

# Основные виды машинного обучения

Простые данные  
Понятные признаки

**Классическое  
обучение**

Когда качество  
правда проблема

**Ансамбли**

Сложные данные  
Непонятно где  
признаки  
Есть вера в чудо

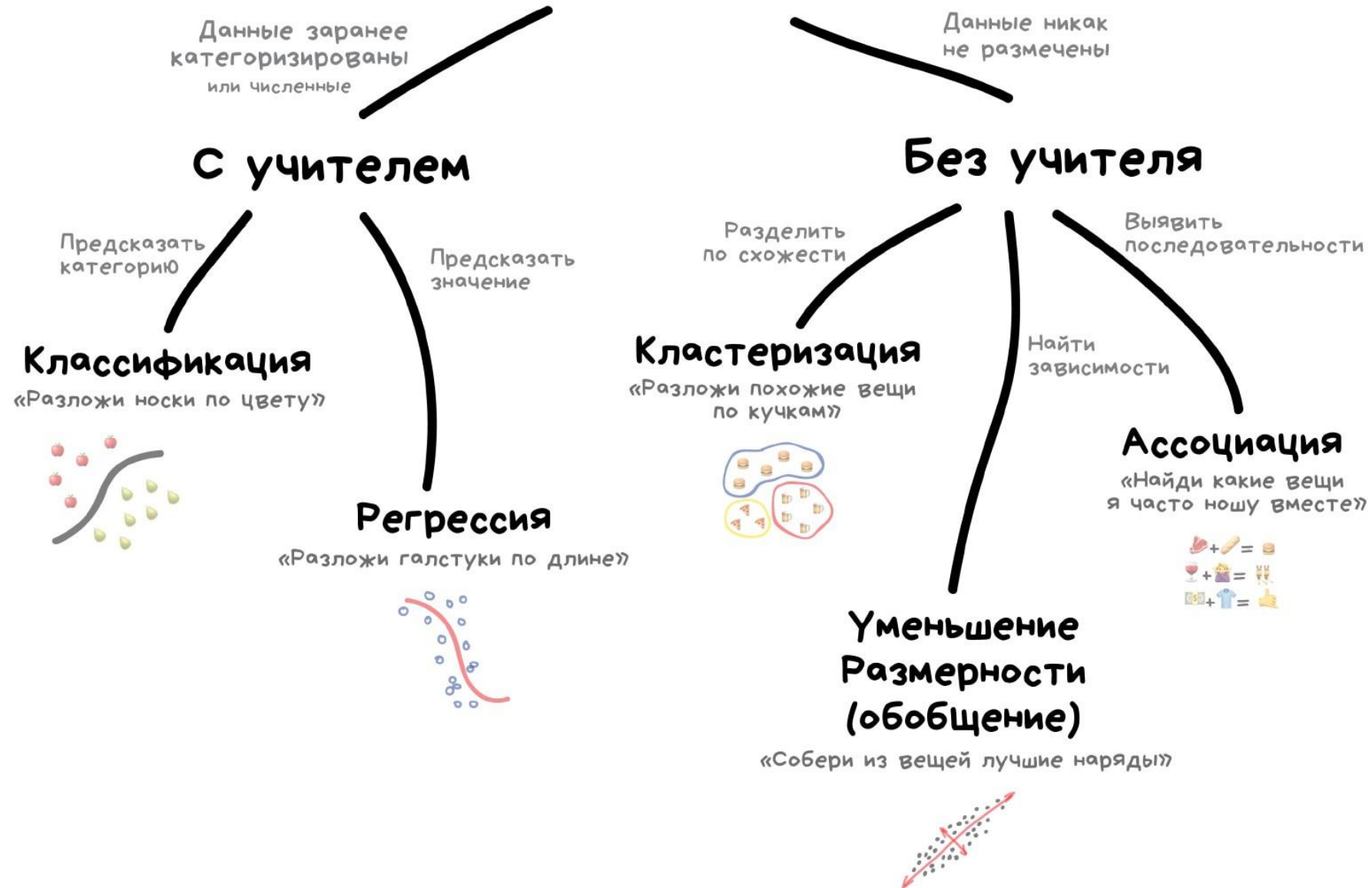
**Нейросети  
и глубокое  
обучение**

Данных нет,  
но есть среда,  
с которой можно  
взаимодействовать

**Обучение  
с подкреплением**

Вечные конкуренты

# Классическое Обучение



**Процедура train test split (разделение на обучающую и тестовую выборки)** — это метод валидации модели, который позволяет оценить, как модель будет работать на новых, ранее невиданных данных. Суть заключается в том, что исходный набор данных делится на две (или три) части:

**Обучающая выборка (training set)** — данные, на которых модель обучается, то есть подбирает внутренние параметры, выявляя зависимости между признаками и целевой переменной.

**Тестовая выборка (testing set)** — данные, которые не использовались при обучении и служат для проверки точности и обобщающей способности модели.

- Иногда дополнительно выделяется **валидационная выборка (validation set)** — промежуточный набор данных, который применяется для настройки гиперпараметров и оптимизации модели во время обучения.

В машинном обучении данные всегда делятся на:

**X (features) — входные признаки, то, на чём модель учится предсказывать;**

**• y (labels / targets) — целевая переменная, то, что нужно предсказать**

Далее происходит разделение данных на две части:

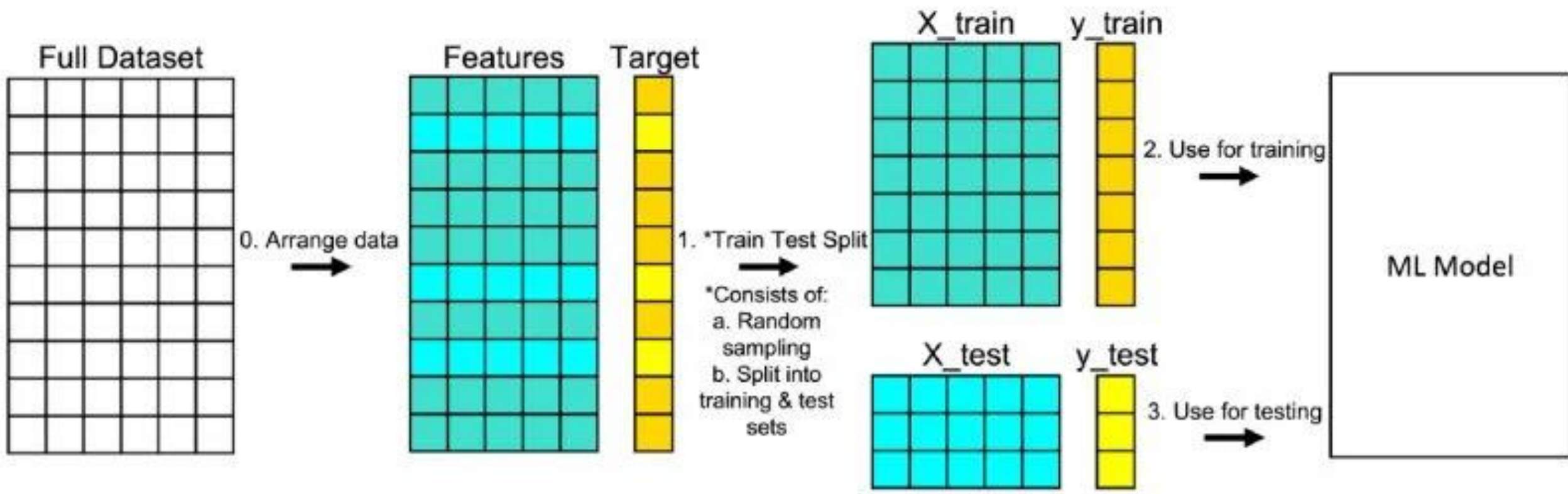
Train set ( $X_{\text{train}}$ ,  $y_{\text{train}}$ ) — обучающая выборка, на которой модель учится выявлять закономерности.

