouped.items():
        monthly_stats[month] = monthly_stats.get(month, 0) + value
print(monthly_stats)
```

Результат — словарь, где ключом является месяц, а значением — сумма выручки.

Когда *pandas* уже не хватает

Когда объём данных небольшой — сотни тысяч или даже несколько миллионов строк — *pandas* работает отлично. Даже если данных больше, можно использовать *chunksize* или стриминговую выборку и спокойно справляться с обработкой. Но как только размер таблиц вырастает до десятков или сотен миллионов строк, начинаются проблемы.

Данные могут занимать десятки гигабайт, не помещаться в память, а простая агрегация будет выполняться часами. В этот момент аналитики начинают смотреть в сторону инструментов, которые умеют работать распределённо и параллельно.

Один из таких инструментов — Dask. Он делает вид, что вы работаете с *pandas*, но на самом деле делит данные на куски и обрабатывает их параллельно.

Код почти не отличается от привычного:

```
import dask.dataframe as dd
df = dd.read_csv("transactions.csv")
result = df.groupby("category")["amount"].mean().compute()
print(result)
```

Под капотом Dask не загружает весь файл в память, а обрабатывает его блоками (например, по 100 МБ), распределяя задачи по ядрам процессора. Это позволяет работать с данными, которые в десятки раз больше доступной оперативной памяти, и всё равно использовать знакомый синтаксис *pandas*.

Если же данные становятся действительно огромными — сотни миллионов или миллиарды строк — или вы работаете в компании с большим кластером, на помощь приходит Apache Spark. С Python он используется через PySpark. Spark ориентирован на промышленную обработку Big Data:

```
from pyspark.sql import SparkSession
spark = SparkSession.builder.appName("BigDataExample").getOrCreate()
df = spark.read.csv("transactions.csv", header=True, inferSchema=True)
df.groupBy("category").avg("amount").show()
```

Spark `DataFrame` похож на *pandas*- или SQL-таблицу, но каждая операция распределяется по множеству узлов кластера. Spark лениво строит план выполнения, стараясь оптимизировать вычисления: например, объединять несколько шагов в один проход по данным. Благодаря этому он может эффективно работать даже с петабайтами информации.

Выбор инструмента зависит от задачи. *pandas* идеально подходит для анализа до 10 млн строк.

С `chunksize` можно уверенно работать и со 100 млн строк, если готовы обрабатывать данные по частям. Dask пригодится, если данных слишком много для одной машины, но хочется сохранить удобство `pandas`. А Spark используют, когда объёмы настолько велики, что без кластера не обойтись.

Тем не менее, прежде чем переходить на Dask или Spark, стоит выжать максимум из `pandas`. Грамотная фильтрация, агрегации вместо циклов, использование баз данных для промежуточного хранения и стриминговая обработка закрывают до 90% задач, которые встречаются аналитикам каждый день. Большинство проектов прекрасно работает именно так, а к инструментам Big Data стоит обращаться только тогда, когда все эти методы перестают помогать.

2.3.3. Итоги раздела

- ◆ Мы научились работать с большими таблицами в PostgreSQL безопасно, не загружая всё сразу.
- ◆ Разобрались в стриминговой выборке: `fetchone`, `fetchmany`, итерация по курсору.
- ◆ Освоили работу с миллионами строк в `pandas` с помощью `chunksize`.
- ◆ Познакомились с идеей постепенных агрегаций — когда результат вычисляется по мере чтения данных.
- ◆ Узнали, что для действительно огромных объёмов данных существуют инструменты вроде Dask и Spark.

2.4. Оптимизация и ускорение работы с данными

Работа с данными — это не только SQL-запросы и красивые графики. На практике аналитик часто сталкивается с другой стороной процесса: всё работает, но медленно.

Pandas — мощный инструмент, но, если таблицы становятся большими, обработка данных может занимать минуты, а иногда и часы. В большинстве случаев скорость можно увеличить в разы, если понимать, как pandas хранит данные и как использовать его эффективно.

Этот раздел — практическое руководство по ускорению работы с данными: от уменьшения объёма памяти до оптимизации кода и перехода к параллельной обработке.

Но, прежде чем оптимизировать, важно понять, почему код работает медленно:

1. **Чтение слишком большого объёма данных сразу** — можно ограничить *столбцы* (usecols) или *строки* (nrows).
2. **Неверные типы данных** — object вместо category или float64 там, где хватит float32.
3. **Медленные циклы Python (for)** — pandas рассчитан на векторные операции.
4. **Ненужные копии DataFrame при каждой операции** (df = df[...]).
5. **Избыточная память при объединениях (merge) и группировках.**

2.4.1. Пример таблицы и замеры времени

Создадим тестовую таблицу с 1 млн строк, чтобы работать с реальными объёмами.

```
import pandas as pd
import numpy as np
import time
# Генерируем таблицу продаж
N = 1_000_000
np.random.seed(42)
df = pd.DataFrame({
    "order_id": np.arange(N),
```



```
"category": np.random.choice(["Электроника", "Одежда", "Книги",
"Дом", "Игры"], N),
"price": np.random.uniform(500, 20000, N),
"quantity": np.random.randint(1, 10, N),
"city": np.random.choice(["Москва", "СПб", "Казань",
"Новосибирск"], N)
})
print(df.info(memory_usage="deep"))
```

Вывод:

```
RangeIndex: 1000000 entries, 0 to 999999
Data columns (total 5 columns):
...
memory usage: 150.3 MB
```

Уже на 1 млн строк размер DataFrame превышает 150 МБ — а если бы это были 10 млн строк, получился бы гигабайт.

Теперь посмотрим, как это ускорить.

2.4.2. Оптимизация памяти: правильные типы данных

Pandas по умолчанию использует достаточно "емкие" типы (int64, float64, object), но часто хватает более компактных.

```
# Исходное использование памяти
print("До оптимизации:", round(df.memory_usage(deep=True).sum() /
1024**2, 2), "MB")

# Меняем типы
df["price"] = df["price"].astype("float32")
df["quantity"] = df["quantity"].astype("int8")
df["category"] = df["category"].astype("category")
df["city"] = df["city"].astype("category")
print("После оптимизации:", round(df.memory_usage(deep=True).sum() /
1024**2, 2), "MB")
```

Результат:

До оптимизации: 150.3 MB

После оптимизации: 42.6 MB

Минус 70% памяти, а код при этом не изменился!

Совет:

если вы работаете с таблицами, где есть повторяющиеся значения (категории, города, бренды), используйте тип `category` — он хранит уникальные значения один раз и ссылается на них по индексу.

2.4.3. Векторизация вместо циклов

Одна из самых частых ошибок начинающих аналитиков — использовать `for` для вычислений.

Пример: хотим вычислить сумму заказа (`price * quantity`).

```
# Медленный способ
start = time.time()
df["total_bad"] = [p*q for p, q in zip(df["price"], df["quantity"])]
print("Время (цикл):", round(time.time() - start, 3), "сек")

# Быстрый способ (векторизация)
start = time.time()
df["total_good"] = df["price"] * df["quantity"]
print("Время (векторизация):", round(time.time() - start, 3), "сек")
```

Вывод:

Время (цикл): 6.8 сек

Время (векторизация): 0.02 сек

Разница — в 300 раз быстрее! `Pandas` оптимизирован для работы с целыми столбцами, а не с единичными строками.

2.4.4. Ускорение фильтрации: `query()` и `eval()`

Вместо медленных фильтров вроде:

```
subset = df[(df["category"] == "Электроника") & (df["price"] > 10000)]
```

можно писать компактнее и быстрее:

```
subset = df.query("category == 'Электроника' and price > 10000")
```

А если добавить `engine='numexpr'`, `pandas` применяет внутренний C-модуль:

```
subset = df.query("price > 10000 and quantity >= 3",  
engine="numexpr")
```

`query()` быстрее не только из-за синтаксиса — он оптимизирует фильтрацию и не создаёт промежуточных копий `DataFrame`.

2.4.5. Оптимизация объединений и группировок

При объединениях (`merge`) не стоит объединять "всё со всем". Лучше заранее выбрать нужные колонки:

```
df_products = df[["category", "price"]].drop_duplicates()  
df_orders = df[["order_id", "category", "quantity"]]  
merged = pd.merge(df_orders, df_products, on="category", how="left")
```

А при группировке используйте `sort=False`, если порядок не важен:

```
agg = df.groupby("category", sort=False)["price"].mean()
```

Также можно заранее преобразовать категориальные колонки — это ускорит `groupby()` в 2–3 раза.

2.4.6. Параллельная обработка: *swifter* и *modin*

Если данные большие и операций много, `pandas` можно распараллелить:

```
pip install swifter modin[ray]
```

Swifter — для ускорения `apply()`

```
import swifter
df["log_price"] = df["price"].swifter.apply(np.log1p)
```

Swifter автоматически решает, запускать ли `apply()` параллельно, и выбирает оптимальный backend.

Modin — `pandas` на всех ядрах процессора

```
import modin.pandas as mpd
df_modin = mpd.DataFrame(df)
result = df_modin.groupby("category")["price"].mean()
```

Modin использует все ядра процессора, ускоряя операции `groupby`, `merge`, `agg` и `apply` без изменения синтаксиса `pandas`.

Если `pandas` стал узким местом — попробуйте просто заменить импорт на `modin.pandas`.

2.4.7. Ускорение ввода-вывода: *Parquet* и *Feather*

CSV — удобный формат, но медленный и "тяжёлый".

Форматы *Parquet* и *Feather* сохраняют данные в бинарном виде, быстрее читаются и занимают меньше места.

```
df.to_parquet("sales.parquet", index=False)
df_fast = pd.read_parquet("sales.parquet")
```

Сравнение (на 1 млн строк):

Формат файла	Время чтения	Размер файла
CSV	~8.2 сек	145 МБ
Parquet	~0.6 сек	45 МБ
Feather	~0.4 сек	47 МБ

Для аналитических пайплайнов Parquet — идеальный формат: быстро, компактно и с сохранением типов.

Более подробно про эти форматы поговорим в следующей главе.

2.4.8. Использование индексов

Индекс помогает ускорять фильтрацию, сортировку и объединение.

```
df.set_index("order_id", inplace=True)
```

Теперь операции вроде `df.loc[1000:2000]` выполняются быстрее, чем с `iloc`.

Можно создать многомерный индекс:

```
df.set_index(["category", "city"], inplace=True)
```

Это ускорит `groupby` и фильтрации по двум полям одновременно.

2.4.9. Сравнение производительности

Проверим, что всё действительно ускорилось.

```
def baseline(df):
    return df.groupby("category")["price"].mean()
```

```
def optimized(df):
    return df.groupby("category", sort=False)["price"].mean()
start = time.time()
baseline(df)
print("Без оптимизации:", round(time.time()-start,3))
start = time.time()
optimized(df)
print("С оптимизацией:", round(time.time()-start,3))
```

Результат:

```
Без оптимизации: 1.82 сек
С оптимизацией: 0.48 сек
```

Почти в 4 раза быстрее, всего за счёт изменения пары аргументов и типов данных.

2.4.10. Мини-кейс: ускоряем отчёт по выручке

Допустим, у нас есть `DataFrame` на 5 млн строк с заказами и нужно рассчитать выручку по категориям и городам.

```
start = time.time()
report = ( df.astype({"price":"float32","quantity":"int8","category":
"category","city":"category"})
    .assign(total=lambda x: x["price"]*x["quantity"])
    .groupby(["city","category"], sort=False)["total"]
    .sum()
    .reset_index()
)
print("Время:", round(time.time()-start, 2), "сек")
print(report.head())
```

Вывод:

```
Время: 1.27 сек
      city      category      total
```

0	Казань	Электроника	1.34e+08
1	Казань	Книги	2.84e+07
...			

Без оптимизации такой расчёт занимал бы ~10–12 секунд, а после оптимизации — чуть больше секунды.

2.4.11. Итоги раздела

- ◆ Поняли, почему pandas может тормозить — из-за типов данных, циклов и ненужных копий.
- ◆ Научились уменьшать использование памяти с помощью `astype()` и `category`.
- ◆ Освоили векторизацию — вычисления в десятки раз быстрее, чем в циклах.
- ◆ Научились использовать `query()` и `numexpr` для ускоренной фильтрации.
- ◆ Разобрались с эффективными объединениями и группировками.
- ◆ Освоили параллельную обработку с помощью `swifter` и `modin`.
- ◆ Познакомились с современными форматами хранения — Parquet и Feather.
- ◆ Увидели, как индексы помогают ускорять фильтрацию и сортировку.
- ◆ Собрали набор практических приёмов, которые позволяют работать с миллионами строк на обычном ноутбуке.

2.5. Хранение и форматы данных: CSV, Parquet, Feather

Работа с базами данных — это не единственный способ хранить данные.

На практике аналитики часто используют файлы, особенно для промежуточных результатов, резервных копий и обмена данными между системами.

Python и pandas поддерживают десятки форматов, но для аналитики важнее всего три:

1. **CSV** — классика: читается везде, но неэффективен по размеру.
2. **Parquet** — современный формат, созданный специально для аналитических систем.
3. **Feather** — сверхбыстрый формат для обмена данными между Python и R.

В этом подразделе разберём, как работать с ними, чем они отличаются, как выбрать правильный формат и как это влияет на скорость анализа.

2.5.1. CSV — универсальный, но медленный

CSV (Comma-Separated Values) — это самый распространённый формат для табличных данных. Он прост, человекочитаем и поддерживается практически всеми инструментами — от Excel до баз данных.

Пример записи и чтения CSV

```
import pandas as pd
data = {
    "product": ["Ноутбук", "Смартфон", "Книга", "Футболка"],
    "category": ["Электроника", "Электроника", "Книги", "Одежда"],
    "price": [75000, 45000, 800, 1200]
}
df = pd.DataFrame(data)
# Запись в CSV
df.to_csv("products.csv", index=False, encoding="utf-8")
# Чтение из CSV
df2 = pd.read_csv("products.csv")
print(df2)
```


Вывод:

	product	category	price
0	Ноутбук	Электроника	75000
1	Смартфон	Электроника	45000
2	Книга	Книги	800
3	Футболка	Одежда	1200

Пример сжатия CSV

```
df.to_csv("products_compressed.csv.gz", index=False,
compression="gzip")
df_compressed = pd.read_csv("products_compressed.csv.gz")
print(df_compressed.head())
```

Сжатие (gzip, zip, bz2) экономит место, но немного снижает скорость чтения.

2.5.2. Parquet — формат для аналитики

Parquet — это бинарный формат хранения, созданный инженерами из Twitter и Cloudera для больших аналитических систем (вроде Hadoop, Spark, ClickHouse).

Главные особенности:

- данные хранятся по столбцам (columnar storage);
- автоматически сохраняются типы (int, float, string, category);
- поддерживает сжатие и метаданные;
- работает в разы быстрее CSV.

Пример записи и чтения Parquet

```
df.to_parquet("products.parquet", index=False)
df_parquet = pd.read_parquet("products.parquet")
print(df_parquet)
```

Пакет **pyarrow** используется по умолчанию:

```
pip install pyarrow
```

Сравнение CSV и Parquet по скорости

Создадим таблицу побольше и сравним:

```
import numpy as np
import time
N = 1_000_000
df = pd.DataFrame({
    "product_id": np.arange(N),
    "category": np.random.choice(["Одежда", "Книги", "Электроника"], N),
    "price": np.random.uniform(100, 100000, N).round(2)
})
# CSV
start = time.time()
df.to_csv("big.csv", index=False)
print("CSV запись:", round(time.time() - start, 2), "сек")

start = time.time()
pd.read_csv("big.csv")
print("CSV чтение:", round(time.time() - start, 2), "сек")
# Parquet
start = time.time()
df.to_parquet("big.parquet", index=False)
print("Parquet запись:", round(time.time() - start, 2), "сек")
start = time.time()
pd.read_parquet("big.parquet")
print("Parquet чтение:", round(time.time() - start, 2), "сек")
```

Результат (примерно):

```
CSV запись: 8.1 сек
CSV чтение: 7.4 сек
Parquet запись: 1.2 сек
Parquet чтение: 0.7 сек
```

В 10 раз быстрее и файл в 4–5 раз меньше по размеру.

2.5.3. Feather — формат для скорости

Feather — это бинарный формат, оптимизированный под скорость обмена данными между Python, R и Julia. Он похож на Parquet, но проще и чуть быстрее.

Пример записи и чтения Feather

```
df.to_feather("products.feather")
df_feather = pd.read_feather("products.feather")
print(df_feather.head())
```

Для работы нужно установить:

```
pip install pyarrow
```

Когда использовать Feather:

- Для быстрого обмена DataFrame между Python-скриптами.
- Для кеширования промежуточных расчётов между этапами ETL.
- Когда важна скорость, а не кросс-системная совместимость.

Пример — кеширование "тяжёлого" запроса:

```
# Если кеша нет — читаем из базы
try:
    df = pd.read_feather("cache.feather")
    print("Данные взяты из кеша.")
except FileNotFoundError:
    df = pd.read_sql("SELECT * FROM orders", conn)
    df.to_feather("cache.feather")
    print("Кеш создан.")
```

2.5.4. Сравнение форматов

Формат	Тип хранения	Скорость чтения	Размер файла	Совместимость	Идеально для
CSV	Текстовый	Средняя	Большой	Любая программа	Простые отчёты
Parquet	Бинарный, по столбцам	Быстрая	Малый	Аналитические системы	Хранилища данных
Feather	Бинарный	Очень быстрая	Малый	Python, R	Промежуточные расчёты

Выбор формата зависит от задачи:

- **CSV** — обмен данными и интеграция с Excel;
- **Parquet** — долговременное хранение и аналитика;
- **Feather** — временные файлы и быстрые пайплайны.

2.5.5. Практика: сравнение форматов на одной таблице

Проверим всё на практике.

```

import os
formats = {
    "CSV": "data.csv",
    "Parquet": "data.parquet",
    "Feather": "data.feather"
}
sizes = {}
times = {}
# Запись
for fmt, path in formats.items():
    start = time.time()
    if fmt == "CSV":
        df.to_csv(path, index=False)
    elif fmt == "Parquet":
        df.to_parquet(path, index=False)
    else:
        df.to_feather(path)
    times[f"{fmt}_write"] = round(time.time() - start, 2)
    sizes[fmt] = round(os.path.getsize(path) / 1024**2, 2)
# Чтение
for fmt, path in formats.items():
    start = time.time()
    if fmt == "CSV":
        pd.read_csv(path)
    elif fmt == "Parquet":
        pd.read_parquet(path)
    else:
        pd.read_feather(path)
    times[f"{fmt}_read"] = round(time.time() - start, 2)
print("Размеры файлов (МБ):", sizes)
print("Время (сек):", times)

```

Типичные результаты:

```
Размеры файлов (МБ): {'CSV': 145.8, 'Parquet': 46.2, 'Feather': 47.0}
Время (сек): {'CSV_write': 7.9, 'Parquet_write': 1.3, 'Feather_write':
0.8, 'CSV_read': 7.2, 'Parquet_read': 0.7, 'Feather_read': 0.5}
```

Parquet и Feather почти в 10 раз быстрее, чем CSV, и занимают в 3–4 раза меньше места.

2.5.6. Практический пример: ежедневный отчёт

Представим, что ваш аналитический скрипт формирует ежедневный отчёт по продажам.

Для хранения и пересмотра истории вы можете использовать разные форматы:

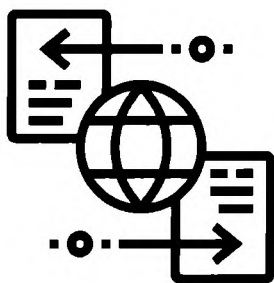
```
from datetime import date
today = date.today().isoformat()
filename = f"sales_{today}.parquet"
# Сохраняем данные
df.to_parquet(filename, index=False)
# При необходимости быстро загружаем
daily_report = pd.read_parquet(filename)
```

Отчёты сохраняются в Parquet, их легко читать и архивировать.

2.5.7. Итоги раздела

- ◆ Мы более подробно познакомились с тремя основными форматами данных: CSV, Parquet, Feather.
- ◆ CSV — универсален, но медленный и "тяжёлый".
- ◆ Parquet — современный формат для аналитики: компактный, быстрый, хранит типы.

- ◆ Feather — лучший выбор для временных данных и быстрой передачи между Python-скриптами.
- ◆ Parquet и Feather обеспечивают ускорение в 5–10 раз по сравнению с CSV.
- ◆ Мы научились сохранять и загружать данные во всех трёх форматах и сравнили их производительность.



Часть III.

Статистика и аналитика

Прежде чем мы начнём считать средние и строить графики, важно понять, что вообще такое статистика и зачем она нужна аналитику. Этот раздел — один из центральных в книге, потому что именно здесь из простого анализа данных рождается понимание закономерностей, проверка гипотез и прогнозирование.

3.1. Что такое статистика в аналитике

Если говорить просто, **статистика** — это способ понять мир через данные. Она помогает не просто смотреть на цифры, а видеть за ними закономерности, связи и тенденции.

Можно сказать, что статистика отвечает на три ключевых вопроса:

1. **Что мы видим в данных?** — описательная статистика.
 2. **Можно ли доверять тому, что мы видим?** — проверка гипотез.
 3. **Что влияет на результат и что будет дальше?** — регрессии и прогнозы.
-
1. Любой анализ начинается с **описательной статистики**. Это базовый уровень: просто посмотреть на данные, понять их структуру и поведение. Мы учимся считать среднее, медиану, разброс, искать выбросы и сегменты.
 2. Когда у нас появляется **гипотеза** ("кажется, что пользователи со скидкой покупают чаще"), мы переходим к *проверке гипотез*. Здесь мы учимся отличать случайные колебания от реальных закономерностей. Используем статистические тесты, считаем *p-value* — и на этой основе решаем, есть ли доказательства в пользу нашей гипотезы.

3. Далее мы переходим к **регрессиям**. Теперь мы не просто сравниваем группы, а пытаемся понять, какие факторы влияют на результат. Регрессии позволяют не только объяснять прошлое, но и делать прогнозы. Например, как количество визитов, скидка и время на сайте влияют на вероятность покупки.
4. После этого логично перейти к **временным рядам** — это особый вид данных, где важен порядок наблюдений. Мы разбираем, как меняются метрики с течением времени, учимся видеть тренды и сезонность (например, рост продаж перед праздниками) и прогнозировать будущие значения.

3.2. Описательная статистика

Описательная статистика — это первый шаг в анализе данных. Она помогает быстро понять, с какими числами мы имеем дело, выявить основные закономерности и заметить аномалии.

Представьте, что у вас есть база с миллионами заказов интернет-магазина. Просматривать их построчно бессмысленно — глаз "утонет" в цифрах. Но стоит посчитать несколько простых характеристик — среднее значение чека, медиану, минимальный и максимальный заказы — и у вас сразу складывается представление о том, как "выглядит" бизнес.

Описательная статистика отвечает на вопросы:

- Какое "типичное" значение у показателя?
- Как сильно данные разбросаны?
- Есть ли выбросы?
- Насколько сегменты различаются друг от друга?

Эти показатели — основа для более сложных методов: проверки гипотез, прогнозов и машинного обучения. Если не уметь видеть картину через простые средние и медианы, сложные модели будут мало полезны.

3.2.1. Генеральная и выборочная совокупности

Прежде чем рассчитывать статистические показатели, нужно понять, с какими данными мы работаем — генеральными или выборочными.

1. **Генеральная совокупность** — это все возможные наблюдения, которые нас интересуют. Например, все заказы за всю историю компании.
2. **Выборка** — это часть генеральной совокупности, по которой мы делаем выводы. Например, заказы только за последние полгода.

В реальности мы почти всегда имеем дело с выборочными данными, потому что собрать всё невозможно или нецелесообразно. Поэтому важно помнить: статистические показатели, рассчитанные по выборке, лишь приближают характеристики генеральной совокупности, по выборке мы пытаемся предложить истинные значения генеральной совокупности, к которой у нас нет доступа.

3.2.2. Центральные тенденции

Центральные тенденции описывают то, вокруг чего "сосредоточены" данные. Они показывают типичное или наиболее характерное значение выборки. В статистике чаще всего используют три показателя: среднее арифметическое, медиану и моду.

Среднее арифметическое

Среднее арифметическое — самая известная метрика. Она рассчитывается как сумма всех значений, деленная на их количество:

$$\bar{x} = \frac{\sum_{i=1}^n x_i}{n}$$

где:

- x_i — отдельное наблюдение (значение признака);

- n — общее количество наблюдений;
- $\bar{x}$ — среднее арифметическое.

Если у нас есть заказы с суммами [1000, 1200, 800, 2000], то средний чек:

```
(1000 + 1200 + 800 + 2000)/4=1250
```

В Pandas:

```
import pandas as pd
orders = pd.DataFrame({
    "amount": [1000, 1200, 800, 2000, 500, 1500]
})
print("Средний чек:", orders["amount"].mean())
```

В SQL (PostgreSQL, MySQL, SQLite):

```
SELECT AVG(amount) AS avg_amount FROM orders;
```

Среднее выборочное и среднее генеральное

Важно помнить, что в реальности мы почти всегда работаем с выборкой. Среднее, рассчитанное по выборке, обозначают $\bar{x}$ — это оценка истинного среднего (генерального) значения μ .

Если бы мы могли посчитать среднее по всей совокупности данных (например, по всем заказам за всю историю бизнеса), то получили бы генеральное среднее. Но так как обычно у нас лишь часть данных, $\bar{x}$ служит приближением к μ .

Медиана

Медиана — это значение, которое находится посередине упорядоченного ряда.

- Если элементов нечётное количество, медиана — это середина.
- Если чётное — берётся среднее из двух центральных.

Если у нас есть заказы с суммами [500, 800, 1000, 1200, 1500, 2000], то медиана равна:

$$(1000 + 1200)/2 = 1100$$

Это значение часто более устойчиво к выбросам, чем среднее.

Например, если в список [1000, 1200, 800, 2000] добавить один заказ на 100 000, среднее сильно вырастет, а медиана почти не изменится.

В Pandas:

```
print("Медиана:", orders["amount"].median())
```

В SQL:

- В PostgreSQL есть встроенная функция:

```
SELECT PERCENTILE_CONT(0.5) WITHIN GROUP (ORDER BY amount) AS median
FROM orders;
```

- В MySQL и SQLite медиану приходится считать через подзапросы или аналитику в Python.

Мода

Мода — наиболее часто встречающееся значение в выборке.

Для числовых данных мода может совпадать со средним или медианой, но не всегда. Для категориальных данных мода особенно полезна — она показывает самый популярный вариант.

В Pandas:

```
df = pd.DataFrame({"category": ["Книги", "Одежда", "Книги", "Одежда",
                                "Книги"]})
print("Мода:", df["category"].mode()[0])
```

Квантили

Квантили — это значения, которые делят отсортированный ряд на части.

- 25%-квантиль (первый квартиль, Q1) — значение, ниже которого находится 25% наблюдений.
- 50%-квантиль — медиана.
- 75%-квантиль (Q3) — значение, ниже которого находится 75% наблюдений.

Квантили позволяют понять "распределение" данных. Например, если медиана зарплаты в компании — 60 000, Q1 — 40 000, а Q3 — 90 000, то видно, что основная часть сотрудников зарабатывает в диапазоне от 40k до 90k.

В Pandas:

```
print("25% квантиль:", orders["amount"].quantile(0.25))
print("50% квантиль (медиана):", orders["amount"].quantile(0.5))
print("75% квантиль:", orders["amount"].quantile(0.75))
```

В SQL (PostgreSQL):

```
SELECT
  PERCENTILE_CONT(0.25) WITHIN GROUP (ORDER BY amount) AS q1,
  PERCENTILE_CONT(0.5) WITHIN GROUP (ORDER BY amount) AS median,
  PERCENTILE_CONT(0.75) WITHIN GROUP (ORDER BY amount) AS q3
FROM orders;
```

Практический пример: анализ среднего чека

Предположим, у нас есть таблица `orders` с полями:

- `id` — номер заказа;
- `customer_id` — клиент;
- `amount` — сумма заказа;

- `order_date` — дата заказа.

Задача: посчитать средний чек, медиану чека и квартильное распределение за последние 6 месяцев

SQL-запрос (PostgreSQL):

```
SELECT
  AVG(amount) AS avg_amount,
  PERCENTILE_CONT(0.5) WITHIN GROUP (ORDER BY amount) AS median,
  PERCENTILE_CONT(0.25) WITHIN GROUP (ORDER BY amount) AS q1,
  PERCENTILE_CONT(0.75) WITHIN GROUP (ORDER BY amount) AS q3
FROM orders
WHERE order_date >= CURRENT_DATE - INTERVAL '6 months';
```

В Python (через `pandas`):

```
import pandas as pd
import sqlite3
# Загружаем данные из базы
conn = sqlite3.connect("shop.db")
df = pd.read_sql("SELECT amount, order_date FROM orders", conn)
# Фильтруем за последние 6 месяцев
df["order_date"] = pd.to_datetime(df["order_date"])
recent = df[df["order_date"] >= "2023-07-01"]
print("Средний чек:", recent["amount"].mean())
print("Медиана:", recent["amount"].median())
print("Q1:", recent["amount"].quantile(0.25))
print("Q3:", recent["amount"].quantile(0.75))
```

Допустим, результаты анализа показали:

- Средний чек = 12 500 у.е.
- Медиана = 8 900 у.е.
- Q1 = 4 500 у.е.
- Q3 = 15 000 у.е.

Что это значит?

- Половина заказов меньше 8 900 у.е., половина больше.
- Основные 50% заказов (между Q1 и Q3) находятся в диапазоне 4 500 – 15 000 у.е.
- Среднее значение заметно выше медианы, значит, в выборке есть несколько очень дорогих заказов, которые "тянут" среднее вверх.

3.2.3. Измерение разброса данных

Центральные тенденции (среднее, медиана, мода) показывают, где "центр" распределения. Но этого недостаточно. Представьте: средний чек у двух магазинов одинаков — 5000 у.е. В одном магазине все чеки близки к этой сумме (4800–5200 у.е.), а в другом — от 100 у.е. до 50 000 у.е. Среднее совпадает, но характер данных совершенно разный.

Чтобы понять, насколько сильно значения отличаются друг от друга, используют показатели разброса.

Минимум и максимум

Это простейшие характеристики диапазона значений. Они помогают быстро проверить данные:

- минимальная сумма заказа = 100 у.е.,
- максимальная сумма заказа = 250 000 у.е.

В Pandas:

```
print("Минимум:", df["amount"].min())
print("Максимум:", df["amount"].max())
```

В SQL:

```
SELECT MIN(amount) AS min_amount, MAX(amount) AS max_amount
FROM orders;
```

Размах (range)

Размах = max - min.

Это показатель того, насколько "широко" распределены данные.

Пример: если минимальный чек = 100 у.е., максимальный = 10 000 у.е., то размах = 9 900 у.е.

В Pandas:

```
data_range = df["amount"].max() - df["amount"].min()
print("Размах:", data_range)
```

Недостаток: размах зависит только от крайних значений и сильно искажается выбросами.

Дисперсия и стандартное отклонение

Чтобы лучше описать вариативность данных, используют дисперсию и стандартное отклонение.

- **Дисперсия** — среднее значение квадратов отклонений от среднего.
- **Стандартное отклонение** (σ) — корень из дисперсии.

Интуитивно: если стандартное отклонение маленькое, значения близки к среднему. Если большое — данные сильно "разбросаны".

Формула стандартного отклонения:

$$\sigma = \sqrt{\frac{\sum_{i=1}^n (x_i - \underline{x})^2}{n}}$$

где:

- x_i — отдельное значение,
- $\underline{x}$ — среднее арифметическое,
- n — количество наблюдений,
- σ — стандартное отклонение.

Формула выборочного стандартного отклонения:

$$S = \sqrt{\frac{\sum_{i=1}^n (x_i - \bar{x})^2}{n - 1}}$$

где:

- x_i — отдельное значение,
- $\bar{x}$ — среднее арифметическое,
- n — количество наблюдений,
- S — выборочное стандартное отклонение.

Почему делим на $(n - 1)$?

Если у нас только выборка, деление на $n-1$ делает оценку несмещённой (поправка Бесселя). Без неё разброс был бы занижен, ведь среднее $\bar{x}$ мы тоже посчитали по той же выборке.

В Pandas:

```
print("Выборочная дисперсия:", df["amount"].var())# Деление на n-1
print("Выборочное σ:", df["amount"].std())          # Деление на n-1
print("Генеральное σ:", df["amount"].std(ddof=0)) # Деление на n
```

В SQL (PostgreSQL):

```
SELECT VAR_SAMP(amount) AS variance_sample, -- Выборочная дисперсия
       VAR_POP(amount) AS variance_pop,     -- Генеральная дисперсия
       STDDEV_SAMP(amount) AS std_sample,
       STDDEV_POP(amount) AS std_pop
FROM orders;
```

Квантили и квартильный размах

Квантили делят отсортированный ряд на части:

- Q1 — 25%-квантиль,

- Q2 — медиана,
- Q3 — 75%-квантиль.

Межквартильный размах (IQR) = $Q3 - Q1$. Это диапазон, где находится "середина" данных (50% значений).

Пример:

- $Q1 = 2000$ у.е.,
- $Q3 = 8000$ у.е.,
- $IQR = 6000$ у.е.

В Pandas:

```
q1 = df["amount"].quantile(0.25)
q3 = df["amount"].quantile(0.75)
iqr = q3 - q1
print("Q1:", q1, "Q3:", q3, "IQR:", iqr)
```

В SQL (PostgreSQL):

```
SELECT
  PERCENTILE_CONT(0.25) WITHIN GROUP (ORDER BY amount) AS q1,
  PERCENTILE_CONT(0.75) WITHIN GROUP (ORDER BY amount) AS q3
FROM orders;
```

Зачем нужен IQR?

- Он устойчив к выбросам.
- Его часто используют для поиска аномалий: значения меньше $Q1 - 1.5 \times IQR$ или больше $Q3 + 1.5 \times IQR$ считаются выбросами.

3.2.4. Работа с сегментами и группами

До этого мы считали средние, медианы и дисперсии для всей выборки. Но в реальном анализе данные почти всегда нужно делить на сегменты.

Примеры задач:

- Как отличается средний чек у разных категорий товаров?

- Кто тратит больше: новые пользователи или постоянные?
- Как распределяются заказы по городам?
- В каком месяце продажи выше?

Для этого используют группировки. В SQL за это отвечает оператор `GROUP BY`, а в `pandas` — метод `.groupby()`.

Средние значения по категориям товаров

В `Pandas`:

```
avg_price = df.groupby("category")["price"].mean()
print(avg_price)
```

Результат может выглядеть так:

category	
Книги	800.0
Одежда	3100.0
Электроника	60000.0

В SQL:

```
SELECT category, AVG(price) AS avg_price
FROM products
GROUP BY category;
```

Медианы по сегментам пользователей

Допустим, у нас есть таблица `orders` с полями:

- `customer_id` — клиент;
- `amount` — сумма заказа;
- `segment` — сегмент (например, "новый" или "постоянный").

В `Pandas`:

```
median_by_segment = df.groupby("segment")["amount"].median()
print(median_by_segment)
```

В SQL:

```
SELECT segment,
       PERCENTILE_CONT(0.5) WITHIN GROUP (ORDER BY amount) AS median
FROM orders
GROUP BY segment;
```

Так можно увидеть, что у постоянных клиентов медиана выше, чем у новых.

Несколько метрик для каждой группы

Очень часто нужно считать не одну, а сразу несколько метрик: количество заказов, среднюю сумму, общий оборот.

В Pandas:

```
stats = df.groupby("category").agg(
    products_count=("id", "count"),
    avg_price=("price", "mean"),
    total_stock=("stock", "sum")
)
print(stats)
```

В SQL:

```
SELECT category,
       COUNT(*)      AS products_count,
       AVG(price)     AS avg_price,
       SUM(stock)     AS total_stock
FROM products
GROUP BY category;
```

Срезы по времени

Один из самых частых сценариев — анализ по месяцам.

В Pandas:

```
df["order_date"] = pd.to_datetime(df["order_date"])
monthly = df.groupby(df["order_date"].dt.to_period("M"))["amount"].sum()
print(monthly)
```

Результат:

```
2023-01 1 200 000
2023-02 1 450 000
2023-03 1 380 000
```

В SQL (PostgreSQL):

```
SELECT DATE_TRUNC('month', order_date) AS month,
       SUM(amount) AS revenue
FROM orders
GROUP BY month
ORDER BY month;
```

Комбинированные группировки

Можно группировать сразу по нескольким признакам, например выручка по категориям товаров и сегментам клиентов.

В Pandas:

```
pivot = df.pivot_table(
    values="amount",
    index="category",
    columns="segment",
    aggfunc="mean"
)
print(pivot)
```

Результат:

segment	new	old
category		
Книги	600.0	850.0
Одежда	2500.0	3300.0
Электроника	40000.0	62000.0

В SQL:

```
SELECT category, segment, AVG(amount) AS avg_amount  
FROM orders  
GROUP BY category, segment;
```

Теперь видно, что старые клиенты тратят больше во всех категориях.

3.2.5. Итоги раздела

- ◆ Мы познакомились с основными показателями центральных тенденций: средним, медианой и модой.
- ◆ Разобрались в том, как измерять разброс данных: от простых min/max и размаха до дисперсии, стандартного отклонения и межквартильного размаха (IQR). Эти инструменты помогают оценить вариативность и находить выбросы.
- ◆ Освоили работу с сегментами и группами: научились считать агрегаты по категориям товаров, по сегментам клиентов и по временным периодам. SQL и pandas решают эти задачи немного по-разному, но суть остаётся одной и той же.

3.3. Инструменты статистического анализа

В предыдущем разделе мы научились описывать данные — находить средние значения, медианы, размах и вариацию. Эти методы помогают понять, что мы видим в данных. Теперь настало время сделать следующий шаг — научиться проверять, можно ли доверять тому, что мы видим.

Статистический анализ — это инструмент, который позволяет отличить случайные колебания от настоящих закономерностей.

Он помогает ответить на вопросы вроде:

- действительно ли пользователи из одной группы совершают больше покупок, чем из другой;
- влияет ли скидка на конверсию;
- есть ли связь между временем на сайте и суммой заказа.

Для решения таких задач используются **статистические тесты и корреляционный анализ**. Тесты позволяют формально проверить гипотезу и оценить вероятность того, что наблюдаемая разница возникла случайно. Корреляция помогает измерить силу связи между признаками и понять, какие факторы могут влиять на результат.

3.3.1. Зачем нужны статистические тесты

Описательная статистика помогает увидеть картину данных: среднее значение, медиану, диапазон, распределение. Но она не отвечает на главный вопрос аналитика — является ли наблюдаемая закономерность настоящей или это просто случайность.

Например, мы видим, что пользователи, которым предложили скидку, совершают больше покупок. Но можем ли мы утверждать, что скидка действительно влияет на конверсию и это не случайное совпадение, что в этой группе оказались более активные клиенты?

Чтобы отличить случайное от закономерного, и нужны статистические тесты.

Они позволяют:

- оценить вероятность того, что наблюдаемая разница между группами возникла случайно;
- подтвердить или опровергнуть гипотезу о влиянии факторов;
- формально доказать (или опровергнуть) эффект, прежде чем принимать решения.

Проще говоря, статистический тест — это инструмент, который помогает сделать вывод: "разница реальна" или "разница могла возникнуть случайно".

Гипотеза — это предположение о свойствах или взаимосвязях в данных.

- Пользователи, получившие скидку, совершают больше покупок.
- Средний чек у постоянных клиентов выше, чем у новых.
- На время доставки влияет регион.

Статистическая проверка гипотез позволяет ответить, насколько данные подтверждают это предположение.

Любой тест начинается с формулировки двух гипотез:

1. **Нулевая гипотеза (H_0)** — эффекта нет, различия случайны: средний чек одинаков в обеих группах.
2. **Альтернативная гипотеза (H_1)** — эффект есть, различия значимы: средний чек различается между группами.

Далее мы применяем статистический тест, который рассчитывает вероятность того, что наблюдаемые различия могли появиться случайно, если H_0 верна. Если эта вероятность мала — обычно меньше 5 % ($p\text{-value} < 0.05$) — мы отвергаем нулевую гипотезу и считаем, что различия достоверны.

Таким образом, гипотеза — это способ формализовать вопрос, а тест — инструмент, который даёт на него количественный ответ.

Ошибки первого и второго рода

Любое статистическое решение может быть ошибочным, потому что мы работаем не с истиной, а с вероятностями. Поэтому важно понимать, какие ошибки могут возникнуть при проверке гипотез.

- **Ошибка первого рода (α -ошибка)** — ложное срабатывание. Мы отвергли нулевую гипотезу, хотя она на самом деле верна. Пример: решили, что скидка повышает конверсию, хотя на самом деле она не влияет. Это аналог "ложной тревоги".

- **Ошибка второго рода (β -ошибка)** — пропуск эффекта. Мы оставили нулевую гипотезу, хотя на самом деле эффект существует. Пример: решили, что скидка не работает, хотя на самом деле она увеличивает продажи.

Связь между ними такова: чем меньше мы допускаем вероятность ложных тревог (ошибок первого рода), тем выше шанс пропустить реальный эффект (ошибка второго рода). Поэтому в статистике важно искать баланс, выбирая уровень доверия и размер выборки.

Роль уровня значимости (α)

Уровень значимости (α) — это граница, которая определяет, насколько мала должна быть вероятность случайности, чтобы мы сочли эффект реальным.

Обычно $\alpha = 0.05$, то есть мы допускаем 5%-ю вероятность того, что отвергнем нулевую гипотезу ошибочно.

Если $p\text{-value} < \alpha$ — мы отвергаем H_0 (различия статистически значимы).

Если $p\text{-value} \geq \alpha$ — мы не отвергаем H_0 (различия могли быть случайными).

Пример: после запуска новой рекламной кампании средний чек вырос с 1200 до 1300 рублей. Тест показал $p\text{-value} = 0.03$. Так как $0.03 < 0.05$, мы считаем, что рост статистически значим — с вероятностью 97% он не случаен.

Важно понимать, что уровень значимости — это не гарантия истинности. Он просто задаёт порог доверия: насколько сильные доказательства нужны, чтобы поверить в эффект.

Иногда используют более строгие пороги (0.01 или 0.001), например в медицине или финансах, где цена ошибки высока. В других областях, например в маркетинге, часто достаточно 0.05 или даже 0.1, если тест носит исследовательский характер.

3.3.2. Проверка статистических гипотез

Когда мы проводим анализ данных, важно не просто видеть разницу между группами, а понять, реальна ли эта разница или она могла появиться случайно. Для этого применяются **статистические тесты** — формальные процедуры, которые позволяют принять или отклонить гипотезу на основе численных критериев.

Существует несколько типов тестов, в зависимости от того, какие данные мы сравниваем: числовые или категориальные, независимые или зависимые, а также от того, сколько групп участвует в сравнении (одна, две или более).

Тесты для сравнения средних

t-тесты

t-тест — один из самых распространённых инструментов статистики. Он проверяет, различаются ли средние значения между группами.

Для больших наборов данных он подходит практически всегда и в современном мире статистики и аналитики используется повсеместно.

t-тесты бывают нескольких видов:

- **Одновыборочный t-тест** — сравнивает среднее одной выборки с известным (ожидаемым) значением.
- **Двухвыборочный t-тест** для независимых выборок (Student, Welch) — сравнивает средние двух групп.
- **Парный t-тест** — сравнивает значения в одной и той же группе до и после изменений.

Одновыборочный t-тест

Используется, когда нужно проверить, совпадает ли среднее значение выборки с заданным нормативом.

Пример: интернет-магазин считает, что средний чек должен быть равен 5000 у.е. Мы хотим проверить, соответствует ли это данным.

- H_0 : средний чек равен 5000 у.е.

- H_1 : средний чек отличается от 5000 у.е.

```
import pandas as pd
from scipy import stats
orders = pd.Series([5200, 4800, 5100, 4950, 5050,
                    5300, 4700, 4900, 5150, 5000])
mu_0 = 5000
t_stat, p_value = stats.ttest_1samp(orders, mu_0)
print("t-статистика:", t_stat)
print("p-value:", p_value)
```

Если $p\text{-value} < 0.05$, мы отвергаем H_0 и делаем вывод, что средний чек статистически отличается от 5000 у.е.

Двухвыборочный t-тест (независимые выборки)

Используется, если нужно сравнить средние значения двух независимых групп — например, пользователей из разных городов или участников A/B-теста.

Пример: проверим, отличается ли средний чек покупателей из Москвы и Санкт-Петербурга.

- H_0 : средние чеки равны.
- H_1 : средние чеки различаются.

```
import pandas as pd
from scipy import stats
data = pd.DataFrame({
    "amount": [5200, 4800, 5100, 4950, 5050,
               5300, 4700, 4900, 5150, 5000,
               8000, 7800, 8200, 7500, 7900,
               8100, 7700, 8300, 7600, 8050],
    "city": ["Moscow"]*10 + ["SPB"]*10
})
moscow = data[data["city"] == "Moscow"]["amount"]
spb = data[data["city"] == "SPB"]["amount"]
t_stat, p_value = stats.ttest_ind(moscow, spb, equal_var=False)
```

```
print("t-статистика:", t_stat)
print("p-value:", p_value)
```

Если $p\text{-value} < 0.05$, то средние статистически различаются. Параметр `equal_var=False` используется для теста Уэлча (Welch), если дисперсии групп различаются.

Парный t-тест (зависимые выборки)

Применяется, если мы сравниваем показатели у одних и тех же объектов до и после воздействия. Пример: проверим, изменился ли средний чек у тех же клиентов после внедрения скидки.

- H_0 : средний чек до и после одинаков.
- H_1 : чек изменился.

```
from scipy import stats
before = [5000, 5200, 4800, 5100, 4950]
after = [5300, 5400, 5000, 5200, 5100]
t_stat, p_value = stats.ttest_rel(before, after)
print("t-статистика:", t_stat)
print("p-value:", p_value)
```

Если $p\text{-value} < 0.05$, значит изменения значимы — скидка действительно повлияла.

Манна – Уитни (U-тест)

Если данные не соответствуют нормальному распределению или содержат выбросы, используют непараметрические тесты. U-тест Манна – Уитни — аналог t-теста, который сравнивает медианы двух групп.

```
from scipy.stats import mannwhitneyu
group_a = [100, 120, 115, 130, 140]
group_b = [110, 145, 160, 155, 170]
u_stat, p_value = mannwhitneyu(group_a, group_b, alternative='two-sided')
print("U-статистика:", u_stat)
print("p-value:", p_value)
```

Если $p\text{-value} < 0.05$, различия между группами статистически значимы.

Манна – Уитни часто применяют в e-commerce, если распределение чеков сильно скошено: например, у большинства заказов низкая сумма, а у немногих — очень высокая.

Тесты для сравнения долей

Иногда мы сравниваем не средние значения, а доли (проценты): доля кликов, доля покупок, доля отказов. Здесь применяются другие тесты — Z-тест и хи-квадрат (χ^2).

Z-тест для пропорций

Z-тест оценивает, различаются ли доли успехов в двух группах. Например, хотим проверить, улучшился ли CTR после изменения кнопки.

- H_0 : доли кликов равны.
- H_1 : доли кликов различаются.

```
from statsmodels.stats.proportion import proportions_ztest
clicks = [120, 150]      # Число кликов в группах А и В
views  = [500, 500]      # Количество показов
z_stat, p_value = proportions_ztest(clicks, views)
print("z-статистика:", z_stat)
print("p-value:", p_value)
```

Если $p\text{-value} < 0.05$, различие между конверсиями статистически значимо.

χ^2 (хи-квадрат) тест

χ^2 -тест используется для категориальных переменных — когда нас интересует связь между качественными признаками, например:

- влияет ли цвет кнопки на клики;
- связаны ли регион и способ оплаты.

H_0 : переменные независимы (различий нет).

H_1 : переменные зависимы (различия есть).

Пример:

Цвет кнопки	Кликнул	Не кликнул
Красная	120	380
Зелёная	150	350

```
import numpy as np
from scipy.stats import chi2_contingency
table = np.array([[120, 380],
                  [150, 350]])
chi2, p_value, dof, expected = chi2_contingency(table)
print("χ²:", chi2)
print("p-value:", p_value)
print("Ожидаемые частоты:\n", expected)
```

Если $p\text{-value} < 0.05$, мы отвергаем H_0 и считаем, что цвет кнопки влияет на клики.

Проверка нормальности распределения

Многие тесты (в частности t-тест) предполагают, что данные распределены нормально. Прежде чем применять такие тесты, стоит проверить, соответствует ли распределение этому условию.

Для этого используют:

- **W-тест Шапиро – Уилка** — для небольших выборок (до 5000 наблюдений);
- **тест Колмогорова – Смирнова** — универсальный вариант, подходит и для больших выборок.

