1. Что такое эмбеддинг? Проведите меня через логику, стоящую за концепцией эмбеддинга.

Эмбеддинг – числовой вектор, созданный для преобразования данных (таких как слова, предложения или изображения) в формат, понятный компьютеру. Представьте, что вы хотите научить модель определять тональность текста. Вы подаете ей на вход предложение: "Этот фильм — великолепное кино".

Человек видит слова и понимает их смысл.

Компьютер видит лишь последовательность символов. Для него "кино", "фильм", "великолепный" — просто наборы байтов. Он не знает, что "кино" и "фильм" — это почти одно и то же, а "великолепный" — это что-то хорошее.

One-Hot Encoding (Кодирование "один-из-многих"):

- Каждому слову в словаре присваивается уникальный индекс. Размер вектора равен размеру словаря (например, 50 000 чисел).
- Все векторы одинаково далеки друг от друга. Расстояние между "кино" и "фильм" такое же, как между "кино" и "вертолет". Модель не может уловить смысловую близость.

А что если мы представим слово не в виде одной "горящей" единицы среди тысяч нулей, а в виде плотного вектора из, скажем, 300 чисел, где каждое число кодирует какой-то скрытый признак?

- Вектор для "король": `[0.8, 0.4, 0.1, -0.9, ...]`
- Вектор для "королева": `[0.7, 0.5, 0.9, -0.8, ...]`
- Вектор для "мужчина": `[0.9, 0.3, 0.1, -0.2, ...]`
- Вектор для "женщина": `[0.8, 0.4, 0.9, -0.1, ...]`

"Слова, которые встречаются в одинаковых контекстах, имеют схожие значения".

1. Мы берем огромный текст (например, всю Википедию).
2. Берем целевое слово (например, "банк").
3. Смотрим на слова вокруг него ("река", "деньги", "счет").
4. Настраиваем нейронную сеть так, чтобы она могла предсказать эти соседние слова по целевому слову (или наоборот). Популярные алгоритмы: Word2Vec, GloVe.
5. В процессе этой настройки веса внутреннего слоя нейронной сети и становятся теми самыми векторами-эмбеддингами

2. Счетные способы построения эмбеддинга: bag-of-words, tf-idf, BM25.

Ключевая идея этих методов: не учить векторы, а вычислять их на основе статистики встречаемости слов в документах.

bag-of-words:

Создаём словарь из всех уникальных слов во всех документах и для каждого документа будет эмбеддинг равный размеру словарю (количество вхождений уникального слова в этом документе)

У нас есть три документа:

1. `"кот сидит на ковре"`
2. `"кот смотрит на котенка"`
3. `"котенок пьет молоко"`

Словарь: `[кот, сидит, на, ковре, смотрит, котенок, пьет, молоко]` (8 слов)

Вектора BoW:

- Док 1: `[1, 1, 1, 1, 0, 0, 0, 0]` (слово "кот" встретилось 1 раз, "сидит" - 1, и т.д.)
- Док 2: `[1, 0, 1, 0, 1, 1, 0, 0]`
- Док 3: `[0, 0, 0, 0, 0, 1, 1, 1]`

TF-IDF (Term Frequency-Inverse Document Frequency):

Это большое улучшение над BoW. TF-IDF не просто считает слова, а взвешивает их, оценивая важность слова для конкретного документа в контексте всей коллекции.

Вектор для документа строится так же, как в BoW, но вместо простого счетчика используется произведение двух величин:

TF (Term Frequency) — Частота слова: Сколько раз слово встретилось в документе. Показывает "локальную" важность.

TF(t, d) = (число раз, когда термин t встретился в док. d) / (общее число слов в док. d)

IDF (Inverse Document Frequency) — Обратная частота документа: Насколько слово редкое или уникальное в масштабах всей коллекции. Показывает "глобальную" важность.

IDF(t, D) = log((общее число док. в коллекции D) / (число док., в которых встречается термин t))

Итоговая формула: TF-IDF(t, d, D) = TF(t, d) * IDF(t, D)

Пример (продолжаем с нашими документами):

- Слово **"кот"** встречается в 2-х документах из 3-х. Его `IDF = log(3/2) ≈ 0.176`.
- Слово **"молоко"** встречается только в 1-м документе. Его `IDF = log(3/1) ≈ 0.477`.

Для Документа 3 ("котенок пьет молоко"):

- Вес для "котенок" будет `TF * IDF = (1/3) * 0.176 ≈ 0.059`
- Вес для "молоко" будет `(1/3) * 0.477 ≈ 0.159`

Вектор для Док 3 будет: `[0, 0, 0, 0, 0, 0.059, 0.33, 0.159]`

Все еще страдает от разреженности и потери порядка слов.

