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:

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

```
Downloading...
```

```
From: https://drive.google.com/uc?id=1eE1FiUkXkcbw0McId4i7qY-L8hH-_Qph
To: /content/archive.zip
84.9MB [00:00, 97.8MB/s]
Archive:  archive.zip
  inflating: training.1600000.processed.noemoticon.csv
```

Импортируем нужные библиотеки:

In [ ]:
```
import math
import random
import string

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

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

In [ ]:
```
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"
```

In [ ]:
```
data = pd.read_csv("training.1600000.processed.noemoticon.csv", encoding="latin", header=None, names=["emotion", "id", "c
```

Посмотрим на данные:

In [ ]:
```
data.head()
```

Out[ ]:

	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...
3	0	1467811184	Mon Apr 06 22:19:57 PDT 2009	NO_QUERY	ElleCTF	my whole body feels itchy and like its on fire
4	0	1467811193	Mon Apr 06 22:19:57 PDT 2009	NO_QUERY	Karoli	@nationwideclass no, it's not behaving at all....

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

In [ ]:
```
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 have a sleepover or what
ever, ill call u
School email won't open  and I have geography stuff on there to revise! *Stupid School* :'(
upper airways problem
Going to miss Pastor's sermon on Faith...
on lunch....dj should come eat with me
@piginthepoke oh why are you feeling 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 knitting it!
```

Как вилим, 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).

In [ ]:
```
indexes = np.arange(data.shape[0])
np.random.shuffle(indexes)
dev_size = math.ceil(data.shape[0] * 0.8)

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:

In [ ]:
```
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 may sometime i hope thou
```

```
gh !
```

In [ ]:
```
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
```

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.

In [ ]:
```
word2vec = api.load("word2vec-google-news-300")
```

```
[==================================================-] 98.3% 1634.2/1662.8MB downloaded
```

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

```
(300,)
```

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

In [ ]:
```
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(w) > 3]
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]
```

Сделаем датасет, который будет по запросу возвращать подготовленные данные.

In [ ]:
```
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.models.Word2Vec):
        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
        line = self.tokenizer.tokenize(text.lower())
        tokens = [w for w in line if all(c not in string.punctuation for c in w)\
                                     and len(w) > 3]  # filtered_line
        return tokens

    def get_embeddings_(self, tokens):
        # Получи эмбеддинги слов и усредни их
        # ***************** ПРИМЕЧАНИЕ:
        # Судя по коду здесь в методе и тому, что для усреднения далее используется
        # отдельная функция average_emb(batch), здесь (в этом методе) усреднение
        # эмбедингов не предполагается. Поэтому выполнена только их НОРМАЛИЗАЦИЯ
        embeddings = np.array([(self.word2vec.get_vector(w) - self.mean) / self.std
                               if w in self.word2vec else np.zeros(self.word2vec.vector_size)
                               for w in tokens])

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

        return embeddings

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

In [ ]:
```
dev = TwitterDataset(dev_data, "text", "emotion", word2vec)
```

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

Превращать твиты в векторы, используя эмбеддинги слов, можно несколькими способами. А именно такими:

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 предлоежнии (которые остались после токенизации и удаления коротких слов, конечно).

In [ ]:
```
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
```

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.

In [ ]:
```
from sklearn.decomposition import PCA


pca = PCA(n_components=2)
# Обучи PCA на эмбеддингах слов
examples["transformed_features"] = pca.fit_transform(np.array(examples["features"]))
```

In [ ]:
```
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
```

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

Out[ ]: **Figure**(id = '1108', ...)

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.

In [ ]:
```
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_last=True, collate_fr
valid_loader = DataLoader(valid, batch_size=batch_size, num_workers=num_workers, shuffle=False, drop_last=False, collate_
```

Определим функции для тренировки и теста модели:

In [ ]:

```
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:
        optimizer.zero_grad()

        features = batch["features"].to(device)
        targets = batch["targets"].to(device)

        # Получи предсказания модели
        pred = model(features)
        loss = criterion(pred, targets) # Посчитай лосс
        # Обнови параметры модели
        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
            pred = model(features)
            loss = criterion(pred, targets) # Посчитай лосс
            acc = torch.sum((pred[:, 1] > 0.5) == targets) / len(targets) # Посчитай точность модели

            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_loader):.4}")

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

Создадим модель, оптимизатор и целевую функцию. Вы можете сами выбрать количество слоев в нейронной сети, ваш любимый оптимизатор и целевую функцию.

