Обработка пользовательских мнений с помощью нейросетей для определения цифровой репутации компании.

Содержание

Аннотация

Проект «Обработка пользовательских мнений с помощью нейросетей для определения цифровой репутации компании» посвящен исследованию возможностей применения нейронных сетей для анализа и оценки пользовательских текстовых отзывов и комментариев с целью определения и улучшения цифровой репутации компании. В рамках проекта была проведена теоретическая работа, освещающая предметную область, и изучены методы обработки текстов.

Практическая часть проекта включала в себя обучение нейронной модели на обширном объеме размеченных данных, а также оптимизацию ее параметров с целью достижения максимальной точности в определении эмоциональной окраски текстов. Полученные результаты подтверждают высокую эффективность разработанной методологии и обозначают перспективу применения данной технологии в различных областях, связанных с анализом и управлением информацией, получаемой из разнообразных неструктурированных источников с целью улучшения цифровой репутации компании.

Часть I

Введение

Обработка естественного языка (Natural Language Processing далее **NLP**) сочетает в себе компьютерную лингвистику, машинное обучение и модели глубокого обучения для обработки человеческого языка. [5]

Компьютерная лингвистика – это наука о понимании и построении моделей человеческого языка с помощью компьютеров и программных инструментов. Исследователи используют методы компьютерной лингвистики, такие как синтаксический и семантический анализ, для создания платформ, помогающих машинам понимать разговорный человеческий язык.

Машинное обучение – это технология, которая обучает компьютер с помощью выборочных данных для повышения его эффективности. Человеческий язык имеет несколько особенностей, таких как сарказм, метафоры, вариации в структуре предложений, а также исключения из грамматики и употребления, на изучение которых у людей уходят годы. Программисты используют методы машинного обучения, чтобы научить приложения NLP распознавать и точно понимать эти функции с самого начала.

Глубокое обучение – это особая область машинного обучения, которая учит компьютеры учиться и мыслить как люди. Это включает нейросеть, состоящую из узлов обработки данных, напоминающих операции человеческого мозга. С помощью глубокого обучения компьютеры распознают, классифицируют и сопоставляют сложные закономерности во входных данных.

В современном мире текстовые данные играют огромную роль. Они используются в различных областях, таких как социальные сети, новостные порталы, бизнес-анализ, перевод текстов с одного языка на другой и многое другое. Задача классификации текста позволяет автоматически определять к какому классу относится текст, что может быть полезно для автоматизации процессов обработки и анализа текстовых данных. В данном проекте мы будем использовать нейронные сети для классификации текста и рассмотрим примеры их применения.

В моей работе **объектом** исследования и дальнейшей разработки стала нейронная сеть, решающая задачу Анализа тональности текста (Sentiment analysis).

Предметом исследования стало создание модели нейросети, способная классифицировать тексты на основе их содержания, с высокой точностью и эффективностью.

Цель – изучить принципы работы нейросетей. Научиться решать задачи обработки естественного языка с помощью нейронных сетей.

Задачи:

- Изучение основных концепций и принципов работы нейронных сетей для классификации текста.
- Сбор и подготовка текстовых данных для обучения нейронной сети.
- Разработка и обучение нейронной сети для классификации текста.
- Оценка качества работы нейронной сети с помощью метрик точности.
- Анализ результатов работы нейронной сети и определение возможных путей улучшения её эффективности.
- Исследование возможностей использования обученной нейронной сети в различных областях.

Структура работы:

1) Теоретические основы (обзор подходов и инструментов)

2) Реализации (а именно: создание нейросети и обучение.)

Часть II

Теоретические основы

Нейронные сети - это компьютерные системы, которые моделируют работу нейронов в головном мозге и используются для решения различных задач, в том числе для обработки текста.

Backpropagation

Backpropagation - это алгоритм обучения нейронной сети, который используется для настройки весов между нейронами. Принцип работы backpropagation заключается в передаче ошибки от выходного слоя к входному слою, чтобы корректировать веса нейронов.

Алгоритм работы backpropagation:

1. Прямой проход: входные данные подаются на входной слой нейронной сети, затем вычисляются значения на каждом слое до выходного слоя.

2. Вычисление ошибки: сравниваются выходные значения нейронной сети с ожидаемыми значениями и вычисляется ошибка.

3. Обратный проход: ошибка передается от выходного слоя к входному слою, корректируя веса нейронов на каждом слое.

4. Обновление весов: после корректировки весов на каждом слое, они обновляются и процесс повторяется до достижения определенного уровня точности.

Оптимизаторы являются ключевым элементом для улучшения процесса обучения с помощью backpropagation. Их работа заключается в изменения скорости обучения и корректировки весов на каждом слое нейронной сети. Некоторые из наиболее популярных оптимизаторов: Adam, RMSprop и Adagrad.

Learning rate - скорость обучения, которая определяет, насколько быстро изменяются веса нейронной сети в процессе обучения. Она является одним из параметров, которые определяются оптимизатором и влияют на процесс обучения

Таблица 1: Оптимизаторы

Название оптимизатора	Формулы подсчёта	Описание параметров
SGD with momentum	$v_t = \gamma v_{t-1} + \eta \nabla_\theta J(\theta)$ $\theta_t = \theta_{t-1} - v_t$	γ - коэффициент затухания η - скорость обучения $\nabla_\theta J(\theta)$ - градиент функции потерь по параметрам θ v_t - скользящее среднее градиента на шаге t θ_t - значения параметров на шаге t
Adagrad	$g_t = \nabla_\theta J(\theta)$ $s_t = s_{t-1} + g_t^2$ $\theta_t = \theta_{t-1} - \frac{\eta}{\sqrt{s_t + \epsilon}} g_t$	g_t - градиент функции потерь на шаге t s_t - сумма квадратов градиентов до шага t ϵ - малое число для стабильности вычислений
Adam	$m_t = \beta_1 m_{t-1} + (1 - \beta_1) g_t$ $v_t = \beta_2 v_{t-1} + (1 - \beta_2) g_t^2$ $\hat{m}_t = \frac{m_t}{1 - \beta_1^t}$ $\hat{v}_t = \frac{v_t}{1 - \beta_2^t}$ $\theta_t = \theta_{t-1} - \frac{\eta}{\sqrt{\hat{v}_t + \epsilon}} \hat{m}_t$	m_t и v_t - оценки первого и второго моментов градиента на шаге t β_1 и β_2 - коэффициенты затухания для моментов $\hat{m}_t$ и $\hat{v}_t$ - скорректированные оценки моментов ϵ - малое число для стабильности вычислений

Для обучения нейронной сети необходимо подготовить набор данных, который будет использоваться для тренировки. Это могут быть различные текстовые документы, размеченные по категориям. Для улучшения качества обучения можно использовать методы предварительной обработки данных, такие как лемматизация, стемминг и удаление стоп-слов.

