Физтех-Школа Прикладной математики и информатики (ФПМИ) МФТИ

Задача определения частей речи, Part-Of-Speech Tagger (POS)

Мы будем решать задачу определения частей речи (POS-теггинга) с помощью скрытой марковской модели (HMM).

```
!pip install rnnmorph
```

```
  Downloading https://files.pythonhosted.org/packages/2d/b1/c9377d472a04fb9b84f59365560d68b5d868b589691f32545eb606b3be48/russian-tagsets-0.6.tar.gz
Requirement already satisfied: tqdm>=4.14.0 in /usr/local/lib/python3.7/dist-packages (from rnnmorph) (4.41.1)
Collecting jsonpickle>=0.9.4
  Downloading https://files.pythonhosted.org/packages/bb/1a/f2db026d4d682303793559f1c2bb425ba3ec0d6fd7ac63397790443f2461/jsonpickle-2.0.0-py2.py3-none-any.whl
Requirement already satisfied: nltk>=3.2.5 in /usr/local/lib/python3.7/dist-packages (from rnnmorph) (3.2.5)
Requirement already satisfied: joblib>=0.11 in /usr/local/lib/python3.7/dist-packages (from scikit-learn>=0.18.1->rnnmorph) (1.0.1)
Requirement already satisfied: keras-preprocessing~=1.1.2 in /usr/local/lib/python3.7/dist-packages (from tensorflow>=1.1.0->rnnmorph) (1.1.2)
Requirement already satisfied: six~=1.15.0 in /usr/local/lib/python3.7/dist-packages (from tensorflow>=1.1.0->rnnmorph) (1.15.0)
Requirement already satisfied: astunparse~=1.6.3 in /usr/local/lib/python3.7/dist-packages (from tensorflow>=1.1.0->rnnmorph) (1.6.3)
Requirement already satisfied: gast==0.3.3 in /usr/local/lib/python3.7/dist-packages (from tensorflow>=1.1.0->rnnmorph) (0.3.3)
Requirement already satisfied: wheel~=0.35 in /usr/local/lib/python3.7/dist-packages (from tensorflow>=1.1.0->rnnmorph) (0.36.2)
Requirement already satisfied: google-pasta~=0.2 in /usr/local/lib/python3.7/dist-packages (from tensorflow>=1.1.0->rnnmorph) (0.2.0)
Requirement already satisfied: protobuf>=3.9.2 in /usr/local/lib/python3.7/dist-packages (from tensorflow>=1.1.0->rnnmorph) (3.12.4)
Requirement already satisfied: termcolor~=1.1.0 in /usr/local/lib/python3.7/dist-packages (from tensorflow>=1.1.0->rnnmorph) (1.1.0)
Requirement already satisfied: flatbuffers~=1.12.0 in /usr/local/lib/python3.7/dist-packages (from tensorflow>=1.1.0->rnnmorph) (1.12)
Requirement already satisfied: typing-extensions~=3.7.4 in /usr/local/lib/python3.7/dist-packages (from tensorflow>=1.1.0->rnnmorph) (3.7.4.3)
Requirement already satisfied: wrapt~=1.12.1 in /usr/local/lib/python3.7/dist-packages (from tensorflow>=1.1.0->rnnmorph) (1.12.1)
Requirement already satisfied: tensorboard~=2.4 in /usr/local/lib/python3.7/dist-packages (from tensorflow>=1.1.0->rnnmorph) (2.4.1)
Requirement already satisfied: tensorflow-estimator<2.5.0,>=2.4.0 in /usr/local/lib/python3.7/dist-packages (from tensorflow>=1.1.0->rnnmorph) (2.4.0)
Requirement already satisfied: grpcio~=1.32.0 in /usr/local/lib/python3.7/dist-packages (from tensorflow>=1.1.0->rnnmorph) (1.32.0)
Requirement already satisfied: opt-einsum~=3.3.0 in /usr/local/lib/python3.7/dist-packages (from tensorflow>=1.1.0->rnnmorph) (3.3.0)
Requirement already satisfied: h5py~=2.10.0 in /usr/local/lib/python3.7/dist-packages (from tensorflow>=1.1.0->rnnmorph) (2.10.0)
Requirement already satisfied: absl-py~=0.10 in /usr/local/lib/python3.7/dist-packages (from tensorflow>=1.1.0->rnnmorph) (0.10.0)
Requirement already satisfied: pyyaml in /usr/local/lib/python3.7/dist-packages (from keras>=2.0.6->rnnmorph) (3.13)
Collecting pymorphy2-dicts-ru<3.0,>=2.4
  Downloading https://files.pythonhosted.org/packages/3a/79/bea0021eeb7eeefde22ef9e96badf174068a2dd20264b9a378f2be1cdd9e/pymorphy2_dicts_ru-2.4.417127.4579844-py2.py3-none-any.whl (8.2MB)
     |████████████████████████████████| 8.2MB 43.2MB/s
Requirement already satisfied: docopt>=0.6 in /usr/local/lib/python3.7/dist-packages (from pymorphy2>=0.8->rnnmorph) (0.6.2)
Collecting dawg-python>=0.7.1
  Downloading https://files.pythonhosted.org/packages/6a/84/ff1ce2071d4c650ec85745766c0047ccc3b5036f1d03559fd46bb38b5eeb/DAWG_Python-0.7.2-py2.py3-none-any.whl
Requirement already satisfied: importlib-metadata; python_version < "3.8" in /usr/local/lib/python3.7/dist-packages (from jsonpickle>=0.9.4->rnnmorph) (3.7.2)
Requirement already satisfied: setuptools in /usr/local/lib/python3.7/dist-packages (from protobuf>=3.9.2->tensorflow>=1.1.0->rnnmorph) (54.1.2)
Requirement already satisfied: requests<3,>=2.21.0 in /usr/local/lib/python3.7/dist-packages (from tensorboard~=2.4->tensorflow>=1.1.0->rnnmorph) (2.23.0)
Requirement already satisfied: werkzeug>=0.11.15 in /usr/local/lib/python3.7/dist-packages (from tensorboard~=2.4->tensorflow>=1.1.0->rnnmorph) (1.0.1)
Requirement already satisfied: google-auth<2,>=1.6.3 in /usr/local/lib/python3.7/dist-packages (from tensorboard~=2.4->tensorflow>=1.1.0->rnnmorph) (1.27.1)
Requirement already satisfied: tensorboard-plugin-wit>=1.6.0 in /usr/local/lib/python3.7/dist-packages (from tensorboard~=2.4->tensorflow>=1.1.0->rnnmorph) (1.8.0)
Requirement already satisfied: google-auth-oauthlib<0.5,>=0.4.1 in /usr/local/lib/python3.7/dist-packages (from tensorboard~=2.4->tensorflow>=1.1.0->rnnmorph) (0.4.3)
Requirement already satisfied: markdown>=2.6.8 in /usr/local/lib/python3.7/dist-packages (from tensorboard~=2.4->tensorflow>=1.1.0->rnnmorph) (3.3.4)
Requirement already satisfied: zipp>=0.5 in /usr/local/lib/python3.7/dist-packages (from importlib-metadata; python_version < "3.8"->jsonpickle>=0.9.4->rnnmorph) (3.4.1)
Requirement already satisfied: idna<3,>=2.5 in /usr/local/lib/python3.7/dist-packages (from requests<3,>=2.21.0->tensorboard~=2.4->tensorflow>=1.1.0->rnnmorph) (2.10)
Requirement already satisfied: urllib3!=1.25.0,!=1.25.1,<1.26,>=1.21.1 in /usr/local/lib/python3.7/dist-packages (from requests<3,>=2.21.0->tensorboard~=2.4->tensorflow>=1.1.0->rnnmorph) (1.24.3)
Requirement already satisfied: certifi>=2017.4.17 in /usr/local/lib/python3.7/dist-packages (from requests<3,>=2.21.0->tensorboard~=2.4->tensorflow>=1.1.0->rnnmorph) (2020.12.5)
Requirement already satisfied: chardet<4,>=3.0.2 in /usr/local/lib/python3.7/dist-packages (from requests<3,>=2.21.0->tensorboard~=2.4->tensorflow>=1.1.0->rnnmorph) (3.0.4)
Requirement already satisfied: rsa<5,>=3.1.4; python_version >= "3.6" in /usr/local/lib/python3.7/dist-packages (from google-auth<2,>=1.6.3->tensorboard~=2.4->tensorflow>=1.1.0->rnnmorph) (4.7.2)
Requirement already satisfied: pyasn1-modules>=0.2.1 in /usr/local/lib/python3.7/dist-packages (from google-auth<2,>=1.6.3->tensorboard~=2.4->tensorflow>=1.1.0->rnnmorph) (0.2.8)
Requirement already satisfied: cachetools<5.0,>=2.0.0 in /usr/local/lib/python3.7/dist-packages (from google-auth<2,>=1.6.3->tensorboard~=2.4->tensorflow>=1.1.0->rnnmorph) (4.2.1)
Requirement already satisfied: requests-oauthlib>=0.7.0 in /usr/local/lib/python3.7/dist-packages (from google-auth-oauthlib<0.5,>=0.4.1->tensorboard~=2.4->tensorflow>=1.1.0->rnnmorph) (1.3.0)
Requirement already satisfied: pyasn1>=0.1.3 in /usr/local/lib/python3.7/dist-packages (from rsa<5,>=3.1.4; python_version >= "3.6"->google-auth<2,>=1.6.3->tensorboard~=2.4->tensorflow>=1.1.0->rnnmorph) (0.4.8)
Requirement already satisfied: oauthlib>=3.0.0 in /usr/local/lib/python3.7/dist-packages (from requests-oauthlib>=0.7.0->google-auth-oauthlib<0.5,>=0.4.1->tensorboard~=2.4->tensorflow>=1.1.0->rnnmorph) (3.1.0)
Building wheels for collected packages: rnnmorph, russian-tagsets
  Building wheel for rnnmorph (setup.py) ... done
  Created wheel for rnnmorph: filename=rnnmorph-0.4.0-cp37-none-any.whl size=10521037 sha256=125585143a5f715d6e4b7df796ef33c1f0ccc48d5d33d1d19f379e6df481920e
  Stored in directory: /root/.cache/pip/wheels/61/74/5d/3c6c523a759b67e6a81677e2aad003321536587d1575a4face
  Building wheel for russian-tagsets (setup.py) ... done
  Created wheel for russian-tagsets: filename=russian_tagsets-0.6-cp37-none-any.whl size=24635 sha256=3b2fc9384a5086f06f7ad2f8e4a119f616ffb97c028d22707983614de6ac1926
  Stored in directory: /root/.cache/pip/wheels/e8/9d/dd/4679aca4031fdb0d3ad65e165ba5343e61441ed7ad587a08e6
Successfully built rnnmorph russian-tagsets
Installing collected packages: pymorphy2-dicts-ru, dawg-python, pymorphy2, russian-tagsets, jsonpickle, rnnmorph
Successfully installed dawg-python-0.7.2 jsonpickle-2.0.0 pymorphy2-0.9.1 pymorphy2-dicts-ru-2.4.417127.4579844 rnnmorph-0.4.0 russian-tagsets-0.6
```

```
import nltk
import pandas as pd
import numpy as np
from collections import OrderedDict, deque, Counter
from nltk.corpus import brown
import matplotlib.pyplot as plt
from copy import deepcopy
```