BM25:

```
BM25(t, d) = IDF(t) * [ (TF(t, d) * (k + 1)) / (TF(t, d) + k * (1 - b + b * (|d| / avgdl))) ]
```

Где:

- `TF(t, d)` и `IDF(t)` — те же понятия, но IDF часто вычисляется по-другому.
- `k` — параметр, контролирующий насыщение частоты. При `k=0` TF вообще не учитывается. При больших `k` BM25 ведет себя как TF-IDF.
- `b` — параметр, контролирующий нормализацию по длине (`0` - нет нормализации, `1` - полная нормализация).
- `|d|` — длина текущего документа.
- `avgdl` — средняя длина документа в коллекции.

Смысл: Дробь справа — это "усовершенствованный TF", который ограничивает влияние очень высоких частот и наказывает/поощряет документы в зависимости от их длины.

3. **Сравните счетные и обучаемые методы построения эмбеддингов: в чем их различия, преимущества и недостатки?**

Счётные (count-based) методы

Строятся по формулам, основываются на статистике частотности слов

✓ Преимущества:

- **Простота** реализации и интерпретации
- **Быстрота вычислений**
- Не требуют больших данных для работы
- **Прозрачность**: легко понять, почему документ получил определённый скор

✗ Недостатки:

- **Игнорируют контекст** и порядок слов
- Размер векторов растёт с размером словаря (высокая размерность, разреженность)

- Не понимают **синонимов**, **полисемии** («банк» как здание или организация)
- Не переносят знания на новые слова (Out-of-Vocabulary)

Обучаемые (learned / distributed) методы

Обучаются **на данных**

✅ Преимущества

- Улавливают **семантику** (смысл слов, контекст)
- **Компактные векторы** (обычно 100–1024 измерения)
- Понимают **синонимы, морфологию, контекстные различия**
- Могут **генерировать эмбеддинги для новых слов** (в случае FastText)
- Современные модели (BERT и далее) учитывают **контекст появления слова**

❌ Недостатки:

- Требуют **обучения** (данные, вычислительные ресурсы)
- Менее интерпретируемы (в отличие от TF-IDF)
- Более **ресурсоёмкие** при применении
- Сложнее адаптировать под специфические задачи без fine-tuning

4. **Построение эмбеддинга с помощью глубинного обучения: Word2vec, GloVe, fastText.**

1. Word2Vec
Word2Vec ключевая идея — **предсказывать слова по их контексту или контекст по слову**.
Логика:
Вместо подсчета частот, мы заставляем нейронную сеть **играть в угадывание**, и в процессе этой игры она выучивает качественные векторы.
Архитектуры:
а) Continuous Bag of Words (CBOW) — "Мешок слов"
- **Что делает:** Предсказывает **целевое слово** по окружающим его **контекстным словам**.
- **Аналогия:** Дайте мне 4 слова из предложения, а я угадаю, какое слово было в центре.
- **Вход:** Несколько контекстных слов (например, "кошка", "улыбкой", "солнечный").
- **Выход (цель для предсказания):** Одно центральное слово ("умывается").
- **Плюсы:** Быстрее обучается, лучше работает с частыми словами.

б) Skip-gram — "Скип-грамма"
- **Что делает:** Предсказывает **контекстные слова** по **целевому слову**.
- **Аналогия:** Дайте мне одно слово, а я угадаю, какие слова обычно его окружают.
- **Вход:** Одно центральное слово ("умывается").
- **Выход (цель для предсказания):** Несколько контекстных слов ("кошка", "улыбкой", "солнечный").
- **Плюсы:** Лучше справляется с редкими словами, часто дает чуть более качественные эмбеддинги.

Как получаются эмбеддинги?
Сеть имеет один скрытый слой без функции активации. После обучения **матрица весов между входным и скрытым слоем** и становится таблицей эмбеддингов слов. Каждому слову соответствует одна строка (или столбец) этой матрицы.
Итог Word2Vec: Мы получаем статические, но семантически насыщенные векторы, которые демонстрируют волшебные свойства арифметики ("король" - "мужчина" + "женщина" ≈ "королева").
2. GloVe (Global Vectors for Word Representation)
GloVe — это гибридный подход, который объединяет **достоинства счетных методов и обучаемых подходов**.
Как строится:
1. Строится **глобальная матрица ко-встречаемости** X. Элемент X_{ij} показывает, сколько раз слово j встречалось в контексте слова i во всем корпусе. Это *счетный* этап.
2. Обучается модель, которая предсказывает значения этой матрицы. Цель функции потерь — научиться так представлять слова в виде векторов, чтобы **скалярное произведение их векторов было пропорционально логарифму их частоты ко-встречаемости**.

Формула цели: w_i * w_j + b_i + b_j ≈ log(X_{ij})
где w_i и w_j — векторы слов, а b_i и b_j — смещения (biases).
Итог GloVe: По сути, GloVe выполняет **разложение матрицы ко-встречаемости** на плотные векторы.
3. fastText

FastText — это прямое развитие идей Word2Vec, которое решает его главную проблему: **обработку редких и неизвестных слов (Out-of-Vocabulary, OOV)**.

