Федеральное государственное автономное
образовательное учреждение высшего образования
«Пермский государственный национальный исследовательский университет»
(ПГНИУ)
Региональный институт непрерывного образования (РИНО ПГНИУ)
Цифровая кафедра

Выпускная аттестационная (квалификационная) работа
по курсу профессиональной переподготовки «Анализ данных»

ОБНАРУЖЕНИЕ СПАМ-СООБЩЕНИЙ С ПОМОЩЬЮ МЕТОДОВ МАШИННОГО ОБУЧЕНИЯ

Разработчики проекта:
Черепанова Людмила Алексеевна,
Коневских Екатерина Сергеевна,
Савиных Елизавета Владимировна

Пермь, 2023

Оглавление

ПАСПОРТ ПРОЕКТА

Название проекта: Обнаружение спам-сообщений с помощью методов машинного обучения.

Сведения об авторах: Савиных Елизавета Владимировна, Коневских Екатерина Сергеевна, Черепанова Людмила Алексеевна.

Цель: разработка и внедрение алгоритмов, способных автоматически классифицировать входящие сообщения по двум категориям: "спам" и "не спам".

Задачи:

1. Выполнить анализ проблемы, обосновать ее актуальность.
2. Осуществить загрузку, очистку и обработку текстового набора данных.
3. Подготовить данные, провести анализ длин и распределение символов в сообщениях.
4. Построить модель с использованием Bag of Words, с помощью логистической регрессии классифицировать сообщения, визуализировать векторные представления для анализа распределения классов, оценить модель.
5. Обучить модель Word2Vec на данных для получения векторных представлений слов, построить свёрточную нейронную сеть (CNN) для анализа текстов, экспериментировать с архитектурами и гиперпараметрами для оптимизации модели.
6. Сравнить результаты CNN с логистической регрессией, чтобы выявить наиболее подходящую модель для задачи классификации.
7. Выполнить интерпретацию полученных результатов и сделать выводы о достижении цели.

Краткое описание проекта:

Проект направлен на разработку модели с использованием методов обработки естественного языка (NLP) для классификации сообщений как спам или не

спам. В процессе работы планируется подготовить данные, проанализировать длины сообщений, построить модель на основе векторного представления с использованием TF-IDF и Word2Vec, а также провести обучение нейронной сети с применением сверхточных нейронных сетей (CNN) для достижения наилучших результатов классификации.

Конкретные ожидаемые результаты:

Создание эффективной модели для классификации сообщений как спам или не спам с использованием методов NLP.

СОДЕРЖАНИЕ ПРОЕКТА

Анализ проблемы исследования

В настоящее время человек четко понимает актуальность проблемы спама, который стал неотъемлемой частью цифровой реальности. Увеличение объема информации и развитие технологий значительно облегчили процесс рассылки сообщений, но вместе с тем привели к распространению нежелательной и часто вредоносной информации.

Спам (англ. spam) - массовая, неперсонифицированная рассылка с использованием специальных программ, коммерческой, политической и иной рекламы или иного вида сообщений людям, не выразившим желания их получать.[1] Рассылка спама может привести к штрафам, блокировке домена или IP-адреса, а также к репутационным потерям.

Для России общий ущерб всех жертв спама превысил 200 млн. долларов в год. Однако эта сумма несопоставима с тем вредом, который данная отрасль интернет-индустрии наносит обществу в целом. Спам представляет серьезную угрозу также для информационной безопасности систем, используемых как в государственном, так и в частном секторе. Не меньший вред наносят спамеры имиджу интернет-рекламы и Интернета в целом, препятствуя развитию перспективных технологий. В силу чрезвычайно высокой прибыльности спамерского бизнеса технические средства борьбы со спамом не всегда достигают своей цели — спамеры без конца изобретают все новые и новые способы обхода фильтров.[2]

Решение данной проблемы одно — бороться со спамом, пытаясь если не полностью его устранить, то хотя бы как-то сократить поток нежелательных для приема и прочтения писем. На помощь приходят современные технологии, а именно: машинное обучение, которое может эффективно бороться со спамом, используя методы классификации, которые позволяют выявлять и фильтровать нежелательные сообщения.

Использование машинного обучения для определения спама делает процесс более гибким и адаптивным, чем традиционные подходы, основанные на правилах, так как модели могут обучаться на реальных данных и улучшаться со временем.

Исходные данные

В данной работе проводится анализ текстового файла, содержащего 5574 строки, каждая из которых включает корректно определённый класс (спам/не спам), после которого следует текст самого сообщения. Ниже мы приводим несколько примеров:

Список колонок анализируемого набора данных:

1) **target –** класс сообщения (спам/не спам)**.**

   Классы:

   - **ham** – сообщение не является спамом.
   - **spam** – сообщение является спамом.

2) **text**– текстовое сообщение.

Необходимо проанализировать данные и определить возможно ли классифицировать сообщения на спам/не спам на основе текста сообщения.

Выдвинем гипотезу исследования: существует зависимость между присутствием определённых слов и фраз в тексте сообщения и его классификацией как спам или не спам. Мы можем использовать этот набор данных для построения модели, которая будет определять, является ли сообщение спамом на основе анализа его текстового содержания.