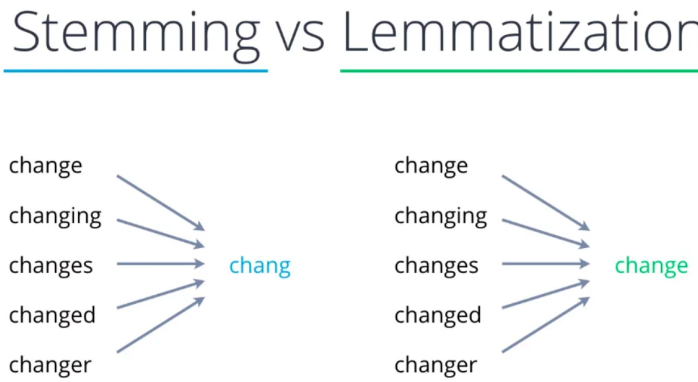


Рис. 1: Лемматизация и Стемминг

Предобработка текста

Предобработка текста является важным шагом перед обучением нейронных сетей для обработки естественного языка. Она включает в себя ряд операций, которые позволяют привести текст к формату, понятному для модели.

Одной из первых операций является токенизация, при которой текст разбивается на отдельные слова или токены. Это может быть достигнуто с помощью различных алгоритмов, таких как методы на основе правил или машинного обучения.

Далее, приводится к нормализованному виду, путем удаления знаков препинания, стоп-слов и других ненужных символов. Это помогает уменьшить размерность входных данных и улучшить качество модели.

Затем текст преобразуется в числовой формат, например, с помощью метода векторизации слов (word embedding), при котором каждое слово представляется в виде вектора чисел. Это позволяет модели работать с числовыми данными и учитывать связи между словами в предложении.

Перед обучением модели проводится операция паддинга (padding), при которой все входные данные приводятся к одной и той же длине.

Word embedding [6]

Word embedding - это метод представления слов в виде векторов чисел. Каждое слово в тексте представляется в виде вектора, где каждая компонента соответствует определенному признаку или характеристике слова.

Word embedding может быть создан с помощью различных алгоритмов, таких как Word2Vec, GloVe и FastText. Эти алгоритмы используют контекстные признаки слов, чтобы определить их семантические свойства и создать векторное представление.

С помощью word embedding модель может учитывать связи между словами в тексте и понимать смысл предложений. Например, слова «кошка» и «собака» будут иметь близкие векторные представления, потому что они относятся к животным, в то время как слово «автомобиль» будет иметь более далекое векторное представление. Word embedding также позволяет модели обраба-

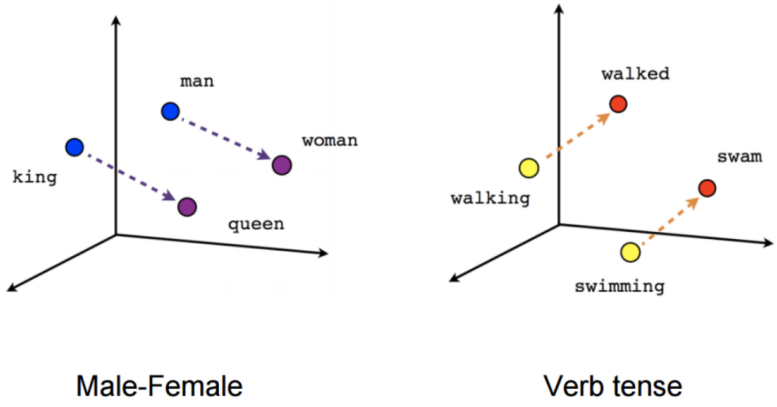


Рис. 2: Word embedding

тывать неизвестные слова, которые не были встречены в обучающих данных. Модель может использовать контекстные признаки, чтобы определить семантику неизвестного слова и создать соответствующее векторное представление.

Чтобы улучшить векторное представление слов, и предоставить возможность нейронной сети «выучить» язык, в 2018 году была представлена нейросеть BERT, которая является одной из самых популярных моделей в NLP.

Намного выгоднее заранее обучить подобное векторное представление слов на каком-нибудь огромном корпусе текстов, например на всей Wikipedia, а в конкретных нейронных сетях использовать уже готовые векторы слов, а не обучать их каждый раз заново.

BERT [3]

Нейронная сеть BERT использует архитектуру трансформеров*(см Приложение) , которая позволяет ей эффективно обрабатывать последовательности данных, такие как тексты. Она состоит из множества слоев, каждый из которых содержит несколько механизмов внимания (attention mechanisms), которые позволяют модели сосредоточиться на наиболее важных частях входных данных.

BERT является бидирекциональной моделью, что означает, что она способна учитывать контекст как слева, так и справа от текущего слова. Это позволяет ей лучше понимать связи между словами в предложении и улучшать качество обработки естественного языка.

Для обучения BERT используется метод маскирования (masking), при котором случайно выбранные слова в предложении заменяются на специальный токен [MASK]. Это позволяет модели учиться предсказывать пропущенные слова и лучше понимать контекст.

Кроме того, BERT использует метод предсказания следующего предложения (next sentence prediction), при котором модель обучается предсказывать, являются ли два предложения последовательными или нет. Это помогает модели лучше понимать связи между предложениями и улучшать качество ответов на вопросы.

BERT является мощной моделью нейронной сети для обработки естественного языка, которая позволяет достигать высокой точности в решении различных задач. Ее архитектура и принципы работы позволяют ей эффективно учитывать контекст и связи между словами в предложении.