Логика:

Вместо того чтобы создавать один вектор для целого слова, fastText представляет слово как **сумму векторов его символов n-grams.**

Как строится:

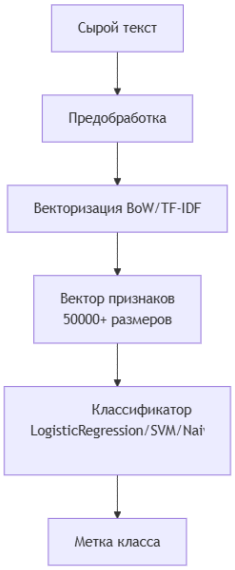
1. Слово разбивается на набор символов n-grams.
    - Например, для слова "кит" и n=3: <ки, кит, ит>
2. Каждому n-gramy присваивается свой вектор.
3. Вектор всего слова — это просто **сумма векторов всех его n-grams**.

Для обучения используется та же архитектура Skip-gram или CBOW, но на вход подаются не индексы целых слов, а индексы их n-grams.

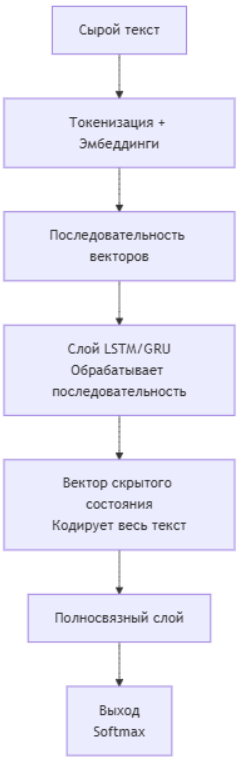
Ключевое преимущество:

- **Работа с неизвестными словами:** Если модель встретила слово "киток", которого не было в обучающей выборке, она все равно может составить его вектор из известных n-grams: <ки, кит, ито, ток, ок>.
- **Лучшие векторы для редких слов:** Редкое слово "глубоководный" может иметь хороший вектор, потому что его части ("глубок", "водн") встречаются в других словах и хорошо обучены.
- **Работа с опечатками и морфологией:** Особенно полезно для языков с богатой морфологией (русский, немецкий, финский), где одно слово имеет много форм.

5. **Какие подходы к задаче классификации текстов вам известны? Проведите меня через архитектуру модели.**



Шаг 1: Предобработка
Токенизация (разбиение на слова)
Приведение к нижнему регистру
Удаление стоп-слов, знаков препинания
Шаг 2: Векторизация
Создается словарь уникальных слов по всему корпусу.
Каждый документ преобразуется в вектор большого размера (размер словаря) с использованием (BoW) или TfidfVectorizer.
Шаг 3: Классификация
Полученный разреженный вектор подается на вход классическому алгоритму машинного обучения.



Слой эмбеддингов: Превращает текст в матрицу [длина_последовательности x размер_эмбеддинга].
RNN-слой (чаще LSTM или GRU):

Ячейки LSTM обрабатывают векторы слов по очереди.

На каждом шаге они обновляют свое скрытое состояние, которое содержит информацию о всей предыдущей последовательности.

Вектор представления: Закодированное представление всего текста — это обычно последнее скрытое состояние RNN.

Финальные слои: Это представление подается на полносвязный слой и классификатор.

Плюсы: Отлично справляется с длинными зависимостями, учитывает весь контекст предложения.
Минусы: Медленнее обучения, чем у CNN, может забывать важную информацию из начала длинного текста.

Выделить фичи/получить эмбеддинги
Умножить фичи на веса
Получить вероятности классов

6. **Расскажите об архитектуре Encoder-Decoder и моделях, используемых в ней (e.g. RNN, LSTM etc.)**

Основная идея: Преобразовать входную последовательность (например, предложение на английском) в выходную последовательность (например, предложение на французском). Ключевая особенность — длины этих последовательностей могут быть разными и не известны заранее.

Архитектура состоит из двух основных компонентов:

Энкодер (Кодировщик):

Задача: "Прочитать" и "понять" входную последовательность.

Принцип работы: Он обрабатывает каждый элемент входной последовательности (например, каждое слово) шаг за шагом и преобразует всю последовательность в единое, фиксированное по размеру внутреннее представление, которое часто называют вектором контекста (context vector) или состоянием (state). Этот вектор старается сжать всю смысловую информацию из входных данных.

Декодер (Декодировщик):

Задача: На основе вектора контекста от энкодера "сгенерировать" выходную последовательность элемент за элементом.