Вам в помощь http://www.nltk.org/book/

Загрузим brown корпус

```
nltk.download('brown')
```

```
[nltk_data] Downloading package brown to /root/nltk_data...
[nltk_data]   Unzipping corpora/brown.zip.
True
```

Существует множество наборов грамматических тегов, или тегсетов, например:

- НКРЯ
- Mystem
- UPenn
- OpenCorpora (его использует pymorphy2)
- Universal Dependencies

Существует не одна система тегирования, поэтому будьте внимательны, когда прогнозируете тег слов в тексте и вычисляете качество прогноза. Можете получить несправедливо низкое качество вашего решения.

На данный момент стандартом является **Universal Dependencies**. Подробнее про проект можно почитать вот тут, а про теги — вот тут

```
nltk.download('universal_tagset')
```

```
[nltk_data] Downloading package universal_tagset to /root/nltk_data...
[nltk_data]   Unzipping taggers/universal_tagset.zip.
True
```

- ADJ: adjective
- ADP: adposition
- ADV: adverb
- AUX: auxiliary
- CCONJ: coordinating conjunction
- DET: determiner
- INTJ: interjection
- NOUN: noun
- NUM: numeral
- PART: particle
- PRON: pronoun
- PROPN: proper noun
- PUNCT: punctuation
- SCONJ: subordinating conjunction
- SYM: symbol
- VERB: verb
- X: other

Мы имеем массив предложений пар (слово-тег)

```
brown_tagged_sents = brown.tagged_sents(tagset="universal")
brown_tagged_sents
```

```
[[('The', 'DET'), ('Fulton', 'NOUN'), ('County', 'NOUN'), ('Grand', 'ADJ'), ('Jury', 'NOUN'), ('said', 'VERB'), ('Friday', 'NOUN'), ('an', 'DET'), ('investigation', 'NOUN'), ('of', 'ADP'), ("Atlanta's", 'NOUN'), ('recent', 'ADJ'), (
```