Test set ( $X_{\text{test}}$ ,  $y_{\text{test}}$ ) — тестовая выборка, которую модель не видит во время обучения и которая используется для оценки качества.

Процесс включает:

a. Random sampling — случайный выбор данных, чтобы избежать смещения (bias);

• b. Split ratio — обычно 70–80 % для обучения, 20–30 % для теста.



После применения `train_test_split` мы получаем пары (`X_train`, `y_train`) и (`X_test`, `y_test`).

- **`X_train`** содержит признаки (входные значения) для тренировочных примеров.
- **`y_train`** содержит *соответствующие метки* для этих же примеров — то есть, истинные выходные значения, которые модель должна «научиться» предсказывать.

далее. [Связать с ошибкой](#)

```
[10] 0
✓ сек. ▶ import pandas as pd
      from sklearn.model_selection import train_test_split

      # Создаём DataFrame
      data = pd.DataFrame({
          "Часы_подготовки": [10, 5, 8, 2, 9],
          "Средний_балл": [85, 70, 80, 60, 88],
          "Пропуски": [1, 3, 2, 5, 0],
          "Итоговая_оценка": [90, 65, 82, 55, 93]
      })

      # X – признаки (то, по чему мы предсказываем)
      X = data[["Часы_подготовки", "Средний_балл", "Пропуски"]]

      # y – целевая переменная (что предсказываем)
      y = data["Итоговая_оценка"]
```

```
[11] 0
✓ сек. X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.4, random_state=42)
```

```
[12] 0
✓ сек. ▶ print("X_train:")
      print(X_train)
      print("\n")
      print("y_train:")
      print(y_train)
```

↔ X\_train:

	Часы_подготовки	Средний_балл	Пропуски
2	8	80	2
0	10	85	1
3	2	60	5

y\_train:

2	82
0	90
3	55

Name: Итоговая\_оценка, dtype: int64



Original Data

X <sub>1</sub>	X <sub>2</sub>	X <sub>p</sub>	Y

`train_test_split()`



X<sub>train</sub>

X <sub>1</sub>	X <sub>2</sub>	X <sub>p</sub>

y<sub>train</sub>

Y

X<sub>test</sub>

X <sub>1</sub>	X <sub>2</sub>	X <sub>p</sub>

y<sub>test</sub>

Y

# Настройка гиперпараметров

**Гиперпараметры** — это параметры алгоритма обучения, которые задаются *до начала обучения* и не обновляются непосредственно во время оптимизации модели (в отличие от весов/коэффициентов модели).

Примеры гиперпараметров:

- Скорость обучения (learning rate) в нейронных сетях;
- Максимальная глубина дерева решений;
- Количество скрытых слоёв или нейронов в слое;
- Параметры регуляризации (например, коэффициент  $\lambda$  в L1 / L2).

# Токенизация (Tokenization)

- **Что делает:**

Разбивает текст на отдельные слова (токены).

"Машинное обучение и интеллект"

→ ["машинное", "обучение", "и", "интеллект"]

**Зачем:**

Чтобы модель могла работать с каждым словом отдельно, а не с целой строкой текста.

**Инструменты:**

- `nltk.word_tokenize()`
- `spacy`
- `re.split()` (регулярные выражения)
- `TfidfVectorizer` делает токенизацию сам по умолчанию.

# Приведение к нижнему регистру (Lowercasing)

## Что делает:

Все слова переводятся в строчные буквы.

- **Пример:**

["Машинное", "Обучение"] → ["машинное", "обучение"]

- **Зачем:**

Чтобы избежать дублирования признаков ("Машинное" и "машинное" — это одно и то же слово).

# Удаление стоп-слов (Stopword removal)

## Что делает:

Убирает частые, но неинформативные слова, которые не несут смысловой нагрузки.

## Примеры:

- В русском: “и”, “в”, “на”, “это”, “для”
- В английском: “the”, “is”, “and”, “in”

## Инструменты:

- `nltk.corpus.stopwords`
- `spacy.lang.ru.stop_words`
- `TfidfVectorizer(stop_words=...)`

# Лемматизация (Lemmatization)

## Что делает:

Приводит слово к **нормальной форме** (лемме) — базовому слову из словаря.

## • Пример:

"анализ", "анализа", "анализом" → "анализ"

"машинного" → "машинный"

## Зачем:

Чтобы все формы одного слова считались одним признаком.

## Инструменты:

- `ru morphology2` — популярный морфоанализатор для русского и казахского.
- `spacy` — если нужна лемматизация для нескольких языков.
- `stanza` — многоязычная альтернатива от Stanford NLP.

# Преобразование текстов в векторы

- **Векторизация текста** — это процесс преобразования текста в числовой формат, который могут понимать и обрабатывать алгоритмы машинного обучения. Текстовые данные по своей природе являются категориальными и неструктурированными, из-за этого обучать модели ИИ прямо на тексте - нельзя, их надо векторизовать.

# Лексиконный подход

**Лексиконный подход** (на основе словаря тонально окрашенных слов, например, *SentiWords*) — это метод анализа тональности текста без обучения модели. Он использует заранее составленный **словарь**, где каждому слову приписан **вес тональности** (положительный, отрицательный или нейтральный).

🔧 Принцип работы:

## 1. Текст → токенизация

Разбиваем предложение на отдельные слова.

Пример: "Этот фильм просто потрясающий" → ["этот", "фильм", "просто", "потрясающий"]

## 2.

### Сопоставление со словарём

Для каждого слова ищем значение в словаре:

- "потрясающий" → +0.9
- "просто" → 0.0
- "фильм" → 0.0

## 3.

### Агрегация полярности

Суммируем или усредняем найденные значения:

$$\text{Score} = \frac{\sum_i \text{tone}(w_i)}{N}$$

где  $\text{tone}(w_i)$  — значение тональности слова,  $N$  — общее число слов.

## 4. Интерпретация результата

- Если  $\text{Score} > 0.1$  → **положительный текст**
- Если  $\text{Score} < -0.1$  → **отрицательный текст**
- Иначе → **нейтральный**



[5]  
0

сек.

```
▶ sentiment_dict = {  
    "отличный": 1.0,  
    "ужасный": -1.0,  
    "плохой": -0.8,  
    "хороший": 0.8,  
    "потрясающий": 1.0,  
    "скучный": -0.6  
}  
  
text = "Фильм был отличный, но немного скучный"  
words = text.lower().split()  
  
scores = [sentiment_dict.get(w, 0) for w in words]  
sentiment_score = sum(scores) / len(words)  
  
if sentiment_score > 0.1:  
    sentiment = "позитивный"  
elif sentiment_score < -0.1:  
    sentiment = "негативный"  
else:  
    sentiment = "нейтральный"  
  
print(f"Тональность: {sentiment} (оценка {sentiment_score:.2f})")
```

☞ Тональность: нейтральный (оценка -0.10)

# Bag of words

- **Bag-of-Words (мешок слов)** — это базовый метод преобразования текстов в векторы для машинного обучения.