Принцип работы: Декодер начинает генерацию (часто с специального токена начала последовательности, <start>). На каждом шаге он использует свое текущее состояние и сгенерированный на предыдущем шаге токен, чтобы предсказать следующий токен. Процесс продолжается до тех пор, пока не будет сгенерирован токен конца последовательности (<end>).
1. Рекуррентные нейронные сети (RNN)
RNN по своей природе обрабатывают данные последовательно, сохраняя "память" о предыдущих элементах через свое скрытое состояние (hidden state). Это скрытое состояние передается от одного шага к другому.
В Encoder-Decoder:
Энкодер (RNN): Последовательно обрабатывает слова входного предложения. Скрытое состояние после обработки последнего слова становится тем самым вектором контекста.
Декодер (RNN): Получает этот вектор контекста в качестве своего начального скрытого состояния и начинает генерировать выходную последовательность.
Проблема: У простых RNN есть проблема исчезающего градиента (vanishing gradient), из-за которой они плохо запоминают долгосрочные зависимости в длинных последовательностях.

В "наивной" архитектуре Encoder-Decoder весь смысл входной последовательности должен быть упакован в один фиксированный вектор контекста. Для длинных предложений это становится "бутылочным горлышком" — декодеру не хватает информации из начала предложения, когда он доходит до его конца.

7. Как вы понимаете механизм внимания? Какие примеры вы можете привести?

Внимание — это механизм, который позволяет нейронной сети "фокусироваться" на наиболее релевантных частях входной информации при выполнении конкретной задачи.

Представьте, что вы читаете сложный текст. Вы не впитываете все слова одинаково — вы выделяете ключевые слова и фразы, чтобы понять суть. Ваш мозг автоматически придает им больший "вес". Механизм внимания в точности имитирует этот процесс.

Ключевая метафора: Это "прожектор", который нейронная сеть может направлять на разные части своих входных данных. Яркость этого прожектора (его "вес") определяется тем, насколько важна данная информация в текущий момент.

Запрос (Query), Ключи (Keys), Значения (Values):

Запрос (Q): Вектор, который представляет "интерес" или "потребность" в текущий момент. (Например, "Что мне нужно знать сейчас, чтобы перевести следующее слово?").

Ключи (K): Векторы, которые представляют "идентификаторы" или "тематику" каждого элемента в источнике. (Например, "Я — слово 'кошка', я — слово 'сидела'").

Значения (V): Векторы, которые несут фактическую информацию из источника. (Часто это те же векторы, что и на выходе энкодера).

Процесс вычисления внимания:

Шаг 1: Скалярное произведение. Мы сравниваем Запрос (Q) с каждым Ключом (K). Чем больше их скалярное произведение, тем более релевантен соответствующий источник для текущего Запроса.

Шаг 2: Взвешивание. Результаты сравнения (scores) пропускаются через функцию Softmax, чтобы получить веса внимания — числа от 0 до 1, сумма которых равна 1. Эти веса и есть "яркость прожектора" для каждого элемента.

Шаг 3: Суммирование. Мы вычисляем взвешенную сумму всех Значений (V), используя полученные веса внимания. Элементы с высоким весом внесут больший вклад в итог.

Результат: Получается контекстный вектор, который является не просто усреднением всей информации, а целенаправленной выжимкой того, что важно именно сейчас.
Примеры, которые иллюстрируют механизм внимания
Пример 1: Машинный перевод
Задача: Перевести с английского на русский: "The cat sat on the mat" -> "Кошка сидела на коврике".

Энкодер преобразует каждое английское слово в вектор.

Декодер начинает генерировать русское предложение.

Когда декодер генерирует слово "Кошка", его Запрос (Q) очень похож на Ключ (K) слова "The cat". Механизм внимания присвоит высокий вес Значению (V) слова "cat". Декодер "посмотрел" на слово "cat".

Когда декодер генерирует слово "сидела", его Запрос будет сильнее всего соответствовать Ключу слова "sat". Прожектор внимания переместится на это слово.

Когда декодер генерирует слово "коврике", он сфокусируется на слове "mat".

Визуализация: Если бы мы построили "карту внимания" (attention map), это была бы матрица, где по горизонтали — английские слова, по вертикали — русские, а яркость ячейки показывала бы, насколько сильно русское слово "внимало" английскому при своей генерации. Мы бы увидели яркую диагональ, но не идеальную, так как порядок слов может меняться.

Пример 2: Суммаризация текста
Задача: Написать заголовок для статьи: "Компания X сегодня объявила о рекордной прибыли за четвертый квартал, которая составила 1 млрд долларов".

Механизм внимания в модели будет оценивать все слова исходного текста.

При генерации заголовка, например, "Компания X: рекордная прибыль 1 млрд долларов", модель будет присваивать высокие веса внимания словам "Компания X", "рекордной", "прибыли", "1 млрд".

Словам вроде "сегодня", "объявила", "за четвертый квартал" будут присвоены низкие веса, так как они менее важны для краткого заголовка.

8. **Проведите меня через архитектуру трансформера.**

Основная идея Трансформера — полностью отказаться от рекуррентных сетей (RNN/LSTM) и использовать исключительно механизмы внимания, что позволяет обрабатывать последовательности параллельно, а не последовательно.
В отличие от RNN, Трансформер обрабатывает все слова одновременно. У него нет врожденного понятия "порядка слов".