Реализация проекта

Этап 1. Подготовка данных к анализу

Загрузим данные в датафрейм и подключим все необходимые библиотеки, которые понадобятся в процессе работы:

```
import pandas as pd
import numpy as np
import sklearn
import keras
import nltk
import re
import codecs
import matplotlib.pyplot as plt
from wordcloud import WordCloud
from nltk.tokenize import RegexpTokenizer
from tensorflow.keras.preprocessing.text import Tokenizer
from  keras.preprocessing.sequence import pad_sequences
from keras.utils import to_categorical
from sklearn.model_selection import train_test_split
from sklearn.feature_extraction.text import CountVectorizer,
TfidfVectorizer
from sklearn.decomposition import PCA, TruncatedSVD
import matplotlib
import seaborn as sns
import matplotlib.patches as mpatches
from sklearn.linear_model import LogisticRegression
from  sklearn.metrics  import  accuracy_score,  f1_score,
precision_score, recall_score, r2_score, classification_report
import itertools
from sklearn.metrics import confusion_matrix
!python -m gensim.downloader --download word2vec-google-news-
300
import gensim
!pip install lime
from lime import lime_text
from sklearn.pipeline import make_pipeline
from lime.lime_text import LimeTextExplainer
import random
from collections import defaultdict
from tensorflow.keras.preprocessing.text  import Tokenizer
from keras.preprocessing.sequence import pad_sequences
from keras.utils import to_categorical
from  keras.layers  import  Dense,  Input,  Flatten,  Dropout,
Concatenate
from keras.layers import Conv1D, MaxPooling1D, Embedding
from keras.layers import LSTM, Bidirectional
from keras.models import Model
```

Загрузим набор данных из CSV-файла, удалим ненужные колонки, присвоим названия столбцам target и text:

```
url = 'spam.csv'
data = pd.read_csv(url,encoding="latin1")
data.drop(columns = ['Unnamed: 2', 'Unnamed: 3', 'Unnamed: 4'], inplace = True)
data.rename(columns = {'v1': 'target', 'v2': 'text'}, inplace = True)
display(data)
```

	target	text
0	ham	Go until jurong point, crazy.. Available only ...
1	ham	Ok lar... Joking wif u oni...
2	spam	Free entry in 2 a wkly comp to win FA Cup fina...
3	ham	U dun say so early hor... U c already then say...
4	ham	Nah I don't think he goes to usf, he lives aro...
...	...	...
5567	spam	This is the 2nd time we have tried 2 contact u...
5568	ham	Will Ì_ b going to esplanade fr home?
5569	ham	Pity, * was in mood for that. So...any other s...
5570	ham	The guy did some bitching but I acted like i'd...
5571	ham	Rofl. Its true to its name

5572 rows × 2 columns

Рисунок 1. Исходный датафрейм

Выведем статистическое описание данных. Узнаем количество уникальных значений и количество наиболее частых значений.

```
data.describe()
```

	target	text
count	5572	5572
unique	2	5169
top	ham	Sorry, I'll call later
freq	4825	30

Рисунок 2. Статистическое описание данных

Видно, что сообщений из категории “не спам” подавляющее большинство, так же есть сообщение, встречающееся 30 раз.

Повторяющиеся сообщения могут помешать обучению модели, поэтому удалим их.

```
data.duplicated().sum()
data = data.drop_duplicates(keep = 'first')
data.shape
```

В результате получили 5169 строк.

Стандартизируем текст. Приведем все слова к нижнему регистру. Для улучшения классификации, различные символы не были удалены.

```
def standardize_text(df, text_field):
    df[text_field] = df[text_field].str.lower()
    return df
questions = standardize_text(data, "text")
questions.to_csv("clean_df.csv")
display(questions)
```

Сохраним очищенный и подготовленный датасет.

```
clean_questions = pd.read_csv("clean_df.csv")
clean_questions.tail()
```

	Unnamed: 0	target	text
5164	5567	spam	this is the 2nd time we have tried 2 contact u...
5165	5568	ham	will ì_ b going to esplanade fr home?
5166	5569	ham	pity, * was in mood for that. so...any other s...
5167	5570	ham	the guy did some bitching but i acted like i'd...
5168	5571	ham	rofl. its true to its name

Рисунок 3. Подготовленный датасет

Построим диаграмму, которая позволит визуально оценить распределение классов наших данных.

```
# Group data by 'target' and plot
clean_questions.groupby('target').size().plot(
    kind='barh',
    color=sns.color_palette('coolwarm', n_colors=2)
# Remove top and right spines
plt.gca().spines[['top', 'right']].set_visible(False)
# Add labels and title
plt.xlabel('Count')
plt.ylabel('Category')
```

```
plt.title('Category Distribution')
# Display the plot
plt.show()
```

