Deep Learning School

Физтех-Школа Прикладной математики и информатики (ФПМИ) МФТИ

Embeddings

Привет! В этом домашнем задании мы с помощью эмбеддингов решим задачу семантической классификации твитов.

Для этого мы воспользуемся предобученными эмбеддингами word2vec.

Для начала скачаем датасет для семантической классификации твитов:

```
!gdown https://drive.google.com/uc?id=1eE1FiUkXkcbw0McId4i7qY-L8hH-_Qph&export=download
!unzip archive.zip
```

Импортируем нужные библиотеки:

```
import math
```



```
import numpy as np
import pandas as pd
import seaborn as sns

import torch
import torch.nn as nn
import nltk
import gensim
import gensim.downloader as api

%matplotlib inline


import warnings
warnings.filterwarnings("ignore")


random.seed(42)
np.random.seed(42)
```

```
torch.random.manual_seed(42)
torch.cuda.random.manual_seed(42)
torch.cuda.random.manual_seed_all(42)

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

```
device
```

```
'cpu'
```

```
data = pd.read_csv("training.1600000.processed.noemoticon.csv", encoding="latin", header=None, names=['
```

Посмотрим на данные:

```
data.head()
```

	emotion	id	date	flag	user	text
0	0	1467810369	Mon Apr 06 22:19:45 PDT 2009	NO_QUERY	_TheSpecialOne_	@switchfoot http://twitpic.com/2y1zl - Awww, t...
1	0	1467810672	Mon Apr 06 22:19:49 PDT 2009	NO_QUERY	scotthamilton	is upset that he can't update his Facebook by ...
2	0	1467810917	Mon Apr 06 22:19:53 PDT 2009	NO_QUERY	mattycus	@Kenichan I dived many times for the ball. Man...

Выведем несколько примеров твитов, чтобы понимать, с чем мы имеем дело:

```
examples = data["text"].sample(10)
print("\n".join(examples))
```

```
@chrishasboobs AHHH I HOPE YOUR OK!!!
@misstoriblack cool , i have no tweet apps  for my razr 2
@TiannaChaos i know  just family drama. its lame.hey next time u hang out with kim n u guys like
School email won't open  and I have geography stuff on there to revise! *Stupid School* :'(
upper airways problem
Going to miss Paston's sermon on Faith...
                    with me
                    eeling like that?
gahh noo!peyton needs to live!this is horrible
@mrstessyman thank you glad you like it! There is a product review bit on the site  Enjoy knittin
```

Сохранено

Как видим, тексты твитов очень "грязные". Нужно предобработать датасет, прежде чем строить для него модель классификации.

Чтобы сравнивать различные методы обработки текста/модели/прочее, разделим датасет на dev(для обучения модели) и test(для получения качества модели).

```
indexes = np.arange(data.shape[0])
np.random.shuffle(indexes)
dev_size = math.ceil(data.shape[0] * 0.8) # 80% данных для обучения модели

dev_indexes = indexes[:dev_size]
test_indexes = indexes[dev_size:]

dev_data = data.iloc[dev_indexes]
```

```
test_data = data.iloc[test_indexes]

dev_data.reset_index(drop=True, inplace=True)
test_data.reset_index(drop=True, inplace=True)
```

Обработка текста

Токенизируем текст, избавимся от знаков пунктуации и выкинем все слова, состоящие менее чем из 4 букв:

```
tokenizer = nltk.WordPunctTokenizer()
line = tokenizer.tokenize(dev_data["text"][0].lower())
print(" ".join(line))
```

```
@ claire_nelson i ' m on the north devon coast the next few weeks will be down in devon again in
```

```
line
```

```
['@',
 'claire_nelson',
 'i',
 "'",
 'm',
 'on',
 'the',
 'north',
 'devon',
 'coast',
 'the',
 'next',
 'few',
 'weeks',
 'will',
 'be',
 'down',
 'in',
 'devon',
 'again',
 'sometime',
 'i',
 'hope',
 'though',
 '!']
```



```
filtered_line = [w for w in line if all(c not in string.punctuation for c in w) and len(w) > 3]
print(" ".join(filtered_line))
```

```
north devon coast next weeks will down devon again sometime hope though
```

```
filtered_line
```

```
['north',
 'devon',
 'coast',
 'next',
 'weeks',
 'will',
```

```
 'down',
 'devon',
 'again',
 'sometime',
 'hope',
 'though']
```

Загрузим предобученную модель эмбеддингов.

Если хотите, можно попробовать другую. Полный список можно найти здесь: https://github.com/RaRe-Technologies/gensim-data.

Данная модель выдает эмбеддинги для **слов**. Строить по эмбеддингам слов эмбеддинги предложений мы будем ниже.