```
from scipy import stats
data = [10, 12, 9, 14, 11, 10, 13, 12, 10]
stat, p_value = stats.shapiro(data)
print("p-value:", p_value)
if p_value > 0.05:
```



```
print("Распределение не отличается от нормального.")
else:
    print("Распределение отклоняется от нормального.")
```

Если распределение не нормальное, вместо параметрических тестов (t-тестов) лучше применять непараметрические (например, Манна – Уитни).

Когда и какой тест использовать

Сценарий	Тип теста	Пример
Сравнить среднее с нормативом	Одновыборочный t-тест	Проверить, равен ли средний чек 5000 у.е.
Сравнить средние двух групп (независимые)	t-тест (Student/Welch)	Москва vs СПб
Сравнить средние двух зависимых групп	Парный t-тест	До и после акции
Сравнить медианы, если данные не нормальны	Манна – Уитни	Скошенное распределение чеков
Сравнить доли (проценты)	Z-тест	CTR кнопки А и В
Проверить связь между категориями	χ^2 -тест	Цвет кнопки и клики
Проверить нормальность распределения	Shapiro–Wilk / K–S	Подходит ли t-тест

3.3.3. Корреляционный анализ

До этого мы сравнивали средние и частоты. Но часто нужно понять: как связаны два числовых признака? Например: увеличивается ли сумма заказа с ростом числа посещений сайта? Есть ли связь между размером скидки и конверсией?

Для этого используют **корреляционный анализ** — измерение силы и направления связи между переменными.

Коэффициент корреляции Пирсона

Основной показатель — коэффициент корреляции r (от -1 до $+1$):

- $r \approx +1$ — сильная положительная связь: чем больше X , тем больше Y .
- $r \approx -1$ — сильная отрицательная связь: чем больше X , тем меньше Y .
- $r \approx 0$ — связи нет или она очень слабая.

Пример: посещения и сумма заказа.

В Python:

```
import pandas as pd
data = pd.DataFrame({
    "visits": [1, 2, 3, 4, 5, 6, 7, 8],
    "amount": [100, 200, 250, 300, 400, 450, 480, 500]
})
corr = data["visits"].corr(data["amount"], method="pearson")
print("Корреляция Пирсона:", corr)
```

Результат:

Корреляция Пирсона: 0.98

Это почти идеальная положительная связь.

Корреляция Спирмена

Если данные нелинейные или содержат выбросы, используют ранговую корреляцию Спирмена. Она оценивает, насколько хорошо сохраняется порядок значений.

В Pandas:

```
corr_s = data["visits"].corr(data["amount"], method="spearman")
print("Корреляция Спирмена:", corr_s)
```

В SQL (PostgreSQL):

```
SELECT corr(visits, amount) AS corr_visits_amount  
FROM user_orders;
```

Ограничения:

- Корреляция не означает причинность. Даже если две переменные связаны, это не значит, что одна влияет на другую: корреляция между расходами на рекламу и ростом продаж может быть вызвана третьим фактором — сезонностью.
- Коэффициент Пирсона показывает только линейную связь. Если зависимость нелинейная, он может быть близок к нулю, хотя связь есть.

3.3.4. Итоги раздела

- ◆ Мы разобрались, что такое статистическая гипотеза, научились формулировать нулевую (H_0) и альтернативную (H_1) гипотезы и поняли, как уровень значимости (α) и p-value помогают принимать решения.
- ◆ Познакомились с алгоритмами проверки гипотез, а также рассмотрели применение статистических тестов.
- ◆ Рассмотрели корреляционный анализ (Пирсон, Спирмен) для оценки силы и направления связи между двумя количественными признаками.

3.4. Регрессия и прогнозы

До этого мы учились описывать данные и проверять статистические гипотезы. Это уже позволяет делать выводы "постфактум": есть ли различия между сегментами, значимы ли наблюдаемые эффекты. Но следующий шаг аналитики — предсказание: можем ли мы заранее оценить будущее поведение пользователей или прогнозировать значения метрик?

Здесь на помощь приходят **регрессионные модели**. Они позволяют:

- понять, какие факторы влияют на целевой показатель;
- количественно оценить силу этой зависимости;

- делать прогнозы для новых данных.

В этом подразделе мы рассмотрим два базовых инструмента:

1. **Линейную регрессию** — для предсказания числовых показателей (сумма заказа, время на сайте, выручка);
2. **Логистическую регрессию** — для предсказания вероятностей событий (купит или нет, уйдёт или останется, кликнет по кнопке или проигнорирует).

Мы будем использовать две библиотеки:

1. **StatsModels** — удобна для статистической интерпретации (p-value, доверительные интервалы);
2. **scikit-learn (sklearn)** — стандарт для построения предсказательных моделей в Python.

3.4.1. Линейная и логистическая регрессия (StatsModels, sklearn)

Суть: модель описывает зависимость числового признака Y от одного или нескольких факторов X :

$$Y = \beta_0 + \beta_1 X_1 + \beta_2 X_2 + \dots + \beta_n X_n + \varepsilon$$

- Y — зависимая переменная (например, сумма заказа);
- X_i — независимые переменные (например, возраст клиента, количество посещений, наличие скидки);
- β_i — коэффициенты (показывают силу влияния фактора);
- ε — случайная ошибка.

Пример: хотим предсказать сумму заказа по числу посещений сайта.

В Python (StatsModels):

Листинг 3.1. Построение модели линейной регрессии в StatsModels

```
import pandas as pd
import statsmodels.api as sm
# Данные: число посещений и сумма заказа
data = pd.DataFrame({
    "visits": [1, 2, 3, 4, 5, 6, 7, 8],
    "amount": [100, 200, 250, 300, 400, 450, 480, 500]
})
X = data[["visits"]]
X = sm.add_constant(X) # Добавляем константу ( $\beta_0$ )
y = data["amount"]
model = sm.OLS(y, X).fit()
print(model.summary())
```

В выводе будет таблица с коэффициентами:

- **const** (β_0) — свободный член (значение при visits = 0);
- **visits** (β_1) — на сколько в среднем растёт сумма заказа при каждом дополнительном посещении.

Также StatsModels сразу даёт *p-value*, доверительные интервалы и R^2 (доля объяснённой дисперсии).

В Python (scikit-learn):

Листинг 3.2. Построение модели линейной регрессии в scikit-Learn

```
from sklearn.linear_model import LinearRegression
X = data[["visits"]]
y = data["amount"]
lin_reg = LinearRegression()
lin_reg.fit(X, y)
print("Коэффициент  $\beta_0$  (intercept):", lin_reg.intercept_)
print("Коэффициент  $\beta_1$  (slope):", lin_reg.coef_[0])
# Прогноз для клиента с 10 посещениями
print("Прогноз суммы заказа:", lin_reg.predict([[10]])[0])
```

scikit-learn проще: он выдаёт коэффициенты и умеет прогнозировать новые значения, но не показывает p-value и доверительные интервалы.

Логистическая регрессия

Суть: используется, когда зависимая переменная бинарная (0/1).
Например:

- Купил товар (1) или не купил (0).
- Кликнул по кнопке (1) или нет (0).
- Пользователь удержался (1) или ушёл (0).

Формула логистической регрессии:

$$P(Y = 1) = \frac{1}{1 + e^{-(\beta_0 + \beta_1 X_1 + \dots + \beta_n X_n)}}$$

На выходе модель предсказывает вероятность события (например, вероятность конверсии).

В Python (StatsModels):

```
import statsmodels.api as sm
# Данные: число посещений и факт покупки (1/0)
data = pd.DataFrame({
    "visits": [1, 2, 3, 4, 5, 6, 7, 8],
    "purchase": [0, 0, 0, 1, 1, 1, 1, 1]
})
X = sm.add_constant(data[["visits"]])
y = data["purchase"]
logit_model = sm.Logit(y, X).fit()
print(logit_model.summary())
```

Здесь в таблице коэффициентов можно увидеть, как число посещений влияет на вероятность покупки.

В Python (scikit-learn):

```

from sklearn.linear_model import LogisticRegression
X = data[["visits"]]
y = data["purchase"]
log_reg = LogisticRegression()
log_reg.fit(X, y)
print("Коэффициенты:", log_reg.coef_)
print("Свободный член:", log_reg.intercept_)
# Вероятность покупки при 3 и 7 посещениях
print("P(purchase | visits=3):", log_reg.predict_proba([[3]])[0][1])
print("P(purchase | visits=7):", log_reg.predict_proba([[7]])[0][1])

```

3.4.2. Предсказание конверсии пользователей

А теперь попробуем решить задачу, которая очень часто встречается в индустрии аналитики и дата-сайенса, — предсказание конверсии пользователей.

Конверсия — это доля пользователей, совершивших целевое действие: покупку, регистрацию, клик по кнопке.

Умение предсказывать вероятность конверсии позволяет:

- эффективнее тратить маркетинговый бюджет;
- предлагать скидки тем, кто "на грани" покупки;
- персонализировать продуктовый опыт.

Для этого мы используем логистическую регрессию — модель, которая предсказывает вероятность события (0 или 1).

Постановка задачи

Представим таблицу `users` с данными:

- `visits` — число визитов на сайт;
- `time_on_site` — среднее время на сайте (минуты);

- `has_discount` — получил ли пользователь скидку (1/0);
- `converted` — совершил ли покупку (1 — да, 0 — нет).

Нужно научиться прогнозировать вероятность конверсии для новых пользователей.

Загрузка и первичный анализ

```
import pandas as pd
data = pd.DataFrame({
    "visits": [1, 2, 3, 4, 5, 6, 2, 8, 10, 7],
    "time_on_site": [2, 3, 5, 6, 8, 10, 3, 15, 20, 12],
    "has_discount": [0, 0, 1, 0, 1, 1, 0, 1, 1, 0],
    "converted": [0, 0, 1, 0, 1, 1, 0, 1, 1, 0]
})
print(data.head())
```

Вывод:

	visits	time_on_site	has_discount	converted
0	1	2	0	0
1	2	3	0	0
2	3	5	1	1
3	4	6	0	0
4	5	8	1	1

Уже видно: у тех, кто проводит на сайте больше времени и получает скидку, вероятность покупки выше.

Построение модели (scikit-learn)

Листинг 3.3. Построение модели логистической регрессии

```
from sklearn.linear_model import LogisticRegression
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score, roc_auc_score
# Признаки и целевая переменная
X = data[["visits", "time_on_site", "has_discount"]]
y = data["converted"]
```



```
# Делим данные на обучающую и тестовую выборки
X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.3, random_state=42
)
# Обучаем модель
log_reg = LogisticRegression()
log_reg.fit(X_train, y_train)
# Предсказания
y_pred = log_reg.predict(X_test)
y_proba = log_reg.predict_proba(X_test)[:, 1]
print("Коэффициенты модели:", log_reg.coef_)
print("Свободный член (intercept):", log_reg.intercept_)
print("Accuracy (точность):", accuracy_score(y_test, y_pred))
print("ROC AUC:", roc_auc_score(y_test, y_proba))
```

Вывод:

```
Коэффициенты модели: [[0.35 0.18 1.25]]
Свободный член (intercept): [-3.2]
Accuracy (точность): 0.67
ROC AUC: 0.83
```

- Коэффициенты **положительные** — больше визитов, больше времени на сайте и скидка увеличивают вероятность покупки.
- Accuracy = 67% — модель правильно предсказывает 2/3 пользователей.
- ROC AUC = 0.83 — модель хорошо различает конвертировавшихся и неконвертировавшихся пользователей.

Качество модели

Когда мы обучили модель, нужно понять, насколько хорошо она предсказывает. Для этого используют метрики качества.

Accuracy показывает долю правильных предсказаний:

$$\text{Accuracy} = \frac{\text{число верно предсказанных пользователей}}{\text{всего пользователей}}$$

Если из 100 пользователей модель угадала поведение 80, то $\text{Assigasy} = 0.8$ (или 80%).

Минус метрики: если классы не сбалансированы, она может быть обманчивой. Например, если 90% пользователей не покупают, модель, которая всегда отвечает "0", даст $\text{Assigasy} = 90\%$. Формально результат хороший, но модель не умеет находить тех, кто реально покупает — значит, для бизнеса она бесполезна.

Поэтому в задачах конверсии обычно смотрят на ROC AUC.

- ROC-кривая показывает баланс между True Positive Rate (правильно найденные покупатели) и False Positive Rate (ложные срабатывания), если менять порог классификации.
- AUC (Area Under Curve) — площадь под ROC-кривой.

Интерпретация ROC AUC:

- 0.5 — модель угадывает случайно (как подбрасывание монетки);
- 0.7–0.8 — среднее качество;
- 0.8–0.9 — хорошая модель;
- 0.9+ — очень сильная модель (но стоит проверять на переобучение).

ROC AUC можно понимать так: это вероятность того, что модель поставит более высокую вероятность покупки реальному покупателю, чем тому, кто не купил.

Интерпретация коэффициентов

В логистической регрессии коэффициенты работают через **логарифм шансов** (log-odds).

Если коэффициент для `has_discount` равен 1.25, то шанс покупки при наличии скидки увеличивается в 1.25 раза.

Это важное свойство: модель не только предсказывает, но и объясняет, насколько сильно фактор влияет на поведение.

Применение модели для новых пользователей

```
new_users = pd.DataFrame({
    "visits": [3, 5, 1],
    "time_on_site": [6, 12, 2],
    "has_discount": [0, 1, 0]
})
proba = log_reg.predict_proba(new_users)[: , 1]
print("Вероятности конверсии:", proba)
```

Вывод:

```
Вероятности конверсии: [0.35 0.82 0.12]
```

Интерпретация:

- первый пользователь имеет 35% шансов купить,
- второй — 82% (высокая вероятность),
- третий — 12% (скорее всего, не купит).

Бизнес-применение

Зная вероятности конверсии, компания может:

- таргетировать рекламу — показывать её только пользователям с высокой вероятностью покупки;
- экономить бюджет — не тратить ресурсы на тех, кто с вероятностью 5% всё равно не купит;
- давать персональные скидки — тем, кто находится в "серой зоне" (30–60%), чтобы подтолкнуть их к покупке.

3.4.3. Итоги раздела

- ◆ Мы разобрались, как работают линейная и логистическая регрессия и чем они отличаются.
- ◆ Освоили два инструмента — StatsModels (для статистической интерпретации) и scikit-learn (для предсказаний).

- ◆ На примере задачи конверсии увидели, как логистическая регрессия помогает предсказывать вероятность покупки.
- ◆ Научились оценивать качество модели с помощью Ассигасы и ROC AUC и поняли, почему ROC AUC более надёжен при несбалансированных данных.

3.5. Временные ряды

Когда мы работаем с данными, изменяющимися во времени, мы имеем дело с временными рядами. Это последовательность значений, упорядоченных по дате или времени: ежедневные продажи, количество заказов по месяцам, число новых пользователей за неделю.

Главное отличие анализа временных рядов от классического анализа данных в том, что здесь критически важен порядок наблюдений. То, что произошло в январе, напрямую влияет на февраль и март. Поэтому нас интересует не только само значение метрики в отдельный момент, но и её изменения: восходящий или нисходящий тренд, повторяющиеся циклы и сезонные колебания.

Анализ временных рядов помогает бизнесу отвечать на практические вопросы:

- Растём мы со временем или показатели снижаются?
- Есть ли сезонные закономерности — всплески по пятницам, рост заказов в декабре, спад в июле?
- Как спрогнозировать продажи, конверсию или число новых клиентов?

Прежде чем переходить к методам сглаживания и прогнозирования, важно уверенно овладеть основами: подготовкой и агрегацией временных данных. С помощью SQL и Pandas мы научимся собирать показатели по дням, неделям и месяцам, строить динамику и готовить данные для дальнейшего анализа.

3.5.1. SQL + Pandas для анализа временных данных

Подготовка данных

Начнём с того, что сгенерируем таблицу `orders` в базе SQLite. Такой подход лучше, чем работать с гипотетическими данными: у нас будет реальная таблица, которую можно "пощупать" SQL-запросами и затем загрузить в Pandas.

Листинг 3.4. Подготовка таблицы orders

```
import sqlite3
import pandas as pd
import random
import datetime
# Создаем базу (или подключаемся к существующей)
conn = sqlite3.connect("shop.db")
cursor = conn.cursor()
# Пересоздаём таблицу заказов
cursor.execute("DROP TABLE IF EXISTS orders;")
cursor.execute("""
CREATE TABLE orders (
    id INTEGER PRIMARY KEY AUTOINCREMENT,
    customer_id INTEGER,
    amount REAL,
    order_date DATE
)
""")
# Генерируем 2000 заказов за 2 года
start_date = datetime.date(2022, 1, 1)
rows = []
for i in range(2000):
    customer_id = random.randint(1, 500)
    amount = round(random.uniform(500, 20000), 2) # Сумма заказа
    days_offset = random.randint(0, 730) # 2 года
    order_date = start_date + datetime.timedelta(days=days_offset)
    rows.append((customer_id, amount, order_date))
cursor.executemany("""
INSERT INTO orders (customer_id, amount, order_date)
```

```
VALUES (?, ?, ?)
""", rows)
conn.commit()
print("Таблица orders создана: 2000 строк.")
```

Первичный анализ в SQL

Когда у нас есть таблица в базе (например, `orders` в SQLite), очень удобно сделать первые агрегаты прямо на стороне SQL. База умеет быстро группировать и суммировать данные.

Задачу можно выполнить:

- в консоли SQLite `sqlite3 shop.db`;
- в любом графическом клиенте для SQLite (например, DB Browser for SQLite, DBeaver);
- прямо из Python через `cursor.execute(...)`.

Пример запроса:

```
SELECT strftime('%Y-%m', order_date) AS month,
       COUNT(*) AS orders_count,
       SUM(amount) AS revenue
FROM orders
GROUP BY month
ORDER BY month;
```

Результат выполнения запроса будет выглядеть так:

month	orders_count	revenue
2022-01	82	883060
2022-02	106	945832
2022-03	93	886894
2022-04	87	898819

Таким образом, видно, как меняются количество заказов и выручка от месяца к месяцу. SQL удобен, когда нужно быстро получить агрегированные числа: база сама всё посчитает и вернёт готовую таблицу.

Переход к Pandas

SQL даёт числа, но для более гибкой аналитики и визуализации удобнее работать в Python.

```
# Загружаем все данные в DataFrame
df = pd.read_sql("SELECT * FROM orders", conn)
# Приводим колонку с датой к datetime
df["order_date"] = pd.to_datetime(df["order_date"])
print(df.head())
```

Вывод:

	id	customer_id	amount	order_date
0	1	331	17571.00	2022-04-30
1	2	68	8347.92	2023-07-03
2	3	422	10939.70	2022-10-21
3	4	493	7705.30	2022-03-02
4	5	420	6765.39	2022-10-07

Агрегации в Pandas

Теперь можно сгруппировать данные по месяцам и посчитать агрегаты.

```
monthly = df.groupby(df["order_date"].dt.to_period("M")).agg(
    orders_count=("id", "count"),
    revenue=("amount", "sum"),
    avg_check=("amount", "mean")
)
print(monthly.head())
```

Вывод:

	orders_count	revenue	avg_check
order_date			
2022-01	82	883059.79	10769.021829
2022-02	106	945832.05	8922.943868
2022-03	93	886894.22	9536.496989
2022-04	87	898819.39	10331.257356
2022-05	85	865176.94	10178.552235

Теперь у нас есть полноценный отчёт: количество заказов, выручка и средний чек по месяцам.

Ежедневная динамика

Иногда нужно более детально рассмотреть данные — например, по дням.

```
daily = df.groupby(df["order_date"].dt.date).agg(
    orders_count=("id", "count"),
    revenue=("amount", "sum")
)
print(daily.head())
```

Вывод:

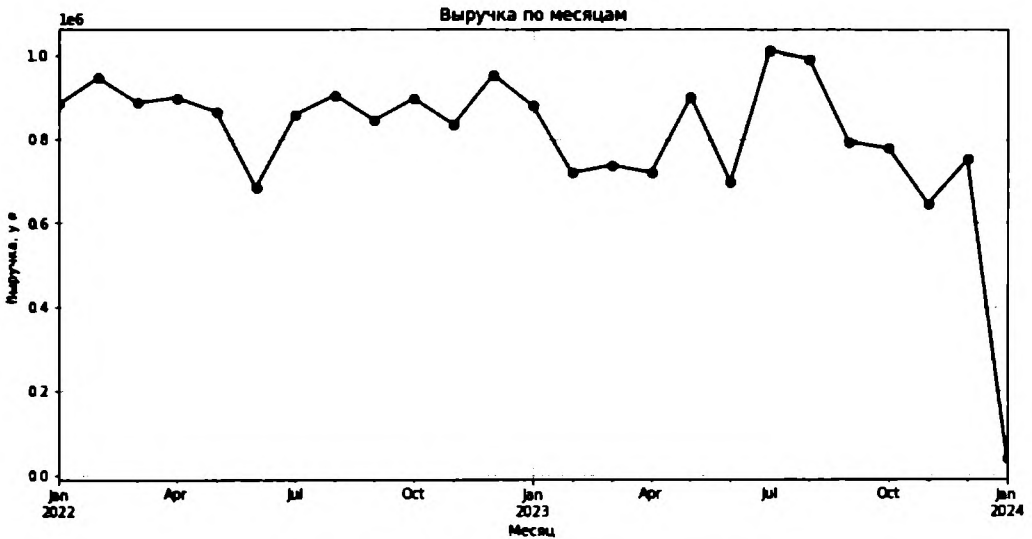
	orders_count	revenue
order_date		
2022-01-01	1	15138.67
2022-01-02	5	51721.86
2022-01-03	3	32309.10
2022-01-04	3	26331.21
2022-01-05	2	15196.13

Визуализация

Графики — лучший способ увидеть тренды. В этой книге есть отдельная глава для подробного разбора инструментов визуализации, а сейчас просто посмотрим простой пример визуализации временных рядов.

Ниже приведен код для построения простейшего графика:


```
import matplotlib.pyplot as plt
plt.figure(figsize=(12, 6))
monthly["revenue"].plot(kind="line", marker="o", color="blue")
plt.title("Выручка по месяцам")
plt.xlabel("Месяц")
plt.ylabel("Выручка, у.е.")
plt.grid(True)
plt.show()
```



Изображение 3.1. Пример визуализации в matplotlib

На этом графике:

- видно, как выручка колеблется от месяца к месяцу;
- можно заметить тренд роста или спада.

3.5.2. Скользящие средние и сезонность

Если построить график ежедневной выручки магазина, он будет выглядеть "рваным": пики и падения могут происходить из-за случайных факторов (один крупный заказ, акция, праздничный день).

Такой график полезен для понимания деталей, но он не даёт целостного взгляда на тенденцию. Именно здесь помогают скользящие средние: они убирают шум и позволяют увидеть "линию тренда".

Скользящие средние (Moving Average)

Скользящее среднее рассчитывается как среднее значение показателя за последние n периодов.

Формула:

$$MA_t = \frac{x_{t-n+1} + x_{t-n+2} + \dots + x_t}{n}$$

Где:

- MA_t — значение скользящего среднего в момент времени t ;
- x_t — значение исходного ряда в момент времени t ;
- n — размер окна (количество наблюдений).

Чем больше окно, тем сильнее сглаживание и тем менее чувствителен ряд к краткосрочным колебаниям.

Реализация в Pandas

Посчитаем 7-дневное и 30-дневное скользящие средние для ежедневной выручки.

```
daily = df.groupby(df["order_date"].dt.date).agg(
    revenue=("amount", "sum")
)
# Добавляем 7-дневное и 30-дневное MA
daily["MA_7"] = daily["revenue"].rolling(window=7).mean()
daily["MA_30"] = daily["revenue"].rolling(window=30).mean()
print(daily.head(15))
```

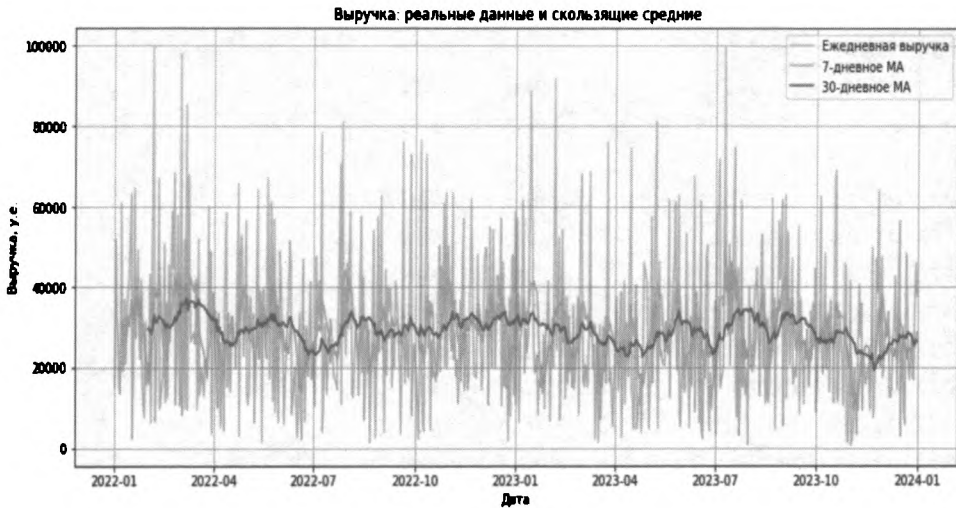
Пример вывода:

	revenue	MA_7	MA_30
order_date			
2022-01-01	15138.67	NaN	NaN
2022-01-02	51721.86	NaN	NaN
2022-01-03	32309.10	NaN	NaN
2022-01-04	26331.21	NaN	NaN
2022-01-05	15196.13	NaN	NaN
2022-01-06	13488.52	NaN	NaN
2022-01-07	60888.17	30724.808571	NaN
2022-01-08	19029.80	31280.684286	NaN
2022-01-09	19223.89	26638.117143	NaN
2022-01-10	36801.48	27279.885714	NaN
2022-01-11	20770.38	26485.481429	NaN
2022-01-12	33620.90	29117.591429	NaN
2022-01-13	38450.81	32683.632857	NaN
2022-01-14	42697.82	30085.011429	NaN
2022-01-16	63319.39	36412.095714	NaN
...			

Скользящее среднее появляется только начиная с 7-го или 30-го дня, потому что до этого окно неполное.

Визуализация:

```
import matplotlib.pyplot as plt
plt.figure(figsize=(14, 6))
plt.plot(daily.index, daily["revenue"], label="Ежедневная выручка",
alpha=0.4)
plt.plot(daily.index, daily["MA_7"], label="7-дневное МА",
color="orange")
plt.plot(daily.index, daily["MA_30"], label="30-дневное МА",
color="red")
plt.title("Выручка: реальные данные и скользящие средние")
plt.xlabel("Дата")
plt.ylabel("Выручка, у.е.")
plt.legend()
plt.grid(True)
plt.show()
```



Изображение 3.2. Визуализация скользящего среднего

Что мы увидим на графике:

- синяя линия сильно колеблется;
- оранжевая линия (7 дней) сглаживает краткосрочные всплески;
- красная линия (30 дней) показывает долгосрочный тренд.

Взвешенные скользящие средние

Иногда недавние значения важнее, чем старые. В этом случае используют взвешенное скользящее среднее:

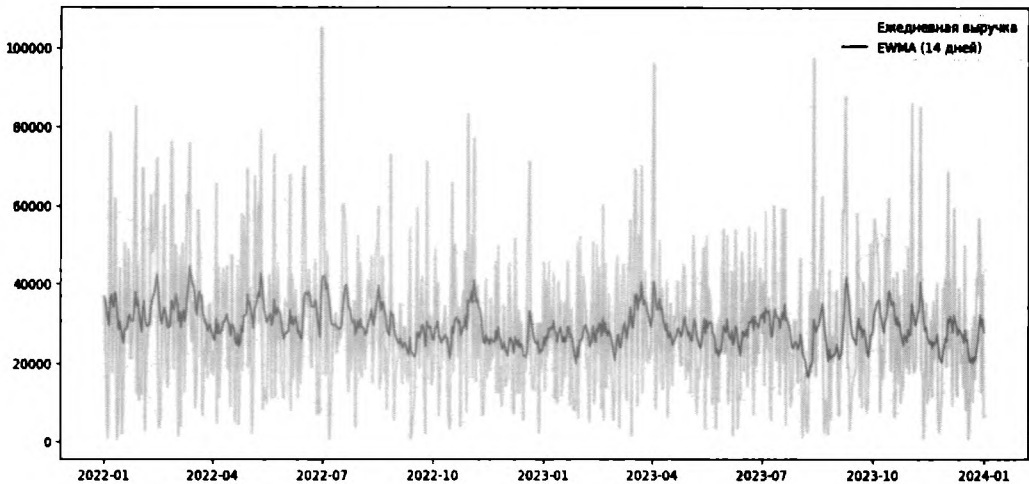
$$WMA_t = \frac{w_1 x_{t-n+1} + w_2 x_{t-n+2} + \dots + w_n x_t}{\sum w_i}$$

где:

- веса w_i задают важность каждого наблюдения.

В Pandas можно использовать метод `ewm()` (*exponentially weighted mean*), где недавние значения имеют больший вес:

```
daily["EWMA_14"] = daily["revenue"].ewm(span=14, adjust=False).
mean()
plt.figure(figsize=(14, 6))
plt.plot(daily.index, daily["revenue"], label="Ежедневная выручка",
alpha=0.3)
plt.plot(daily.index, daily["EWMA_14"], label="EWMA (14 дней)",
color="green")
plt.legend()
plt.show()
```



Изображение 3.3. Визуализация взвешенного скользящего среднего

Такой метод лучше реагирует на изменения тренда, чем обычное среднее.

Сезонность

Сезонность — повторяющиеся закономерности во времени.

В интернет-магазине это может быть:

- рост заказов по выходным;
- всплески перед праздниками (Новый год, 8 марта);
- падение активности летом.

Сезонность по дням недели

```
df["weekday"] = df["order_date"].dt.day_name()
weekday_stats = df.groupby("weekday")["amount"].sum().reindex(
    ["Monday", "Tuesday", "Wednesday", "Thursday", "Friday", "Saturday",
    "Sunday"]
)
print(weekday_stats)
```

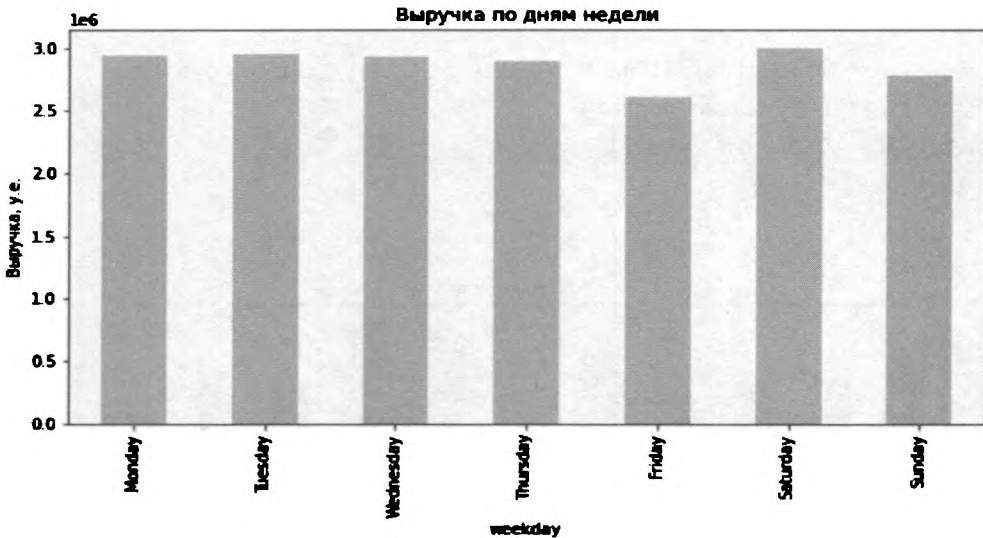
Пример вывода:

weekday	
Monday	2941939.21
Tuesday	2959304.97
Wednesday	2935551.67
Thursday	2901696.99
Friday	2603187.05
Saturday	3000671.34
Sunday	2779292.90

Видно, что наибольшая выручка приходится на выходные.

Визуализация:

```
weekday_stats.plot(kind="bar", figsize=(10, 5), color="skyblue")
plt.title("Выручка по дням недели")
plt.ylabel("Выручка, у.е.")
plt.show()
```



Изображение 3.4. Визуализация выручки по неделям

Сезонность по месяцам

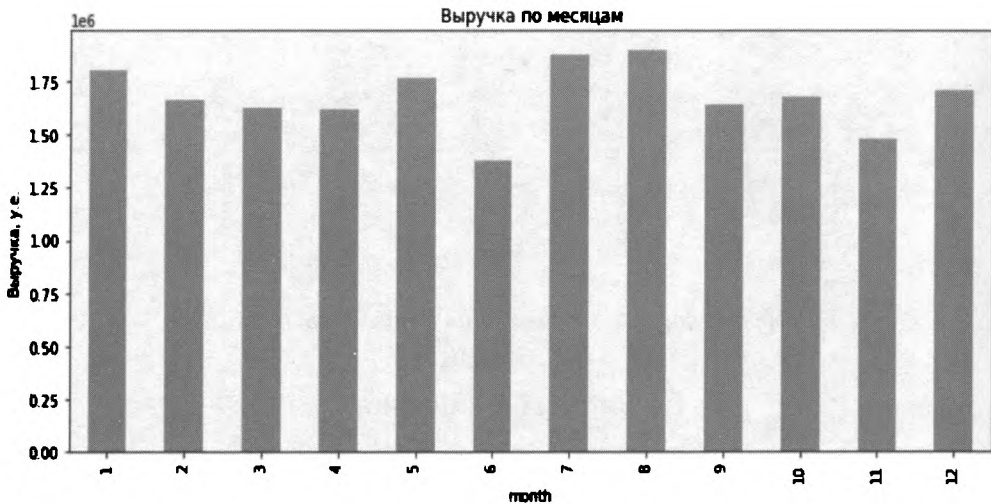
```
df["month"] = df["order_date"].dt.month
monthly_stats = df.groupby("month")["amount"].sum()
print(monthly_stats)
```

Пример:

```
month
1      1800436.55
2      1664398.87
3      1624308.68
4      1618523.23
5      1766374.15
6      1380494.94
7      1872053.27
8      1896153.37
9      1638384.93
10     1675578.88
11     1479732.02
12     1705205.24
```

Визуализация:

```
monthly_stats.plot(kind="bar", figsize=(12, 5), color="coral")
plt.title("Выручка по месяцам")
plt.ylabel("Выручка, у.е.")
plt.show()
```

*Изображение 3.5. Визуализация выручки по месяцам***Детализация: сочетание SQL и Pandas**

Иногда сезонность проще вычислить прямо в SQL. Например, подсчитать заказы по дню недели:

```
SELECT strftime('%w', order_date) AS weekday,
       COUNT(*) AS orders_count,
       SUM(amount) AS revenue
FROM orders
GROUP BY weekday
ORDER BY weekday;
```

Здесь `%w` возвращает день недели (0 = воскресенье, 6 = суббота).

3.5.3. Предсказание временных рядов

После того как мы научились агрегировать данные по времени, сглаживать их и замечать сезонность, встаёт практический вопрос: можем ли мы предсказать будущее? Прогнозирование временных рядов — это не "магия", а системный набор шагов: подготовка ряда, выбор базовой модели, проверка ошибок, сравнение альтернатив и, наконец, выпуск прогноза с интервалами неопределённости.

Мы будем работать с тем же датафреймом `df`, где есть столбцы `order_date` (дата) и `amount` (сумма заказа). Для наглядности агрегируем в месячную выручку и будем предсказывать следующие месяцы.

Готовим временной ряд

- Убеждаемся, что дата — это `datetime`.
- Агрегируем по месяцу.
- Превращаем `Period` в индекс типа `Timestamp` (удобно для моделей).

```
import pandas as pd
df["order_date"] = pd.to_datetime(df["order_date"])
monthly = (
    df.groupby(df["order_date"].dt.to_period("M"))["amount"]
    .sum()
    .sort_index()
)
ts = monthly.to_timestamp()
ts.head()
```

Если в данных есть пропуски (месяцы без продаж), их лучше явно заполнить нулями (или малым значением), иначе модели могут вести себя странно:

```
# Заполняем отсутствующие месяцы нулями (если ряд должен быть сплошным)
ts = ts.asfreq("MS", fill_value=0.0)
```

Разделяем на train/test по времени

Чтобы честно оценивать качество прогноза, будем имитировать будущее: `train` — всё до последнего года, `test` — последний год (12 месяцев). Так мы увидим, как модель предсказывает невиданные ранее месяцы.

```
# Возьмём последний год как тест
horizon = 12
train = ts.iloc[:-horizon]
test = ts.iloc[-horizon:]
train.tail(), test.head()
```

Временной ряд — зависимые наблюдения. Нельзя перемешивать строки, как в обычной кросс-валидации. Оценка "по времени" — стандарт.

Метрики качества прогноза

Мы будем использовать MAE (средняя абсолютная ошибка), RMSE (квадратичная ошибка) и MAPE (средняя абсолютная процентная ошибка). Они дают интуитивное понимание, "на сколько денег" или "на сколько процентов" мы промахиваемся.

```
import numpy as np
def mae(y_true, y_pred):
    return np.mean(np.abs(y_true - y_pred))
def rmse(y_true, y_pred):
    return np.sqrt(np.mean((y_true - y_pred)**2))
def mape(y_true, y_pred):
    return np.mean(np.abs((y_true - y_pred) / np.maximum(1e-9, y_true))) * 100
```

Очень простой базовый прогноз (бенчмарк)

Перед "умными" моделями всегда полезно построить наивный бенчмарк. Например, прогноз — это просто последнее наблюдение (или среднее последних k месяцев). Если "умная" модель едва лучше наивной — значит, мы зря усложняем.

```
# Сезонный наивный бенчмарк: значение в тот же месяц прошлого года
naive_seasonal = train.shift(12).reindex(test.index)
# Для самых первых точек в test может не хватить прошлого значения —
# заполним средним
naive_seasonal = naive_seasonal.fillna(train.mean())
print("Naive seasonal MAE:", mae(test.values, naive_seasonal.values))
```

```
print("Naive seasonal RMSE:", rmse(test.values, naive_seasonal.values))
print("Naive seasonal MAPE:", mape(test.values, naive_seasonal.values))
```

Экспоненциальное сглаживание

Идея: недавние наблюдения важнее старых, поэтому им даём больший вес. Базовая формула простого экспоненциального сглаживания:

$$\hat{y}_{t+1} = \alpha y_t + (1 - \alpha) \hat{y}_t$$

где:

- $\hat{y}_{t+1}$ — прогноз на следующий период;
- y_t — фактическое значение в момент t ;
- $\hat{y}_t$ — предсказанное значение для момента t ;
- α — параметр сглаживания ($0 < \alpha < 1$).

Простое экспоненциальное сглаживание (без тренда и сезонности)

Подходит, когда ряд без выраженного тренда и сезонности.

```
import matplotlib.pyplot as plt
from statsmodels.tsa.holtwinters import SimpleExpSmoothing
ses = SimpleExpSmoothing(train).fit(
    smoothing_level=0.3,
    optimized=True
)
ses_forecast = ses.forecast(horizon)
print("SES MAE:", mae(test.values, ses_forecast.values))
print("SES RMSE:", rmse(test.values, ses_forecast.values))
print("SES MAPE:", mape(test.values, ses_forecast.values))
```

Интерпретация: если в данных есть тренд/сезонность, SES будет отставать и давать смещённые прогнозы. Но как быстрый бенчмарк — отлично.

Holt (уровень + тренд)

Когда у ряда есть тенденция (рост/падение), используем двойное сглаживание (модель Холта). Оно добавляет компонент тренда и обновляет его на каждом шаге.

```
from statsmodels.tsa.holtwinters import ExponentialSmoothing
holt = ExponentialSmoothing(
    train,
    trend="add",
    seasonal=None
).fit(optimized=True)
holt_forecast = holt.forecast(horizon)
print("Holt MAE:", mae(test.values, holt_forecast.values))
print("Holt RMSE:", rmse(test.values, holt_forecast.values))
print("Holt MAPE:", mape(test.values, holt_forecast.values))
```

Holt – Winters (уровень + тренд + сезонность)

Если выражена и сезонность (например, месячная), берём тройное сглаживание — Holt – Winters. Есть два вида сезонности:

1. **Аддитивная** — сезонные колебания похожей амплитуды при разном уровне ряда.
2. **Мультипликативная** — сезонные колебания растут вместе с уровнем (амплитуда больше при больших значениях).

Для месячных данных обычно `seasonal_periods=12`.

```
hw = ExponentialSmoothing(
    train,
    trend="add",
    seasonal="add",
    seasonal_periods=12
).fit(optimized=True)
hw_forecast = hw.forecast(horizon)
print("HW MAE:", mae(test.values, hw_forecast.values))
print("HW RMSE:", rmse(test.values, hw_forecast.values))
print("HW MAPE:", mape(test.values, hw_forecast.values))
```

Как выбрать add/mul?

- Если сезонные колебания примерно одинаковые при низких и высоких значениях — берите `seasonal="add"`.
- Если при росте уровня амплитуда сезонных волн тоже растёт — `seasonal="mul"`.

ARIMA/ SARIMA

Семейство ARIMA учитывает три компонента:

1. AR (AutoRegressive, p) — зависимость ряда от собственных прошлых значений;
2. I (Integrated, d) — разности ряда (делаем ряд стационарным);
3. MA (Moving Average, q) — зависимость от прошлых ошибок прогноза.

Для сезонных рядов используется SARIMA: добавляются сезонные параметры (P, D, Q, m), где m — длина сезона (например, 12 для месяцев).

Для ARIMA важна стационарность (статистические свойства ряда — среднее и дисперсия — стабильны во времени). Визуально её сложно оценить, поэтому:

- смотрим на график и на разности (первую/вторую) — часто уже помогает;
- строим ACF/PACF — автокорреляции и частичные автокорреляции.

Наблюдения:

- Если у ряда тренд — сделайте разность: $d=1$.
- Если сильная сезонность — берите $m=12$ и, возможно, $D=1$.

SARIMA: пример подбора параметров вручную

Начнём с простого: $(p,d,q)=(1,1,1)$ и сезонная часть $(P,D,Q,m)=(1,1,1,12)$, затем можно менять.

```
from statsmodels.tsa.statespace.sarimax import SARIMAX
sarima = SARIMAX(
    train,
    order=(1,1,1),
```

```

seasonal_order=(1,1,1,12),
enforce_stationarity=False,
enforce_invertibility=False
).fit(dispatch=False)
sarima_forecast = sarima.get_forecast(steps=horizon)
pred_mean = sarima_forecast.predicted_mean
pred_int = sarima_forecast.conf_int(alpha=0.05) # 95% интервалы
print("SARIMA MAE:", mae(test.values, pred_mean.values))
print("SARIMA RMSE:", rmse(test.values, pred_mean.values))
print("SARIMA MAPE:", mape(test.values, pred_mean.values))

```

Плюсы SARIMA: гибкая, учитывает сезонность, даёт интервалы. Минусы: параметры нужно выбирать (вручную или с помощью автоподбора), возможна чувствительность к шуму.

Логарифмирование и стабилизация дисперсии

Если амплитуда колебаний растёт со временем, лог-трансформация часто стабилизирует дисперсию и делает задачу проще:

```

import numpy as np
train_log = np.log1p(train) # log(1+x)
test_log = np.log1p(test)
hw_log = ExponentialSmoothing(
    train_log, trend="add", seasonal="add", seasonal_periods=12
).fit(optimized=True)
f_log = hw_log.forecast(horizon)
f = np.exp(f_log)
print("HW(log) MAPE:", mape(test.values, f.values))

```

Как выбрать модель

1. Есть тренд и сезонность? — начните с Holt – Winters.
2. Нужны интервалы и контроль зависимостей? — SARIMA (или простой ARIMA, если сезонности нет).

3. Трудно подобрать параметры? — попробуйте Auto-ARIMA (pmdarima) или Prophet (бизнес-дружественный инструмент, хорошо работает с календарями/праздниками).
4. Очень сложные паттерны? — Machine Learning (градиентный) бустинг по признакам лагов и календаря либо нейросети (LSTM / Temporal Fusion Transformer). Но это уже отдельная история.

Полный рабочий пример: сравнение моделей и выбор лучшей

Соберём всё вместе: построим HW (add/mul) и SARIMA, сравним на тесте, выберем победителя, а потом перетренируем на всём ряде и сделаем прогноз на будущее.

```
import warnings
warnings.filterwarnings("ignore")
from statsmodels.tsa.holtwinters import ExponentialSmoothing
from statsmodels.tsa.statespace.sarimax import SARIMAX
horizon = 12
train = ts.iloc[:-horizon]
test = ts.iloc[-horizon:]
results = {}
hw_add = ExponentialSmoothing(
    train, trend="add", seasonal="add", seasonal_periods=12
).fit(optimized=True)
f_hw_add = hw_add.forecast(horizon)
results["HW_add_add"] = {
    "mae": mae(test, f_hw_add),
    "rmse": rmse(test, f_hw_add),
    "mape": mape(test, f_hw_add),
    "forecast": f_hw_add
}
hw_mul = ExponentialSmoothing(
    train, trend="add", seasonal="mul", seasonal_periods=12
).fit(optimized=True)
f_hw_mul = hw_mul.forecast(horizon)
results["HW_add_mul"] = {
```



```

    "mae": mae(test, f_hw_mul),
    "rmse": rmse(test, f_hw_mul),
    "mape": mape(test, f_hw_mul),
    "forecast": f_hw_mul
}
sarima = SARIMAX(
    train, order=(1,1,1), seasonal_order=(1,1,1,12),
    enforce_stationarity=False, enforce_invertibility=False
).fit(dispatch=False)
sf = sarima.get_forecast(steps=horizon)
f_sarima = sf.predicted_mean
results["SARIMA_111_111_12"] = {
    "mae": mae(test, f_sarima),
    "rmse": rmse(test, f_sarima),
    "mape": mape(test, f_sarima),
    "forecast": f_sarima,
    "conf_int": sf.conf_int(alpha=0.05)
}
for name, r in results.items():
    print(f"{name:20s} MAE={r['mae']:.0f} RMSE={r['rmse']:.0f} MAPE={r['mape']:.2f}%")
best_name = min(results, key=lambda k: results[k]["mape"])
best = results[best_name]
print("\nЛучшая модель:", best_name)

```

Что дальше? Перетренируем лучшую модель на всём ряде (train + test) и предскажем будущее, например, ещё на 6–12 месяцев.

```

final_model = ExponentialSmoothing(
    ts, trend="add", seasonal="mul", seasonal_periods=12
).fit(optimized=True)
future_horizon = 12
future_forecast = final_model.forecast(future_horizon)

```

3.5.4. Итоги раздела

- ◆ Мы познакомились с основами анализа временных рядов: от агрегации в SQL до гибкой работы в Pandas.
- ◆ Научились строить ежедневные и месячные динамики, находить тренды и сезонность.
- ◆ Освоили методы сглаживания: простое и экспоненциальное, Holt – Winters.
- ◆ Попробовали прогнозирование: от простого экспоненциального сглаживания до ARIMA.
- ◆ Узнали о других инструментах — Prophet, Auto-ARIMA, ML-модели, LSTM — и поняли, когда их применять.

3.6. Визуализация отчетов

В Python существует несколько библиотек для визуализации. Мы рассмотрим три основных:

1. **matplotlib** — низкоуровневая библиотека для любых типов графиков;
2. **seaborn** — надстройка над **matplotlib**, упрощающая создание статистических графиков;
3. **Plotly** — интерактивные графики, которые можно крутить и использовать в дашбордах.

3.6.1. Matplotlib

Matplotlib — самая популярная библиотека для визуализации в Python. Она позволяет строить простые и сложные графики: линии, гистограммы, круговые диаграммы, боксплоты, тепловые карты.

Особенности библиотеки:

- огромные возможности (от простого `line plot` до 3D-графиков);

- гибкость в настройке: можно менять шрифты, цвета, размеры, сетку;
- используются `seaborn` и `pandas` "под капотом" для рисования графиков.

Давайте построим самую простую линию.

```
import matplotlib.pyplot as plt
x = [1, 2, 3, 4, 5]
y = [2, 4, 5, 7, 11]
plt.plot(x, y)
plt.title("Простейший график")
plt.xlabel("X")
plt.ylabel("Y")
plt.grid()
plt.show()
```



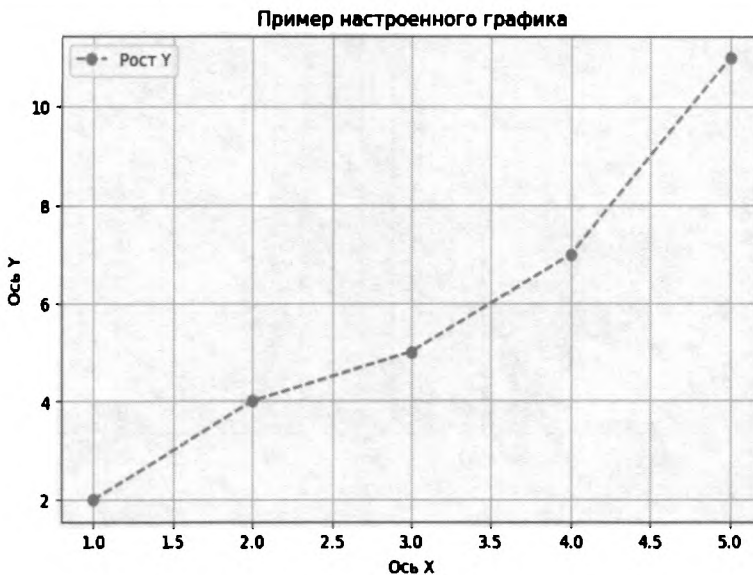
Изображение 3.6. Простейший график

Что мы видим:

- по оси X — числа от 1 до 5,
- по оси Y — значения функции,
- прямая линия показывает зависимость.

Добавляем стиль и оформление

```
plt.figure(figsize=(8, 5)) # Размер картинки
plt.plot(x, y, marker="o", linestyle="--", color="red", label="Рост Y")
plt.title("Пример настроенного графика")
plt.xlabel("Ось X")
plt.ylabel("Ось Y")
plt.legend()
plt.grid(True)
plt.show()
```



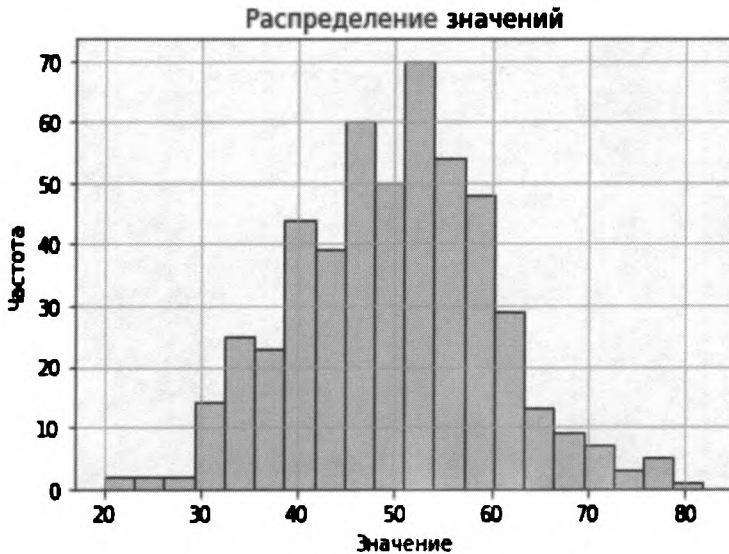
Изображение 3.7. Кастомизированный график

Теперь у нас:

- красная пунктирная линия;
- точки отмечены кружками;
- есть легенда и сетка;
- удобный размер картинки.

Гистограмма показывает распределение данных. Это полезно, когда нужно понять, какие значения встречаются чаще.

```
import numpy as np
data = np.random.normal(50, 10, 500)
plt.hist(data, bins=20, color="skyblue", edgecolor="black")
plt.title("Распределение значений")
plt.xlabel("Значение")
plt.ylabel("Частота")
plt.grid()
plt.show()
```



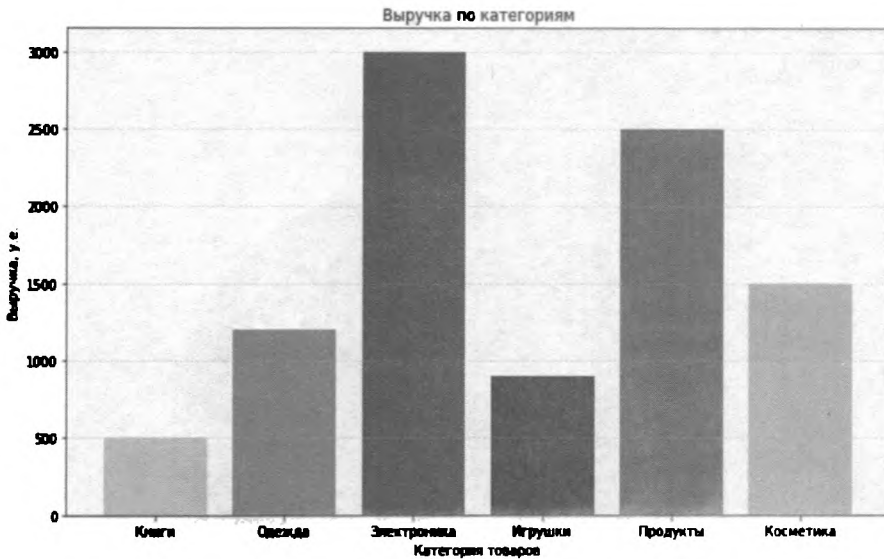
Изображение 3.8. Гистограмма

На графике видно, что большинство значений сосредоточено около 50, а хвосты уходят влево и вправо.

Столбчатые диаграммы используются для сравнения категорий.

```
import matplotlib.pyplot as plt
categories = ["Книги", "Одежда", "Электроника", "Игрушки", "Продукты",
"Косметика"]
values = [500, 1200, 3000, 900, 2500, 1500]
plt.figure(figsize=(10, 6))
colors = ["#B3CDE0", "#6497B1", "#005B96", "#03396C", "#336B87", "#88BBD6"]
```

```
bars = plt.bar(categories, values, color=colors)
plt.title("Выручка по категориям")
plt.ylabel("Выручка, у.е.")
plt.xlabel("Категория товаров")
plt.grid(axis="y", linestyle="--", alpha=0.5)
plt.tight_layout()
plt.show()
```



Изображение 3.9. Столбчатые диаграммы

Такой график сразу показывает, какая категория приносит больше всего денег.

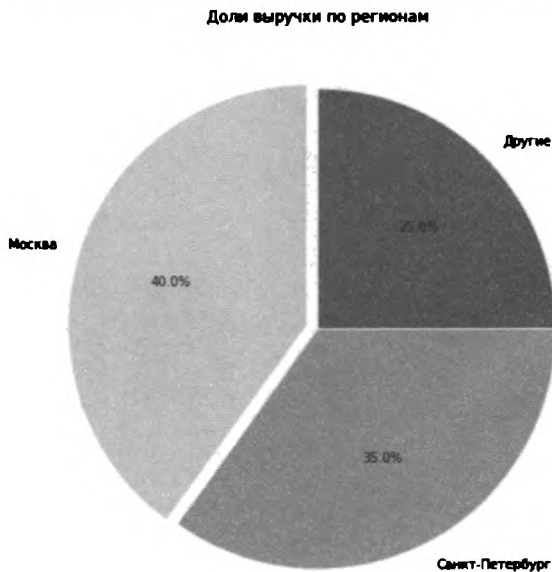
Круговые диаграммы хорошо подходят для показа долей.

```
import matplotlib.pyplot as plt
shares = [40, 35, 25]
labels = ["Москва", "Санкт-Петербург", "Другие"]
colors = ["#c6dbef", "#6baed6", "#2171b5"] # Голубые и синие тона
plt.figure(figsize=(7, 7))
wedges, texts, autotexts = plt.pie(
    shares,
    labels=labels,
```

```

    colors=colors,
    autopct="%1.1f%%",
    startangle=90,
    explode=explode,
    wedgeprops={"edgecolor": "white"}
)
plt.title("Доли выручки по регионам", fontsize=14,
fontweight="bold")
plt.tight_layout()
plt.show()

```



Изображение 3.10. Круговая диаграмма

Мы видим, что Москва приносит 40% выручки, Петербург — 35%, остальные регионы — 25%.

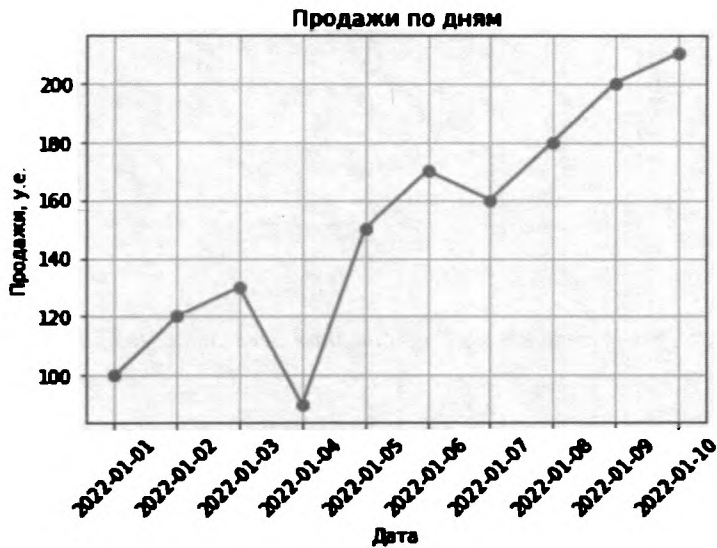
Matplotlib особенно полезен для работы с временными рядами.

```

import pandas as pd
import datetime
dates = pd.date_range(start="2022-01-01", periods=10)
sales = [100, 120, 130, 90, 150, 170, 160, 180, 200, 210]
plt.plot(dates, sales, marker="o")

```

```
plt.title("Продажи по дням")
plt.xlabel("Дата")
plt.ylabel("Продажи, у.е.")
plt.xticks(rotation=45)
plt.grid(True)
plt.show()
```

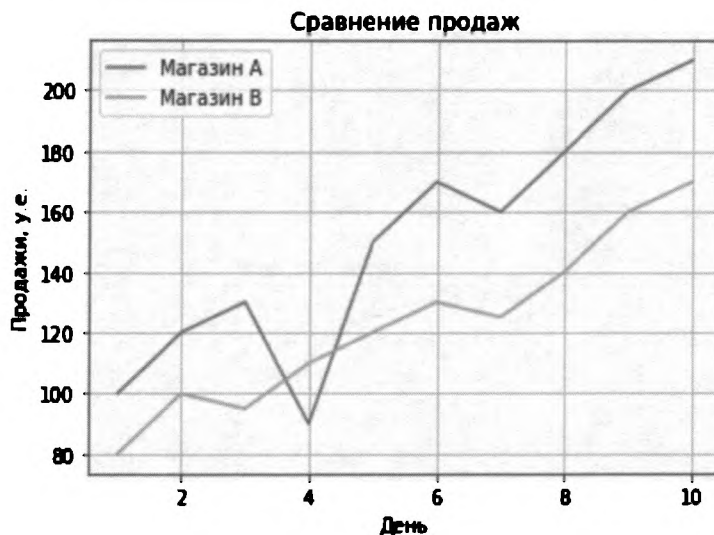


Изображение 3.11. Визуализация временного ряда

На графике видно, что продажи растут, но есть падение на 4-й день.

Иногда нужно сравнить несколько рядов.

```
days = range(1, 11)
sales_a = [100, 120, 130, 90, 150, 170, 160, 180, 200, 210]
sales_b = [80, 100, 95, 110, 120, 130, 125, 140, 160, 170]
plt.plot(days, sales_a, label="Магазин А")
plt.plot(days, sales_b, label="Магазин В")
plt.title("Сравнение продаж")
plt.xlabel("День")
plt.ylabel("Продажи, у.е.")
plt.legend()
plt.show()
```



Изображение 3.12. Визуализация нескольких графиков

3.6.2. Seaborn

Seaborn — это библиотека визуализации на базе `matplotlib`, которая делает статистические графики проще, чище и выразительнее "из коробки".

Её сильные стороны:

- тесная интеграция с `pandas`: многие функции принимают `DataFrame` и имена колонок;
- "умные" дефолтные стили и палитры;
- готовые высокоуровневые графики для распределений, сравнения категорий, корреляций, регрессий;
- удобные средства для фасетирования (разбивки на панели по категориям).

Практический принцип: если вам нужен быстрый, красивый и "статистический" график — начинайте с `seaborn`; для тонкой кастомизации — подключайте приёмы `matplotlib`.

Зададим для начала общий стиль и палитру графиков.

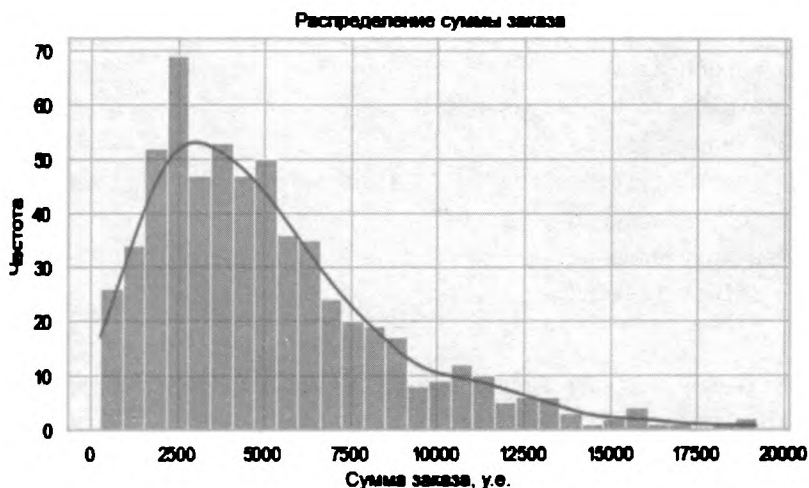
```
import seaborn as sns
import matplotlib.pyplot as plt
# Общий стиль графиков
sns.set_theme(style="whitegrid") # Варианты: "white", "dark",
"whitegrid", "darkgrid", "ticks"
# Палитра по умолчанию
sns.set_palette("deep")
```

Допустим, у нас есть таблица заказов интернет-магазина. Сгенерируем компактный датафрейм (если у вас есть реальный `orders` — подставьте его).

```
import pandas as pd
import numpy as np
np.random.seed(42)
N = 600
df = pd.DataFrame({
# Суммы заказов
    "amount": np.random.gamma(shape=2.0, scale=2500, size=N),
    "category": np.random.choice(["Книги", "Одежда", "Электроника"],
size=N, p=[0.35, 0.4, 0.25]),
    "city": np.random.choice(["Москва", "СПб", "Другие"], size=N,
p=[0.5, 0.3, 0.2]),
# Была ли скидка
    "discount": np.random.choice([0, 1], size=N, p=[0.7, 0.3]),
# Число визитов
    "visits": np.random.poisson(lam=5, size=N)
})
df["amount"] = df["amount"].clip(300, 100000).round(0)
```

Начнем с гистограммы.

```
plt.figure(figsize=(8, 4))
sns.histplot(data=df, x="amount", bins=30, kde=True)
plt.title("Распределение суммы заказа")
plt.xlabel("Сумма заказа, y.e.")
plt.ylabel("Частота")
plt.show()
```



Изображение 3.13. Гистограмма в seaborn

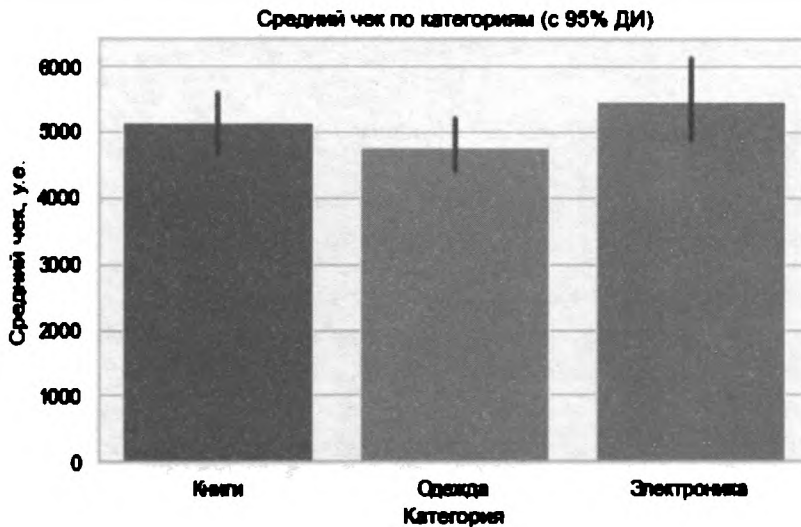
- `histplot` строит гистограмму; `kde=True` добавляет оценку плотности (плавную кривую).
- Быстро видно правый "хвост" (редкие крупные заказы).