Поэтому в практической части для представления слов в форме векторов будем использовать предобученную BERT. То есть реализация нейронной сети будет в формате дообучения BERT под нашу задачу.

Часть III

Практическая часть

Задача: Анализ тональности текста (Sentiment analysis) датасета с Kaggle.

Код проекта на Google Colab.

Для реализации нейронной сети я выбрала библиотеку PyTorch [4] на Python по нескольким причинам:

1. *Простота и удобство использования:* PyTorch имеет простой и интуитивно понятный интерфейс, что делает его легким в освоении и использовании.

2. *Гибкость:* PyTorch позволяет гибко определять структуру нейронных сетей и изменять ее в процессе обучения, что может быть очень полезно при экспериментировании с различными архитектурами.

3. *Высокая производительность:* PyTorch использует оптимизированные вычисления на графических процессорах (GPU), что позволяет значительно ускорить процесс обучения.

1. Введение

Цель - описать процесс классификации эмоций в текстах с помощью нейросети на основе датасета «Emotions Dataset for NLP». Для достижения этой цели были выполнены следующие шаги: подготовка данных, создание модели нейросети, обучение модели и оценка результатов.

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
from tqdm import tqdm

import torch
import torch.nn as nn
from torch.utils.data import Dataset, DataLoader
from transformers import BertTokenizer, BertModel

from sklearn.preprocessing import LabelEncoder
from sklearn.metrics import f1_score, roc_auc_score, roc_curve
from sklearn.metrics import classification_report, auc
```

Рис. 3: Импортируемые бибилиотеки

Импортируем необходимые бибилиотеки: Pandas и Numpy для работы с датасетом, matplotlib.pyplot для отрисовки граффиков, tqdm для оборачивания итерируемых объектов, torch для написания нейросети, LabelEncoder из sklearn.preprocessing для кодировки категориальных переменных и f1_score, roc_auc_score, roc_curve, classification_report, auc из sklearn.metrics для вывода необходимых метрик.

2. Описание датасета

Датасет «Emotions Dataset for NLP» содержит 416809 текстовых сообщений, каждое из которых относится к одной из пяти эмоций: радость, грусть, злость, страх и нейтральность.

```
train_df = pd.read_csv('train.txt', names=['text', 'emotion'], sep=';')
val_df = pd.read_csv('val.txt', names=['text', 'emotion'], sep=';')
test_df = pd.read_csv('test.txt', names=['text', 'emotion'], sep=';')

train_df.head()
```

	text	emotion
0	i didnt feel humiliated	sadness
1	i can go from feeling so hopeless to so damned...	sadness
2	im grabbing a minute to post i feel greedy wrong	anger
3	i am ever feeling nostalgic about the fireplac...	love
4	i am feeling grouchy	anger

Рис. 4: Загрузка train, val и test

Выберем максимальную длинну слова о MAX_LEN = 64, так как почти все текста будут иметь такую длину

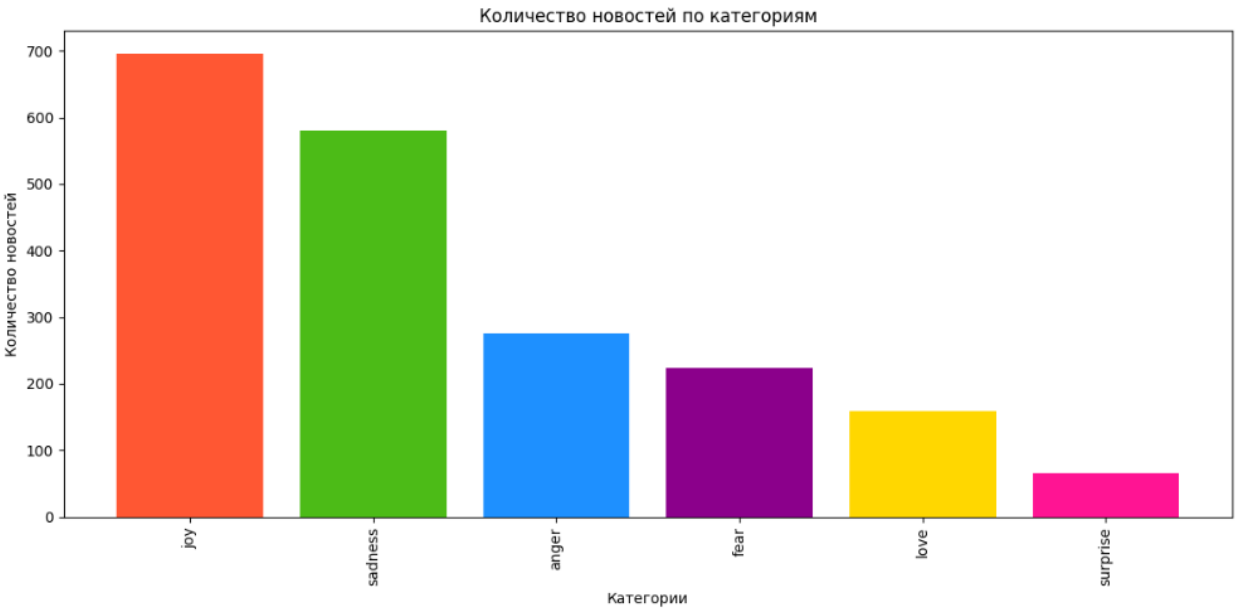


Рис. 5: Распределение классов

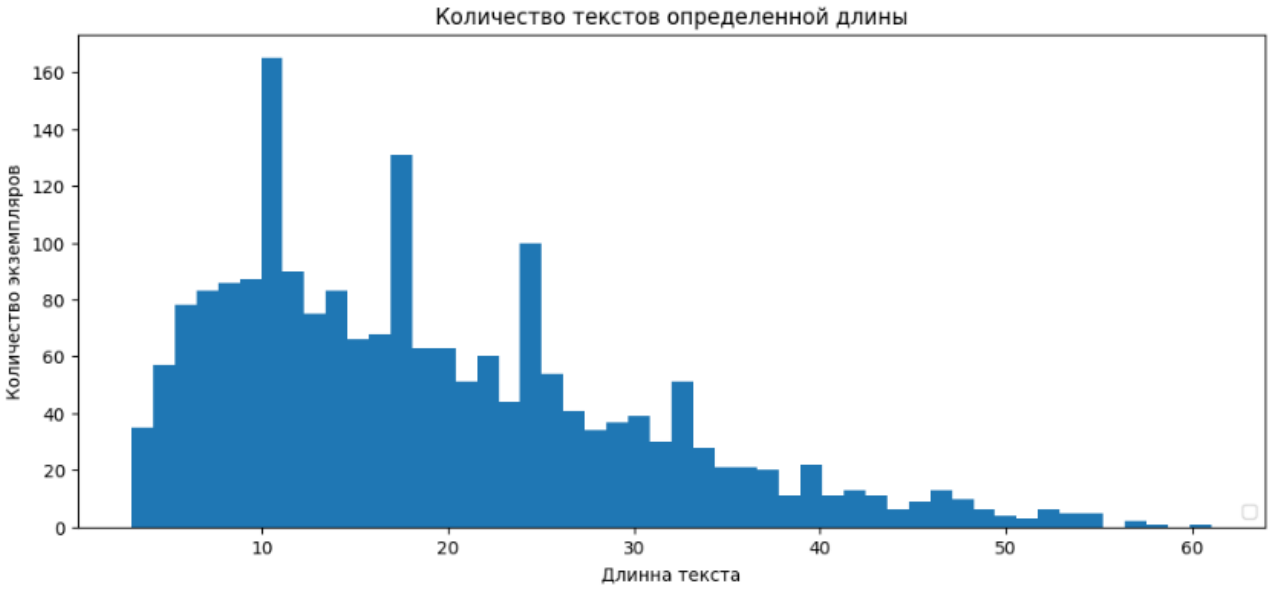


Рис. 6: Статистика распределения длин слов с датасете

3. Подготовка данных

Перед обучением модели необходимо было подготовить данные. Для этого были выполнены следующие шаги:

- Токенизация: каждое сообщение было разбито на отдельные слова.

- Перевод меток классов в числа: каждой эмоции было присвоено уникальное числовое значение.

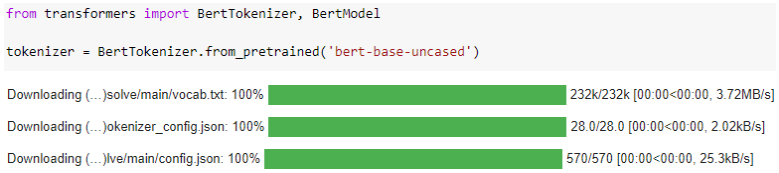


Рис. 7: Загрузка токенайзера