Первое предложение

```
brown_tagged_sents[0]
```

```
[('The', 'DET'),
 ('Fulton', 'NOUN'),
 ('County', 'NOUN'),
 ('Grand', 'ADJ'),
 ('Jury', 'NOUN'),
 ('said', 'VERB'),
 ('Friday', 'NOUN'),
 ('an', 'DET'),
 ('investigation', 'NOUN'),
 ('of', 'ADP'),
 ("Atlanta's", 'NOUN'),
 ('recent', 'ADJ'),
 ('primary', 'NOUN'),
 ('election', 'NOUN'),
 ('produced', 'VERB'),
 ('``', '.'),
 ('no', 'DET'),
 ('evidence', 'NOUN'),
 ("''", '.'),
 ('that', 'ADP'),
 ('any', 'DET'),
 ('irregularities', 'NOUN'),
 ('took', 'VERB'),
 ('place', 'NOUN'),
 ('.', '.')]
```

Все пары (слово-тег)

```
brown_tagged_words = brown.tagged_words(tagset='universal')
brown_tagged_words
```

```
[('The', 'DET'), ('Fulton', 'NOUN'), ...]
```

Проанализируйте данные, с которыми Вы работаете. Используйте `nltk.FreqDist()` для подсчета частоты встречаемости тега и слова в нашем корпусе. Под частой элемента подразумевается кол-во этого элемента в корпусе.

```
# Приведем слова к нижнему регистру
brown_tagged_words = list(map(lambda x: (x[0].lower(), x[1]), brown_tagged_words))
```

```
print('Кол-во предложений: ', len(brown_tagged_sents))
tags = [tag for (word, tag) in brown_tagged_words] # наши теги
words = [word for (word, tag) in brown_tagged_words] # наши слова

tag_num = pd.Series(Counter(tags)).sort_values(ascending=False) # тег - кол-во тега в корпусе
word_num = pd.Series(Counter(words)).sort_values(ascending=False) # слово - кол-во слова в корпусе
```

```
Кол-во предложений:  57340
```

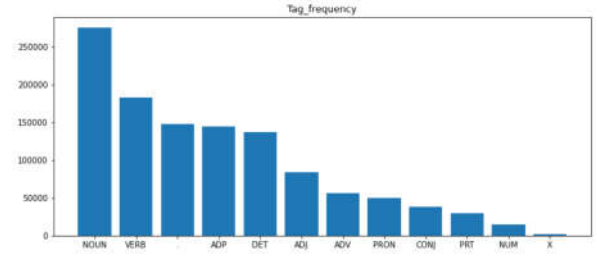
```
tag_num
```

```
NOUN    275558
VERB    182750
.       147565
ADP     144766
DET     137019
ADJ      83721
ADV      56239
PRON     49334
CONJ     38151
PRT      29829
NUM      14874
X         1386
dtype: int64
```

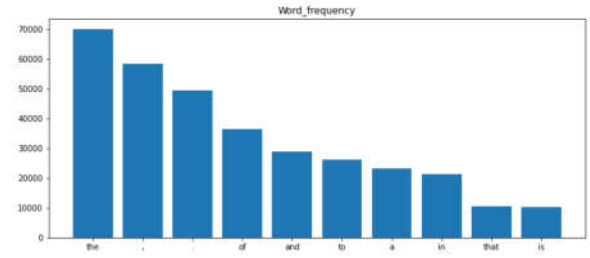
```
word_num
```

```
the             69971
,               58334
.               49346
of              36412
and             28853
                ...
waist-length        1
caveat              1
patrician           1
unfit               1
stupefying          1
Length: 49815, dtype: int64
```

```
plt.figure(figsize=(12, 5))
plt.bar(tag_num.index, tag_num.values)
plt.title("Tag_frequency")
plt.show()
```




```
plt.figure(figsize=(12, 5))
plt.bar(word_num.index[:10], word_num.values[:10])
plt.title("Word_frequency")
plt.show()
```



Вопрос 1:

- Кол-во слова `cat` в корпусе?

```
word_num['cat']
```

```
23
```

Вопрос 2:

- Самое популярное слово с самым популярным тегом?
  (*сначала выбираете слова с самым популярным тегом, а затем выбираете самое популярное слово из уже выбранных*)

```
most_popular_tag = tag_num.index[0]
print(f"most popular tag is {most_popular_tag}")
words_of_most_popular_tag = [word for (word, tag) in brown_tagged_words if tag == most_popular_tag]
words_of_most_popular_tag_num = pd.Series(Counter(words_of_most_popular_tag)).sort_values(ascending=False)
most_popular_word = words_of_most_popular_tag_num.index[0]
most_popular_word_frequency = words_of_most_popular_tag_num[most_popular_word]
print(f"most popular word '{most_popular_word}' occured {most_popular_word_frequency} times")
```

```
most popular tag is NOUN
most popular word 'time' occured 1597 times
```

Впоследствии обучение моделей может занимать слишком много времени, работайте с подвыборкой, например, только текстами определенных категорий.

Категории нашего корпуса:

```
brown.categories()
```

```
['adventure',
 'belles_lettres',
 'editorial',
 'fiction',
 'government',
 'hobbies',
 'humor',
 'learned',
 'lore',
 'mystery',
 'news',
 'religion',
 'reviews',
 'romance',
 'science_fiction']
```

Будем работать с категорией humor

Сделайте случайное разбиение выборки на обучение и контроль в отношении 9:1.

```
brown_tagged_sents = brown.tagged_sents(tagset="universal", categories='humor')
# Приведем слова к нижнему регистру
my_brown_tagged_sents = []
for sent in brown_tagged_sents:
    my_brown_tagged_sents.append(list(map(lambda x: (x[0].lower(), x[1]), sent)))
my_brown_tagged_sents = np.array(my_brown_tagged_sents)

from sklearn.model_selection import train_test_split
train_sents, test_sents = train_test_split(my_brown_tagged_sents, test_size=0.1, random_state=0)
```

```
/usr/local/lib/python3.7/dist-packages/ipykernel_launcher.py:6: VisibleDeprecationWarning: Creating an ndarray from ragged nested sequences (which is a list-or-tuple of lists-or-tuples-or ndarrays with different lengths or shapes) i
```

```
len(train_sents)
```

```
947
```

```
len(test_sents)
```

```
106
```

Метод максимального правдоподобия для обучения модели

- $S = s_0, s_1, \ldots, s_N$ - скрытые состояния, то есть различные теги
- $O = o_0, o_1, \ldots, o_M$ - различные слова
- $a_{ij} = p(s_j|s_i)$ - вероятность того, что, находясь в скрытом состоянии s_i, мы попадем в состояние s_j (элемент матрицы A)

- $b_{k,j} = p(o_k|s_j)$ - вероятность того, что при скрытом состоянии s_j находится слово o_k (элемент матрицы B)

$$x_t \in O, y_t \in S$$

(x_t, y_t) - слово и тег, стоящие на месте t ⇒

- X - последовательность слов
- Y - последовательность тегов

Требуется построить скрытую марковскую модель (class HiddenMarkovModel) и написать метод fit для настройки всех её параметров с помощью оценок максимального правдоподобия по размеченным данным (последовательности пар слово+тег):

- Вероятности переходов между скрытыми состояниями $p(y_t|y_{t-1})$ посчитайте на основе частот биграмм POS-тегов.
- Вероятности эмиссий наблюдаемых состояний $p(x_t|y_t)$ посчитайте на основе частот "POS-тег - слово".
- Распределение вероятностей начальных состояний $p(y_0)$ задайте равномерным.