## Суть метода

Каждый текст (предложение, документ) представляется как **вектор частот слов**, где:

- Каждое уникальное слово из корпуса становится **признаком (feature)**.
- Вектор отражает, **сколько раз** каждое слово встречается в документе.
- То есть мы **игнорируем порядок слов**, оставляя только факты их появления — отсюда название "*мешок слов*".

Document D1	<i>The child makes the dog happy</i> the: 2, dog: 1, makes: 1, child: 1, happy: 1
Document D2	<i>The dog makes the child happy</i> the: 2, child: 1, makes: 1, dog: 1, happy: 1



	child	dog	happy	makes	the	<b>BoW Vector representations</b>
D1	1	1	1	1	2	<b>[1,1,1,1,2]</b>
D2	1	1	1	1	2	<b>[1,1,1,1,2]</b>

	about	bird	heard	is	the	word	you
About the bird, the bird, bird bird bird	1	5	0	0	2	0	0
You heard about the bird	1	1	1	0	1	0	1
The bird is the word	0	1	0	1	2	1	0

# TF-IDF (Term Frequency–Inverse Document Frequency)

- TF-IDF — это способ **оценить важность слова** (*термина*) в документе относительно всего корпуса текстов. Он помогает понять, какие слова действительно **характерны** для документа, а не просто часто встречаются везде (например, “и”, “the”, “это”).

# TF-IDF (Term Frequency–Inverse Document Frequency)

**TF-IDF (Term Frequency–Inverse Document Frequency)** — одного из самых популярных методов для оценки важности слов в документе относительно корпуса.

$$\text{TF-IDF}(t, d) = \text{TF}(t, d) \times \log \frac{N}{\text{DF}(t)}$$

где:

- **t** — термин (слово);
- **d** — документ;
- **N** — общее число документов в корпусе;
- **DF(t)** — число документов, в которых встречается термин *t*;
- **TF(t, d)** — частота термина *t* в документе *d*.

# Интерпретация

- **TF (Term Frequency)** — показывает, насколько часто слово встречается в документе. Например:

$$TF(t, d) = \frac{\text{число вхождений } t \text{ в } d}{\text{общее число слов в } d}$$

- **IDF (Inverse Document Frequency)** — показывает, насколько уникально слово для корпуса:

$$IDF(t) = \log \frac{N}{DF(t)}$$

Если слово встречается во всех документах, то  $DF(t)=N$ , и  $IDF(t)=0$ .

→ Слово **не информативно** (например, “и”, “the”, “это”).

- Если слово встречается только в одном документе, то  $DF(t)=1$ , и  $IDF(t)$  велико.

→ Слово **редкое и важное**.

## Пример. Пусть корпус состоит из 3 документов:

Документ	Текст
$d_1$	«машинное обучение и искусственный интеллект»
$d_2$	«машинное обучение для анализа данных»
$d_3$	«интеллектуальные системы и анализ данных»

### Для слова «машинное»

- $N = 3$
- Встречается в  $d_1$  и  $d_2 \rightarrow DF(\text{машинное}) = 2$
- В  $d_1$ : 4 слова  $\rightarrow TF = 1/4 = 0.25$
- $IDF = \log(3/2) = 0.405$  (натуральный логарифм)

$$TF-IDF = 0.25 \times 0.405 = 0.101$$

### Для слова «интеллект»

- Встречается в  $d_1$  и  $d_3 \rightarrow DF = 2$
- В  $d_1$ :  $TF = 1/4 = 0.25$
- $IDF = 0.405$
- $TF-IDF = 0.101$

### Для слова «данных»

- Встречается в  $d_2$  и  $d_3 \rightarrow DF = 2$
- В  $d_2$ :  $TF = 1/5 = 0.2$
- $IDF = 0.405$
- $TF-IDF = 0.081$

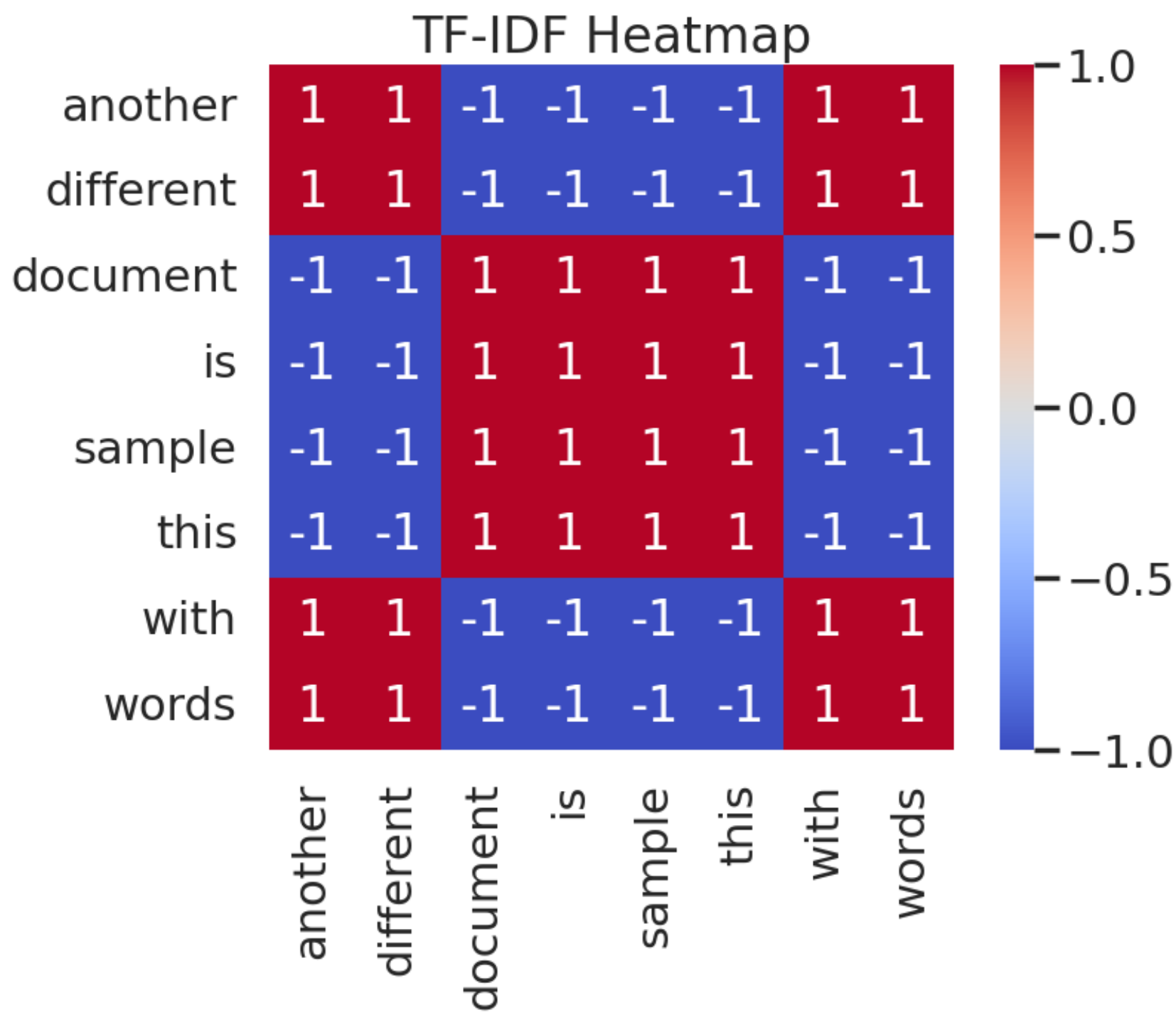
## Интерпретация

- Чем выше **TF-IDF**, тем важнее слово для данного документа.
- Высокий **TF**, но низкий **IDF**  $\rightarrow$  слово частое, но общее (“и”, “для”).
- Низкий **TF**, но высокий **IDF**  $\rightarrow$  слово редкое, но информативное.