1) **Вход в энкодер** - векторы, несущие информацию и о слове, и о его позиции
2) **Блок энкодера** (Повторяется N раз, например, 6 раз)
Каждый блок энкодера состоит из двух подслоев:

Подслой 1: Self-Attention
Что делает: Позволяет каждому слову в последовательности "взаимодействовать" со всеми другими словами, включая себя самого.
Как: Используется Многоголовое внимание (Multi-Head Attention). Несколько механизмов внимания ("головы") работают параллельно, каждая изучая разные типы зависимостей (например, одна — синтаксические связи, другая — смысловые).
Результат: Для каждого слова мы получаем новый вектор, который является "взвешенной суммой" векторов всех слов в предложении. Этот вектор теперь содержит контекст всего предложения.

Подслой 2: Feed-Forward Network
Что делает: Это обычная полносвязная нейронная сеть, которая применяется независимо и идентично к каждому позиционно-кодированному вектору, выходящему из слоя внимания.
Зачем: Она выполняет нелинейное преобразование и further обрабатывает информацию, полученную от внимания.
Важно: Она работает независимо для каждого слова, в отличие от слоя внимания, который смешивал информацию между словами.

После каждого из двух подслоев (Внимания и FFN) происходит следующее:

Исходный вход к подслою (например, x) складывается с выходом этого подслоя (например, SelfAttention(x)). Это помогает градиенту лучше распространяться при обучении и предотвращает исчезание градиента.
Слой Нормализации (LayerNorm): Результат сложения нормализуется.
Формула: Attn(Q,K,V)=softmax(QK^T/√d_k)V; multi-head = параллельные «ракурсы».
3) Декодер генерирует выходную последовательность ("Я студент") по одному токену за раз. Он также состоит из стопки идентичных блоков. На вход декодеру подается сдвинутая на один шаг вправо выходная последовательность с специальным токеном «eos».Так же, как и на входе, применяется позиционное кодирование.

Блок декодера (Повторяется N раз)
Каждый блок декодера содержит ТРИ подслоя:
Подслой 1: Masked Self-Attention
Что делает: Позволяет каждому "сгенерированному" слову взаимодействовать только с предыдущими сгенерированными словами. Как и в энкодере, здесь также есть остаточное соединение и нормализация.

Подслой 2: Encoder-Decoder Attention
Это сердце связи между энкодером и декодером.
Как работает:
Query (Запрос) поступает от выхода предыдущего подслоя декодера. Это "вопрос" декодера: "Что мне нужно из входного предложения, чтобы сгенерировать следующее слово?".
Keys и Values поступают от финального выхода энкодера. Это "база знаний" о входном предложении.
Результат: На этом шаге декодер "расставляет фокус" по входному предложению. Например, генерируя слово "студент", декодер будет сильнее всего "внимать" слову "student" из входной последовательности.
Снова следует остаточное соединение и нормализация.

Подслой 3: Опять Feed-Forward Network

4) Финальный слой и вывод
Векторы, выходящие из последнего блока декодера, пропускаются через линейный слой который проецирует их в пространство размером с размер словаря выходного языка.
Затем применяется функция Softmax, которая преобразует логиты в вероятности для каждого слова в словаре.
Модель выбирает слово с наибольшей вероятностью как следующее слово в последовательности.
Это слово подается обратно на вход декодера на следующем шаге, и процесс повторяется до генерации токена <end>.

9. Расскажите про стратегии генерации текста: Beam search, генерация с температурой, top-P sampling etc.

Beam search:
Как работает:
На каждом шаге мы поддерживаем k кандидатов (лучей).
Для каждого кандидата мы рассматриваем N наиболее вероятных следующих слов.
Это дает нам k * N возможных последовательностей.
Мы выбираем k последовательностей с наивысшим общим счетом (сумма логарифмов вероятностей) и отбрасываем остальные.
Процесс повторяется, пока каждая последовательность в луче не закончится (достигнет токена <end>).
Склонен к повторениям.
Temperature Sampling:
Это стратегия, которая добавляет случайность (randomness) в процесс выбора, делая текст более разнообразным и креативным.
Как работает:
Мы меняем распределение вероятностей модели перед сэмплированием с помощью параметра температура (T).
Формула: P_modified(word) = softmax(logits(word) / T)
Эффект температуры:
T = 1: Стандартное распределение, без изменений.
T < 1 (например, 0.5): Делает распределение более "пиковым". Вероятности высоковероятных слов увеличиваются, а низковероятных — уменьшаются. Текст становится более предсказуемым и консервативным.
T > 1 (например, 1.5): Делает распределение более "плоским". Вероятности разных слов выравниваются. Текст становится более случайным, креативным (но иногда и бессвязным).
Применение: Используется, когда нужна креативность и разнообразие (написание стихов, диалоги чат-ботов, идеи).
top-P sampling
Мы задаем параметр p (например, 0.9).
Сортируем все слова по убыванию вероятности.
Берем наименьший возможный набор слов, сумма вероятностей которых превышает p.
Пересчитываем распределение вероятностей для этого набора и сэмплируем из него.