Пример $X = [x_0, x_1], Y = [y_0, y_1]$:

$$p(X, Y) = p(x_0, x_1, y_0, y_1) = p(y_0) \cdot p(x_0, x_1, y_1|y_0) = p(y_0) \cdot p(x_0|y_0) \cdot p(x_1, y_1|x_0, y_0) =$$
$$= p(y_0) \cdot p(x_0|y_0) \cdot p(y_1|x_0, y_0) \cdot p(x_1|x_0, y_0, y_1) = (\text{в силу условий нашей модели}) =$$
$$= p(y_0) \cdot p(x_0|y_0) \cdot p(y_1|y_0) \cdot p(x_1|y_1) \Rightarrow$$

Для последовательности длины $n+1$:

$$p(X, Y) = p(x_0 \dots x_{n-1}, y_0 \dots y_{n-1}) \cdot p(y_n|y_{n-1}) \cdot p(x_n|y_n)$$

Алгоритм Витерби для применения модели

Требуется написать метод .predict для определения частей речи на тестовой выборке. Чтобы использовать обученную модель на новых данных, необходимо реализовать алгоритм Витерби. Это алгоритм динамиеского программирования, с помощью которого мы будем находить наиболее вероятную последовательность скрытых состояний модели для фиксированной последовательности слов:

$$\hat{Y} = \arg\max_Y p(Y|X) = \arg\max_Y p(Y, X)$$

Пусть $Q_{t,s}$ - самая вероятная последовательность скрытых состояний длины t с окончанием в состоянии s. $q_{t,s}$ - вероятность этой последовательности.

$$(1)\ q_{t,s} = \max_{s'} q_{t-1,s'} \cdot p(s|s') \cdot p(o_t|s)$$

$Q_{t,s}$ можно восстановить по argmax-ам.

```
class HiddenMarkovModel:
    def __init__(self):

        pass

    def fit(self, train_tokens_tags_list):
        """
        train_tokens_tags_list: массив предложений пар слово-тег (выборка для train)
        """
        tags = [tag for sent in train_tokens_tags_list
                for (word, tag) in sent]
        words = [word for sent in train_tokens_tags_list
                 for (word, tag) in sent]

        tag_num = pd.Series(Counter(tags)).sort_index()
        word_num = pd.Series(Counter(words)).sort_values(ascending=False)

        self.tags = tag_num.index
        self.words = word_num.index

        A = pd.DataFrame({'{}'.format(tag) : [0] * len(tag_num) for tag in tag_num.index}, index=tag_num.index)
        B = pd.DataFrame({'{}'.format(tag) : [0] * len(word_num) for tag in tag_num.index}, index=word_num.index)

        # Вычисляем матрицу A и B по частотам слов и тегов

        # sent - предложение
        # sent[i][0] - i слово в этом предложении, sent[i][1] - i тег в этом предложении
        for sent in train_tokens_tags_list:
            for i in range(len(sent)):
                B.loc[sent[i][0], sent[i][1]] += 1 # текущая i-пара слово-тег (обновите матрицу B аналогично A)
                if len(sent) - 1 != i: # для последнего тега нет следующего тега
                    A.loc[sent[i][1], sent[i + 1][1]] += 1 # пара тег-тег


        # переходим к вероятностям

        # нормируем по строке, то есть по всем всевозможным следующим тегам
        A = A.divide(A.sum(axis=1), axis=0)

        # нормируем по столбцу, то есть по всем всевозможным текущим словам
        B = B / np.sum(B, axis=0)

        self.A = A
        self.B = B

        return self


    def predict(self, test_tokens_list):
        """
        test_tokens_list : массив предложений пар слово-тег (выборка для test)
        """
        test_tokens_list_copy = deepcopy(test_tokens_list)

        predict_tags = OrderedDict({i : np.array([]) for i in range(len(test_tokens_list_copy))})

        for i_sent in range(len(test_tokens_list_copy)):

            current_sent = test_tokens_list_copy[i_sent] # текущее предложение
            len_sent = len(current_sent) # длина предложения

            q = np.zeros(shape=(len_sent + 1, len(self.tags)))
            q[0] = 1 # нулевое состояние (равномерная инициализация по всем s)
            back_point = np.zeros(shape=(len_sent + 1, len(self.tags))) # # argmax

            for t in range(len_sent):

                # если мы не встречали такое слово в обучении, то вместо него будет
                # самое популярное слово с самым популярным тегом (вопрос 2)
                if current_sent[t] not in self.words:
                    current_sent[t] = most_popular_word

                # через max выбираем следующий тег
                for i_s in range(len(self.tags)):

                    s = self.tags[i_s]

                    # формула (1)
                    q[t + 1][i_s] = np.max(q[t] *
                        self.A.loc[:, s] *
                        self.B.loc[current_sent[t], s])

```

```
                    # argmax формула(1)

                    # argmax, чтобы восстановить последовательность тегов
                    back_point[t + 1][i_s] = (q[t] * self.A.loc[:, s] *
                        self.B.loc[current_sent[t], s]).reset_index()[s].idxmax() # индекс

            back_point = back_point.astype('int')

            # выписываем теги, меняя порядок на реальный
            back_tag = deque()
            current_tag = np.argmax(q[len_sent])
            for t in range(len_sent, 0, -1):
                back_tag.appendleft(self.tags[current_tag])
                current_tag = back_point[t, current_tag]

            predict_tags[i_sent] = np.array(back_tag)

        return predict_tags
```

Обучите скрытую марковскую модель:

```
hmm = HiddenMarkovModel()
hmm.fit(train_sents)
```

```
<__main__.HiddenMarkovModel at 0x7fc298a3c150>
```

Проверьте работу реализованного алгоритма на следующих модельных примерах, проинтерпретируйте результат.