```
word2vec = api.load("word2vec-google-news-300")
```

```
emb_line = [word2vec.get_vector(w) for w in filtered_line if w in word2vec]
print(sum(emb_line).shape)
```

```
(300,)
```

```
len(emb_line), emb_line[0].shape
```

```
(12, (300,))
```

Нормализуем эмбеддинги, прежде чем обучать на них сеть. (наверное, вы помните, что нейронные сети гораздо лучше обучаются на нормализованных данных)

```
mean = np.mean(word2vec.vectors, 0)
std = np.std(word2vec.vectors, 0)
norm_emb_line = [(word2vec.get_vector(w) - mean) / std for w in filtered_line if w in word2vec and len(
print(sum(norm_emb_line).shape)
print([all(norm_emb_line[i] == emb_line[i]) for i in range(len(emb_line))])
```

```
(300,)
[False, False, False, False, False, False, False, False, False, False, False, False]
```

Сохранено

```
].shape
```

```
(12, (300,))
```

Сделаем датасет, который будет по запросу возвращать подготовленные данные.

```
from torch.utils.data import Dataset, random_split
```

```
class TwitterDataset(Dataset):
    def __init__(self, data: pd.DataFrame, feature_column: str, target_column: str, word2vec: gensim.mo
        self.tokenizer = nltk.WordPunctTokenizer()

        self.data = data

        self.feature_column = feature_column
        self.target_column = target_column

        self.word2vec = word2vec
```

```
        self.label2num = lambda label: 0 if label == 0 else 1
        self.mean = np.mean(word2vec.vectors, axis=0)
        self.std = np.std(word2vec.vectors, axis=0)

    def __getitem__(self, item):
        text = self.data[self.feature_column][item]
        label = self.label2num(self.data[self.target_column][item])

        tokens = self.get_tokens_(text)
        embeddings = self.get_embeddings_(tokens)

        return {"feature": embeddings, "target": label}

    def get_tokens_(self, text):
        # Получи все токены из текста и профильтруй их
        return [word for word in self.tokenizer.tokenize(text.lower()) if all(c not in string.punctuati

    def get_embeddings_(self, tokens):
        # Получи эмбеддинги слов и усредни их
        embeddings = [(self.word2vec.get_vector(word) - self.mean) / self.std for word in tokens if wor

        if len(embeddings) == 0:
            embeddings = np.zeros((1, self.word2vec.vector_size))
        else:
            embeddings = np.array(embeddings)
            if len(embeddings.shape) == 1:
                embeddings = embeddings.reshape(-1, 1)

        return embeddings

    def __len__(self):
        return self.data.shape[0]


dev = TwitterDataset(dev_data, "text", "emotion", word2vec)
```

Отлично, мы готовы с помощью эмбеддингов слов превращать твиты в векторы и обучать нейронную сеть.

Сохранено

льзуя эмбеддинги слов, можно несколькими способами. А именно

Average embedding (2 балла)

Это самый простой вариант, как получить вектор предложения, используя векторные представления слов в предложении. А именно: вектор предложения есть средний вектор всех слов в предложении (которые остались после токенизации и удаления коротких слов, конечно).

```
indexes = np.arange(len(dev))
np.random.shuffle(indexes)
example_indexes = indexes[::1000]

examples = {"features": [np.mean(dev[i]["feature"], axis=0) for i in example_indexes],
            "targets": [dev[i]["target"] for i in example_indexes]}
print(len(examples["features"]))
```