Word	TF		IDF	TF*IDF	
	A	B		A	B
The	1/7	1/7	$\log(2/2) = 0$	0	0
Car	1/7	0	$\log(2/1) = 0.3$	0.043	0
Truck	0	1/7	$\log(2/1) = 0.3$	0	0.043
Is	1/7	1/7	$\log(2/2) = 0$	0	0
Driven	1/7	1/7	$\log(2/2) = 0$	0	0
On	1/7	1/7	$\log(2/2) = 0$	0	0
The	1/7	1/7	$\log(2/2) = 0$	0	0
Road	1/7	0	$\log(2/1) = 0.3$	0.043	0
Highway	0	1/7	$\log(2/1) = 0.3$	0	0.043

Index	→	0	1	2	3	4	5	6	7	8
		and	document	first	is	one	second	the	third	this
"This is the first document."		0	0.46979139	0.58028582	0.38408524	0	0	0.38408524	0	0.38408524
"This document is the second document."		0	0.6876236	0	0.28108867	0	0.53864762	0.28108867	0	0.28108867
"And this is the third one."		0.51184851	0	0	0.26710379	0.51184851	0	0.26710379	0.51184851	0.26710379
"Is this the first document?"		0	0.46979139	0.58028582	0.38408524	0	0	0.38408524	0	0.38408524



[4]

0

сек.

```
• # Импортируем класс TF-IDF векторизатора из sklearn
  from sklearn.feature_extraction.text import TfidfVectorizer

  # 📖 Набор документов (корпус)
  docs = [
      "машинное обучение и интеллект",
      "машинное обучение для анализа данных",
      "интеллект и анализ данных"
  ]

  # 💎 Создаём объект векторизатора TF-IDF
  # Он преобразует тексты в числовые векторы, где каждый элемент – вес TF-IDF слова
  vectorizer = TfidfVectorizer()

  # 💎 Обучаем векторизатор на текстах и одновременно преобразуем тексты в матрицу признаков
  # fit_transform() = fit() + transform()
  # Результат – разреженная матрица размера (число документов × число уникальных слов)
  X = vectorizer.fit_transform(docs)

  # 💎 Получаем список всех уникальных слов (фич) корпуса
  print(vectorizer.get_feature_names_out())

  # 💎 Преобразуем разреженную матрицу X в обычный массив для наглядности
  # Каждая строка – документ, каждый столбец – слово из словаря
  # Значения – TF-IDF вес слова в документе
  print(X.toarray())
```

```
➦ ['анализ' 'анализа' 'данных' 'для' 'интеллект' 'машинное' 'обучение']
[[0.          0.          0.          0.          0.57735027 0.57735027
  0.57735027]
 [0.          0.51741994 0.3935112  0.51741994 0.          0.3935112
  0.3935112 ]
 [0.68091856 0.          0.51785612 0.          0.51785612 0.
  0.          ]]
```

[6]  
0

✓  
сек.

```
import pandas as pd

df = pd.DataFrame(X.toarray(), columns=vectorizer.get_feature_names_out())
print(df.round(3))
```

	анализ	анализа	данных	для	интеллект	машинное	обучение
0	0.000	0.000	0.000	0.000	0.577	0.577	0.577
1	0.000	0.517	0.394	0.517	0.000	0.394	0.394
2	0.681	0.000	0.518	0.000	0.518	0.000	0.000

Каждая строка — документ, каждый столбец — слово.

То есть:

строка 0 → первый документ: "машинное обучение и интеллект"

строка 1 → второй документ: "машинное обучение для анализа данных"

строка 2 → третий документ: "интеллект и анализ данных"

# Классификаторы для анализа тональности

- В задаче тональности у нас типичная **многоклассовая классификация (3 класса)**  
→ можно применять стандартные алгоритмы из scikit-learn.

## 1. Логистическая регрессия (Logistic Regression)


```
from sklearn.linear_model import LogisticRegression  
model = LogisticRegression(max_iter=1000)
```

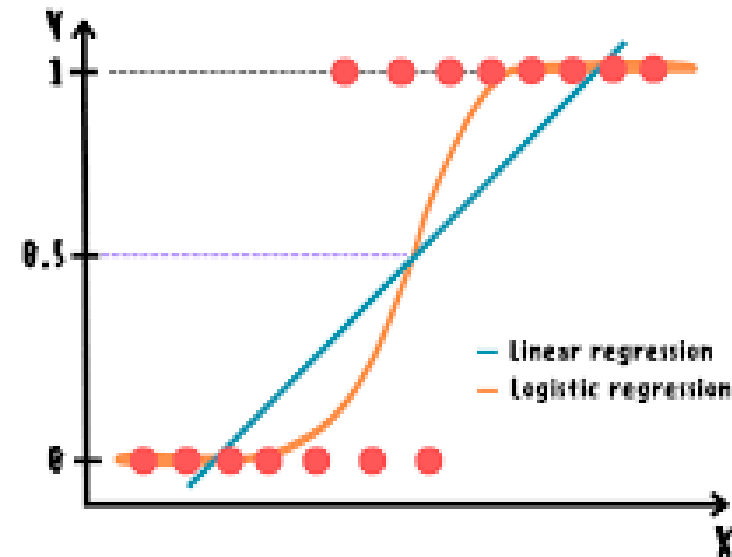
**Описание:** линейный классификатор, который оценивает вероятность классов через логистическую функцию.

### Плюсы:

- Простая и интерпретируемая модель.
- Отлично работает с TF-IDF и BOW признаками.
- Быстро обучается, даёт высокое качество на текстах.

### Минусы:

- Не захватывает нелинейные зависимости.
- Требует нормированных (взвешенных) признаков.
-  **Рекомендуется как базовый вариант для текстовых задач.**



- **2. Наивный Байес (Naive Bayes Classifier)**


```
from sklearn.naive_bayes import MultinomialNB  
model = MultinomialNB()
```

**Описание:** вероятностная модель, основанная на предположении независимости признаков.

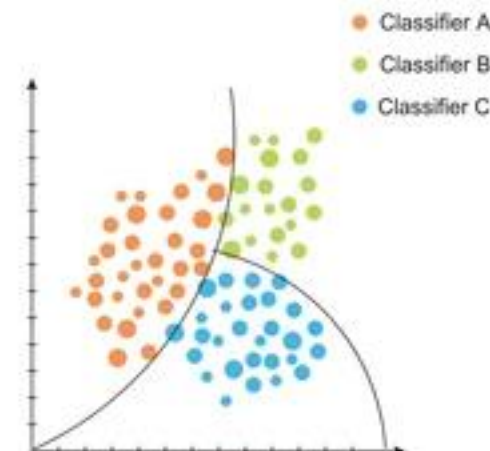
**Плюсы:**

- Очень быстрая.
- Хорошо работает на частотных признаках (TF-IDF, CountVectorizer).
- Часто используется как baseline в NLP.