```
le = LabelEncoder()
train_df['emotion'] = le.fit_transform(train_df['emotion'])
val_df['emotion'] = le.transform(val_df['emotion'])
val_df.head()
```

	text	emotion
0	im feeling quite sad and sorry for myself but ...	4
1	i feel like i am still looking at a blank canv...	4
2	i feel like a faithful servant	3
3	i am just feeling cranky and blue	0
4	i can have for a treat or if i am feeling festive	2

Рис. 8: Кодирование категориальных переменных

Стандартный класс Dataset расширяется методами __init__, __len__, __getitem__. В методе __init__ инициализируем тексты, метки, максимальную дину текста в токенах, а так же токенайзер.

Метод len возвращает длину нашего датасета. Метод getitem возвращает словарь, который состоит из самого исходного текста, списка токенов, маски внимания, а также метки класса. Отдельно хочется остановить на настройках токенизатора с помощью метода encode_plus().

В этом методе мы указываем токенизатору, что исходный текст нужно обрамлять служебными токенами add_special_tokens=True, а также дополнять полученные векторы до максимально длины padding='max_len'.

Далее указываем shuffle=True для перемешки значений тренировочной выборки. TRAIN_BATCH_SIZE = 16 VALID_BATCH_SIZE = 16

Создадим Dataset класс для нашего набора данных. [7]

```
class EmotionDataset(Dataset):
    def __init__(self, df, tokenizer, max_len):
        self.df = df
        self.tokenizer = tokenizer
        self.max_len = max_len

    def __len__(self):
        return len(self.df)

    def __getitem__(self, index):
        text = str(self.df.loc[index, 'text'])
        emotion = self.df.loc[index, 'emotion']

        encoding = self.tokenizer.encode_plus(
            text,
            add_special_tokens=True,
            truncation=True,
            max_length=self.max_len,
            return_token_type_ids=False,
            padding='max_length',
            return_attention_mask=True,
            return_tensors='pt',
        )

        return {
            'text': text,
            'input_ids': encoding['input_ids'].flatten(),
            'attention_mask': encoding['attention_mask'].flatten(),
            'emotion': torch.tensor(emotion, dtype=torch.long)
        }
```

Рис. 9: EmotionDataset

Создаем DataLoader для итерации по бачам.

```
train_dataset = EmotionDataset(train_df, tokenizer, MAX_LEN )
train_loader = DataLoader(train_dataset, batch_size=TRAIN_BATCH_SIZE ,
                          shuffle=True)

val_dataset = EmotionDataset(val_df, tokenizer, MAX_LEN)
val_loader = DataLoader(val_dataset, batch_size=VALID_BATCH_SIZE )
```

Рис. 10: DataLoader

4. Создание модели нейросети

Архитектура BertClassifier:

- Слой входных эмбеддингов (input embeddings layer): преобразует входные данные в векторы фиксированной длины, используя предобученную модель BERT.
- Скрытый слой (Dropout): с коэффициентом 0.2 для предотвращения переобучения.
- Скрытый слой (Linear): содержит 64 нейронов и функцию активации ReLU.
- Скрытый слой (Dropout): с коэффициентом 0.2.
- Выходной слой (Linear): содержит 6 нейронов (по числу классов).