```
1280
```

```
# Посмотрим на то, что из себя представляют "features"
```

```
examples["features"][0].shape
```

```
(300,)
```

```
examples["features"][0]
```

```
array([ 1.8990873e-01,  1.1539176e-02,  3.6923146e-01,  2.0414203e-01,
        6.5201277e-01,  1.2293394e+00,  1.6557020e-01, -6.7087388e-01,
        3.5630524e-01,  1.2866460e-01,  3.0959654e-01,  1.8795593e-01,
       -9.9852806e-01, -1.6869114e-01, -2.8444505e-01,  8.4852129e-01,
       -1.9497351e-01,  3.2395500e-01,  8.6606628e-01,  9.5698464e-01,
        8.1324840e-01, -1.2951770e+00, -4.4475859e-01, -6.8922269e-01,
       -5.9648907e-01,  1.2602067e-01,  2.9267946e-01,  5.4685616e-01,
       -1.4842993e-01, -4.2440183e-02,  4.5046088e-01,  4.6453416e-01,
       -6.4742941e-01, -3.6611453e-01, -3.2342875e-01, -3.0353677e-01,
        3.1811464e-01, -9.9456012e-01, -9.1858707e-02, -1.9013677e-02,
       -7.7859753e-01,  5.2730364e-01, -5.2146800e-02,  3.6187701e-02,
        3.3385864e-01, -1.7134695e-01, -2.2494353e-01,  4.5342457e-01,
       -2.9525690e-02,  2.9464537e-02,  6.7867488e-01,  3.9340010e-01,
       -5.6723422e-01, -2.8645679e-01,  5.1626396e-01, -1.1997807e+00,
       -5.7937104e-01, -9.0618533e-01, -3.8439956e-01, -3.8538319e-01,
       -2.5370520e-01,  2.6081902e-01,  5.4961526e-01, -9.3035561e-01,
       -6.1073877e-02,  5.6741673e-01,  3.0133346e-01,  3.8573962e-02,
        2.2656364e-02, -7.9435110e-01,  4.5492244e-01, -2.0452160e-01,
        3.7451667e-01,  5.0119007e-01, -3.2384625e-01, -4.1671798e-01,
        5.3644663e-01, -1.6257130e-01, -8.0588043e-01,  5.7113701e-01,
       -5.3272331e-01,  4.9115935e-01, -5.1877224e-01,  7.2114599e-01,
       -6.7661130e-01, -5.7643837e-01, -9.9461091e-01, -2.2027747e-01,
        3.7604101e-02, -2.1922469e-02,  1.1243998e-01, -4.3353909e-01,
        1.4500503e-01,  6.6859289e-03, -9.0164220e-01, -3.3341759e-01,
        2.1511978e-01, -7.2997677e-01, -7.5655019e-01, -5.5833042e-02,
        1.3633148e-01, -7.7846134e-01,  1.3513650e-01, -3.4412330e-01,
        3.6896554e-01, -3.5577673e-01,  2.4607405e-01,  3.1070599e-01,
        2.2796759e-02, -4.5618075e-01,  2.7373698e-01,  3.2289574e-01,
       -2.0345783e-01, -5.0823301e-01, -4.8075110e-01,  7.5829130e-01,
        4.8250294e-01, -8.8868640e-02,  6.0687089e-01, -3.9109387e-02,
        3.5262036e-01,  1.2557191e-01,  7.6976500e-02,  1.6961110e-01,
        6.2182760e-01,  4.7873595e-01, -3.2651791e-01, -5.5655837e-01,
       -6.8316448e-01,  7.5986546e-01,  2.6736873e-01, -1.5351702e-01,
        6.9842130e-02, -1.6256180e-01, -1.1235073e+00,  5.2509403e-01,
                                   130e-01,  2.7941486e-01,  1.2986468e+00,
                                   466e-01, -6.0172416e-02, -1.1743480e-01,
                                   526e-02, -1.2441653e-01,  7.0906371e-01,
       -4.3886051e-02, -3.7981176e-01, -2.7757078e-02,  1.9425710e-01,
        1.6194253e-01, -2.0829470e-01,  5.0358236e-01, -2.0517056e-01,

       -2.7601379e-01, -5.1146090e-01,  5.4255128e-02, -4.6121258e-01,
        6.4728510e-01,  1.6293863e-02,  2.5465378e-01,  2.3581579e-01,
        3.9009801e-01, -3.9274728e-01, -1.0105843e-01, -2.3705889e-01,
       -1.1647777e-01,  4.7990525e-01, -3.3084026e-01,  1.1589920e-02,
        3.0853078e-01, -2.4463683e-01,  6.9103914e-01, -2.6546529e-01,
        1.5228857e+00,  9.4053254e-02,  1.5555365e-01, -1.9884418e-01,
        9.0168849e-02,  7.1009457e-01, -5.7233274e-01,  1.6016565e-01,
        4.5757821e-01,  6.2606287e-01, -1.6246919e-01, -3.2462999e-03,
       -6.7829001e-01, -1.0388204e+00, -4.7292411e-03,  8.6004221e-01,
       -5.3856200e-01, -1.5142804e-01,  2.1046947e-01, -1.3409723e-01,
       -3.0081251e-01,  5.0914492e-02,  2.8817758e-01,  1.0188939e-01,
        1.6027343e+00, -2.0448303e-01, -3.8171703e-01, -7.1548355e-01,
       -2.5301066e-01, -6.9351059e-01, -2.0069674e-02,  6.3181035e-02,
        3.3311489e-01, -1.9342774e-01,  1.0367320e+00, -2.8107890e-01,
       -1.7346799e-01, -3.9737058e-01, -3.2240999e-01,  2.9525450e-01,
        7.0284808e-01, -8.3022624e-02, -6.9759178e-01,  5.7596946e-01,
        4.8580110e-02, -6.0554600e-01,  4.2111240e-02,  3.4606346e-01,
       -1.6900261e-01, -5.1634765e-01, -3.9363796e-01, -1.0428983e+00],
```

Сохранено

```
            dtype=float32)
```

Давайте сделаем визуализацию полученных векторов твитов тренировочного (dev) датасета. Так мы увидим, насколько хорошо твиты с разными target значениями отделяются друг от друга, т.е. насколько хорошо усреднение эмбеддингов слов предложения передает информацию о предложении.

Для визуализации векторов надо получить их проекцию на плоскость. Сделаем это с помощью `PCA`. Если хотите, можете вместо PCA использовать TSNE: так у вас получится более точная проекция на плоскость (а значит, более информативная, т.е. отражающая реальное положение векторов твитов в пространстве). Но TSNE будет работать намного дольше.