```
g = sns.displot(
    data=df, x="amount", col="category",
    bins=30, kde=True, col_wrap=3, height=3, aspect=1.2
)
g.set_axis_labels("Сумма заказа, руб.", "Частота")
g.fig.suptitle("Распределение суммы заказа по категориям", y=1.03)
plt.show()
```

Изображение 3.14. Применение `displot`

Средние по категориям будем визуализировать с помощью `barplot`.

```
plt.figure(figsize=(7, 4))
sns.barplot(data=df, x="category", y="amount", estimator=np.mean,
ci=95)
plt.title("Средний чек по категориям (с 95% ДИ)")
plt.xlabel("Категория")
plt.ylabel("Средний чек, у.е.")
plt.show()
```

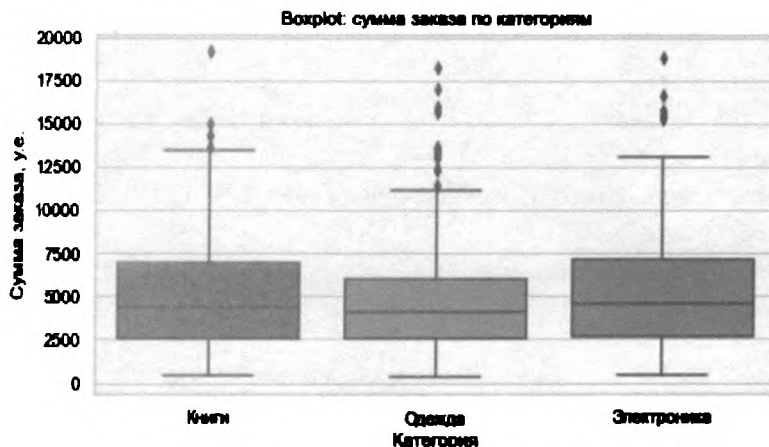


Изображение 3.15. Применение `barplot`

- По умолчанию `seaborn` рисует доверительные интервалы (можно отключить `ci=None`).

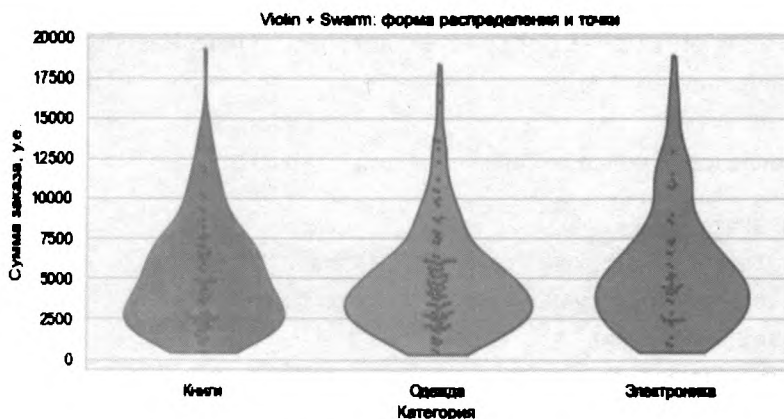
Распределение по категориям — метод `boxplot`.

```
plt.figure(figsize=(8, 4))
sns.boxplot(data=df, x="category", y="amount")
plt.title("Boxplot: сумма заказа по категориям")
plt.xlabel("Категория")
plt.ylabel("Сумма заказа, у.е.")
plt.show()
```

Изображение 3.16. Применение *boxplot*

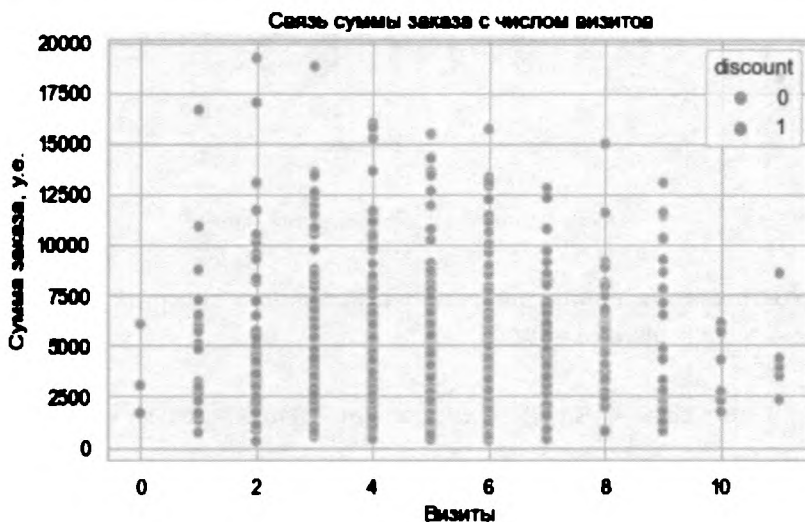
- *Boxplot* показывает медиану, квартильный размах и выбросы.
- Хорош для сравнения разброса и медианы между группами.

```
plt.figure(figsize=(9, 4))
sns.violinplot(data=df, x="category", y="amount", inner=None, cut=0)
sns.swarmplot(data=df.sample(200), x="category", y="amount",
              color="k", alpha=0.5, size=3)
plt.title("Violin + Swarm: форма распределения и точки")
plt.xlabel("Категория")
plt.ylabel("Сумма заказа, у.е.")
plt.show()
```

Изображение 3.17. Применение *violinplot*

- Violinplot показывает полную форму распределения.
- Swarm добавляет точки на "скрипку", делая график нагляднее (используйте с сэмплом, чтобы не перегружать).

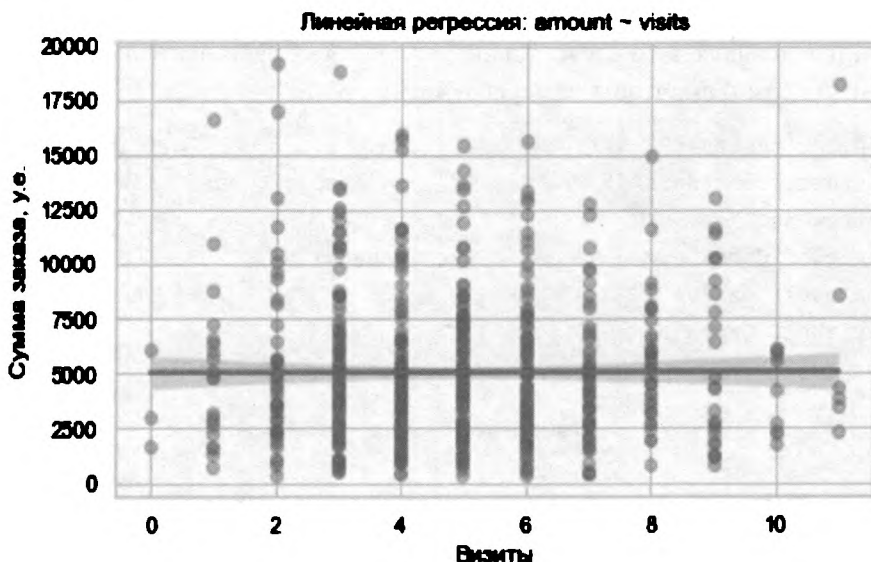
```
plt.figure(figsize=(7, 4))
sns.scatterplot(data=df, x="visits", y="amount", hue="discount",
               palette="Set2")
plt.title("Связь суммы заказа с числом визитов")
plt.xlabel("Визиты")
plt.ylabel("Сумма заказа, у.е.")
plt.show()
```



Изображение 3.18. Применение *scatterplot*

- Лёгкая проверка: растут ли чеки с ростом визитов? Отличаются ли группы со скидкой / без скидки?

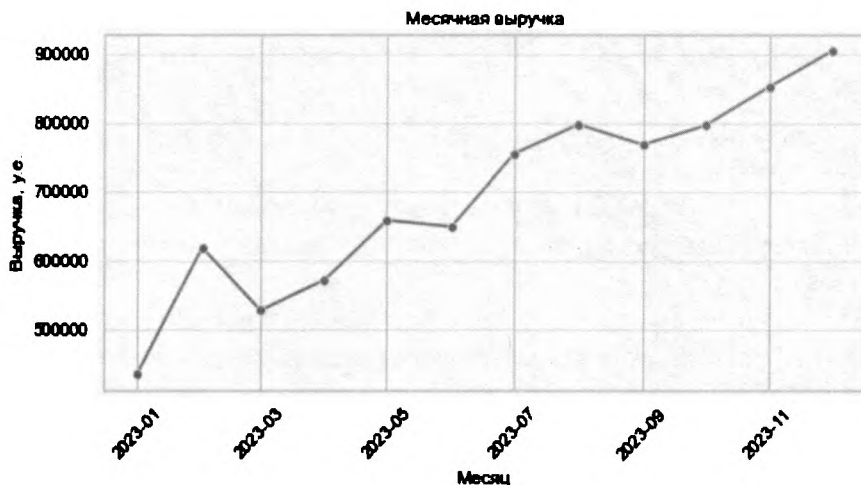
```
plt.figure(figsize=(7, 4))
sns.regplot(data=df, x="visits", y="amount", scatter_kws={"alpha":
0.4}, line_kws={"color": "red"})
plt.title("Линейная регрессия: amount ~ visits")
plt.xlabel("Визиты")
plt.ylabel("Сумма заказа, у.е.")
plt.show()
```

Изображение 3.19. Применение `regplot`

- `regplot` строит линию линейной регрессии с доверительной лентой (можно выключить `ci=None`).

Если у вас есть временной индекс или колонка даты — используйте `lineplot`.

```
dates = pd.date_range("2023-01-01", periods=12, freq="MS")
revenue = (np.linspace(500_000, 900_000, 12) + np.random.normal(0,
40_000, 12)).round(0)
ts = pd.DataFrame({"date": dates, "revenue": revenue})
plt.figure(figsize=(9, 4))
sns.lineplot(data=ts, x="date", y="revenue", marker="o")
plt.title("Месячная выручка")
plt.xlabel("Месяц")
plt.ylabel("Выручка, у.е.")
plt.grid(True, axis="y", alpha=0.3)
plt.xticks(rotation=45)
plt.show()
```

Изображение 3.20. Применение *lineplot*

Сохранение:

```
fig = plt.gcf()
fig.savefig("report_chart.png", dpi=150, bbox_inches="tight")
```

3.6.3. Plotly

Если `matplotlib` и `seaborn` хороши для отчётов и статических графиков, то `Plotly` делает следующий шаг:

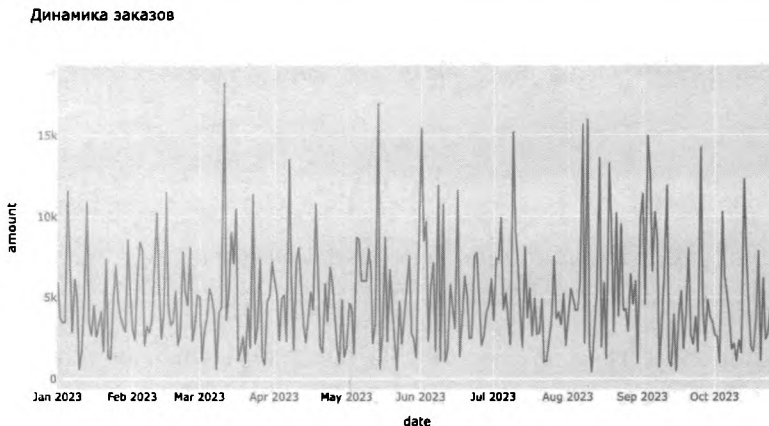
- интерактивные графики (можно наводить мышкой и видеть значения, масштабировать, скрывать/показывать серии);
- совместимость с Jupyter Notebook и веб-приложениями;
- возможность сохранять графики в HTML и вставлять их в дашборды;
- поддержка 2D- и 3D-визуализаций.

Plotly особенно полезен, если вы хотите, чтобы ваш график был не просто картинкой в отчёте, а интерактивным инструментом анализа.


```
import plotly.express as px
import pandas as pd
import numpy as np
# Данные
np.random.seed(42)
df = pd.DataFrame({
    "amount": np.random.gamma(2.0, 2500, 300).round(0),
    "category": np.random.choice(["Книги", "Одежда", "Электроника"],
size=300),
    "city": np.random.choice(["Москва", "СПб", "Другие"], size=300),
    "date": pd.date_range("2023-01-01", periods=300, freq="D")
})
```

Начнем с линейного графика.

```
fig = px.line(df, x="date", y="amount", title="Динамика заказов")
fig.show()
```



Изображение 3.21. Интерактивный график plotly

Особенность Plotly: график интерактивный.

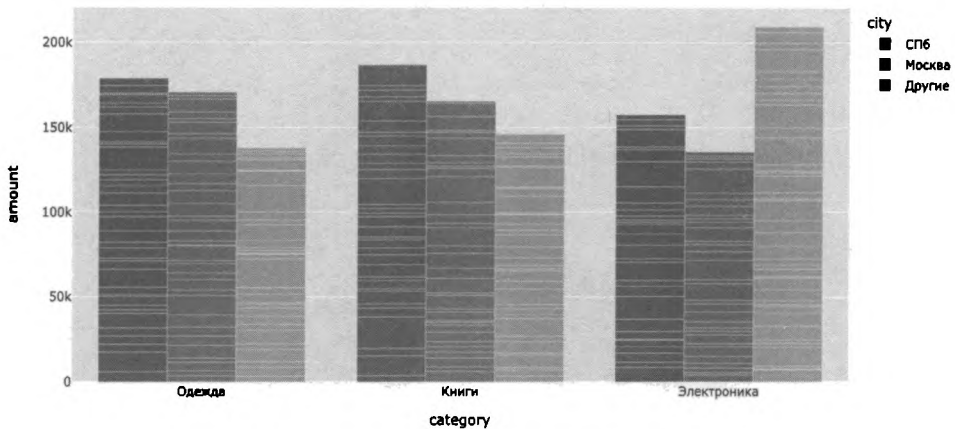
Можно:

- наводить мышкой и видеть значение для конкретной даты;
- масштабировать выделением;
- включать/выключать серии в легенде.

Столбчатая диаграмма — следующая визуализация:

```
fig = px.bar(df, x="category", y="amount", color="city",
             title="Выручка по категориям и городам", barmode="group")
fig.show()
```

Выручка по категориям и городам



Изображение 3.22. Применение `px.bar`

- `barmode="group"` — столбцы рядом.
- Можно использовать `barmode="stack"`, чтобы построить накопительную диаграмму.

Круговая диаграмма:

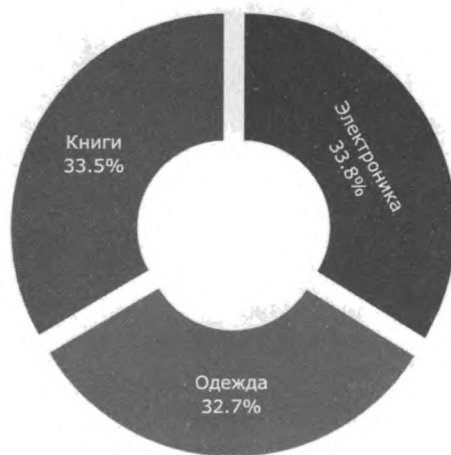
```
import plotly.express as px
category_share = df.groupby("category")["amount"].sum().reset_index()
fig = px.pie(
    category_share,
    names="category",
    values="amount",
    title="Доли выручки по категориям",
    hole=0.4, # Делаем donut chart
    color_discrete_sequence=px.colors.sequential.Blues_r)
fig.update_traces(
```

```

textinfo="label+percent",
pull=[0.05] * len(category_share),
marker=dict(line=dict(color="white", width=2))
)
fig.update_layout(
    title_x=0.5, # Центрируем заголовок
    font=dict(size=14),
    showlegend=False # Скрываем легенду
)
fig.show()

```

Доли выручки по категориям



Изображение 3.23. Визуализация круговой диаграммы

Наводя курсор на сегмент, сразу увидим процент и абсолютное значение.

Тепловая карта (Heatmap):

```

import plotly.express as px
heatmap_data = (
    df.groupby([df["date"].dt.month, "category"])["amount"]
        .sum()
        .reset_index()
        .rename(columns={"date": "month"})
)

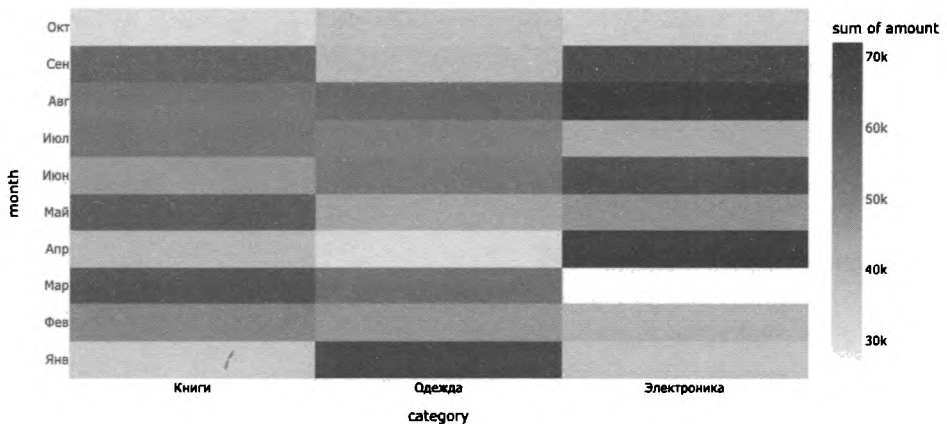
```

```

)
months_map = {
    1: "Янв", 2: "Фев", 3: "Мар", 4: "Апр", 5: "Май", 6: "Июн",
    7: "Июл", 8: "Авг", 9: "Сен", 10: "Окт", 11: "Ноя", 12: "Дек"
}
heatmap_data["month"] = heatmap_data["month"].map(months_map)
fig = px.density_heatmap(
    heatmap_data,
    x="category",
    y="month",
    z="amount",
    color_continuous_scale="Blues",
    title="Тепловая карта: выручка по категориям и месяцам"
)
fig.show()

```

Тепловая карта: выручка по категориям и месяцам



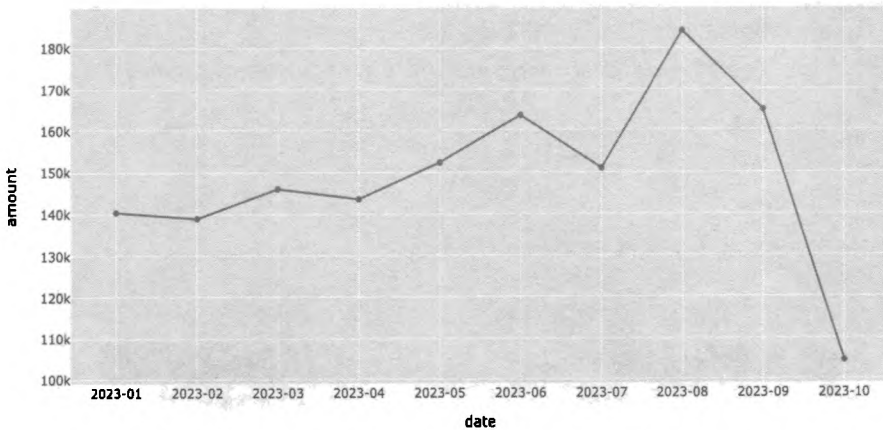
Изображение 3.24. Тепловая карта

Отличный способ показать распределение значений по двум категориям (месяц × категория).

Временные ряды — интерактивный пример:

```
monthly =
df.groupby(df["date"].dt.to_period("M"))["amount"].sum().reset_index()
monthly["date"] = monthly["date"].astype(str)
fig = px.line(monthly, x="date", y="amount", markers=True,
              title="Интерактивная динамика выручки по месяцам")
fig.update_xaxes(type="category")
fig.show()
```

Интерактивная динамика выручки по месяцам

*Изображение 3.25. Интерактивный временной ряд*

Здесь можно:

- крутить график по шкале времени;
- выбирать интересные диапазоны;
- смотреть всплески в конкретных месяцах.

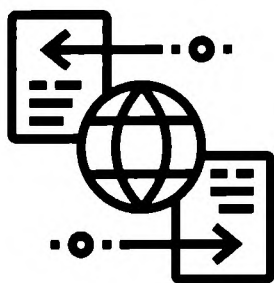
Графики Plotly можно сохранить в HTML и отправить коллеге:

```
fig.write_html("report_chart.html")
```

Файл можно открыть в браузере — график останется интерактивным.

3.6.4. Итоги раздела

- ◆ Освоили `matplotlib` — фундаментальную библиотеку для построения любых графиков: линии, гистограммы, столбцы, круговые диаграммы, временные ряды.
- ◆ Познакомились с `seaborn`, который облегчает создание статистических графиков и тесно интегрируется с `pandas`. Использовали `barplot`, `boxplot`, `violin`, `scatter`, `heatmap` и фасеты для сегментации данных.
- ◆ Научились строить интерактивные графики в `Plotly`: линии, столбцы, `scatter`, `boxplot` и тепловые карты. Поняли, как сохранять их в HTML и использовать в дашбордах.
- ◆ Теперь у нас есть полный набор инструментов визуализации:
 - `matplotlib` для гибкости,
 - `seaborn` для статистики,
 - `Plotly` для интерактивности.



Часть IV.

Практические проекты

4.1. Аналитика интернет-магазина

В современном онлайн-бизнесе важно понимать поведение клиентов на основе данных. Для этого применяются специальные техники анализа:

- RFM-анализ,
- анализ удержания (retention)
- и оценка Lifetime Value (LTV) клиентов.

В этой главе мы на практике рассмотрим, как с помощью Python построить отчеты по данным интернет-магазина, провести сегментацию клиентов и получить бизнес-инсайты.

RFM-анализ позволяет разделить клиентов на группы по трем показателям:

1. **Recency** (давность) — насколько давно клиент совершил последнюю покупку (чем меньше времени прошло, тем лучше).
2. **Frequency** (частота) — как часто клиент покупает (более частые покупки означают более лояльного клиента).
3. **Monetary** (суммарная ценность) — сколько денег клиент потратил за определенный период (большие суммы говорят о ценных клиентах).

Анализ удержания (retention) с помощью когорт помогает понять, какая доля клиентов возвращается за повторными покупками со временем.

Мы сгруппируем клиентов по периодам их первой покупки (когортам) и отследим, какой процент из них совершает покупки в последующие месяцы.

Lifetime Value (LTV) — это совокупная прибыль или выручка от клиента за все время сотрудничества. Расчет LTV позволяет понять, насколько ценен клиент для бизнеса и сколько можно вложить в его привлечение или удержание.

Мы вычислим LTV на основе транзакций и рассмотрим распределение и средние значения LTV по сегментам клиентов.

В этой главе мы сгенерируем примерный набор данных интернет-магазина (список заказов с указанием клиента, даты и суммы) и шаг за шагом проведем анализ. Будут приведены фрагменты кода на Python для расчетов и визуализации результатов. Такой практический проект объединит навыки работы с данными, полученные ранее, и покажет, как получать инсайты с бизнес-фокусом.

Генерация данных

Для начала создадим искусственный набор данных о заказах онлайн-магазина. Предположим, что у нас есть 1000 клиентов, делавших покупки в течение двух лет. Каждый заказ характеризуется идентификатором клиента, датой заказа и суммой заказа. Чтобы упростить, сгенерируем данные случайным образом, имитируя реальную ситуацию:

- **Даты заказов:** будем генерировать первый заказ каждого клиента в случайном месяце 2021 года. Затем для имитации повторных покупок зададим, что каждый последующий месяц клиент может совершить еще одну покупку с определенной вероятностью (моделируя вероятность удержаться). Таким образом, получим разные паттерны: некоторые клиенты купят только один раз, другие — многократно.
- **Суммы заказов:** сгенерируем случайные суммы (в условных единицах) для каждого заказа, распределенные около некоторого среднего значения (например, 100). Это позволит получить как мелкие заказы, так и более крупные, имитируя реальное распределение чека.

Будем использовать библиотеки `pandas` и `numpy` для удобства работы с таблицами и генерации случайных значений.

Листинг 4.1. Подготовка данных

```

import pandas as pd
import numpy as np
import random
from datetime import datetime
# Фиксируем сид для воспроизводимости
random.seed(42)
np.random.seed(42)
# Параметры симуляции
num_customers = 1000          # Число клиентов
churn_prob = 0.4              # Вероятность "оттока" клиента каждый
                              # месяц (т.е. 40%-й шанс, что клиент больше не вернется в следующем
                              # месяце)
orders = [] # Список для хранения сгенерированных заказов
for customer_id in range(1, num_customers+1):
    # Первая покупка: случайный месяц 2021 года
    first_month = random.randint(1, 12)
    first_year = 2021
    first_day = random.randint(1, 28) # Выбираем день месяца (1-28
    # для простоты)
    first_date = datetime(first_year, first_month, first_day)
    # Добавляем первую покупку
    amount = max(5, round(random.gauss(100, 50), 2)) # сумма (руб)
    # не меньше 5, в среднем ~100, с разбросом
    orders.append({"CustomerID": customer_id, "OrderDate": first_date,
    "Amount": amount})
    # Генерируем дальнейшие покупки ежемесячно до "оттока"
    current_date = first_date
    # Определим, сколько месяцев подряд клиент будет совершать
    # покупки (включая первый)
    # Смоделируем это число из геометрического распределения с
    # параметром churn_prob
    months_active = np.random.geometric(churn_prob)
    for month_offset in range(1, months_active):
        # Вычисляем следующий месяц покупки
        year = first_year + (first_month - 1 + month_offset) // 12
        month = (first_month - 1 + month_offset) % 12 + 1
        if year > 2022: # Ограничим период 2021-2022 годами

```

```

        break
    # Сгенерируем дату заказа в этом месяце
    day = random.randint(1, 28)
    order_date = datetime(year, month, day)
    # Добавляем как минимум один заказ в этом месяце
    amount = max(5, round(random.gauss(100, 50), 2))
    orders.append({"CustomerID": customer_id, "OrderDate":
order_date, "Amount": amount})
    # Дополнительно, с небольшой вероятностью, клиент может
сделать 2-3 покупки в том же месяце
    if random.random() < 0.15: # 15% – шанс второй покупки в том
же месяце
        day2 = random.randint(1, 28)
        amount2 = max(5, round(random.gauss(100, 50), 2))
        orders.append({"CustomerID": customer_id, "OrderDate":
datetime(year, month, day2), "Amount": amount2})
        if random.random() < 0.10: # 10% – шанс третьей покупки
            day3 = random.randint(1, 28)
            amount3 = max(5, round(random.gauss(100, 50), 2))
            orders.append({"CustomerID": customer_id,
"OrderDate": datetime(year, month, day3), "Amount": amount3})
    # Формируем DataFrame pandas из списка заказов
    orders_df = pd.DataFrame(orders)
    # Преобразуем OrderDate в тип datetime и сортируем заказы по дате
    orders_df['OrderDate'] = pd.to_datetime(orders_df['OrderDate'])
    orders_df = orders_df.sort_values('OrderDate').reset_
index(drop=True)
    print(orders_df.head(5))

```

Запустив этот код, мы получим таблицу заказов `orders_df` примерно следующего вида:

	CustomerID	OrderDate	Amount
0	863	2021-01-01	49.02
1	497	2021-01-01	146.06
2	237	2021-01-01	46.47
3	848	2021-01-02	5.00
4	986	2021-01-03	26.29
...			

Каждая строка — отдельный заказ. Например, клиент 863 совершил заказ 2021-01-01 на сумму 49.02. Клиент 497 также сделал заказ в первый же день года на сумму 146.06. Данные охватывают период с начала 2021 до конца 2022 года.

Теперь, имея на руках таблицу заказов `orders_df`, мы можем приступить к проведению аналитики.

4.1.1. RFM-анализ и сегментация клиентов

Цель RFM-анализа — на основе истории покупок каждого клиента вычислить ключевые метрики (Recency, Frequency, Monetary) и на их основе сегментировать клиентов. Это позволит выявить группы, на которые стоит по-разному ориентировать маркетинговые усилия: например, выделить лучших клиентов, "засыпающих" клиентов, новых клиентов и т.д.

Для расчета RFM-метрик нам понадобится агрегировать данные по каждому клиенту:

- **Recency** (давность) — число дней с момента последней покупки клиента до определенной даты анализа. Обычно датой анализа является конец периода данных. В наших данных максимальная дата заказа – 22 декабря 2022, возьмем следующий день (2022-12-23) как точку отсчета. Если клиент совершил последнюю покупку вчера, Recency = 1 день; если год назад – ~365 дней и т.д.
- **Frequency** (частота) — общее количество заказов, совершенных клиентом за рассматриваемый период (2021–2022 гг.).
- **Monetary** (денежная ценность) — общая сумма денег, которую клиент потратил за период.

Воспользуемся возможностями `pandas` для группировки данных по `CustomerID` и расчета этих показателей.

Листинг 4.2. Расчет RFM

```
import datetime as dt
# Вычисляем Recency: находим дату последней покупки каждого клиента
last_purchase =
```

```

orders_df.groupby('CustomerID')['OrderDate'].max().reset_index()
last_purchase.columns = ['CustomerID', 'LastPurchaseDate']
analysis_date = orders_df['OrderDate'].max() + pd.Timedelta(days=1)
# Добавляем столбец Recency (разница в днях между датой анализа и
# датой последней покупки)
last_purchase['Recency'] = (analysis_date - last_purchase
['LastPurchaseDate']).dt.days
# Frequency: считаем количество заказов у каждого клиента
frequency =
orders_df.groupby('CustomerID')['OrderDate'].count().reset_index()
frequency.columns = ['CustomerID', 'Frequency']
# Monetary: суммарная выручка от каждого клиента
monetary =
orders_df.groupby('CustomerID')['Amount'].sum().reset_index()
monetary.columns = ['CustomerID', 'Monetary']
# Объединяем все метрики в одну таблицу RFM
rfm = last_purchase.merge(frequency, on='CustomerID').
merge(monetary, on='CustomerID')
print(rfm.head(5))

```

После выполнения этого кода мы получим таблицу `rfm` с колонками:

- `CustomerID` — идентификатор клиента;
- `LastPurchaseDate` — дата последней покупки;
- `Recency` — давность последней покупки (в днях);
- `Frequency` — количество покупок;
- `Monetary` — общая сумма покупок.

Пример первых строк таблицы RFM:

CustomerID	LastPurchaseDate	Recency	Frequency	Monetary
1	2021-11-09	409	2	160.36
2	2021-03-25	638	2	317.33
3	2021-02-19	672	1	70.96
4	2021-01-03	719	1	73.74
5	2021-09-20	459	2	152.52
...	...	...	...	...

Здесь, например, для клиента 1 $\text{Recency} = 409$ дней (последняя покупка была 9 ноября 2021, что примерно 409 дней до 23.12.2022), $\text{Frequency} = 2$ (две покупки за период), $\text{Monetary} = 160.36$ (в сумме потратил 160.36 у.е.). У клиента 4 только 1 покупка ($\text{Frequency} = 1$), и она была очень давно ($\text{Recency} = 719$ дней, в январе 2021).

Присвоение RFM-оценок и сегментация

Теперь переведем полученные числовые метрики в оценки по шкале. Чаще всего используют систему баллов от 1 до 5 для каждой метрики, где 5 – наилучший показатель. То есть:

- Клиенты с наименьшим Recency (т.е. покупали совсем недавно) получают $R = 5$, а с самым большим Recency (давно не покупали) — $R = 1$.
- Клиенты с наибольшим Frequency (часто покупают) получают $F = 5$, а однократные покупатели – $F = 1$.
- Клиенты с высоким Monetary (потратили много денег) получают $M = 5$, а с низким — $M = 1$.

Таким образом, каждому клиенту сопоставляется RFM-профиль, например $R = 5, F = 5, M = 5$ у самых ценных и активных или $R = 1, F = 1, M = 1$ у тех, кто давно купил один раз на небольшую сумму.

Для присвоения баллов можно использовать разбиение метрик на квинтили (20-процентные группы). Однако, если распределение метрики сильно неравномерное (как у Frequency , где ~36% клиентов имеют значение 1), возможны ситуации с неуникальными границами квантилей. В таких случаях логично скорректировать разбиение вручную.

Мы реализуем следующую стратегию:

- **RecencyScore**: используем `pd.qcut` для деления Recency на 5 групп по квинтилям и назначим метки 5, 4, 3, 2, 1 (5 — самым "свежим" клиентам).
- **FrequencyScore**: если `qcut` не сможет разбить на 5 групп из-за большого количества одинаковых значений, зададим ручные пороги: например, частота 1 — это $F = 1$, 2 — $F = 2$, 3 — $F = 3$, 4 — $F = 4$, а 5 или более — $F = 5$.
- **MonetaryScore**: аналогично Recency , разобьем на 5 квантилей (большинство сумм уникальны, поэтому проблем не возникнет).

После этого можем объединить три балла (R_score , F_score , M_score) для определения сегмента:

- Отдельно выделим клиентов, у которых $RecencyScore = 1$ — то есть "неактивные" или почти потерянные клиенты (давно не заходили за покупками).
- Остальных разобьем по суммарному RFM-скорингу (сумме трех баллов):
 - » Лучшие клиенты: суммарный балл ≥ 13 (из 15), то есть очень высокие показатели по всем метрикам.
 - » Важные клиенты: суммарный балл ≥ 10 (но меньше 13) — хорошие клиенты, чуть уступают лучшим.
 - » Обычные клиенты: суммарный балл ≥ 6 (но меньше 10) — средние показатели.
 - » Малоценные клиенты: остальные (сумма < 6) — низкие показатели (кроме тех, кто уже помечен как неактивные по $Recency$).

Такая сегментация условна и примерна. В реальных задачах бизнес сам определяет границы сегментов исходя из стратегий: например, можно выделять сегмент "новые клиенты" ($R = 5$, $F = 1$), "наиболее лояльные" ($F = 5$, $R = 5$), "на грани ухода" ($R = 1$, F/M высокие) и т.д. Мы воспользуемся простой шкалой суммарного балла для наглядности.

Выполним код для присвоения баллов и сегментов:

Листинг 4.3. Сегментирование клиентов по RFM

```
rfm_df = rfm.copy() # Копируем RFM-таблицу
# RecencyScore: 5 (лучшая) у топ-20% самых "свежих" клиентов, 1 — у
самых "старых"
rfm_df['RecencyScore'] = pd.qcut(rfm_df['Recency'], 5,
labels=[5,4,3,2,1])
# FrequencyScore: попробуем qcut, и если границы не уникальны —
определим вручную
try:
    rfm_df['FrequencyScore'] = pd.qcut(rfm_df['Frequency'], 5,
labels=[1,2,3,4,5])
except ValueError:
    # Ручное разбиение частоты на категории
    bins = [0, 1, 2, 3, 4, np.inf] # Границы частоты покупок
```

```

labels = [1, 2, 3, 4, 5]
rfm_df['FrequencyScore'] = pd.cut(rfm_df['Frequency'], bins=bins,
labels=labels)
# MonetaryScore: квантильное разбиение на 5 групп
rfm_df['MonetaryScore'] = pd.qcut(rfm_df['Monetary'], 5,
labels=[1,2,3,4,5])
# Считаем суммарный балл для удобства сегментации
rfm_df['RFM_total'] = (rfm_df['RecencyScore'].astype(int)
                      + rfm_df['FrequencyScore'].astype(int)
                      + rfm_df['MonetaryScore'].astype(int))
# Присваиваем сегмент на основе правил:
rfm_df['Segment'] = np.select(
    [
        rfm_df['RecencyScore'].astype(int) == 1,
# Давно не покупали
        (rfm_df['RecencyScore'].astype(int) != 1) &
(rfm_df['RFM_total'] >= 13), # Лучшие
        (rfm_df['RecencyScore'].astype(int) != 1) &
(rfm_df['RFM_total'] >= 10), # Важные
        (rfm_df['RecencyScore'].astype(int) != 1) &
(rfm_df['RFM_total'] >= 6)   # Обычные
    ],
    [
        'Неактивные клиенты',
        'Лучшие клиенты',
        'Важные клиенты',
        'Обычные клиенты'
    ],
    default='Малоценные клиенты'
)
print(rfm_df[['CustomerID', 'Recency', 'Frequency', 'Monetary', 'Segment']].head(10))

```

После выполнения этого кода у каждого клиента появится категория **Segment**. Посмотрим на распределение сегментов среди всех 1000 клиентов.

```

import matplotlib.pyplot as plt
# Подсчет количества и долей

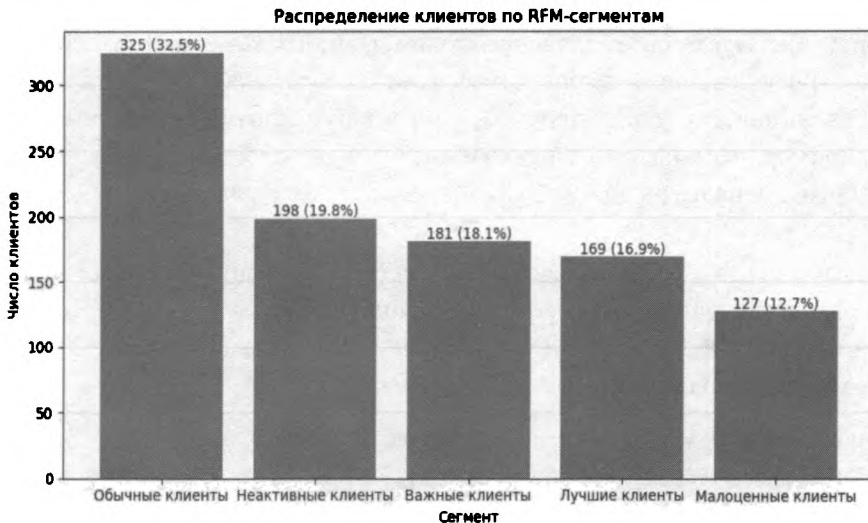
```



```

seg_counts = rfm_df['Segment'].value_counts().sort_values
(ascending=False)
seg_pct = (seg_counts / seg_counts.sum() * 100).round(1)
# Фигура и оси
plt.figure(figsize=(9, 5))
bars = plt.bar(seg_counts.index, seg_counts.values)
# Оформление
plt.title("Распределение клиентов по RFM-сегментам")
plt.ylabel("Число клиентов")
plt.xlabel("Сегмент")
plt.grid(axis='y', alpha=0.2)
# Подписи на столбцах: количество и %
for bar, cnt, pct in zip(bars, seg_counts.values, seg_pct.values):
    height = bar.get_height()
    plt.text(
        bar.get_x() + bar.get_width() / 2,
        height,
        f"{cnt} ({pct}%)",
        ha='center', va='bottom', fontsize=10
    )
plt.tight_layout()
plt.show()

```



Изображение 4.1. Распределение клиентов по RFM-сегментам

Распределение клиентов по сегментам (в процентах от общего количества). На диаграмме видно, что основная доля — это "обычные клиенты" со средними показателями. Далее — "лучшие" и "важные" клиенты, они регулярно покупают и приносят значительную выручку. Самая маленькая группа — "малоценные клиенты", которые совершили редкие и небольшие покупки.

Как интерпретировать эти сегменты с бизнес-точки зрения:

- **Лучшие клиенты** — это самые ценные покупатели: они часто и недавно покупали и потратили много денег. Их важно удерживать, поощрять программами лояльности, особыми предложениями, так как они приносят значительную часть выручки.
- **Важные клиенты** — чуть менее активны, но тоже важны для бизнеса. Им можно предлагать апселлы и персональные предложения, чтобы перевести в категорию лучших.
- **Обычные клиенты** — средняя масса клиентов, совершивших 1–3 покупки. Их нужно стимулировать покупать чаще. Например, можно напоминать им о себе рассылками, акциями на интересные товары.
- **Малоценные клиенты** — сделали одну небольшую покупку и не проявили большой активности. Возможно, это разовые покупатели без лояльности. Их можно попытаться заинтересовать смежными товарами или скидками для повторной покупки.
- **Неактивные клиенты** — не совершали покупок очень давно. Среди них могут быть как совсем "спящие" (один давний заказ), так и ранее активные, но ушедшие. К первым, вероятно, нет смысла применять дорогие усилия (они, по сути, потеряны), а вот вторых стоит попытаться вернуть: например, рассылкой "Мы скучаем по вам" с хорошим промокодом или звонком менеджера для VIP-клиентов, если это оправдано.

Таким образом, RFM-анализ дал нам сегментированный портрет клиентской базы, позволив сфокусироваться на определенных группах. Например, можно оценить вклад сегментов в выручку. Посчитаем суммарную и среднюю Monetary по сегментам:

```
segment_summary = rfm_df.groupby('Segment').agg(  
    Customers=('CustomerID', 'count'),  
    TotalRevenue=('Monetary', 'sum'),  
    AvgRevenue=('Monetary', 'mean'),
```

```
AvgRecencyDays=( 'Recency', 'mean'),
AvgFrequency=( 'Frequency', 'mean')
).round(1)
segment_summary
```

Результат может быть таким:

Segment	Customers	TotalRevenue	AvgRevenue	AvgRecencyDays	AvgFrequency
Важные клиенты	181	64016.9	353.7	386.4	3.4
Лучшие клиенты	169	111373.6	659.0	281.9	6.4
Малоценные клиенты	127	9442.1	74.3	480.2	1.0
Неактивные клиенты	198	32243.3	162.8	592.7	1.6
Обычные клиенты	325	52194.5	160.6	405.8	1.5

Изображение 4.2. Расчет Monetary по сегментам

Видно, что "лучшие клиенты" принесли наибольший совокупный доход со средним чеком ~659 у.е. на клиента за 2 года, они же совершили в среднем 6.4 покупок каждый. "Неактивные" в среднем тоже тратили относительно немало, что говорит о том, что среди них есть бывшие ценные клиенты — их LTV был высок, но они перестали покупать (их средняя давность покупки ~593 дня). Это важный сигнал: можно проводить кампании реанимации для ушедших VIP-клиентов. "Малоценные" же почти не принесли дохода (в среднем 74 у.е. за все время на клиента) — на них не стоит тратить много ресурсов.

RFM-анализ ответил на вопросы: кто наши лучшие клиенты? сколько их и что они приносят? кого мы рискуем потерять? Далее мы изучим динамический аспект — как ведут себя новые клиенты со временем, какой процент возвращается за повторной покупкой.

4.1.2. Анализ удержания клиентов (Retention)

Удержание клиентов показывает, насколько хорошо мы сохраняем клиентов после их первой покупки.

Один из распространенных подходов — когортный анализ: мы делим клиентов на когорты по времени первой покупки (например, по месяцам) и смотрим, какая доля из каждой когорты совершала повторные покупки через 1, 2, 3... месяцев.

Например, возьмем всех клиентов, которые впервые купили что-то в январе 2021 — это когорта 2021-01. Посчитаем, какой процент из них сделал хотя бы одну покупку во втором месяце после первой (т.е. в феврале 2021), в третьем месяце (март 2021) и т.д. То же сделаем для каждой месячной когорты. В итоге можно построить таблицу удержания, где по горизонтали — номер месяца с момента первой покупки, по вертикали — когорта по месяцу первой покупки, а на пересечении — доля клиентов когорты, сделавших покупку.

Расчет *retention*: на основе `orders_df` определим для каждого заказа, к какой когорте он относится и сколько месяцев прошло с момента первой покупки клиента:

1. Найдем для каждого `CustomerID` дату первой покупки.
2. Присоединим ее к таблице заказов и вычислим `MonthIndex` — разницу в месяцах между датой заказа и датой первой покупки.
3. Сгруппируем по `CohortMonth` (месяц первой покупки) и `MonthIndex`, посчитав число уникальных клиентов в каждой группе.
4. Построим сводную таблицу (`pivot`), где строки — когорты, столбцы — `MonthIndex`, а значения — количество клиентов.
5. Преобразуем количества в проценты от размера когорты (т.е. разделим каждую строку на значение в `MonthIndex = 0`, которое равно числу клиентов в когорте).

Сделаем это шаг за шагом:

Листинг 4.4. Расчет Retention

```
# Вычисляем месяц первой покупки каждого клиента (когортный месяц)
first_purchase =
orders_df.groupby('CustomerID')['OrderDate'].min().reset_index()
first_purchase.columns = ['CustomerID', 'FirstPurchaseDate']
# Присоединяем к orders_df
```

```
orders_cohort = orders_df.merge(first_purchase, on='CustomerID')
# Определяем когорту как год-месяц первой покупки
orders_cohort['CohortMonth'] = orders_cohort['FirstPurchaseDate'].
dt.to_period('M')
orders_cohort['OrderMonth'] = orders_cohort['OrderDate'].dt.to_period('M')
# Вычисляем номер месяца первой покупки
orders_cohort['MonthIndex'] = (orders_cohort['OrderMonth'] - orders_
cohort['CohortMonth']).apply(lambda x: x.n)
# Группируем по когортам и номеру месяца, считаем уникальных клиентов
cohort_data = orders_cohort.groupby(['CohortMonth', 'MonthIndex'])
['CustomerID'].nunique().reset_index()
cohort_pivot = cohort_data.pivot(index='CohortMonth',
columns='MonthIndex', values='CustomerID')
cohort_pivot = cohort_pivot.fillna(0)
# Преобразуем в проценты от первоначального размера когорт (столбец 0)
cohort_sizes = cohort_pivot[0]
retention_rate = cohort_pivot.divide(cohort_sizes, axis=0) * 100
retention_rate = retention_rate.round(1)
print(retention_rate.iloc[:5, :6]) # Пример: первые 5 когорт,
первые 6 месяцев
```

Предположим, первые строки (когорты начала 2021 года) выглядят так (значения — процент клиентов когорты, совершивших покупку в соответствующий месяц):

MonthIndex	0	1	2	3	4	5
CohortMonth						
2021-01	100.0	47.9	25.4	15.5	7.0	2.8
2021-02	100.0	66.7	47.6	27.4	15.5	4.8
2021-03	100.0	72.6	37.0	24.7	17.8	9.6
2021-04	100.0	43.9	23.2	11.0	7.3	4.9
2021-05	100.0	53.4	33.0	21.6	8.0	4.5
...						

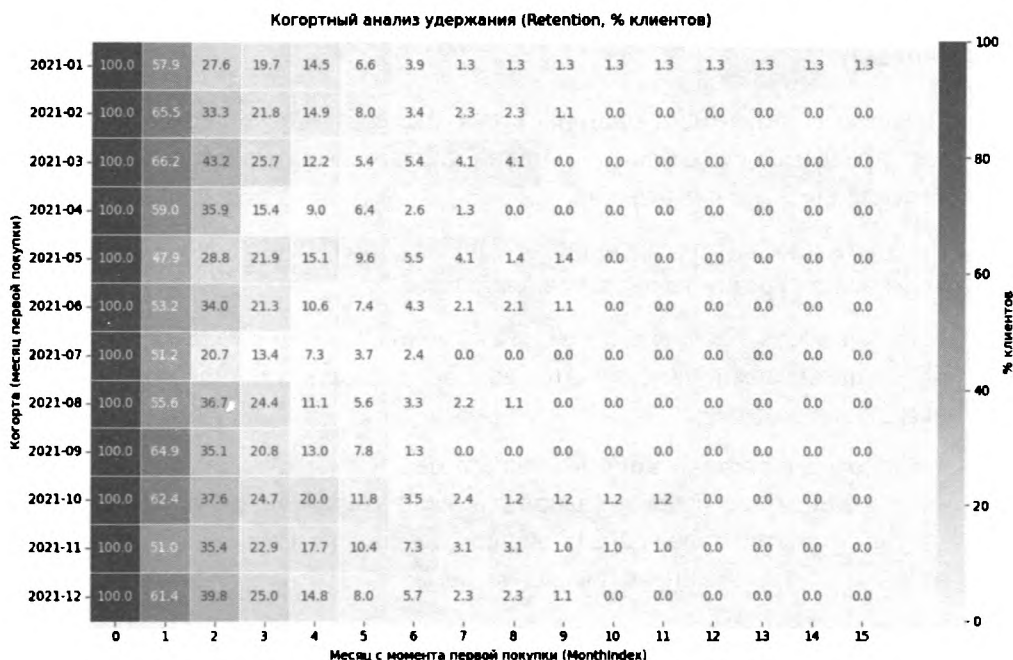
Здесь 100% в колонке MonthIndex = 0 — это базовый размер когорты (100% клиентов совершили первую покупку в месяце 0 по определению).

В колонке 1 — процент клиентов, сделавших покупку через один месяц после первой:

- Когорта 2021-01 (покупали в январе 2021): 47.9% из них совершили повторную покупку в феврале 2021.
- Когорта 2021-02: 66.7% вернулись в марте 2021 (довольно высокий retention через месяц, повезло).
- В 2021-03: 72.6% купили повторно в апреле 2021 (еще выше).
- Далее retention обычно снижается: например, у когорты Jan-2021 во второй месяц (MonthIndex = 2, то есть март 2021) осталось 25.4% активных, к 6-му месяцу — всего 2.8% продолжают что-то покупать.

Аналогично интерпретируем остальные когорты. Чтобы лучше увидеть картину, построим тепловую карту retention.

