Конспект по теме "Языковые представления"

Embeddings

Чтобы машины воспринимали слова, картинки или аудио, их преобразовывают в векторный вид. Когда работают с текстом, его тоже переводят в векторный формат, или векторные представления. Частный случай этих представлений — **word embeddings** (англ. «слова-вложения»; «эмбеддинги»). Работают они так: сложная структура (текст) вкладывается в более простую — вектор.

Векторы-эмбеддинги содержат данные о соотношении разных слов и их свойствах. Привычное понимание свойства слова, его смысла и контекста справедливо и для машинного обучения.

Свойства — это скрытые смыслы слова. Смысл, или семантика слова, — это лексическое значение слова, его отличие от других слов. Но слово обычно не живёт само по себе, его окружают другие слова. Именно контекст и определяет смысл слов.

Эти понятия пригодятся нам в определении близости слов в их векторном представлении. Так **близкие**, или **похожие векторы**, могут отображать похожие слова. Сходство векторов рассчитывается знакомым вам евклидовым расстоянием: чем меньше расстояние, тем сильнее похожи векторы.

Word2vec

Упрощённо разберём, как работает *word2vec*. **Смысл** слов определяется их контекстом. Задача *word2vec* — предсказать: соседи или нет — заданные слова. Слова считаются соседями, если находятся в одном **«окне»** (максимальном расстоянии между словами). Выходит, пара слов — это признаки, а являются ли они соседями — целевой признак.

Следующая задача *word2vec* — научить модель отличать истинные пары соседей от случайных. А это уже походит на задачу бинарной классификации, где признаки — это слова, а целевой признак — ответ на вопрос: перед нами истинные слова-соседи или нет.

Embeddings для классификации

Применим векторное представление к задаче анализа текстов. Допустим, у нас есть корпус текстов, которые нужно классифицировать. Тогда наша модель будет состоять из двух блоков:

1. Модели перевода слов в векторное представление: исходные тексты преобразуются в векторы.
2. Модели классификации: на основе векторов текстов получаются прогнозы.

Разберём, как работает эта схема:

1. Прежде чем перейти к векторному представлению слов, проводится знакомая вам предобработка текста:

- Выполняют токенизацию каждого текста, то есть его разбивают на слова;
- Слова лемматизируют: приводят к начальной словарной форме (более сложные модели, например, *BERT*, этого не требуют: они сами понимают формы слов);
- Текст очищают от стоп-слов и ненужных символов;
- Для корректной работы алгоритма добавляют маркеры начала и конца предложения (они приравниваются к токенам).

2. На выходе у каждого исходного текста образуется свой список токенов.

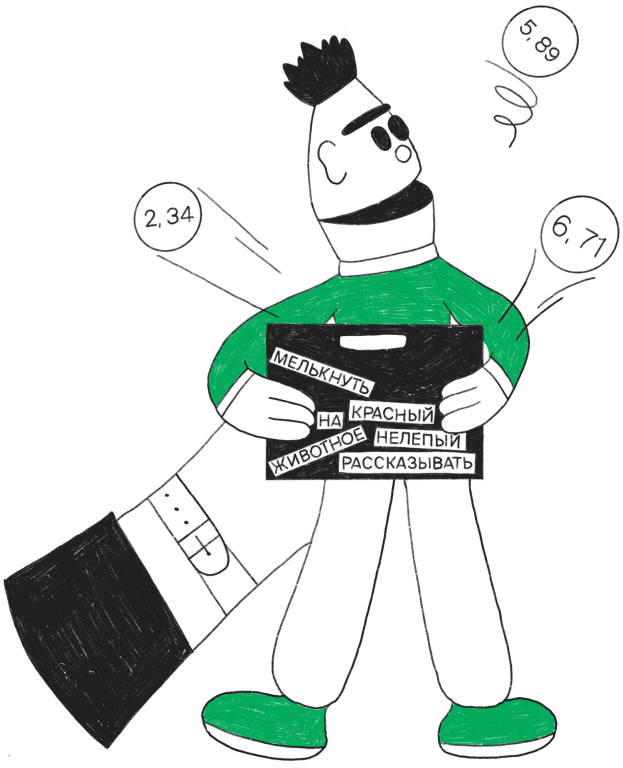
3. Затем токены передают модели, которая переводит их в векторные представления. Для этого модель обращается к составленному заранее словарю токенов. На выходе для каждого текста образуются векторы заданной длины.