In [ ]:
```
def make_model(vector_size, num_classes):
    model_ = nn.Sequential(
                nn.Linear(vector_size, 1000),
                nn.ReLU(),
                nn.Linear(1000, num_classes),
            )
    return model_
```

In [ ]:
```
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-2
num_epochs = 1

model = make_model(vector_size, num_classes)  # Твоя модель
model = model.cuda()
criterion = nn.CrossEntropyLoss()  # Твой лосс
optimizer = Adam(model.parameters(), lr=lr)  # 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?**

Мои мысли по поводу метрики

Я бы сказал, что в данной задаче логичнее пользоваться лоссом. С точностью есть опасность, что выборка несбалансированная (мы этого не знаем) и по этой причине ее показания могут быть не совсем релевантными

In [ ]:
```
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"]
```

```
{'Test Loss': 0.525543437242508, 'Test Acc': 0.70152734375}
```

In [ ]:
```
test_loader = DataLoader(
    TwitterDataset(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.pt", map_location=device))

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

```
{'Test Loss': 0.522842538432953, 'Test Acc': 0.6919709714456869}
```

Поиграемся с параметрами

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

model = make_model(vector_size, num_classes)  # Твоя модель
model = model.cuda()
criterion = nn.CrossEntropyLoss()  # Твой лосс
optimizer = Adam(model.parameters(), lr=lr)  # Твой оптимайзер

# train
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_1.pt")
        best_metric = log["Test Loss"]

# test
test_loader = DataLoader(
    TwitterDataset(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_1.pt", map_location=device))

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

```
{'Test Loss': 0.5188404930830002, 'Test Acc': 0.71798046875}
```

```
{'Test Loss': 0.5206245120150593, 'Test Acc': 0.7170870357428115}
```

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. 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.

Вариант 1: Эмбеддингом незнакомого слова будет сумма эмбеддингов всех слов из его контекста

Скорректируем класс Датасета для реализации этого варианта задания

In [ ]:
```
class TwitterDatasetContext(TwitterDataset):
    def __init__(self, data: pd.DataFrame, feature_column: str, target_column: str, word2vec: gensim.models.Word2Vec,
                 window_size=3):
        super().__init__(data, feature_column, target_column, word2vec)
        self.window_size = window_size

    def get_embeddings_(self, tokens):
        # Получи эмбеддинги слов и нормализуй их
        embeddings = np.array([(self.word2vec.get_vector(w) - self.mean) / self.std
                               if w in self.word2vec else self.get_context_emb(idx, tokens)
                               for idx, w in enumerate(tokens)])

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

        return embeddings

    def get_context_emb(self, idx, tokens):
        start_idx = max(0, idx - self.window_size)
        end_idx = min(len(tokens), idx + self.window_size + 1)
        pos_in_window = self.window_size
        if idx - self.window_size < 0:  # start of the sentence
            pos_in_window += idx - self.window_size

        co_words = tokens[start_idx:end_idx]  # cuts window from sentence
        co_words = np.delete(co_words, pos_in_window)  # deletes central word from context

        context_emb = np.zeros(self.word2vec.vector_size)
        emb_num = 0
        for word in co_words:
            if word in self.word2vec:
                emb = (self.word2vec.get_vector(word) - self.mean) / self.std
                context_emb += emb
                emb_num += 1
        if emb_num != 0:
            context_emb /= emb_num
        return context_emb
```