```
# 1280*300 -> 1280*2 - преобразование PCA, которого мы добиваемся

from sklearn.decomposition import PCA

pca = PCA(n_components=2)
examples["transformed_features"] = pca.fit_transform(examples["features"]) # Обучи PCA на эмбеддингах с

examples["transformed_features"].shape
```

```
(1280, 2)
```

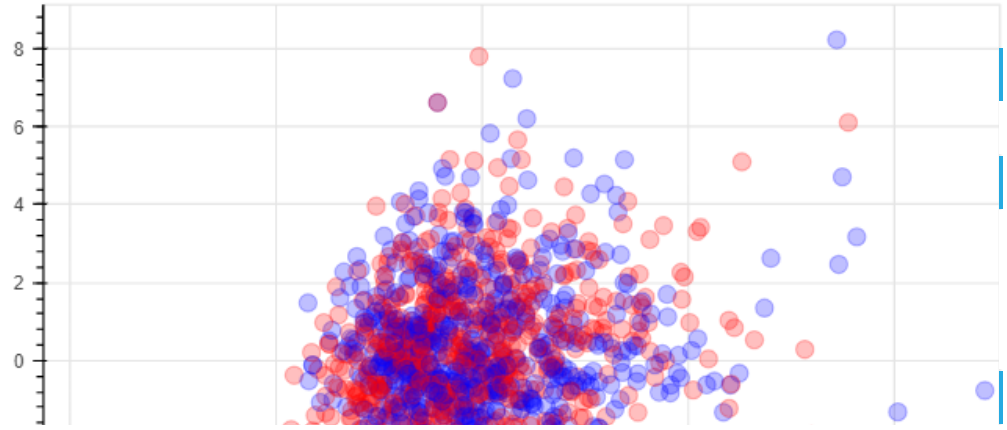
```
import bokeh.models as bm, bokeh.plotting as pl
from bokeh.io import output_notebook
output_notebook()

def draw_vectors(x, y, radius=10, alpha=0.25, color='blue',
                 width=600, height=400, show=True, **kwargs):
    """ draws an interactive plot for data points with auxilirary info on hover """
    data_source = bm.ColumnDataSource({ 'x' : x, 'y' : y, 'color': color, **kwargs })

    fig = pl.figure(active_scroll='wheel_zoom', width=width, height=height)
    fig.scatter('x', 'y', size=radius, color='color', alpha=alpha, source=data_source)

    fig.add_tools(bm.HoverTool(tooltips=[(key, "@" + key) for key in kwargs.keys()]))
    if show: pl.show(fig)
    return fig


draw_vectors(
    examples["transformed_features"][:, 0],
    examples["transformed_features"][:, 1],
    color=[["red", "blue"][t] for t in examples["targets"]]
    )
```



Скорее всего, на визуализации нет четкого разделения твитов между классами. Это значит, что по полученным нами векторам твитов не так-то просто определить, к какому классу твит пренадлежит. Значит, обычный линейный классификатор не очень хорошо справится с задачей. Надо будет делать глубокую (хотя бы два слоя) нейронную сеть.

Подготовим загрузчики данных. Усреднее векторов будем делать в "батчевалке"(`collate_fn`). Она используется для того, чтобы собирать из данных `torch.Tensor` батчи, которые можно отправлять в модель.

```
from torch.utils.data import DataLoader


batch_size = 1024
num_workers = 4

def average_emb(batch):
    features = [np.mean(b["feature"], axis=0) for b in batch]
    targets = [b["target"] for b in batch]

    return {"features": torch.FloatTensor(features), "targets": torch.LongTensor(targets)}


train_size = math.ceil(len(dev) * 0.8)

train, valid = random_split(dev, [train_size, len(dev) - train_size])

train_loader = DataLoader(train, batch_size=batch_size, num_workers=num_workers, shuffle=True, drop_las
valid_loader = DataLoader(valid, batch_size=batch_size, num_workers=num_workers, shuffle=False, drop_la
```

Сохранено

Определим функции для тренировки и теста модели:

```
from tqdm.notebook import tqdm


def training(model, optimizer, criterion, train_loader, epoch, device="cpu"):
    pbar = tqdm(train_loader, desc=f"Epoch {e + 1}. Train Loss: {0}")
    model.train()

    for batch in pbar:
        features = batch["features"].to(device)
        targets = batch["targets"].to(device)

        # Получи предсказания модели
        # Посчитай лосс
        # Обнови параметры модели
```

```
            optimizer.zero_grad()

            logits = model(features)
            loss = criterion(logits, targets)
            loss.backward()
            optimizer.step()

            pbar.set_description(f"Epoch {e + 1}. Train Loss: {loss:.4}")


    def testing(model, criterion, test_loader, device="cpu"):
        pbar = tqdm(test_loader, desc=f"Test Loss: {0}, Test Acc: {0}")

        mean_loss = 0
        mean_acc = 0

        model.eval()
        with torch.no_grad():

            for batch in pbar:
                features = batch["features"].to(device)
                targets = batch["targets"].to(device)

                # Получи предсказания модели
                # Посчитай лосс
                # Посчитай точность модели

                logits = model(features)
                loss = criterion(logits, targets)
                acc = (logits.argmax(dim=1) == targets).type(torch.float).mean()

                mean_loss += loss.item()
                mean_acc += acc.item()

                pbar.set_description(f"Test Loss: {loss:.4}, Test Acc: {acc:.4}")

        pbar.set_description(f"Test Loss: {mean_loss / len(test_loader):.4}, Test Acc: {mean_acc / len(test

        return {"Test Loss": mean_loss / len(test_loader), "Test Acc": mean_acc / len(test_loader)}
```

елевую функцию. Вы можете сами выбрать количество слоев в
имизатор и целевую функцию.


```
    class NeuralNetwork(nn.Module):
        def __init__(self, embeddings_dim, n_classes):
            super(NeuralNetwork, self).__init__()
            self.linear_1 = nn.Linear(embeddings_dim, 256)
            self.relu = nn.ReLU()
            self.linear_2 = nn.Linear(256, 128)
            self.linear_3 = nn.Linear(128, n_classes)

        def forward(self, x):
            x = self.relu(self.linear_1(x))
            x = self.relu(self.linear_2(x))

            logits = self.linear_3(x)
            return logits


    import torch.nn as nn
    from torch.optim import Adam
```

```
# Не забудь поиграться с параметрами ;)
vector_size = dev.word2vec.vector_size
num_classes = 2
lr = 1e-4
num_epochs = 2

model = NeuralNetwork(embeddings_dim=vector_size, n_classes=num_classes) # Твоя модель
model = model.to(device)
criterion = nn.CrossEntropyLoss() # Твой лосс
optimizer = torch.optim.Adam(model.parameters()) # Твой оптимайзер
```

Наконец, обучим модель и протестируем её.

После каждой эпохи будем проверять качество модели на валидационной части датасета. Если метрика стала лучше, будем сохранять модель. **Подумайте, какая метрика (точность или лосс) будет лучше работать в этой задаче?**

```
best_metric = np.inf
for e in range(num_epochs):
    training(model, optimizer, criterion, train_loader, e, device)
    log = testing(model, criterion, valid_loader, device)
    print(log)
    if log["Test Loss"] < best_metric:
        torch.save(model.state_dict(), "model.pt")
        best_metric = log["Test Loss"]
```

```
Epoch 1. Train Loss: 0.4959: 100% 1000/1000 [02:28<00:00, 8.19it/s]
Test Loss: 0.4839, Test Acc: 0.7578: 100% 250/250 [00:37<00:00, 6.05it/s]
{'Test Loss': 0.5037948249578476, 'Test Acc': 0.75125390625}
Epoch 2. Train Loss: 0.4811: 100% 1000/1000 [02:34<00:00, 9.29it/s]
Test Loss: 0.4664, Test Acc: 0.7607: 100% 250/250 [00:38<00:00, 9.16it/s]
{'Test Loss': 0.49697556138038634, 'Test Acc': 0.755859375}
```

```
test_loader = DataLoader(
                          ", "emotion", word2vec),
    num_workers=num_workers,
    shuffle=False,
    drop_last=False,
    collate_fn=average_emb)

model.load_state_dict(torch.load("model.pt", map_location=device))