```
class BertClassifier(nn.Module):

    def __init__(self, bert_model_name, dropout=0.2):

        super(BertClassifier, self).__init__()

        self.bert = BertModel.from_pretrained(bert_model_name)
        self.dropout1 = nn.Dropout(dropout)
        self.fc1 = nn.Linear(768, 256)
        self.relu = nn.ReLU()
        self.dropout2 = nn.Dropout(dropout)
        self.fc2 = nn.Linear(256, 6)

    def forward(self, input_ids, attention_mask):

        outputs = self.bert(input_ids=input_ids,
                            attention_mask=attention_mask)
        pooled_output = outputs.pooler_output
        pooled_output = self.dropout1(pooled_output)
        output = self.fc1(pooled_output)
        output = self.relu(output)
        output = self.dropout2(output)
        logits = self.fc2(output)

        return logits
```

Рис. 11: BertClassifier

5. Обучение модели

```python
def plot_train_history(train_losses, train_f1_scores, val_losses, val_f1_scores):
    """
    Функция отрисовки процесса обучения модели
    """
    plt.figure(figsize=(16, 6))

    # Loss на обучающей выборке
    plt.subplot(1, 2, 1)
    plt.plot(train_losses, label='Train Loss')
    plt.plot(val_losses, label='Validation Loss')
    plt.xlabel('Epoch')
    plt.ylabel('Loss')
    plt.title('Train and Validation Loss')
    plt.legend()

    # F1-мера на обучающей и валидационной выборках
    plt.subplot(1, 2, 2)
    plt.plot(train_f1_scores, label='Train F1 Score')
    plt.plot(val_f1_scores, label='Validation F1 Score')
    plt.xlabel('Epoch')
    plt.ylabel('F1 Score')
    plt.title('Train and Validation F1 Score')
    plt.legend()

    plt.show()
```

Рис. 12: Функция отрисовки процесса обучения

```python
def generate_classification_report(model, eval_dataloader, device):
    """
    Функция выводящая отчет по всей модели

    """

    y_true = []  # Список для хранения истинных меток
    y_pred = []  # Список для хранения предсказанных меток
    y_scores = []  # Список для хранения вероятностей для ROC-AUC

    # Переводим модель в режим оценки
    model.eval()

    with torch.no_grad():
        for data in eval_dataloader:
            # переносим данные на device
            input_ids = data['input_ids'].to(device)
            attention_mask = data['attention_mask'].to(device)
            labels = data['emotion'].to(device)

            # получаем предсказанные значения

            outputs = model(input_ids, attention_mask)

            # Применяем софтмакс для получения вероятностей классов
            probabilities = torch.softmax(outputs, dim=1)

            # Получаем предсказанные классы как индексы с наибольшей вероятностью
            _, predicted = torch.max(probabilities, 1)

            # Преобразуем тензоры в списки
            y_true.extend(labels.cpu().numpy())
            y_pred.extend(predicted.cpu().numpy())

            # Получаем вероятности для ROC-AUC
            y_scores.extend(probabilities.cpu().numpy())

    # Генерируем отчет с использованием classification_report
    report = classification_report(y_true, y_pred)

    # Calculate micro-average ROC-AUC
    micro_fpr, micro_tpr, _ = roc_curve(np.array(y_true), np.array(y_scores)[:, 1], pos_label=1)
    micro_roc_auc = auc(micro_fpr, micro_tpr)
```

Рис. 13: Функция вывода отчета report() часть 1

```
# Calculate macro-average ROC-AUC
n_classes = len(np.unique(y_true))
macro_fpr = dict()
macro_tpr = dict()
macro_roc_auc = dict()
for i in range(n_classes):
    macro_fpr[i], macro_tpr[i], _ = roc_curve(np.array(y_true) == i, np.array(y_scores)[:, i])
    macro_roc_auc[i] = auc(macro_fpr[i], macro_tpr[i])

# Calculate macro-average ROC-AUC by averaging over classes
macro_roc_auc = np.mean(list(macro_roc_auc.values()))

# Print ROC-AUC scores
print(f'Micro-average ROC-AUC: {micro_roc_auc}')
print(f'Macro-average ROC-AUC: {macro_roc_auc}')

plt.figure(figsize=(8, 6))
lw = 2
for i in range(n_classes):
    plt.plot(macro_fpr[i], macro_tpr[i], lw=lw, label=f'Class {i} (area = {macro_roc_auc:0.2f})')

plt.plot([0, 1], [0, 1], color='navy', lw=lw, linestyle='--')
plt.xlim([0.0, 1.0])
plt.ylim([0.0, 1.05])
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')
plt.title('Receiver Operating Characteristic (ROC) Curve')
plt.legend(loc="lower right")
plt.show()

return report
```

Рис. 14: Функция вывода отчета report() часть 2

```python
def calculate_f1_score(preds, labels):
    pred_flat = torch.argmax(preds, dim=1).cpu().numpy()
    labels_flat = labels.cpu().numpy()
    f1 = f1_score(labels_flat, pred_flat, average='weighted')
    return f1

def train_and_evaluate(model, train_data_loader, val_data_loader, optimizer, criterion, device, num_epochs):
    model.to(device)

    train_losses = []  # Список для хранения потерь на обучающем наборе данных
    val_losses = []    # Список для хранения потерь на валидационном наборе данных
    train_f1_scores = []  # Список для хранения F1-оценок на обучающем наборе данных
    val_f1_scores = []    # Список для хранения F1-оценок на валидационном наборе данных

    for epoch in range(num_epochs):
        model.train()
        train_loss = 0.0
        train_f1 = 0.0

        for batch in tqdm(train_data_loader):
            input_ids = batch['input_ids'].to(device)
            attention_mask = batch['attention_mask'].to(device)
            labels = batch['emotion'].to(device)

            optimizer.zero_grad()
            outputs = model(input_ids, attention_mask)
            loss = criterion(outputs, labels)
            loss.backward()
            optimizer.step()

            train_loss += loss.item()
            train_f1 += calculate_f1_score(outputs, labels)

        avg_train_loss = train_loss / len(train_data_loader)
        avg_train_f1 = train_f1 / len(train_data_loader)

        model.eval()
        val_loss = 0.0
        val_f1 = 0.0

        with torch.no_grad():
            for batch in tqdm(val_data_loader):
                input_ids = batch['input_ids'].to(device)
                attention_mask = batch['attention_mask'].to(device)
                labels = batch['emotion'].to(device)

                outputs = model(input_ids, attention_mask)
                loss = criterion(outputs, labels)
```

Рис. 15: train_and_evaluate() часть 1