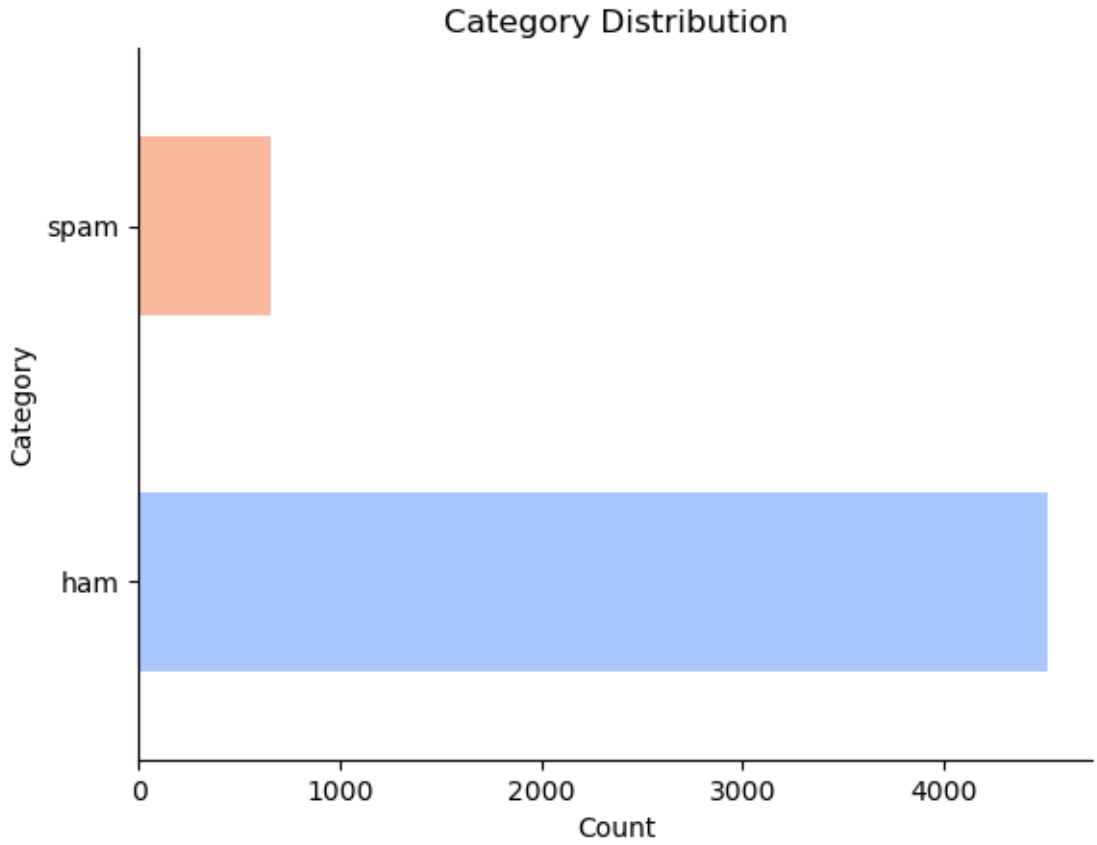

Рисунок 4. Распределение классов

По диаграмме видно, что наши классы не сбалансированы.

Балансировать мы их не будем, посмотрим, как обучится наша модель на несбалансированных данных.

Далее построим карту слов. Карта слов — это удобный способ увидеть, какие слова наиболее распространены в тексте. Он часто применяется в анализе текста, чтобы быстро получить представление о тематике и ключевых словах.

```
a = " ".join(clean_questions["text"])
en_cloud = WordCloud(max_words=100,
background_color='white',
                     width=800, height=400,
                     collocations=False).generate(a)
fig = plt.figure(figsize=(20, 10))
plt.imshow(en_cloud);
```

Рисунок 5. Карта слов

Как видно из нашей карты слов, чаще всего употребляются совершенно обычные слова, свойственные не только спам-сообщениям.

С помощью библиотеки NLTK очистим текст, разбив его на отдельные слова и сохраним результат в новый столбец tokens.

Этап 2. Подготовка данных к обучению

С помощью библиотеки NLTK очистим текст, разбив его на отдельные слова и сохраним результат в новый столбец tokens.

```
    tokenizer = RegexpTokenizer(r'\w+')
    clean_questions["tokens"] =
clean_questions["text"].apply(tokenizer.tokenize)
    clean_questions.head()
```

	Unnamed: 0	target	text	tokens
0	0	ham	go until jurong point, crazy.. available only ...	[go, until, jurong, point, crazy, available, o...
1	1	ham	ok lar... joking wif u oni...	[ok, lar, joking, wif, u, oni]
2	2	spam	free entry in 2 a wkly comp to win fa cup fina...	[free, entry, in, 2, a, wkly, comp, to, win, f...
3	3	ham	u dun say so early hor... u c already then say...	[u, dun, say, so, early, hor, u, c, already, t...
4	4	ham	nah i don't think he goes to usf, he lives aro...	[nah, i, don, t, think, he, goes, to, usf, he,...

Рисунок 6. Обновленный датасет

Выполним статистическую обработку текстовых данных: подсчет слов, определение размера словаря и предложение максимальной длины.

```
    all_words = [word for tokens in clean_questions["tokens"] for word in tokens]
    sentence_lengths = [len(tokens) for tokens in clean_questions["tokens"]]
    VOCAB = sorted(list(set(all_words)))
    print("%s words total, with a vocabulary size of %s" % (len(all_words), len(VOCAB)))
    print("Max sentence length is %s" % max(sentence_lengths))
```

В результате получили всего 82652 слова, при объеме словарного запаса 8713 и предложение максимальной длины, состоящее из 190 слов.

Создадим диаграмму, отображающую распределение длин предложений.