print(testing(model, criterion, test_loader, device=device))
```

```
Test Loss: 0.5302, Test Acc: 0.75: 100% 313/313 [00:46<00:00, 8.28it/s]
{'Test Loss': 0.49849567026756825, 'Test Acc': 0.7552447334265175}
```


Embeddings for unknown words (8 баллов)

Пока что использовалась не вся информация из текста. Часть информации фильтровалось – если слова не было в словаре эмбеддингов, то мы просто превращали слово в нулевой вектор(МОЙ

КОММЕНТАРИЙ: не согласен... мы не превращали слово в нулевой вектор, а пропускали его). Хочется использовать информацию по-максимуму. Поэтому рассмотрим другие способы обработки слов, которых нет в словаре. А именно:

- Для каждого незнакомого слова будем запоминать его контекст(слова слева и справа от этого слова). Эмбеддингом нашего незнакомого слова будет сумма эмбеддингов всех слов из его контекста. (4 балла)
- Для каждого слова текста получим его эмбеддинг из Tfidf с помощью `TfidfVectorizer` из sklearn. Итоговым эмбеддингом для каждого слова будет сумма двух эмбеддингов: предобученного и Tfidf-ного. Для слов, которых нет в словаре предобученных эмбеддингов, результирующий эмбеддинг будет просто полученный из Tfidf. (4 балла)

Реализуйте оба варианта **ниже**. Напишите, какой способ сработал лучше и ваши мысли, почему так получилось.

```
# 1-ый вариант
# Для реализации 1-ого варианта переопредлим метод get_embeddings_ класса TwitterDataset


class Custom_TwitterDataset_v1(TwitterDataset):
    def __init__(self, data: pd.DataFrame, feature_column: str, target_column: str, word2vec: gensim.mo
        super(Custom_TwitterDataset_v1, self).__init__(data, feature_column, target_column, word2vec)
        self.window_size = window_size

    def get_embeddings_(self, tokens):
        embeddings = []

        for i, word in enumerate(tokens):
            if word not in self.word2vec: # если слова нет среди предобученных эмбеддингов, расчитываем
                left_context = max(0, i - self.window_size)
                right_context = min(len(tokens) - 1, i + self.window_size)

                context = tokens[left_context: i] + tokens[i + 1: right_context]
                embedding = [(self.word2vec.get_vector(context_word) - self.mean) / self.std for contex

                if not len(embedding):
                    embedding = np.zeros(self.word2vec.vector_size)
                else:
                    sum(embedding, axis=0)

                mbedding)
            else:
                embeddings.append((self.word2vec.get_vector(word) - self.mean) / self.std)

        if len(embeddings) == 0:
            embeddings = np.zeros((1, self.word2vec.vector_size))
        else:
            embeddings = np.array(embeddings)
            if len(embeddings.shape) == 1:
                embeddings = embeddings.reshape(-1, 1)

        return embeddings


custom_dev_v1 = Custom_TwitterDataset_v1(dev_data, "text", "emotion", word2vec)


train_v1, valid_v1 = random_split(custom_dev_v1, [train_size, len(custom_dev_v1) - train_size])

train_loader_v1 = DataLoader(train_v1, batch_size=batch_size, num_workers=num_workers, shuffle=True, dr
valid_loader_v1 = DataLoader(valid_v1, batch_size=batch_size, num_workers=num_workers, shuffle=False, d
```

Сохранено

```
valid_loader_v1 = DataLoader(valid_v1, batch_size=batch_size, num_workers=num_workers, shuffle=False, c
```

```
best_metric = np.inf
for e in range(num_epochs):
    training(model, optimizer, criterion, train_loader_v1, e, device)
    log = testing(model, criterion, valid_loader_v1, device)
    print(log)
    if log["Test Loss"] < best_metric:
        torch.save(model.state_dict(), "model_v1.pt")
        best_metric = log["Test Loss"]
```

```
Epoch 1. Train Loss: 0.498: 100% 1000/1000 [02:45<00:00, 6.96it/s]
Test Loss: 0.4567, Test Acc: 0.7773: 100% 250/250 [00:41<00:00, 6.52it/s]
{'Test Loss': 0.4912253110408783, 'Test Acc': 0.7586015625}
Epoch 2. Train Loss: 0.4867: 100% 1000/1000 [02:43<00:00, 6.57it/s]
Test Loss: 0.4673, Test Acc: 0.7754: 100% 250/250 [00:41<00:00, 8.34it/s]
{'Test Loss': 0.48985003197193144, 'Test Acc': 0.75958203125}
```

```
test_loader_v1 = DataLoader(
    Custom_TwitterDataset_v1(test_data, "text", "emotion", word2vec),
    batch_size=batch_size,
    num_workers=num_workers,
    shuffle=False,
    drop_last=False,
    collate_fn=average_emb)

model.load_state_dict(torch.load("model_v1.pt", map_location=device))

print(testing(model, criterion, test_loader_v1, device=device))
```

```
Test Loss: 0.528, Test Acc: 0.7285: 100% 313/313 [00:55<00:00, 6.75it/s]
{'Test Loss': 0.4944828003168868, 'Test Acc': 0.7570418580271565}
```