- 'He can stay'
- 'a cat and a dog'
- 'I have a television'
- 'My favourite character'

```
sents = [['He', 'can', 'stay'], ['a', 'cat', 'and', 'a', 'dog'], ['I', 'have', 'a', 'television'],
         ['My', 'favourite', 'character']]

predicted_tags = hmm.predict(sents)
predicted_tags
```

```
OrderedDict([(0, array(['NOUN', 'VERB', 'VERB'], dtype='<U4')),
             (1, array(['DET', 'NOUN', 'CONJ', 'DET', 'NOUN'], dtype='<U4')),
             (2, array(['NOUN', 'VERB', 'DET', 'NOUN'], dtype='<U4')),
             (3, array(['NOUN', 'NOUN', 'NOUN'], dtype='<U4'))])
```

Вопрос 3:

- Какой тег вы получили для слова `can`?

```
predicted_tags[0][sents[0].index('can')]
```

```
'VERB'
```

Вопрос 4:

- Какой тег вы получили для слова `favourite`?

```
predicted_tags[3][sents[3].index('favourite')]
```

```
'NOUN'
```

Примените модель к отложенной выборке Брауновского корпуса и подсчитайте точность определения тегов (accuracy). Сделайте выводы.

```
def accuracy_score(model, sents):
    true_pred = 0
    num_pred = 0

    for sent in sents:
        tags = [tag for (word, tag) in sent]
        words = [word for (word, tag) in sent]

        predicted_tags = model.predict([words])

        true_pred += (predicted_tags[0] == tags).astype(int).sum()
        num_pred += len(tags)

    return true_pred / num_pred
```

```
hmm_tagger_humor_accuracy = accuracy_score(hmm, test_sents)
print("HMM Tagger Humor Accuracy:", hmm_tagger_humor_accuracy * 100, '%')
```

```
HMM Tagger Humor Accuracy: 88.82847256549678 %
```

Вопрос 5:

- Какое качество вы получили(округлите до одного знака после запятой)?

```
print(f"{hmm_tagger_humor_accuracy:.1f}")
```

```
0.9
```

DefaultTagger

Вопрос 6:

- Какое качество вы бы получили, если бы предсказывали любой тег, как самый популярный тег на выборке train(округлите до одного знака после запятой)?

Вы можете испоьзовать DefaultTagger(метод tag для предсказания частей речи предложения)

```
from nltk.tag import DefaultTagger
default_tagger = DefaultTagger(most_popular_tag)
```

```
true_pred = 0
num_pred = 0

for sent in test_sents:
    tags = [tag for (word, tag) in sent]
    words = [word for (word, tag) in sent]

    predicted_tags = default_tagger.tag(words)
    predicted_tags = [tag for (word, tag) in predicted_tags]
```

```
    true_pred += (np.asarray(predicted_tags) == np.asarray(tags)).astype(int).sum()
    num_pred += len(sent)

default_tagger_humor_accuracy = true_pred / num_pred

print("Default Tagger Humor Accuracy:", default_tagger_humor_accuracy * 100, '%')
print(f"{default_tagger_humor_accuracy:.1f}")
```

```
Default Tagger Humor Accuracy: 20.217498764211566 %
0.2
```

NLTK, Rnnmorph

Вспомним первый семинар нашего курса. В том семинаре мы с вами работали с некоторыми библиотеками.

Не забудьте преобразовать систему тэгов из `'en-ptb'` в `'universal'` с помощью функции `map_tag` или используйте `tagset='universal'`

```
from nltk.tag.mapping import map_tag
```

```
import nltk
nltk.download('averaged_perceptron_tagger')

true_pred = 0
num_pred = 0

for sent in test_sents:
    tags = [tag for (word, tag) in sent]
    words = [word for (word, tag) in sent]

    predicted_tags = nltk.pos_tag(words, tagset='universal')
    predicted_tags = [tag for (word, tag) in predicted_tags]

    true_pred += (np.asarray(predicted_tags) == np.asarray(tags)).astype(int).sum()
    num_pred += len(sent)

nltk_tagger_humor_accuracy = true_pred / num_pred

print("NLTK Tagger Humor Accuracy:", nltk_tagger_humor_accuracy * 100, '%')
print(f"{nltk_tagger_humor_accuracy:.1f}")
```

```
[nltk_data] Downloading package averaged_perceptron_tagger to
[nltk_data]     /root/nltk_data...
[nltk_data]   Unzipping taggers/averaged_perceptron_tagger.zip.
NLTK Tagger Humor Accuracy: 89.22392486406328 %
0.9
```

```
from rnnmorph.predictor import RNNMorphPredictor
predictor = RNNMorphPredictor(language="en")

true_pred = 0
num_pred = 0

for sent in test_sents:
    tags = [tag for (word, tag) in sent]
    words = [word for (word, tag) in sent]

    forms  = predictor.predict(words)
    predicted_tags = [form.pos for form in forms]

    true_pred += (np.asarray(predicted_tags) == np.asarray(tags)).astype(int).sum()
    num_pred += len(sent)

rnn_morph_tagger_humor_accuracy = true_pred / num_pred

print("RNN Morph Tagger Humor Accuracy:", rnn_morph_tagger_humor_accuracy * 100, '%')
print(f"{rnn_morph_tagger_humor_accuracy:.1f}")
```

```
[nltk_data] Downloading package wordnet to /root/nltk_data...
[nltk_data]   Unzipping corpora/wordnet.zip.
[nltk_data] Downloading package averaged_perceptron_tagger to
[nltk_data]     /root/nltk_data...
[nltk_data]   Package averaged_perceptron_tagger is already up-to-
[nltk_data]       date!
[nltk_data] Downloading package universal_tagset to /root/nltk_data...
[nltk_data]   Package universal_tagset is already up-to-date!
WARNING:tensorflow:Layer LSTM_1_forward will not use cuDNN kernel since it doesn't meet the cuDNN kernel criteria. It will use generic GPU kernel as fallback when running on GPU
WARNING:tensorflow:Layer LSTM_1_backward will not use cuDNN kernel since it doesn't meet the cuDNN kernel criteria. It will use generic GPU kernel as fallback when running on GPU
WARNING:tensorflow:Layer LSTM_0 will not use cuDNN kernel since it doesn't meet the cuDNN kernel criteria. It will use generic GPU kernel as fallback when running on GPU
WARNING:tensorflow:Layer LSTM_0 will not use cuDNN kernel since it doesn't meet the cuDNN kernel criteria. It will use generic GPU kernel as fallback when running on GPU
WARNING:tensorflow:Layer LSTM_0 will not use cuDNN kernel since it doesn't meet the cuDNN kernel criteria. It will use generic GPU kernel as fallback when running on GPU
RNN Morph Tagger Humor Accuracy: 62.827483934750376 %
0.6
```

Вопрос 7:

- Какое качество вы получили, используя каждую из двух библиотек? Сравните их результаты.
- Качество с библиотекой rnnmorph должно быть хуже, так как там используется немного другая система тэгов. Какие здесь отличия?

```
print("NLTK Tagger Humor Accuracy:", nltk_tagger_humor_accuracy * 100, '%')
print("RNN Morph Tagger Humor Accuracy:", rnn_morph_tagger_humor_accuracy * 100, '%')
```

```
NLTK Tagger Humor Accuracy: 89.22392486406328 %
RNN Morph Tagger Humor Accuracy: 62.827483934750376 %
```

Посмотрим, какие теги использует библиотека RNNMorph

```
predictor.model.grammeme_vectorizer_output.all_grammemes['POS']
```

```
{'ADJ',
 'ADP',
 'ADV',
 'AUX',
 'CCONJ',
 'DET',
 'INTJ',
 'NOUN',
 'NUM',
 'PART',
 'PRON',
 'PROPN',
 'PUNCT',
 'SCONJ',
 'SYM',
 'Unknown',
 'VERB',
 'X'}
```