**Минусы:**

- Игнорирует взаимодействие между словами.
- Может давать чрезмерно уверенные прогнозы.
-  **Отличный выбор для текстов малого объёма и учебных примеров.**

Naive Bayes Classifier



shutterstock.com · 2397899617

## SVM (Support Vector Machine)

```
from sklearn.svm import LinearSVC
```


```
model = LinearSVC()
```

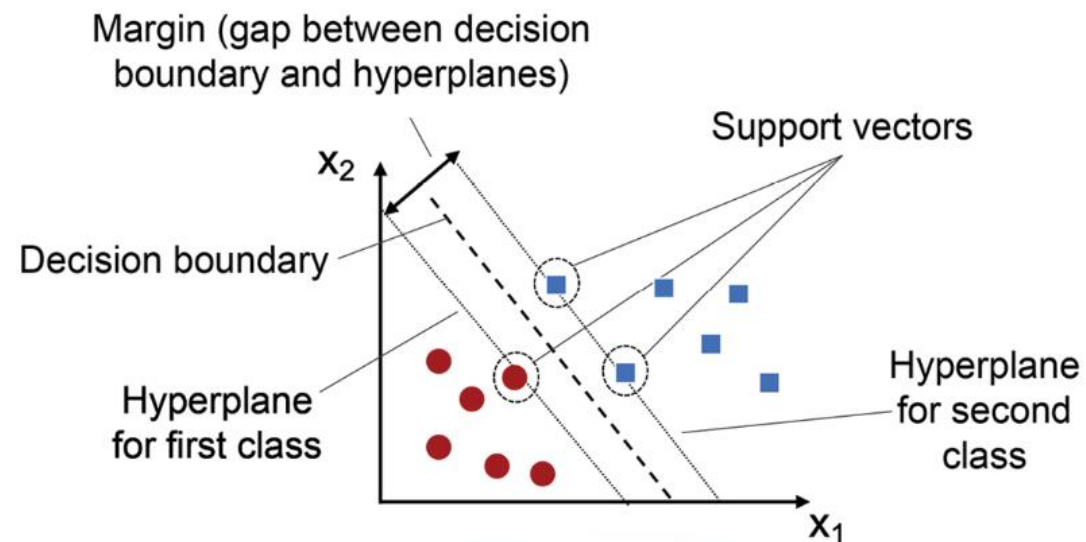
**Описание:** линейный метод, который ищет оптимальную разделяющую гиперплоскость.

### Плюсы:

- Отлично работает на разреженных данных (TF-IDF).
- Часто даёт **лучшие результаты**, чем логистическая регрессия.

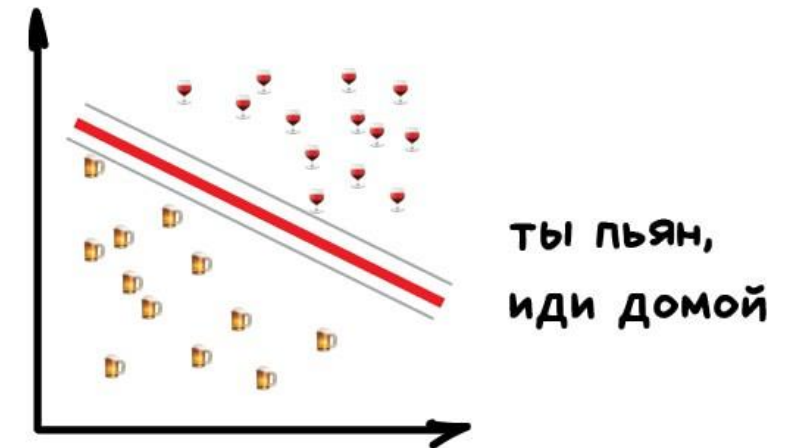
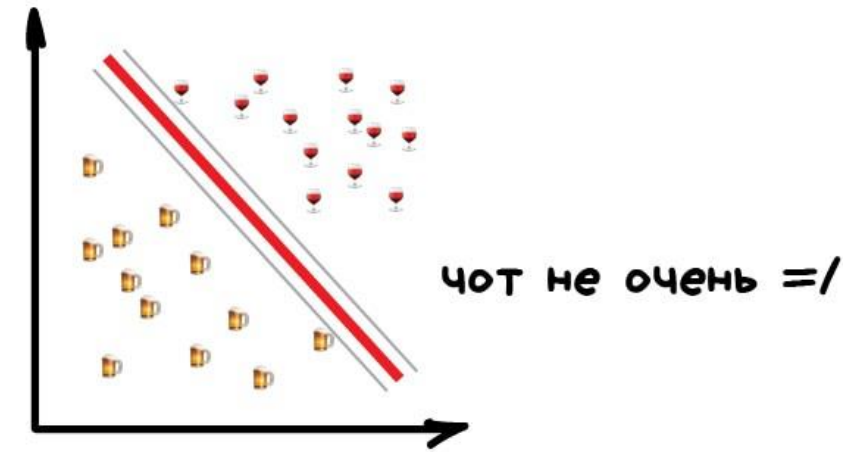
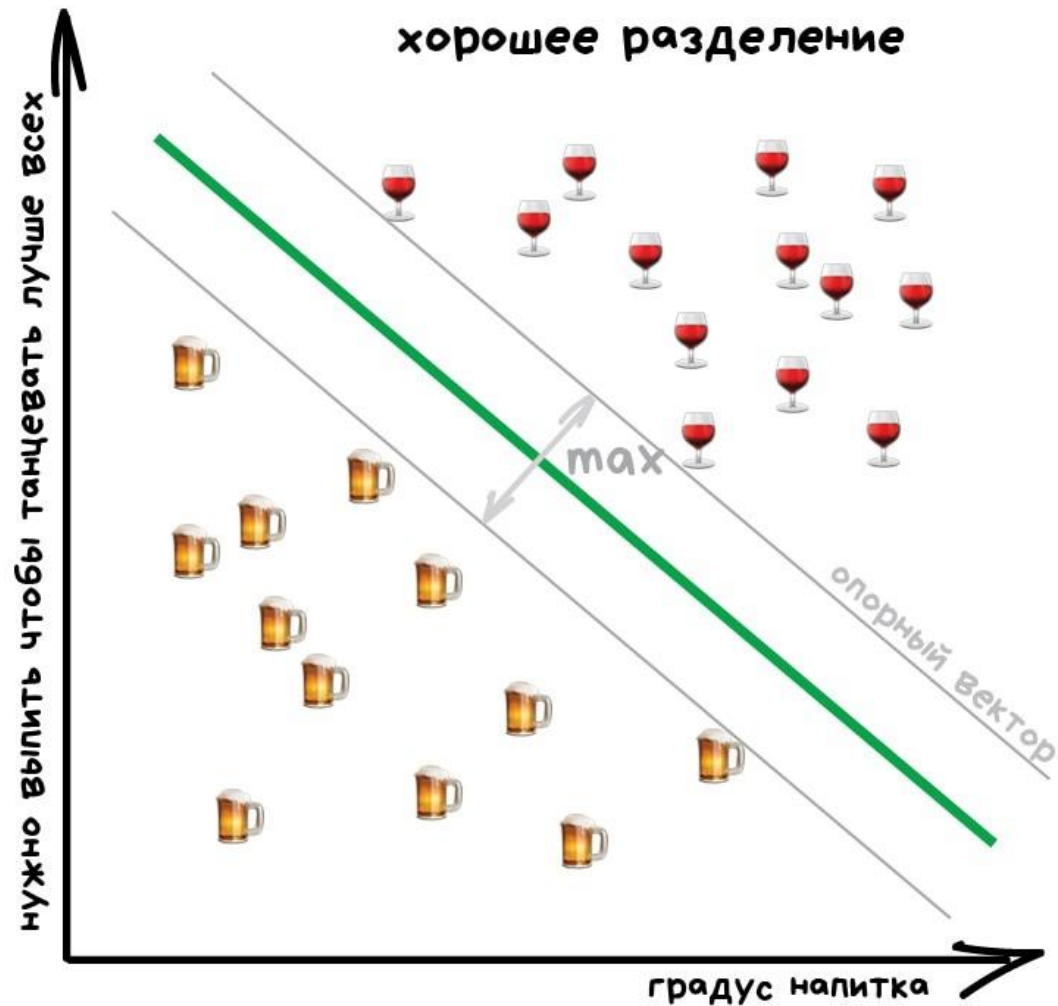
### Минусы:

- Не выдаёт вероятности (только классы).
- Обучение медленнее, чем у LR или NB.
-  Рекомендуется для финального эксперимента, если нужно максимальное качество.





# Разделяем виды алкоголя



Метод Опорных Векторов


# Random Forest

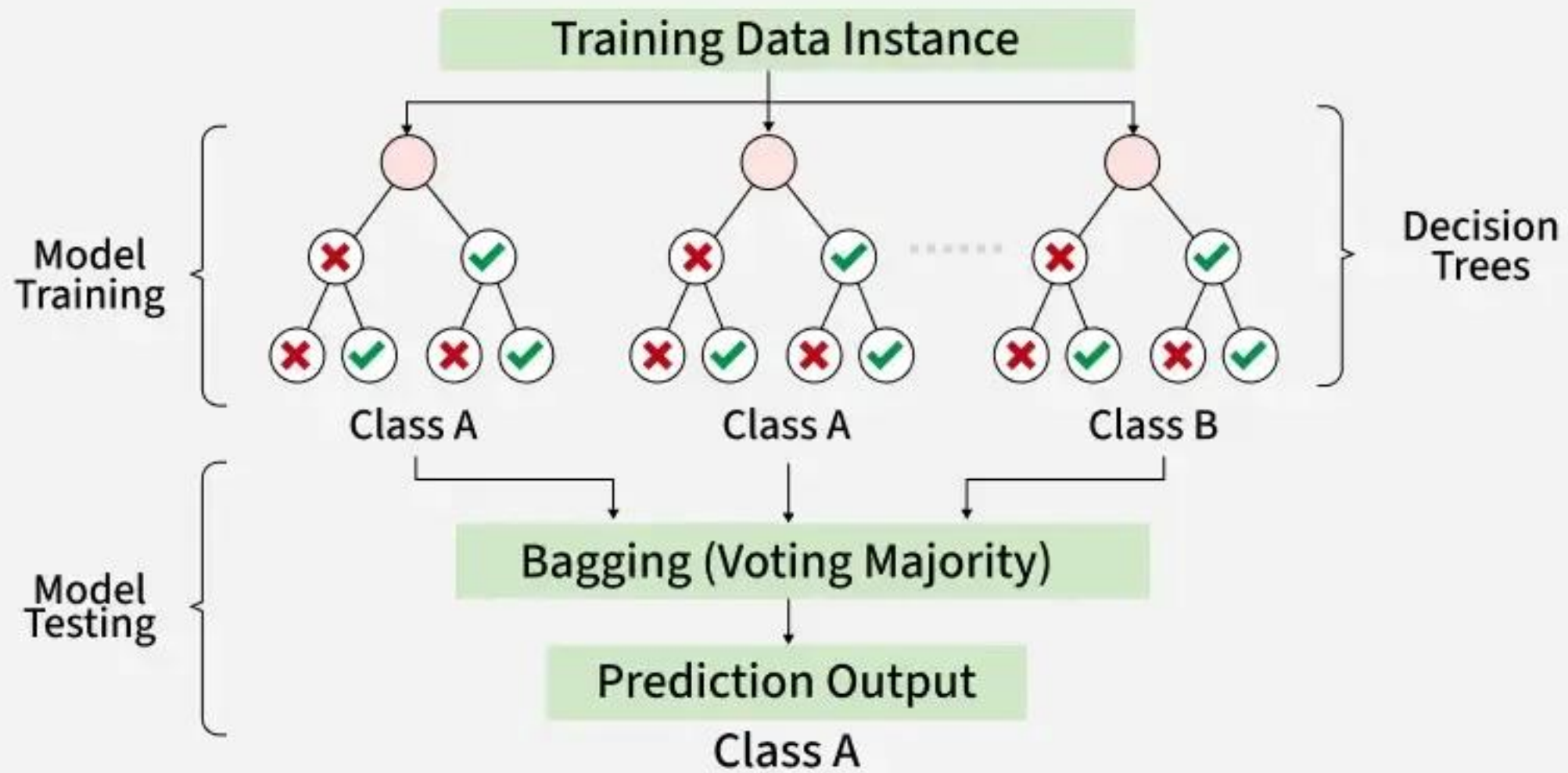
```
from sklearn.ensemble import RandomForestClassifier  
model = RandomForestClassifier()
```