Преимущество: Top-p автоматически адаптируется к "остроте" распределения на каждом шаге. Он гарантирует, что мы всегда будем выбирать из достаточно вероятных слов, без фиксированного и потенциально вредного ограничения на количество.

10. Проведите меня через алгоритм токенизации Byte-Pair Encoding (BPE, BBPE), WordPiece, SentencePiece etc.

BPE
Фаза 1: Подготовка данных
Исходный текст разбивается на символы: "low" → ["l", "o", "w"] и создается начальный словарь = все уникальные символы
Подсчитываем частоты всех пар соседних символов
Находим самую частую пару и объединяем ее в новый токен
Добавляем новый токен в словарь
Повторяем до достижения нужного размера словаря
BBPE
То же самое, что и BPE, но работает на уровне байтов (256 возможных "символов")
Начальный словарь = 256 байтов
WordPiece
Объединяет пары, которые максимизируют правдоподобие данных
score(AB) = freq(AB) / (freq(A) * freq(B)) Объединяем пару с максимальным score.
Текст разбивается на слова (по пробелам, знакам препинания)
Подсчитывается частота каждого слова
Слова разбиваются на символы - это начальный словарь

Исходный текст: "playing players played play"

Частоты слов: {"playing": 5, "players": 3, "played": 2, "play": 1}

Начальное разбиение:

"playing" -> ["p", "l", "a", "y", "i", "n", "g"] (частота 5)

"players" -> ["p", "l", "a", "y", "e", "r", "s"] (частота 3)

"played" -> ["p", "l", "a", "y", "e", "d"] (частота 2)

"play" -> ["p", "l", "a", "y"] (частота 1)

Словарь: [p,l,a,y,i,n,g,e,r,s,d]

Score(p, l) = 11/(11 * 11) = 1/11

Score(e,r) = 1/(5*1)=0.2
SentencePiece
Текст напрямую → токены. Пробел представляется как специальный символ (например, _)

11. Что такое метрики BLEU, ROUGE, perplexity etc. и для чего они используются? Как еще мы можем померить качество модели?

Что это?
Perplexity измеряет, насколько хорошо языковая модель предсказывает следующий токен. Чем ниже перплексия, тем лучше модель.
Как вычисляется:

$$Perplexity = \exp\left(-\frac{1}{N} * \Sigma \log P(word_i \mid context)\right)$$

Где:
$P(word_i \mid context)$ - вероятность, которую модель присваивает правильному следующему слову
N - общее количество слов в тестовом наборе. Если модель на тестовом наборе дает перплексию 25, это значит, что в среднем она "выбирает" из 25 равновероятных слов.
BLEU измеряет качество машинного перевода путем сравнения с человеческими переводами (референсами).
Ключевая идея:
Сравнивает n-граммы машинного перевода с n-граммами эталонных переводов.

$$\text{BP} = \begin{cases} 1 & \text{if } c > r \\ e^{1-r/c} & \text{if } c \le r \end{cases} \qquad \text{BLEU} = \text{BP} \cdot \exp\left(\sum_{n=1}^{N} w_n \log p_n\right)$$

p-1 = # совпавших униграмм / # всех униграмм в кандидате

p-2 = # совпавших биграмм / # всех биграмм в кандидате
ROUGE измеряет качество суммаризации путем подсчета перекрытия между машинной и эталонной суммами.
ROUGE-N = # совпавших n-грамм / # n-грамм в референсе
ROUGE-L: На основе самой длинной общей подпоследовательности (LCS)
Precision_L = LCS(candidate, reference) / len(candidate)
Recall_L = LCS(candidate, reference) / len(reference)
F1_L = 2 * Precision_L * Recall_L / (Precision_L + Recall_L)

12. Проведите меня через модели семейства BERT.

Модель видит все слова одновременно в обе стороны.
Он как редактор — видит весь текст сразу и может анализировать любые связи между словами.
Специальные токены:
[CLS] — классификация (в начале последовательности)
[SEP] — разделитель предложений
[MASK] — маскирование для обучения
BERT обучался на двух задачах одновременно:
1) Модель должна предсказать оригинальное слово
Нюансы:
80%: заменяем на [MASK]
10%: заменяем на случайное слово
10%: оставляем без изменений

2) Next Sentence Prediction (NSP) — 50% пар
Как работает:
Берем два предложения A и B
50% случаев: B — настоящее следующее предложение
50% случаев: B — случайное предложение из корпуса
Модель предсказывает, следует ли B за A

13. Проведите меня через модели семейства GPT.

GPT — это авторегрессионная языковая модель, которая предсказывает следующее слово на основе предыдущих.
Ключевая идея:
"Прочитай всё, что было до, и предскажи следующее слово"
Обучена на задаче предсказания следующего токена
Использует только Decoder часть Трансформера
Аналогия: GPT — это очень начитанный писатель, который всегда пишет текст слева направо, опираясь только на то, что уже написал.
Архитектурные особенности:
Маскированное самовнимание
Авторегрессивная генерация
Позиционные эмбеддинги(Учитывают порядок слов)

14. Сравните архитектуры BERT, GPT и T5: в чем их ключевые различия?