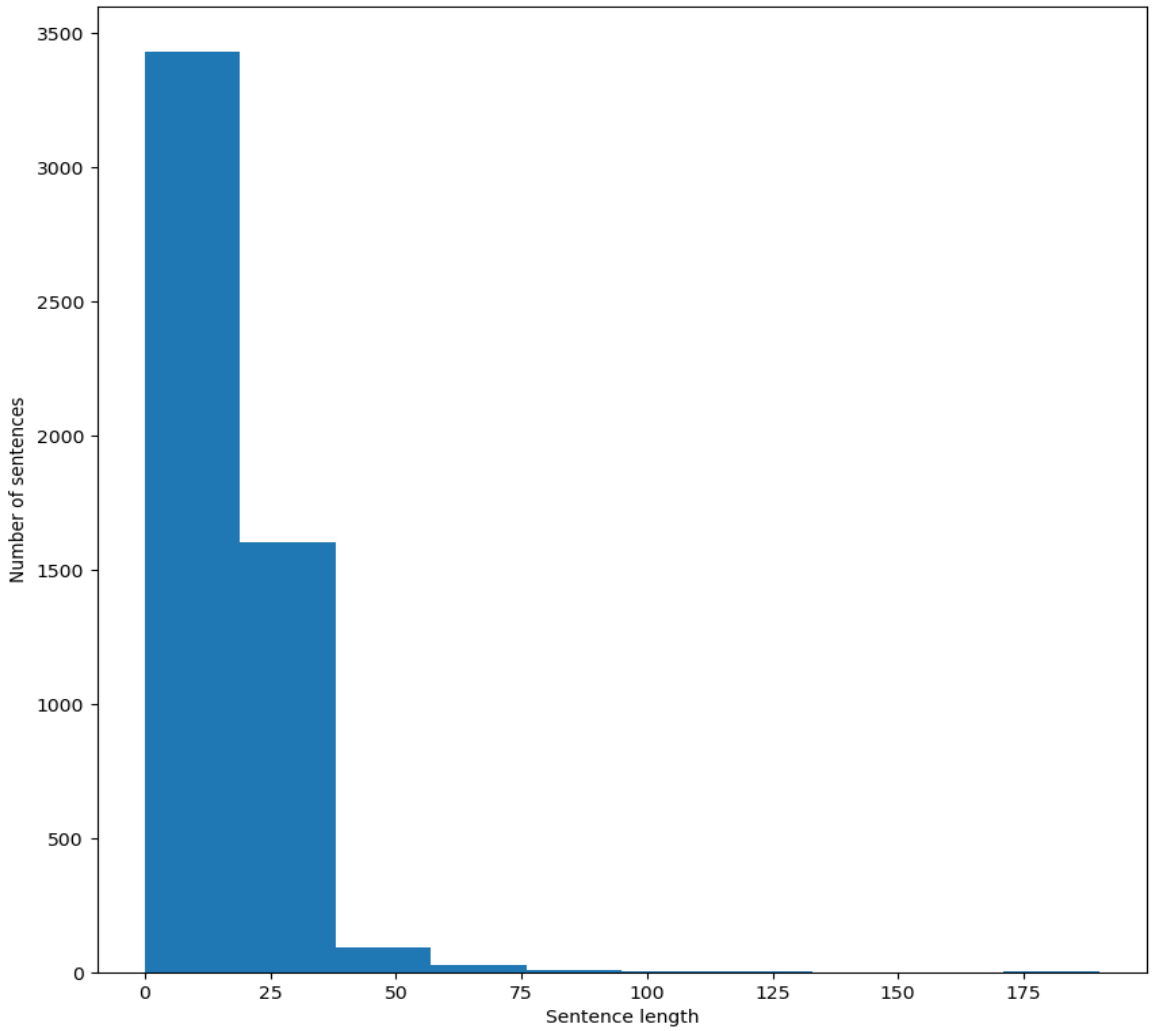

Рисунок 7. Диаграмма длин сообщений

В большинстве своём сообщения короткие, менее 20 слов. Максимальная длина сообщения - 190 слов.

Этап 3. Построение моделей

3.1. Bag of words

Самый простой подход, с которого мы можем начать, — это использовать модель мешка слов и применить к ней логистическую регрессию. Мешок слов - мы можем построить словарь всех уникальных слов в нашем датасете, и сопоставить уникальный индекс каждому слову в словаре. Каждое предложение тогда можно будет отобразить списком, длина которого равна числу уникальных слов в нашем словаре, а в каждом индексе в этом списке будет хранится, сколько раз данное слово встречается в предложении. То есть мешок слов просто связывает индекс с каждым словом в нашем словаре и встраивает каждое предложение в виде списка нулей, с 1 в каждом индексе, соответствующем слову, присутствующему в предложении.

```
def cv(data):
    count_vectorizer = CountVectorizer()
    emb = count_vectorizer.fit_transform(data)
    return emb, count_vectorizer

from sklearn.preprocessing import LabelEncoder
#Замена категориальных признаков на качественные
label_enc = LabelEncoder()
clean_questions["target"] =
label_enc.fit_transform(clean_questions["target"])
list_corpus = clean_questions["text"].tolist()
list_labels = clean_questions["target"].tolist()
```

Разобьём данные на обучающую и тестовую выборки, используя соотношение 8 к 2:

```
X_train, X_test, y_train, y_test =
train_test_split(list_corpus, list_labels, test_size=0.2,
random_state=40)

X_train_counts, count_vectorizer = cv(X_train)
X_test_counts = count_vectorizer.transform(X_test)
```

На обучающей выборке построим модель линейной регрессии, используя в качестве целевой и факторных переменных отобранные ранее столбцы.

Визуализируем векторные представления.

В словаре "Спам-сообщения" содержится около 9 000 слов. Это означает, что каждое предложение будет отражено вектором длиной 9 000. Этот вектор будет содержать преимущественно нули, поскольку каждое предложение содержит лишь малое подмножество из нашего словаря.