Посмотрим, какие теги в нашем датасете

```
tag_num
```

```
NOUN    275558
VERB    182750
.       147565
ADP     144766
DET     137019
ADJ      83721
ADV      56239
PRON     49334
CONJ     38151
PRT      29829
NUM      14874
X         1386
dtype: int64
```

Видим, что в нашем датасете используется подмножество тегов из RNN Morph. Из-за этого и низкий accuracy

BiLSTMTagger

Подготовка данных

Изменим структуру данных

```
pos_data = [list(zip(*sent)) for sent in brown_tagged_sents]
print(pos_data[0])
```

```
[('It', 'was', 'among', 'these', 'that', 'Hinkle', 'identified', 'a', 'photograph', 'of', 'Barco', '!', '!'), ('PRON', 'VERB', 'ADP', 'DET', 'ADP', 'NOUN', 'VERB', 'DET', 'NOUN', 'ADP', 'NOUN', '.', '.')]
```

До этого мы писали много кода сами, теперь пора эксплуатировать pytorch

```
from torchtext.legacy.data import Field, BucketIterator
import torchtext

# наши поля
WORD = Field(lower=True)
TAG = Field(unk_token=None) # все токены нам извсетны

# создаем примеры
examples = []
for words, tags in pos_data:
    examples.append(torchtext.legacy.data.Example.fromlist([list(words), list(tags)], fields=[('words', WORD), ('tags', TAG)]))
```

Вот один наш пример:

```
print(vars(examples[0]))
```

```
{'words': ['it', 'was', 'among', 'these', 'that', 'hinkle', 'identified', 'a', 'photograph', 'of', 'barco', '!', '!'], 'tags': ['PRON', 'VERB', 'ADP', 'DET', 'ADP', 'NOUN', 'VERB', 'DET', 'NOUN', 'ADP', 'NOUN', '.', '.']}
```

Теперь формируем наш датасет

```
# кладем примеры в наш датасет
dataset = torchtext.legacy.data.Dataset(examples, fields=[('words', WORD), ('tags', TAG)])

train_data, valid_data, test_data = dataset.split(split_ratio=[0.8, 0.1, 0.1])

print(f"Number of training examples: {len(train_data.examples)}")
print(f"Number of validation examples: {len(valid_data.examples)}")
print(f"Number of testing examples: {len(test_data.examples)}")
```

```
Number of training examples: 842
Number of validation examples: 106
Number of testing examples: 105
```

Построим словари. Параметр `min_freq` выберете сами. При построении словаря используем только **train**

```
WORD.build_vocab(train_data, min_freq=0.1)
TAG.build_vocab(train_data)

print(f"Unique tokens in source (ru) vocabulary: {len(WORD.vocab)}")
print(f"Unique tokens in target (en) vocabulary: {len(TAG.vocab)}")

print(WORD.vocab.itos[::200])
print(TAG.vocab.itos)
```

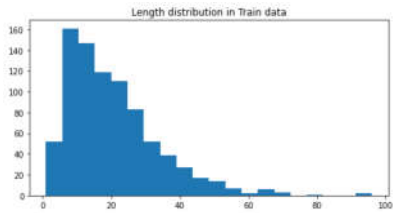
```
Unique tokens in source (ru) vocabulary: 3983
Unique tokens in target (en) vocabulary: 13
['<unk>', 'woman', 'de', 'lap', 'carefully', 'laid', 'socially', 'anybody', 'buckling', 'confessing', 'disturb', 'finished', 'hex', 'ladle', 'mistaken', 'pebbles', 'rapture', 'scattered', 'staunch', 'unbearable']
['<pad>', 'NOUN', 'VERB', '.', 'DET', 'ADP', 'ADJ', 'PRON', 'ADV', 'CONJ', 'PRT', 'NUM', 'X']
```

```
print(vars(train_data.examples[9]))
```

```
{'words': ['and', 'so', 'they', 'were', 'consistently', 'true', 'to', 'their', 'principles', '.'], 'tags': ['CONJ', 'ADP', 'PRON', 'VERB', 'ADV', 'ADJ', 'ADP', 'DET', 'NOUN', '.']}
```

Посмотрим с насколько большими предложениями мы имеем дело

```
length = map(len, [vars(x)['words'] for x in train_data.examples])

plt.figure(figsize=[8, 4])
plt.title("Length distribution in Train data")
plt.hist(list(length), bins=20);
```




```
max_sent_length = max([len(x.words) for x in train_data.examples])
max_sent_length
```

```
96
```

Для обучения `BiLSTM` лучше использовать colab

```
import torch
from torch import nn
```

```
import torch.nn.functional as F
import torch.optim as optim

device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
device
```

```
device(type='cuda')
```

Для более быстрого и устойчивого обучения сгруппируем наши данные по батчам

```
# Бьем нашу выборку на батч, не забывая сначала отсортировать выборку по длине
def _len_sort_key(x):
    return len(x.words)

BATCH_SIZE = 32

train_iterator, valid_iterator, test_iterator = BucketIterator.splits(
    (train_data, valid_data, test_data),
    batch_size = BATCH_SIZE,
    device = device,
    sort_key=_len_sort_key
)
```

```
# посморим  на количество батчей
list(map(len, [train_iterator, valid_iterator, test_iterator]))
```

```
[27, 4, 4]
```

Модель и её обучение

Инициализируем нашу модель

```
class LSTMTagger(nn.Module):

    def __init__(self, input_dim, emb_dim, hid_dim, output_dim, dropout, bidirectional=False):
        super().__init__()

        self.embeddings = nn.Embedding(input_dim, emb_dim)
        self.dropout = nn.Dropout(dropout)

        self.rnn = nn.LSTM(emb_dim, hid_dim, bidirectional=bidirectional)
        # если bidirectional, то предсказываем на основе конкатенации двух hidden
        self.tag = nn.Linear((1 + bidirectional) * hid_dim, output_dim)

    def forward(self, sent):
        #sent = [sent len, batch size]

        # не забываем применить dropout к embedding
        embedded = self.embeddings(sent)
        embedded = self.dropout(embedded)

        output, _ = self.rnn(embedded)
        #output = [sent len, batch size, hid dim * n directions]

        prediction = self.tag(output)

        return prediction

# параметры модели
INPUT_DIM = len(WORD.vocab)
OUTPUT_DIM = len(TAG.vocab)
EMB_DIM = 300
HID_DIM = 50
DROPOUT = 0.5
BIDIRECTIONAL = True

model = LSTMTagger(INPUT_DIM, EMB_DIM, HID_DIM, OUTPUT_DIM, DROPOUT, BIDIRECTIONAL).to(device)

# инициализируем веса
def init_weights(m):
    for name, param in m.named_parameters():
        nn.init.uniform_(param, -0.08, 0.08)

model.apply(init_weights)
```

```
LSTMTagger(
  (embeddings): Embedding(3983, 300)
  (dropout): Dropout(p=0.5, inplace=False)
  (rnn): LSTM(300, 50, bidirectional=True)
  (tag): Linear(in_features=100, out_features=13, bias=True)
)
```