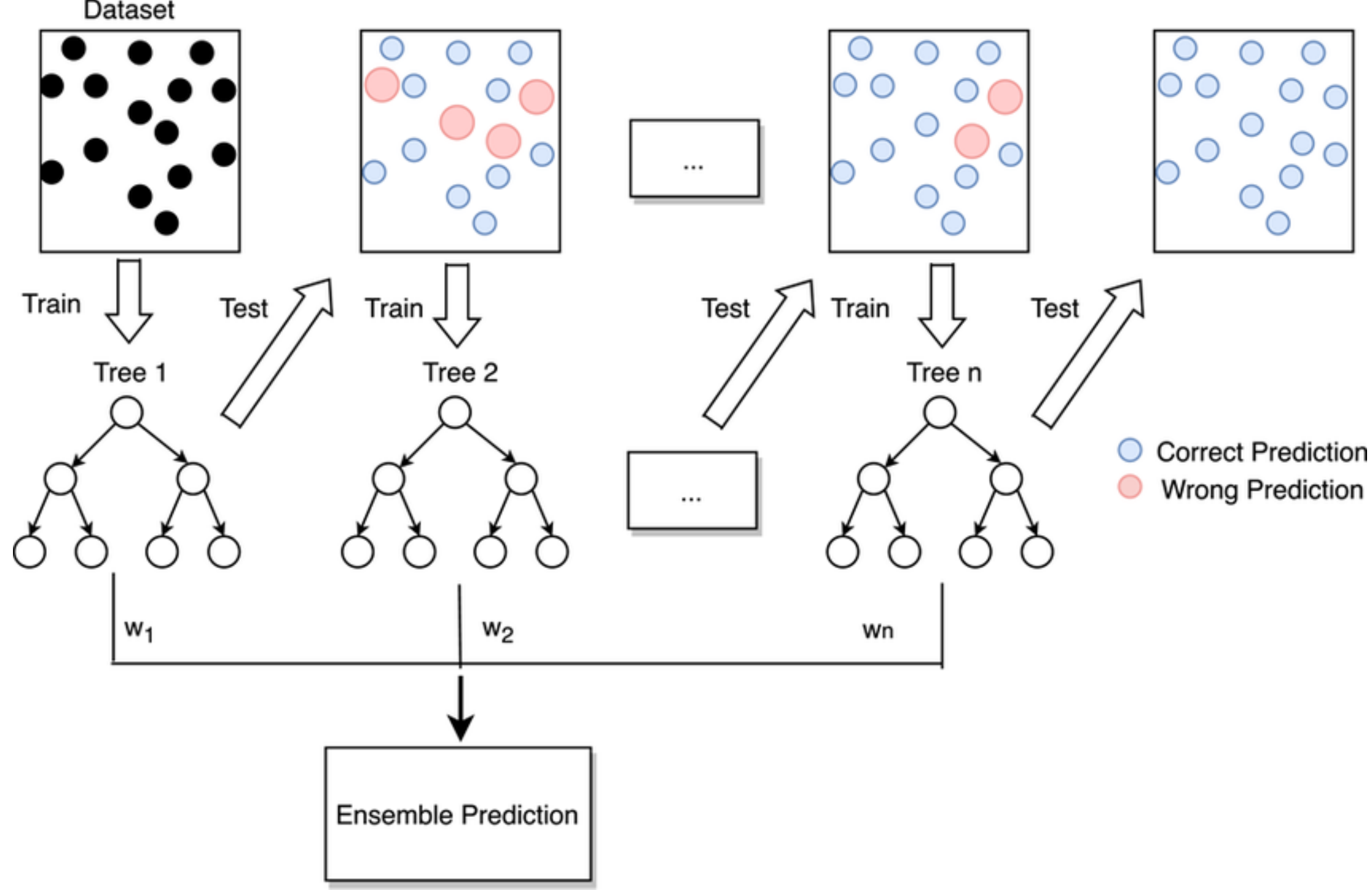
## Плюсы:

- Учитывает нелинейности.
- Можно использовать с TF-IDF или числовыми признаками (например, длина текста, лексиконный балл).

## Минусы:

- Много памяти, неэффективно на больших текстовых матрицах.
- TF-IDF матрицы очень разреженные → деревья неэффективны.
-  Редко используется для “чистого текста”, но полезен в гибридных моделях.





# Давать ли кредит?



Дерево Решений

# Gradient Boosting

Gradient Boosting — тоже ансамбль деревьев, **но они обучаются последовательно, и каждое новое дерево исправляет ошибки предыдущих.**

## ⚙️ Как работает:

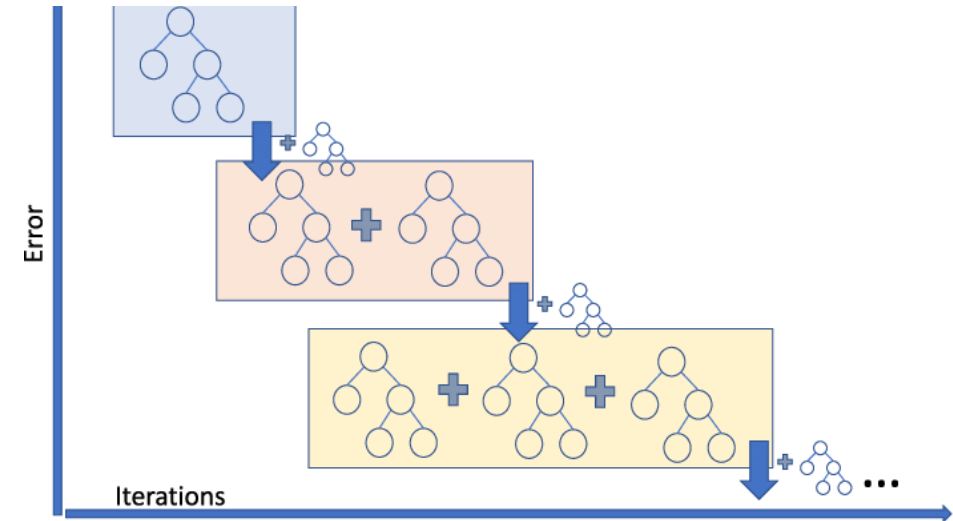
1. Первое дерево строится на исходных данных.
2. Следующее дерево обучается **на ошибках предыдущего**.
3. Вес каждого дерева зависит от того, насколько оно улучшает результат.
4. Итоговый прогноз — сумма (или взвешенная сумма) всех деревьев.

## 📈 Преимущества:

- **Очень высокая точность;**
- Может моделировать сложные, **нелинейные зависимости;**
- Даёт оценку важности признаков;
- Используется во всех **соревнованиях Kaggle** и продакшн-системах.

## ⚠️ Недостатки:

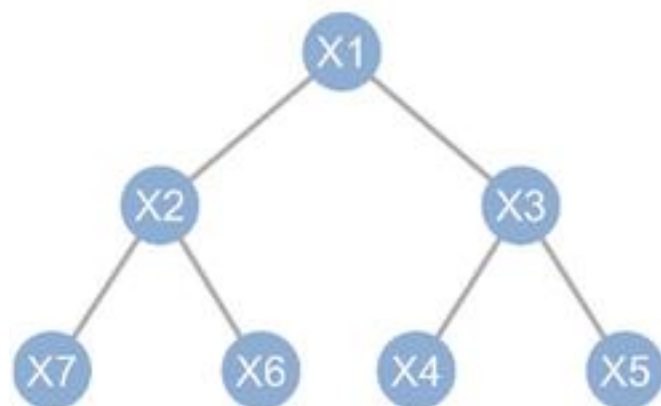
- Требуется **тонкая настройка гиперпараметров** (learning rate, depth, n\_estimators);
- **Медленнее обучается**, чем Random Forest;
- Более подвержен переобучению без регуляризации.



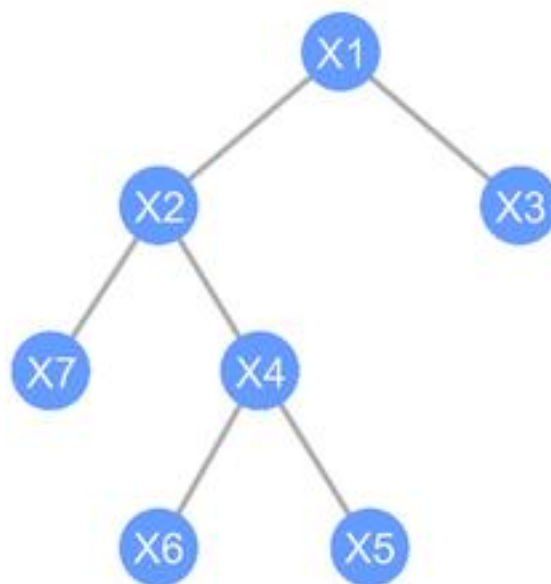
# XGBoost / LightGBM / CatBoost

Характеристика	XGBoost	LightGBM	CatBoost
Разработчик	Университет Вашингтона	Microsoft	Yandex
Скорость	★ ★ ★	★ ★ ★ ★	★ ★
Точность	★ ★ ★ ★	★ ★ ★ ★	★ ★ ★ ★ ★
Работа с категориальными	Нет	Частично	✓ Отлично
GPU-поддержка	✓	✓	✓
Простота настройки	Средняя	Средняя	Легче
Устойчивость к переобучению	Средняя	Средняя	Высокая
Рекомендуется для	Табличных, числовых данных	Больших корпусов	Смешанных, категориальных