In [ ]:
```
dev = TwitterDatasetContext(dev_data, "text", "emotion", word2vec)
```

```
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_last=True, collate_fr
valid_loader = DataLoader(valid, batch_size=batch_size, num_workers=num_workers, shuffle=False, drop_last=False, collate_
```

In [ ]:

```
vector_size = dev.word2vec.vector_size
num_classes = 2
lr = 1e-2
num_epochs = 1

model = make_model(vector_size, num_classes)  # Твоя модель
model = model.cuda()
criterion = nn.CrossEntropyLoss()  # Твой лосс
optimizer = Adam(model.parameters(), lr=lr)  # Твой оптимайзер

# train
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_con.pt")
        best_metric = log["Test Loss"]

# test
test_loader = DataLoader(
    TwitterDatasetContext(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_con.pt", map_location=device))

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

```
{'Test Loss': 0.5132736160755157, 'Test Acc': 0.68583984375}

{'Test Loss': 0.5133158340812111, 'Test Acc': 0.6853596745207667}
```

Вариант 2: Итоговым эмбеддингом для каждого слова будет сумма двух эмбеддингов: предобученного и Tfidf-ного

Tfidf (пробуем обработку)

In [ ]:
```
dev_data["text"]
```

Out[ ]:
```
0          @Claire_Nelson i'm on the north devon coast th...
1          @jhicks  i will think of you on Sunday!   Who ...
2          Out in the garden with the kids debating wheth...
3          @FrVerona thank u my love...u've shown me the ...
4          is with @jonasbrosfan1  going to buy LVATT tog...
                                ...
1279995    @yajtyler thats so sweet =] they like you  and...
1279996       Really sad the NBA playoffs are over for good
1279997    @JuJuBeanz15 LOL Im glad that u have that pers...
1279998           I have to go to the GYM  it's toooooo hot.
1279999    @Natalie_McLife okay will do hen it has finish...
Name: text, Length: 1280000, dtype: object
```

In [ ]:
```
def to_processed_corpus(text):
    # Получи все токены из текста и профильтруй их
    line = tokenizer.tokenize(text.lower())
    tokens = [w for w in line if all(c not in string.punctuation for c in w)\
                                 and len(w) > 3]  # filtered_line
    return " ".join(tokens)
```

In [ ]:
```
%%time
train_corpus = dev_data["text"][train.indices].apply(to_processed_corpus)
print(len(train_corpus))
```

```
1024000
CPU times: user 22.1 s, sys: 181 ms, total: 22.3 s
Wall time: 22.3 s
```

In [ ]:
```
train_corpus[0]
```

Out[ ]:
```
'north devon coast next weeks will down devon again sometime hope though'
```

In [ ]:
```
import sklearn
from sklearn.feature_extraction.text import TfidfVectorizer
```

In [ ]:
```
%%time
```

```
tfidf_vectorizer = TfidfVectorizer()
X = tfidf_vectorizer.fit_transform(train_corpus)
print(X.shape)
```

```
(1024000, 469010)
CPU times: user 11.9 s, sys: 161 ms, total: 12.1 s
Wall time: 12.1 s
```

SVD для уменьшения числа фич в Tfidf

Пробуем **Усеченный SVD** для уменьшения числа признаков до 300 для складывания с эмбедингами.

In [ ]:
```
# %%time
# from sklearn.decomposition import TruncatedSVD

# svd = TruncatedSVD(n_components=300)
# X_300 = svd.fit_transform(X)
```

Резюме по Усеченному SVD

Если мы вместо **max_features** пытаемся регулировать количество фич в **tfidf** через **PCA** - этот вариант не проходит. Преобразовать не получается из-за переполнения ОЗУ.
ПРИМЕЧАНИЕ: конкатенировать все фичи (их несколько десятков или даже сотен тысяч) тоже не получилось.

Используем параметр **max_features** для регулирования количества фич

In [ ]:
```
%%time
tfidf_vectorizer = TfidfVectorizer(max_features=300)
X = tfidf_vectorizer.fit_transform(train_corpus)
print(X.shape)
```