```
            val_loss += loss.item()
            val_f1 += calculate_f1_score(outputs, labels)

    avg_val_loss = val_loss / len(val_data_loader)
    avg_val_f1 = val_f1 / len(val_data_loader)

    train_losses.append(avg_train_loss)
    val_losses.append(avg_val_loss)
    train_f1_scores.append(avg_train_f1)
    val_f1_scores.append(avg_val_f1)

    print(f'Epoch {epoch + 1}/{num_epochs}')
    print(f'Train Loss: {avg_train_loss:.4f} | Train F1 Score: {avg_train_f1:.4f}')
    print(f'Val Loss: {avg_val_loss:.4f} | Val F1 Score: {avg_val_f1:.4f}')
    print('-' * 50)
# Визуализация результатов
plot_train_history(train_losses, train_f1_scores, val_losses, val_f1_scores)

report = generate_classification_report(model, val_data_loader, device)
print(report)

return model
```

Рис. 16: train_and_evaluate() часть 2

Подготовка к обучнию:

Функция train_and_evaluate выполняет обучение и оценку модели машинного обучения. Входные параметры включают в себя модель (model), загрузчики данных для обучения и валидации (train_data_loader и val_data_loader), оптимизатор (optimizer), функцию потерь (criterion), устройство (device), и количество эпох (num_epochs).

На каждой эпохе функции выполняется следующие действия:

1) Обучение модели:

Модель переводится в режим обучения (model.train()). Для каждого батча в обучающем загрузчике данных, модель делает прямой проход, вычисляя потери (loss) и обновляя веса с помощью оптимизатора. Потери и F1-оценки для обучающего набора данных накапливаются.

2) Расчет средних метрик на обучающем наборе:

Суммарные потери и F1-оценки деленые на количество батчей дает средние потери и F1-оценки для обучающего набора данных.

3) Оценка на валидационном наборе данных:

Модель переводится в режим оценки (model.eval()). Для каждого батча в валидационном загрузчике данных, модель делает прямой проход, вычисляя

потери и F1-оценки для валидационного набора данных.

4) Расчет средних метрик на валидационном наборе:

Суммарные потери и F1-оценки для валидационного набора данных деленые на количество батчей дает средние потери и F1-оценки для валидационного набора данных.

5) Сохранение метрик и весов модели:

Значения средних потерь и F1-оценок для обучающей и валидационной выборок сохраняются в соответствующие списки (train_losses, val_losses, train_f1_scores, и val_f1_scores) для последующей визуализации. Веса модели сохраняются на каждой эпохе.

6) Вывод информации:

Для каждой эпохи выводятся потери и F1-оценки для обучающего и валидационного наборов данных.

7) Визуализация процесса обучения:

Функция plot_train_history используется для визуализации процесса обучения, отображая графики потерь и F1-оценок на обучающем и валидационном наборах данных на протяжении всех эпох.

8) Генерация отчета о классификации и ROC-AUC оценок:

Функция generate_classification_report генерирует отчет о классификации и вычисляет микро- и макро-средние ROC-AUC оценки для модели.

9) Возвращение обученной модели и метрик:

В конце функции, она возвращает обученную модель, что позволяет использовать её для последующего прогнозирования, и списки метрик для анализа результатов обучения.

Обучение:

Создадим модель классификатора на основе предобученной BERT-модели ('bert-base-uncased'). Создадим веса классов, так как датасет несбалансированный. Определим функцию потерь (кросс-энтропию) и устройство, на котором будет выполняться обучение (если доступна видеокарта - CUDA, иначе CPU).

Задаём оптимизатор Adam для обновления весов модели. Переместим модель на выбранное устроство.

Далее начинаем обучение, вызвав функцию train_and_evaluate() с использо-

ванием переданных параметров.

В результате выполнения этой строки кода будует получена обученная модель bert, выведен процесс обучения и отчет по обученной модели.

```
model = BertClassifier('bert-base-uncased', dropout=0.5)

# определяем устройство
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')

 # создадим веса для каждого класса
class_weights_tensor = torch.FloatTensor([class_weights[i] for i in range(len(class_weights))]).to(device)

# используем Кросс энтропию для задачи многоклассовой классификации
criterion = nn.CrossEntropyLoss(weight=class_weights_tensor)