4. На финальном этапе модели передают признаки (векторы). И она прогнозирует эмоциональную окраску текста — 0 («отрицательная») или 1 («положительная»).

BERT

BERT (англ. *Bidirectional Encoder Representations from Transformers*) — нейронная сеть для создания модели языка. Её разработали в компании *Google*, чтобы повысить релевантность результатов поиска. Этот алгоритм понимает контекст запросов, а не просто анализирует фразы. Для машинного обучения она ценна тем, что помогает строить векторные представления. Причём в анализе текстов применяют уже предобученную на большом корпусе модель. Такие предобученные версии *BERT* годятся для работы с текстами на 104 языках мира, включая русский.

BERT — это результат эволюции модели *word2vec*. В ходе её развития были придуманы и другие модели: **FastText**, **GloVe** (англ. *Global Vectors for Word Representation*), **ELMO** (англ. *Embeddings from Language Models*) и **GPT** (англ. *Generative Pre-Training Transformer*). Сейчас самые точные — это *BERT* и *GPT-2*, которого нет в открытом доступе.



BERT учитывает контекст не только соседних слов, но и более дальних родственников. Работает так:

- На входе модель получает, например, такую фразу: «Красный клюв тупика *[MASK]* на голубом *[MASK]*», где **MASK** — это неизвестные слова, будто закрытые маской. Модель должна угадать эти спрятанные слова.
- Модель обучается определять, связаны ли в предложении слова между собой. У нас были скрыты такие слова: «мелькнул» и «небе». Модель должна понять, что одно слово — продолжение другого. Скажем, если вместо «мелькнул» спрятать слово «прополз», то связи модель не найдёт.

RuBERT и предобработка

Повторим задачу. Перед вами большой датасет с твитами. Нужно научиться определять, какие твиты негативной тональности, а какие — позитивной. Чтобы решить эту задачу, из открытого репозитория DeepPavlov возьмём модель *RuBERT*, обученную на разговорном русскоязычном корпусе.

Решим эту задачу на **PyTorch**. Глубоко разбираться в средствах этой библиотеки мы не будем. Она применяется в задачах обработки естественного текста и компьютерного зрения. А нам нужна для работы с моделями типа *BERT*. Они находятся в библиотеке **transformers**. Импортируем их:

```
import torch
import transformers
```

Прежде чем перевести тексты в векторы, подготовим их. У *RuBERT* есть собственный **токенизатор**. Это инструмент, который разбивает и преобразует исходные тексты в список токенов, которые есть, например, в словаре RuBERT. Лемматизация не требуется.

Начинаем предобработку текстов:

1. Инициализируем токенизатор как объект класса *BertTokenizer()*. Передадим ему аргумент *vocab_file* — это файл со словарём, на котором обучалась модель. Он может быть, например, в текстовом формате (*txt*).

```
tokenizer = transformers.BertTokenizer(
    vocab_file='vocab.txt')
```

2. Преобразуем текст в номера токенов из словаря методом **encode()**:

```
tokenizer.encode('Очень удобно использовать уже готовый трансформатор текста',
                  add_special_tokens=True)
```

Для корректной работы модели мы указали аргумент *add_special_tokens* (англ. «добавить специальные токены»), равный *True*. Это значит, что к любому преобразуемому тексту добавляется токен начала (101) и токен конца текста (102).