```
(1024000, 300)
CPU times: user 11.2 s, sys: 136 ms, total: 11.4 s
Wall time: 11.4 s
```

In [ ]:
```
len(tfidf_vectorizer.get_feature_names())
```

Out[ ]:
```
300
```

In [ ]:
```
tfidf_vectorizer.transform([train_corpus[2].split()[0]]).toarray()[0].shape
```

Out[ ]:
```
(300,)
```

Сделаем класс Датасета под эту задачу.

Реализуем **2 подварианта**:

А) когда мы фичи складываем, параметр **mode = "sum"**

Б) второй вариант - фичи конкатенируем, параметр **mode = "concat"**

In [ ]:
```
class TwitterDatasetTfidf(TwitterDataset):
    def __init__(self, data: pd.DataFrame, feature_column: str, target_column: str, word2vec: gensim.models.Word2Vec,
                 tfidf_vectorizer: sklearn.feature_extraction.text.TfidfVectorizer, mode: str="sum"):
        super().__init__(data, feature_column, target_column, word2vec)
        self.tfidf_vectorizer = tfidf_vectorizer
        self.mode = mode
        self.vector_size = None

    def get_embeddings_(self, tokens):
        # Получи эмбеддинги слов и нормализуй их
        embeddings = np.array([(self.word2vec.get_vector(w) - self.mean) / self.std
                               if w in self.word2vec else np.zeros(self.word2vec.vector_size)
                               for w in tokens])
        tfidf_emb = self.get_tfidf_emb(tokens)
        # print("tfidf_emb.shape", tfidf_emb.shape, "embeddings.shape", embeddings.shape)
        # raise

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

        if self.mode == "sum":
            if embeddings.shape == tfidf_emb.shape:
                embeddings += tfidf_emb
            else:
                raise ValueError("embeddings shape should be match tfidf_emb shape")
        elif self.mode == "concat":
            embeddings = np.hstack((embeddings, tfidf_emb))
        else:
            raise ValueError("mode should be one of 'sum' or 'concat'")
        # print("after hstack embeddings.shape", embeddings.shape)
        self.vector_size = embeddings.shape[1]
        return embeddings

    def get_tfidf_emb(self, tokens):
```

```
        # tfidf_emb = np.array([np.zeros(len(self.tfidf_vectorizer.get_feature_names()))
        #                       for w in tokens])
        tfidf_emb = np.array([self.tfidf_vectorizer.transform([w]).toarray()[0]
                              for w in tokens])
        return tfidf_emb

    @staticmethod
    def to_processed_corpus(text):
        """
        предобрабатываем  тексты для обучения tfidf_vectorizer так же, как мы
        предобрабатывали тексты для работы с эмбедингами
        """
        line = tokenizer.tokenize(text.lower())
        tokens = [w for w in line if all(c not in string.punctuation for c in w)\
                                     and len(w) > 3]  # filtered_line
        return " ".join(tokens)

    def tfidf_vectorizer_fit(self, train_indices):
        train_corpus = self.data["text"][train_indices].apply(self.to_processed_corpus)
        self.tfidf_vectorizer.fit(train_corpus)
        if self.mode == "sum":
            self.vector_size = self.word2vec.vector_size
        elif self.mode == "concat":
            self.vector_size = self.word2vec.vector_size + len(self.tfidf_vectorizer.get_feature_names())
        else:
            raise ValueError("mode should be one of 'sum' or 'concat'")
```

ПРИМЕЧАНИЕ

tfidf_vectorizer обучаем только на текстах из **train**. Тексты для обучения предобрабатываем так же, как мы предобрабатывали тексты для работы с эмбедингами (чтобы было соответствие)

In [ ]:
```
# import warnings
# warnings.filterwarnings("ignore")  # to on use "default"
```

a) Подвариант A: Складываем фичи

mode = 'sum'

In [ ]:
```
%%time
batch_size = 1024
num_workers = 4
tfidf_vectorizer = TfidfVectorizer(max_features=300)
```