# определяем оптимизатор
optimizer = torch.optim.Adam(model.parameters(), lr=LEARNING_RATE)

# перемещаем model на device
model.to(device)

bert = train_and_evaluate(model, train_loader, val_loader, optimizer, criterion, device, EPOCHS)
```

Рис. 17: Обучение

Зададим гиперпарметры: EPOCHS = 16, LEARNING_RATE = 1e-05 Обученная модель достигает метрики F1-score на тренировочной выборке 98.7% и на валидационной 87.2%.

Посмотрим на процесс обучения. Видим что loss на train и val падает, значит модель действительно обучается. F1-score растёт и разрыв между значениями F1-score на train и val не такой уж и большой, значит удалось избежать переобучения.

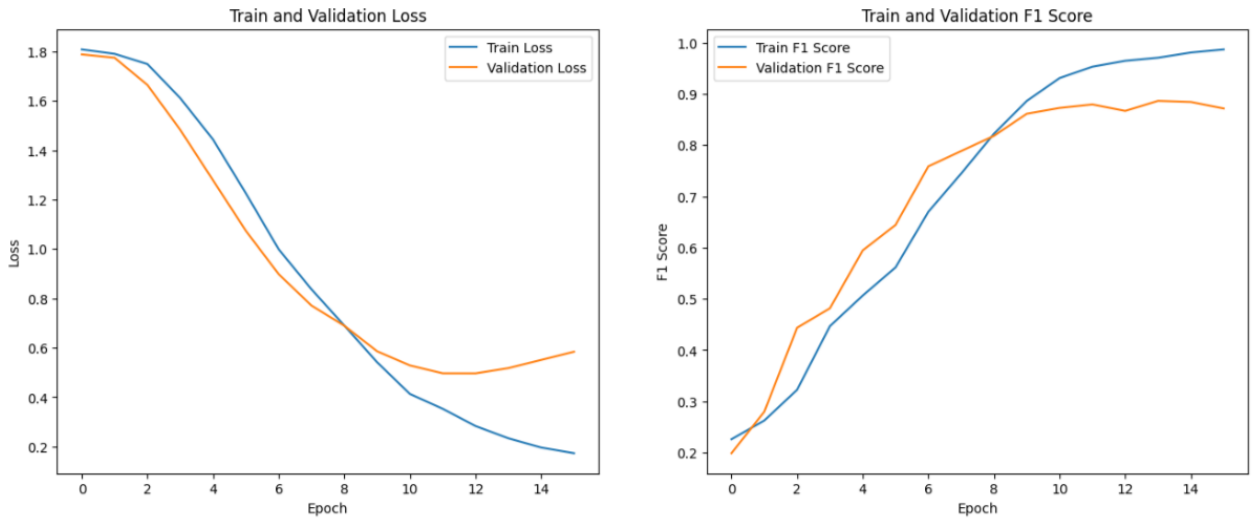
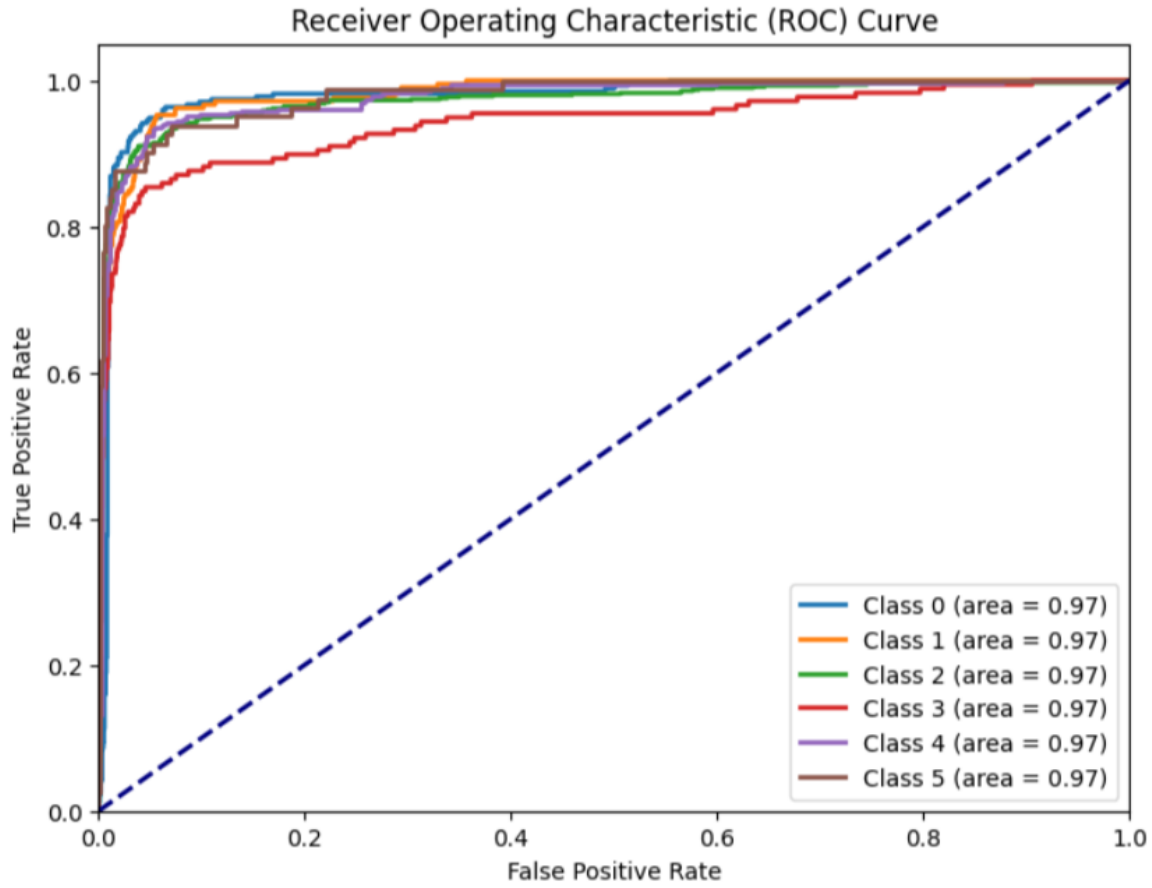


Рис. 18: Процесс обучения

```
Micro-average ROC-AUC: 0.98118351905787
Macro-average ROC-AUC: 0.9716444283011554
```



	precision	recall	f1-score	support
0	0.74	0.95	0.83	275
1	0.76	0.85	0.80	212
2	0.93	0.91	0.92	704
3	0.87	0.70	0.78	178
4	0.95	0.85	0.89	550
5	0.80	0.83	0.81	81
accuracy			0.87	2000
macro avg	0.84	0.85	0.84	2000
weighted avg	0.88	0.87	0.87	2000

Рис. 19: Подробный отчет по метрикам

6. Оценка результатов

Для оценки результатов, напишем функцию predict(), принимающую обученную модель (model) и тестовую выбоку.

```python
def predict(model, test_data):
    # Создадим EmotionDataset и DataLoader на основе test_data
    test = EmotionDataset(test_data, tokenizer, MAX_LEN)
    test_dataloader = DataLoader(test, batch_size=VALID_BATCH_SIZE)

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

    if use_cuda:
        model = model.to(device)

    model.eval()

    pred = []
    target = []

    for data in tqdm(test_dataloader):
        input_ids = data['input_ids'].to(device)
        attention_mask = data['attention_mask'].to(device)
        labels = data['emotion'].to(device)
        target.append(labels)

        with torch.no_grad():
            outputs = model(input_ids, attention_mask)
            pred.append(outputs)

    # Копируем и объединяем предсказания
    pred = np.concatenate([t.cpu().numpy() for t in pred], axis=0)
    # Копируем и объединяем истинные метки
    target = np.concatenate([t.cpu().numpy() for t in target], axis=0)

    average_f1_score = calculate_f1_score(torch.tensor(pred), torch.tensor(target))
    report = classification_report(np.argmax(pred, axis=1), target)

    print(f'\n  F1 Score на тесте: {average_f1_score:.3f}')
    print(report)
```

Рис. 20: функция predict()