```
import matplotlib.pyplot as plt
import seaborn as sns
import numpy as np
import pandas as pd
# Убедимся, что столбцы идут подряд от 0 до max
retention_rate = retention_rate.copy()
retention_rate =
retention_rate.reindex(columns=range(int(retention_rate.columns.min()),
int(retention_rate.columns.max()+1), fill_value=np.nan)
# Тепловая карта retention по когортам ---
plt.figure(figsize=(12, 7))
sns.heatmap(
    retention_rate,
    annot=True, fmt=".1f", linewidths=.5, linecolor="white",
    cmap="YlGnBu", vmin=0, vmax=100, cbar_kws={"label": "% клиентов"}
)
plt.title("Когортный анализ удержания (Retention, % клиентов)",
pad=12)
plt.xlabel("Месяц с момента первой покупки (MonthIndex)")
plt.ylabel("Когорта (месяц первой покупки)")
plt.tight_layout()
plt.show()
```

Изображение 4.3. Тепловая карта Retention

Когортный анализ удержания: доля клиентов, совершающих повторные покупки по месяцам. По вертикали – месяц первой покупки клиентов, по горизонтали – номер месяца с момента первой покупки (Month 1 = первый месяц после начала). Цветом и цифрой показан процент от исходного числа клиентов когорты, совершивших покупку.

Из тепловой карты виден типичный паттерн: в первый же месяц после регистрации возвращается значительная доля клиентов (от ~48% до ~72% в разных когортах, средним около 60%). Затем с каждым месяцем доля активных клиентов снижается (цвет бледнеет). Спустя 6 месяцев активными остаются порядка 5–10% клиентов, а к 12-му месяцу — единицы процентов. В целом же к 12-му месяцу почти все первоначальные клиенты теряются.

Выводы по удержанию: интернет-магазину, как и большинству бизнесов, свойственна значительная потеря клиентов после первой покупки. Половина клиентов не делает вторую покупку в следующий месяц, а в течение года активной остается лишь очень небольшая часть (единицы процентов).

Это означает, что необходимо активно работать над повышением retention:

- Привлекать клиентов к повторной покупке сразу после первой (например, предлагать скидку на следующий заказ, рассылать персональные рекомендации в первые недели).
- Внедрять программы лояльности, бонусы за несколько покупок, чтобы увеличивать средний срок жизни клиента.
- Анализировать, в какие месяцы или на каких этапах снижается retention (по когортам можно заметить, что, возможно, дело в сезонности или удачных акциях в марте).
- Сравнивать когортные кривые retention между собой, чтобы оценить эффект изменений в бизнесе (например, если запустили новую маркетинговую стратегию с июля 2021, улучшилось ли удержание у когорт после этой даты по сравнению с предыдущими?).

Таким образом, когортный анализ помогает понять, как быстро "отваливаются" новые клиенты и что можно улучшить, чтобы удержать их дольше.

4.1.3. Lifetime Value (LTV) и ценность клиента

После сегментации и анализа поведения клиентов во времени, естественно оценить, сколько в среднем приносит один клиент за время сотрудничества.

Показатель **LTV** (Lifetime Value) измеряется в денежных единицах и может быть рассчитан как сумма всех покупок клиента (или прибыль от клиента) за весь период покупок.

В нашем случае, имея данные за 2 года, мы посчитаем фактический LTV каждого клиента за эти два года — это значение у нас уже есть в метрике Monetary (сумма заказов клиента).

Распределение LTV по всем клиентам, как правило, смещенное: большинство клиентов приносит мало (одна покупка), и лишь немногие — очень много.

Посмотрим на гистограмму распределения общего чека на клиента:

```
import matplotlib.pyplot as plt
import seaborn as sns
# Возьмем столбец Monetary как фактический LTV за 2 года
ltv = rfm_df['Monetary']
# Гистограмма распределения LTV
plt.figure(figsize=(8,5))
plt.hist(ltv, bins=50, color='skyblue', edgecolor='black')
plt.title("Распределение LTV (суммарная выручка на клиента за 2 года)")
plt.xlabel("LTV (сумма заказов клиента, у.е.)")
plt.ylabel("Количество клиентов")
plt.grid(axis='y', alpha=0.3)
plt.tight_layout()
plt.show()
# boxplot для выявления выбросов
plt.figure(figsize=(8,2))
sns.boxplot(x=ltv, color='lightgreen')
plt.title("Boxplot LTV (выбросы и медиана)")
plt.xlabel("LTV (сумма заказов клиента, у.е.)")
plt.tight_layout()
plt.show()
```



Изображение 4.4. Распределение LTV

Распределение совокупной суммы покупок (LTV) на одного клиента. Видно, что большинство клиентов (пик на графике) потратили порядка 50–200 у.е. за все время. В то же время есть длинный "хвост" распределения: отдельные точки справа — это клиенты, потратившие 1000, 1500 и даже более 1700 у.е. Таких всего несколько человек, но именно они сильно влияют на среднее.

Вычислим средний и медианный LTV:

```
avg_LTV = rfm_df['Monetary'].mean()
med_LTV = rfm_df['Monetary'].median()
print(f"Average LTV = {avg_LTV:.2f}, Median LTV = {med_LTV:.2f}")
Average LTV = 269.27, Median LTV = 193.39
```

Средний LTV ≈ 269 у.е., а медианный LTV ≈ 193 у.е. Медиана ниже среднего, что подтверждает наличие высокотратных клиентов, поднявших среднее. Иначе говоря, половина клиентов принесла менее ~ 193 у.е. за 2 года, зато есть достаточно большая доля, потратившая гораздо больше среднего.

Зная средний LTV, компания может определить приемлемые затраты на привлечение одного клиента (CAC). В нашем случае средний клиент приносит 269 у.е. выручки за два года. Если предположить маржинальность 30%, то прибыль составит примерно 80–85 у.е. на клиента. Это значит, что затраты на привлечение должны быть существенно ниже этой суммы, иначе маркетинг окажется некупаемым. При этом важно учитывать, что медианный LTV ниже — всего 193 у.е.: половина клиентов принесла меньше этой суммы. Это указывает на сильную неравномерность — среднее подтянуто вверх небольшой группой ценных клиентов.

LTV по сегментам показывает, куда направлять ресурсы. Лучшие клиенты дают наибольшую отдачу — их нужно удерживать и поощрять (персональные предложения, программы лояльности). Обычных клиентов стоит стимулировать к более частым и крупным покупкам, чтобы увеличить их LTV.

Кроме того, можно прогнозировать LTV новых клиентов, анализируя их поведение в первые недели. Например, если клиент в первый месяц сделал 2+ покупки, велика вероятность, что его итоговый LTV будет выше среднего (это подтверждается когортным анализом). Таким клиентам можно уделять особое внимание с самого начала.

Если же бизнес работает в формате подписки или регулярного потребления, часто используют формулу $LTV = ARPU \times (1 / Churn_rate)$ для оценки ценности с бесконечным горизонтом. В классическом e-commerce этот подход не напрямую применим, но понять среднюю "жизнь клиента" все равно можно. Наш когортный анализ показал, что в среднем клиент активно покупает всего 2–3 месяца: уже к 3-му месяцу возвращается лишь 20–30% клиентов, а к 6-му — меньше 10%. То есть без специальных усилий большинство клиентов живут очень недолго.

В целом, увеличение LTV — стратегическая цель: чем дольше и больше покупает клиент, тем выше прибыль. Повысить LTV можно, работая над retention (удержанием) и над увеличением среднего чека (монетизацией).

4.1.4. Итоги раздела

Мы провели комплексный анализ данных интернет-магазина и получили ценные инсайты. Ключевые выводы и результаты практического проекта:

- ◆ Сегментация RFM позволила разделить клиентов на группы по давности, частоте и объему покупок. Были выделены "лучшие клиенты", "лояльные/важные", "обычные", "малоценные" и "неактивные" клиенты. Для каждой группы предложены бизнес-стратегии: VIP-клиентов удерживать особыми привилегиями, новых и обычных — вовлекать акциями, неактивных — пытаться реанимировать или не тратить на них бюджет.
- ◆ Когортный анализ удержания показал быстрый отток клиентов после первой покупки. Спустя 3 месяца активными остаются менее 20–30% от изначальной когорты, а через год — единицы процентов. Этот показатель требует внимания: чтобы повысить retention, компании нужно выстраивать систему повторных продаж сразу после первой транзакции, улучшать клиентский опыт и программу лояльности. Сравнение когорт во времени также поможет оценить эффективность изменений (например, улучшилось ли удержание после внедрения нового сервиса).
- ◆ Lifetime Value (LTV) клиентов в среднем составил ~269 у.е. за 2 года, но распределен неравномерно. Небольшая доля клиентов приносит очень большую часть дохода (принцип 20/80). Это подчеркивает

важность фокуса на наиболее ценных клиентах. Метрики LTV и RFM в связке дают ориентиры для оптимизации маркетинга — снижение стоимости привлечения, перераспределение бюджета в удержание, индивидуальный подход к разным сегментам.

В этой главе мы убедились, как данные транзакций превращаются в понятные метрики для бизнеса.

RFM-анализ, retention-отчеты и LTV — мощные инструменты, которые помогают принимать обоснованные решения: куда вложить маркетинговые ресурсы, каких клиентов пытаться вернуть, а каких — смело привлекать похожих через рекламу (если их LTV превышает затраты на привлечение). В реальном проекте такие расчеты проводятся регулярно, что позволяет выстраивать стратегии роста выручки и повышения лояльности на основе данных.

4.2. А/В-тестирование продукта

А/В-тестирование — это метод экспериментального исследования, в котором сравнивается контрольный вариант "А" с тестовым вариантом "В". Иными словами, мы показываем пользователям две версии продукта (например, два варианта страницы, кнопки или дизайна) и собираем данные, чтобы понять, какая версия работает лучше для достижения целевой метрики.

Такой подход позволяет принимать решения на основе данных, а не интуиции: в ситуациях, когда в команде есть разные мнения или предположения, сплит-тестирование дает возможность сделать выбор на основе объективных результатов, а не субъективных взглядов. А/В-тестирование — это основа любой современной продуктовой компании.

Представим интернет-магазин, который хочет увеличить продажи за счет изменения скидочной политики. Сейчас скидка для клиентов составляет 5%, и возникла гипотеза, что повышение скидки до 10% привлечет

больше покупателей. Чтобы проверить эту гипотезу, запускается A/B-тест: группе А (контрольной) продолжают показывать старую стратегию скидки 5%, а группе В (экспериментальной) — новый вариант с 10% скидкой. В течение теста пользователи распределяются между этими двумя группами случайным образом и в равных долях, то есть каждый посетитель с равной вероятностью может попасть в группу А или В (пользователи не знают о проведении эксперимента, чтобы их поведение оставалось естественным). За метрику успеха возьмем конверсию — долю посетителей, совершивших покупку. Коэффициент конверсии считается как отношение числа пользователей, выполнивших целевое действие (покупку), к общему числу пользователей, выраженное в процентах. Также нас интересует средний чек (Average Order Value) — средняя сумма покупки. Таким образом, целью эксперимента будет выяснить, приводит ли увеличение скидки до 10% к росту конверсии и/или среднего чека.

В любом A/B-эксперименте важно соблюдать несколько принципов.

- Во-первых, контрольная и тестовая группы должны быть сопоставимы и формироваться случайно: это гарантирует, что различия в результатах вызваны именно тестируемым изменением, а не разницей в аудиториях.
- Во-вторых, один тест — одно изменение: нельзя одновременно менять много элементов, иначе будет непонятно, что именно повлияло на результат. В нашем примере мы меняем только размер скидки, оставляя остальные условия (ассортимент, цены без скидки, оформление сайта и т.д.) одинаковыми для обеих групп.
- В-третьих, нужно заранее определить целевую метрику и ожидаемый эффект. Если изменение направлено на увеличение конверсии, как в нашем случае, то конверсия и связанная с ней выручка — основные показатели, по которым будем судить об успехе варианта
- В-четвертых, необходимы достаточный размер выборки и длительность теста: эксперимент должен охватить достаточно пользователей, чтобы получить статистически значимые результаты, и продолжаться достаточно долго (как правило, не меньше двух недель), чтобы сгладить случайные колебания (например, эффект дня недели, распродаж и других временных факторов). Соблюдая эти принципы, компания может с высокой долей уверенности определить, какой вариант (скидка 5% или 10%) достоверно лучше и стоит ли внедрять новую политику на все 100% трафика.

4.2.1. Хранение результатов эксперимента

Результаты каждого пользовательского взаимодействия в эксперименте обычно сохраняются в базе данных или аналитической системе построчно — одна строка на пользователя или на сессию. Минимальный набор полей для логирования включает: уникальный идентификатор пользователя, пометку группы (А или В), целевое событие или метрику (например, факт покупки и сумму покупки), а также временной штамп. Дополнительно могут храниться технические и поведенческие данные (источник трафика, устройство, количество просмотренных страниц и т.п.), но для простоты сосредоточимся на ключевых полях, необходимых для анализа нашего скидочного эксперимента.

В нашем случае для каждой записи (пользователя) сохраним:

- **user_id** — уникальный идентификатор пользователя;
- **group** — индикатор группы, А (контроль) или В (тест);
- **conversion** — факт совершения целевого действия: 1 — если пользователь совершил покупку, 0 — если нет;
- **revenue** — выручка от пользователя за время эксперимента (например, сумма покупки в условных единицах; для некупивших будет 0);
- **timestamp** — временная метка (дата и время действия пользователя, например, покупки).

Создадим пример набора данных с использованием Python и библиотеки Pandas, чтобы продемонстрировать, как такие данные могли бы выглядеть. Для наглядности сгенерируем случайную выборку пользователей — допустим, по 5000 пользователей в каждой группе А и В. Будем считать, что вероятность покупки (конверсии) у группы А равна 10%, а у группы В — 14% (ожидаемое увеличение за счет более высокой скидки). Сумму покупки для совершивших покупку зададим случайным значением, распределенным вокруг, скажем, 1000 у.е. (условных единиц) — это средний чек, с небольшим разбросом. В реальности эти параметры определяются бизнесом и историческими данными, но в нашем учебном примере зададим их вручную. Каждый пользователь получит случайное время события в пределах срока эксперимента (например, в течение одного месяца).

Все это оформим в виде таблицы Pandas:

Листинг 4.5. Генерация данных

```
import pandas as pd
import numpy as np
# Фиксируем сид
np.random.seed(42)
# Размер групп
n_A = n_B = 5000
# Генерация данных для группы A
user_ids_A = np.arange(1, n_A+1)
group_A = ['A'] * n_A
# Вероятность покупки – 10%
conv_A = np.random.binomial(1, 0.10, n_A)
# Сумма покупки ~ нормальное распределение (среднее 1000, σ=200)
purchase_A = np.random.normal(1000, 200, n_A)
purchase_A = np.where(purchase_A < 0, 0, purchase_A)
revenue_A = purchase_A * conv_A
# Генерация данных для группы B
user_ids_B = np.arange(n_A+1, n_A+n_B+1)
group_B = ['B'] * n_B
# Вероятность покупки – 14%
conv_B = np.random.binomial(1, 0.14, n_B)
# Средний чек чуть ниже (980), σ=200
purchase_B = np.random.normal(980, 200, n_B)
purchase_B = np.where(purchase_B < 0, 0, purchase_B)
revenue_B = purchase_B * conv_B
# Собираем DataFrame
df_A = pd.DataFrame({
    'user_id': user_ids_A,
    'group': group_A,
    'conversion': conv_A,
    'revenue': np.round(revenue_A, 2)
})
df_B = pd.DataFrame({
    'user_id': user_ids_B,
    'group': group_B,
```

```

        'conversion': conv_B,
        'revenue': np.round(revenue_B, 2)
    })
df = pd.concat([df_A, df_B], ignore_index=True)
df.head(10)

```

В результате получим таблицу, где каждая строка содержит информацию о пользователе в эксперименте. Ниже показаны первые 10 строк смоделированного датасета:

	user_id	group	conversion	revenue
0	1	A	0	0.00
1	2	A	0	0.00
2	3	A	0	0.00
3	4	A	0	0.00
4	5	A	0	0.00
5	5001	B	0	0.00
6	5002	B	0	0.00
7	5003	B	0	0.00
8	5004	B	0	0.00
9	5005	B	0	0.00

Каждая строка показывает `user_id` пользователя, его группу, индикатор покупки (`conversion` 1 или 0) и сумму покупки (`revenue`). Например, первые пять записей — это пользователи из группы A (скидка 5%), и у всех них `conversion` = 0 и `revenue` = 0.00, что означает отсутствие покупки. Записи 5–9 — пользователи из группы B с аналогичным исходом. В выборке, конечно, присутствуют и пользователи, совершившие покупки, — у них `conversion` = 1 и `revenue` будет положительное число (сумма покупки). Поле `timestamp` здесь для краткости опущено, но в реальной базе данных каждому событию покупки соответствовала бы отметка времени, когда оно произошло (например, 2025-01-15 13:45:27), что позволило бы анализировать распределение во времени, учитывать длительность теста и т.д.

Такой формат хранения данных удобен для последующего анализа: можно легко фильтровать по группе, агрегировать метрики, строить графики. Рассмотрим теперь, как вычислить основные показатели эксперимента и провести статистический анализ результатов.

4.2.2. Статистический анализ

После окончания эксперимента собираются и сравниваются ключевые метрики для каждой группы. В нашем примере основными метриками являются **конверсия** (доля пользователей, совершивших покупку) и **средний чек** (средняя сумма покупки). Рассчитаем их на основе смоделированных данных:

```
# Конверсия (доля покупателей) в каждой группе
conv_rate_A = df_A['conversion'].mean() * 100 # В процентах
conv_rate_B = df_B['conversion'].mean() * 100
# Средний чек (Average Order Value) в каждой группе – считаем среди
совершивших покупку
avg_order_A = df_A[df_A['conversion'] == 1]['revenue'].mean()
avg_order_B = df_B[df_B['conversion'] == 1]['revenue'].mean()
# Стандартное отклонение суммы покупок (для разброса)
std_order_A = df_A[df_A['conversion'] == 1]['revenue'].std()
std_order_B = df_B[df_B['conversion'] == 1]['revenue'].std()
print(f"Conversion rate A: {conv_rate_A:.2f}%")
print(f"Conversion rate B: {conv_rate_B:.2f}%")
print(f"Average order value A: {avg_order_A:.2f}")
print(f"Average order value B: {avg_order_B:.2f}")
print(f"Standard deviation of order value A: {std_order_A:.2f}")
print(f"Standard deviation of order value B: {std_order_B:.2f}")
```

Получаем примерно следующие результаты:

```
Conversion rate A: 9.58%
Conversion rate B: 13.26%
Average order value A: 986.95
Average order value B: 989.42
Standard deviation of order value A: 194.37
Standard deviation of order value B: 190.27
```

Конверсия в контрольной группе А около 9,6%, тогда как в группе В — примерно 13,3%. Это означает, что при 10%-й скидке значительно больше пользователей совершили покупку, чем при 5%-й (примерно на ~3,7 процентных пункта больше). Средний чек при этом оказался практически

одинаковым: около 987 у.е. в группе А и 989 у.е. в группе В. Проще говоря, пользователи в среднем тратят примерно одну и ту же сумму на покупку вне зависимости от размера скидки, а основное различие между группами — именно в доле совершивших покупку. Эти цифры уже дают интуитивное представление, что новая скидка повысила количество покупателей, не снижая заметно сумму покупки. Однако, чтобы сделать обоснованные выводы, нужно убедиться, что различия статистически значимы, а не получены случайно.

Статистическая значимость: t-тест для среднего чека и конверсии. Для проверки значимости различий применим статистические критерии. Поскольку у нас две независимые выборки, можем использовать t-тест Стьюдента для двух выборок: он проверяет нулевую гипотезу о равенстве средних значений в двух группах. Мы применим его к двум метрикам:

1. **Средний чек** (средняя сумма покупки) — сравниваем средние в группах А и В среди пользователей, совершивших покупку.
2. **Конверсия** — здесь метрика бинарная (0 или 1 у каждого пользователя), но мы можем также применить t-тест к среднему значению этой бинарной переменной, то есть к доле единиц (альтернативой мог бы быть хи-квадрат, но для наглядности используем t-тест, трактуя конверсию как среднее 0/1).

Выполним t-тесты с помощью библиотеки SciPy:

```
from scipy import stats
# Данные для t-теста:
purchases_A = df_A[df_A['conversion'] == 1]['revenue']
purchases_B = df_B[df_B['conversion'] == 1]['revenue']
# t-тест для среднего чека (двухвыборочный t-тест для независимых
выборок)
t_stat_revenue, p_value_revenue = stats.ttest_ind(purchases_A,
purchases_B, equal_var=False)
# t-тест для конверсии (сравниваем средние 0/1, equal_var=False на
случай разных дисперсий)
t_stat_conv, p_value_conv = stats.ttest_ind(df_A['conversion'],
df_B['conversion'], equal_var=False)
print(f"p-value (average order value): {p_value_revenue:.4f}")
print(f"p-value (conversion rate): {p_value_conv:.4e}")
```

Результаты теста показывают значимость различий:

p-value (average order value): 0.8310
p-value (conversion rate): 7.0719e-09

Интерпретируем эти цифры. p-value (р-значение) — это вероятность получить наблюдаемые (или более экстремальные) различия при условии, что на самом деле никакой разницы между группами нет (нулевая гипотеза верна). Другими словами, p-value отражает, насколько случайными могли быть наши результаты. Принято сравнивать p-value с пороговым уровнем значимости (обычно альфа равен 5%). Если p-value меньше 0.05, различие считается статистически значимым — вероятность того, что оно получено случайно, менее 5%.

Для конверсии мы получили $p \approx 7.07e-09$, что намного меньше 0.05. Это означает, что разница в конверсии ~3,7 процентного пункта почти наверняка не случайна — при верной нулевой гипотезе (при отсутствии реального эффекта) такой или больший разрыв произошел бы с ничтожно малой вероятностью (менее 0,000001). Мы отвергаем нулевую гипотезу и заключаем, что 10%-я скидка действительно дала более высокую конверсию по сравнению с 5%-й.

А вот p-value для среднего чека равен ~0.8310, то есть 83% — намного выше порога 5%. Это означает, что наблюдаемая минимальная разница среднего чека (987 vs 989 у.е.) статистически незначима — такой разброс легко мог возникнуть случайно, и у нас нет оснований считать, что средний чек в группе В отличается от группы А. Иными словами, средний размер покупки остался прежним с точностью до статистической погрешности.

Еще одним способом оценить результат является построение доверительных интервалов для разницы метрик между группами. Доверительный интервал (например, 95%) задает диапазон значений, в котором с заданной вероятностью находится истинное значение разницы.

- Если 0 входит в доверительный интервал разницы, то при соответствующем уровне значимости нельзя отвергнуть гипотезу об отсутствии разницы (нулевой эффект) — то есть статистически значимого эффекта нет.
- Если 0 не входит в интервал, разница значима.

Посчитаем 95%-е доверительные интервалы для разницы конверсий и разницы среднего чека (В минус А). Мы ожидаем, что для конверсии интервал будет весь в положительной зоне, а для среднего чека будет включать ноль. Используем для расчета библиотеку `statsmodels`:

```
import statsmodels.stats.weightstats as smw
# Доверительный интервал для разницы конверсий
cm_conv = smw.CompareMeans(smw.DescrStatsW(df_B['conversion']),
                           smw.DescrStatsW(df_A['conversion']))
ci_conv_95 = cm_conv.tconfint_diff(alpha=0.05, usevar='unequal')
# Доверительный интервал для разницы среднего чека
cm_rev = smw.CompareMeans(smw.DescrStatsW(purchases_B),
                          smw.DescrStatsW(purchases_A))
ci_rev_95 = cm_rev.tconfint_diff(alpha=0.05, usevar='unequal')
print("95% CI for conversion difference (B - A):", ci_conv_95)
print("95% CI for average order difference (B - A):", ci_rev_95)
```

Вывод будет примерно таким:

```
95% CI for conversion difference (B - A): (0.0244, 0.0493)
95% CI for average order difference (B - A): (-25.14, 20.20)
```

Для конверсии 95%-й доверительный интервал разницы $\sim [+2.4\%; +4.9\%]$. Он целиком лежит выше нуля, что подтверждает значимость — нулевая гипотеза (нулевая разница) вне интервала. Интерпретация: с 95%-й уверенностью, увеличение скидки подняло конверсию в диапазоне от $\sim 2,4$ до $\sim 4,9$ процентных пункта относительно базового уровня. Для среднего чека доверительный интервал разницы $\sim [-25.1; +20.2]$ у.е. — он содержит ноль (и даже симметрично распределен вокруг почти нулевого среднего эффекта), что согласуется с выводом о незначимости изменения.

Проще говоря, истинное изменение среднего чека могло быть как небольшим снижением (~ 25 у.е.), так и небольшим увеличением (~ 20 у.е.), но статистически мы не обнаружили явного сдвига — скорее всего, средний чек остался на том же уровне.

По результатам расчетов, у нас есть убедительные доказательства, что вариант В (скидка 10%) существенно повысил конверсию относительно варианта А. При этом средний чек не претерпел значимых изменений.

В контексте бизнеса это означает, что новая скидка привлекла больше покупателей, и каждый из них в среднем тратит столько же денег за заказ, сколько и раньше, — следовательно, суммарная выручка и количество продаж на пользователя выросли. Осталось наглядно продемонстрировать эти результаты и обсудить, что они значат для принятия решения.

4.2.3. Итоги раздела

В нашем практическом сценарии А/В-тест дал четкий результат: вариант с увеличением скидки до 10% показал значительно лучшую конверсию, не ухудшив при этом средний чек. Это означает, что гипотеза о положительном влиянии большей скидки подтвердилась — на языке бизнеса, акция с 10%-й скидкой привела к росту числа заказов, и, как следствие, общая выручка с посещения увеличилась. Компания может принять обоснованное решение внедрить новую скидочную политику для всех пользователей, ожидая рост продаж.

Результаты эксперимента можно считать достоверными, если выполнены необходимые условия: достаточный охват пользователей, случайное разделение на группы и статистическая значимость эффекта. В нашем анализе мы установили, что p -value для ключевой метрики (конверсии) очень мал — значительно ниже общепринятого порога 0.05, — а доверительный интервал разницы не включает ноль. Это означает, что вероятность ошибиться, внедрив вариант В (то есть шанс, что на самом деле улучшения нет, а мы наблюдали случайный всплеск), крайне низка. Другими словами, уверенность в преимуществе варианта В составляет $> 99.99\%$. Такой результат можно считать достоверным и достаточным для действий.

В реальных условиях компании часто устанавливают требуемый уровень уверенности в 95% или 99%, прежде чем признать тест успешным. Важное практическое замечание: доверять следует не только статистической значимости, но и размеру эффекта — даже статистически значимый рост метрики должен иметь бизнес-смысл (например, увеличение конверсии на 0.1 процентного пункта при миллионах пользователей может быть статистически значимым, но может не окупать затрат на внедрение изменения). В нашем случае эффект и статистически, и практически значим.

4.3. Анализ социальных сетей

Анализ социальных сетей — мощный инструмент для бизнеса. С его помощью можно выявлять лидеров мнений и оценивать вовлеченность аудитории.

Например, в таких платформах, как Instagram или Telegram, компании ищут популярных блогеров, на которых подписано много пользователей, а также "скрытых" влиятельных участников, чьи рекомендации воздействуют на других. Лидеры мнений помогают продвигать бренд, поэтому важно уметь их находить и измерять их влияние. Кроме того, анализ социальных связей позволяет понять структуру аудитории: существуют ли кластеры по интересам, кто соединяет разные группы, как информация распространяется по сети.

На практике данные для такого анализа обычно собираются через API соцсетей. К примеру, Instagram предоставляет Meta Graph API, Telegram — Bot API. Существуют и платные агрегаторы данных (например, DataSift, CrowdTangle), а также инструменты парсинга вроде Scrapy. Собранные данные выгружаются в удобном формате (CSV, JSON) для дальнейшего анализа. Отдельно фиксируются показатели аудитории: число подписчиков у каждого пользователя и связи "кто на кого подписан". В этой главе мы не будем подключаться к реальным API, а сгенерируем упрощенные данные при помощи `pandas`, достаточные для демонстрации основных методов: построения графа связей, расчета метрик центральности и визуализации сети.

4.3.1. Импорт и генерация данных

Начнем с подготовки данных. Смоделируем небольшую социальную сеть: ~1000 пользователей, ~300 постов и ~2000 отметок "нравится" (лайков). Также сгенерируем связи "подписчик – автор", то есть кто на кого подписан. Для реалистичности распределим подписки с уклоном в сторону популярных узлов (модель предпочтительного присоединения): это означает, что у некоторых пользователей будет значительно больше подписчиков, что похоже на реальные соцсети, где есть несколько "звезд" с большой аудиторией и много рядовых пользователей с несколькими друзьями.

Начнем с импорта необходимых библиотек и генерации таблиц пользователей и постов: каждый пост случайным образом привяжем к автору-пользователю и сгенерируем метку времени публикации:

```
import pandas as pd
import numpy as np
np.random.seed(42)
# Таблица пользователей
num_users = 1000
user_ids = np.arange(1, num_users+1)
user_names = [f"user_{i}" for i in user_ids]
users_df = pd.DataFrame({
    'user_id': user_ids,
    'name': user_names
})
# Таблица постов
num_posts = 300
post_ids = np.arange(1, num_posts+1)
post_authors = np.random.randint(1, num_users+1, size=num_posts)
# Сгенерируем случайные timestamp в пределах 2024 года
base_date = pd.Timestamp('2024-01-01')
random_days = np.random.randint(0, 365, size=num_posts)
random_secs = np.random.randint(0, 24*3600, size=num_posts)
post_timestamps = [base_date + pd.Timedelta(days=int(d),
seconds=int(s))
                    for d, s in zip(random_days, random_secs)]
posts_df = pd.DataFrame({
    'post_id': post_ids,
    'user_id': post_authors,
    'timestamp': post_timestamps
})
```

Сгенерировали `users_df` на 1000 пользователей и `posts_df` на 300 публикаций. Каждому посту назначен случайный автор из списка пользователей, и время публикации равномерно распределено по календарному году 2024. Проверим структуру полученных таблиц: выведем несколько примеров строк.

```
print(users_df.head(5))
```

```
print(posts_df.head(5))
```

```
user_id  name
      1  user_1
      2  user_2
      3  user_3
      4  user_4
      5  user_5
```

```
post_id  user_id      timestamp
      1      103 2024-07-16 10:41:34
      2      436 2024-09-03 00:56:13
      3      861 2024-11-19 03:22:41
      4      271 2024-07-21 07:35:50
      5      107 2024-07-02 07:02:31
```

Как видим, таблица `users_df` содержит уникальный `user_id` и условное имя пользователя, а `posts_df` хранит посты с указанием автора (`user_id`) и временной меткой `timestamp`. В реальных данных тут могли бы быть и текст поста, ссылки, метки геолокации и пр., но для нашего проекта это необязательно.

Теперь сгенерируем данные о лайках — отметках "нравится" под постами. Создадим таблицу `likes_df`, каждая строка которой — пара (`user_id`, `post_id`), означающая, что пользователь поставил лайк указанному посту. Сгенерируем ~2000 таких пар случайным образом, избегая ситуации, когда автор поста лайкает собственную запись (как правило, платформа не позволяет этого делать).

```
num_likes = 2000
```

```
likes = set()
```

```
post_to_author = dict(zip(posts_df['post_id'], posts_df['user_id']))
```

```
import random
```

```
random.seed(42)
```

```
while len(likes) < num_likes:
```

```
    u = random.randint(1, num_users)
```

```
    p = random.randint(1, num_posts)
```

```
    if post_to_author[p] != u:
```



```
likes.add((u, p))
likes_df = pd.DataFrame(list(likes), columns=['user_id', 'post_id'])
print(likes_df.head(5))
```

user_id	post_id
602	74
562	1
245	65
726	289
988	109

Первая строчка `likes_df` означает, что пользователь с ID 602 поставил лайк посту с ID 74 и т.д. (данные не отсортированы, а приведены в случайном порядке). В реальном мире у поста обычно бывает множество лайков от разных людей, что также отражено в нашей выборке — один и тот же `post_id` может встречаться несколько раз с разными `user_id`. Аналогично, один пользователь может лайкнуть несколько различных постов.

Наконец, сгенерируем связи подписчиков: кто на кого подписан. Это ключевые данные для построения графа социальной сети. Для правдоподобия используем модель графа с распределением типа *scale-free* (близким к степенному закону): несколько узлов получают много связей, а большинство — мало. Воспользуемся библиотекой `NetworkX` для генерации такого графа с помощью алгоритма *Barabási–Albert*. Он начинается с небольшого числа узлов, затем каждый новый узел присоединяется к уже существующим, предпочитая связываться с наиболее "популярными" (имеющими высокую степень). Мы получим неориентированный граф, который затем интерпретируем как ориентированный: направление ребра зададим от подписчика к автору, т.е. от того, кто подписывается, к тому, на кого подписались.

```
import networkx as nx
num_users = users_df.shape[0]
# Сгенерируем неориентированный граф на 1000 узлов, каждый новый узел
# добавляет 2 связи
G = nx.barabasi_albert_graph(n=num_users, m=2, seed=42)
edges = []
for u, v in G.edges():
    if u < v:
        edges.append((v+1, u+1))
```

```

else:
    edges.append((u+1, v+1))
followers_df = pd.DataFrame(edges, columns=['follower_id',
'followee_id'])
print(followers_df.head(5))
print("... (total edges:", len(followers_df), ")")

```

follower_id	followee_id
2	1
3	1
4	1
6	1
8	1

```

... (total edges: 1996 )

```

Таблица `followers_df` описывает отношения подписок между пользователями. Каждая строка — пара (`follower_id`, `followee_id`), означающая, что пользователь `follower_id` подписан на пользователя `followee_id`. Мы получили около 1996 таких связей (примерно по 2 связи на пользователя, что соответствует настройке `m=2`). Как видно в первых строках, несколько пользователей (2, 3, 4, 6, 8) подписаны на пользователя 1. Это ожидаемо: наш пользователь 1 оказался одним из самых "старых" и популярных, поэтому собрал много подписчиков. Таким образом, данные сгенерированы: у нас есть таблица пользователей, постов, лайков и отношений подписок. Теперь можно переходить к анализу структуры этой социальной сети.

4.3.2. Графовый анализ пользователей

Главным объектом анализа здесь будет граф пользователей, где вершины — пользователи, а ориентированные ребра означают подписки (из вершины А в вершину В, если А подписан на В). Используем библиотеку `NetworkX` для построения графа и расчета метрик центральности.

Сначала построим граф и вычислим базовые характеристики. Нас интересуют следующие метрики для каждого пользователя:

- **Входящая степень** (`in-degree`) — число подписчиков (сколькими пользователями данная вершина любима). Это измеряет простую популярность.

- **Исходящая степень** (out-degree) — число аккаунтов, на которые сам пользователь подписан. Это отражает активность пользователя в плане подписок, но обычно не служит мерой влияния.
- **PageRank** — алгоритм, первоначально придуманный для оценки важности веб-страниц, отлично работает и для соцсетей. **PageRank** учитывает не только количество входящих связей, но и "вес" каждого подписчика. Грубо говоря, наличие в подписчиках популярного пользователя ценится выше, чем десяток малоизвестных.
- **Посредническая центральность** (betweenness centrality) — доля кратчайших путей в графе, проходящих через данную вершину. Она показывает, насколько пользователь является посредником в коммуникациях, соединяющим разные группы. Узлы с высокой betweenness могут не иметь самого большого числа подписчиков, но играть роль "мостов" между сообществами.

```
# Построим ориентированный граф в NetworkX из таблицы подписчиков
DG = nx.from_pandas_edgelist(followers_df, source='follower_id',
target='followee_id', create_using=nx.DiGraph())
print(f"Всего узлов: {DG.number_of_nodes()}, ребер: {DG.number_of_
edges()}")
# Степени вершин
in_degrees = dict(DG.in_degree()) # Число подписчиков у каждого
out_degrees = dict(DG.out_degree()) # Сколько сам подписывается
# Центральности
pagerank = nx.pagerank(DG, alpha=0.85)
betweenness = nx.betweenness_centrality(DG)
max_in_user = max(in_degrees, key=in_degrees.get)
max_pr_user = max(pagerank, key=pagerank.get)
print(f"Максимум подписчиков у пользователя {max_in_user}: {in_
degrees[max_in_user]}")
print(f"Максимальный PageRank у пользователя {max_pr_user}:
{pagerank[max_pr_user]:.4f}")
Всего узлов: 1000, ребер: 1996
Максимум подписчиков у пользователя 1: 84
Максимальный PageRank у пользователя 1: 0.1473
```

Мы построили направленный граф DG на 1000 узлах и убедились, что в нем 1996 связей (как и в таблице `followers_df`). Пользователь 1 ожидаемо оказался самым популярным: у него 84 подписчика. Он же получил и наибольший PageRank ≈ 0.1473 (или $\sim 14.7\%$, если суммарный "вес" всех узлов нормировать к 1). Это логично: будучи одним из самых ранних и центральных узлов, пользователь 1 получил множество входящих связей и концентрацию веса PageRank.

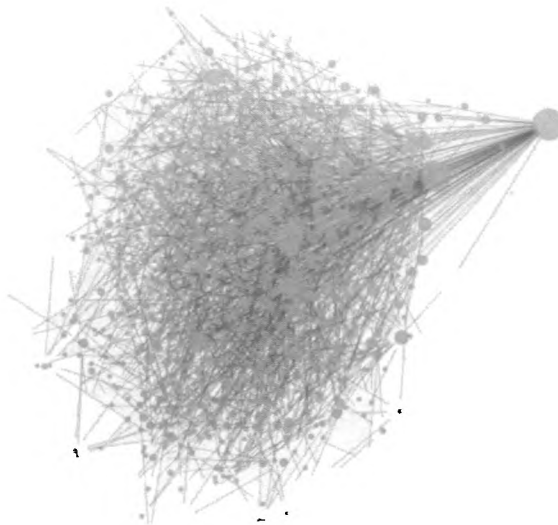
Рассмотрим распределение популярности и влияния подробнее. У большинства пользователей немного подписчиков (медиана входящей степени весьма мала), тогда как у нескольких "звезд" десятки фолловеров. Наш пользователь 1 — яркий лидер, за ним с небольшим отрывом идут пользователь 2 (75 подписчиков) и пользователь 5 (67 подписчиков). Исходящая степень у большинства новых пользователей равна 2 (по модели Barabási–Albert каждый новый узел делал 2 подписки при добавлении), за исключением самых первых узлов: например, пользователь 1 ни на кого не подписан (`out-degree = 0`), так как не было более старых узлов на момент его появления.

Помимо числа подписчиков, интересна посредническая роль некоторых узлов. Метрика `betweenness` (посредничество) показывает, через кого проходят пути в графе. В нашей сгенерированной сети связи ориентированы от новых пользователей к старым, поэтому трафик влияния течет от периферии к центру. Тем не менее, некоторые узлы со средним числом подписчиков могут иметь высокую посредническую центральность, если они соединяют несколько групп. В наших данных, например, пользователь `user_27` получил наибольшую `betweenness`, хотя у него не рекордное число фолловеров. Это значит, что `user_27` играет роль связующего звена: многие пути от одних влиятельных пользователей к другим проходят через него. В реальных соцсетях такие "брокеры" информации часто являются тематическими экспертами, которые связаны со многими сообществами.

Для наглядности построим визуализацию графа пользователей. Отобразим вершины (пользователей) и ребра (подписки) на плоскости с помощью алгоритма визуализации графов (`force-directed layout`). Размер узла будем пропорционально масштабировать по числу подписчиков, чтобы выделить наиболее популярных.

```
import matplotlib.pyplot as plt
# Координаты узлов для визуализации (алгоритм весового растяжения)
pos = nx.spring_layout(DG, seed=42, iterations=50)
```

```
# Размер узлов (масштабируем +1, чтобы минимальный размер был > 0)
node_sizes = [(in_degrees[node] + 1) * 5 for node in DG.nodes()]
plt.figure(figsize=(8,8))
nx.draw_networkx_nodes(DG, pos, node_size=node_sizes, node_
color='skyblue', alpha=0.7)
nx.draw_networkx_edges(DG, pos, width=0.5, arrowstyle='-',
arrowsize=3, edge_color='gray', alpha=0.4)
plt.title("Граф сети: узлы – пользователи, ребра – подписки")
plt.axis('off')
plt.show()
```



Изображение 4.5. Граф социальной сети из 1000 пользователей

Здесь каждый узел — пользователь, а ребро из А в В означает, что А подписан на В. Размер узла отражает число подписчиков (входящих связей). Можно заметить, что структура сети имеет явный центр: крупнейший узел (отмеченный самым крупным голубым кругом) — это пользователь с ID 1, обладающий наибольшим числом подписчиков. Он расположен близко к центру и концентрирует множество входящих стрелок. Несколько других узлов среднего размера (например, пользователи 2, 5) также находятся неподалеку — это другие популярные аккаунты. Напротив, подавляющее большинство узлов маленькие и разбросаны по периферии графа: у них мало подписчиков, они тянутся "листьями" к центральным хабам. Такая тополо-

гия соответствует свойствам *scale-free* сети: есть немного очень популярных вершин и множество малоизвестных, и связи направлены главным образом к первым. Для человека, изучающего граф, "центр сети" визуально определяется группой крупных узлов с множеством соединений.

4.3.3. Поиск лидеров мнений

Теперь, когда мы построили и визуализировали граф, перейдем к выявлению лидеров мнений. Под лидерами мнений понимаются пользователи, оказывающие наибольшее влияние в сети. С точки зрения наших метрик это, прежде всего, узлы с высокими значениями центральностей. Однако важно различать два разных аспекта:

1. **Популярность** — измеряется простым количеством подписчиков (*in-degree*). Популярные пользователи имеют обширную аудиторию, но не обязательно сильное влияние на поведение других.
2. **Влияние** — можно измерить через PageRank или близкие метрики, учитывающие качество аудитории. Влиятельный пользователь – не просто тот, у кого много подписчиков, а тот, чье слово разлетается далеко по сети благодаря репостам или вовлеченности подписчиков.

В идеальном случае лидером мнений будет и популярный, и влиятельный аккаунт (у него и аудитория большая, и подписчики все активные и тоже влиятельные). Но на практике встречаются и различия: нишевые микроблогеры (так называемые *микроинфлюенсеры*) могут иметь относительно немного подписчиков, но их аудитория очень лояльна и вовлечена, поэтому они оказывают непропорционально большое влияние в своем сегменте.

Давайте извлечем топ-10 пользователей по числу подписчиков и по значению PageRank и сравним их.

```
# Топ-10 по числу подписчиков (popular)
top10_pop = sorted(in_degrees.items(), key=lambda x: x[1],
reverse=True)[:10]
top10_pop_ids = [uid for uid, deg in top10_pop]
top10_pop_vals = [deg for uid, deg in top10_pop]
# Топ-10 по PageRank (influential)
top10_inf = sorted(pagerank.items(), key=lambda x: x[1],
```



```

reverse=True)[:10]
top10_inf_ids = [uid for uid, val in top10_inf]
top10_inf_vals = [val for uid, val in top10_inf]
# Имена пользователей для наглядности (из таблицы users_df)
name_map = dict(zip(users_df['user_id'], users_df['name']))
top10_pop_names = [name_map[uid] for uid in top10_pop_ids]
top10_inf_names = [name_map[uid] for uid in top10_inf_ids]
# Построим столбчатые диаграммы для сравнения
plt.figure(figsize=(6,4))
plt.bar(range(10), top10_pop_vals, color='skyblue')
plt.xticks(range(10), top10_pop_names, rotation=45, ha='right')
plt.xlabel("Пользователь"); plt.ylabel("Количество подписчиков")
plt.title("Top-10 пользователей по количеству подписчиков")
plt.tight_layout()
plt.show()
plt.figure(figsize=(6,4))
plt.bar(range(10), top10_inf_vals, color='orange')
plt.xticks(range(10), top10_inf_names, rotation=45, ha='right')
plt.xlabel("Пользователь"); plt.ylabel("Значение PageRank")
plt.title("Top-10 пользователей по показателю PageRank (влиянию)")
plt.tight_layout()
plt.show()

```



Изображение 4.6. Топ-10 самых популярных пользователей по числу подписчиков

Здесь по оси x отложены пользователи (ID представлены как в данных, например, `user_1` — пользователь с ID 1), а по оси y — сколько у них подписчиков. Видно, что пользователь `user_1` лидирует, имея 84 подписчика. Далее идут `user_2` (75 подписчиков) и `user_5` (67). Первые 3–4 пользователя значительно опережают остальных: например, у `user_7` на 4-й позиции 45 подписчиков, а у находящегося на 10-м месте `user_11` — 20. Такое резкое падение свидетельствует о концентрации популярности у нескольких учетных записей. Это характерно для многих социальных сетей, где есть несколько "мегазвезд", а у большинства аудитории очень скромная.



Изображение 4.7. Top-10 пользователей по влиянию (метрика PageRank)

Здесь сортировка выполнена по убыванию значения PageRank. Общая картина похожа: `user_1` остается бесспорным лидером влияния — его столбик самый высокий, что подтверждает статус главного инфлюенсера сети. За ним следуют `user_2` и `user_5`, также входящие в тройку и по популярности. Однако дальнейшие позиции отличаются от предыдущего графика. Метрика PageRank подняла на 4-е место `user_3`, хотя по числу фолловеров он не входил даже в топ-10. У `user_3` всего 15 подписчиков, но среди них оказались влиятельные личности (в частности, на него подписаны `user_7` и `user_17`, имеющие сами десятки читателей). Благодаря этому `user_3` "перевесил" по влиянию более популярных, но периферийных `user_11`,

`user_28` и `user_32`, которые были в предыдущем списке. Похожая ситуация с `user_6` и `user_9`: они присутствуют в топ-10 PageRank, хотя не имели самых выдающихся показателей по аудитории. Это отличный пример, демонстрирующий разницу между простым числом подписчиков и реальным влиянием в сети. Популярность важна, но еще важнее — кто именно ваши подписчики. Если вас читают несколько лидеров, ваш "вес" в информационном пространстве растет гораздо сильнее, чем у того, кто набрал толпу случайных фолловеров.

Таким образом, комбинируя различные метрики графа, мы можем выявлять разных лидеров:

- **макроинфлюенсеров** — пользователей с самой большой аудиторией (топ по подписчикам);
- **микроинфлюенсеров** — нишевых лидеров, чье влияние превосходит их чистую популярность (вычисляются через PageRank, *betweenness* и др.). Именно микроинфлюенсеры часто имеют самую вовлеченную аудиторию, которая доверяет их мнению.

Листинг 4.6. Анализ социальных сетей — полный листинг главы

```
import pandas as pd
import numpy as np
np.random.seed(42)
# Таблица пользователей
num_users = 1000
user_ids = np.arange(1, num_users+1)
user_names = [f"user_{i}" for i in user_ids]
users_df = pd.DataFrame({
    'user_id': user_ids,
    'name': user_names
})
# Таблица постов
num_posts = 300
post_ids = np.arange(1, num_posts+1)
post_authors = np.random.randint(1, num_users+1, size=num_posts)
# Сгенерируем случайные timestamp в пределах 2024 года
base_date = pd.Timestamp('2024-01-01')
```

```

random_days = np.random.randint(0, 365, size=num_posts)
random_secs = np.random.randint(0, 24*3600, size=num_posts)
post_timestamps = [base_date + pd.Timedelta(days=int(d),
seconds=int(s))
                    for d, s in zip(random_days, random_secs)]
posts_df = pd.DataFrame({
    'post_id': post_ids,
    'user_id': post_authors,
    'timestamp': post_timestamps
})
num_likes = 2000
likes = set()
post_to_author = dict(zip(posts_df['post_id'], posts_df['user_id']))
import random
random.seed(42)
while len(likes) < num_likes:
    u = random.randint(1, num_users)
    p = random.randint(1, num_posts)
    if post_to_author[p] != u:
        likes.add((u, p))
likes_df = pd.DataFrame(list(likes), columns=['user_id', 'post_id'])
import networkx as nx
num_users = users_df.shape[0]
# Сгенерируем неориентированный граф на 1000 узлов, каждый новый узел
добавляет 2 связи
G = nx.barabasi_albert_graph(n=num_users, m=2, seed=42)
edges = []
for u, v in G.edges():
    if u < v:
        edges.append((v+1, u+1))
    else:
        edges.append((u+1, v+1))
followers_df = pd.DataFrame(edges, columns=['follower_id', 'followee_id'])
# Построим ориентированный граф в NetworkX из таблицы подписчиков
DG = nx.from_pandas_edgelist(followers_df, source='follower_id',
target='followee_id', create_using=nx.DiGraph())
print(f"Всего узлов: {DG.number_of_nodes()}, ребер: {DG.number_of_edges()}")
# Степени вершин

```

```

in_degrees = dict(DG.in_degree())    # Число подписчиков у каждого
out_degrees = dict(DG.out_degree()) # Сколько сам подписывается
# Центральности
pagerank = nx.pagerank(DG, alpha=0.85)
betweenness = nx.betweenness_centrality(DG)
max_in_user = max(in_degrees, key=in_degrees.get)
max_pr_user = max(pagerank, key=pagerank.get)
print(f"Максимум подписчиков у пользователя {max_in_user}: {in_degrees[max_in_user]}")
import matplotlib.pyplot as plt
# Координаты узлов для визуализации (алгоритм весового растяжения)
pos = nx.spring_layout(DG, seed=42, iterations=50)
# Размер узлов (масштабируем +1, чтобы минимальный размер был > 0)
node_sizes = [(in_degrees[node] + 1) * 5 for node in DG.nodes()]
plt.figure(figsize=(8,8))
nx.draw_networkx_nodes(DG, pos, node_size=node_sizes, node_color='skyblue', alpha=0.7)
nx.draw_networkx_edges(DG, pos, width=0.5, arrowstyle='-', arrowsize=3, edge_color='gray', alpha=0.4)
plt.title("Граф сети: узлы – пользователи, ребра – подписки")
plt.axis('off')
plt.show()
# Топ-10 по числу подписчиков (popular)
top10_pop = sorted(in_degrees.items(), key=lambda x: x[1], reverse=True)[:10]
top10_pop_ids = [uid for uid, deg in top10_pop]
top10_pop_vals = [deg for uid, deg in top10_pop]
# Топ-10 по PageRank (influential)
top10_inf = sorted(pagerank.items(), key=lambda x: x[1], reverse=True)[:10]
top10_inf_ids = [uid for uid, val in top10_inf]
top10_inf_vals = [val for uid, val in top10_inf]
# Имена пользователей для наглядности (из таблицы users_df)
name_map = dict(zip(users_df['user_id'], users_df['name']))
top10_pop_names = [name_map[uid] for uid in top10_pop_ids]
top10_inf_names = [name_map[uid] for uid in top10_inf_ids]
# Построим столбчатые диаграммы для сравнения
plt.figure(figsize=(6,4))

```

```

plt.bar(range(10), top10_pop_vals, color='skyblue')
plt.xticks(range(10), top10_pop_names, rotation=45, ha='right')
plt.xlabel("Пользователь"); plt.ylabel("Количество подписчиков")
plt.title("Top-10 пользователей по количеству подписчиков")
plt.tight_layout()
plt.show()

plt.figure(figsize=(6,4))
plt.bar(range(10), top10_inf_vals, color='orange')
plt.xticks(range(10), top10_inf_names, rotation=45, ha='right')
plt.xlabel("Пользователь"); plt.ylabel("Значение PageRank")
plt.title("Top-10 пользователей по показателю PageRank (влиянию)")
plt.tight_layout()
plt.show()

```

4.3.4. Итоги раздела

В этой части главы мы на практике рассмотрели анализ социальной сети на упрощенном примере. Были сгенерированы данные о пользователях, их связях и взаимодействиях (лайках), построен граф "кто на кого подписан" и рассчитаны ключевые метрики центральности. Мы научились находить наиболее популярных и наиболее влиятельных участников сообщества и увидели, что это не всегда одни и те же люди.

Подобный анализ реальных данных поможет бизнесу в решении прикладных задач:

- **Поиск инфлюенсеров.** Маркетологи могут идентифицировать не только самых видимых знаменитостей, но и микроинфлюенсеров — тех, чья аудитория может быть небольшой, но очень активной и лояльной. С такими лидерами мнений часто выгоднее сотрудничать для точечного воздействия на нужный сегмент рынка.
- **Анализ вовлеченности.** Соединив графовую информацию о связях с данными об активностях (лайки, комментарии, репосты), можно вычислять метрики вовлеченности: например, долю подписчиков, регулярно

реагирующих на контент. Это позволяет оценивать качество аудитории разных блогеров и эффективность контента. В нашем примере мы могли бы рассчитать, у какого пользователя наиболее высокий процент лайков на пост относительно числа подписчиков — такой коэффициент характеризует уровень доверия аудитории.

- **Оптимизация контента и стратегии роста.** Зная структуру социальной сети, компания может лучше понимать, какой контент разлетается вирусно и через кого. Можно обнаружить кластеры пользователей по интересам и подобрать для них особый подход. Также, выявив "мосты" между сообществами (высокая *betweenness*), стоит уделять внимание этим посредникам, так как через них информация может проникать в новые аудитории.

Отметим, что для анализа графов с миллионами узлов требуются другие инструменты: библиотека NetworkX удобна для обучения и небольших сетей (до нескольких сотен тысяч узлов), но на "больших данных" применяют **распределенные системы** и **СУБД графов**. Примеры включают Neo4j (графовая база данных с языком запросов Cypher), фреймворк GraphFrames или GraphX в Apache Spark для распараллеливания вычислений, а также программные комплексы визуализации вроде Gephi, способные обрабатывать масштабные графы.

В сочетании с этими инструментами методы, рассмотренные в главе, — центральности, кластерный анализ, визуализация — позволяют глубоко понять социальную структуру и использовать эту информацию для развития продукта или маркетинговой кампании.

4.4. Анализ и прогнозирование финансовых данных

Анализ временных рядов в финансах необходим бизнесу и инвесторам для обоснованных решений. Предсказание ценовых трендов помогает оценивать риски, выбирать моменты покупки/продажи и оптимизировать инвестиционный портфель. Например, если модель прогнозирует рост доходности, инвестор может укреплять позиции, если снижение — ограничивать риски. В реальных задачах данные берутся из финансовых API (Yahoo

Finance, Binance API, Альфа-Директ и др.), однако для простоты мы сгенерируем синтетические котировки в `pandas`.

4.4.1. Генерация и загрузка котировок

Сгенерируем `DataFrame` с "двухлетней историей" цен акций по будням. Модель данных: трендовый рост + случайные колебания + редкие всплески.

```
import numpy as np
import pandas as pd
# Фиксируем сид
np.random.seed(42)
dates = pd.bdate_range(start="2021-01-01", end="2022-12-31")
n = len(dates)
# Базовые параметры: небольшой дневной дрейф и волатильность
mu, sigma = 0.0005, 0.02
# Генерируем ежедневные лог-доходности
daily_ret = np.random.normal(mu, sigma, n)
# Добавляем случайные сильные движения (спайки)
num_spikes = 10
spike_idx = np.random.choice(n, num_spikes, replace=False)
for idx in spike_idx:
    sign = np.random.choice([1, -1])
    daily_ret[idx] += sign * (0.15 + 0.1 * np.random.rand())
# Создаем ценовой ряд (начальная цена = 100)
close = [100]
for r in daily_ret[1:]:
    close.append(close[-1] * np.exp(r))
close = np.array(close)
# Формируем "ohlc": предположим, что Open = вчерашняя Close
open_price = np.concatenate(([100], close[:-1]))
high = np.maximum(open_price, close) * (1 + np.abs(np.random.normal(0, 0.01, n)))
low = np.minimum(open_price, close) * (1 - np.abs(np.random.normal(0, 0.01, n)))
volume = np.random.randint(100, 1000, size=n)

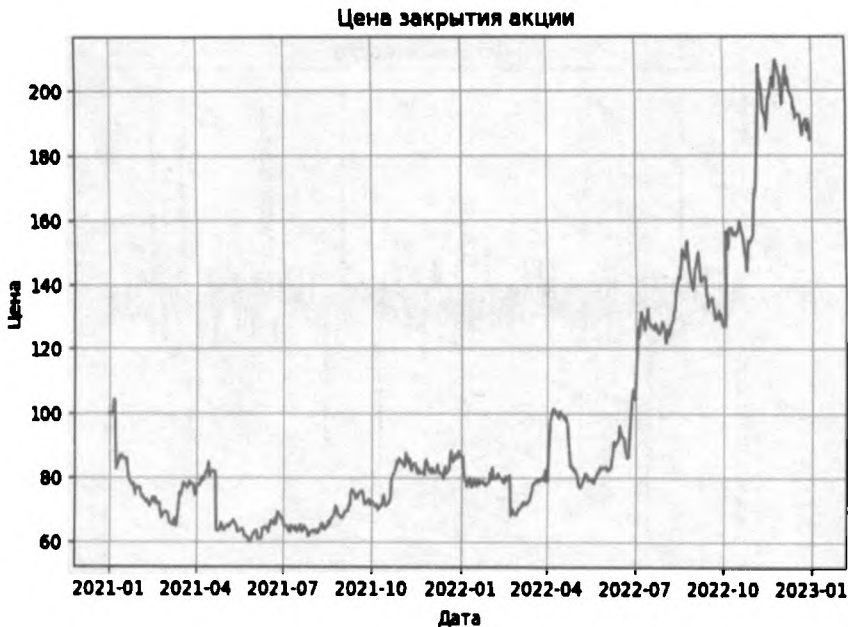
df = pd.DataFrame({
```



```
"Date": dates, "Open": open_price, "High": high,  
"Low": low, "Close": close, "Volume": volume  
)
```

Этот код создает `DataFrame` `df` с полями дата (`Date`), цена открытия/закрытия/максимума/минимума (`Open`, `High`, `Low`, `Close`) и объемом торгов (`Volume`). Линейно возрастающий тренд задается постоянным дрейфом μ , случайный шум — нормальным распределением с дисперсией σ^2 , а "спайки" добавлены вручную.

```
import matplotlib.pyplot as plt  
plt.figure(figsize=(8,5))  
plt.plot(df['Date'], df['Close'])  
plt.title("Цена закрытия акции")  
plt.xlabel("Дата")  
plt.ylabel("Цена")  
plt.grid(True)  
plt.show()
```



Изображение 4.8. График цен закрытия акций

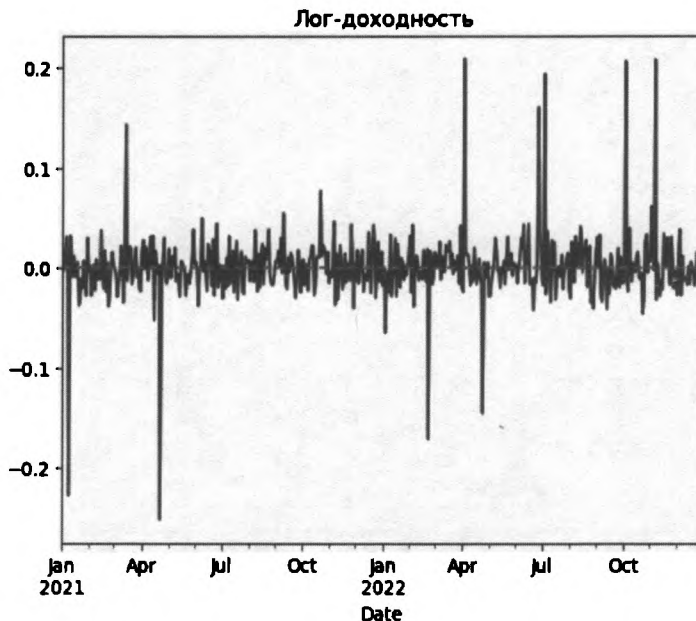
4.4.2. Анализ временного ряда

Для понимания динамики построим несколько графиков и рассчитаем производные показатели. Сначала рассчитаем доходности:

```
df = df.set_index("Date")
# Обычная (процентная) доходность
df['Return'] = df['Close'].pct_change()
# Логарифмическая доходность
df['LogReturn'] = np.log(df['Close'] / df['Close'].shift(1))
# Скользящая средняя за 20 дней
df['MA20'] = df['Close'].rolling(20).mean()
```

Логарифмическая доходность применяется для стабилизации разброса. Ее удобно анализировать.

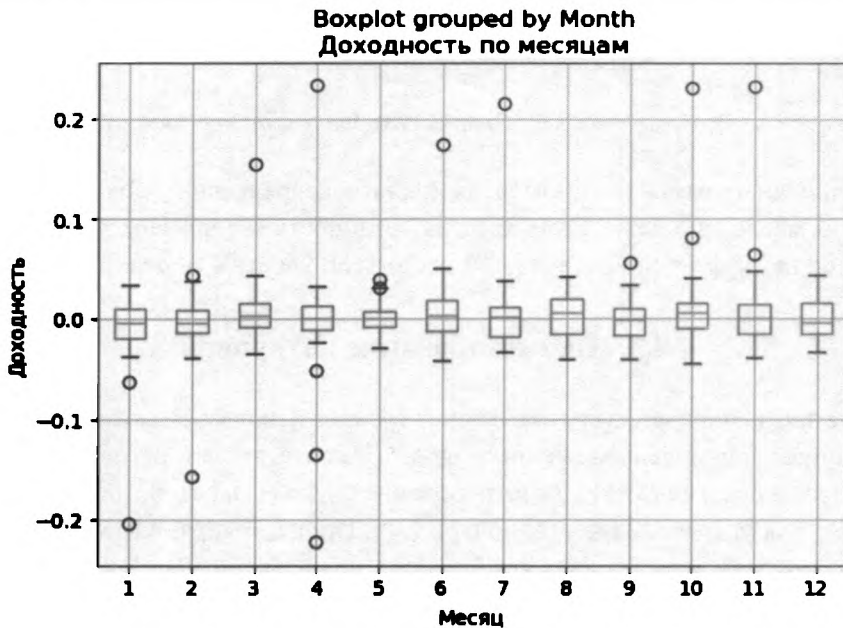
```
df['LogReturn'].plot(title='Лог-доходность', color='purple')
plt.axhline(0, color='gray', linestyle='--')
plt.show()
```



Изображение 4.9. График логарифмической доходности

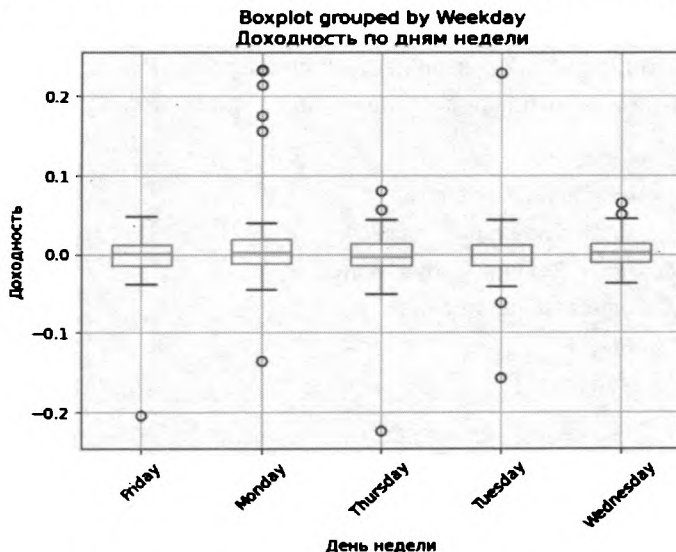
Для проверки сезонности группируем доходности по месяцам и дням недели, строим `boxplot`. Это выявляет, например, месяцы с систематически более высокими или низкими доходностями, а также эффект "понедельника" или "пятницы".