Подсчитаем количество обучаемых параметров нашей модели

```
def count_parameters(model):
    return sum([len(param) for param in model.parameters()])

print(f'The model has {count_parameters(model):,} trainable parameters')
```

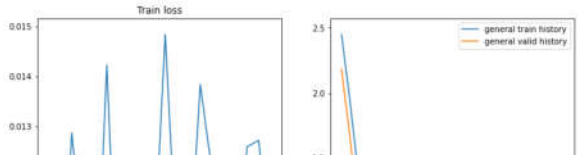
```
The model has 5,609 trainable parameters
```

Погнали обучать

```
PAD_IDX = TAG.vocab.stoi['<pad>']
optimizer = optim.Adam(model.parameters())
criterion = nn.CrossEntropyLoss(ignore_index = PAD_IDX)

def train(model, iterator, optimizer, criterion, clip, train_history=None, valid_history=None):
    model.train()

    epoch_loss = 0
    history = []
    for i, batch in enumerate(iterator):
        words = batch.words
        tags = batch.tags

        # words = pad_tensor(words, max_sent_length, BATCH_SIZE).to(device)
        # tags = pad_tensor(tags, max_sent_length, BATCH_SIZE).to(device)

        optimizer.zero_grad()

        output = model(words)

        #tags = [sent len, batch size]
        #output = [sent len, batch size, output dim]

        tags = tags.view(-1)
        output = output.view(len(tags), -1)

        #tags = [sent len * batch size]
        #output = [sent len * batch size, output dim]
```

```
        loss = criterion(output, tags)

        loss.backward()

        # Gradient clipping(решение проблемы взрыва граденты), clip - максимальная норма вектора
        torch.nn.utils.clip_grad_norm_(model.parameters(), max_norm=clip)

        optimizer.step()

        epoch_loss += loss.item()

        history.append(loss.cpu().data.numpy())
        if (i+1)%10==0:
            fig, ax = plt.subplots(nrows=1, ncols=2, figsize=(12, 8))

            clear_output(True)
            ax[0].plot(history, label='train loss')
            ax[0].set_xlabel('Batch')
            ax[0].set_title('Train loss')

            if train_history is not None:
                ax[1].plot(train_history, label='general train history')
                ax[1].set_xlabel('Epoch')
            if valid_history is not None:
                ax[1].plot(valid_history, label='general valid history')
            plt.legend()

            plt.show()


    return epoch_loss / len(iterator)

def evaluate(model, iterator, criterion):
    model.eval()

    epoch_loss = 0

    history = []

    with torch.no_grad():

        for i, batch in enumerate(iterator):
            words = batch.words
            tags = batch.tags

            # words = pad_tensor(words, max_sent_length, BATCH_SIZE).to(device)
            # tags = pad_tensor(tags, max_sent_length, BATCH_SIZE).to(device)

            output = model(words)

            #tags = [sent len, batch size]
            #output = [sent len, batch size, output dim]

            tags = tags.view(-1)
            output = output.view(len(tags), -1)

            #tags = [sent len * batch size]
            #output = [sent len * batch size, output dim]

            loss = criterion(output, tags)

            epoch_loss += loss.item()

    return epoch_loss / len(iterator)

def epoch_time(start_time, end_time):
    elapsed_time = end_time - start_time
    elapsed_mins = int(elapsed_time / 60)
    elapsed_secs = int(elapsed_time - (elapsed_mins * 60))
    return elapsed_mins, elapsed_secs
```

```
import time
import math
import matplotlib
matplotlib.rcParams.update({'figure.figsize': (16, 12), 'font.size': 14})
import matplotlib.pyplot as plt
%matplotlib inline
from IPython.display import clear_output

train_history = []
valid_history = []

N_EPOCHS = 30
CLIP = 2

best_valid_loss = float('inf')

for epoch in range(N_EPOCHS):

    start_time = time.time()

    train_loss = train(model, train_iterator, optimizer, criterion, CLIP, train_history, valid_history)
    valid_loss = evaluate(model, valid_iterator, criterion)

    end_time = time.time()

    epoch_mins, epoch_secs = epoch_time(start_time, end_time)

    if valid_loss < best_valid_loss:
        best_valid_loss = valid_loss
        torch.save(model.state_dict(), 'best-val-model.pt')

    train_history.append(train_loss)
    valid_history.append(valid_loss)
    print(f'Epoch: {epoch+1:02} | Time: {epoch_mins}m {epoch_secs}s')
    print(f'\tTrain Loss: {train_loss:.3f} | Train PPL: {math.exp(train_loss):7.3f}')
    print(f'\t Val. Loss: {valid_loss:.3f} |  Val. PPL: {math.exp(valid_loss):7.3f}')
```



Применение модели

```
def accuracy_model(model, iterator):
    model.eval()

    true_pred = 0
    num_pred = 0

    with torch.no_grad():
        for i, batch in enumerate(iterator):
            words = batch.words
            tags = batch.tags

            output = model(words)

            #output = [sent len, batch size, output dim]

            output = torch.argmax(output, dim=2)

            #output = [sent len, batch size]
            predict_tags = output.cpu().numpy()
            true_tags = tags.cpu().numpy()

            true_pred += np.sum((true_tags == predict_tags) & (true_tags != PAD_IDX))
            num_pred += np.prod(true_tags.shape) - (true_tags == PAD_IDX).sum()

    return true_pred / num_pred
```

```
lstm_tagger_humor_accuracy = accuracy_model(model, test_iterator)
print("LSTM Tagger Humor Accuracy:", lstm_tagger_humor_accuracy * 100, '%')
```

```
LSTM Tagger Humor Accuracy: 90.32674118658642 %
```

Вы можете улучшить качество, изменяя параметры модели. Но чтобы добиться нужного качества, вам неообходимо взять все выборку, а не только категорию `humor`.

```
brown_tagged_sents = brown.tagged_sents(tagset="universal")
```

Вам неоходимо добиться качества не меньше, чем `accuracy = 93 %`

Создадим и обучим HMM на всем корпусе

```
# Приведем слова к нижнему регистру
my_brown_tagged_sents = []
for sent in brown_tagged_sents:
    my_brown_tagged_sents.append(list(map(lambda x: (x[0].lower(), x[1]), sent)))
my_brown_tagged_sents = np.array(my_brown_tagged_sents)

from sklearn.model_selection import train_test_split
train_sents, test_sents = train_test_split(my_brown_tagged_sents, test_size=0.1, random_state=0)

hmm = HiddenMarkovModel()
hmm.fit(train_sents)

hmm_tagger_full_accuracy = accuracy_score(hmm, test_sents)
print("HMM Tagger Full Accuracy:", hmm_tagger_full_accuracy * 100, '%')
```