Для того, чтобы выяснить, захватывают ли наши векторные представления (embeddings), релевантную нашей задаче информацию (например, сообщение является спамом или нет), стоит попробовать визуализировать их и посмотреть, насколько хорошо разделены эти классы. Поскольку словари обычно являются очень большими и визуализация данных на 9 000 измерений невозможна, подходы вроде метода главных компонент (PCA) помогают спроецировать данные на два измерения.

```
    def plot_LSA(test_data, test_labels, savepath="PCA_demo.csv", plot=True):
            lsa = TruncatedSVD(n_components=2)
            lsa.fit(test_data)
            lsa_scores = lsa.transform(test_data)
            color_mapper = {label:idx for idx,label in enumerate(set(test_labels))}
            color_column = [color_mapper[label] for label in test_labels]
            colors = ['orange','blue']
            if plot:
                plt.scatter(lsa_scores[:,0], lsa_scores[:,1], s=8, alpha=.8, c=test_labels, cmap=matplotlib.colors.ListedColormap(colors))
                red_patch = mpatches.Patch(color='orange', label='ham')
                green_patch = mpatches.Patch(color='blue', label='spam')
                plt.legend(handles=[red_patch, green_patch], prop={'size': 30})


    fig = plt.figure(figsize=(16, 16))
    plot_LSA(X_train_counts, y_train)
    plt.show()
```

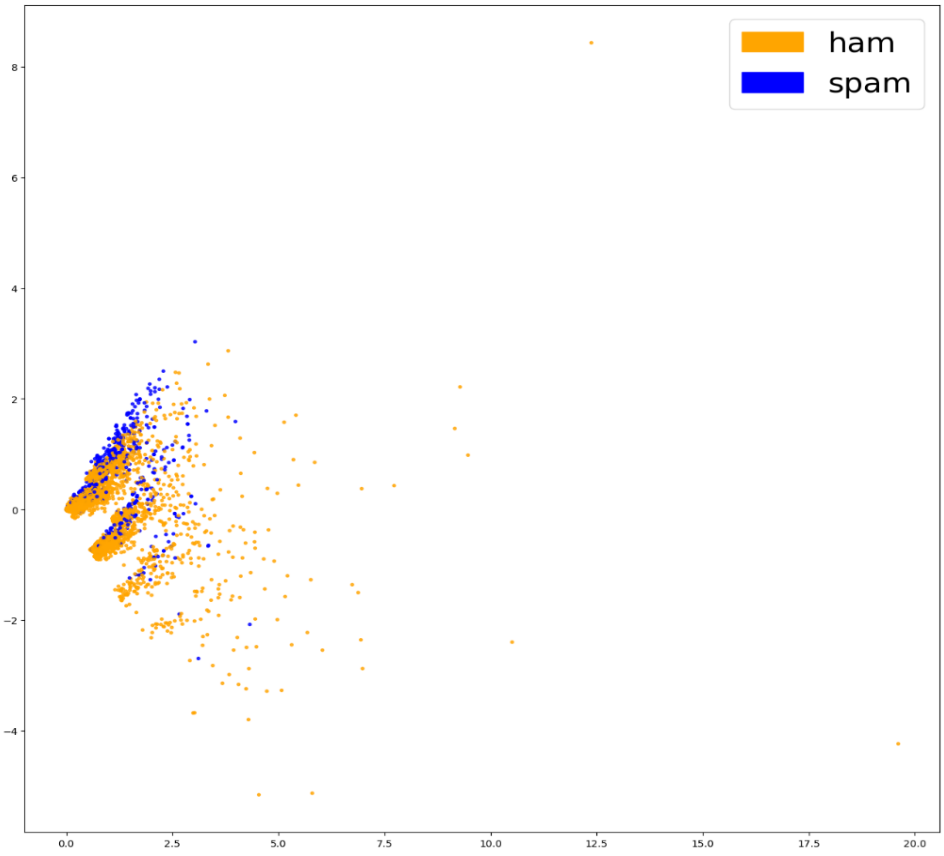

Рисунок 8. Разделение классов

Разделение не очень хорошее, попробуем применить логистическую регрессию.

```
    clf = LogisticRegression(C=30.0, class_weight='balanced',
solver='newton-cg', n_jobs=-1, random_state=40)
    clf.fit(X_train_counts, y_train)
    y_predicted_counts = clf.predict(X_test_counts)
    from sklearn.metrics import classification_report,
confusion_matrix
```

Вычислим метрики:

```
    def get_metrics(y_test, y_predicted):
        # true positives / (true positives+false positives)
        precision = precision_score(y_test, y_predicted,
pos_label=None,
                                    average='weighted')
        # true positives / (true positives + false negatives)
        recall = recall_score(y_test, y_predicted,
pos_label=None,
                              average='weighted')
```

```
        # harmonic mean of precision and recall
        f1 = f1_score(y_test, y_predicted, pos_label=None,
average='weighted')

        r2 = r2_score(y_test, y_predicted)
        # true positives + true negatives/ total
        accuracy = accuracy_score(y_test, y_predicted)
        return accuracy, precision, recall, f1, r2

    accuracy, precision, recall, f1, r2 = get_metrics(y_test,
y_predicted_counts)
    print("accuracy = %.3f, precision = %.3f, recall = %.3f, f1
= %.3f, r2 = %.3f" % (accuracy, precision, recall, f1,r2))


    accuracy = 0.985, precision = 0.984, recall = 0.985, f1 =
0.984, r2 = 0.847
```

Результат: accuracy = 0.985, precision = 0.984, recall = 0.985, f1 = 0.984, r2 = 0.847.

Модель демонстрирует исключительно высокую точность, точность (precision) и полноту (recall). Она очень надежна как в правильном определении положительных случаев, так и в избегании ложных срабатываний. Почти идеальная F1-мера подтверждает эту сбалансированную производительность.