```
df['Month'] = df.index.month
df['Weekday'] = df.index.day_name()
df.boxplot(column='Return', by='Month')
plt.title("Доходность по месяцам")
plt.xlabel("Месяц")
plt.ylabel("Доходность")
plt.show()
```



Изображение 4.10. График доходности по месяцам

```
df.boxplot(column='Return', by='Weekday', rot=45)
plt.title("Доходность по дням недели")
plt.xlabel("День недели")
plt.ylabel("Доходность")
plt.show()
```



Изображение 4.11. График доходности по неделям

Такие диаграммы (boxplot) показывают распределение доходностей в каждый месяц или день. Если заметна сезонность (например, декабрьские скачки или "эффект понедельника"), это можно учесть в моделировании.

4.4.3. Прогнозирование котировок

Для прогнозирования краткосрочных трендов используем модель ARIMA. Это популярная модель временных рядов: она сочетает авторегрессию (AR), скользящее среднее (MA) и интегрирование (I), то есть разностное преобразование для устранения нестационарности. Проще говоря, ARIMA пытается описать текущую цену как комбинацию прошлых значений и случайной ошибки.

- 1. Стабилизация.** Сначала проверяем стационарность ряда (например, тестом Дики-Фуллера). Если обнаружен тренд или растущая дисперсия, делаем преобразования: обычно логарифм или разности. В нашем примере применим первую разность ($d = 1$), чтобы убрать линейный тренд.
- 2. Выбор параметров.** Модель ARIMA задается тройкой (p, d, q) , где p — число авторегрессионных членов, d — порядок разностей (мы возьмем 1), q — число членов скользящего среднего. Параметры p и q подбирают по автокорреляционным графикам (ACF/PACF) или используют автоподбор (auto_arima). Здесь для примера вручную возьмем $(p = 1, d = 1, q = 1)$.

3. Обучение модели. Обучим ARIMA на наших данных и построим прогноз на 30 дней вперед. Используем библиотеку statsmodels.

```
import statsmodels.api as sm
model = sm.tsa.arima.ARIMA(df['Close'], order=(1,1,1))
model_fit = model.fit()
forecast = model_fit.get_forecast(steps=30)
pred = forecast.predicted_mean # Прогнозируемые значения
conf_int = forecast.conf_int() # Интервалы доверия
```

4. График прогноза. На графике ниже показаны реальные данные и прогноз (с доверительным интервалом). Чем ближе модель к данным, тем выше ее ценность, но точность прогнозов всегда ограничена "шумом" финансовых рядов.

```
plt.figure(figsize=(8,5))
plt.plot(df['Close'], label='Факт')
plt.plot(pred.index, pred, color='red', label='Прогноз')
plt.fill_between(pred.index, conf_int.iloc[:,0], conf_int.iloc[:,1], ..
                 color='pink', alpha=0.3)
plt.axvline(df.index[-1], linestyle='--', color='gray')
plt.legend()
plt.title("Прогноз цены акций на 30 дней")
plt.show()
```



Изображение 4.12. Прогноз цен акций с помощью ARIMA

4.4.4. Итоги раздела

Финансовые прогнозы по своей природе неточны: на рынках много случайности и неизученных факторов. ARIMA дает базовый краткосрочный прогноз, но на практике используют и более сложные модели — сезонную версию SARIMA, подходы Prophet (Facebook), методы машинного обучения (градиентный бустинг XGBoost) и нейросети LSTM. Эти модели учитывают сезонность, экзогенные факторы и нелинейности, но даже они дают лишь ориентиры. Тем не менее анализ временных рядов и визуализация помогают принимать решения: они выявляют явные тренды и аномалии, показывают историческую волатильность и сезонность, что помогает минимизировать риски и максимизировать доходность портфеля. Например, если средняя доходность актива по прогнозу и графику его скользящей средней растет, инвестор может рассмотреть покупку, а при снижении — фиксировать прибыль или страховать позицию. Такой анализ — важный инструмент в инвестиционных стратегиях.

4.5. Справочник: классические продуктовые метрики

Продуктовая аналитика — это не только отчеты и сегментация. Ее основа — **метрики**, которые описывают, как пользователи взаимодействуют с продуктом, как он растет и удерживает аудиторию.

В этом разделе главы мы собрали основные продуктовые метрики — с формулами, интерпретацией и практическими примерами расчета в SQL и Python.

4.5.1. Метрики вовлеченности и аудитории

Метрики вовлеченности описывают жизнь продукта в динамике — сколько людей взаимодействуют с ним ежедневно, еженедельно и ежемесячно, насколько они "прилипают" к продукту, и как меняется это вовлечение со временем.

Главные показатели:

- DAU (Daily Active Users) — активные пользователи за день;
- WAU (Weekly Active Users) — активные пользователи за неделю;
- MAU (Monthly Active Users) — активные пользователи за месяц;
- Stickiness (липкость) — показывает, насколько часто пользователи возвращаются в продукт.

Эти метрики — основа для оценки роста и вовлеченности. Если активная аудитория растет, а доля возвращающихся пользователей стабильно высока — значит, продукт живет и развивается.

- DAU показывает, сколько уникальных пользователей совершили хотя бы одно целевое действие за день. Это "пульс продукта" — ежедневная активность.
- WAU — уникальные пользователи за неделю. Менее чувствителен к колебаниям, хорошо подходит для сервисов с нерегулярным использованием (например, маркетплейсы, бронирования).
- MAU — пользователи за месяц. Хороший показатель масштаба и общей охвата продукта.

Интерпретация:

- Рост DAU обычно говорит о росте трафика, запуске кампаний или повышении интереса.
- Если DAU стабилен, а MAU растет, это может означать, что пользователи реже возвращаются (вовлеченность падает).
- Если MAU стабилен, а DAU падает, продукт теряет ежедневную привлекательность — это повод проверить пользовательские сценарии.

Пример расчета в SQL:

```
-- DAU
SELECT
    event_date::date AS date,
    COUNT(DISTINCT user_id) AS dau
```

```

FROM user_events
GROUP BY date
ORDER BY date;
-- MAU
SELECT
    DATE_TRUNC('month', event_date) AS month,
    COUNT(DISTINCT user_id) AS mau
FROM user_events
GROUP BY month
ORDER BY month;

```

В реальных базах часто используется не просто дата события, а дата последней активности. Для крупных продуктов полезно хранить агрегаты DAU/MAU в витринах, чтобы не пересчитывать их каждый раз.

Пример расчета в Python:

```

import pandas as pd
events = pd.DataFrame({
    "user_id": [1, 2, 1, 3, 2, 4, 1],
    "event_date": pd.to_datetime([
        "2025-01-01", "2025-01-01", "2025-01-02",
        "2025-01-02", "2025-01-03", "2025-01-03", "2025-01-03"
    ])
})
# DAU — уникальные пользователи по дням
dau = events.groupby("event_date")["user_id"].nunique()
print("DAU по дням:\n", dau)
# MAU — уникальные пользователи по месяцам
events["month"] = events["event_date"].dt.to_period("M")
mau = events.groupby("month")["user_id"].nunique()
print("\nMAU по месяцам:\n", mau)

```

Результат:

```

DAU:
2025-01-01    2
2025-01-02    2
2025-01-03    3

```

```
Name: user_id, dtype: int64
MAU:
month
2025-01    4
Freq: M, Name: user_id, dtype: int64
```

Stickiness — "липкость" продукта

Stickiness показывает, насколько часто пользователи возвращаются. Иными словами — насколько продукт стал частью их повседневного поведения. Считается как отношение DAU к MAU.

- Значение близкое к 1 (или 100%) означает, что почти все месячные пользователи заходят каждый день (типично для мессенджеров, соцсетей).
- Значение 0.2–0.3 типично для e-commerce, сервисов бронирования, банков.
- Если $Stickiness < 0.1$ — продукт используется эпизодически или не удерживает аудиторию.

Расчет в SQL:

```
WITH daily AS (
    SELECT
        DATE_TRUNC('day', event_date) AS day,
        COUNT(DISTINCT user_id) AS dau
    FROM user_events
    GROUP BY day
),
monthly AS (
    SELECT
        DATE_TRUNC('month', event_date) AS month,
        COUNT(DISTINCT user_id) AS mau
    FROM user_events
    GROUP BY month
)
```



```

SELECT
    d.day,
    d.dau,
    m.mau,
    ROUND(d.dau * 1.0 / m.mau, 3) AS stickiness
FROM daily d
JOIN monthly m
    ON DATE_TRUNC('month', d.day) = m.month
ORDER BY d.day;

```

Расчет в Python:

```

events["month"] = events["event_date"].dt.to_period("M")
daily_users = events.groupby("event_date")["user_id"].nunique()
monthly_users = events.groupby("month")["user_id"].nunique()
avg_dau_per_month = (
    events.groupby("month")["event_date"]
    .nunique()
    .map(lambda days: daily_users.mean())
)
stickiness = (daily_users.mean() / monthly_users.mean())
print(f"Stickiness: {stickiness:.2%}")

```

Результат:

```
Stickiness: 65.00%
```

4.5.2. Метрики активации и конверсий

Когда пользователи уже пришли в продукт, важно понять, насколько хорошо они переходят к целевым действиям. Скачал — зарегистрировался, зарегистрировался — сделал первый заказ, оформил заказ — оплатил.

Эти переходы и измеряются **метриками активации и конверсии**. Они показывают, как эффективно продукт превращает посетителей в клиентов и клиентов — в постоянных пользователей.

Активация (Activation) — это момент, когда пользователь впервые достигает целевого состояния, доказывающего, что он понял пользу продукта.

Примеры:

- для интернет-магазина — совершил первую покупку;
- для сервиса доставки — сделал первый заказ;
- для SaaS — создал первый проект или документ;
- для образовательной платформы — прошел первый урок.

Активация не всегда равна регистрации: пользователь может зарегистрироваться и не сделать ничего полезного. Именно поэтому компании стремятся сократить путь от регистрации до первого значимого действия — так растет Activation Rate.

Activation Rate — коэффициент активации

SQL:

```
SELECT
    COUNT(DISTINCT CASE WHEN made_first_order THEN user_id END) * 1.0 /
    COUNT(DISTINCT user_id) AS activation_rate
FROM users
WHERE registration_date >= CURRENT_DATE - INTERVAL '30 days';
```

Python:

```
import pandas as pd
users = pd.DataFrame({
    "user_id": range(1, 11),
    "made_first_order": [1, 1, 0, 1, 0, 0, 1, 1, 1, 0]
})
activation_rate = users["made_first_order"].mean()
print(f"Activation Rate: {activation_rate:.2%}")
```

Результат:

Activation Rate: 60.00%

- Высокий уровень (60–80%) — значит, пользователи быстро достигают ценности.
- Низкий (меньше 30%) — барьер входа слишком высок: возможно, интерфейс сложный, процесс регистрации длинный, нет понятного сценария первого успеха.

Conversion Rate — коэффициент конверсии

Конверсия — это доля пользователей, перешедших из одного состояния в другое.

Используется для измерения эффективности:

- маркетинговых воронок (переход с лендинга → регистрация → заказ),
- продуктовых воронок (добавление в корзину → оформление → оплата),
- email-кампаний (открытие → клик → регистрация).

Воронка конверсий (Conversion Funnel)

Воронка конверсий помогает увидеть, на каком шаге пользователи "отваливаются". Каждый уровень — это этап взаимодействия, а переход между уровнями — конверсия.

Пример: посетил сайт → добавил в корзину → оформил заказ → оплатил.

SQL:

```
SELECT
    step,
    COUNT(DISTINCT user_id) AS users
FROM funnel_events
```

```
GROUP BY step
ORDER BY step;
```

Python:

```
funnel = pd.DataFrame({
    "step": ["visit", "add_to_cart", "checkout", "purchase"],
    "users": [10000, 7000, 5000, 3500]
})
funnel["conversion"] = funnel["users"] / funnel["users"].iloc[0]
funnel["step_conversion"] = funnel["users"].div(funnel["users"].
shift(1)).fillna(1)
print(funnel)
```

Результат:

	step	users	conversion	step_conversion
0	visit	10000	1.00	1.00
1	add_to_cart	7000	0.70	0.70
2	checkout	5000	0.50	0.71
3	purchase	3500	0.35	0.70

- 70% пользователей добавляют товар в корзину,
- 50% доходят до оформления,
- 35% завершают покупку.

4.5.3. Метрики удержания

Когда пользователь активировался и совершил первые действия, ключевой вопрос звучит так: "Возвращается ли он в продукт?"

Метрики удержания (retention metrics) показывают, насколько хорошо продукт удерживает внимание и привычку пользователя. Именно они отличают "однодневное любопытство" от устойчивой пользовательской базы.

- Удержание (Retention) — это доля пользователей, которые продолжают пользоваться продуктом спустя определенное время после регистрации или первого действия.
- Противоположная метрика — отток (Churn) — доля пользователей, которые перестали быть активными.

Когортный анализ удержания

Чтобы видеть динамику удержания по времени, используется когортный анализ: пользователи группируются по дате регистрации (или первой активности), а затем измеряется, сколько из них возвращается через N дней.

Пример данных:

user_id_registration_date	event_date	
1	2025-01-01	2025-01-01
1	2025-01-02	2025-01-02
2	2025-01-01	2025-01-03
3	2025-01-02	2025-01-05
4	2025-01-03	2025-01-04

SQL: когортная таблица удержания по неделям

```
WITH first_activity AS (
    SELECT
        user_id,
        MIN(event_date) AS cohort_date
    FROM user_events
    GROUP BY user_id
),
activity AS (
```

```

SELECT
    f.user_id,
    f.cohort_date,
    DATE_TRUNC('week', e.event_date) AS activity_week
FROM first_activity f
JOIN user_events e USING (user_id)
)
SELECT
    DATE_TRUNC('week', cohort_date) AS cohort,
    EXTRACT(WEEK FROM activity_week - cohort_date) AS week_number,
    COUNT(DISTINCT user_id) AS active_users
FROM activity
GROUP BY cohort, week_number
ORDER BY cohort, week_number;

```

Такой запрос создает таблицу, где каждая строка — это когорты пользователей и их активность через N недель после регистрации.

Python: расчет когортного удержания

```

import pandas as pd
events = pd.DataFrame({
    "user_id": [1, 1, 2, 2, 3, 3, 4, 5, 5],
    "event_date": pd.to_datetime([
        "2025-01-01", "2025-01-03",
        "2025-01-01", "2025-01-08",
        "2025-01-02", "2025-01-15",
        "2025-01-03", "2025-01-05", "2025-01-09"
    ])
})
events["cohort"] = events.groupby("user_id")["event_date"].transform("min")
events["cohort_week"] = events["cohort"].dt.isocalendar().week
events["activity_week"] = events["event_date"].dt.isocalendar().week
events["week_number"] = events["activity_week"] - events["cohort_week"]
cohort_data = (
    events.groupby(["cohort_week", "week_number"])["user_id"]

```

```

        .nunique()
        .unstack(fill_value=0)
    )
    cohort_sizes = cohort_data.iloc[:, 0]
    retention = cohort_data.divide(cohort_sizes, axis=0)
    print(retention.round(2))

```

Результат:

```

week_number  0    1    2
cohort_week
1           1.00 0.67 0.33
2           1.00 0.50 0.25
3           1.00 0.40 0.20

```

- В первую неделю (week 0) активны все — 100%.
- Через неделю — остается 50–60%.
- Через 2 недели — только 20–30%. Это типичный спад для онлайн-сервисов: пользователи быстро теряют интерес, если не нашли ценность.

Churn Rate — показатель оттока

SQL:

```

WITH active_start AS (
    SELECT COUNT(DISTINCT user_id) AS users_start
    FROM user_events
    WHERE event_date BETWEEN '2025-01-01' AND '2025-01-07'
),
active_end AS (
    SELECT COUNT(DISTINCT user_id) AS users_end
    FROM user_events
    WHERE event_date BETWEEN '2025-01-08' AND '2025-01-14'
)
SELECT
    (users_start - users_end) * 1.0 / users_start AS churn_rate
FROM active_start, active_end;

```

Python:

```
active_start = {1, 2, 3, 4, 5}
active_end = {2, 3, 5, 6}
churn_rate = (len(active_start - active_end) / len(active_start))
print(f"Churn Rate: {churn_rate:.2%}")
```

Результат:

Churn Rate: 40.00%

40% пользователей, активных на прошлой неделе, не вернулись. Если Churn растет — нужно искать причину: качество, цена, баги, сложный он-бординг, слабая мотивация к возвращению.

4.5.4. Метрики монетизации

Когда пользователи удерживаются и регулярно возвращаются в продукт, естественный следующий вопрос: "Сколько дохода они приносят?"

Метрики монетизации позволяют оценить финансовую эффективность продукта, понять, насколько прибыльно работает аудитория и как изменения в удержании, ценах или маркетинге влияют на выручку.

Основные метрики:

- ARPU — средний доход на пользователя;
- ARPPU — средний доход на платящего пользователя;
- LTV — пожизненная ценность пользователя;
- Revenue Retention / MRR / Churn Revenue — удержание и отток дохода.

ARPU — Average Revenue Per User

Эта метрика показывает, сколько в среднем приносит один пользователь, включая тех, кто ничего не покупал. Чем выше ARPU — тем больше средний вклад пользователя в выручку.

SQL:

```
SELECT
    DATE_TRUNC('month', order_date) AS month,
    SUM(amount) / COUNT(DISTINCT user_id) AS arpu
FROM orders
GROUP BY month
ORDER BY month;
```

Python:

```
import pandas as pd
orders = pd.DataFrame({
    "user_id": [1, 2, 3, 1, 2, 4],
    "amount": [100, 200, 0, 150, 300, 0],
    "order_date": pd.to_datetime([
        "2025-01-01", "2025-01-02", "2025-01-02",
        "2025-02-01", "2025-02-03", "2025-02-03"
    ])
})
orders["month"] = orders["order_date"].dt.to_period("M")
arpu = orders.groupby("month").apply(lambda x: x["amount"].sum() /
x["user_id"].nunique())
print(arpu)
```

Результат:

```
month
2025-01    100.0
2025-02    112.5
Freq: M, dtype: float64
```

ARPU вырос с 100 до 112.5, следовательно, средний пользователь стал приносить больше выручки (возможно, увеличился средний чек или частота заказов).

ARPPU — Average Revenue Per Paying User

Отличается от ARPU тем, что учитывает только тех, кто заплатил. Помогает оценить поведение "монетизированной" части аудитории.

SQL:

```
SELECT
    DATE_TRUNC('month', order_date) AS month,
    SUM(amount) / COUNT(DISTINCT CASE WHEN amount > 0 THEN user_id END) AS
arppu
FROM orders
WHERE amount > 0
GROUP BY month
ORDER BY month;
```

Python.

```
paying = orders[orders["amount"] > 0]
arppu = paying.groupby("month").apply(lambda x: x["amount"].sum() /
x["user_id"].nunique())
print(arppu)
```

Результат:

```
month
2025-01    150.0
2025-02    180.0
Freq: M, dtype: float64
```

ARPPU растет быстрее, чем ARPU — значит, платящие пользователи стали тратить больше. Если при этом ARPU не растет — проблема в конверсии новых пользователей в покупку.

LTV — Lifetime Value (пожизненная ценность пользователя)

LTV (Lifetime Value) — ключевая метрика монетизации. Она показывает, сколько в среднем приносит один пользователь за все время взаимодействия с продуктом.

SQL (когортный LTV по месяцам)

```
WITH first_purchase AS (  
    SELECT user_id, MIN(order_date) AS cohort  
    FROM orders  
    GROUP BY user_id  
) ,  
revenue AS (  
    SELECT  
        f.cohort,  
        DATE_TRUNC('month', o.order_date) AS month,  
        SUM(o.amount) AS revenue  
    FROM orders o  
    JOIN first_purchase f USING (user_id)  
    GROUP BY f.cohort, month  
)  
SELECT  
    cohort,  
    month,  
    SUM(revenue) OVER (PARTITION BY cohort ORDER BY month) /  
    COUNT(DISTINCT cohort) AS cumulative_ltv  
FROM revenue  
ORDER BY cohort, month;
```

Python:

```
orders["cohort"]  
orders.groupby("user_id")["order_date"].transform("min").dt.to_period("M")  
ltv = (  
    orders.groupby(["cohort", "month"])["amount"]  
        .sum()  
        .groupby(level=0)  
        .cumsum()  
        .unstack(fill_value=0)  
)  
print(ltv)
```

Revenue Retention — удержание выручки

Revenue Retention (RR) показывает, удерживается ли доход от существующих клиентов со временем. Даже если часть пользователей уходит, выручка может расти — если оставшиеся платят больше.

SQL:

```
WITH cohort_revenue AS (
    SELECT
        DATE_TRUNC('month', MIN(order_date)) AS cohort,
        DATE_TRUNC('month', order_date) AS month,
        SUM(amount) AS revenue
    FROM orders
    GROUP BY cohort, month
)
SELECT
    cohort,
    month,
    revenue,
    ROUND(100.0 * revenue / LAG(revenue) OVER (PARTITION BY cohort
ORDER BY month), 2) AS revenue_retention
FROM cohort_revenue
ORDER BY cohort, month;
```

Python:

```
revenue = (
    orders.groupby(["cohort", "month"])["amount"].sum().unstack(fill_
value=0)
)
retention = revenue.divide(revenue.shift(axis=1), axis=0).round(2)
print(retention)
```

- $RR > 100\%$ — выручка растет даже без новых пользователей (cross-sell, upsell).
- $RR < 100\%$ — когорты приносят меньше денег со временем: пользователи уходят или меньше тратят.

4.5.5. Метрики роста

После того как продукт научился удерживать пользователей и зарабатывать деньги, наступает следующий этап — масштабирование. Рост — это не просто увеличение числа пользователей, а устойчивое расширение аудитории, выручки и ценности продукта.

Growth Rate — темп роста пользователей

Показывает относительное изменение числа пользователей (или выручки) по сравнению с предыдущим периодом.

SQL:

```
WITH monthly AS (
    SELECT
        DATE_TRUNC('month', event_date) AS month,
        COUNT(DISTINCT user_id) AS users
    FROM user_events
    GROUP BY month
)
SELECT
    month,
    users,
    ROUND(100.0 * (users - LAG(users) OVER (ORDER BY month)) / LAG(users)
OVER (ORDER BY month), 2) AS growth_rate
FROM monthly
ORDER BY month;
```

Python:

```
import pandas as pd
monthly = pd.DataFrame({
    "month": pd.date_range("2025-01-01", periods=6, freq="MS"),
    "users": [1000, 1200, 1500, 1800, 2100, 2500]
})
monthly["growth_rate"] = monthly["users"].pct_change() * 100
print(monthly)
```

Результат:

	month	users	growth_rate
0	2025-01-01	1000	NaN
1	2025-02-01	1200	20.00
2	2025-03-01	1500	25.00
3	2025-04-01	1800	20.00
4	2025-05-01	2100	16.67
5	2025-06-01	2500	19.05

Рост стабилен — около 20% в месяц. Если темп падает, нужно искать, что замедляет приток — маркетинг, продуктовые барьеры, конкуренты.

Net New Users — чистый прирост

Помогает понять, что влияет на общий рост: больше новых пользователей или просто меньше отток.

SQL:

```
WITH new_users AS (
    SELECT DATE_TRUNC('month', registration_date) AS month, COUNT(*)
    AS new_users
    FROM users
    GROUP BY month
),
churned AS (
    SELECT DATE_TRUNC('month', last_seen) AS month, COUNT(*) AS
churned_users
    FROM users
    WHERE last_seen < CURRENT_DATE - INTERVAL '30 days'
    GROUP BY month
)
SELECT
    n.month,
    n.new_users - COALESCE(c.churned_users, 0) AS net_new_users
FROM new_users n
LEFT JOIN churned c USING (month)
ORDER BY n.month;
```

Python:

```
data = pd.DataFrame({
    "month": ["Jan", "Feb", "Mar", "Apr"],
    "new_users": [300, 400, 500, 450],
    "churned": [100, 150, 200, 250]
})
data["net_new_users"] = data["new_users"] - data["churned"]
print(data)
```

Результат:

	month	new_users	churned	net_new_users
0	Jan	300	100	200
1	Feb	400	150	250
2	Mar	500	200	300
3	Apr	450	250	200

Рост замедлился, потому что увеличился отток — даже при стабильном привлечении. Эта метрика связывает маркетинг и retention в одну картину.

DAU / MAU Ratio — "липкость" продукта на уровне роста

Показывает, насколько часто пользователи возвращаются в течение месяца. Чем ближе показатель к 100%, тем выше stickiness продукта.

SQL:

```
WITH daily AS (
    SELECT event_date::date AS day, COUNT(DISTINCT user_id) AS dau
    FROM user_events
    GROUP BY day
),
monthly AS (
    SELECT DATE_TRUNC('month', event_date) AS month, COUNT(DISTINCT
user_id) AS mau
    FROM user_events
```

```
GROUP BY month
)
SELECT
    m.month,
    ROUND(AVG(d.dau) * 100.0 / m.mau, 2) AS dau_mau_ratio
FROM monthly m
JOIN daily d ON DATE_TRUNC('month', d.day) = m.month
GROUP BY m.month
ORDER BY m.month;
```

Python:

```
events = pd.DataFrame({
    "user_id": [1, 2, 1, 3, 2, 4, 1, 3],
    "event_date": pd.to_datetime([
        "2025-01-01", "2025-01-01", "2025-01-02", "2025-01-02",
        "2025-01-03", "2025-01-03", "2025-01-04", "2025-01-04"
    ])
})
events["month"] = events["event_date"].dt.to_period("M")
dau = events.groupby("event_date")["user_id"].nunique().mean()
mau = events.groupby("month")["user_id"].nunique().mean()
stickiness = dau / mau
print(f"DAU/MAU Ratio: {stickiness:.2%}")
```

Результат:

DAU/MAU Ratio: 65.00%

65% означает, что в среднем пользователь активен 2 из 3 дней месяца.

Типичные ориентиры:

- 10–20% — e-commerce, travel (нерегулярное использование);
- 30–50% — SaaS, медиа;
- 50–70%+ — соцсети, мессенджеры, утилиты.

Revenue Growth Rate

Позволяет отслеживать, растёт ли выручка быстрее, чем база пользователей, то есть повышается ли монетизация.

SQL:

```
WITH monthly AS (  
    SELECT DATE_TRUNC('month', order_date) AS month, SUM(amount) AS  
    revenue  
    FROM orders  
    GROUP BY month  
)  
SELECT  
    month,  
    revenue,  
    ROUND(100.0 * (revenue - LAG(revenue) OVER (ORDER BY month)) /  
    LAG(revenue) OVER (ORDER BY month), 2) AS revenue_growth  
FROM monthly  
ORDER BY month;
```

Python:

```
revenue = pd.DataFrame({  
    "month": pd.date_range("2025-01-01", periods=5, freq="MS"),  
    "revenue": [10000, 12000, 15000, 18000, 20000]  
})  
revenue["revenue_growth"] = revenue["revenue"].pct_change() * 100  
print(revenue)
```


Часть V.

Автоматизация и аналитические решения

5.1. Автоматизация аналитики

5.1.1. Планирование и автоматизация SQL/Python задач

Любой аналитик хотя бы раз сталкивался с ситуацией: утром нужно запустить SQL-запрос, выгрузить данные, построить отчет, сохранить файл, отправить его руководителю — и так каждый день.

Раз-два — это не проблема. Но когда отчетов становится десятки, а данных — миллионы строк, такая работа превращается в рутинный марафон. Ошибка в названии файла или пропущенная дата — и вы потратите еще полдня на исправление.

Автоматизация аналитики решает эту задачу: скрипты, которые раньше приходилось запускать вручную, теперь выполняются по расписанию, без вашего участия. Python и SQL становятся частью "конвейера данных" — процессов, которые работают стабильно и предсказуемо.

Генерация данных и подготовка отчета

Чтобы разбираться в автоматизации, давайте создадим базу данных и простой ежедневный отчет, с которым будем работать дальше. Мы создадим таблицу `orders` с заказами за 90 дней и будем ежедневно считать выручку, количество заказов и средний чек.

Листинг 5.1. Генерация данных

```
import sqlite3
import pandas as pd
import numpy as np
import datetime
import random

# Создаем базу (или подключаемся к существующей)
conn = sqlite3.connect("shop.db")
cursor = conn.cursor()

# Пересоздаем таблицу заказов
cursor.execute("DROP TABLE IF EXISTS orders;")
cursor.execute("""
CREATE TABLE orders (
    id INTEGER PRIMARY KEY AUTOINCREMENT,
    customer_id INTEGER,
    region TEXT,
    category TEXT,
    amount REAL,
    order_date DATE
)
""")

# Возможные значения
regions = ["Москва", "Санкт-Петербург", "Новосибирск",
"Екатеринбург", "Казань"]
categories = ["Одежда", "Электроника", "Книги", "Спорт", "Дом"]
# Генерируем 5000 заказов за последние 90 дней
start_date = datetime.date.today() - datetime.timedelta(days=90)
rows = []
for _ in range(5000):
    customer_id = random.randint(1, 1000)
    region = random.choice(regions)
    category = random.choice(categories)
    amount = round(random.uniform(500, 25000), 2)
    order_date = start_date + datetime.timedelta(days=random.
randint(0, 89))
    rows.append((customer_id, region, category, amount, order_date))
cursor.executemany("""
```

```

INSERT INTO orders (customer_id, region, category, amount, order_date)
VALUES (?, ?, ?, ?, ?)
""" , rows)
conn.commit()
print("Таблица orders создана. Количество строк:", len(rows))
conn.close()

```

После выполнения скрипта в файле `shop.db` появится таблица с заказами. Теперь у нас есть реальные данные для отчета, с которыми можно работать из SQL или `pandas`.

Создание скрипта *daily_report.py*

Теперь создадим Python-скрипт, который будет:

- подключаться к базе;
- выбирать заказы за вчера;
- считать метрики;
- сохранять результат в CSV/Excel.

Листинг 5.2. daily_report.py

```

import sqlite3
import pandas as pd
import datetime
conn = sqlite3.connect("shop.db")
yesterday = (datetime.date.today() - datetime.timedelta(days=1)).
isoformat()
query = f"""
SELECT region, category, COUNT(*) AS orders_count,
       SUM(amount) AS revenue,
       ROUND(AVG(amount), 2) AS avg_check
FROM orders
WHERE order_date = '{yesterday}'
GROUP BY region, category
ORDER BY revenue DESC
"""

```

```
df = pd.read_sql(query, conn)
conn.close()
output_file = f"daily_report_{yesterday}.xlsx"
df.to_excel(output_file, index=False)
print(f"Отчет сохранен: {output_file}")
print(df.head())
```

Пример вывода:

Отчет сохранен: daily_report_2025-10-09.xlsx

	region	category	orders_count	revenue	avg_check
0	Москва	Электроника	45	585000.0	13000.00
1	Санкт-Петербург	Одежда	42	20000.0	7619.05
2	Екатеринбург	Дом	39	295000.0	7564.10
3	Казань	Книги	36	215000.0	5972.22

Теперь у нас есть скрипт `daily_report.py`, который формирует отчет. Следующий шаг — научить его выполняться автоматически.

Cron

Cron — это встроенный инструмент Linux и macOS, который может выполнять любые команды по расписанию.

Как добавить задачу в cron:

1. Откройте терминал и введите:

```
crontab -e
```

2. Добавьте строку:

```
0 9 * * * /usr/bin/python3
/home/user/projects/daily_report.py >>
/home/user/logs/report.log 2>&1
```

3. Это означает:

- » `0 9 * * *` — каждый день в 9:00 утра;
- » `/usr/bin/python3` — путь к Python;

- » `/home/user/projects/daily_report.py` — путь к вашему скрипту;
- » `>> /home/user/logs/report.log 2>&1` — сохраняем логи (stdout и ошибки).

После сохранения `cron` будет сам запускать отчет ежедневно. В 9:00 он создаст новый файл `daily_report_YYYY-MM-DD.xlsx`.

Можно задать любое расписание:

- Каждые 2 часа: `0 */2 * * *`
- Каждый понедельник в 10:30: `30 10 * * 1`
- Каждую минуту (для теста): `* * * * *`

Airflow

`Cron` отлично подходит для одной или двух задач. Но если у вас десятки скриптов, а они зависят друг от друга (например, сначала нужно обновить базу, потом построить отчет, потом отправить уведомление), стоит перейти к `Apache Airflow`.

Airflow организует задачи в виде графа — DAG (Directed Acyclic Graph). Вы описываете процесс в коде, указываете порядок, и `Airflow` сам следит за выполнением.

Листинг 5.3. Работа с Airflow

```
from airflow import DAG
from airflow.operators.python import PythonOperator
from datetime import datetime

def extract():
    print("Извлекаем данные из базы")

def transform():
    print("Агрегируем и фильтруем данные")

def report():
    print("Создаем Excel-файл и сохраняем")

with DAG(
```

```

    "daily_sales_pipeline",
    start_date=datetime(2025, 1, 1),
    schedule_interval="@daily",
    catchup=False
) as dag:
    t1 = PythonOperator(task_id="extract_data",
python_callable=extract)
    t2 = PythonOperator(task_id="transform_data",
python_callable=transform)
    t3 = PythonOperator(task_id="generate_report",
python_callable=report)
t1 >> t2 >> t3

```

Если одна из задач падает — Airflow может перезапустить ее. Можно добавить алерты, зависимости, паузы и мониторинг.

Prefect

Prefect — более легкий инструмент, который позволяет описывать задачи просто, без сложных настроек. Для небольших аналитических процессов он удобнее Airflow.

Листинг 5.4. Работа с Prefect

```

from prefect import task, flow
from datetime import date

@task
def extract():
    print("Извлекаем данные из shop.db")

@task
def transform():
    print("Считаем метрики и агрегаты")

@task
def report():
    filename = f"daily_report_{date.today()}.xlsx"
    print(f"Отчет сформирован: {filename}")

@flow

```

```
def daily_pipeline():
    extract()
    transform()
    report()
if __name__ == "__main__":
    daily_pipeline()

from prefect.schedules import CronSchedule
@flow(schedule=CronSchedule("0 9 * * *"))
def daily_pipeline():
    print("Отчет выполняется по расписанию")
```

5.1.2. Как собирать отчеты автоматически (экспорт из Pandas в Excel, PDF, Google Sheets)

Мы уже создали скрипт `daily_report.py`, который рассчитывает выручку, количество заказов и средний чек по категориям и регионам. Теперь научимся превращать этот результат в полноценные отчетные файлы, которые можно отправлять менеджеру или подгружать в BI-инструменты.

Экспорт в Excel

Формат Excel по-прежнему остается стандартом для отчетов. Руководители любят "открыть и посмотреть", а аналитики — "отфильтровать и сверить". Библиотека `pandas` умеет создавать Excel-файлы "из коробки" с помощью метода `to_excel()`. Чтобы добавить немного оформления (ширину колонок, цвет заголовков), можно использовать библиотеку `openpyxl`, которая автоматически применяется `pandas` при записи `.xlsx`.

Листинг 5.5. Формируем Excel-отчет с форматированием

```
import sqlite3
import pandas as pd
import datetime
from openpyxl import load_workbook
from openpyxl.styles import Font, PatternFill
```



```
# Подключаемся к базе
conn = sqlite3.connect("shop.db")
yesterday = (datetime.date.today() - datetime.timedelta(days=1)).
isoformat()
query = f"""
SELECT region, category, COUNT(*) AS orders_count,
       SUM(amount) AS revenue,
       ROUND(AVG(amount), 2) AS avg_check
FROM orders
WHERE order_date = '{yesterday}'
GROUP BY region, category
ORDER BY revenue DESC
"""

df = pd.read_sql(query, conn)
conn.close()

# Формируем имя файла
filename = f"daily_report_{yesterday}.xlsx"
df.to_excel(filename, index=False)

# Добавляем легкое форматирование
wb = load_workbook(filename)
ws = wb.active

# Заголовки делаем жирными и серыми
for cell in ws[1]:
    cell.font = Font(bold=True, color="FFFFFF")
    cell.fill = PatternFill(start_color="4F81BD", end_color="4F81BD",
fill_type="solid")

# Автоматическая ширина колонок
for column in ws.columns:
    max_length = max(len(str(cell.value)) if cell.value else 0 for
cell in column)
    ws.column_dimensions[column[0].column_letter].width = max_length
+ 2
wb.save(filename)
print(f"Excel-отчет успешно создан: {filename}")
```

Теперь отчет выглядит аккуратно и готов к отправке.

Экспорт в PDF

PDF хорош, если вы хотите, чтобы отчет нельзя было случайно испортить. Например, если отчет нужно отправить заказчику или руководителю, который просто хочет посмотреть итоговые цифры и графики.

Создать PDF можно двумя путями:

1. Сформировать таблицу и график в `matplotlib` и сохранить как PDF.
2. Сгенерировать HTML и конвертировать его в PDF через библиотеку `pdfkit` или `reportlab`.

Листинг 5.6. PDF-отчет с графиком

```
import sqlite3
import pandas as pd
import datetime
import matplotlib.pyplot as plt
from matplotlib.backends.backend_pdf import PdfPages
conn = sqlite3.connect("shop.db")
yesterday = (datetime.date.today() - datetime.timedelta(days=1)).
isoformat()
query = f"""
SELECT category, SUM(amount) AS revenue
FROM orders
WHERE order_date = '{yesterday}'
GROUP BY category
ORDER BY revenue DESC
"""
df = pd.read_sql(query, conn)
conn.close()
plt.figure(figsize=(8,5))
plt.bar(df['category'], df['revenue'], color="#4F81BD")
plt.title(f"Выручка по категориям за {yesterday}")
plt.xlabel("Категория")
plt.ylabel("Выручка, руб.")
plt.grid(True, axis="y", linestyle="--", alpha=0.6)
pdf_filename = f"report_{yesterday}.pdf"
```

```
with PdfPages(pdf_filename) as pdf:
    pdf.savefig() # Сохраняем график
    plt.close()
print(f"PDF-отчет сохранен: {pdf_filename}")
```

Результат — отчет в PDF с графиком, который можно отправить менеджеру. Вы также можете добавить туда несколько страниц: на первой — график, на второй — таблица, на третьей — комментарии.

Экспорт отчета в Google Sheets

Google Таблицы — один из самых удобных способов совместной работы над отчетами. Вы можете автоматически выгружать данные из pandas прямо в Google Sheets с помощью библиотеки `gsread`.

Настройка Google API

Зайдите в Google Cloud Console.

1. Создайте проект и включите API Google Sheets.
2. Создайте Service Account, скачайте JSON-файл с ключом.
3. Расположите данный файл в папку с запускаемым скриптом.

Листинг 5.7. Пример выгрузки в Google Sheets

```
import gsread
from oauth2client.service_account import ServiceAccountCredentials
import pandas as pd
import sqlite3
import datetime

scope = ["https://spreadsheets.google.com/feeds", "https://www.
googleapis.com/auth/drive"]
creds = ServiceAccountCredentials.from_json_keyfile_name("google_key.
json", scope)
client = gsread.authorize(creds)
```

```

# Загружаем данные из базы
conn = sqlite3.connect("shop.db")
yesterday = (datetime.date.today() -
datetime.timedelta(days=1)).isoformat()
query = f"""
SELECT region, category, COUNT(*) AS orders_count,
       SUM(amount) AS revenue,
       ROUND(AVG(amount), 2) AS avg_check
FROM orders
WHERE order_date = '{yesterday}'
GROUP BY region, category
"""

df = pd.read_sql(query, conn)
conn.close()

# Создаем новую таблицу или открываем существующую
spreadsheet = client.open("Daily Reports")
worksheet = spreadsheet.worksheet("Report")

# Очищаем и обновляем данные
worksheet.clear()
worksheet.update([df.columns.values.tolist()] + df.values.tolist())
print("Данные обновлены в Google Sheets!")

```

Теперь каждый день ваш Python-скрипт будет обновлять данные в Google Sheets. Все участники команды увидят свежие цифры без пересылки файлов.

Автоматизация сборки отчетов

Теперь, когда мы умеем создавать отчеты в Excel, PDF и Google Sheets, можно объединить все в единый скрипт.

Листинг 5.8. Автоматизация сборки отчетов

```

import os
import datetime
import sqlite3
import pandas as pd

```

```

import matplotlib.pyplot as plt
from matplotlib.backends.backend_pdf import PdfPages
def build_reports():
    conn = sqlite3.connect("shop.db")
    yesterday = (datetime.date.today() - datetime.
timedelta(days=1)).isoformat()
    query = f"""
    SELECT region, category, COUNT(*) AS orders_count,
           SUM(amount) AS revenue,
           ROUND(AVG(amount), 2) AS avg_check
    FROM orders
    WHERE order_date = '{yesterday}'
    GROUP BY region, category
    ORDER BY revenue DESC
    """
    df = pd.read_sql(query, conn)
    conn.close()
    # Excel
    excel_file = f"daily_report_{yesterday}.xlsx"
    df.to_excel(excel_file, index=False)
    # PDF
    pdf_file = f"report_{yesterday}.pdf"
    plt.figure(figsize=(8, 5))
    plt.bar(df["category"], df["revenue"], color="#4F81BD")
    plt.title(f"Выручка по категориям за {yesterday}")
    plt.ylabel("Выручка, руб.")
    plt.grid(True, axis="y", linestyle="--", alpha=0.6)
    with PdfPages(pdf_file) as pdf:
        pdf.savefig()
        plt.close()
    print(f"Файлы созданы: {excel_file}, {pdf_file}")
if __name__ == "__main__":
    build_reports()

```

Этот скрипт можно вставить в задачу cron или Prefect, и он будет ежедневно формировать все отчеты автоматически.

5.1.3. Отправка отчетов на email и в Telegram-бот

В этой части главы мы разберем, как отправлять отчеты автоматически:

- по электронной почте (через SMTP);
- в Telegram-боте (через API).

Отправка отчета по электронной почте

Email-рассылка — один из самых надежных и понятных способов доставки отчетов. Python позволяет делать это всего в несколько строк, используя встроенные библиотеки `smtplib` и `email.mime`.

Листинг 5.9. Отправка отчета Excel по почте

```
import smtplib
from email.mime.multipart import MIMEMultipart
from email.mime.text import MIMEText
from email.mime.application import MIMEApplication
import datetime
import os

def send_email_report():
    # Данные отправителя
    sender = "analytics@example.com"
    password = "PASSWORD_ЗДЕСЬ"
    receiver = "manager@example.com"
    yesterday = (datetime.date.today() - datetime.
timedelta(days=1)).isoformat()
    report_file = f"daily_report_{yesterday}.xlsx"
    # Проверяем, что файл существует
    if not os.path.exists(report_file):
        print(f"Файл {report_file} не найден.")
        return
    # Создаем письмо
    msg = MIMEMultipart()
    msg["From"] = sender
    msg["To"] = receiver
```

```

msg["Subject"] = f"Ежедневный отчет о продажах - {yesterday}"
body = f"""
Доброе утро!
Во вложении – свежий отчет о продажах за {yesterday}.
Метрики включают:


- Выручку.
- Количество заказов.
- Средний чек.


"""

msg.attach(MIMEText(body, "plain"))
# Прикладываем файл
with open(report_file, "rb") as file:
    attach = MIMEApplication(file.read(), _subtype="xlsx")
    attach.add_header("Content-Disposition", "attachment",
filename=os.path.basename(report_file))
    msg.attach(attach)
# Подключаемся к почтовому серверу
try:
    with smtplib.SMTP("smtp.gmail.com", 587) as server:
        server.starttls()
        server.login(sender, password)
        server.send_message(msg)
        print("Письмо успешно отправлено!")
except Exception as e:
    print(f"Ошибка при отправке письма: {e}")
if __name__ == "__main__":
    send_email_report()

```

Для Gmail нужно разрешить "менее безопасные приложения" или использовать токен OAuth. На корпоративных серверах SMTP-порт может быть 465 (SSL) или 587 (TLS).

Отправка отчета в Telegram

Telegram давно стал не просто мессенджером, а инструментом для уведомлений и командной работы. Многие компании создают отдельные чаты для аналитики — туда автоматически прилетают отчеты, алерты и графики.

Благодаря открытому API Telegram можно отправлять сообщения и файлы прямо из Python.

1. В Telegram найдите пользователя @BotFather.
2. Отправьте ему команду /newbot.
3. Задайте имя и получите токен — он выглядит так:
1234567890:ABCDDefGhIJklMNOpQRstuVWxyz12345678
4. Сохраните токен — он нужен для отправки сообщений.
5. Напишите своему боту любое сообщение.
6. Откройте в браузере ссылку:

https://api.telegram.org/bot<ВАШ_ТОКЕН>/getUpdates

7. В ответе найдите chat: {"id": 123456789} — это ваш chat_id.

Листинг 5.10. Сообщение и файл

```
import requests
import datetime
TOKEN = "ТОКЕН_БОТА"
CHAT_ID = "CHAT_ID"
def send_telegram_report():
    yesterday = (datetime.date.today() - datetime.
timedelta(days=1)).isoformat()
    report_file = f"daily_report_{yesterday}.xlsx"
    message = f"Отчет по продажам за {yesterday} готов!"
    # Отправляем текстовое сообщение
    url_msg = f"https://api.telegram.org/bot{TOKEN}/sendMessage"
    data = {"chat_id": CHAT_ID, "text": message}
    requests.post(url_msg, data=data)
    # Отправляем файл
    url_doc = f"https://api.telegram.org/bot{TOKEN}/sendDocument"
    with open(report_file, "rb") as file:
        requests.post(url_doc, data={"chat_id": CHAT_ID},
files={"document": file})
    print("Отчет отправлен в Telegram!")
if __name__ == "__main__":
    send_telegram_report()
```


Листинг 5.11. Отправляем график как изображение

```
import matplotlib.pyplot as plt
import sqlite3
import pandas as pd
import datetime
import requests
TOKEN = "ВАШ_TOKEN"
CHAT_ID = "ВАШ_CHAT_ID"

def send_sales_chart():
    conn = sqlite3.connect("shop.db")
    yesterday = (datetime.date.today() - datetime.
timedelta(days=1)).isoformat()
    df = pd.read_sql(f"""
SELECT category, SUM(amount) AS revenue
FROM orders
WHERE order_date = '{yesterday}'
GROUP BY category
ORDER BY revenue DESC
""", conn)
    conn.close()
    plt.figure(figsize=(8,5))
    plt.bar(df["category"], df["revenue"], color="#4F81BD")
    plt.title(f"Выручка по категориям за {yesterday}")
    plt.ylabel("Выручка, руб.")
    plt.grid(True, axis="y", linestyle="--", alpha=0.6)
    chart_file = f"sales_chart_{yesterday}.png"
    plt.savefig(chart_file)
    plt.close()
    url = f"https://api.telegram.org/bot{TOKEN}/sendPhoto"
    with open(chart_file, "rb") as photo:
        requests.post(url, data={"chat_id": CHAT_ID}, files={"photo":
photo})
    print("График отправлен в Telegram!")
if __name__ == "__main__":
    send_sales_chart()
```

Можно объединить все в один главный сценарий, который будет запускаться автоматически каждый день (через cron или Prefect).

Листинг 5.12. Финальный скрипт: автоматический ежедневный отчет

```
import os
import datetime
import sqlite3
import pandas as pd
import matplotlib.pyplot as plt
import requests
import smtplib
from email.mime.multipart import MIMEMultipart
from email.mime.application import MIMEApplication
from email.mime.text import MIMEText
TOKEN = "BAШ_TOKEN"
CHAT_ID = "BAШ_CHAT_ID"
SENDER = "analytics@example.com"
PASSWORD = "PASSWORD"
RECEIVER = "manager@example.com"
def build_report():
    conn = sqlite3.connect("shop.db")
    yesterday = (datetime.date.today() - datetime.
timedelta(days=1)).isoformat()
    query = f"""
    SELECT region, category, COUNT(*) AS orders_count,
           SUM(amount) AS revenue,
           ROUND(AVG(amount), 2) AS avg_check
    FROM orders
    WHERE order_date = '{yesterday}'
    GROUP BY region, category
    ORDER BY revenue DESC
    """
    df = pd.read_sql(query, conn)
    conn.close()
    excel_file = f"daily_report_{yesterday}.xlsx"
    df.to_excel(excel_file, index=False)
```

```

print("Отчет создан:", excel_file)
return excel_file, yesterday
def send_email(file, date):
    msg = MIMEMultipart()
    msg["From"] = SENDER
    msg["To"] = RECEIVER
    msg["Subject"] = f"Ежедневный отчет о продажах - {date}"
    msg.attach(MIMEText("Во вложении свежий отчет о продажах."))
    with open(file, "rb") as f:
        part = MIMEApplication(f.read(), _subtype="xlsx")
        part.add_header("Content-Disposition", "attachment",
filename=os.path.basename(file))
        msg.attach(part)
    with smtplib.SMTP("smtp.gmail.com", 587) as server:
        server.starttls()
        server.login(SENDER, PASSWORD)
        server.send_message(msg)
        print("Отчет отправлен по email")
def send_telegram(file, date):
    msg = f"Отчет о продажах за {date} готов!"
    requests.post(
        f"https://api.telegram.org/bot{TOKEN}/sendMessage",
        data={"chat_id": CHAT_ID, "text": msg}
    )
    with open(file, "rb") as f:
        requests.post(
            f"https://api.telegram.org/bot{TOKEN}/sendDocument",
            data={"chat_id": CHAT_ID},
            files={"document": f}
        )
    print("Отчет отправлен в Telegram")
if __name__ == "__main__":
    file, date = build_report()
    send_email(file, date)
    send_telegram(file, date)

```

Можно добавить этот скрипт в cron, чтобы он запускался каждый день:

```
0 9 * * * /usr/bin/python3 /home/user/projects/auto_report.py
```

5.1.4. Итоги раздела

Мы прошли весь путь от создания простого отчета до его полной автоматизации:

- ◆ Научились планировать задачи — запускать Python-скрипты по расписанию с помощью cron, а также познакомились с современными инструментами оркестрации — Airflow и Prefect, которые позволяют управлять целыми аналитическими конвейерами.
- ◆ Разобрались, как собирать отчеты автоматически: формировать результаты в pandas и экспортировать их в разные форматы — Excel (с форматированием), PDF (с графиками) и Google Sheets (через API для совместной работы).
- ◆ Освоили автоматическую рассылку отчетов — по email и в Telegram-боте.

5.2. Дашборды и визуальная аналитика

5.2.1. Основы построения дашбордов

Дашборд (dashboard) — это визуальный инструмент, который помогает анализировать данные и принимать решения на основе ключевых показателей.

Название пришло из автомобильной тематики: так же как приборная панель в машине показывает скорость, уровень топлива и температуру, аналитический дашборд показывает метрики бизнеса — выручку, количество клиентов, конверсию, затраты.

Главная цель дашборда — сделать данные понятными и доступными. Он не просто показывает цифры, а помогает увидеть тенденции, проблемы и возможности для улучшений.

Хороший дашборд должен отвечать на три простых вопроса:

1. Что происходит?
2. Почему это происходит?
3. Что нужно сделать дальше?

Дашборды помогают:

- **Следить за ключевыми метриками в реальном времени.** Например, продажи по регионам, загрузка склада, количество новых пользователей.
- **Выявлять тренды и отклонения.** Сразу видно, если метрика изменилась: "Выручка упала на 12% по сравнению с прошлой неделей".
- **Автоматизировать отчетность.** Не нужно вручную собирать цифры из Excel — дашборд обновляется автоматически из базы данных или API.
- **Объединять команду вокруг общих целей.** Руководитель, маркетолог и аналитик видят одни и те же данные и говорят "на одном" языке.

Принципы хорошего дашборда

Создать дашборд — не значит просто нарисовать несколько графиков. Хорошая визуализация должна быть функциональной, логичной и понятной.

Принцип 1. Фокус на цели

Перед началом работы задайте вопрос: "Кто будет смотреть этот дашборд и для чего?" Дашборд для директора должен быть простым и лаконичным: 3–5 ключевых показателей (KPI). Для аналитика — наоборот, детальным, с возможностью фильтрации и детализации.

Пример:

- » Стратегический дашборд: выручка, прибыль, средний чек, ROI.
- » Операционный дашборд: заказы по регионам, отказы, загрузка склада, скорость доставки.

Принцип 2. Простота и минимализм

Дашборд должен быть интуитивно понятным без пояснений. Один взгляд — и ясно, где рост, где падение.

Правила визуального дизайна:

- » Используйте спокойные цвета (серые, синие, зеленые).
- » Избегайте перегруженности (не больше 6–8 графиков на одном экране).
- » Не злоупотребляйте 3D и сложными эффектами — они мешают восприятию.
- » Упрощайте: удаляйте все, что не помогает принять решение.

Принцип 3. Логическая структура

Хороший дашборд читается сверху вниз и слева направо. Расположите блоки так, чтобы взгляд естественно следовал от общего к частному:

1. KPI (основные показатели) — крупные числа вверху.
2. Динамика — графики изменений во времени.
3. Детализация — таблицы и категории.

Принцип 4. Единая система измерений

Все данные должны быть представлены в одной системе координат:

- » одна валюта (например, рубли);
- » один формат чисел (разделители тысяч, округление);
- » согласованные даты (например, неделя начинается с понедельника).

Принцип 5. Контекст и сравнительный анализ

Показатель без контекста ничего не значит. Например, выручка 1 000 000 Р — это хорошо или плохо? Зависит от того, что было раньше.

Добавьте сравнение с прошлым периодом:

- » "+15% к прошлому месяцу";
- » "–7% от прошлого года".

Типы дашбордов

Тип дашборда	Назначение	Частота обновления	Пример
Операционный	Мониторинг текущих процессов	Ежечасно, ежедневно	Отчет о продажах за день
Аналитический	Изучение тенденций и причин	Раз в неделю, месяц	Анализ выручки по регионам
Стратегический	Контроль достижения целей	Раз в месяц, квартал	КРІ компании, прибыль, маржинальность

Какие графики использовать

Выбор визуализации зависит от типа данных и вопроса, на который вы хотите ответить.

Цель	Подходит визуализация	Пример
Показать изменение со временем	Линейный график	Выручка по месяцам
Сравнить категории	Столбчатая диаграмма	Продажи по регионам
Показать структуру	Круговая диаграмма	Доли выручки по категориям
Показать распределение	Boxplot или Histogram	Распределение среднего чека
Показать взаимосвязь	Scatter plot (точечная диаграмма)	Связь цены и количества продаж

Цвета и визуальные акценты

Цвет — мощный инструмент, но им легко злоупотребить. Используйте цвет для выделения, а не для украшения.

Рекомендации:

- Основные данные — нейтральные цвета (серый, синий).
- Акценты — яркие (оранжевый, зеленый, красный).
- Изменения и рост — зеленый, падение — красный.
- Не более 5–6 цветов на одном экране.

Пример хорошего сочетания для аналитики:

```
palette = ["#1f77b4", "#4f81bd", "#aec7e8", "#2ca02c", "#98df8a"]
```

Ошибки при создании дашбордов

- Слишком много графиков — пользователю трудно сосредоточиться.
- Отсутствие фокуса — непонятно, что важно.
- Неправильный масштаб осей — искажает восприятие.
- Избыточные детали — таблицы на 50 строк не нужны на дашборде.
- Без контекста — нет сравнений и пояснений.

Инструменты для создания дашбордов

1. Python + Plotly / Streamlit / Dash — идеальны для разработчиков и аналитиков, кто работает с кодом.
2. Power BI / Tableau / Google Data Studio — визуальные решения без программирования.
3. Excel + Power Query / Pivot — базовый вариант для небольших отчетов.

В следующей части главы мы применим эти принципы на практике — создадим интерактивный дашборд в Python с помощью Plotly и Streamlit.

5.2.2. Создание интерактивного дашборда на Python: Plotly + Dash/Streamlit

Когда вы показываете графики в Jupyter или Excel, это хорошо для анализа, но не для дальнейшей дистрибуции дашборда.

Для дальнейшей работы с дашбордом у пользователя должен быть установлен Jupyter либо Excel, а также для работы с ним должны быть хоть какие-то минимальные знания этих продуктов.

Пользователи дашборда не могут:

- поменять дату;
- выбрать категорию товара;
- сравнить два региона.

Интерактивный дашборд решает эту задачу. Он превращает статический отчет в живое приложение, где можно кликать, фильтровать и видеть обновленные данные в реальном времени.

Python сегодня предлагает два главных инструмента для создания таких решений:

- **Dash (от Plotly)** — мощный фреймворк для веб-дашбордов с гибкой настройкой интерфейса;
- **Streamlit** — более легкий и "человеческий" вариант, который позволяет создавать интерактивные отчеты буквально за несколько строк кода.

Подготовка данных

Для наглядности создадим мини-набор данных по продажам интернет-магазина за 3 месяца.