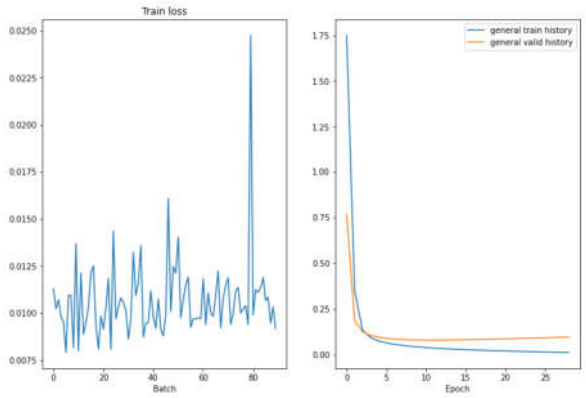
```
/usr/local/lib/python3.7/dist-packages/ipykernel_launcher.py:5: VisibleDeprecationWarning: Creating an ndarray from ragged nested sequences (which is a list-or-tuple of lists-or-tuples-or ndarrays with different lengths or shapes) i
  """
HMM Tagger Full Accuracy: 96.26295331104619 %
```

Создадим и обучим LSTMTagger на всем корпусе

```
pos_data = [list(zip(*sent)) for sent in brown_tagged_sents]

examples = []
for words, tags in pos_data:
    examples.append(torchtext.legacy.data.Example.fromlist([list(words), list(tags)], fields=[('words', WORD), ('tags', TAG)]))

dataset = torchtext.legacy.data.Dataset(examples, fields=[('words', WORD), ('tags', TAG)])

train_data, valid_data, test_data = dataset.split(split_ratio=[0.8, 0.1, 0.1])

WORD.build_vocab(train_data, min_freq=0.1)
TAG.build_vocab(train_data)

BATCH_SIZE = 512

train_iterator, valid_iterator, test_iterator = BucketIterator.splits(
    (train_data, valid_data, test_data),
    batch_size = BATCH_SIZE,
    device = device,
    sort_key=_len_sort_key
)
```

```
INPUT_DIM = len(WORD.vocab)
OUTPUT_DIM = len(TAG.vocab)
EMB_DIM = 300
HID_DIM = 50
DROPOUT = 0.5
BIDIRECTIONAL = True

model = LSTMTagger(INPUT_DIM, EMB_DIM, HID_DIM, OUTPUT_DIM, DROPOUT, BIDIRECTIONAL).to(device)
model.apply(init_weights)

PAD_IDX = TAG.vocab.stoi['<pad>']
optimizer = optim.Adam(model.parameters())
criterion = nn.CrossEntropyLoss(ignore_index = PAD_IDX)

train_history = []
valid_history = []

N_EPOCHS = 30
CLIP = 2

best_valid_loss = float('inf')

for epoch in range(N_EPOCHS):
    start_time = time.time()
```

```
    start_time = time.time()

    train_loss = train(model, train_iterator, optimizer, criterion, CLIP, train_history, valid_history)
    valid_loss = evaluate(model, valid_iterator, criterion)

    end_time = time.time()

    epoch_mins, epoch_secs = epoch_time(start_time, end_time)

    if valid_loss < best_valid_loss:
        best_valid_loss = valid_loss
        torch.save(model.state_dict(), 'best-val-model.pt')

    train_history.append(train_loss)
    valid_history.append(valid_loss)
    print(f'Epoch: {epoch+1:02} | Time: {epoch_mins}m {epoch_secs}s')
    print(f'\tTrain Loss: {train_loss:.3f} | Train PPL: {math.exp(train_loss):7.3f}')
    print(f'\t Val. Loss: {valid_loss:.3f} |  Val. PPL: {math.exp(valid_loss):7.3f}')
```




```
Epoch: 30 | Time: 0m 5s
        Train Loss: 0.011 | Train PPL:   1.011
         Val. Loss: 0.096 |  Val. PPL:   1.101
```

```
best_model = LSTMTagger(INPUT_DIM, EMB_DIM, HID_DIM, OUTPUT_DIM, DROPOUT, BIDIRECTIONAL).to(device)
best_model.load_state_dict(torch.load('best-val-model.pt'))
lstm_tagger_full_accuracy = accuracy_model(best_model, test_iterator)
print("LSTM Tagger Full Accuracy:", lstm_tagger_full_accuracy * 100, '%')
assert lstm_tagger_full_accuracy * 100 >= 93
```

```
LSTM Tagger Full Accuracy: 97.62193210603455 %
```

Пример решение нашей задачи:

```
def print_tags(model, data):
    model.eval()

    with torch.no_grad():
        words, _ = data
        example = torch.LongTensor([WORD.vocab.stoi[elem] for elem in words]).unsqueeze(1).to(device)

        output = model(example).argmax(dim=-1).cpu().numpy()
        tags = [TAG.vocab.itos[int(elem)] for elem in output]

        for token, tag in zip(words, tags):
            print(f'{token:15s}{tag}')
```

```
print_tags(model, pos_data[-1])
```

```
From            NOUN
what            DET
I               NOUN
was             VERB
able            ADJ
to              ADP
gauge           NOUN
in              ADP
a               DET
swift           ADJ
,               .
greedy          ADJ
glance          NOUN
,               .
the             DET
figure          NOUN
inside          ADP
the             DET
coral-colored   ADJ
boucle          NOUN
dress           NOUN
was             VERB
stupefying      VERB
.               .
```

- Сравните результаты моделей HiddenMarkov, LstmTagger:
  - при обучение на маленькой части корпуса, например, на категории humor
  - при обучении на всем корпусе

```
print("HMM Tagger Humor Accuracy:", hmm_tagger_humor_accuracy * 100, '%')
print("Default Tagger Humor Accuracy:", default_tagger_humor_accuracy * 100, '%')
print("NLTK Tagger Humor Accuracy:", nltk_tagger_humor_accuracy * 100, '%')
print("RNN Morph Tagger Accuracy:", rnn_morph_tagger_humor_accuracy * 100, '%')
print("LSTM Tagger Humor Accuracy:", lstm_tagger_humor_accuracy * 100, '%')

print("HMM Tagger Full Accuracy:", hmm_tagger_full_accuracy * 100, '%')
print("LSTM Tagger Full Accuracy:", lstm_tagger_full_accuracy * 100, '%')
```

```
HMM Tagger Humor Accuracy: 88.82847256549678 %
Default Tagger Humor Accuracy: 20.2174987642115 66 %
NLTK Tagger Humor Accuracy: 89.22392486406328 %
RNN Morph Tagger Accuracy: 62.827483934750376 %
LSTM Tagger Humor Accuracy: 90.32674118658642 %
HMM Tagger Full Accuracy: 96.26295331104619 %
LSTM Tagger Full Accuracy: 97.62193210603455 %
```

Выводы

- LSTM Tagger лучше на малой части корпуса (категория humor) чем HMM Tagger
- LSTM Tagger лучше на всем корпусе чем HMM Tagger