```
# Для реализации 2-ого варианта реализуем вспомогательные функции для создания словаря tfidf_embeddings
#
```

Сохранено

```
кения разряженной матрицы tfidf корпуса документов
```

```
Requirement already satisfied: sparsesvd in /usr/local/lib/python3.7/dist-packages (0.2.2)
Requirement already satisfied: scipy>=0.6.0 in /usr/local/lib/python3.7/dist-packages (from spars
Requirement already satisfied: cython in /usr/local/lib/python3.7/dist-packages (from sparsesvd)
Requirement already satisfied: numpy>=1.13.3 in /usr/local/lib/python3.7/dist-packages (from scip
```

```
from sparsesvd import sparsesvd
from sklearn.feature_extraction.text import TfidfVectorizer

def get_tfidf_embeddings(corpus_docs, embedding_size=300):
    tfidf_vectorizer = TfidfVectorizer(min_df=5)
    sparse_matrix_tfidf = tfidf_vectorizer.fit_transform(corpus_docs)

    u, s, vt = sparsesvd(sparse_matrix_tfidf.tocsc(), embedding_size)
    k, v = tfidf_vectorizer.get_feature_names(), vt.T

    return {k[i]: v[i] for i in range(len(v))}
```

```
tfidf_embeddings = get_tfidf_embeddings(corpus_docs=data["text"].values, embedding_size=300)
```

```
class Custom_TwitterDataset_v2(TwitterDataset):
    def __init__(self, data: pd.DataFrame, feature_column: str, target_column: str, word2vec: gensim.mo
        super(Custom_TwitterDataset_v2, self).__init__(data, feature_column, target_column, word2vec)

        self.tfidf_embeddings = tfidf_embeddings

    '''def get_tokens_(self, text):
        if not hasattr(self, "tokens"):
            nltk.download("wordnet")
            nltk.download("stopwords")
            self.tokenize = nltk.word_tokenize
            self.norm = nltk.WordNetLemmatiizer().lemmatize
            self.stopWords = set(nltk.corpus.stopwords.words("english"))

            self.check = lambda token: ((token.isalnum()) and (token not in self.stopWords))
            self.tokens = [self.norm(token) for token in self.tokenize(text.lower()) if self.check(toke
        return self.tokens'''

    def get_embeddings_(self, tokens):
        embeddings = []

        for i, word in enumerate(tokens):
            if word not in self.word2vec:
                if word in self.tfidf_embeddings:
                    embeddings.append((self.tfidf_embeddings[word] - self.mean) / self.std)
                else:
                    pass
            else:
                if word in self.tfidf_embeddings:
                    embeddings.append((self.word2vec.get_vector(word) + self.tfidf_embeddings[word] - s
                else:
                    embeddings.append((self.word2vec.get_vector(word) - self.mean) / self.std)


        if len(embeddings) == 0:
                                    1, self.word2vec.vector_size))

            embeddings = np.array(embeddings)
            if len(embeddings.shape) == 1:
                embeddings = embeddings.reshape(-1, 1)

        return embeddings


custom_dev_v2 = Custom_TwitterDataset_v2(dev_data, "text", "emotion", word2vec)


train_v2, valid_v2 = random_split(custom_dev_v2, [train_size, len(custom_dev_v2) - train_size])

train_loader_v2 = DataLoader(train_v2, batch_size=batch_size, num_workers=num_workers, shuffle=True, dr
valid_loader_v2 = DataLoader(valid_v2, batch_size=batch_size, num_workers=num_workers, shuffle=False, d


best_metric = np.inf
for e in range(num_epochs):
    training(model, optimizer, criterion, train_loader_v2, e, device)
    log = testing(model, criterion, valid_loader_v2, device)
    print(log)
```

Сохранено

```
        print(log)
        if log["Test Loss"] < best_metric:
            torch.save(model.state_dict(), "model_v2.pt")
            best_metric = log["Test Loss"]
```

```
Epoch 1. Train Loss: 0.4638: 100% 1000/1000 [04:19<00:00, 5.90it/s]
Test Loss: 0.4614, Test Acc: 0.7773: 100% 250/250 [01:06<00:00, 3.89it/s]
{'Test Loss': 0.4801198878288269, 'Test Acc': 0.7663671875}
Epoch 2. Train Loss: 0.48: 100% 1000/1000 [04:25<00:00, 6.37it/s]
Test Loss: 0.4644, Test Acc: 0.7842: 100% 250/250 [01:08<00:00, 4.57it/s]
{'Test Loss': 0.48139447152614595, 'Test Acc': 0.76566796875}
```