Модель	**Архитектура**	**Направленность**
BERT	Encoder-only	Двунаправленная - видит все слова сразу
GPT	Decoder-only	Односторонняя - видит только предыдущие слова
T5	Encoder-Decoder	Двунаправленная + авторегрессивная

Основное применение

Модель	Основные задачи
BERT	• Классификация • NER • Извлечение информации • Понимание текста
GPT	• Генерация текста • Диалоги • Творческие задачи
T5	• **Все задачи как text-to-text** • Перевод • Суммаризация • Классификация • Рефразинг

Input-Output формат

Модель	Формат
BERT	[CLS] текст [SEP] → Класс/Эмбеддинги
GPT	промпт → продолжение
T5	"задача: текст" → результат

Ключевой вывод:
BERT — лучший для понимания текста
GPT — лучший для генерации текста
T5 — универсальный подход "все задачи как text-to-text"

15. In-context learning: zero-shot, few-shot prompts, chain-of-thought.

In-Context Learning (ICL)
Суть: Способность модели выполнять задачи без обновления весов, используя только инструкции и примеры в промпте.
1. Zero-Shot Learning
Только инструкция, без примеров
Модель использует знания, полученные при предобучении
2. Few-Shot Learning
Инструкция + несколько примеров
Модель изучает паттерн по примерам
3. Chain-of-Thought (CoT)
Пошаговое рассуждение перед ответом
Особенно эффективно для сложных логических и математических задач

16. Parameter-Efficient Finetuning (PEFT): Soft Prompts, (Q)LoRA, IA3.

Parameter-Efficient Fine-Tuning (PEFT) — методы, которые позволяют настраивать большие модели с минимальными затратами.
Зачем нужен PEFT?
Проблема: Полный fine-tuning больших моделей (GPT, LLaMA) требует:
💰 Огромной GPU памяти
⏳ Много времени
💾 Места для хранения весов (десятки GB)
Решение: PEFT — дообучение только небольшой части параметров
1. Soft Prompts (Prompt Tuning)
Идея:
Заморозить всю модель
Обучает только несколько токенов в начале промпта
2. LoRA (Low-Rank Adaptation)

Идея:
Заморозить оригинальные веса модели
Для каждой матрицы W добавить низкоранговую адаптацию
Формула:
$h = Wx + BAx$
Где:
W — замороженные оригинальные веса
A, B — маленькие обучаемые матрицы (ранг $r \ll d$)

3. QLoRA — "Сжатая" версия LoRA

Проблема:
Даже с LoRA большие модели не помещаются в память
Решение:
Сжимаем модель до 4-бит (как zip-архив)
Применяем LoRA поверх сжатой модели
Во время обучения: временно распаковываем только нужные части

4. IA³ (Infused Adapter by Inhibiting and Amplifying Inner Activations)

Идея:
Добавляем learnable коэффициенты которые "усиливают" или "ослабляют" определенные нейроны.
Умножаем на активации внутри модели
Формула:
$h = (Wx) * l$
Где l — обучаемый вектор-множитель

17. **Проведите меня через Reinforcement Learning with Human Feedback (RLHF): Proximal Policy Optimization (PPO) и Direct Preference Optimization (DPO).**

Проблема: Модель после предобучения знает язык, но:
Может генерировать вредоносный контент
Не следует инструкциям
Не учитывает человеческие предпочтения

RLHF — это процесс "воспитания" модели, чтобы она была полезной, честной и безопасной.

Классический RLHF (3 шага)

Шаг 1: Обучение Reward Model (Модели Вознаграждения)
Процесс:
Берем промпты и генерируем несколько ответов
Люди оценивают, какой ответ лучше: A > B > C
Обучаем модель предсказывать человеческие предпочтения
Шаг 2: Fine-tuning с PPO
Аналогия: Актер (модель) получает награду от критика (Reward Model) за хорошую игру.
PPO (Proximal Policy Optimization) — правила обучения:
Генерация: Модель генерирует ответы
Оценка: Reward Model оценивает каждый ответ
Корректировка: Модель обновляет веса, чтобы генерировать более "вознаграждаемые" ответы
Стабильность: PPO следит, чтобы модель не менялась слишком резко
Проблемы классического RLHF
Сложность: Нужно обучать две модели (Policy + Reward)
Нестабильность: PPO может "сломать" модель
Вычисления: Очень ресурсоемко