Посмотрим на матрицу ошибок.

```
    def plot_confusion_matrix(cm, classes,
                              normalize=False,
                              title='Confusion matrix',
                              cmap=plt.cm.winter):
        if normalize:
            cm = cm.astype('float') / cm.sum(axis=1)[:,
np.newaxis]
        plt.imshow(cm, interpolation='nearest', cmap=cmap)
        plt.title(title, fontsize=30)
        plt.colorbar()
        tick_marks = np.arange(len(classes))
        plt.xticks(tick_marks, classes, fontsize=20)
        plt.yticks(tick_marks, classes, fontsize=20)

        fmt = '.2f' if normalize else 'd'
        thresh = cm.max() / 3.

        for i, j in itertools.product(range(cm.shape[0]),
range(cm.shape[1])):
```

```
            plt.text(j,     i,     format(cm[i,     j],     fmt),
horizontalalignment="center",
                     color="white"  if  cm[i,  j]  <  thresh  else
"black", fontsize=40)

        plt.tight_layout()
        plt.ylabel('True label', fontsize=30)
        plt.xlabel('Predicted label', fontsize=30)

        return plt

    cm = confusion_matrix(y_test, y_predicted_counts)
    fig = plt.figure(figsize=(10, 10))
    plot  =  plot_confusion_matrix(cm,  classes=['ham',  'spam'],
normalize=False, title='Confusion matrix')
    plt.show()
    print(cm)
```

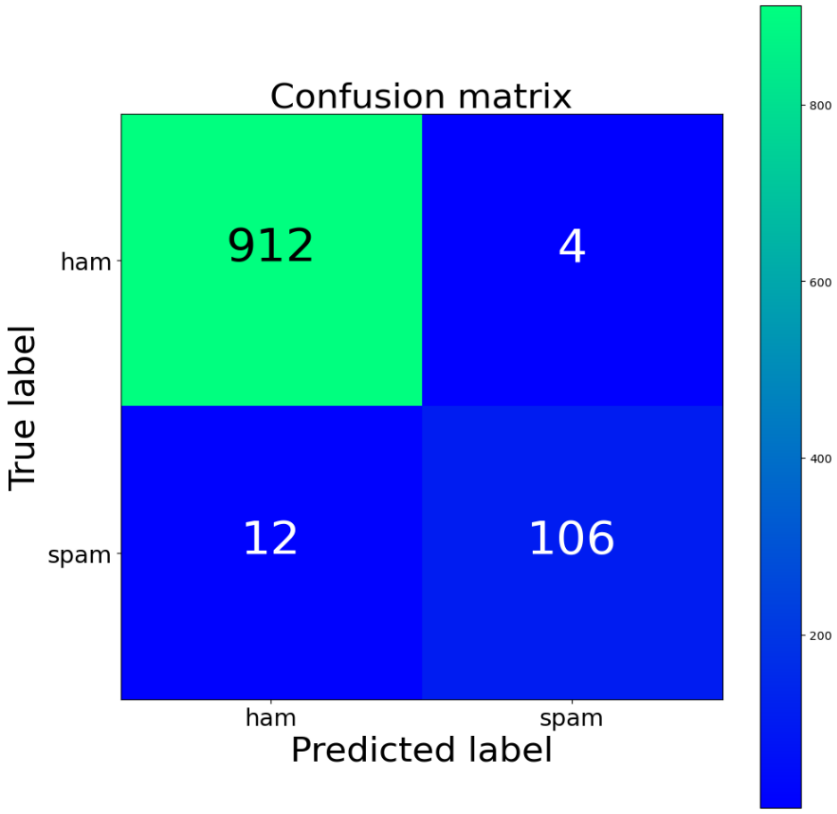

Рисунок 9. Матрица ошибок

Чтобы произвести валидацию нашей модели и интерпретировать ее предсказания, важно посмотреть на то, какие слова она использует для принятия решений. Если наши данные смещены, наш классификатор произведет точные предсказания на выборочных данных, но модель не сможет

достаточно хорошо обобщить их в реальном мире. На диаграмме ниже показаны наиболее значимые слова для классов спам-сообщений и обычных сообщений. Составление диаграмм, отражающих значимость слов, не составляет трудностей в случае использования «мешка слов» и логистической регрессии, поскольку мы просто извлекаем и ранжируем коэффициенты, которые модель использует для своих предсказаний.

Посмотрим, на каких признаках наш классификатор делает предсказание.

```
    def get_most_important_features(vectorizer, model, n=5):
      index_to_word = {v:k for k,v in
vectorizer.vocabulary_.items()}
    # loop for each class
      classes ={}
      for class_index in range(model.coef_.shape[0]):
        word_importances = [(el, index_to_word[i]) for i,el in
enumerate(model.coef_[class_index])]
        sorted_coeff = sorted(word_importances, key = lambda x :
x[0], reverse=True)
        tops = sorted(sorted_coeff[:n], key = lambda x : x[0])
        bottom = sorted_coeff[-n:]
        classes[class_index] = {
          'tops':tops,
          'bottom':bottom,
        }
      return classes

    importance = get_most_important_features(count_vectorizer,
clf, 10)
    def plot_important_words(top_scores, top_words,
bottom_scores, bottom_words, name):
        y_pos = np.arange(len(top_words))
        top_pairs = [(a,b) for a,b in zip(top_words, top_scores)]
        top_pairs = sorted(top_pairs, key=lambda x: x[1])

        bottom_pairs = [(a,b) for a,b in zip(bottom_words,
bottom_scores)]
        bottom_pairs = sorted(bottom_pairs, key=lambda x: x[1],
reverse=True)

        top_words = [a[0] for a in top_pairs]
        top_scores = [a[1] for a in top_pairs]

        bottom_words = [a[0] for a in bottom_pairs]
        bottom_scores = [a[1] for a in bottom_pairs]

        fig = plt.figure(figsize=(10, 10))
```

```
        plt.subplot(121)
        plt.barh(y_pos,bottom_scores, align='center', alpha=0.5)
        plt.title('ham', fontsize=20)
        plt.yticks(y_pos, bottom_words, fontsize=14)
        plt.suptitle('Key words', fontsize=16)
        plt.xlabel('Importance', fontsize=20)

        plt.subplot(122)
        plt.barh(y_pos,top_scores, align='center', alpha=0.5)
        plt.title('spam', fontsize=20)
        plt.yticks(y_pos, top_words, fontsize=14)
        plt.suptitle(name, fontsize=16)
        plt.xlabel('Importance', fontsize=20)

        plt.subplots_adjust(wspace=0.8)
        plt.show()

    top_scores = [a[0] for a in importance[0]['tops']]
    top_words = [a[1] for a in importance[0]['tops']]
    bottom_scores = [a[0] for a in importance[0]['bottom']]
    bottom_words = [a[1] for a in importance[0]['bottom']]
    plot_important_words(top_scores, top_words, bottom_scores,
bottom_words, "Most important words for relevance")
```