```
predict(bert, test_df)
```

```
100%|██████████| 125/125 [00:08<00:00, 15.56it/s]
  F1 Score на тесте: 0.978
              precision    recall  f1-score   support

           0       1.00      0.93      0.96       296
           1       0.99      0.98      0.98       227
           2       0.99      0.98      0.99       698
           3       0.95      0.97      0.96       155
           4       0.96      1.00      0.98       557
           5       1.00      0.99      0.99        67

    accuracy                           0.98      2000
   macro avg       0.98      0.97      0.98      2000
weighted avg       0.98      0.98      0.98      2000
```

Рис. 21: Оценка результатов на test

F1-score на тестовой выборке достиг значения 97.8% а Accuracy все 98% ! Для улучшения результатов необходимо провести дополнительные исследования и адаптировать модель к конкретным задачам.

7. Возможное использование

Данную нейросеть можно использовать в следующих задачах:

1. Мониторинг социальных медиа: нейросеть может анализировать тональность сообщений в социальных сетях, так компании могут анализировать отношение пользователей к их продуктам или услугам и принимать меры для улучшения своей репутации.

2. Анализ мнений в опросах: нейросеть может использоваться для анализа мнений в опросах. Это может помочь политическим кампаниям исследовать общественное мнение и принимать меры для улучшения своей стратегии.

3. Анализ обращений клиентов: нейросеть может использоваться для анализа обращений клиентов в службу поддержки. Это может помочь компаниям быстро выявлять проблемы и решать их.

Часть IV

Вывод

В ходе выполнения задачи классификации эмоций в текстах с использованием дообученной нейронной сети BERT на основе датасета «Emotions Dataset for NLP» была успешно разработана модель, демонстрирующая выдающуюся точность (f1-score) на уровне 97.8%. Эти результаты подчеркивают, что дообученные нейросети, вроде BERT, представляют собой эффективный инструмент для классификации эмоций в текстах.

Проект «Обработка текстов с помощью нейронных сетей. Задача классификации текста» приобретает большое значение для бизнеса и общества в целом, поскольку предоставляет средство эффективного анализа обширных объемов текстовых данных и поддерживает принятие информированных решений. Создание инструмента, способствующего ускорению и упрощению анализа текстовых данных, становится ключевым этапом в развитии технологий и повышении эффективности компаний и организаций. Разработка точной нейросетевой модели для классификации текстовых данных играет определяющую роль в успехе проекта и способствует значительному улучшению результатов анализа текстовых данных.

Список иллюстраций

Список таблиц

Список источников

[1] Клуб Теха Лекций от МФТИ

[2] Трансформер (модель машинного обучения)

[3] BERT — state-of-the-art языковая модель для 104 языков

[4] PyTorch

[5] Глубинное обучение для автоматической обработки текстов

[6] Векторное представление слов

[7] BERT для классификации русскоязычных текстов

Приложение

*** Transformers.** [2] Основным принципом работы Transformers является использование механизма внимания (attention mechanism), который позволяет модели фокусироваться на наиболее важных частях текста.

Архитектура Transformers состоит из нескольких слоев, каждый из которых содержит блоки кодировщика и декодировщика. Каждый блок кодировщика и декодировщика содержит многослойный перцептрон (multi-layer perceptron) и механизм внимания.

Процесс работы модели начинается с подачи текстовых данных на вход кодировщика. Затем модель использует механизм внимания для определения наиболее важных частей текста и создает векторное представление для каждого слова. Далее, эти векторные представления передаются на декодировщик, который генерирует ответ на основе полученных данных.

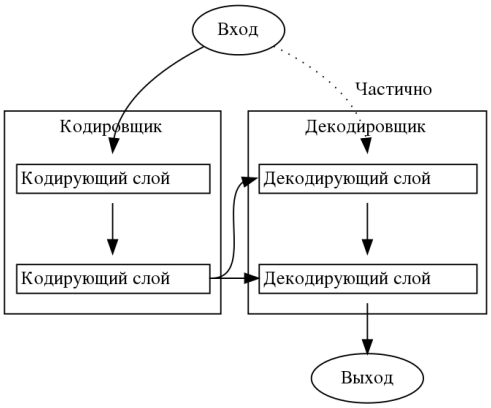


Рис. 22: Трансформер

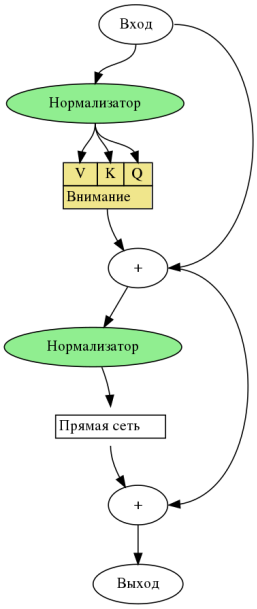


Рис. 23: Кодирующий слой

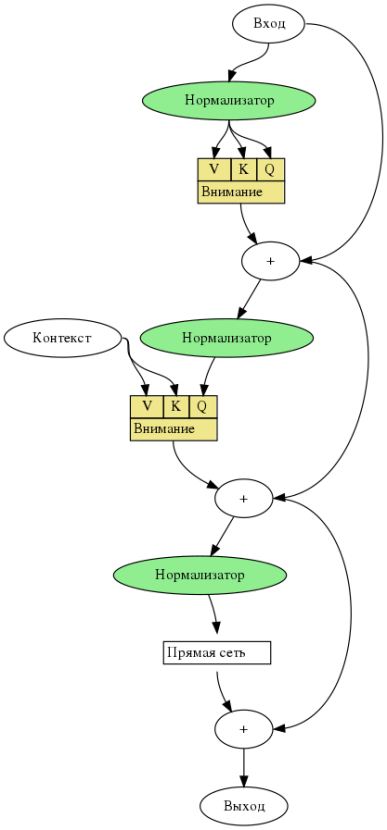


Рис. 24: Декодирующий слой