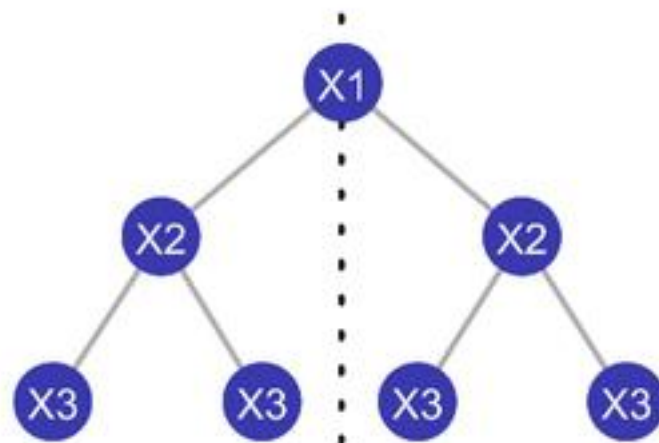
XGBoost



LightGBM



CatBoost





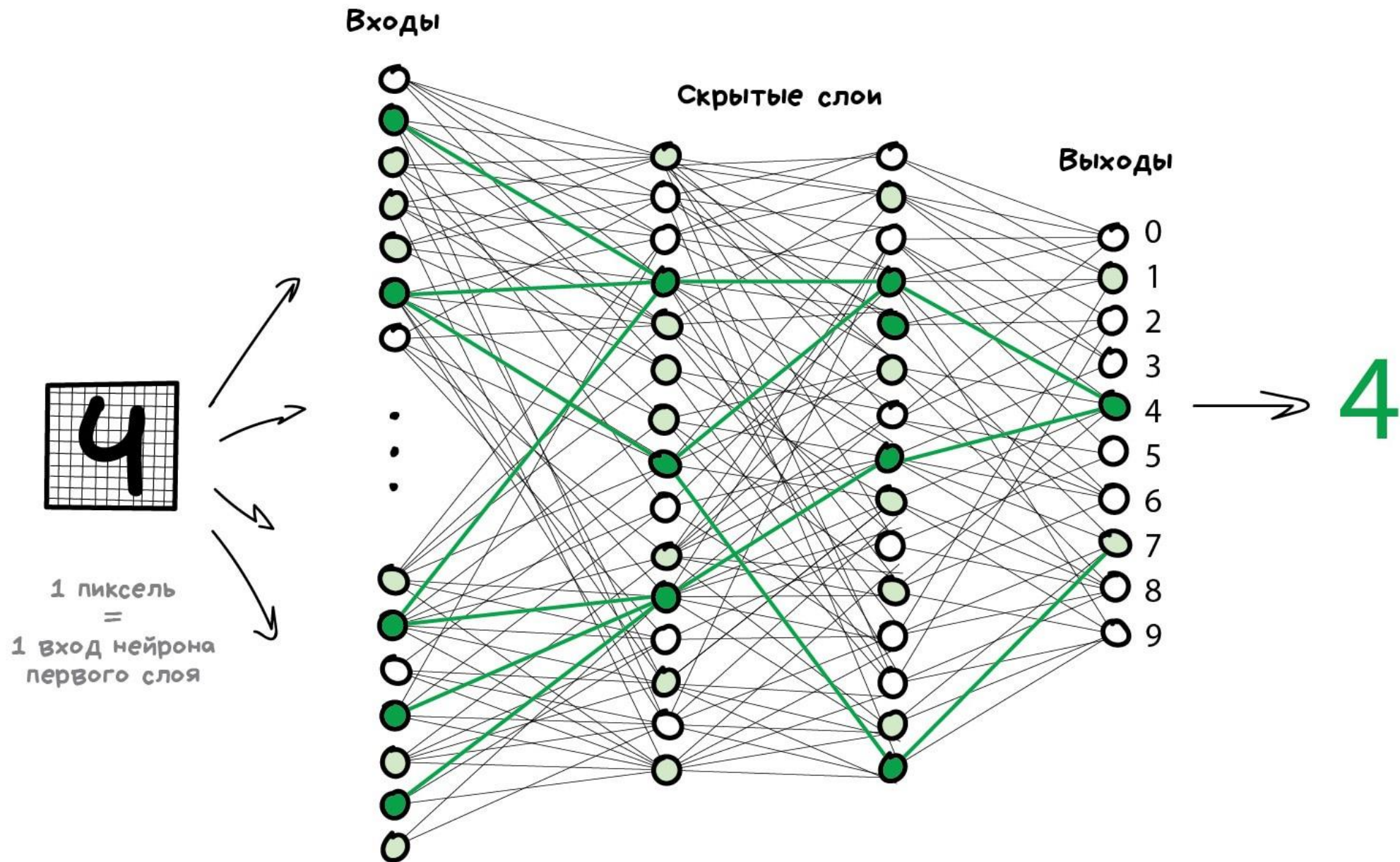
# Нейронные сети

## ➤ MLP (многослойный перцептрон)

- **Многослойный перцептрон (MLP)** — это классическая **нейронная сеть прямого распространения (feed-forward neural network)**. Она состоит из **входного слоя**, **одного или нескольких скрытых слоёв** и **выходного слоя**.

### Структура MLP

- 1. Входной слой** — принимает числовые признаки (например, TF-IDF-вектор текста).
- 2. Скрытые слои** — нейроны, которые обучаются выявлять сложные связи между признаками.
- 3. Выходной слой** — выдаёт вероятности классов (негатив, нейтраль, позитив).







Многослойный Перцептрон (MLP)

# Почему MLP работает

- Каждая нейронная связь **взвешена (weights)** → сеть учится выделять важные признаки.
- Нелинейные функции (ReLU, sigmoid, tanh) позволяют моделировать **сложные зависимости**, которые линейные модели (Logistic Regression) не могут уловить.
- Хорошо подходит для **плотных векторов** (например, после SVD, Word2Vec, BERT).

## Преимущества MLP

-  Улавливает **нелинейные зависимости** между признаками.
-  Гибко настраивается (количество слоёв, нейронов, функций).
-  Может работать с **плотными эмбедингами (Word2Vec, BERT)**.
-  Универсальный — подходит для любых типов данных.

## Недостатки

-  Требуется больше данных, чем логистическая регрессия.
-  Медленнее обучается, особенно на разреженных TF-IDF.
-  Чувствителен к гиперпараметрам (глубина, скорость обучения).

Модель	Тип	Скорость	Качество	Особенности
Logistic Regression	Линейная	★ ★ ★ ★	★ ★ ★ ★	Отлично для TF-IDF
Naive Bayes	Вероятностная	★ ★ ★ ★ ★	★ ★ ★	Простой baseline
SVM (LinearSVC)	Линейная	★ ★ ★	★ ★ ★ ★ ★	Хорошо работает на текстах
Random Forest	Деревья	★ ★	★ ★	Плохо на разреженных данных
XGBoost / LightGBM	Бустинг	★ ★	★ ★ ★ ★	Хорошо с гибридными фичами
MLP (нейросеть)	Нелинейная	★	★ ★ ★ ★	Для продвинутых задач
BERT / LLM	Трансформер	⚡	★ ★ ★ ★ ★	Требует GPU и большого корпуса