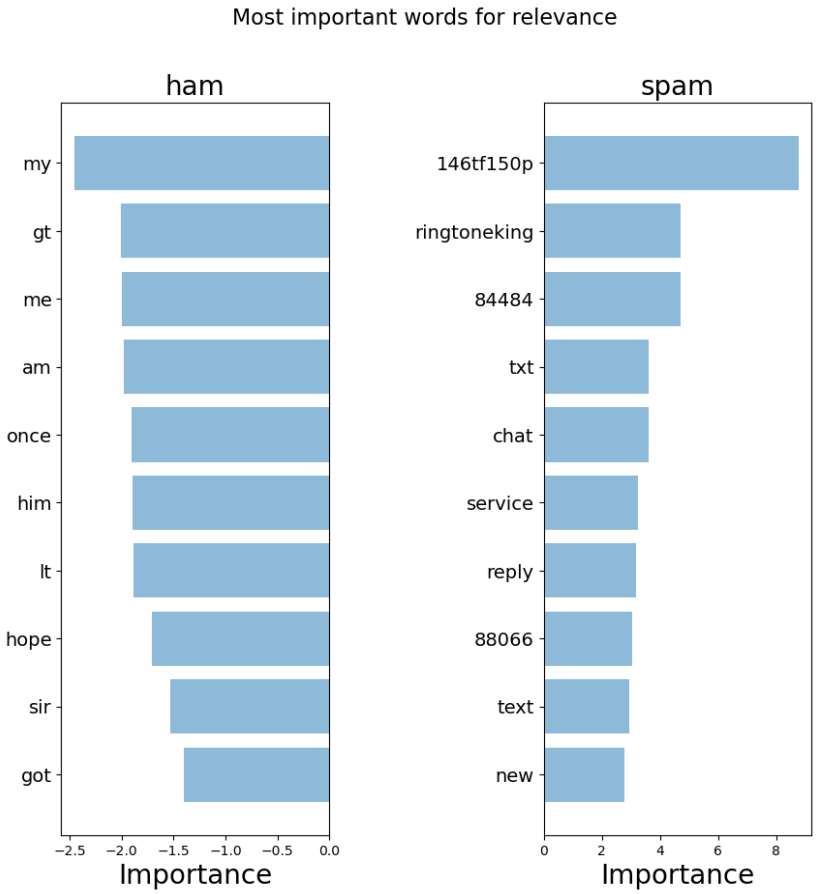

Рисунок 10. График важных слов.

3.2. TF-IDF

Итак, сейчас наш «мешок слов» имеет дело с огромным словарем из различных слов и все эти слова для него равнозначны. Однако, некоторые из этих слов встречаются очень часто, и лишь добавляют шума нашим предсказаниям. Поэтому далее мы постараемся найти способ представить предложения таким образом, чтобы они могли учитывать частоту слов, и посмотрим, сможем ли мы получить больше полезной информации из наших данных.

Чтобы помочь нашей модели сфокусироваться на значимых словах, мы можем использовать скоринг TF-IDF (Term Frequency, Inverse Document Frequency) поверх нашей модели «мешка слов». TF-IDF взвешивает на основании того, насколько они редки в нашем датасете, понижая в приоритете слова, которые встречаются слишком часто и просто добавляют шум. Ниже приводится проекция метода главных компонент, позволяющая оценить наше новое представление.

```
def tfidf(data):
    tfidf_vectorizer = TfidfVectorizer()

    train = tfidf_vectorizer.fit_transform(data)

    return train, tfidf_vectorizer

X_train_tfidf, tfidf_vectorizer = tfidf(X_train)
X_test_tfidf = tfidf_vectorizer.transform(X_test)
fig = plt.figure(figsize=(16, 16))
plot_LSA(X_train_tfidf, y_train)
plt.show()
```