3. Применим метод **padding**, чтобы после токенизации длины исходных текстов в корпусе были равными. Только при таком условии будет работать модель *BERT*. Пусть стандартной длиной вектора *n* будет длина наибольшего во всём датасете вектора. Остальные векторы дополним нулями:

```
vector = tokenizer.encode('Очень удобно использовать уже готовый трансформатор текста',
                          add_special_tokens=True)
n = 280
# англ. вектор с отступами
padded = vector + [0]*(n - len(vector))
print(padded)
```

Теперь поясним модели, что нули не несут значимой информации. Это нужно для компоненты модели, которая называется «внимание» (англ. *attention*). Отбросим эти токены и «создадим маску» для действительно важных токенов, то есть укажем нулевые и не нулевые значения:

```
attention_mask = np.where(padded != 0, 1, 0)
print(attention_mask.shape)
```

Эмбеддинги RuBERT

Инициализируем конфигурацию **BertConfig**. В качестве аргумента передадим ей *JSON*-файл с описанием настроек модели. **JSON** (англ. *JavaScript Object Notation*) — это организованный по ключам поток цифр, букв, двоеточий и фигурных скобок, который возвращает сервер при запросе.

Затем инициализируем саму модель класса **BertModel**. Передадим ей файл с предобученной моделью и конфигурацией:

```
config = transformers.BertConfig.from_json_file('bert_config.json')
model = transformers.BertModel.from_pretrained('rubert_model.bin', config=config)
```

Начнём преобразование текстов в эмбеддинги. Это может занять несколько минут, поэтому подключим библиотеку **tqdm**. Она нужна, чтобы наглядно показать индикатор прогресса. В *Jupyter* применим функцию *notebook()* из этой библиотеки:

```
from tqdm import notebook
```

Эмбеддинги модель *BERT* создаёт батчами. Чтобы хватило оперативной памяти, сделаем размер батча небольшим:

```
batch_size = 100
```

Сделаем цикл по батчам. Отображать прогресс будет функция *notebook()*:

```
# сделаем пустой список для хранения эмбеддингов твитов
embeddings = []

for i in notebook.tqdm(range(padded.shape[0] // batch_size)):
    ...
```

Преобразуем данные в формат **тензоров** (англ. *tensor*) — многомерных векторов в библиотеке *torch*. Тип данных *LongTensor* хранит числа в «длинном формате», то есть выделяет на каждое число 64 бита.

```
# преобразуем данные
batch = torch.LongTensor(padded[batch_size*i:batch_size*(i+1)])
# преобразуем маску
attention_mask_batch = torch.LongTensor(attention_mask[batch_size*i:batch_size*(i+1)])
```

Чтобы получить эмбеддинги для батча, передадим модели данные и маску:

```
batch_embeddings = model(batch, attention_mask=attention_mask_batch)
```

Для ускорения вычисления функцией **no_grad()** в библиотеке *torch* укажем, что градиенты не нужны: модель *BERT* обучать не будем.

```
with torch.no_grad():
    batch_embeddings = model(batch, attention_mask=attention_mask_batch)
```

Из полученного тензора извлечём нужные элементы и добавим в список всех эмбеддингов:

```
# преобразуем элементы методом numpy() к типу numpy.array
embeddings.append(last_hidden_states[0][:,0,:].numpy())
```

Получаем такой цикл:

```
embeddings = []

for i in notebook.tqdm(range(padded.shape[0] // batch_size)):
    batch = torch.LongTensor(padded[batch_size*i:batch_size*(i+1)])
    attention_mask_batch = torch.LongTensor(attention_mask[batch_size*i:batch_size*(i+1)])

    with torch.no_grad():
        batch_embeddings = model(batch, attention_mask=attention_mask_batch)

    embeddings.append(last_hidden_states[0][:,0,:].numpy())
```

Соберём все эмбеддинги в матрицу признаков вызовов функции *concatenate()*:

```
features = np.concatenate(embeddings)
```

Признаки готовы, можно обучать модель!