DPO (Direct Preference Optimization) — Упрощенная версия

Идея:
Зачем учить отдельного "критика", если можно сразу сказать модели, какие ответы хорошие?
Аналогия: Вместо того чтобы нанимать критика и актера, мы сразу тренируем актера на примерах "хорошо/плохо".
Как работает:
Собираем данные предпочтений: Промпт + Хороший ответ + Плохой ответ
Одним шагом обучаем модель предпочитать хорошие ответы

```
{
   "prompt": "Напиши приветствие",
   "chosen": "Здравствуйте! Как я могу вам помочь?",    # ХОРОШО
   "rejected": "Привет. Чего надо?"                     # ПЛОХО
}
```

18. **Что такое Retrieval-Augmented Generation (RAG)? Опишите компоненты RAG-системы. Dense Retrieval vs. Sparse Retrival: в чем преимущества и недостатки?**

RAG (Retrieval-Augmented Generation) — это подход, когда модель ищет информацию в базе знаний перед генерацией ответа.
Простая аналогия:
Обычная GPT: Студент, который отвечает по памяти
RAG: Студент, который проверяет учебник перед ответом
Проблема, которую решает RAG:
Галлюцинации (выдумывание фактов)
Устаревшая информация в модели
Незнание специфичных данных (документы компании и т.д.)
Компоненты RAG-системы
1. База знаний (Knowledge Base)
Что: Документы, статьи, базы данных, PDF-файлы
Пример: Вся документация компании, законы, медицинские статьи
2. Retriever (Поисковый модуль)
Что: Ищет релевантные документы по запросу
Как: Преобразует запрос и документы в векторы, ищет похожие
3. Generator (Генератор)
Что: Языковая модель (GPT, LLaMA и т.д.)
Задача: Генерировать ответ на основе найденных документов

Sparse Retrieval (Разреженный поиск)
Как работает:
Считает совпадение слов между запросом и документом
Методы: TF-IDF, BM25
Пример:
Запрос: "собака бежит"
Документ: "собака быстро бежит по полю"
TF-IDF: считает частоту слов "собака", "бежит"
Преимущества:
✅ Прозрачность: Понятно, почему нашел документ (по совпадению слов)
✅ Быстрота: Простые вычисления
✅ Хорошо для ключевых слов: "Python tutorial", "купить iPhone"
Недостатки:
❌ Проблема синонимов: "автомобиль" ≠ "машина"
❌ Словоформы: "бежит" ≠ "бежать" ≠ "бегу"
❌ Смысловой поиск: "веселый фильм" ≠ "комедия"
Dense Retrieval (Плотный поиск)
Как работает:
Преобразует текст в векторы (эмбеддинги) и ищет семантически похожие
Методы: DPR, Sentence-BERT, OpenAI Embeddings
Пример:
Запрос: "собака бежит" → Вектор [0.1, 0.8, -0.2, ...]
Документ: "пёс несется" → Вектор [0.12, 0.79, -0.18, ...]
Косинусная близость: 0.95 ✅
Преимущества:
✅ Понимает смысл: "автомобиль" ≈ "машина" ≈ "vehicle"
✅ Учет контекста: "яблоко" (фрукт) vs "Apple" (компания)
✅ Улучшает качество поиска для сложных запросов
Недостатки:
❌ Требует обучения или использования предобученных моделей
❌ Вычислительно сложнее
❌ Менее интерпретируемо — непонятно, почему нашел
Гибридный подход
Часто используют оба метода вместе:
Общий счет = $\alpha \times$ Dense_Score + $(1-\alpha) \times$ Sparse_Score
Пример:
Запрос: "Как настроить Wi-Fi роутер?"
Dense: находит "инструкция по настройке беспроводной сети"

Sparse: находит "Wi-Fi роутер настройка руководство"

19. Дайте определение агента. В чем отличие агента от классической LLM?

Агент — это LLM + способность действовать в окружающей среде для достижения цели.
Одношаговость vs Многошаговость
1. Пассивность vs Активность
Классическая LLM:

python
Пассивно отвечает на вопросы
вход: "Какая погода в Москве?"
выход: "Сейчас в Москве +15°C, солнечно"
Агент:

python
Активно выполняет действия
вход: "Узнай погоду в Москве и предложи, что надеть"
процесс:
1. Вызов API погоды → "Москва: +15°C, солнечно"
2. Анализ → "Легкая куртка, солнцезащитные очки"
3. Действие → Отправляет рекомендацию в чат
2. Одношаговость vs Многошаговость
Классическая LLM:
Один запрос → один ответ
Не помнит предыдущие взаимодействия в рамках задачи
Агент:
Ставит цель → планирует шаги → выполняет → корректирует
Сохраняет контекст всей миссии
Пример планирования агента:
Цель: "Организуй встречу с коллегами"
План агента:
1. Проверить календарь → найти свободное время
2. Написать коллегам → узнать их доступность
3. Согласовать время → создать событие в календаре
4. Отправить напоминание за час до встречи
3. Ответ vs Результат
Классическая LLM:
Продукт: текстовый ответ
Критерий успеха: правдивость, полезность ответа
Агент:
Продукт: выполненная задача
Критерий успеха: достижение поставленной цели