```
test_loader_v2 = DataLoader(
    Custom_TwitterDataset_v2(test_data, "text", "emotion", word2vec),
    batch_size=batch_size,
    num_workers=num_workers,
    shuffle=False,
    drop_last=False,
    collate_fn=average_emb)

model.load_state_dict(torch.load("model_v2.pt", map_location=device))

print(testing(model, criterion, test_loader_v2, device=device))
```

```
Test Loss: 0.5107, Test Acc: 0.7441: 100% 313/313 [01:21<00:00, 5.63it/s]
{'Test Loss': 0.49097904572471646, 'Test Acc': 0.759837385183706}
```

```
# Значения метрик(train + val data), которые были получены только с помощью word2vec

# на втором скрине(test data)
```

```
Epoch 1. Train Loss: 0.4959: 100% 1000/1000 [02:28<00:00, 8.19it/s]
Test Loss: 0.4820, Test Acc: 0.7578: 100% 250/250 [00:37<00:00, 6.05it/s]
{'Test Loss': 0.4820, 'Test Acc': 0.75125390625}
Epoch 2. Train Loss: 0.4811: 100% 1000/1000 [02:34<00:00, 9.29it/s]
Test Loss: 0.4664, Test Acc: 0.7607: 100% 250/250 [00:38<00:00, 9.16it/s]
{'Test Loss': 0.49697556138038634, 'Test Acc': 0.755859375}

Test Loss: 0.5302, Test Acc: 0.75: 100% 313/313 [00:46<00:00, 8.28it/s]
{'Test Loss': 0.49849567026756825, 'Test Acc': 0.7552447334265175}
```

Сохранено

```
# Были реализованы 2 дополнительных способа обработки слов(которые не содержались в словаре предобученн

# 1-ый способ состоят в том, чтобы предсавить "незнакомое" слово его контентом(ближайшие к нему соседи

# 2-ой способ состоит в том, чтобы приплюсовать к эмбеддингам слова, эмбеддинги tfidf, полученные на об
```

```
# 1-ый способ

# Значения метрик(train + val data), которые были получены только с помощью word2vec

# на втором скрине(test data)
```

```
Epoch 1. Train Loss: 0.498: 100% 1000/1000 [02:45<00:00, 6.96it/s]
Test Loss: 0.4567, Test Acc: 0.7773: 100% 250/250 [00:41<00:00, 6.52it/s]
{'Test Loss': 0.4912253110408783, 'Test Acc': 0.7586015625}
Epoch 2. Train Loss: 0.4867: 100% 1000/1000 [02:43<00:00, 6.57it/s]
Test Loss: 0.4673, Test Acc: 0.7754: 100% 250/250 [00:41<00:00, 8.34it/s]
{'Test Loss': 0.48985003197193144, 'Test Acc': 0.75958203125}
```

```
Test Loss: 0.528, Test Acc: 0.7285: 100% 313/313 [00:55<00:00, 6.75it/s]
{'Test Loss': 0.4944828003168868, 'Test Acc': 0.7570418580271565}
```

```
# 2-ой способ

# Значения метрик(train + val data), которые были получены только с помощью word2vec

# на втором скрине(test data)
```

```
Epoch 1. Train Loss: 0.4638: 100% 1000/1000 [04:19<00:00, 5.90it/s]
Test Loss: 0.4614, Test Acc: 0.7773: 100% 250/250 [01:06<00:00, 3.89it/s]
{'Test Loss': 0.4801198878288269, 'Test Acc': 0.7663671875}
Epoch 2. Train Loss: 0.48: 100% 1000/1000 [04:25<00:00, 6.37it/s]
250/250 [01:08<00:00, 4.57it/s]
'Test Acc': 0.76566796875}
```

Сохранено

```
Test Loss: 0.5107, Test Acc: 0.7441: 100% 313/313 [01:21<00:00, 5.63it/s]
{'Test Loss': 0.49097904572471646, 'Test Acc': 0.759837385183706}
```

```
# Выводы:

# Конечно же делать выводы и обобщать их на все случаи, учитывая лишь частный случай мы не можем, но ко
# большее кол-во эпох особо эффекта не привнесло) и при данной тестовой выборке дают следующие результа

# По скринам выше(на них приведены логи метрик и лосс на обучении, валидации и тесте), 2 методики, кото
# по сравнению с тем, когда используется исключительно word2vec.
# На самом деле - это ожидаемый результат т.к. бех доп. подъодов мы просто теряем информацию о словах,
# очень сильно коррелирует с тем, какой именно word2vec мы используем(смотря на каком корпусе документо
```

```
# Между 2-мя способами лучше себя показал подход с использванием tfidf.
# Могу предположить, что 1-ый подход показал себя хуже т.к. представить слово через усреднение его конт
# данный подход будет корректен...

# А подход tfidf лишь вносит дополнительную информацию в пространство word2vec, что позволяет модели лу
```

✓ 0 сек. выполнено в 19:09

Сохранено