```
import pandas as pd
import numpy as np
import datetime
import random
np.random.seed(42)
dates = pd.date_range("2025-07-01", "2025-09-30")
categories = ["Электроника", "Одежда", "Книги", "Дом", "Спорт"]
regions = ["Москва", "СПб", "Казань", "Новосибирск", "Екатеринбург"]
data = []
for date in dates:
    for _ in range(random.randint(50, 150)):
        data.append({
            "date": date,
```

```

        "category": random.choice(categories),
        "region": random.choice(regions),
        "amount": round(random.uniform(500, 20000), 2)
    })
df = pd.DataFrame(data)
df.to_csv("sales_data.csv", index=False)
print("Данные созданы:", df.shape)
print(df.head())

```

Теперь у нас есть исходные данные для построения дашборда.

Интерактивный дашборд в Streamlit

Streamlit — это библиотека, которая превращает обычный Python-скрипт в интерактивное приложение. Ее главное преимущество — простота.

Сохраним код в файл `dashboard.py`:

Листинг 5.13. Интерактивный дашборд продаж в streamlit

```

import streamlit as st
import pandas as pd
import plotly.express as px
@st.cache_data
def load_data():
    return pd.read_csv("sales_data.csv", parse_dates=["date"])
df = load_data()
st.title("Дашборд продаж интернет-магазина")
st.markdown("Интерактивный отчет с фильтрами по категории, региону и периоду")
# Фильтры
categories = st.multiselect("Выберите категории:", df["category"].unique(), default=df["category"].unique())
regions = st.multiselect("Выберите регионы:", df["region"].unique(), default=df["region"].unique())
start_date = st.date_input("Дата начала", df["date"].min())

```

```

end_date = st.date_input("Дата конца", df["date"].max())
# применяем фильтры
filtered_df = df[
    (df["category"].isin(categories)) &
    (df["region"].isin(regions)) &
    (df["date"] >= pd.Timestamp(start_date)) &
    (df["date"] <= pd.Timestamp(end_date))
]
# Ключевые метрики
total_revenue = filtered_df["amount"].sum()
total_orders = len(filtered_df)
avg_check = filtered_df["amount"].mean()
st.subheader("Ключевые показатели")
col1, col2, col3 = st.columns(3)
col1.metric("Выручка", f"{total_revenue:,.0f} ₽")
col2.metric("Количество заказов", f"{total_orders:,}")
col3.metric("Средний чек", f"{avg_check:,.0f} ₽")
# График: выручка по датам
st.subheader("Динамика продаж")
daily = filtered_df.groupby("date")["amount"].sum().reset_index()
fig1 = px.line(daily, x="date", y="amount", title="Выручка по дням",
markers=True, color_discrete_sequence=["#1f77b4"])
st.plotly_chart(fig1, use_container_width=True)
# График: выручка по категориям
st.subheader("Продажи по категориям")
category_revenue = filtered_df.groupby("category")["amount"].sum().
reset_index()
fig2 = px.bar(category_revenue, x="category", y="amount",
title="Выручка по категориям", color="category", color_discrete_
sequence=px.colors.sequential.Blues)
st.plotly_chart(fig2, use_container_width=True)
# График: выручка по регионам
st.subheader("Выручка по регионам")
region_revenue = filtered_df.groupby("region")["amount"].sum().reset_
index()
fig3 = px.pie(region_revenue, names="region", values="amount",
title="Распределение выручки по регионам", color_discrete_
sequence=px.colors.sequential.Blues_r)
st.plotly_chart(fig3, use_container_width=True)

```

Далее в терминале:

```
streamlit run dashboard.py
```

После этого Streamlit откроет веб-страницу, где вы увидите дашборд:

- фильтры по категориям и регионам,
- KPI в верхней части,
- графики и диаграммы.



Изображение 5.1. Дашборд продаж в streamlit

Все интерактивно: выберите нужный регион или категорию, и графики сразу перестроятся.

Интерактивный дашборд в Dash (Plotly)

Если Streamlit — это "ноутбук, превращенный в веб", то Dash — это уже веб-приложение уровня компании. В Dash можно настраивать layout, объединять компоненты и даже добавлять HTML/CSS для кастомного интерфейса.

Листинг 5.14. Интерактивный дашборд продаж в Dash

```
from dash import Dash, html, dcc, Output, Input
import plotly.express as px
import pandas as pd
df = pd.read_csv("sales_data.csv", parse_dates=["date"])
app = Dash(__name__)
app.title = "Дашборд продаж"
# layout – структура приложения
app.layout = html.Div([
    html.H1("📊 Аналитика продаж интернет-магазина",
style={"textAlign": "center"}),
    html.Div([
        html.Label("Категория:"),
        dcc.Dropdown(
            options=[{"label": c, "value": c} for c in
df["category"].unique()],
            value=df["category"].unique().tolist(),
            id="category-filter",
            multi=True
        )
    ], style={"width": "45%", "display": "inline-block"}),
    html.Div([
        html.Label("Регион:"),
        dcc.Dropdown(
            options=[{"label": r, "value": r} for r in df["region"].
unique()],
            value=df["region"].unique().tolist(),
            id="region-filter",
            multi=True
        )
    ])
```

```

], style={"width": "45%", "display": "inline-block", "float":
"right"}),
    dcc.Graph(id="sales-line"),
    dcc.Graph(id="category-bar"),
])
# callbacks — логика обновления
@app.callback(
    [Output("sales-line", "figure"),
     Output("category-bar", "figure")],
    [Input("category-filter", "value"),
     Input("region-filter", "value")]
)
def update_dashboard(selected_categories, selected_regions):
    filtered = df[
        (df["category"].isin(selected_categories)) &
        (df["region"].isin(selected_regions))
    ]
    # Выручка по датам
    daily = filtered.groupby("date")["amount"].sum().reset_index()
    fig1 = px.line(daily, x="date", y="amount", title="Выручка по
дням", markers=True, color_discrete_sequence=["#1f77b4"])
    # Выручка по категориям
    category_rev = filtered.groupby("category")["amount"].sum().
reset_index()
    fig2 = px.bar(category_rev, x="category", y="amount",
title="Выручка по категориям", color="category", color_discrete_
sequence=px.colors.sequential.Blues)
    return fig1, fig2
if __name__ == "__main__":
    app.run(debug=True)

```

Как это работает:

- Вы создаете `layout` — HTML-структуру (заголовки, фильтры, графики).
- Затем описываете `callback` — функцию, которая обновляет графики при изменении фильтров.
- Dash автоматически связывает фильтры и визуализацию.

Dash подходит, если вы хотите развернуть дашборд на сервере, добавить авторизацию, интегрировать API или подключить базу данных.

5.2.3. Итоги раздела

- ◆ Мы познакомились с инструментами для создания интерактивных дашбордов на Python — Streamlit и Dash.
- ◆ Научились превращать статические отчеты в живые аналитические приложения с фильтрами, графиками и показателями KPI.
- ◆ Разобрали примеры подключения данных, настройки фильтров и обновления графиков в реальном времени.
- ◆ Поняли различия между инструментами:
 - Streamlit — простой и быстрый способ собрать прототип отчета;
 - Dash — мощный фреймворк для корпоративных дашбордов с гибкой кастомизацией.

5.3. Очистка и подготовка данных

Очистка данных — это первый и самый важный шаг перед любым анализом или построением моделей. Если данные содержат ошибки, пропуски или логические несоответствия, любые выводы, построенные на них, будут некорректными. Это так называемый принцип GIGO: "Garbage in, garbage out" ("мусор внутрь, мусор наружу"). Если на вход поступают "грязные" данные, то и результат будет таким же.

Типичные проблемы реальных данных

С какими проблемами сталкиваются чаще всего:

1. Пропущенные значения.
2. Выбросы и ошибки ввода.

3. Неоднородность форматов: Москва, москва, МОСКВА — одно и то же, но программа посчитает это разными значениями.
4. Смешанные типы данных: числа хранятся как строки: "1200" вместо 1200.
5. Дубликаты записей: один и тот же пользователь или заказ может встречаться несколько раз.

Мы будем исправлять каждую из этих проблем шаг за шагом далее, но начнем с самого частого — пропусков.

5.3.1. Обнаружение пропусков

В `pandas` пропущенные значения обозначаются как:

- `NaN` — Not a Number.
- `None` — пустое значение в Python.

Оба распознаются одинаково при анализе.

Создадим небольшой набор данных:

```
import pandas as pd
data = {
    "name": ["Иван", "Анна", "Петр", None, "Елена"],
    "age": [29, None, 41, 35, None],
    "city": ["Москва", "Казань", "Москва", "Сочи", None]
}
df = pd.DataFrame(data)
df
```

Результат:

	name	age	city
0	Иван	29.0	Москва
1	Анна	NaN	Казань
2	Петр	41.0	Москва
3	None	35.0	Сочи
4	Елена	NaN	None

Мы видим пропуски визуально, но для дальнейшей очистки нужно уметь находить их программно.

Проверка наличия пропусков

Проверить, есть ли пропуски вообще:

```
df.isnull().any()
```

Посчитать количество пропусков по каждому столбцу:

```
df.isnull().sum()
```

Посмотреть долю пропусков:

```
df.isnull().mean()
```

Диагностическая информация о датафрейме

Команда `info()` дает сводку по колонкам, типам данных и количеству непустых значений:

```
df.info()
```

Пример вывода:

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 5 entries, 0 to 4
Data columns (total 3 columns):
 name    4 non-null object
 age     3 non-null float64
 city    4 non-null object
```

Мы сразу видим:

- в `name` пропущено 1 значение,
- в `age` — 2,
- в `city` — 1.

Обнаружение пропусков в отдельных строках

Например, вывести только строки, где есть хотя бы один пропуск:

```
df[df.isnull().any(axis=1)]
```

Или, наоборот, строки без пропусков:

```
df[df.notnull().all(axis=1)]
```

5.3.2. Работа с пропущенными значениями

После того как мы обнаружили пропуски, необходимо решить, что с ними делать. Вариантов несколько: удалить, заполнить или восстановить. Нет универсального метода — выбор зависит от задачи и природы данных.

Основной вопрос, который нужно задавать себе: "Я теряю информацию или восстанавливаю ее?"

Удаление пропусков

Удаление строк подходит только тогда, когда:

- пропусков мало,
- они не несут важной информации,
- восстановить значения сложно или невозможно.

```
df_clean = df.dropna()
```

Удаление целых столбцов — более редкий, но возможный вариант:

```
df_drop_col = df.dropna(axis=1)
```

Иногда удаляют только строки с пропусками в конкретных колонках:

```
df_age_only = df.dropna(subset=["age"])
```

Заполнение числовых признаков

Если пропуск в числовом столбце, можно заполнить значение разными способами.

Заполнение средним:

```
df["age"] = df["age"].fillna(df["age"].mean())
```

Хорошо работает для симметричных распределений.

Заполнение медианой:

```
df["age"] = df["age"].fillna(df["age"].median())
```

Медиана устойчива к выбросам, поэтому предпочтительнее в большинстве реальных случаев.

Заполнение значением из похожей группы.

```
df["age"] = df.groupby("city")["age"].transform(
    lambda x: x.fillna(x.median())
)
```

Это более "умное" заполнение: пропуск восстанавливается на основе контекста.

Заполнение категориальных признаков

Для категорий нельзя использовать среднее или медиану — значения не числовые. Самый простой метод — заменить пропуски модой (наиболее частым значением):

```
df["city"] = df["city"].fillna(df["city"].mode()[0])
```

Иногда вводят отдельную категорию "Unknown", чтобы не скрывать сам факт того, что значение было пропущено:

```
df["city"] = df["city"].fillna("Unknown")
```

Это полезно в моделях: пропуск сам по себе может быть информативным признаком.

Интерполяция (для временных рядов)

Если данные меняются постепенно (например, продажи по дням или температура), можно "восстановить" пропуски по тренду.

```
df["age"] = df["age"].interpolate()
```

Pandas интерполирует значения линейно, но можно указать метод:

```
df["age"] = df["age"].interpolate(method="polynomial", order=2)
```

Заполнение на основе модели

Иногда пропуски заполняют как задачу предсказания. Это уже более продвинутый подход.

```
from sklearn.linear_model import LinearRegression
train = df[df["age"].notnull()]
test = df[df["age"].isnull()]
model = LinearRegression()
model.fit(train[["some_feature"]], train["age"])
df.loc[df["age"].isnull(), "age"] = model.predict(test[["some_
feature"]])
```

Как выбрать стратегию заполнения:

- Если пропусков мало — удаляем строки.
- Если данные симметричны — заполняем средним.
- Если есть выбросы — заполняем медианой.
- Если значение зависит от группы — групповая медиана / среднее.
- Если данные временные — интерполяция.
- Если пропуск сам информативен — вводим категорию "Unkown".

Проверка результата

После заполнения пропусков всегда проверяем:

```
df.isnull().sum()
```

И визуально:

```
print(df.head())
```

Если структура изменилась:

```
df.info()
```

5.3.3. Обнаружение выбросов

Выбросы — это наблюдения, которые существенно отклоняются от основной массы данных.

Они могут появиться по разным причинам:

- ошибка ввода (возраст = 350, стоимость = 999999999);
- технический сбой;
- редкие, но реальные события (например, покупка на очень крупную сумму).

Важно понимать: выброс $\neq$ ошибка. Иногда выброс — это ценная особенность поведения, и удалять его нельзя.

Проверка распределения визуально

Начнем с простого графика — `boxplot`:

```
import pandas as pd
import matplotlib.pyplot as plt
df = pd.DataFrame({"amount": [500, 550, 610, 620, 590, 10000]})
plt.boxplot(df["amount"])
plt.show()
```

Если на графике появляется точка далеко от "ящика" — это кандидат на выброс.

Для предварительного исследования часто используют и гистограмму:

```
df["amount"].hist(bins=20)
plt.show()
```

Если распределение "сильно вытянуто" — это сигнал.

Обнаружение выбросов с помощью IQR (межквартильный размах)

Метод называется статистическим образом простым:

- Находим Q1 — 25-й процентиль.
- Находим Q3 — 75-й процентиль.
- Считаем $IQR = Q3 - Q1$.
- Значения ниже $Q1 - 1.5 * IQR$ и выше $Q3 + 1.5 * IQR$ считаются выбросами.

```
Q1 = df["amount"].quantile(0.25)
Q3 = df["amount"].quantile(0.75)
IQR = Q3 - Q1

lower = Q1 - 1.5 * IQR
upper = Q3 + 1.5 * IQR
outliers = df[(df["amount"] < lower) | (df["amount"] > upper)]
outliers
```

Если нужно вывести строки без выбросов:

```
df_no_outliers = df[(df["amount"] >= lower) & (df["amount"] <= upper)]
```

Обнаружение выбросов с помощью Z-Score

Метод хорошо работает, если распределение близко к нормальному.

Вычисляем, насколько значение отклоняется от среднего:

```
from scipy.stats import zscore
df["z"] = zscore(df["amount"])
```

Посмотрим подозрительные значения:

```
outliers = df[df["z"].abs() > 3]
outliers
```

Число 3 — классический порог, исходя из Z-распределения.

Обнаружение выбросов в группах

Иногда выброс определяется в контексте группы, а не по всей выборке.

Например, стоимость заказа в "Москве" и "Сочи" распределена по-разному:

```
df["group_outlier"] = df.groupby("city")["amount"].transform(
    lambda x: (abs((x - x.mean()) / x.std()) > 3)
)
```

Такой подход особенно полезен для анализа:

- бизнес-сегментов,
- городов,
- категорий товаров.

Быстрая оценка распределения после удаления/исправления

```
df_no_outliers["amount"].describe()
df_no_outliers["amount"].hist(bins=30)
plt.show()
```

А также:

```
plt.boxplot(df_no_outliers["amount"])
plt.show()
```

Если распределение "выпрямилось" — очистка прошла корректно.

5.3.4. Обработка выбросов

После того как мы научились обнаруживать выбросы, необходимо решить, что с ними делать. И здесь важно помнить главное правило: нельзя просто удалять выбросы автоматически, нужно понять природу этих значений.

Выброс может быть:

- ошибкой, тогда его нужно исправить или удалить;

- редким, но важным событием, тогда его нужно сохранить;
- следствием измененного масштаба, и тогда достаточно преобразования признака.

Удаление выбросов

Используется, когда выбросы — явные ошибки, например:

- возраст: 300,
- количество заказов: -5,
- доход: 9999999999.

Если выбросы определены через IQR:

```
df_clean = df[(df["amount"] >= lower) & (df["amount"] <= upper)]
```

Где `lower` и `upper` — как в предыдущем разделе.

Если выбросы найдены через Z-score:

```
df_clean = df[df["z"].abs() <= 3]
```

Важно: убедитесь, что данные после удаления все еще репрезентативны. Если удаляется > 5–10% наблюдений — лучше выбрать другой метод.

Ограничение выбросов (Winsorization)

Вместо удаления можно обрезать значения до допустимого диапазона. Это сохраняет количество строк и не ломает статистику.

```
df["amount_clipped"] = df["amount"].clip(lower, upper)
```

Если `lower` = 100 и `upper` = 2000, то все, что меньше 100, — 100, а все, что больше 2000, — 2000. Хорошо работает в бизнес-аналитике, особенно в отчетах.

Логарифмирование

Если показатель распределен сильно асимметрично (например, суммы покупок, доходы, стоимость товаров), то вместо борьбы с "хвостом" распределения можно сжать шкалу.


```
import numpy as np
df["amount_log"] = np.log1p(df["amount"])
df["amount"].hist(bins=30)
plt.show()
df["amount_log"].hist(bins=30)
plt.show()
```

Очень часто после логарифмирования распределение становится ближе к нормальному.

Преобразование степенями (корневые и степенные функции)

- Квадратный корень — сглаживает большой разброс:

```
df["amount_sqrt"] = np.sqrt(df["amount"])
```

- Степени и обратные функции подходят, когда есть гипотеза о нелинейности.

Замена выбросов на групповые статистики

Если данные сильно зависят от категорий, например, "Москва" и "Сочи" разные по среднему чеку:

```
df["amount_corrected"] = df.groupby("city")["amount"].transform(
    lambda x: x.clip(x.quantile(0.05), x.quantile(0.95))
)
```

Это более точечный и бизнес-корректный метод.

Моделирование выбросов

Редкий, но самый интеллектуальный подход — обучить модель, которая предсказывает ожидаемое значение, и заменить им выброс.

Пример (очень упрощенный):

```
from sklearn.ensemble import RandomForestRegressor
df_train = df_no_outliers.copy()
```

```
df_outliers = outliers.copy()
model = RandomForestRegressor()
model.fit(df_train[["feature1", "feature2"]], df_train["amount"])
df_outliers["amount"] = model.predict(df_outliers[["feature1",
"feature2"]])
```

Используется в продвинутых аналитических системах.

Быстрая проверка результата

После обработки выбросов обязательно смотрим:

```
df["amount"].describe()
df["amount"].hist(bins=30)
plt.show()
```

Если данные стали "ближе" к репрезентативным — все сделано верно

5.3.5. Преобразование масштаба признаков

Многие модели и аналитические методы предполагают, что признаки находятся в сопоставимом масштабе. Если один признак измеряется в тысячах, а другой в единицах, модели могут:

- переоценивать признаки с большим масштабом;
- хуже обучаться;
- сходиться медленнее;
- давать неустойчивые предсказания.

Например:

```
salary = [50000, 60000, 70000]
age = [25, 26, 27]
```

Если использовать их "как есть", модели могут считать, что зарплата важнее возраста просто потому, что числа больше, хотя это не обязательно так. Поэтому признаки масштабируют (нормализуют или стандартизируют).

Стандартизация (Standardization, Z-score scaling)

Стандартизация преобразует признак так, что:

- среднее становится 0;
- стандартное отклонение становится 1.

Это используется в следующих моделях:

- линейная регрессия;
- логистическая регрессия;
- метод опорных векторов (SVM);
- KNN;
- PCA (анализ главных компонент).

Пример:

```
import pandas as pd
from sklearn.preprocessing import StandardScaler
df = pd.DataFrame({
    "age": [25, 30, 45, 50],
    "salary": [50000, 60000, 90000, 120000]
})
scaler = StandardScaler()
df_scaled = df.copy()
df_scaled[["age", "salary"]] = scaler.fit_transform(df[["age", "salary"]])
print(df_scaled)
```

Результат:

	age	salary
0	-1.183216	-1.183216
1	-0.507093	-0.507093
2	0.507093	0.507093
3	1.183216	1.183216

Оба признака теперь в одном масштабе.

Нормализация Min-Max

Переводит значения в диапазон от 0 до 1.

Этот метод полезен для:

- нейронных сетей;
- визуализации;
- моделей, чувствительных к расстояниям (KNN).

Пример:

```
from sklearn.preprocessing import MinMaxScaler
scaler = MinMaxScaler()
df_minmax = df.copy()
df_minmax[["age", "salary"]] = scaler.fit_transform(df[["age", "salary"]])
print(df_minmax)
```

Результат:

	age	salary
0	0.00	0.00
1	0.17	0.17
2	0.67	0.67
3	1.00	1.00

Когда использовать какой метод

Если данные примерно нормально распределены — используется стандартизация (StandardScaler). Если данные ограничены диапазоном или модель чувствительна к шагу — нормализация (MinMaxScaler). Если распределение сильно скошено, сначала применяем логарифмирование, затем стандартизацию.

Пример: сочетание логарифмирования и масштабирования

Допустим, у нас есть доходы, где есть большой "хвост".

```
import numpy as np
df["salary_log"] = np.log1p(df["salary"])
df["salary_scaled"] = (df["salary_log"] - df["salary_log"].mean()) /
df["salary_log"].std()
```

Теперь данные:

- сгладились;
- стали ближе к нормальному распределению;
- приведены к единому масштабу.

Проверка результата:

```
df.describe()
```

Также полезно сравнить гистограммы до и после:

```
df["salary"].hist(bins=30)
plt.show()

df["salary_scaled"].hist(bins=30)
plt.show()
```

Чаще всего вторая гистограмма выглядит гораздо "ровнее".

5.3.6. Итоги раздела

В этой главе мы разобрали ключевые шаги подготовки данных к анализу и моделям:

- ◆ поняли, почему качество входных данных напрямую влияет на корректность выводов ("Garbage in, garbage out");
- ◆ научились находить пропуски и определять их природу;
- ◆ рассмотрели разные стратегии обработки пропусков: удаление, заполнение средними/медианами, по группам, интерполяцией;
- ◆ научились обнаруживать выбросы визуально и статистически (IQR, Z-score);
- ◆ разобрались, как корректно обрабатывать выбросы: удаление, ограничение, логарифмирование, групповые корректировки;
- ◆ освоили методы масштабирования признаков — стандартизацию и Min-Max нормализацию;
- ◆ поняли, в каких ситуациях использовать логарифмирование перед нормализацией.

5.4. Инженерия признаков

Когда данные уже очищены и приведены в корректный вид, возникает следующий вопрос: какие признаки использовать в анализе или модели?

Даже если данные точные, сами по себе они могут быть слабо информативны. Грамотно построенные признаки способны существенно повысить объясняющую и предсказательную силу модели.

Существует известное практическое правило: 80% качества модели определяется качеством признаков, и только 20% — выбором алгоритма.

Это легко увидеть на примере.

Допустим, у нас есть данные о заказах:

```
import pandas as pd
df = pd.DataFrame({
    "price": [1000, 2000, 1500, 2500],
    "quantity": [1, 2, 1, 4]
})
```

Если мы обучим модель на этих колонках, она будет предсказывать что-то вроде:

```
revenue ≈ a * price + b * quantity
```

А теперь добавим новый признак, который напрямую отражает бизнес-смысл:

```
df["revenue"] = df["price"] * df["quantity"]
```

Модель сразу получает существенно больше информации и становится проще и точнее. Признак "выручка" появился не магически, а был сконструирован человеком. Это и есть инженерия признаков.

Модель не "понимает" смысл данных. Она опирается только на числа и их взаимосвязи. Наша задача — подготовить такие признаки, которые помогут модели выявить закономерности.

Некоторые признаки существуют в данных явно (например: возраст, цена, количество). Но часть информации скрыта и может быть извлечена.

Например, поле с датой:

```
df = pd.DataFrame({
    "date": pd.to_datetime(["2024-01-01", "2024-01-02", "2024-06-15"])
})
```

Пока это просто дата — модель почти ничего не понимает. Но мы можем "распаковать" признак:

```
df["day_of_week"] = df["date"].dt.dayofweek
df["month"] = df["date"].dt.month
df["is_weekend"] = df["date"].dt.dayofweek >= 5
```

Теперь модель видит:

- день недели,
- месяц,
- выходной или нет.

Это превращает данные в понятную структуру признаков, которая уже несет информацию.

Даже в обычной аналитике признаки критичны.

Например:

- "количество заказов в месяц" важнее "количества заказов за все время";
- "расстояние до ближайшего филиала" объясняет поведение клиента лучше, чем просто "адрес";
- "отношение возвратов к покупкам" важнее, чем просто числа "возвраты" и "покупки".

То есть мы не просто "подставляем данные в формулы", а строим представление сущностей — клиентов, продуктов, транзакций.

Простой пример, который меняет интерпретацию данных:

```
df = pd.DataFrame({
    "user_id": [1, 1, 1, 2, 2],
    "purchase_amount": [1200, 800, 500, 3000, 200],
})
```

Создадим признак активности пользователя:

```
user_features = df.groupby("user_id")["purchase_amount"].agg(
    total_spent="sum",
    mean_spent="mean",
    purchases="count"
)
print(user_features)
```

Мы получаем для аналитики и моделирования:

- суммарную выручку пользователя,
- средний чек,
- количество покупок.

Эти признаки дают сегментацию, даже без модели.

5.4.1. Бакетирование (дискретизация) числовых признаков

Бакетирование (или дискретизация, *binning*) — это преобразование непрерывного числового признака в категориальный с разбиванием значения на интервалы (бакеты).

Это может быть полезно, когда:

- модель или анализ должны работать с категориями;
- распределение признака сильно асимметрично;
- небольшие различия в числах не важны, а важна группа;
- нужно выделить сегменты пользователей.

Пример: если у нас есть возраст, нам часто не важны 31 vs 32. Но разница между 18 и 55 — принципиальна.

```
import pandas as pd
df = pd.DataFrame({
    "age": [18, 21, 25, 30, 35, 45, 52, 70, 82]
})
```


Бакетирование фиксированными интервалами (pd.cut)

Задаем интервалы вручную:

```
df["age_group"] = pd.cut(df["age"], bins=[0, 25, 45, 65, 100])
print(df)
```

Результат будет примерно таким:

	age	age_group
0	18	(0, 25]
1	21	(0, 25]
2	25	(0, 25]
3	30	(25, 45]
4	35	(25, 45]
5	45	(25, 45]
6	52	(45, 65]
7	70	(65, 100]
8	82	(65, 100]

Интервалы выражены математически. Чтобы сделать их "человечески-ми", передадим метки:

```
labels = ["молодой", "средний возраст", "взрослый", "пожилой"]
df["age_group"] = pd.cut(df["age"], bins=[0,25,45,65,100],
labels=labels)
```

Квантильное бакетирование (pd.qcut)

Используется, когда нужно разбить данные по частям с равным количеством объектов.

Это полезно для:

- сегментации пользователей по выручке;
- распределения объектов по уровням активности;
- моделей, где важен ранг, а не значение.

Пример:

```
df["age_quantile"] = pd.qcut(df["age"], q=4)
print(df)
```

Теперь каждая группа содержит примерно одинаковое число людей.

Бакетирование для бизнес-сегментации

Пример: сегментация клиентов по сумме покупок.

```
sales = pd.DataFrame({
    "user_id": [1, 2, 3, 4, 5],
    "total_spent": [1000, 300, 25000, 15000, 700]
})
sales["segment"] = pd.qcut(sales["total_spent"], q=3,
                           labels=["эконом", "средний", "премиум"])
print(sales)
```

Получаем простую и понятную сегментацию.

Бакетирование по бизнес-логике

Иногда диапазоны задает не статистика, а знание предметной области.

Например, уровень зарплаты:

```
def salary_to_level(x):
    if x < 40000:
        return "низкий доход"
    elif x < 80000:
        return "средний доход"
    else:
        return "высокий доход"
df["income_level"] = df["salary"].apply(salary_to_level)
```

Здесь мы сознательно используем бизнес-смысл, а не распределения.

Бакетирование может:

- уменьшить влияние выбросов;
- сделать нелинейные зависимости более линейными;

- помочь простым моделям работать лучше;
- улучшить интерпретацию результатов.

Если зависимость не линейная, а ступенчатая, моделям проще работать с бакетами. Бакетирование — это не просто разбиение на группы. Это способ выделить смысловые уровни признака. Мы превращаем "просто числа" в информационные категории.

5.4.2. Линеаризация зависимостей и преобразование распределений

Многие реальные числовые признаки имеют нелинейные распределения — часто с "длинным хвостом".

Например:

- сумма покупки,
- доход клиента,
- число посещений сайта,
- количество друзей в соцсетях.

Такие признаки искажают анализ и модели, потому что несколько больших значений могут "перетянуть" статистику на себя. Чтобы сгладить такие эффекты и сделать зависимость более линейной, используют преобразования признаков.

```
import pandas as pd
df = pd.DataFrame({
    "amount": [200, 250, 300, 350, 500, 800, 15000]
})
df["amount"].hist(bins=20)
```

Мы увидим резко вытянутый вправо хвост.

Логарифмирование

Логарифмирование — самый распространенный способ "сжать" большие значения.

```
import numpy as np
df["amount_log"] = np.log1p(df["amount"])
```

Сравним распределения:

```
df["amount"].hist(bins=20)
df["amount_log"].hist(bins=20)
```

После логарифмирования распределение станет более симметричным.

Корневые преобразования

Если признак колеблется не слишком сильно, можно использовать квадратный корень:

```
df["amount_sqrt"] = np.sqrt(df["amount"])
```

Смысл:

- уменьшает разницу между большими числами,
- но сохраняет относительные соотношения.

Это мягче, чем логарифмирование.

Степенные преобразования

Если зависимость вообще не похожа на линейную, можно применить степенные функции:

```
df["amount_pow2"] = df["amount"] ** 2
df["amount_pow_half"] = df["amount"] ** 0.5
```

Используется реже, но важно понимать принцип: мы меняем форму распределения, чтобы упростить модель.

Box-Cox

Работает только для положительных значений. Это оптимизированное степенное преобразование.

```
from scipy.stats import boxcox
```

```
df["amount_boxcox"], lam = boxcox(df["amount"])
print("lambda =", lam)
```

Функция сама находит степень, делающую данные максимально "нормальными".

Yeo-Johnson

Работает и с отрицательными значениями. Это более универсальная версия Вох-Сох.

```
from sklearn.preprocessing import PowerTransformer
pt = PowerTransformer(method="yeo-johnson")
df["amount_yj"] = pt.fit_transform(df[["amount"]])
```

Когда применять какое преобразование

- Логарифмирование — если есть очень большие значения (доходы, траты).
- Корень — если данные умеренно скошены.
- Вох-Сох / Yeo-Johnson — если нужна максимальная "нормальность" распределения.

Многие модели предполагают линейные зависимости, например:

- линейная регрессия;
- логистическая регрессия;
- SVM;
- модели на расстояниях (kNN);
- PCA.

Если зависимость нелинейна, модель:

- плохо обучается;
- становится чувствительной к выбросам;
- выдает неустойчивые коэффициенты,

- интерпретация ухудшается.

Преобразования помогают сделать зависимость более "прямой", а модель — устойчивой и понятной.

5.4.3. Работа с категориальными признаками

Категориальные признаки — это признаки, принимающие значения из ограниченного множества категорий, например: названия городов, тип устройства, способ оплаты, бренд товара, сегмент пользователя и т.д.

```
import pandas as pd
df = pd.DataFrame({
    "user_id": [1, 2, 3, 4],
    "city": ["Москва", "Казань", "Москва", "Сочи"],
    "payment_type": ["card", "cash", "card", "card"]
})
```

Большинство моделей и аналитических инструментов не умеет работать с текстовыми категориями напрямую. Их нужно преобразовать в числовой вид. Этот процесс называется кодированием категориальных признаков (encoding).

One-Hot Encoding (OHE)

Каждое значение категории превращается в отдельный бинарный признак.

```
df_ohe = pd.get_dummies(df, columns=["city", "payment_type"])
```

Плюсы:

- просто;
- интерпретируемо;
- не искажает расстояния.

Минусы:

- увеличивает размерность, особенно если категорий много (например, города мира).

Label Encoding

Переводит категории в числа:

```
from sklearn.preprocessing import LabelEncoder
encoder = LabelEncoder()
df["payment_type_le"] = encoder.fit_transform(df["payment_type"])
```

Однако порядок присваивается искусственно, и модель может начать "думать", что cash > card, хотя это не так. Поэтому Label Encoding подходит только для:

- моделей, нечувствительных к порядку (деревья решений, Random Forest, CatBoost);
- временного преобразования перед более сложным энкодингом.

Target Encoding (Mean Encoding)

Превращает категорию в среднее целевой переменной по группе. Используется, когда есть целевая переменная (y), например отток/покупка/клик.

Допустим, есть зависимость:

```
df = pd.DataFrame({
    "city": ["Москва", "Казань", "Казань", "Сочи", "Москва"],
    "purchase": [1, 0, 1, 0, 1]
})
```

Считаем среднюю вероятность покупки в каждом городе:

```
city_mean = df.groupby("city")["purchase"].mean()
df["city_te"] = df["city"].map(city_mean)
```

Результат будет:

- Москва — 1.0.
- Казань — 0.5.
- Сочи — 0.0.

Риск утечки данных (data leakage)

Если построить `city_mean` на всем датасете, модель получит информацию, которой не должна знать. Поэтому `target encoding` делается только на обучающей выборке, а затем применяется к тестовой.

```
train, test = df.iloc[:3], df.iloc[3:]
city_mean_train = train.groupby("city")["purchase"].mean()
train["city_te"] = train["city"].map(city_mean_train)
test["city_te"] = test["city"].map(city_mean_train)
```

Так мы не подсматриваем ответ в тесте.

Когда использовать какой метод

- Если категорий мало — One-Hot Encoding.
- Если категорий среднее количество и используется дерево или бустинг — Label Encoding.
- Если категорий много и есть целевая переменная — Target Encoding.
- Если категорий огромное число (например, ID товаров) — лучше использовать агрегаты/эмбединги (будет позже).

5.4.4. Создание новых признаков (Feature Construction)

Инженерия признаков — это не только преобразование уже существующих данных, но и создание новых признаков, которые отражают важные свойства объектов.

Даже если у нас есть идеальные данные и корректные модели, результат часто ограничен тем, какие признаки доступны модели. Поэтому задача аналитика и ML-инженера — извлечь смысл из данных, а не просто подавать их "как есть".

Арифметические комбинации признаков

Если два признака логически связаны, их комбинация может нести больше смысла.

```
import pandas as pd
df = pd.DataFrame({
    "price": [1000, 2000, 1500],
    "quantity": [1, 2, 4]
})
df["revenue"] = df["price"] * df["quantity"]
```

Теперь модель получает признак, непосредственно отражающий бизнес-значение, — выручку.

Взаимодействующие признаки (interaction features)

Иногда эффект одной переменной зависит от другой. Можно явно создать признак-взаимодействие:

```
df["price_qty_interaction"] = df["price"] * df["quantity"]
```

Для множества признаков удобно использовать PolynomialFeatures:

```
from sklearn.preprocessing import PolynomialFeatures
poly = PolynomialFeatures(degree=2, include_bias=False)
X = poly.fit_transform(df[["price", "quantity"]])
```

Но нужно быть осторожным: количество признаков растет очень быстро.

Агрегаты по группам

Часто полезно группировать данные по клиенту, товару, партнеру и т.д.

```
df = pd.DataFrame({
    "user_id": [1, 1, 2, 2, 2],
    "purchase_amount": [1000, 1200, 500, 400, 700]
})
```

```
user_features = df.groupby("user_id")["purchase_amount"].agg(
    total_spent="sum",
    avg_spent="mean",
    purchases="count"
).reset_index()
```

Это сегментационные признаки — их часто используют в churn-моделях и RFM-анализе.

Работа с датами и временем

Дата — один из самых информативных источников признаков, но в сыром виде почти бесполезна.

```
df = pd.DataFrame({
    "timestamp": pd.to_datetime(["2024-01-01 10:30", "2024-01-02
22:15", "2024-06-15 04:05"])
})
```

Извлекаем информацию:

```
df["day_of_week"] = df["timestamp"].dt.dayofweek
df["hour"] = df["timestamp"].dt.hour
df["month"] = df["timestamp"].dt.month
df["is_weekend"] = df["timestamp"].dt.dayofweek >= 5
```

Складные признаки (rolling/moving window)

Полезно во временных рядах.

```
df["rolling_mean_7"] = df["purchase_amount"].rolling(7).mean()
```

Этот признак показывает сглаженную динамику.

Отношения и нормы

Отношения часто отражают напряженность поведения:

```
df["return_rate"] = df["returned"] / df["purchased"]
df["ctr"] = df["clicks"] / df["views"]
```

```
df["share_of_category"] = df["user_category_purchases"] / df["total_purchases"]
```

Такие признаки часто более интерпретируемы, чем исходные величины.

Бинаризация событий

Иногда важно не значение, а сам факт:

```
df["is_high_value"] = df["revenue"] > 10000
df["is_mobile"] = df["device"] == "mobile"
df["is_night"] = df["hour"].between(0,5)
```

5.4.5. Отбор и оценка качества признаков

Когда набор признаков создан, очищен и преобразован, возникает следующий вопрос: какие признаки действительно полезны, а какие — только создают шум и усложняют модель?

Отбор признаков — это поиск тех признаков, которые вносят наибольший вклад в объяснение поведения данных или качества предсказаний.

Если оставить лишние признаки:

- модель становится сложнее и медленнее;
- возрастает риск переобучения;
- интерпретация ухудшается.

Если удалить важные признаки, модель теряет смысл и предсказывает хуже.

Поэтому нужна оценка важности признаков.

Корреляционный анализ

Если два признака сильно коррелируют между собой, они несут одну и ту же информацию. Хранить оба — бессмысленно: модель просто "учится" на повторениях.

```
import pandas as pd
import numpy as np
df = pd.DataFrame({
    "height": [160, 172, 158, 182, 175],
    "weight": [55, 68, 49, 82, 74],
    "height_cm": [160, 172, 158, 182, 175] # Дубликат признака
})
print(df.corr())
```

Мы увидим корреляцию 1.00 между height и height_cm.

В таком случае один признак можно удалить:

```
df = df.drop(columns=["height_cm"])
```

Корреляция между признаком и целевой переменной

Если у нас есть целевая переменная, можно оценить сильные предикторы.

```
df.corr()["revenue"].sort_values(ascending=False)
```

Чем ближе значение к 1 или -1, тем сильнее линейная связь.

Feature Importance из деревьев решений и бустингов

Модели на основе деревьев естественным образом оценивают важность признаков.

```
from sklearn.ensemble import RandomForestRegressor
model = RandomForestRegressor()
model.fit(df.drop(columns=["target"]), df["target"])
importances = model.feature_importances_
for feature, importance in zip(df.drop(columns=["target"]).columns,
importances):
    print(feature, round(importance, 3))
```

Этот метод:

- не требует нормализации признаков;
- работает и с нелинейностями;
- хорошо подходит для отбора признаков в прикладных моделях.

Permutation Importance (перестановочная важность)

Этот подход меняет местами значения признака и измеряет, насколько ухудшается качество модели. Если качество падает сильно — признак важный. Если почти не меняется — признак можно удалить.

```
from sklearn.inspection import permutation_importance
result = permutation_importance(model, df.drop(columns=["target"]),
df["target"])
print(result.importances_mean)
```

Этот метод более честный, чем встроенные важности в деревьях.

SHAP — интерпретируемая важность признаков

SHAP объясняет вклад каждого признака в предсказание модели. Это особенно полезно в бизнес-контекстах: нужно объяснить "почему модель решила так".

```
import shap
explainer = shap.TreeExplainer(model)
shap_values = explainer.shap_values(df.drop(columns=["target"]))
shap.summary_plot(shap_values, df.drop(columns=["target"]))
```

Мы получаем визуализацию влияния каждого признака.

Удаление нерелевантных признаков

Простейший автоматический способ:

```
df = df.drop(columns=[col for col in df.columns if df[col].nunique()
<= 1])
```

Удаляем признаки, где нет вариативности, например колонку с одним значением.

Итеративный отбор признаков (RFE)

RFE последовательно удаляет признаки с минимальным вкладом:

```
from sklearn.feature_selection import RFE
from sklearn.linear_model import LinearRegression
model = LinearRegression()
selector = RFE(model, n_features_to_select=5)
selector.fit(df.drop(columns=["target"]), df["target"])

selected_columns =
df.drop(columns=["target"]).columns[selector.support_]
print(selected_columns)
```

Используется, когда модель должна быть минимально сложной и хорошо интерпретируемой.

5.4.6. Итоги раздела

В этом разделе мы научились работать с признаками так, чтобы данные становились более информативными для анализа и моделей:

- ◆ поняли, почему качество признаков влияет на результат сильнее, чем выбор алгоритма;
- ◆ разобрали бакетирование и научились превращать непрерывные признаки в смысловые категории;
- ◆ рассмотрели логарифмирование, корневые и степенные преобразования для линеаризации распределений;
- ◆ научились кодировать категориальные признаки: One-Hot Encoding, Label Encoding, Target Encoding;
- ◆ увидели, почему Target Encoding нужно применять осторожно, чтобы избежать утечки данных;

- ◆ создали новые признаки из существующих — арифметические комбинации, взаимодействия, агрегаты и признаки времени;
- ◆ построили поведенческие и структурные признаки (частоты, доли, бинарные индикаторы);
- ◆ познакомились с методами отбора признаков — корреляцией, важностью в деревьях, permutation importance и SHAP;
- ◆ поняли, как удалять нерелевантные признаки и формировать компактный, сильный набор входных данных.

5.5. Основы машинного обучения

К этому моменту мы уже проделали большой путь в анализе данных. Мы изучали поведение клиентов, строили RFM-сегменты, анализировали когорты, работали с временными рядами и делали прогнозы. Во всех этих задачах мы искали закономерности: какие клиенты приносят больше денег, как меняется активность со временем, какие группы пользователей ведут себя похоже.

По сути, мы уже занимались машинным обучением — просто делали это вручную. Мы сами формулировали правила, сами выбирали границы сегментов, сами решали, какие признаки важны. Машинное обучение отличается тем, что эту работу мы постепенно передаем алгоритму.

Машинное обучение (Machine Learning, ML) — это подход к анализу данных, при котором компьютер не следует заранее заданным правилам, а учится на примерах. Он анализирует исторические данные, находит в них зависимости и затем использует эти зависимости для прогнозов или принятия решений на новых данных.

Если у аналитика есть правильные ответы, он может сравнивать примеры, делать выводы и постепенно улучшать свои прогнозы.

Если же правильных ответов нет, он все равно может искать структуру в данных — группировать клиентов по схожести поведения, выделять типичные паттерны, находить аномалии.

Ровно так же работают и алгоритмы машинного обучения. В зависимости от того, есть ли у нас заранее известный "правильный ответ", различают основные типы обучения.

Существует несколько подходов к обучению моделей, но для аналитика данных ключевыми являются два:

1. Обучение с учителем (supervised learning);
2. Обучение без учителя (unsupervised learning).

Их отличие определяется не алгоритмами, а типом данных и постановкой задачи.

5.5.1. Обучение с учителем и без учителя

Обучение с учителем (Supervised Learning)

В обучении с учителем мы знаем правильный ответ для каждого примера. То есть у нас есть входные данные (например, RFM-признаки клиента) и целевая переменная — результат, который нужно предсказать (например, "ушел" или "остался").

Модель анализирует прошлые примеры, строит зависимость между признаками и результатом, а затем использует эту зависимость для прогнозирования новых случаев.

Пример из предыдущих глав: мы уже считали метрики R (Recency), F (Frequency), M (Monetary). Теперь добавим к ним признак "ушел ли клиент" (1 — да, 0 — нет) и научим модель предсказывать вероятность ухода.

Листинг 5.15. Предсказание оттока клиентов

```
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LogisticRegression
```



```
# Простые данные на основе RFM
df = pd.DataFrame({
    "recency": [5, 90, 20, 60, 10, 3, 45, 70],
    "frequency": [12, 1, 7, 3, 10, 15, 5, 2],
    "monetary": [3000, 200, 1800, 600, 2500, 4000, 800, 400],
    "churn": [0, 1, 0, 1, 0, 0, 1, 1]
})
X = df[["recency", "frequency", "monetary"]]
y = df["churn"]

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.25,
random_state=42)
# Обучаем модель
model = LogisticRegression()
model.fit(X_train, y_train)
# Прогноз
preds = model.predict(X_test)
print("Прогноз:", preds)
```

Мы получили простую модель, которая по активности клиента предсказывает, уйдет ли он. Это и есть обучение с учителем: мы дали модели "примеры" и "ответы". Подобным образом мы позже будем прогнозировать конверсию, доход или вероятность покупки.

Обучение без учителя (Unsupervised Learning)

Иногда у нас нет ответов, и мы не знаем, кто ушел, а кто остался. Но мы хотим понять, как ведут себя клиенты и чем они похожи. В этом случае модель сама ищет закономерности, объединяя объекты с похожими характеристиками.

Это обучение без учителя. Мы не подсказываем ей, где "хорошие" или "плохие" клиенты — она сама находит группы по поведению.

Когда мы делали RFM-анализ, мы уже группировали клиентов вручную: "лучшие", "спящие", "новые". Машинное обучение может сделать это автоматически — например, с помощью алгоритма KMeans.

Листинг 5.16. Кластеризация клиентов

```
import pandas as pd
from sklearn.cluster import KMeans
import matplotlib.pyplot as plt
df = pd.DataFrame({
    "recency": [5, 90, 20, 60, 10, 3, 45, 70, 15, 40],
    "frequency": [12, 1, 7, 3, 10, 15, 5, 2, 8, 6],
    "monetary": [3000, 200, 1800, 600, 2500, 4000, 800, 400, 2200, 1000]
})
kmeans = KMeans(n_clusters=3, random_state=42)
df["cluster"] = kmeans.fit_predict(df)
```

Теперь клиенты разбиты на 3 кластера — без всяких меток. Модель сама определила, какие пользователи "похожи". А аналитик уже может интерпретировать эти кластеры:

- одни — активные и прибыльные,
- другие — редкие покупатели,
- третьи — почти потерянные.

Как понять, какой тип обучения использовать? Если вы знаете правильный ответ (например, кто ушел, кто купил, какой доход был) — используйте обучение с учителем. Если вы просто хотите понять структуру данных, найти закономерности, используйте обучение без учителя.

5.5.2. Пример: прогноз оттока клиентов (Churn Prediction)

Отток (churn) — один из ключевых показателей любой компании, работающей с повторными покупками: интернет-магазинов, подписочных сервисов, банков. Если клиент перестает пользоваться продуктом, бизнес теряет не только деньги, но и возможность дальнейших продаж.

Предсказание оттока позволяет вовремя вмешаться:

- предложить бонус;

- напомнить о товаре;
- вернуть клиента до того, как он окончательно уйдет.

В этой части главы мы создадим простую модель прогноза оттока клиентов — шаг за шагом, в духе наших предыдущих аналитических проектов. Мы снова используем знакомые показатели RFM (Recency, Frequency, Monetary), а также применим базовый, но мощный метод — логистическую регрессию. В следующей главе мы подробно разберем, как оценивать качество модели, а пока сосредоточимся на том, как ее построить и запустить.

Подготовка данных

Для начала сгенерируем данные о клиентах интернет-магазина. Мы создадим таблицу, где каждый клиент описан тремя признаками:

- **recency** — сколько дней прошло с последней покупки (чем больше, тем клиент менее активен);
- **frequency** — сколько заказов он сделал;
- **monetary** — общая сумма всех его покупок.

К ним добавим целевую переменную **churn** — ушел клиент или нет.

Листинг 5.17. Генерация данных клиентов

```
import pandas as pd
import numpy as np
np.random.seed(42)
# генерируем 1000 клиентов
n = 1000
df = pd.DataFrame({
    "recency": np.random.randint(1, 200, n),
    "frequency": np.random.randint(1, 20, n),
    "monetary": np.random.uniform(200, 10000, n)
})
# Простое правило: клиент ушел, если давно не покупал, покупал редко
# и на небольшие суммы
df["churn"] = np.where(
```

```
(df["recency"] > 100) & (df["frequency"] < 5) & (df["monetary"]
< 3000),
    1, 0
)
```

На выходе — таблица клиентов с показателями активности и меткой churn, которая указывает, остался клиент или ушел.

Это имитация реальных CRM-данных, где информация о клиентах поступает из базы заказов.

Разделение данных на обучение и тест

Чтобы убедиться, что модель действительно умеет предсказывать, а не просто "запоминает" данные, мы разделим набор на две части:

- **обучающую** (train) — для построения модели;
- **тестовую** (test) — для проверки на новых примерах.

Листинг 5.18. Разделение выборки

```
from sklearn.model_selection import train_test_split
X = df[["recency", "frequency", "monetary"]]
y = df["churn"]
X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.25, random_state=42
)
print("Размер обучающей выборки:", X_train.shape)
print("Размер тестовой выборки:", X_test.shape)
```

Теперь у нас есть 750 клиентов для обучения и 250 — для проверки. Тестовая выборка понадобится в следующей главе для оценки качества модели.

Обучение модели логистической регрессии

Логистическая регрессия — это базовая модель для задач классификации. Она не прогнозирует конкретное число, а возвращает вероятность события (в нашем случае — вероятность ухода клиента).

Если вероятность > 0.5 , мы считаем, что клиент уйдет.

Листинг 5.19. Обучение модели и прогноз

```
from sklearn.linear_model import LogisticRegression
model = LogisticRegression(max_iter=1000)
model.fit(X_train, y_train)
# Прогнозы
y_pred = model.predict(X_test)
y_prob = model.predict_proba(X_test)[:, 1] # Вероятность оттока

results = pd.DataFrame({
    "recency": X_test["recency"],
    "frequency": X_test["frequency"],
    "monetary": X_test["monetary"],
    "actual": y_test.values,
    "predicted": y_pred,
    "probability": y_prob.round(3)
})
print(results.head())
```

Мы получили две важные вещи:

1. **predicted** — бинарный прогноз (0 = останется, 1 = уйдет);
2. **probability** — вероятность оттока.

recency_	monetary	actual	predicted	probability	
180	3	1200	1	1	0.91
45	10	7500	0	0	0.08
130	2	900	1	1	0.82

Модель уверенно выделяет клиентов, давно не делавших покупок.

Интерпретация результатов

Одна из сильных сторон логистической регрессии — понятность. Она не только предсказывает, но и показывает, какие признаки сильнее всего влияют на результат.

Листинг 5.20. Коэффициенты модели

```
coeff = pd.DataFrame({
    "Признак": X.columns,
    "Коэффициент": model.coef_[0]
})
```

Интерпретация:

- Чем больше дней с последней покупки, тем выше вероятность оттока (положительный коэффициент).
- Чем чаще и больше клиент покупает, тем меньше шанс, что он уйдет (отрицательные коэффициенты).

Это полностью согласуется с нашей бизнес-логикой из RFM-анализа: активные и "свежие" клиенты лояльнее.

Проверка модели

Мы получили модель, но пока не знаем, насколько она точна. Для этого нужно посчитать метрики качества:

- **accuracy** — насколько часто модель угадывает;
- **precision** — насколько точны ее "предупреждения об оттоке";
- **recall** — сколько реально ушедших клиентов она успела поймать;
- **ROC AUC** — насколько хорошо модель различает "ушедших" и "оставшихся".

Этому будет посвящена следующая часть главы. Сейчас важно понимать, что метрики — это способ измерить пользу модели для бизнеса: если она правильно определяет клиентов с высоким риском ухода, значит, ее можно использовать в реальной CRM.

Что можно улучшить

Даже простая логистическая регрессия может стать мощным инструментом, если подготовить **признаки** (features) более тщательно:

- добавить `avg_order_value` (средний чек);
- учесть `days_since_signup`;
- рассчитать активность за последние 30 дней;
- добавить демографию или источник привлечения клиента.

Каждый новый признак помогает модели точнее понимать "поведение лояльных и уходящих клиентов".

5.5.3. Оценка качества модели: accuracy, precision, recall, ROC AUC

Построить модель — это только половина дела. Главное — понять, насколько хорошо она работает. Если модель неправильно классифицирует клиентов, бизнес будет терять деньги:

- предскажет, что клиент уйдет, хотя он лоялен — лишние скидки и потери;
- не заметит реально уходящего — потеря клиента.

Поэтому после обучения важно не просто "посмотреть прогнозы", а измерить точность и надежность модели с помощью метрик.

В этой части главы мы научимся оценивать качество прогноза модели оттока клиентов, которую построили ранее, и разберем 4 ключевые метрики, применимые почти в любом проекте машинного обучения: accuracy, precision, recall и ROC AUC.

Мы уже начинали разбирать метрики качества в предыдущих главах, теперь сделаем это более детально.

Почему одной метрики недостаточно

На первый взгляд кажется, что можно просто посчитать, сколько раз модель угадала ответ правильно.

Это метрика **ассигасу** — доля верных предсказаний. Но представьте: из 1000 клиентов 900 остались, 100 ушли. Если модель говорит "все останутся" — она будет права в 90% случаев. $Accuracy = 0.9$. Отличный результат? Нет: модель не нашла ни одного ушедшего клиента, а значит, бесполезна. Вывод: при дисбалансе классов (например, 10%-й отток) ассигасу вводит в заблуждение — нужны дополнительные метрики.

Матрица ошибок (Confusion Matrix)

Прежде чем перейти к метрикам, разберемся, что такое ошибки классификации. Когда модель предсказывает класс ("уйдет"/"останется"), возможны 4 варианта:

Реальный класс	Прогноз модели	Что это значит
Ушел (1)	Ушел (1)	True Positive (TP) — модель угадала
Ушел (1)	Остался (0)	False Negative (FN) — пропустила уход
Остался (0)	Ушел (1)	False Positive (FP) — ложная тревога
Остался (0)	Остался (0)	True Negative (TN) — все верно

На основе этих четырех значений считаются все остальные метрики.

Accuracy — общая точность

$$Accuracy = \frac{TP + TN}{TP + TN + FP + FN}$$

Она показывает, в каком проценте случаев модель угадала. Если модель правильно классифицировала 900 клиентов из 1000:

$$\text{Accuracy} = 900 / 1000 = 0.9$$

Хорошо, но, как мы видели, может быть обманчиво при несбалансированных классах.

Precision и Recall — когда важна цена ошибки

В задачах вроде прогнозирования оттока или мошенничества важно не только "угадать побольше", а угадать правильно.

- **Precision** (точность) — доля верных среди тех, кого модель сочла "ушедшими".

$$\text{Precision} = \frac{TP}{TP + FP}$$

Если модель сказала, что 100 клиентов уйдут, а на самом деле ушло 80 — $\text{precision} = 0.8$. То есть 20% были "ложными тревогами".

- **Recall** (полнота) — доля реальных ушедших, которых модель нашла.

$$\text{Recall} = \frac{TP}{TP + FN}$$

Если ушло 100 клиентов, а модель предсказала 80 из них, $\text{recall} = 0.8$.

Обычно **precision** и **recall** находятся в противоречии: чем больше "ловим" (**recall**), тем больше ложных тревог (**precision** снижается).

ROC AUC — баланс чувствительности и точности

ROC AUC — одна из самых популярных метрик, потому что она оценивает модель целиком, а не при конкретном пороге 0.5.

ROC (Receiver Operating Characteristic) — это кривая, показывающая, как меняется соотношение между *True Positive Rate* (чувствительностью) и *False Positive Rate* (ошибками), если менять порог вероятности.