```
dev = TwitterDatasetTfidf(dev_data, "text", "emotion", word2vec, tfidf_vectorizer, mode="sum")

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

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

dev.tfidf_vectorizer_fit(train.indices)  # обучаем tfidf_vectorizer только на train данных

train_loader = DataLoader(train, batch_size=batch_size, num_workers=num_workers, shuffle=True, drop_last=True, collate_fr
valid_loader = DataLoader(valid, batch_size=batch_size, num_workers=num_workers, shuffle=False, drop_last=False, collate_
```

In [ ]:

```
vector_size = dev.vector_size
num_classes = 2
lr = 1e-2
num_epochs = 1

model = make_model(vector_size, num_classes)  # Твоя модель
model = model.cuda()
criterion = nn.CrossEntropyLoss()  # Твой лосс
optimizer = Adam(model.parameters(), lr=lr)  # Твой оптимайзер

# train
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_tfidf-sum.pt")
        best_metric = log["Test Loss"]

# test
test_loader = DataLoader(
    TwitterDatasetTfidf(test_data, "text", "emotion", word2vec, tfidf_vectorizer, mode="sum"),
    batch_size=batch_size,
    num_workers=num_workers,
    shuffle=False,
    drop_last=False,
    collate_fn=average_emb)

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

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

```
{'Test Loss': 0.5201045408248901, 'Test Acc': 0.6684765625}
```

```
{'Test Loss': 0.5209477536213665, 'Test Acc': 0.6682963508386581}
```

б) Подвариант Б: Конкатенируем фичи

mode = 'concat'

In [ ]:
```
%%time

batch_size = 1024
num_workers = 4
tfidf_vectorizer = TfidfVectorizer(max_features=300)

dev = TwitterDatasetTfidf(dev_data, "text", "emotion", word2vec, tfidf_vectorizer, mode="concat")

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

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

dev.tfidf_vectorizer_fit(train.indices)  # обучаем tfidf_vectorizer только на train данных

train_loader = DataLoader(train, batch_size=batch_size, num_workers=num_workers, shuffle=True, drop_last=True, collate_fr
valid_loader = DataLoader(valid, batch_size=batch_size, num_workers=num_workers, shuffle=False, drop_last=False, collate_
```

```
CPU times: user 36.5 s, sys: 600 ms, total: 37.1 s
Wall time: 37.1 s
```

In [ ]:
```
vector_size = dev.vector_size
num_classes = 2
lr = 1e-2
num_epochs = 1

model = make_model(vector_size, num_classes)  # Твоя модель
model = model.cuda()
criterion = nn.CrossEntropyLoss()  # Твой лосс
optimizer = Adam(model.parameters(), lr=lr)  # Твой оптимайзер

# train
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_tfidf-concat.pt")
        best_metric = log["Test Loss"]

# test
test_loader = DataLoader(
    TwitterDatasetTfidf(test_data, "text", "emotion", word2vec, tfidf_vectorizer, mode="concat"),
    batch_size=batch_size,
    num_workers=num_workers,
    shuffle=False,
    drop_last=False,
    collate_fn=average_emb)

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

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

```
{'Test Loss': 0.5142746659517288, 'Test Acc': 0.71913671875}
```

```
{'Test Loss': 0.5148940229187378, 'Test Acc': 0.7188748003194888}
```

Резюме

Из реализованных вариантов лучше всего сработал вариант с Tfidf, подвариант с конкатенацией.
На мой взгляд это логично.
Вариант в контекстами вместо неизвестного слова заметного прироста не дал, поскольку информация о контекстах берется из того же самого текста, который уже есть в фичах. То есть чего-то совсем нового не появляется.
При складывании эмбедингов и Tfidf в фичи вносится шум, поэтому прироста тоже особого нет.
А при конкатенации эмбедингов и Tfidf есть некоторый прирост аккураси, поскольку информация объединяется, на мой взгляд, более оптимально и без зашумления фич.