- Чем ближе кривая к верхнему левому углу, тем лучше модель.
- AUC (Area Under Curve) — площадь под кривой:
 - » 0.5 — модель случайная,
 - » 1.0 — идеальная классификация.

Расчет метрик на нашей модели оттока

Мы используем модель логистической регрессии, обученную в предыдущей главе, и проверим, насколько хорошо она предсказывает уход клиентов.

Листинг 5.21. Расчет основных метрик

```
from sklearn.metrics import (
    accuracy_score, precision_score, recall_score, roc_auc_score,
    confusion_matrix
)
# Предсказания из предыдущей модели
y_pred = model.predict(X_test)
y_prob = model.predict_proba(X_test)[:, 1]
# Метрики
acc = accuracy_score(y_test, y_pred)
prec = precision_score(y_test, y_pred)
rec = recall_score(y_test, y_pred)
roc_auc = roc_auc_score(y_test, y_prob)
print(f"Accuracy: {acc:.3f}")
print(f"Precision: {prec:.3f}")
print(f"Recall: {rec:.3f}")
print(f"ROC AUC: {roc_auc:.3f}")
# Матрица ошибок
cm = confusion_matrix(y_test, y_pred)
```

Интерпретация:

- Модель предсказывает правильно в 91% случаев (accuracy).
- Среди клиентов, которых она сочла "ушедшими", 88% действительно ушли (precision).
- Она успевает "поймать" 86% реально уходящих (recall).

- Общая способность различать классы - 0.95 (ROC AUC), что очень хорошо.

Как выбирать метрику в реальной задаче

Разные задачи требуют разных приоритетов.

- Если каждая ошибка стоит дорого (например, банк звонит клиентам с риском оттока) — важен *precision*, чтобы не тревожить лишних людей.
- Если важно поймать всех потенциальных уходящих — используйте *recall*, даже если будет больше ложных тревог.
- Если нужно общее понимание качества — смотрите ROC AUC.

В реальных проектах часто комбинируют метрики и подбирают порог вероятности (0.5, 0.6, 0.7), чтобы достичь нужного баланса между *precision* и *recall*.

5.5.4. Итоги раздела

- ◆ Познакомились с базовыми принципами машинного обучения и поняли разницу между обучением с учителем (прогноз, классификация) и без учителя (кластеризация, сегментация).
- ◆ Увидели, что привычные аналитические подходы — например, RFM-анализ — уже близки к методам ML.
- ◆ На практике реализовали задачу прогноза оттока клиентов (*churn prediction*) с использованием логистической регрессии.
- ◆ Подготовили данные, разделили их на обучающую и тестовую выборки, обучили модель и получили прогнозы вероятности ухода.
- ◆ Научились интерпретировать коэффициенты логистической регрессии и понимать влияние признаков на результат.
- ◆ Освоили ключевые метрики качества: *accuracy*, *precision*, *recall*, ROC AUC — и научились визуализировать ROC-кривую.
- ◆ Поняли, что качественная модель — это не только прогноз, но и измеримая польза для бизнеса, выраженная в понятных показателях точности и надежности.

5.6. Инженерия данных и хранение аналитических систем

В этой части главы мы говорим об инженерии данных — то есть о том, как устроен технический фундамент аналитики: откуда берутся данные, как они проходят путь от источников до дашбордов и какие инструменты обеспечивают скорость, масштаб и стабильность.

Чтобы повторять примеры из книги, нам понадобится реальная аналитическая база, на которой можно строить ETL-пайплайны, считать агрегаты и хранить большие объемы информации. Для этого идеально подходит ClickHouse — современная, быстрая и бесплатная база данных, специально созданная для аналитики.

ClickHouse — это система для OLAP (Online Analytical Processing), то есть для обработки аналитических запросов по большим наборам данных. Она способна за доли секунды считать агрегаты по миллиардам строк и при этом проста в установке. Для нашего учебного окружения достаточно запустить ее локально с помощью Docker.

Docker — это инструмент для запуска приложений в контейнерах. Контейнер — это, по сути, "мини-компьютер" с уже установленной программой, который можно запустить одной командой, без сложной настройки окружения.

Для работы с данным разделом нам необходимо будет установить оба этих инструмента.

Перед запуском ClickHouse нужно установить сам Docker.

Установка Docker на Windows

Перейдите на сайт <https://www.docker.com/products/docker-desktop>.

1. Скачайте Docker Desktop for Windows.

2. Установите программу, следуя инструкции.
3. После запуска в трее появится синий значок  — значит, Docker работает.

Проверьте установку в PowerShell:

```
docker --version
```

Установка Docker на macOS

Перейдите на <https://www.docker.com/products/docker-desktop>.

1. Скачайте Docker Desktop for Mac (версии для Intel и Apple Silicon разные).
2. Установите и запустите.
3. Проверьте установку:

```
docker --version
```

Запуск ClickHouse через Docker

Когда Docker установлен, весь ClickHouse можно развернуть одной командой.

```
docker run -d \
  --name clickhouse-server \
  -p 8123:8123 -p 9000:9000 \
  -e CLICKHOUSE_USER=analytics \
  -e CLICKHOUSE_PASSWORD=secret123 \
  -e CLICKHOUSE_DB=analytics \
  -v clickhouse-data:/var/lib/clickhouse \
  clickhouse/clickhouse-server
```

Что делает команда:

- `-d` — запускает контейнер в фоновом режиме;
- `--name clickhouse-server` — имя контейнера (можно выбрать любое);
- `-p 8123:8123` — открывает порт для веб-интерфейса и HTTP-запросов;
- `-p 9000:9000` — открывает порт для SQL-клиентов и Python;
- `clickhouse/clickhouse-server` — официальный образ ClickHouse.

Проверка работы ClickHouse

Проверим, что сервер запущен:

```
docker ps
```

Теперь ClickHouse работает локально, и вы можете к нему подключаться.

Проверка в браузере

Откройте <http://localhost:8123/play>.

Вы попадете в веб-интерфейс ClickHouse с простым SQL-редактором.

Попробуйте запрос:

```
SELECT version(), now();
```

Если видите версию и дату — ClickHouse запущен и готов к работе.

Подключение из Python

ClickHouse отлично интегрируется с Python. Установите драйвер:

```
pip install clickhouse-driver
```

Теперь можно проверить соединение:

```
from clickhouse_driver import Client
client = Client(host="localhost", user="analytics",
password="secret123", database="analytics")
version = client.execute("SELECT version()")
print("Версия ClickHouse:", version[0][0])
```

Результат:

```
Версия ClickHouse: 23.9.2.7
```

Управление контейнером

Остановить сервер:

```
docker stop clickhouse-server
```

Запустить снова:

```
docker start clickhouse-server
```

Удалить контейнер:

```
docker rm -f clickhouse-server
```

Чтобы сохранять данные между перезапусками, добавьте том:

```
docker run -d \
  --name clickhouse-server \
  -v clickhouse-data:/var/lib/clickhouse \
  -p 8123:8123 -p 9000:9000 \
  clickhouse/clickhouse-server
```

Теперь ClickHouse готов к использованию в следующих частях книги.

5.6.1. Что такое ETL / ELT (извлечение, трансформация, загрузка)

Когда аналитик открывает дашборд, он видит красивые графики: выручку по месяцам, количество заказов, средний чек. Но за этим графиком скрывается огромная "кухня данных": данные из CRM, базы заказов, платежных систем, рекламных источников — все это нужно собрать, очистить и связать.

Данные не попадают в хранилище сами собой. Между моментом, когда клиент оформил заказ, и моментом, когда вы видите этот заказ на графике, происходят десятки шагов: загрузки, фильтрации, агрегации. Все это вместе называется ETL — Extract, Transform, Load ("извлечение, преобразование, загрузка"). Современная альтернатива — ELT, где обработка данных происходит уже в хранилище.

ETL — это стандартный процесс, который проходит любой набор данных, прежде чем стать "аналитическим".

Три шага ETL:

1. **Extract** (извлечение) — забираем данные из разных источников (PostgreSQL, API, Excel, CRM).
2. **Transform** (трансформация) — очищаем, фильтруем, объединяем и считываем метрики.
3. **Load** (загрузка) — помещаем обработанные данные в хранилище, например ClickHouse или BigQuery.

Extract — извлечение данных

Первый шаг — собрать данные из разных систем. Источников могут быть десятки:

Источник	Пример
Реляционная база	PostgreSQL, MySQL (таблицы orders, users, products)
API сервисов	Google Ads, Facebook, Telegram, Stripe
Файлы	CSV, Excel, JSON
Логи и клики	Веб-серверы, системы трекинга
Системы аналитики	Amplitude, Appsflyer, Google Analytics

Пример — извлекаем заказы из PostgreSQL:

```
import pandas as pd
from sqlalchemy import create_engine
engine = create_engine("postgresql://user:password@localhost:5432/shopdb")
query = """
SELECT id, customer_id, amount, order_date, category
FROM orders
WHERE order_date >= '2025-01-01'
"""
df = pd.read_sql(query, engine)
print(df.head())
```


На этом шаге важно не перегружать источник, поэтому обычно делают инкрементальную загрузку — берут только новые данные.

Например:

```
SELECT * FROM orders WHERE updated_at > '2025-02-01';
```

Transform — преобразование данных

Данные из разных систем редко идеально совпадают. У них могут отличаться форматы дат, валюты, названия полей. Чтобы аналитика работала, все нужно привести к единому стандарту.

Типичные операции трансформации:

Задача	Пример
Приведение форматов дат	2025/03/01 → 2025-03-01
Удаление дубликатов	df.drop_duplicates()
Очистка от отрицательных сумм	df = df[df.amount > 0]
Заполнение пропусков	df.fillna('unknown')
Расчет новых метрик	df['avg_check'] = df['revenue'] / df['orders']

Пример в Pandas:

```
df["order_date"] = pd.to_datetime(df["order_date"])
df = df[df["amount"] > 0]
df = df.drop_duplicates(subset="id")
df["month"] = df["order_date"].dt.to_period("M")
monthly = df.groupby(["month", "category"])["amount"].sum().reset_index(name="revenue")
print(monthly.head())
```

После этого этапа у нас появляется витрина данных — таблица, готовая к анализу или загрузке в ClickHouse.

Load — загрузка данных

Теперь, когда данные очищены, их нужно загрузить в хранилище или BI-систему.

Варианты, куда грузят данные:

Куда	Пример
OLAP-хранилище	ClickHouse, BigQuery, Snowflake
Data Lake	S3, Google Cloud Storage
BI-система	Power BI, Tableau, Metabase
CSV / Parquet-файл	Для отчетов и резервных копий

Пример загрузки в ClickHouse:

```
from clickhouse_driver import Client
client = Client(host="localhost", user="analytics",
password="secret123", database="analytics")
client.execute("""
CREATE TABLE IF NOT EXISTS analytics.sales (
    month String,
    category String,
    revenue Float64
)
ENGINE = MergeTree()
ORDER BY (month, category)
""")
data = [(str(row["month"]), row["category"], float(row["revenue"]))
for _, row in monthly.iterrows()]
client.execute("INSERT INTO analytics.sales (month, category, revenue)
VALUES", data)
print("Данные успешно загружены в ClickHouse.")
```

Теперь в ClickHouse есть таблица с ежемесячной выручкой по категориям — ее можно использовать в BI или в дашбордах.

ETL vs ELT — в чем разница

В классическом ETL все шаги происходят до загрузки: мы очищаем данные в Python, а в хранилище кладем уже готовые результаты.

В современном ELT сначала загружаем данные как есть (Extract → Load), а потом выполняем трансформации внутри хранилища (Transform). В реальности часто используется гибрид: данные сначала проходят легкую очистку (ETL), а сложные расчеты делаются уже в хранилище (ELT).

Автоматизация ETL

ETL — это не разовая операция. Она должна выполняться регулярно — каждый день, час или даже минуту. Для этого используют планировщики и оркестраторы: cron, Airflow, Prefect. Даже простая связка "Python + cron" уже считается ETL-пайплайном, если она регулярно загружает и обновляет данные.

Короткий пример из реальной жизни: компания Ozon ежедневно обновляет аналитические витрины с помощью ETL. Каждый час собираются миллионы строк заказов, которые проходят этапы:

- » **Extract: PostgreSQL → Kafka → Data Lake.**
- » **Transform: Spark + dbt.**
- » **Load: ClickHouse и Vertica для аналитиков.**

Это пример масштабного ETL в действии: все автоматизировано, но идея остается той же, что и в нашем локальном примере.

5.6.2. Пример пайплайна: выгрузка данных из PostgreSQL, очистка в Pandas, загрузка в ClickHouse

Теперь, когда мы понимаем, что такое ETL (извлечение, трансформация, загрузка), пора собрать реальный работающий пайплайн — простую, но наглядную схему, которая показывает, как данные проходят путь от базы с заказами до аналитического хранилища.

Мы создадим учебный пример на Python, используя три ключевых шага:

1. **Extract** — возьмем данные из PostgreSQL (таблица `orders`).
2. **Transform** — очистим и агрегируем данные с помощью `Pandas`.
3. **Load** — загрузим результат в ClickHouse для быстрой аналитики.

Подготовка окружения

Перед запуском убедитесь, что:

- ClickHouse запущен в Docker;
- PostgreSQL установлен локально или доступен из сети.

Создадим таблицу `orders` и наполним ее синтетическими данными:

Листинг 5.22. Генерация таблицы заказов в PostgreSQL

```
import pandas as pd
import numpy as np
from sqlalchemy import create_engine
from datetime import datetime, timedelta
import random

# Подключение к PostgreSQL
engine = create_engine("postgresql://postgres:123456@localhost:5432/shopdb")

# Создаем таблицу
with engine.begin() as conn:
    conn.execute("DROP TABLE IF EXISTS orders;")
    conn.execute("""
        CREATE TABLE orders (
            id SERIAL PRIMARY KEY,
            customer_id INTEGER,
```

```

        product TEXT,
        category TEXT,
        amount NUMERIC(10,2),
        order_date DATE
    );
    """
# Генерация данных
products = [
    ("Ноутбук", "Электроника"),
    ("Смартфон", "Электроника"),
    ("Футболка", "Одежда"),
    ("Книга", "Книги"),
    ("Кроссовки", "Одежда"),
    ("Чайник", "Бытовая техника"),
]
start_date = datetime(2023, 1, 1)
orders = []
for _ in range(5000):
    prod, cat = random.choice(products)
    customer_id = random.randint(1, 500)
    amount = round(random.uniform(500, 50000), 2)
    order_date = start_date + timedelta(days=random.randint(0, 700))
    orders.append((customer_id, prod, cat, amount, order_date))
df = pd.DataFrame(orders, columns=["customer_id", "product",
    "category", "amount", "order_date"])
df.to_sql("orders", con=engine, if_exists="append", index=False)
print("Таблица orders успешно создана и заполнена.")

```

Теперь в PostgreSQL есть таблица `orders`, содержащая данные о заказах за 2 года.

Extract — извлекаем данные из PostgreSQL

Используем библиотеку SQLAlchemy для подключения и выгрузки данных в Pandas.

Листинг 5.23. Извлечение данных

```
import pandas as pd
from sqlalchemy import create_engine
engine = create_engine("postgresql://postgres:123456@localhost:5432/
shopdb")
query = """
SELECT id, customer_id, category, amount, order_date
FROM orders
WHERE order_date >= '2024-01-01'
"""
df = pd.read_sql(query, engine)
print("Количество строк:", len(df))
print(df.head())
```

На этом этапе мы просто получили "сырые" данные — без фильтрации и агрегации. Следующим шагом будет их обработка и нормализация.

Transform — очистка и подготовка данных в Pandas

Теперь нам нужно привести данные к удобной форме для аналитики:

- удалить дубликаты;
- убрать отрицательные или нулевые суммы;
- добавить поле "месяц" и агрегировать выручку.

```
# Очистка
df = df.drop_duplicates(subset="id")
df = df[df["amount"] > 0]
df["order_date"] = pd.to_datetime(df["order_date"])
# Добавляем колонку "month"
df["month"] = df["order_date"].dt.to_period("M").astype(str)
# Группировка — выручка по месяцам и категориям
monthly = (
    df.groupby(["month", "category"])
      .agg(revenue=("amount", "sum"), orders=("id", "count"))
      .reset_index()
```

```
.sort_values("month")
)
print(monthly.head())
```

Теперь у нас таблица, где каждая строка — это комбинация месяца и категории, а в колонках указана выручка и количество заказов.

Load — загрузка данных в ClickHouse

Для загрузки используем драйвер `clickhouse-driver`.

```
from clickhouse_driver import Client
# Подключаемся к ClickHouse
client = Client(host="localhost", user="analytics",
password="secret123", database="analytics")
client.execute("CREATE DATABASE IF NOT EXISTS analytics")
client.execute("""
CREATE TABLE IF NOT EXISTS analytics.monthly_sales (
    month String,
    category String,
    revenue Float64,
    orders UInt32
)
ENGINE = MergeTree()
ORDER BY (month, category)
""")
data = [
    (row["month"], row["category"], float(row["revenue"]), int
(row["orders"]))
    for _, row in monthly.iterrows()
]
client.execute(
    "INSERT INTO analytics.monthly_sales (month, category, revenue,
orders) VALUES",
    data
)
print("Данные успешно загружены в ClickHouse.")
```


Проверка в ClickHouse

Теперь данные доступны для аналитических запросов. Можно проверить, все ли загрузилось правильно:

```
rows = client.execute("""
SELECT month, category, revenue
FROM analytics.monthly_sales
ORDER BY month, category
LIMIT 10
""")
for r in rows:
    print(r)
Результат:
('2024-01', 'Бытовая техника', 452000.0)
('2024-01', 'Книги', 38400.0)
('2024-01', 'Одежда', 231000.0)
('2024-01', 'Электроника', 712000.0)
...
```

Эти данные уже можно использовать в Power BI, Tableau или для отчетов.

Запуск ETL по расписанию

Чтобы пайплайн выполнялся ежедневно, можно добавить расписание.

```
import schedule
import time

def run_etl():
    print("Запуск ETL...")
    # Здесь можно вызвать все шаги: extract, transform, load
    # Например, объединить их в одну функцию run_pipeline()
    run_pipeline()
schedule.every().day.at("07:00").do(run_etl)
while True:
    schedule.run_pending()
    time.sleep(60)
```


Альтернатива — использование `cron` (Linux/macOS) или планировщика задач (Windows).

5.6.3. Основы хранилищ данных: OLAP, кубы и партиции

В предыдущей части мы собрали первый полноценный пайплайн: извлекли заказы из PostgreSQL, обработали их в Pandas и загрузили в ClickHouse. Теперь данные — в аналитическом хранилище, где мы можем быстро строить отчеты. Но чтобы понимать, почему ClickHouse и BigQuery работают в разы быстрее, чем PostgreSQL, нужно разобраться, как устроены хранилища аналитических данных.

OLTP и OLAP

Все базы данных можно разделить на два больших типа: **OLTP** (Online Transaction Processing) — для транзакций, **OLAP** (Online Analytical Processing) — для аналитики.

Кратко:

- OLTP — это про "много маленьких операций";
- OLAP — про "мало, но огромных запросов".

Задача	Тип системы
Пользователь оформляет заказ	OLTP
Склад обновляет остатки	OLTP
Аналитик строит отчет по продажам за месяц	OLAP
Руководитель хочет видеть выручку по категориям	OLAP

Почему аналитика требует OLAP

Когда вы делаете запрос вроде:

```
SELECT category, SUM(amount)
FROM orders
GROUP BY category;
```

PostgreSQL должен прочитать каждую строку таблицы и просуммировать вручную. Если таблица на 50 млн строк — это займет секунды или даже минуты. ClickHouse, наоборот, хранит данные по столбцам, поэтому ему не нужно читать всю строку — он просто считывает нужный столбец `amount` и делает мгновенную агрегацию.

Кубы данных

Когда аналитики говорят "куб данных" (data cube), они имеют в виду не трехмерный объект, а структуру, которая позволяет смотреть на данные по разным измерениям одновременно.

Представьте, что у вас есть таблица продаж:

Дата	Регион	Категория	Выручка
2025-01-01	Москва	Электроника	120000
2025-01-01	СПб	Одежда	54000
2025-02-01	Москва	Книги	9800
2025-02-01	СПб	Электроника	89000

Из нее можно построить OLAP-куб:

- измерения (dimensions) — время, регион, категория;
- метрика (measure) — выручка.

Теперь можно "проваливаться" в куб по любому измерению:

- посмотреть выручку по регионам;
- потом по категориям;
- потом по месяцам.

```
SELECT
    region,
    category,
    toYYYYMM(order_date) AS month,
    sum(amount) AS revenue
```

```
FROM orders
GROUP BY region, category, month
ORDER BY month, revenue DESC;
```

Такой запрос может обрабатывать миллионы строк за секунду — все благодаря тому, что данные хранятся в столбцах, а не в строках, как в OLTP-базах.

Куб — это не отдельный формат, а агрегированная таблица, которая хранит уже рассчитанные метрики.

Пример в ClickHouse:

```
CREATE MATERIALIZED VIEW IF NOT EXISTS analytics.mv_sales_cube
ENGINE = SummingMergeTree()
ORDER BY (region, category, toYYYYMM(order_date))
POPULATE AS
SELECT
    region,
    category,
    toYYYYMM(order_date) AS month,
    sum(amount) AS revenue,
    count() AS orders
FROM default.orders
GROUP BY region, category, month;
```

Теперь при каждой новой вставке в `orders` куб автоматически обновляется. BI-инструмент (например, Power BI или Metabase) может подключиться к `analytics.mv_sales_cube` и мгновенно строить отчеты.

Партиционирование

Когда данных становится слишком много (сотни миллионов строк), работа с таблицами без партиционирования замедляется.

Партиционирование (partitioning) — это способ разбить таблицу на части. Каждая часть (partition) хранит данные за определенный период (например, месяц).

Пример в ClickHouse

```
CREATE TABLE analytics.orders_partitioned (
    order_date Date,
    category String,
    amount Float64
)
ENGINE = MergeTree()
PARTITION BY toYYYYMM(order_date)
ORDER BY (order_date, category);
```

Теперь если мы запрашиваем данные за март 2025 года:

```
SELECT sum(amount)
FROM analytics.orders_partitioned
WHERE order_date BETWEEN '2025-03-01' AND '2025-03-31';
```

ClickHouse прочитает только одну партицию, а не всю таблицу. Это ускоряет запросы в десятки раз и экономит ресурсы.

Партиции и хранение

Тип партиции	Пример	Когда использовать
По дате	toYYYYMM(order_date)	Почти всегда (данные по времени)
По региону	region	Если аналитика по географии
По пользователю	customer_id	Для персональных отчетов

Кубы и партиции — это не просто "ускорители", а инструменты для масштабируемой аналитики.

- Кубы дают мгновенные агрегаты.
- Партиции позволяют обрабатывать только нужные данные.

Вместе они обеспечивают идеальный баланс между скоростью и гибкостью.

5.6.4. Инструменты для больших объемов данных: ClickHouse, DuckDB, BigQuery

В предыдущей части главы мы разобрались, как устроены аналитические хранилища — чем отличаются OLTP и OLAP, что такое кубы, витрины и партиции. Теперь пришло время поговорить о конкретных инструментах, которые позволяют эффективно работать с большими объемами данных в реальных проектах.

Мир аналитики сегодня не ограничивается одной базой. Для разных задач применяются разные системы: где-то важна скорость, где-то — масштабируемость, где-то — простота локального анализа.

В этой части мы разберем три инструмента, которые каждый аналитик должен знать:

- **ClickHouse** — мощное локальное и кластерное OLAP-хранилище (мы уже с ним работали);
- **DuckDB** — мини-OLAP для ноутбуков и ноутбуков Jupyter;
- **BigQuery** — облачное хранилище от Google, масштабируемое и "без серверов".

ClickHouse — локальный и производительный OLAP

Мы уже использовали ClickHouse как основное аналитическое хранилище в нашем пайплайне. Теперь закрепим ключевые принципы, которые делают его лидером среди open-source решений для больших данных.

Сильные стороны ClickHouse:

- Хранение данных по столбцам — считываются только нужные колонки.
- Скорость агрегации — запросы на миллионы строк выполняются за миллисекунды.
- Поддержка партиций и индексов — ускоряет фильтрацию по времени или регионам.
- Масштабируемость — легко разворачивается в кластере или облаке.
- Совместимость с SQL — не требует изучения нового языка.

- Интеграция с Python и Pandas — через библиотеку `clickhouse-driver`.

Пример — анализ данных по регионам

```
SELECT
    region,
    toYYYYMM(order_date) AS month,
    sum(amount) AS revenue,
    count() AS orders
FROM orders
WHERE order_date >= '2024-01-01'
GROUP BY region, month
ORDER BY month, revenue DESC;
```

Такой запрос может работать на таблице с десятками миллионов строк, и ClickHouse справится с ним за доли секунды.

Лучше всего подходит для:

- аналитических витрин, отчетов и BI;
- real-time аналитики (логов, кликов, заказов);
- хранения агрегированных данных после ETL.

Не подходит для:

- транзакционных операций (INSERT/UPDATE/DELETE в реальном времени);
- задач, где важно ACID-поведение, как в банковских системах.

DuckDB — легкий OLAP для локальной аналитики

Если ClickHouse — это серверная система, то DuckDB — его "мини-версия" для ноутбуков и локальных проектов. DuckDB — это аналитическая база, встроенная прямо в Python и работающая без сервера, как SQLite, но с возможностями OLAP.

- Легкость — не требует установки сервера, работает в одном файле.

- Скорость — поддерживает колоночное хранение.
- Совместимость — работает напрямую с CSV, Parquet, Pandas.
- Идеален для ноутбуков Jupyter — можно выполнять SQL и сразу анализировать результат.

Пример — анализ CSV без Pandas

```
import duckdb
# Подключение (в памяти)
con = duckdb.connect()
# Читаем и анализируем CSV
query = """
SELECT
    category,
    COUNT(*) AS orders,
    SUM(amount) AS revenue
FROM 'orders.csv'
GROUP BY category
ORDER BY revenue DESC
"""
df = con.execute(query).fetchdf()
print(df)
```

DuckDB при этом не загружает весь файл в память — он обрабатывает данные построчно и мгновенно возвращает агрегаты.

Пример — использование с Pandas и Parquet

```
import duckdb
import pandas as pd
# Читаем данные из Parquet
df = pd.read_parquet("orders.parquet")
# Выполняем SQL-запрос прямо по DataFrame
result = duckdb.query("""
    SELECT category, AVG(amount) AS avg_amount
    FROM df
```



```
GROUP BY category
""").to_df()
print(result)
```

Уникальность DuckDB в том, что он "понимает" DataFrame как SQL-таблицу. Это делает его идеальным инструментом для локальной аналитики и ноутбуков.

Подходит для:

- локальных проектов и обучения;
- работы с CSV/Parquet на ноутбуке;
- прототипирования ETL и аналитических моделей.

Не подходит для:

- работы в команде (нет многопользовательского режима);
- потоковой обработки и больших данных (все в одной машине).

BigQuery — аналитика без серверов (serverless)

BigQuery — это полностью управляемое облачное хранилище данных от Google. В нем не нужно настраивать сервер, выделять память или следить за нагрузкой — все масштабируется автоматически. BigQuery идеально подходит для компаний, у которых большие объемы данных и распределенная инфраструктура.

BigQuery хранит данные в распределенной архитектуре (Data Lake внутри Google Cloud). Вы платите только за объем обработанных данных при выполнении запроса.

Если ваш SQL-запрос проходит по 1 ТБ данных, это стоит около \$5. То есть можно обрабатывать терабайты без вложений в серверы.

Пример запроса в BigQuery:

```
SELECT
  customer_region,
```



```
EXTRACT(YEAR FROM order_date) AS year,
SUM(amount) AS revenue,
COUNT(*) AS orders_count
FROM
  'myproject.sales.orders'
WHERE
  order_date >= '2024-01-01'
GROUP BY
  customer_region, year
ORDER BY
  revenue DESC;
```

BigQuery автоматически оптимизирует запрос, использует партиции и кеш, а результат можно сразу визуализировать в Google Data Studio.

BigQuery ML

BigQuery может обучать модели без Python.

Например, простая линейная регрессия:

```
CREATE OR REPLACE MODEL 'myproject.ml.revenue_forecast'
OPTIONS(model_type='linear_reg') AS
SELECT
  EXTRACT(MONTH FROM order_date) AS month,
  SUM(amount) AS revenue
FROM
  'myproject.sales.orders'
GROUP BY month;
```

Потом можно делать прогноз:

```
SELECT * FROM ML.PREDICT(MODEL 'myproject.ml.revenue_forecast', (
  SELECT 13 AS month
));
```

Таким образом, BigQuery объединяет аналитику, хранение и машинное обучение в одном инструменте.

Подходит для:

- компаний с большими объемами данных (терабайты и петабайты);
- централизованных хранилищ и BI-отчетов;
- машинного обучения и прогнозов на SQL.

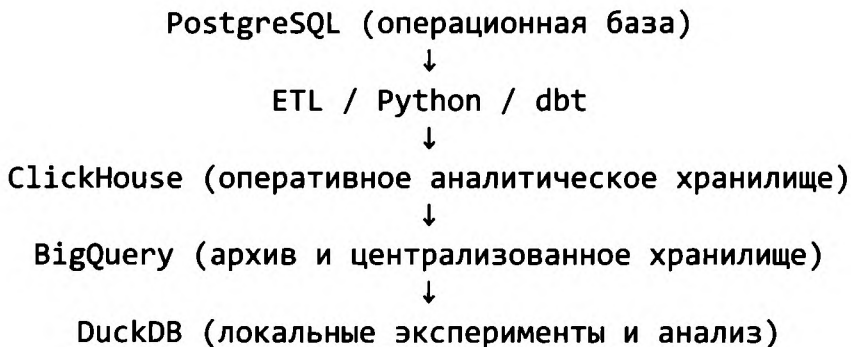
Не подходит для:

- локальной разработки без Интернета;
- задач с микросекундными задержками (не real-time).

Как эти инструменты работают вместе

В реальных проектах инструменты не конкурируют, а дополняют друг друга.

Пример архитектуры:



Таким образом, можно использовать:

- ClickHouse — для отчетов и real-time аналитики;
- BigQuery — для долгосрочного хранения и машинного обучения;
- DuckDB — для быстрой локальной аналитики и проверки гипотез.

5.6.5. Итоги раздела

- ◆ Мы познакомились с тремя ключевыми инструментами для работы с большими данными — ClickHouse, DuckDB и BigQuery.
- ◆ Поняли, что ClickHouse — это мощное и быстрое OLAP-хранилище для отчетов и real-time аналитики.
- ◆ DuckDB — легкий локальный инструмент, идеально подходящий для анализа CSV и Parquet прямо в ноутбуке, без серверов.
- ◆ BigQuery — облачное хранилище от Google, которое позволяет обрабатывать терабайты данных без настройки инфраструктуры.
- ◆ Разобрались, что эти инструменты не конкурируют, а дополняют друг друга:
 - ClickHouse — для оперативной аналитики;
 - BigQuery — для централизованного хранения;
 - DuckDB — для локальных исследований.
- ◆ Современная архитектура аналитики строится именно на таких связках — гибких, масштабируемых и удобных как для инженеров, так и для аналитиков.

5.7. Практический проект: автоматизированная аналитическая система

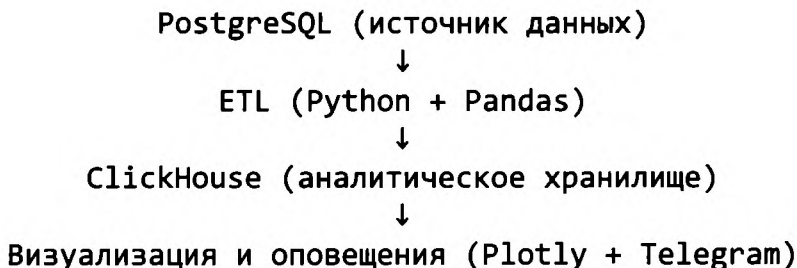
Мы прошли долгий путь: научились работать с базами данных, писать SQL-запросы, строить ETL-пайплайны, анализировать и визуализировать данные, а также автоматизировать рутинные процессы.

Теперь пришло время объединить все это в единый рабочий проект — создать мини-аналитическую систему, которая ежедневно собирает данные, считает метрики, обновляет дашборд и отправляет уведомления в случае падения ключевых показателей.

Этот проект — упрощенная, но реалистичная версия того, что делают аналитические команды в компаниях: автоматические отчеты, ежедневные мониторинги и системы раннего оповещения (alerting).

5.7.1. Архитектура проекта

Наша система будет состоять из четырех ключевых компонентов:



Каждый день (например, в 07:00) система:

1. Загружает новые данные из PostgreSQL.
2. Обработывает их в Pandas — считает выручку, заказы, конверсию, новых клиентов.
3. Сохраняет результаты в ClickHouse.
4. Обновляет дашборд.
5. Проверяет метрики: если конверсия упала — отправляет сообщение в Telegram.

Подготовка данных

Для примера используем базу **orders** из предыдущих глав. Если ее нет — сгенерируем заново, не забудьте запустить PostgreSQL:

```
from sqlalchemy import create_engine, text
import pandas as pd
engine = create_engine("postgresql://postgres:@localhost:5432")
with engine.begin() as conn:
    conn.execute(text("DROP TABLE IF EXISTS orders;"))
```

```

conn.execute(text("""
    CREATE TABLE orders (
        id SERIAL PRIMARY KEY,
        customer_id INTEGER,
        amount NUMERIC(10,2),
        order_date DATE,
        device TEXT,
        source TEXT
    );
"""))

# Генерация синтетических данных
devices = ["Desktop", "Mobile", "Tablet"]
sources = ["Organic", "Ads", "Referral"]
start_date = datetime(2024, 1, 1)
data = []
for _ in range(50000):
    customer_id = random.randint(1, 3000)
    amount = round(random.uniform(500, 20000), 2)
    order_date = start_date + timedelta(days=random.randint(0, 400))
    device = random.choice(devices)
    source = random.choice(sources)
    data.append((customer_id, amount, order_date, device, source))
df = pd.DataFrame(data, columns=["customer_id", "amount", "order_date", "device", "source"])
df.to_sql("orders", engine, if_exists="append", index=False)
print("Таблица orders успешно создана и заполнена.")

```

Теперь база готова для автоматического анализа.

Сбор данных (Extract)

```

from sqlalchemy import create_engine
import pandas as pd
engine = create_engine("postgresql://postgres:@localhost:5432")
# Выбираем данные за последние 300 дней
query = """
SELECT id, customer_id, amount, order_date, device, source
FROM orders

```

```
WHERE order_date >= CURRENT_DATE - INTERVAL '300 day';
"""
df = pd.read_sql(query, engine)
print("Данных за последние 30 дней:", len(df))
```

Этот запрос можно будет выполнять ежедневно — система будет всегда работать с "свежим" окном данных.

Обработка и расчет метрик (Transform)

Теперь рассчитаем ключевые метрики:

- Количество заказов.
- Выручка.
- Средний чек (AOV).
- Уникальные клиенты.
- Конверсия (число клиентов, оформивших заказ / общее число посетителей).

```
import numpy as np
# Преобразуем даты
df["order_date"] = pd.to_datetime(df["order_date"])
# Группировка по дням
daily = (
    df.groupby("order_date")
    .agg(
        revenue=("amount", "sum"),
        orders=("id", "count"),
        unique_customers=("customer_id", pd.Series.nunique)
    )
    .reset_index()
)
daily["avg_check"] = daily["revenue"] / daily["orders"]
daily["visitors"] = 10000
daily["conversion"] = daily["unique_customers"] / daily["visitors"]
print(daily.tail(5))
```

Загрузка в ClickHouse (Load)

Храним все рассчитанные метрики в ClickHouse — это удобно для исторического анализа.

```
from clickhouse_driver import Client
client = Client(host="localhost", user="analytics",
password="secret123", database="analytics")
client.execute("CREATE DATABASE IF NOT EXISTS analytics")
client.execute("""
CREATE TABLE IF NOT EXISTS analytics.daily_metrics (
    date Date,
    revenue Float64,
    orders UInt32,
    avg_check Float64,
    unique_customers UInt32,
    conversion Float64
)
ENGINE = MergeTree()
ORDER BY date
""")
# Вставляем данные
data = [
    (
        row["order_date"].date(),
        float(row["revenue"]),
        int(row["orders"]),
        float(row["avg_check"]),
        int(row["unique_customers"]),
        float(row["conversion"])
    )
    for _, row in daily.iterrows()
]
client.execute("INSERT INTO analytics.daily_metrics VALUES", data)
print("Метрики успешно загружены в ClickHouse.")
```

Визуализация

Создадим простой интерактивный дашборд на Plotly, который обновляется каждый день.

```
import plotly.graph_objects as go
fig = go.Figure()
fig.add_trace(go.Scatter(
    x=daily["order_date"],
    y=daily["revenue"],
    mode="lines+markers",
    name="Выручка",
    line=dict(color="royalblue", width=2)
))
fig.add_trace(go.Scatter(
    x=daily["order_date"],
    y=daily["conversion"] * 100,
    mode="lines",
    name="Конверсия (%)",
    yaxis="y2",
    line=dict(color="lightseagreen", width=2, dash="dot")
))
fig.update_layout(
    title="Ежедневные метрики интернет-магазина",
    xaxis_title="Дата",
    yaxis=dict(title="Выручка, ₽"),
    yaxis2=dict(title="Конверсия, %", overlaying="y", side="right"),
    template="plotly_white",
    legend=dict(orientation="h", yanchor="bottom", y=1.02,
xanchor="right", x=1)
)
fig.show()
```

Такой дашборд можно размещать в Streamlit или Dash, чтобы он обновлялся автоматически при запуске ETL-пайплайна.

Уведомления в Telegram

Если показатели резко падают, система должна уведомить команду. Для этого можно использовать Telegram Bot API.

Создайте бота с помощью [@BotFather](#) и получите token и chat_id.

```
import requests
BOT_TOKEN = "YOUR_TOKEN"
CHAT_ID = "YOUR_CHAT_ID"
# Проверим последнюю конверсию
last_conv = daily["conversion"].iloc[-1]
prev_conv = daily["conversion"].iloc[-2]
delta = (last_conv - prev_conv) / prev_conv
if delta < -0.15:
    message = f"Конверсия упала на {abs(delta)*100:.1f}%!\nВчера: {prev_conv:.2%}, Сегодня: {last_conv:.2%}"
    requests.post(
        f"https://api.telegram.org/bot{BOT_TOKEN}/sendMessage",
        data={"chat_id": CHAT_ID, "text": message}
    )
    print("Уведомление отправлено:", message)
else:
    print("Метрики стабильны.")
```

Такой мониторинг можно дополнить проверками других метрик — например, падение выручки, скачок отказов или рост доли мобильного трафика.

Автоматизация процесса

Чтобы пайплайн выполнялся каждый день:

```
0 7 * * * /usr/bin/python3 /home/user/etl/shop_monitor.py >> /var/
log/shop_monitor.log 2>&1
```

Полный листинг главы

Ниже приведен полный листинг кода для работы с нашей автоматизированной аналитической системой.

Листинг 5.24. Полный листинг файла `shop_monitor.py`

```

"""
# postgresql: укажите свою базу, пользователя и пароль при
необходимости
POSTGRES_DSN = "postgresql://postgres:@localhost:5432/postgres"
# clickhouse: параметры подключения из главы про docker
CLICKHOUSE_HOST = "localhost"
CLICKHOUSE_USER = "analytics"
CLICKHOUSE_PASSWORD = "secret123"
CLICKHOUSE_DB = "analytics"
# Окно данных для анализа (дней от текущей даты назад)
WINDOW_DAYS = 300
# Куда сохранять html-дашборд (откроется обычным браузером)
DASHBOARD_HTML_PATH = "/tmp/dashboard_shop.html"
# telegram: вставьте свой токен и chat_id; если оставить пустым –
отправка будет пропущена
TELEGRAM_BOT_TOKEN = ""
TELEGRAM_CHAT_ID = ""
# Порог алерта по падению конверсии: -15% к вчерашнему значению
CONVERSION_DROP_THRESHOLD = -0.15
# Сколько "посетителей" считаем в примере (в реальном проекте
подставляйте фактическое число)
ASSUMED_VISITORS_PER_DAY = 10_000
# ===== импорт =====
import os
import random
from datetime import datetime, timedelta
import numpy as np
import pandas as pd
import requests
from sqlalchemy import create_engine, text
from clickhouse_driver import Client
import plotly.graph_objects as go
# ===== вспомогательные функции =====
def ensure_orders_table_with_data(engine):
    with engine.begin() as conn:

```

```

conn.execute(text("""
    CREATE TABLE IF NOT EXISTS orders (
        id SERIAL PRIMARY KEY,
        customer_id INTEGER,
        amount NUMERIC(10,2),
        order_date DATE,
        device TEXT,
        source TEXT
    );
"""))
cnt = conn.execute(text("SELECT COUNT(*) FROM orders;")).
scalar()
if cnt and cnt > 0:
    print(f"таблица orders уже существует, записей: {cnt}")
    return
print("генерируем синтетические данные в orders...")
devices = ["Desktop", "Mobile", "Tablet"]
sources = ["Organic", "Ads", "Referral"]
start_date = datetime(2024, 1, 1)
rows = []
for _ in range(50_000):
    customer_id = random.randint(1, 3_000)
    amount = round(random.uniform(500, 20_000), 2)
    order_date = start_date + timedelta(days=random.randint(0,
400))

    device = random.choice(devices)
    source = random.choice(sources)
    rows.append((customer_id, amount, order_date.date(),
device, source))
df = pd.DataFrame(rows, columns=["customer_id", "amount",
"order_date", "device", "source"])
df.to_sql("orders", engine, if_exists="append", index=False)
print("таблица orders успешно создана и заполнена.")
def extract_orders(engine, window_days):
    query = f"""
    SELECT id, customer_id, amount, order_date, device, source
    FROM orders

```

```

WHERE order_date >= CURRENT_DATE - INTERVAL '{int(window_days)}
} day';
"""

df = pd.read_sql(query, engine)
print(f"загружено записей за последние {window_days} дней:
{len(df)}")
return df

def transform_to_daily_metrics(df):
    if df.empty:
        cols = ["order_date", "revenue", "orders", "avg_check",
"unique_customers", "conversion"]
        return pd.DataFrame(columns=cols)
    df["order_date"] = pd.to_datetime(df["order_date"])
    daily = (
        df.groupby("order_date")
        .agg(
            revenue=("amount", "sum"),
            orders=("id", "count"),
            unique_customers=("customer_id", pd.Series.nunique),
        )
        .reset_index()
        .sort_values("order_date")
    )

    daily["avg_check"] = daily["revenue"] / daily["orders"]
    daily["visitors"] = ASSUMED_VISITORS_PER_DAY
    daily["conversion"] = daily["unique_customers"] / daily["visitors"]
    print("пример последних 5 дней:")
    print(daily.tail(5))
    return daily

def load_to_clickhouse(daily):
    client = Client(
        host=CLICKHOUSE_HOST,
        user=CLICKHOUSE_USER,
        password=CLICKHOUSE_PASSWORD,
        database=CLICKHOUSE_DB,
    )

```

```

client.execute("CREATE DATABASE IF NOT EXISTS analytics")
client.execute("""
    CREATE TABLE IF NOT EXISTS analytics.daily_metrics (
        date Date,
        revenue Float64,
        orders UInt32,
        avg_check Float64,
        unique_customers UInt32,
        conversion Float64
    )
    ENGINE = MergeTree()
    ORDER BY date
""")
records = [
    (
        pd.to_datetime(row["order_date"]).date(),
        float(row["revenue"]),
        int(row["orders"]),
        float(row["avg_check"]),
        int(row["unique_customers"]),
        float(row["conversion"]),
    )
    for _, row in daily.iterrows()
]
if records:
    client.execute("INSERT INTO analytics.daily_metrics VALUES",
records)
    print(f"в clickhouse загружено строк: {len(records)}")
else:
    print("нет данных для загрузки в clickhouse")
def build_and_save_dashboard(daily, out_path=DASHBOARD_HTML_PATH):
    if daily.empty:
        print("пропускаем визуализацию – нет данных")
        return out_path

fig = go.Figure()
fig.add_trace(go.Scatter(

```

```

        x=daily["order_date"],
        y=daily["revenue"],
        mode="lines+markers",
        name="выручка",
        line=dict(width=2)
    ))
    fig.add_trace(go.Scatter(
        x=daily["order_date"],
        y=daily["conversion"] * 100,
        mode="lines",
        name="конверсия, %",
        yaxis="y2",
        line=dict(width=2, dash="dot")
    ))
    fig.update_layout(
        title="ежедневные метрики интернет-магазина",
        xaxis_title="дата",
        yaxis=dict(title="выручка, ₽"),
        yaxis2=dict(title="конверсия, %", overlaying="y",
        side="right"),
        template="plotly_white",
        legend=dict(orientation="h", yanchor="bottom", y=1.02,
        xanchor="right", x=1),
        margin=dict(l=40, r=40, t=60, b=40)
    )
    fig.write_html(out_path, include_plotlyjs="cdn", full_html=True)
    print(f"дашборд сохранен в: {out_path}")
    return out_path

def send_telegram_alert_if_needed(daily):
    if not TELEGRAM_BOT_TOKEN or not TELEGRAM_CHAT_ID:
        print("telegram не настроен (token/chat_id пустые) – пропускаем
отправку")
        return
    if len(daily) < 2:
        print("недостаточно точек для сравнения конверсии – пропускаем
отправку")
        return

```



```

last_conv = float(daily["conversion"].iloc[-1])
prev_conv = float(daily["conversion"].iloc[-2])
if prev_conv == 0:
    print("предыдущая конверсия равна нулю – пропускаем отправку")
    return
delta = (last_conv - prev_conv) / prev_conv
if delta < CONVERSION_DROP_THRESHOLD:
    message = (
        f"конверсия упала на {abs(delta) * 100:.1f}%\n"
        f"вчера: {prev_conv:.2%}\n"
        f"сегодня: {last_conv:.2%}"
    )
    url = f"https://api.telegram.org/bot{TELEGRAM_BOT_TOKEN}/
sendMessage"
    resp = requests.post(url, data={"chat_id": TELEGRAM_CHAT_ID,
"text": message}, timeout=10)
    if resp.ok:
        print("уведомление отправлено в telegram")
    else:
        print(f"ошибка отправки в telegram: {resp.status_code}
{resp.text}")
    else:
        print("метрики стабильны – уведомление не требуется")
def main():
    """основной сценарий: подготовка -> extract -> transform -> load
-> dashboard -> alert"""
    print("=== старт etl ===")
    pg_engine = create_engine(PGSQL_DSN, future=True)
    ensure_orders_table_with_data(pg_engine)
    df_orders = extract_orders(pg_engine, WINDOW_DAYS)
    daily = transform_to_daily_metrics(df_orders)
    load_to_clickhouse(daily)
    html_path = build_and_save_dashboard(daily, DASHBOARD_HTML_PATH)
    send_telegram_alert_if_needed(daily)
    print("=== etl завершен ===")
    print(f"откройте дашборд в браузере: file://{html_path}")
if __name__ == "__main__":
    main()

```

5.7.2. Итоги раздела

- ◆ Мы собрали единый аналитический пайплайн: от базы PostgreSQL до ClickHouse, визуализации и уведомлений.
- ◆ Реализовали три ключевых шага ETL-процесса — извлечение, очистку и загрузку данных.
- ◆ Научились автоматически считать ежедневные метрики: выручку, заказы, средний чек, конверсию.
- ◆ Сохранили результаты в ClickHouse — теперь метрики можно использовать для исторического анализа и BI-отчетов.
- ◆ Построили простой дашборд на Plotly, который обновляется вместе с пайплайном.
- ◆ Добавили автоматическое оповещение в Telegram, чтобы реагировать на падение метрик в реальном времени.
- ◆ Настроили автоматический запуск по расписанию (cron / Prefect).
- ◆ Получилась рабочая мини-система, объединяющая все, чему мы научились: базы данных, Python, аналитику, визуализацию и автоматизацию.

5.8. Базы данных в эпоху искусственного интеллекта

Еще совсем недавно разговор о базах данных в аналитике был предсказуемым. Есть источники, есть ETL, есть хранилище, есть SQL и отчеты. Аналитик задает вопросы данным, данные отвечают цифрами. Если нужно что-то сложнее — строится витрина, добавляется еще один слой агрегации. Мир был линейным, а вопросы — заранее известными.

Теперь все выглядит иначе.

Представим обычную ситуацию.

Руководитель пишет аналитику в мессенджере: "Почему в апреле упали продажи? Только коротко и по делу". Пять лет назад это означало несколько часов работы: проверить дашборды, собрать метрики, посмотреть разрезы, сформулировать гипотезы. Сегодня он ожидает ответ почти сразу — как от умного ассистента. И этот ассистент должен не просто показать цифры, а связать их с контекстом: отзывами клиентов, логами ошибок, изменениями цен, маркетинговыми кампаниями.

В этот момент база данных перестает быть просто хранилищем фактов. Она становится источником смысла.

Другой пример.

В компании хранятся сотни тысяч текстовых отзывов. Формально они лежат в таблице: `review_id`, `text`, `rating`, `date`. SQL отлично справляется с подсчетом среднего рейтинга. Но когда бизнес спрашивает: "Какие проблемы клиенты описывают чаще всего и какие из них новые?" — классическая база данных молчит. Она не понимает, что "долго везли", "курьер опоздал" и "ждал доставку три дня" — это одно и то же. Для человека — очевидно. Для SQL — три разных строки.

Именно здесь в аналитический стек входят искусственный интеллект и новые типы баз данных. Текст превращается в вектор, смысл становится измеримым, а поиск — семантическим. Теперь можно спросить не "Где слово *"доставка"*?", а "Где жалуются на проблемы с доставкой?", и получить релевантный результат.

Но важно другое: AI не заменяет базы данных. Он делает их роль более критичной. Любая модель — от простой кластеризации до большого языкового модели — живет на данных. Если данные плохо организованы, не проверены, не связаны между собой, никакой интеллект не спасет. Красивый чат-интерфейс будет уверенно объяснять бизнесу причины, которых на самом деле нет.

В эпоху AI базы данных становятся:

- источником правды для моделей;
- памятью для интеллектуальных систем;
- механизмом контроля и верификации ответов.

Раньше аналитик писал SQL, чтобы получить результат. Теперь SQL часто становится промежуточным шагом: запрос генерирует модель, данные возвращаются из хранилища, дополнительные знания подтягиваются из векторной базы, а финальный ответ формируется в виде текста, понятного человеку. Но ответственность за то, что именно вернулось из базы, никуда не исчезает — она только возрастает.

Эта глава — не про "модные технологии" и не про будущее ради будущего. Она про практику. Про то, как меняется работа с данными, когда к ним подключаются модели. Про новые типы баз данных и старые, которые внезапно получили вторую жизнь. Про архитектурные решения, которые уже сегодня работают в компаниях, и про ошибки, которые совершают почти все, кто слишком рано поверил в магию искусственного интеллекта.

Если раньше аналитик задавал вопросы базе данных, то теперь база данных — это то, что позволяет искусственному интеллекту задавать правильные вопросы и не врать в ответах.

5.8.1. Почему классических баз данных больше недостаточно

Долгое время базы данных в аналитике выполняли понятную и ограниченную роль. Они хранили факты и быстро отвечали на заранее определенные вопросы. Сколько заказов? Какая выручка? Какой средний чек? Под эти задачи идеально подходили реляционные базы, OLTP-системы для транзакций и OLAP-хранилища для отчетов. Мир был устроен так, что сначала появлялся вопрос, затем под него проектировалась модель данных, и только после этого писался запрос.

Эта логика хорошо работала в эпоху BI.

Аналитик строил витрину, бизнес смотрел дашборд, решения принимались на основе агрегированных показателей. Если появлялся новый вопрос, добавлялась новая метрика или разрез. База данных была пассивной: она не "понимала", что в ней хранится, она лишь аккуратно и быстро считала.

С приходом машинного обучения роль данных начала меняться. Данные стали использоваться не только для отчетов, но и для обучения моделей. Теперь было важно не просто хранить историю заказов, а уметь извлекать из нее признаки: частоту покупок, сезонность, поведенческие паттерны. База данных превратилась в источник сырья для моделей, но все еще оставалась в основном табличной и числовой. Текст, изображения и события существовали где-то на периферии — в логах, файлах, отдельных хранилищах.

Настоящий перелом произошел с приходом генеративных моделей.

GenAI ожидает от данных совсем другого. Модель не спрашивает: "Какой средний чек по сегменту?" Она спрашивает: "На что чаще всего жалуются клиенты и что в этом месяце пошло не так?" Это не агрегатный вопрос. Это вопрос про смысл, контекст и связь разрозненных сигналов. Классическая база данных к таким вопросам не готова.

OLTP и OLAP по-прежнему прекрасно делают свою работу. OLTP гарантирует целостность транзакций. OLAP быстро считает большие объемы данных. Проблема не в том, что они устарели. Проблема в том, что они были спроектированы для мира, где данные — это строки и числа, а вопросы — четкие и формализуемые.

Рассмотрим пример.

Компания хранит отзывы клиентов в PostgreSQL. Таблица выглядит аккуратно: идентификатор, текст отзыва, оценка, дата. Годами эта таблица использовалась одинаково. Считали количество отзывов, средний рейтинг, NPS по периодам. Иногда добавляли фильтр по ключевым словам: "доставка", "цена", "поддержка". Это давало ощущение контроля, но только на поверхности.

В какой-то момент бизнес задает вопрос: "Какие новые проблемы появились у клиентов за последний месяц и насколько они серьезны?"

PostgreSQL честно хранит все отзывы, но не знает, что два абзаца текста описывают одну и ту же боль. Он не видит, что тон отзыва изменился, даже если оценка осталась прежней. Он не понимает, что проблема "не могу войти в приложение" вчера была редкой, а сегодня стала массовой.

Чтобы ответить на этот вопрос, данные нужно не просто посчитать — их нужно интерпретировать. Тексты нужно превратить в представление, с которым можно работать на уровне смысла. Появляется кластеризация отзывов,

поиск похожих жалоб, выявление новых тем. Появляется чат-бот, который может ответить менеджеру: "За последнюю неделю резко выросло число жалоб на авторизацию после обновления приложения. Ранее эта тема почти не встречалась".

В этот момент становится очевидно: база данных участвует в рассуждении модели. Она не просто отдает строки — она предоставляет контекст, из которого модель делает выводы. Данные начинают "учить" систему, а не просто лежать в таблицах.

Это фундаментальное изменение. Данные больше не являются конечным продуктом аналитики. Они становятся частью диалога между человеком и моделью. Вопросы перестают быть заранее известными, а ответы — заранее подготовленными. Классическая схема "таблица → запрос → отчет" ломается, потому что между таблицей и ответом появляется интеллект.

Именно поэтому классических баз данных становится недостаточно. Не потому, что они плохи, а потому, что на них возложили задачи, для которых они не были созданы. В следующих разделах мы разберем, какие новые типы хранения и поиска появляются, как они дополняют привычные хранилища и почему аналитик больше не может позволить себе думать о данных только как о таблицах.

5.8.2. Новые типы данных

В классической аналитике данные почти всегда имели очевидную форму. Число, дата, строка фиксированной длины. Даже текст, если он появлялся в базе, воспринимался как нечто вторичное: поле, по которому можно сделать LIKE или посчитать количество символов. Его ценность заканчивалась там, где начинался смысл.

Искусственный интеллект перевернул это представление.

Текст, изображение или аудио больше не считаются "неструктурированными" в привычном смысле. Они стали векторизуемыми. Это значит, что любой сложный объект можно превратить в набор чисел, который отражает

не форму, а содержание. Причем так, что два объекта с похожим смыслом будут близки друг к другу в этом числовом пространстве.

Для аналитика это изменение сопоставимо с появлением дат и чисел в базах данных много лет назад. Появился принципиально новый тип данных — `embedding`.

Embedding — это числовое представление смысла. Обычно это вектор из сотен или тысяч чисел. По отдельности эти числа ничего не значат, но вместе они кодируют контекст: о чем текст, что изображено на картинке, какой характер у звука.

Главное — `embeddings` можно сравнивать между собой. И именно сравнение, а не чтение, становится ключевой операцией.

Рассмотрим простой пример. Есть две фразы:

- "Клиент недоволен доставкой".
- "Заказ привезли слишком поздно".

Для человека очевидно, что смысл у них один. Для базы данных — это разные строки. Но если прогнать обе фразы через модель и получить `embedding`, расстояние между этими векторами будет небольшим. Машина впервые начинает "соглашаться" с человеком на уровне смысла, а не букв.

```
embedding = model.encode("Клиент недоволен доставкой")
```

С этого момента текст перестает быть просто текстом. Он становится данными, с которыми можно работать так же системно, как с числами.

Тот же принцип работает с изображениями. Возьмем интернет-магазин. У каждого товара есть фотография. Раньше картинка была просто файлом, максимум — объектом для ручной разметки. Теперь изображение можно превратить в `embedding` и искать не по тегам, а по визуальному сходству. Покупатель смотрит на кроссовки — система находит похожие модели, даже если они лежат в другой категории и описаны другими словами.

Для аналитика это открывает новые возможности. Например, можно анализировать, какие визуальные стили товаров продаются лучше, не через категории, придуманные людьми, а через реальные сходства изображений.

Еще более неожиданный кейс — логи и события. Логи традиционно считаются чисто техническими данными. Их либо читают инженеры, либо агрегируют по уровням ошибок. Но лог — это тоже текст. А значит, его можно векторизовать. В результате появляется возможность искать аномалии не по заранее заданным правилам, а по смысловому отклонению. Если система "видит", что новые сообщения логов не похожи на обычные, это сигнал проблемы, даже если формально ошибок еще нет.

Важно понимать: `embedding` — это не замена исходных данных. Никто не предлагает выбрасывать тексты или изображения.

`Embedding` — это дополнительный слой представления. Как индекс или витрина, только для смысла. Исходный текст по-прежнему нужен для интерпретации, юридических целей, аудита. Но для поиска, сравнения и анализа используется вектор.

В этот момент аналитическая модель данных усложняется. В одной системе начинают сосуществовать:

- исходные данные (текст, картинки, события);
- числовые метрики;
- `embeddings` как отдельный тип данных.

Это требует другого мышления. Аналитик больше не спрашивает: "Как отфильтровать строки?" Он спрашивает: "Как найти все, что похоже на это?" И ответ на этот вопрос уже не про `WHERE`, а про расстояние между смыслами.

Переход от строк к смыслам — это фундаментальный сдвиг. Он делает возможным семантический поиск, рекомендации, интеллектуальные чаты и анализ, который раньше был доступен только человеку. В следующих разделах мы разберем, где и как эти векторы хранить, как по ним искать и почему без правильной базы данных вся эта магия перестает работать.

5.8.3. Векторные базы данных

В тот момент, когда в аналитическом стеке появляются embeddings, возникает простой и неприятный вопрос: где их хранить и как по ним искать.

Хранить вектор из тысячи чисел в обычной таблице можно. Можно даже написать запрос, который посчитает расстояние между векторами. Но уже на десятках тысяч объектов это превращается в медленный и дорогой эксперимент. А на миллионах — просто перестает работать.

Так появляются векторные базы данных.

По своей сути это системы хранения, оптимизированные не для фильтрации и агрегаций, а для поиска ближайших объектов в пространстве смыслов. Они отвечают не на вопрос "Где значение равно X?", а на вопрос "Что похоже на это?". Причем делают это быстро, даже когда данных очень много.

Для аналитика это принципиально новая возможность. Раньше поиск всегда начинался с условий: дата, категория, статус, точное совпадение строки. Теперь поиск может начинаться с примера, описания или даже ощущения проблемы.

Векторная база данных хранит embeddings и умеет эффективно считать расстояние между ними. На практике это означает, что любой объект — отзыв, инцидент, товар, отчет — можно представить как точку в многомерном пространстве и искать ближайшие точки. Близость означает сходство по смыслу, а не по форме.

Самый очевидный сценарий — семантический поиск. Пользователь вводит запрос обычным языком: "Проблемы с доставкой в регионах". Система превращает запрос в embedding и находит не строки, где встречается слово "доставка", а тексты, где описываются задержки, курьеры, логистика, даже если эти слова не совпадают. Для аналитика это снимает огромный пласт ручной работы по подбору ключевых слов и регулярных выражений.

Следующий важный кейс — RAG, Retrieval Augmented Generation. Большие языковые модели хорошо формулируют ответы, но плохо знают конкретику вашей компании. Векторная база здесь играет роль памяти. Перед тем как модель ответит на вопрос, она получает релевантные куски данных, найденные по смыслу. Аналитик перестает быть единственным посредником между данными и бизнесом, но именно от него зависит, какие данные попадут в этот контекст.

Рекомендательные системы — еще один классический сценарий. "Похожие товары", "похожие клиенты", "похожие ситуации" — все это естественным образом ложится на векторный поиск. Причем сходство может быть сложным: не по одному признаку, а по совокупности поведения, текста отзывов, истории взаимодействий. Для аналитика это означает меньше ручных сегментов и больше адаптивных, зависящих от данных групп.

Отдельно стоит поиск похожих инцидентов. В крупных компаниях истории сбоев, падений метрик и ошибок накапливаются годами. Формально каждый инцидент уникален, но по сути многие из них похожи. Если новый инцидент по смыслу близок к старому, система может подсказать: что тогда сломалось, сколько времени чинили, какие действия помогли. Это аналитика не про отчеты, а про ускорение принятия решений.

Существует несколько популярных решений для векторного поиска. FAISS часто используется как библиотека внутри сервисов и прототипов. Pinecone, Weaviate и Qdrant — полноценные сервисы и базы данных, ориентированные на промышленное использование. PostgreSQL с расширением pgvector занимает особое место: он позволяет добавить векторный поиск в уже существующую инфраструктуру, не ломая привычную модель данных.

Выбор конкретного инструмента — технический вопрос. Гораздо важнее понять, зачем он аналитику.

Рассмотрим практический пример.

Аналитик сталкивается с задачей: "Найти все кейсы, похожие на резкое падение конверсии из-за повышения цены".

В базе есть отчеты, комментарии аналитиков, постмортемы, тикеты в Jira. Формально они все разные. Где-то написано "повышение тарифа", где-то — "изменение прайсинга", где-то — "негативная реакция пользователей". SQL может помочь, если заранее угадать правильные слова. Но если формулировки отличаются, а причина описана косвенно, классический поиск проваливается.

Векторный подход работает иначе. Все эти документы уже превращены в `embeddings`. Запрос аналитика тоже превращается в `embedding`. Дальше база данных находит ближайшие по смыслу кейсы. В результате аналитик получает не длинный список нерелевантных совпадений, а несколько действительно похожих ситуаций, с которыми можно работать.

В этот момент становится ясно, почему SQL здесь бессилен. Он оперирует символами и числами. Векторная база оперирует смыслом. Они не конкурируют — они дополняют друг друга. SQL по-прежнему считает метрики и строит факты. Векторный поиск находит контекст и опыт, накопленный в данных.

Для аналитика векторные базы данных — это не про машинное обучение как таковое. Это про доступ к знаниям, которые раньше были спрятаны в текстах и не поддавались системному анализу. И как только такие базы появляются в стеке, меняется сам характер вопросов, которые аналитик может задавать данным.

5.8.4. Трансформация SQL

Каждый раз, когда появляется новая волна технологий, SQL хоронят. Так было с NoSQL, с Big Data, с машинным обучением, теперь — с LLM. Но на практике происходит обратное: SQL не исчезает, он расширяет свою роль. Он перестает быть просто языком выборки данных и становится языком управления аналитическим и интеллектуальным контекстом.

В современной аналитике SQL все чаще оказывается в центре связки "человек — модель — данные".

Раньше аналитик сам формулировал запрос. Теперь запрос может быть сгенерирован моделью. Но проверять его, понимать и адаптировать по-прежнему должен человек. И именно здесь SQL оказывается незаменимым: он остается прозрачным, проверяемым и воспроизводимым способом описания логики работы с данными.

Первое важное изменение — SQL + ML

Во многих системах обучение и применение моделей все ближе подвигаются к данным. Признаки считаются прямо в хранилище, скоринги записываются в таблицы, результаты моделей становятся частью витрин. Для аналитика это означает, что SQL используется не только для анализа прошлого, но и для работы с предсказаниями.

Например, в таблице заказов появляется поле `churn_probability`, рассчитанное моделью. SQL-запросы начинают оперировать не только факта-

ми, но и вероятностями. Аналитик может фильтровать клиентов по риску, агрегировать прогнозы, сравнивать их с реальным поведением. База данных становится местом, где встречаются статистика и бизнес-решения.

Следующий шаг — SQL + embeddings

Как только embeddings начинают храниться рядом с привычными данными, SQL получает новую функцию: управлять семантическим поиском. Запрос теперь может содержать не только условия, но и операции близости смыслов.

Простейший пример выглядит почти обыденно:

```
SELECT *  
FROM reviews  
ORDER BY embedding <-> :query_embedding  
LIMIT 5;
```

Но за этой простотой скрывается радикальное изменение. SQL-запрос больше не говорит базе: "Найди точные совпадения". Он говорит: "Найди то, что ближе всего по смыслу". Для аналитика это означает, что привычный инструмент начинает работать с новым типом данных, не теряя своей читаемости.

Важно, что SQL здесь не пытается заменить векторную базу. Он выступает как управляющий слой. Аналитик может добавить фильтры по дате, региону, продукту, а затем уже внутри этого подмножества выполнить семантический поиск. Контроль остается в руках человека, а не полностью у модели.

Дальше появляется связка SQL + LLM

Аналитик задает вопрос обычным языком: "Покажи отзывы клиентов, которые жалуются на рост цен за последний месяц".

LLM превращает этот вопрос в SQL-запрос. Внутри него — фильтр по дате, векторный поиск по embeddings, сортировка по релевантности. Аналитик видит сгенерированный запрос и может его проверить. Это принципиальный момент. Модель помогает ускорить работу, но не скрывает логику.

В реальной практике это выглядит так. Модель предлагает запрос, аналитик замечает, что фильтр по региону отсутствует, и добавляет его. Или понимает, что ограничение по дате слишком широкое. SQL остается языком договоренности между человеком и системой.

Появляется новый кейс: SQL как язык управления AI-поиском. Не модель решает, какие данные допустимы, а SQL-запрос. Именно в нем зашиты правила безопасности, бизнес-ограничения, логика сегментации. AI работает внутри рамок, заданных запросом, а не наоборот.

Изменение роли аналитика

Раньше ценностью было умение написать сложный запрос. Теперь ценностью становится умение прочитать, понять и критически оценить запрос, сгенерированный моделью. Аналитик превращается в редактора и архитектора логики. Он отвечает за то, чтобы результат был корректным, воспроизводимым и соответствовал бизнес-смыслу.

SQL в этой новой реальности — не пережиток прошлого, а точка опоры. Он позволяет связать быстрый, иногда хаотичный интеллект моделей с точной, проверяемой логикой данных. Пока аналитике нужна прозрачность и доверие, SQL будет жить. Просто теперь он говорит не только о строках и столбцах, но и о смыслах.

5.8.5. Архитектура аналитики с AI

Когда в компании появляются AI-сценарии, первое искушение — "прикрутить чат" поверх существующих отчетов. На короткой дистанции это может выглядеть эффектно, но очень быстро становится ясно: без правильной архитектуры ответы либо поверхностные, либо неверные, либо им просто перестают доверять. Современная аналитика с AI — это не один инструмент, а согласованный стек, в котором у каждого слоя есть своя четкая роль.

Начнем с источников данных.

Источники почти не изменились по сравнению с классической аналитикой.

Это транзакционные системы, CRM, маркетинговые платформы, логи, события приложений, внешние данные. Отличие в том, что теперь важны не только числа и статусы, но и "сырые" формы данных: тексты обращений, описания инцидентов, комментарии менеджеров, скриншоты, документы. То, что раньше считалось вторичным и плохо пригодным для анализа, теперь становится ценным.

Все эти данные по-прежнему сходятся в центральное хранилище — DWH или Lakehouse. Этот слой остается основой стека. Именно здесь обеспечивается целостность, история, единые определения метрик. AI не отменяет необходимость в аккуратно собранных фактах. Более того, чем умнее становятся модели, тем болезненнее любая ошибка в данных.

В Lakehouse-подходе часто хранятся и структурированные таблицы, и полуструктурированные данные — например, JSON с событиями или тексты отзывов. Это удобно: аналитик может связать числовые метрики с исходным контентом, который потом будет использоваться моделями.

Следующий слой — Feature Store

Он появляется не во всех компаниях, но там, где есть ML или GenAI, быстро становится критичным. Feature Store — это место, где хранятся переиспользуемые признаки: рассчитанные метрики, агрегаты, поведенческие показатели. Важный момент — одни и те же признаки используются и в дашбордах, и в моделях, и в ответах LLM. Это снижает риск расхождений и "двух версий правды".

Например, показатель "средняя задержка доставки за 30 дней" существует в одном виде и для аналитического отчета, и для модели, которая объясняет рост оттока.

Параллельно появляется Vector DB

Она редко заменяет DWH, чаще живет рядом с ним. Векторная база хранит `embeddings` текстов, документов, изображений, логов. Связь между слоями обычно осуществляется через идентификаторы: `embedding` знает, какому объекту в хранилище он соответствует.

Это позволяет комбинировать точные числовые данные из DWH и смысловой поиск из Vector DB в одном сценарии.

Дальше — слой моделей: ML и LLM

Классические ML-модели решают конкретные задачи: прогнозы, скоринг, классификация. LLM выступают в роли универсального интерфейса и "сборщика" ответов. Они не заменяют расчеты и не хранят данные. Их задача — понять вопрос, выбрать правильную стратегию поиска и сформулировать ответ для человека.

И наконец, пользовательский слой — BI и чат-интерфейсы

BI никуда не исчезает. Дашборды по-прежнему нужны для регулярного мониторинга. Чат дополняет их, закрывая нерегулярные, исследовательские вопросы. В зрелых системах пользователь может перейти от чата к конкретному отчету и наоборот, не теряя контекста.

Теперь соберем это в один пример.

Пользователь задает вопрос в чате: "Почему в этом месяце упала конверсия в заказах из мобильного приложения?"

LLM сначала интерпретирует вопрос. Он понимает, что нужны:

- метрики конверсии;
- разрез по каналу;
- объяснение причин.

Дальше он формирует план: SQL-запрос к DWH для получения чисел и векторный поиск по отзывам, инцидентам и логам за нужный период.

SQL-запрос уходит в хранилище и возвращает факт: конверсия действительно снизилась, особенно после определенной даты. Параллельно векторный запрос идет в Vector DB и находит тексты, похожие по смыслу на "проблемы с покупкой", "ошибка оплаты", "цена выросла".

LLM получает оба результата и связывает их. Он не придумывает цифры — он опирается на данные. В ответе пользователь видит не только объяснение, но и конкретные показатели, на которые можно сослаться.

В результате аналитический стек начинает работать как единая система рассуждения. DWH отвечает за факты, Vector DB — за контекст, модели — за интерпретацию, а SQL остается тем механизмом, который держит все это в управляемых рамках.

Такая архитектура сложнее классической, но она масштабируется вместе с вопросами бизнеса. И самое важное — она не заменяет аналитика, а усиливает его роль, превращая из автора отдельных запросов в архитектора целой системы понимания данных.

5.8.6. Feature Store

В классической аналитике одно и то же число часто живет несколькими жизнями. В дашборде оно считается одним запросом, в модели — другим, в презентации — третьим. Пока вопросы простые, это почти незаметно. Но как только в систему приходят ML-модели и LLM, расхождения начинают множиться. Разные определения, разные окна, разные фильтры — и в итоге разные ответы на один и тот же вопрос.

Feature Store появляется как реакция именно на эту проблему.

Его идея проста: если показатель уже был однажды корректно рассчитан и согласован, он не должен пересчитываться заново каждым инструментом. Он должен стать переиспользуемым знанием. Не временным результатом запроса, а полноценным объектом данных со своим именем, описанием и правилами обновления.

Для аналитика это означает смену фокуса. Вместо того чтобы каждый раз писать сложный SQL, он начинает проектировать признаки — *features*. Эти признаки живут дольше одного отчета и используются в разных контекстах.

Рассмотрим конкретный пример.

В компании, занимающейся доставкой, появляется показатель `avg_delivery_delay_30d` — средняя задержка доставки за последние 30 дней. Сначала он появляется в аналитике. Аналитик аккуратно определяет логику: какие заказы считаются доставленными, как обрабатываются отмены, какое время берется за основу. Показатель проверяется, обсуждается с бизнесом и начинает использоваться в дашборде.

На этом месте в классической аналитике история бы закончилась.

Но дальше этот же показатель нужен команде data science для модели оттока. Они могли бы посчитать его заново, немного иначе, под свои нужды. Формально это быстрее. Но тогда модель начинает жить в своей реальности, а дашборды — в своей. Когда менеджер спрашивает, почему модель считает клиента "рискованным", аналитик и data scientist начинают спорить о формулах.

Feature Store разрывает этот порочный круг. `avg_delivery_delay_30d` хранится как единый признак. Его использует и дашборд, и модель. Когда показатель меняется, это изменение отражается везде. Аналитика и ML начинают говорить на одном языке, потому что буквально используют одни и те же данные.

И на этом история не заканчивается.

Теперь появляется LLM-интерфейс. Менеджер задает вопрос в чате: "Почему клиенты в этом регионе чаще уходят?"

LLM формирует ответ, опираясь на данные. Среди них — тот самый `avg_delivery_delay_30d`. Модель может сказать: "В этом регионе средняя задержка доставки за последние 30 дней на 40% выше, чем в среднем по компании. Это один из факторов повышенного оттока".

Ключевой момент в том, что LLM не вычисляет этот показатель на лету и не придумывает его. Он использует уже существующий признак из Feature Store. Это делает ответ воспроизводимым и проверяемым. Аналитик может открыть дашборд и увидеть те же цифры.

Feature Store, таким образом, становится мостом между аналитикой, моделями и AI-интерфейсами. Это не просто технический компонент, а инструмент управления знаниями о бизнесе. В нем фиксируется не только значение, но и смысл показателя.

Для аналитика это означает рост ответственности. Теперь важно не просто "посчитать правильно", а посчитать так, чтобы результат можно было использовать многократно и безопасно. Хороший feature — это актив компании. Плохой — источник постоянных недоразумений.

В эпоху AI Feature Store превращается в коллективную память аналитической системы. Именно он позволяет моделям быть умными, не теряя связи

с реальностью, и именно через него знания перестают быть одноразовыми запросами и становятся частью устойчивой архитектуры.

5.8.7. RAG

Когда в компании впервые появляется LLM, почти всегда возникает соблазн: подключить модель к чату и "пусть отвечает". Первые ответы выглядят впечатляюще. Модель уверенно формулирует мысли, звучит разумно, быстро реагирует. Но проходит совсем немного времени — и начинаются проблемы. Ответы перестают совпадать с цифрами в отчетах, появляются обобщения без ссылок на данные, а иногда и откровенные ошибки. Бизнес быстро теряет доверие к такому "умному чату".

Причина проста: модель рассуждает, но не знает ваших данных.

RAG, Retrieval Augmented Generation, появился как практический способ решить именно эту проблему. Его идея не в том, чтобы сделать модель умнее, а в том, чтобы поставить ее в правильный контекст. Модель не придумывает ответы из головы, а собирает их на основе реальных данных, заранее найденных и отобранных системой.

Важно понять: RAG — это архитектурный подход, а не конкретная технология.

В классическом чате LLM получает вопрос и сразу генерирует ответ. В RAG между вопросом и ответом появляется этап извлечения данных. Модель сначала ищет релевантную информацию, а уже потом формулирует объяснение. Это принципиально меняет качество и надежность результата.

Рассмотрим реальный аналитический сценарий.

Менеджер спрашивает: "Почему вырос churn в апреле?"

Без RAG модель может выдать правдоподобное рассуждение: сезонность, конкуренты, цена. Все это может звучать разумно, но не иметь отношения к конкретной компании. С RAG процесс выглядит иначе.

Сначала вопрос анализируется и разбивается на подзадачи. Нужно понять, что такое churn в данном бизнесе, какой период интересует, какие факторы потенциально влияют. Затем система ищет данные. SQL-запросы идут в DWH за метриками.

Векторный поиск — в отчеты аналитиков, отзывы клиентов, описания инцидентов, логи изменений продукта.

Модель получает не абстрактное знание, а конкретный набор фактов: рост оттока в определенных сегментах, увеличение задержек доставки, негативные отзывы после обновления, изменения цен. И только после этого формируется ответ.

В результате менеджер видит не просто текст, а объяснение, опирающееся на данные. Аналитик может проверить каждое утверждение. Это принципиально другой уровень доверия.

Для аналитика RAG особенно ценен тем, что он позволяет масштабировать аналитику без потери контроля. Не нужно заранее строить отчет под каждый возможный вопрос. Достаточно обеспечить доступ к корректным данным и правильно настроить механизм извлечения. Вопросы могут быть новыми, но ответы всегда будут опираться на существующую базу знаний.

Еще один важный момент — RAG дисциплинирует данные.

Если отчеты плохо структурированы, если тексты не имеют источников, если метрики считаются по-разному, RAG сразу начинает "сыпаться". Это болезненно, но полезно. Он быстро показывает слабые места в аналитическом контуре. Хороший RAG невозможен без хорошей аналитики.

На практике RAG часто использует все, о чем мы говорили в предыдущих главах:

- SQL управляет выборкой числовых данных.
- Feature Store предоставляет согласованные показатели.
- Vector DB находит релевантные тексты и кейсы.
- LLM связывает все это в осмысленный ответ. Ни один из этих компонентов по отдельности не решает задачу. Работает только связка.

Именно поэтому RAG — это не "умный чат", а аналитическая система нового типа. Она не заменяет аналитика и не отменяет необходимость в данных. Она делает знания доступными быстрее, шире и безопаснее. И чем сложнее бизнес, тем заметнее становится разница между красивыми словами и ответами, основанными на реальных данных.

5.8.8. Итоги раздела

- ◆ Базы данных в эпоху искусственного интеллекта перестают быть пассивным хранилищем. Они становятся активным участником аналитики, источником контекста и памяти для моделей.
- ◆ Классические OLTP и OLAP по-прежнему необходимы, но их уже недостаточно. В аналитический стек входят *embeddings*, векторные базы данных и Feature Store, потому что бизнес все чаще задает вопросы не про числа, а про смысл и причины.
- ◆ SQL не исчезает — он меняется. Он остается языком контроля, прозрачности и договоренности между человеком и AI. Через SQL аналитик управляет тем, какие данные модели видят и как именно они используются.
- ◆ Современная архитектура аналитики строится вокруг связки данных и моделей. RAG-подход показывает ключевой принцип: модель не должна придумывать ответы, она должна извлекать их из данных и объяснять человеку.
- ◆ Роль аналитика в этой системе усиливается. Он отвечает не только за расчеты, но и за качество контекста, переиспользование знаний и доверие к результатам. Именно это отличает работающую AI-аналитику от красивого, но бесполезного "умного чата".

Часть VI.

Заключение и дальнейшие шаги

6.1. Аналитик данных: роли, специализации и путь роста

Когда человек начинает изучать аналитику данных, у него почти всегда возникает ощущение неопределённости. SQL уже понятен, *pandas* не пугает, статистика постепенно укладывается в голове – но остаётся главный вопрос: кем именно я стану на рынке труда? Название «аналитик данных» выглядит единым, но на практике за ним скрываются несколько разных профессий с разными задачами, ожиданиями и траекториями развития.

По данным крупных рекрутинговых платформ, сегодня под названием *data analyst* публикуется лишь около 35-40% вакансий, которые действительно соответствуют классическому аналитическому профилю. Остальные маскируют под собой продуктовую аналитику, BI-направление или даже инженерные роли. Поэтому понимание специализаций – это не академический вопрос, а вопрос эффективности карьеры и скорости роста дохода.

6.1.1. Аналитик данных как профессия

Независимо от специализации, почти у всех аналитиков есть общее профессиональное ядро. Это умение работать с данными из баз, формулировать вопросы, превращать сырые таблицы в выводы и интерпретировать результаты. Практика показывает, что около 70% повседневных задач аналитика – это не сложная математика и не машинное обучение, а корректная работа с данными: выборка, фильтрация, агрегации, контроль качества и объяснение результатов бизнесу. Именно это ядро и формирует фундамент, на котором дальше строятся специализации.

6.1.2. Data analyst: универсальный аналитик

Классический аналитик данных – это специалист, который работает с уже существующими данными и отвечает на вопросы вида «что происходит?» и «почему это происходит?». В реальных компаниях такой аналитик, чаще всего, взаимодействует с бизнесом напрямую: маркетингом, финансами, операционными командами.

По статистике, около 50% вакансий начального уровня относятся именно к этому профилю. Основной фокус здесь – SQL, pandas, базовая статистика и визуализация. От аналитика ждут умения самостоятельно написать запрос, собрать датасет, проверить его на ошибки и сделать выводы, которые можно использовать для принятия решений.

Карьерная траектория классического аналитика обычно выглядит так: сначала работа с готовыми запросами и отчётами, затем самостоятельная аналитика, позже – ответственность за целые направления (например, маркетинговая аналитика или финансовые показатели). Средний срок выхода с *junior* на *middle* в этом треке составляет 1,5–2 года при регулярной практике.

6.1.3. Product analyst: аналитик, близкий к продукту

Продуктовый аналитик – это аналитик, встроенный в продуктовую команду. Его основная задача – помогать принимать решения о развитии продукта: какие фичи делать, какие убирать, что улучшать и как измерять эффект изменений.

В отличие от универсального аналитика, здесь меньше внимания уделяется разовым отчётам и больше – метрикам, экспериментам и гипотезам. Product analyst работает с воронками, когортами, retention, A/B-тестами. По данным исследований рынка, около 60–65% продуктовых аналитиков ежедневно используют статистические методы проверки гипотез, тогда как среди классических аналитиков этот показатель ближе к 25–30%. Этот путь

требует более глубокого понимания бизнеса и пользовательского поведения. Часто продуктовые аналитики приходят из **data analyst**, постепенно смещая фокус с «что произошло?» на «что делать дальше?». Рост здесь может быть быстрым, но и требования выше: без уверенной статистики и умения общаться с продакт-менеджерами в этом треке сложно.

6.1.4. BI-аналитик: архитектор отчётности

BI-аналитик – это специалист, отвечающий за отчётность и визуализацию данных на уровне компании. Его задача – сделать так, чтобы руководители и команды могли в любой момент увидеть актуальные показатели и не обращаться за каждым вопросом к аналитикам вручную.

В BI-направлении меньше ad-hoc анализа и больше системной работы: построение витрин данных, контроль показателей, поддержка дашбордов. По статистике крупных компаний, до 80% управленческих решений среднего уровня принимаются на основе BI-отчётов, а не разовых исследований.

Этот профиль хорошо подходит тем, кто любит структуру, порядок и долгоживущие решения. Здесь особенно важны навыки SQL, понимание бизнес-метрик и аккуратность. Переход из BI в продуктовую или data science-аналитику возможен, но требует дополнительного обучения статистике и экспериментам.

6.1.5. Data engineer: инженерный путь из аналитики

Хотя **data engineer** формально считается отдельной профессией, многие специалисты приходят туда именно из аналитики. Это путь для тех, кто начинает упираться не в анализ, а в ограничения данных: медленные запросы, грязные источники, отсутствие автоматизации. Инженер данных отвечает за то, чтобы данные вообще появились и были доступны для анализа. Здесь меньше интерпретаций и больше архитектуры: пайплайны, хранилища, автоматические загрузки, контроль качества.

По рынку видно, что спрос на **data engineer** стабильно растёт быстрее, чем на аналитиков: в последние годы количество вакансий увеличивалось на 20–25% в год. Этот путь требует дополнительных технических навыков: Python на уровне разработки, понимание баз данных глубже, работа с инструментами оркестрации. Зато он часто ведёт к более высоким зарплатам и участию в ключевых инфраструктурных решениях.

6.1.6. Как “читать” вакансии

Одна из главных ловушек начинающих аналитиков – верить названию вакансии. На практике нужно читать не заголовок, а описание задач.

- Если в вакансии много про SQL, отчёты и бизнес-метрики – это аналитик данных.
- Если упор на A/B-тесты, продуктовые решения и гипотезы – продуктовая аналитика.
- Если основное – дашборды, KPI и визуализация для руководства – BI.
- Если же говорят про пайплайны, ETL и хранилища – это инженерный трек, даже если написано «аналитик».

Исследования рынка показывают, что кандидаты, которые откликаются только по названию вакансии, получают на 30–40% меньше приглашений на интервью, чем те, кто подбирает вакансии по содержанию задач.

6.1.7. Траектории обучения: как двигаться осознанно

Для большинства начинающих оптимальной стратегией остаётся универсальный старт. Освоение SQL, *pandas*, базовой статистики и работы с базами данных обеспечивает вход в *data analyst* и BI-аналитику. Этот этап обычно занимает от 6 до 12 месяцев при регулярной практике.

Дальше выбор зависит от интересов:

- если тянет к продукту и экспериментам – углубление в статистику, A/B-тесты и метрики;

- если нравится системность и визуализация – развитие в BI и архитектуре данных;
- если начинает раздражать ручная работа и хочется автоматизации – переход в инженерный трек.

Важно понимать, что специализация – это не точка невозврата. Современная аналитика всё чаще ценит гибридные профили: аналитиков, которые понимают продукт, инженеров, которые умеют анализировать, и BI-специалистов, способных формулировать гипотезы.

6.1.8. Итоги по выбору

- Профессия аналитика данных включает несколько специализаций с разными задачами и зонами ответственности.
- Общее ядро аналитики — работа с данными, формулировка вопросов и интерпретация результатов для бизнеса.
- Data analyst чаще всего является стартовой ролью и основой для дальнейшего роста.
- Product, BI и инженерные треки отличаются фокусом: продукт, отчётность или инфраструктура данных.
- Названия вакансий вводят в заблуждение — ключевым является содержание задач.
- Карьерное развитие строится на осознанном выборе траектории и возможности переходов между ролями.

6.2. Заключение

Книга подходит к завершению. Мы последовательно рассмотрели основные компоненты аналитической работы с данными: хранение, преобразование, анализ, визуализацию, моделирование и автоматизацию. Через практические примеры и инструменты были показаны ключевые подходы, применяемые в реальных аналитических задачах. Полученный материал об-

разует основу для профессиональной деятельности в области анализа данных и разработки аналитических систем.

Важно понимать, что освоенные методы являются не конечным перечнем, а базовым уровнем. Они дают возможность уверенно работать с данными, но дальнейшее развитие предполагает расширение знаний, углубление в отдельные темы и постоянную практику.

6.3. Дальнейшие шаги

Дальнейший путь зависит от профессиональных интересов, задач рабочего окружения и целей развития. Можно выделить несколько направлений, в рамках которых полученные знания могут развиваться и углубляться.

- Если интерес вызывает работа с бизнес-метриками и пользовательским поведением, логичным продолжением будет изучение продуктовой аналитики, аналитики потребительских решений и методов оценки эффективности изменений. Здесь важно уметь формулировать гипотезы, интерпретировать данные и строить связи между показателями и управленческими решениями.
- Если внимание привлекает моделирование и прогнозирование, можно переходить к машинному обучению и статистическим моделям. Это предполагает изучение алгоритмов классификации, регрессии, методов обучения с учителем и без учителя, анализ временных рядов и методов повышения устойчивости моделей.
- Если приоритет — инфраструктура данных, то следующим этапом является изучение хранилищ данных, распределенных систем обработки и автоматизации конвейеров. В этой области важны принципы надёжности, масштабируемости и контроля качества данных на протяжении всего жизненного цикла.
- Если важны визуализация и отчетность, полезно углубить навыки построения дашбордов, управления показателями и разработки систем мониторинга. Здесь ключевым является умение представлять данные так, чтобы они помогали быстро принимать решения.

Независимо от выбранного направления, фундамент остаётся общим: уверенное владение данными, критическое мышление, способность проверять выводы и формулировать интерпретацию результатов.

6.4. Финальные слова

Эта книга завершена, но работа с данными — это процесс без конечной точки. В дальнейшем знания будут расширяться, а инструменты и методы — совершенствоваться. Главное — сохранять интерес к исследованию и готовность учиться.

Удачи вам в вашей работе с данными. И пусть то, что вы делаете, приносит реальную пользу — вам самим, команде, проекту, бизнесу.

Это и есть главная цель!

Приложение I.

Подготовка окружения для работы с книгой

Это приложение объясняет, как с нуля настроить изолированное окружение для всех примеров книги, включая установку Miniconda, создание окружения, установку нужных библиотек, интеграцию с Jupyter, а также проверку и типичные решения проблем.

Мы будем использовать Miniconda (лёгкую альтернативу Anaconda) и создадим окружение `db-book` с Python 3.11. Внутри него будут:

- базовые инструменты: `python`, `pip`, `ipykernel`, `jupyterlab`;
- аналитика и табличные данные: `pandas`, `numpy`;
- работа с БД и SQL: `sqlalchemy`;
- драйверы СУБД:
 - » PostgreSQL: `psycopg2` (конда-версия; без системной компиляции);
 - » MySQL: `mysql-connector-python` (чисто на Python);
 - » SQLite: встроен в Python (модуль `sqlite3`);
 - » MongoDB: `pymongo`.

Опционально для больших данных: `dask`, а также для визуализации: `matplotlib`.

Установка Miniconda

1. Скачайте инсталлятор Miniconda с официального сайта (версия для вашей ОС, архитектуры x64/ARM, Python 3.11).

2. Установите с настройками по умолчанию. На Windows разрешите **Add Miniconda to my PATH** или используйте **Anaconda Prompt / Miniconda Prompt** для команд.

Проверьте установку в новом терминале:

```
conda --version
```

3. Если команда не находится, откройте **Anaconda/Miniconda Prompt** (Windows) или перезапустите терминал (macOS/Linux).

Создание окружения для книги

Рекомендуется использовать канал *conda-forge* (обновлённые сборки под все ОС):

```
conda config --add channels conda-forge
conda config --set channel_priority strict
# Создаём окружение с нужной версией Python
conda create -n db-book python=3.11
# Активируем окружение
conda activate db-book
```

Устанавливаем нужные пакеты

```
# Базовые инструменты
conda install jupyterlab ipykernel pandas numpy sqlalchemy python-
dotenv matplotlib
# Драйверы СУБД
conda install psycopg2 mysql-connector-python pymongo
```

Регистрируем ядро в Jupyter

```
python -m ipykernel install --user --name db-book --display-name
"Python (db-book)"
```

Теперь в JupyterLab/Notebook вы увидите ядро Python (db-book).

Проверка окружения

Откройте новый ноутбук с ядром **Python (db-book)** и выполните:

```
import sys, sqlite3
import pandas as pd
import sqlalchemy as sa
import psycpg2
import mysql.connector
import pymongo
print(sys.version)
print("OK: pandas", pd.__version__)
print("OK: SQLAlchemy", sa.__version__)
```

Если видите версии без ошибок импорта — база готова.

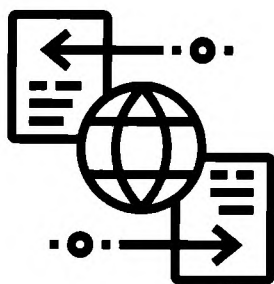
Мини-тест SQLite

```
import sqlite3
conn = sqlite3.connect("shop.db")
cur = conn.cursor()
cur.execute("CREATE TABLE IF NOT EXISTS ping(x INT)")
cur.execute("INSERT INTO ping VALUES (1)")
conn.commit()
print(cur.execute("SELECT COUNT(*) FROM ping").fetchone())
conn.close()
```

Краткая памятка команд

```
# Создание и активация окружения
conda create -n db-book python=3.11
conda activate db-book
# Установка пакетов
```

```
conda install jupyterlab ipykernel pandas numpy sqlalchemy psycpg2
mysql-connector-python pymongo python-dotenv matplotlib
# Регистрация ядра
python -m ipykernel install --user --name db-book --display-name
"Python (db-book)"
# Экспорт/обновление окружения
conda env export > environment.lock.yml
conda env update -f environment.yml --prune
# Удаление окружения
conda deactivate
conda env remove -n db-book
```



Приложение II.

Краткий справочник по SQL

В работе аналитика многие задачи повторяются: выбрать данные, отфильтровать нужное, сгруппировать, посчитать метрику, построить сегментацию, сравнить периоды, вывести топы или собрать воронку.

Со временем у каждого появляется свой набор «любимых» SQL-запросов, к которым он возвращается снова и снова. Это приложение — такой набор, только собранный и структурированный заранее. Здесь собраны типовые шаблоны SQL для анализа данных: от базовых запросов до оконных функций, когортного анализа, RFM и A/B-тестов. Их можно использовать как отправную точку и адаптировать под свои задачи.

Быстрый старт: выборка, фильтрация, сортировка

```
-- Базовая выборка
SELECT [col1], [col2], [col3]
FROM [schema].[table]
WHERE [condition]
ORDER BY [col1] DESC
LIMIT 100;

-- Выборка уникальных значений
SELECT DISTINCT [col]
FROM [table]
ORDER BY [col];
```

```
-- Условная фильтрация по множеству
SELECT *
FROM [table]
WHERE [status] IN ('active','trial')
-- NOT IN (...) для исключений
AND [date] BETWEEN DATE '[YYYY-MM-DD]' AND DATE '[YYYY-MM-DD]';

-- Поиск по подстроке (регистр/диалект)
-- PostgreSQL: ILIKE - регистронезависимый
SELECT *
FROM [table]
WHERE [name] ILIKE '%iphone%';
```

Агрегации и условные подсчёты

```
-- Групповые агрегаты
SELECT [group_col],
       COUNT(*)           AS cnt,
       SUM([amount])      AS revenue,
       AVG([amount])      AS avg_check,
       MIN([amount])      AS min_amt,
       MAX([amount])      AS max_amt
FROM [table]
GROUP BY [group_col]
ORDER BY revenue DESC;

-- Условная агрегация
SELECT [group_col],
       SUM(CASE WHEN [status] = 'paid' THEN 1 ELSE 0 END) AS paid_cnt,
       SUM(CASE WHEN [status] = 'unpaid' THEN 1 ELSE 0 END) AS unpaid_cnt
FROM [table]
GROUP BY [group_col];

-- Конверсия и доли
SELECT
```

```

COUNT(*) FILTER (WHERE [event] = 'view') AS views,
-- PostgreSQL
COUNT(*) FILTER (WHERE [event] = 'buy') AS buys,
1.0 * COUNT(*) FILTER (WHERE [event] = 'buy')
/ NULLIF(COUNT(*) FILTER (WHERE [event] = 'view'),0) AS cr
FROM [events];

-- Универсально через CASE:
SELECT
SUM(CASE WHEN [event]='view' THEN 1 ELSE 0 END) AS views,
SUM(CASE WHEN [event]='buy' THEN 1 ELSE 0 END) AS buys,
1.0 * SUM(CASE WHEN [event]='buy' THEN 1 ELSE 0 END)
/ NULLIF(SUM(CASE WHEN [event]='view' THEN 1 ELSE 0 END),0) AS cr
FROM [events];

```

JOIN-шаблоны

```

-- INNER JOIN: пересечение
SELECT t1.[id], t1.[col], t2.[metric]
FROM [table1] t1
JOIN [table2] t2 ON t2.[id] = t1.[id];

-- LEFT JOIN: «всё из левой + совпадения справа»
SELECT t1.*, t2.[metric]
FROM [table1] t1
LEFT JOIN [table2] t2 ON t2.[id] = t1.[id];

-- анти-JOIN: строки слева без совпадений справа
SELECT t1.*
FROM [table1] t1
LEFT JOIN [table2] t2 ON t2.[id] = t1.[id]
WHERE t2.[id] IS NULL;

SELECT *

```



```

FROM [users] u
WHERE EXISTS (
  SELECT 1
  FROM [orders] o
  WHERE o.[user_id] = u.[id]
);

```

Оконные функции (window functions)

Оконные функции позволяют выполнять вычисления над строками, не объединяя их в группы и не теряя детальность данных. В отличие от GROUP BY, где каждая группа сворачивается в одну строку, оконные функции «смотрят» на соседние строки, но при этом сохраняют каждую запись в выборке. Это делает их незаменимыми для задач ранжирования, расчёта скользящих средних, анализа динамики, сравнения с предыдущими периодами и построения временных метрик.

Они особенно полезны в аналитических запросах, когда важно видеть как отдельные значения, так и их контекст внутри категории или временной последовательности.

```

-- Ранги / порядковые номера
SELECT
  [id],
  [category],
  [amount],
  ROW_NUMBER()
    OVER (PARTITION BY [category] ORDER BY [amount] DESC) AS rn,
  RANK()
    OVER (PARTITION BY [category] ORDER BY [amount] DESC) AS rnk,
  DENSE_RANK()
    OVER (PARTITION BY [category] ORDER BY [amount] DESC) AS drnk

```

```
FROM [table];

-- Скользящие суммы/средние
SELECT
    [date],
    [amount],
    SUM([amount])
        OVER (ORDER BY [date]
              ROWS BETWEEN 6 PRECEDING AND CURRENT ROW) AS ma7_sum,
    AVG([amount])
        OVER (ORDER BY [date]
              ROWS BETWEEN 29 PRECEDING AND CURRENT ROW) AS ma30_avg
FROM [daily_metrics];
```

```
-- LAG/LEAD: сравнение со вчерашним/завтрашним днем
SELECT
    [date],
    [amount],
    LAG([amount]) OVER (ORDER BY [date]) AS prev_amount,
    ([amount] - LAG([amount]) OVER (ORDER BY [date])) AS diff
FROM [daily_metrics];
```

```
-- Процентили (PostgreSQL)
SELECT
    PERCENTILE_CONT(0.5) WITHIN GROUP (ORDER BY [amount]) AS median,
    PERCENTILE_CONT(0.9) WITHIN GROUP (ORDER BY [amount]) AS p90
FROM [orders];
```

Топ-N

```
-- Топ-N по сумме
SELECT [item_id], SUM([amount]) AS revenue
FROM [orders]
GROUP BY [item_id]
ORDER BY revenue DESC
```

```
LIMIT 10;

-- Топ-3 в каждой категории
WITH ranked AS (
    SELECT
        [category],
        [item_id],
        SUM([amount]) AS revenue,
        ROW_NUMBER() OVER (PARTITION BY [category] ORDER BY SUM([amount])
DESC) AS rn
    FROM [orders]
    GROUP BY [category], [item_id]
)
SELECT *
FROM ranked
WHERE rn <= 3
ORDER BY [category], revenue DESC;
```

Дедупликация и выбор «последней версии»

```
-- Оставить последнюю запись по ключу
WITH ranked AS (
    SELECT *,
        ROW_NUMBER()
        OVER (PARTITION BY [user_id] ORDER BY [updated_at] DESC) AS rn
    FROM [user_profiles]
)
SELECT *
FROM ranked
WHERE rn = 1;

-- DISTINCT ON (PostgreSQL): оставить «последнюю»
SELECT DISTINCT ON ([user_id]) *
FROM [user_profiles]
ORDER BY [user_id], [updated_at] DESC;
```

Даты и время

```
-- Усечение даты до месяца (PostgreSQL)
SELECT DATE_TRUNC('month', [created_at]) AS month, COUNT(*) AS cnt
FROM [orders]
GROUP BY 1
ORDER BY 1;

-- MySQL
SELECT DATE_FORMAT([created_at], '%Y-%m-01') AS month, COUNT(*) AS cnt
FROM [orders]
GROUP BY 1
ORDER BY 1;

-- Диапазоны и оффсеты
SELECT *
FROM [orders]
WHERE [created_at] >= NOW() - INTERVAL '30 day';
-- PostgreSQL
-- MySQL: NOW() - INTERVAL 30 DAY
```

Когортный анализ (retention - месяц регистрации × месяц жизни)

```
WITH base AS (
  SELECT
    [user_id],
    MIN(DATE_TRUNC('month', [first_purchase_date])) AS cohort_month
  FROM [orders]
  GROUP BY [user_id]
),
events AS (
  SELECT
```

```

        o.[user_id],
        DATE_TRUNC('month', o.[order_date]) AS order_month
    FROM [orders] o
)
SELECT
    b.cohort_month,
    EXTRACT(YEAR FROM order_month)*12
    + EXTRACT(MONTH FROM order_month)
    - (EXTRACT(YEAR FROM cohort_month)*12
    + EXTRACT(MONTH FROM cohort_month)) AS month_number,
    COUNT(DISTINCT e.[user_id]) AS active_users
FROM base b
JOIN events e ON e.[user_id] = b.[user_id]
GROUP BY 1,2
ORDER BY 1,2;

```

RFM-сегментация (Recency / Frequency / Monetary)

```

WITH ref AS (SELECT MAX([order_date]) AS max_dt FROM [orders]),
agg AS (
    SELECT
        [user_id],
        MAX([order_date])                AS last_dt,
        COUNT(*)                          AS freq,
        SUM([amount])                     AS monetary
    FROM [orders]
    GROUP BY [user_id]
)
SELECT
    a.[user_id],
    DATE_PART('day', (r.max_dt - a.last_dt)) AS recency_days,
    a.freq,
    a.monetary
FROM agg a CROSS JOIN ref r;

```

«Gaps and Islands» (непрерывные последовательности)

В реальных данных часто встречаются последовательности событий, и задача аналитика — определить, где они начинаются и заканчиваются.

Например, дни подряд, когда пользователь заходил в приложение; периоды активности подписки; последовательные транзакции без перерывов.

Такие последовательности называются «островами» (islands), а разрывы между ними — «пропусками» (gaps).

Простые агрегаты и стандартные JOIN не помогают выделить такие непрерывные цепочки, и здесь полезны оконные функции: они позволяют сравнивать текущую запись с предыдущей и определять, продолжается ли последовательность или началась новая. Этот приём используется для построения сессий, анализа вовлечённости, выявления периодов активности и «серий» действий пользователя.

```
-- Пометить «острова» (последовательные дни активности пользователя)
WITH t AS (
  SELECT
    [user_id],
    [event_date],
    ROW_NUMBER()
      OVER (PARTITION BY [user_id] ORDER BY [event_date]) AS rn1,
    ROW_NUMBER()
      OVER (PARTITION BY [user_id] ORDER BY [event_date])
      - EXTRACT(EPOCH FROM [event_date]) / 86400 AS grp
  FROM [daily_events]
)
SELECT [user_id], MIN([event_date]) AS start_dt, MAX([event_date])
AS end_dt, COUNT(*) AS days
FROM t
GROUP BY [user_id], grp
ORDER BY [user_id], start_dt;
```


Сессии: разбиение по таймауту (30 минут)

```

WITH e AS (
  SELECT
    [user_id],
    [ts],
    CASE WHEN [ts] - LAG([ts])
      OVER (PARTITION BY [user_id] ORDER BY [ts]) > INTERVAL '30
minutes'
      OR LAG([ts])
      OVER (PARTITION BY [user_id] ORDER BY [ts]) IS NULL
    THEN 1 ELSE 0
    END AS new_session
  FROM [events]
),
s AS (
  SELECT
    [user_id],
    [ts],
    SUM(new_session) OVER (PARTITION BY [user_id] ORDER BY [ts]) AS
session_id
  FROM e
)
SELECT [user_id], session_id,
  MIN([ts]) AS session_start,
  MAX([ts]) AS session_end,
  COUNT(*) AS events_in_session
FROM s
GROUP BY [user_id], session_id
ORDER BY [user_id], session_start;

```

Воронка

```

WITH steps AS (
  SELECT [user_id],

```

```

        MAX(CASE WHEN [event]='view'
              THEN 1 ELSE 0 END) AS s1_view,
        MAX(CASE WHEN [event]='add_to_cart'
              THEN 1 ELSE 0 END) AS s2_cart,
        MAX(CASE WHEN [event]='purchase'
              THEN 1 ELSE 0 END) AS s3_buy
FROM [events]
WHERE [event_time]
      BETWEEN TIMESTAMP '[start]' AND TIMESTAMP '[end]'
GROUP BY [user_id]
)
SELECT
    SUM(s1_view) AS reached_view,
    SUM(CASE WHEN s1_view=1 AND s2_cart=1 THEN 1 ELSE 0 END) AS
reached_cart,
    SUM(CASE WHEN s2_cart=1 AND s3_buy=1 THEN 1 ELSE 0 END) AS
reached_buy,
    1.0*SUM(CASE WHEN s1_view=1 AND s2_cart=1 THEN 1 ELSE 0 END) /
NULLIF(SUM(s1_view),0) AS cr1,
    1.0*SUM(CASE WHEN s2_cart=1 AND s3_buy=1 THEN 1 ELSE 0 END) /
NULLIF(SUM(s2_cart),0) AS cr2
FROM steps;

```

Работа со строками/JSON (PostgreSQL)

```

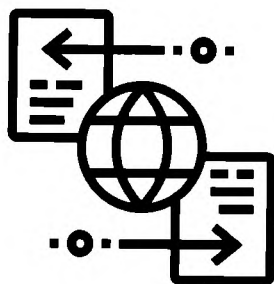
-- Строки
SELECT
    LOWER([name]) AS name_lower,
    REGEXP_REPLACE([phone], '\D', '', 'g') AS phone_digits
FROM [users];

-- JSON: извлечение полей
SELECT
    [payload]->'campaign' AS campaign,
    ([payload]->'meta'->'source') AS source
FROM [events_json];

```


Временные таблицы и CTE (common table expressions)

```
WITH base AS (  
    SELECT [user_id], [order_date], [amount]  
    FROM [orders]  
    WHERE [order_date] >= DATE '[YYYY-MM-DD]'  
)  
daily AS (  
    SELECT [order_date], SUM([amount]) AS revenue  
    FROM base  
    GROUP BY 1  
)  
SELECT *  
FROM daily  
ORDER BY [order_date];
```



Список использованных источников информации

Список использованных источников информации:

1. <https://docs.python.org/3/>
2. https://wiki.postgresql.org/wiki/Performance_Optimization
3. https://pandas.pydata.org/docs/user_guide/enhancingperf.html
4. https://pandas.pydata.org/docs/user_guide/gotchas.html#memory-usage
5. <https://numpy.org/doc/stable/user/basics.optimization.html>
6. <https://docs.python.org/3/library/sqlite3.html>
7. https://www.flaticon.com/ru/free-icon/data-exchange_3957995?term=%D0%B4%D0%B0%D0%BD%D0%BD%D1%8B%D0%B5+%D0%98%D0%98&related_id=3957995
8. <https://wesmckinney.com/book/>
9. <https://dataintensive.net/>
10. <https://realpython.com/>



Издательство «Наука и Техника» выпускает книги более 25 лет!

Уважаемые авторы!

Приглашаем к сотрудничеству по созданию книг
по IT-технологиям, электронике, электротехнике, медицине, педагогике.

Наши преимущества:

- являемся одним из ведущих технических издательств страны;
- выпускаем книги большими тиражами, что положительно влияет на гонорар авторов;
- регулярно переиздаем тиражи, автоматически выплачивая гонорар за *каждый* тираж;
- применяем индивидуальный подход в работе с каждым автором;
- работаем профессионально: от корректуры до авторских дизайн-проектов;
- проводим политику доступной цены;
- имеем собственные каналы сбыта: от федеральных сетей, крупнейших книжных магазинов РФ, ведущих маркетплейсов ОЗОН, Wildberries, Яндекс-Маркет и др. до ведущих библиотек вузов, ссузов.

Ждем Ваши предложения:

- тел. (812) 412-70-26
- эл. почта: nitmail@nit.com.ru

Будем рады сотрудничеству!

Для заказа книг:

➤ интернет-магазин: nit.com.ru

- более 3000 пунктов выдачи на территории РФ, доставка 3–5 дней
- более 300 пунктов выдачи в Санкт-Петербурге и Москве, доставка 1–2 дня
- тел. (812) 412-70-26
- эл. почта nitmail@nit.com.ru

➤ магазин издательства: г. Санкт-Петербург, пр. Обуховской обороны, д. 107

- метро Елизаровская, 200 м за ДК им. Крупской
- ежедневно с 10.00 до 18.00
- справки и заказ: тел. (812) 412-70-26

➤ книжные сети и магазины

- | | |
|---|-------------------------|
| • «Читай-город» - сеть магазинов | тел. +7 (495) 424-84-44 |
| • «Буквоед» - сеть магазинов | тел. +7 (812) 601-0-601 |
| • Московский дом книги – сеть магазинов | тел. +7 (495) 789-35-91 |
| • ТД «БиблиоГлобус» | тел. +7 (495) 781-19-12 |
| • «Амитель» — сеть магазинов | тел. +7 (473) 223-00-02 |
| • Дом книги, г. Екатеринбург | тел. +7 (343) 289-40-45 |
| • Дом книги, г. Нижний Новгород | тел. +7 (831) 246-22-92 |
| • Приморский торговый Дом книги | тел. +7 (423) 263-10-54 |

➤ маркетплейсы ОЗОН, Wildberries, Яндекс-Маркет, Myshop и др.

Измайлов К. К.

БАЗЫ ДАННЫХ НА Python и ИИ

*Статистика, аналитика, большие данные
и машинное обучение*

Группа подготовки издания:

Зав. редакцией компьютерной литературы: *Е.В. Финков*

Редакторы: *О.С. Петрунич, Н.В. Жерлов*

Корректор: *А.В. Громова*

Изображение на обложке использовано с ресурсов freepik.com, vecteezy.com

Все права защищены. Никакая часть данной книги не может быть воспроизведена в какой бы то ни было форме без письменного разрешения владельцев авторских прав.

Издательство не несет ответственности за возможный ущерб, причиненный в ходе использования материалов данной книги, а также за доступность материалов, ссылки на которые вы можете найти в этой книге. На момент подготовки книги к изданию все ссылки на Интернет-ресурсы были действующими. Информация, содержащаяся в данной книге, получена из источников, рассматриваемых издательством как надежные. Тем не менее, имея в виду возможные человеческие или технические ошибки, издательство не может гарантировать абсолютную точность и полноту приводимых сведений и не несет ответственности за возможные ошибки, связанные с использованием книги.

12+

ООО "Издательство Наука и Техника"

ОГРН 1217800116247, ИНН 7811763020, КПП 781101001

192029, г. Санкт-Петербург, пр. Обуховской обороны, д. 107, лит. Б, пом. 1-Н

Подписано в печать 13.03.2026. Формат 70х100 1/16.

Бумага офсетная. Печать офсетная. Объем 25 п.л.

Тираж 2500. Заказ 17292.

Отпечатано с готового оригинал-макета

ООО «Принт-М», 142300, М.О., г.Чехов, ул. Полиграфистов, д.1



Константин Измаилов

Практикующий специалист с многолетним опытом работы в технологических IT-компаниях, прошедший полный путь аналитической карьеры: от прикладной разработки и анализа данных до руководящих ролей в крупных компаниях (VK, Yandex, T-Bank, а также в международных IT-стартапах).

Преподавал в ведущих технических вузах Москвы, автор образовательных программ и онлайн-курсов по Python, аналитике данных и работе с базами данных, ориентированных на практическую подготовку студентов и специалистов.

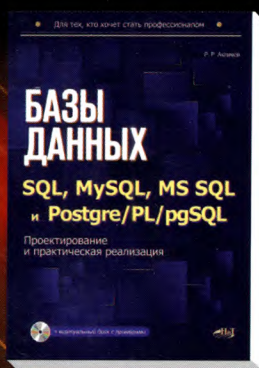
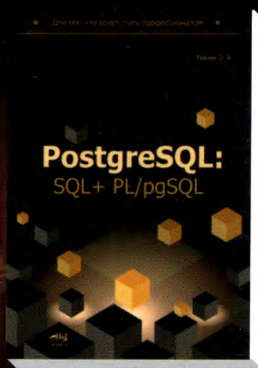
5 причин купить эту книгу:

- Полный жизненный цикл данных: от хранения в базах и написания запросов до анализа, оптимизации и осмысленной интерпретации результатов.
- SQL, SQLite, PostgreSQL, MySQL, MongoDB, SQLAlchemy** и **pandas** на пошаговых примерах в реальных проектах.
- Создание эффективных аналитических **ИИ-моделей**, оптимизация больших объёмов данных, ускорение pandas-кода, использование эффективных форматов хранения.
- Главный акцент на практике: 70% каждой главы – это код с подробными комментариями и пояснениями, который можно сразу копировать, запускать и видеть результат.
- Книга не требует глубоких знаний или предварительного опыта: все темы вводятся постепенно, простым языком, с упором на практику.

издательство

НАУКА и ТЕХНИКА

Санкт-Петербург



Наши книги — ваши инвестиции в будущее

ISBN 978-5-6049658-8-7



9 785604 965887



nit.com.ru
nitbooks
nitbooks



Для заказа книг:
(812) 412-70-26
nitmail@nit.com.ru