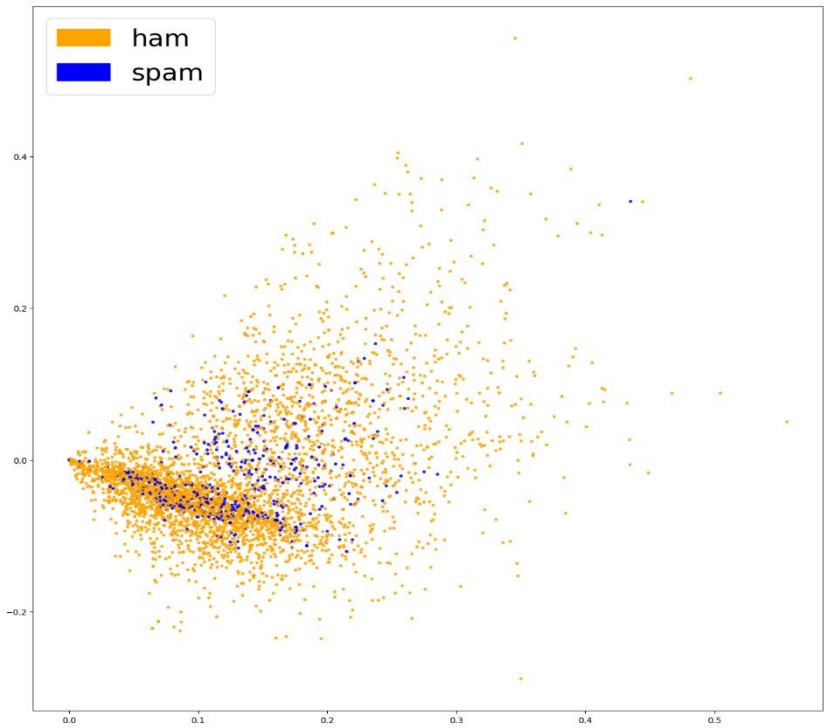

Рисунок 11. Разделение классов.

Разделение наших классов стало более чётким.

```
    clf_tfidf = LogisticRegression(C=30.0, class_weight='balanced', solver='newton-cg',
                                   multi_class='multinomial', n_jobs=-1, random_state=40)
    clf_tfidf.fit(X_train_tfidf, y_train)

    y_predicted_tfidf = clf_tfidf.predict(X_test_tfidf)

    accuracy_tfidf, precision_tfidf, recall_tfidf, f1_tfidf, r2_tfidf = get_metrics(y_test, y_predicted_tfidf)
    print("accuracy = %.3f, precision = %.3f, recall = %.3f, f1 = %.3f, r2 = %.3f" % (accuracy_tfidf, precision_tfidf, recall_tfidf, f1_tfidf, r2_tfidf))
```

Результат: accuracy = 0.986, precision = 0.987, recall = 0.986, f1 = 0.987, r2 = 0.866

Точность модели стала еще лучше!

```
    cm2 = confusion_matrix(y_test, y_predicted_tfidf)
    fig = plt.figure(figsize=(10, 10))
    plot = plot_confusion_matrix(cm2, classes=['spam','ham'], normalize=True, title='Confusion matrix')
    plt.show()
    print("TFIDF confusion matrix")
    print(cm2)
    print("BoW confusion matrix")
    print(cm)
```

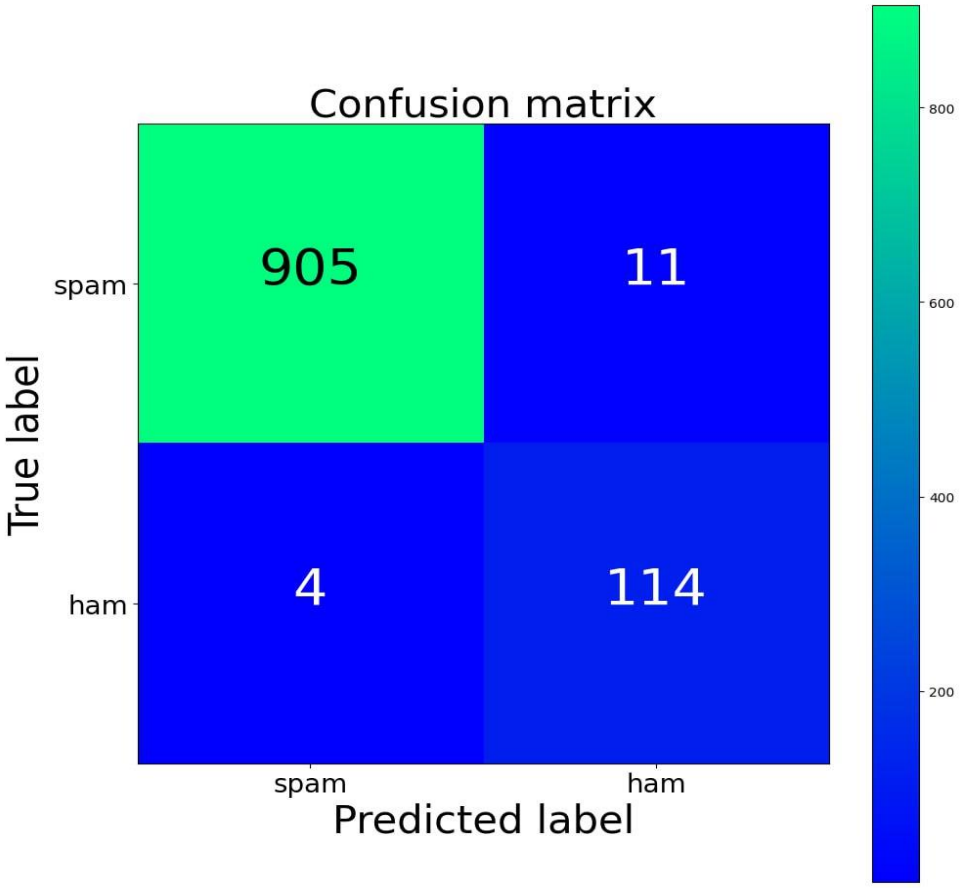

Рисунок 12. Матрица ошибок

Важность признаков для предсказания.

```
     importance_tfidf                                                =
get_most_important_features(tfidf_vectorizer, clf_tfidf, 10)
     top_scores = [a[0] for a in importance_tfidf[0]['tops']]
     top_words = [a[1] for a in importance_tfidf[0]['tops']]
     bottom_scores = [a[0] for a in importance_tfidf[0]['bottom']]
     bottom_words = [a[1] for a in importance_tfidf[0]['bottom']]

     plot_important_words(top_scores,  top_words,  bottom_scores,
bottom_words, "Most important words for relevance")
```

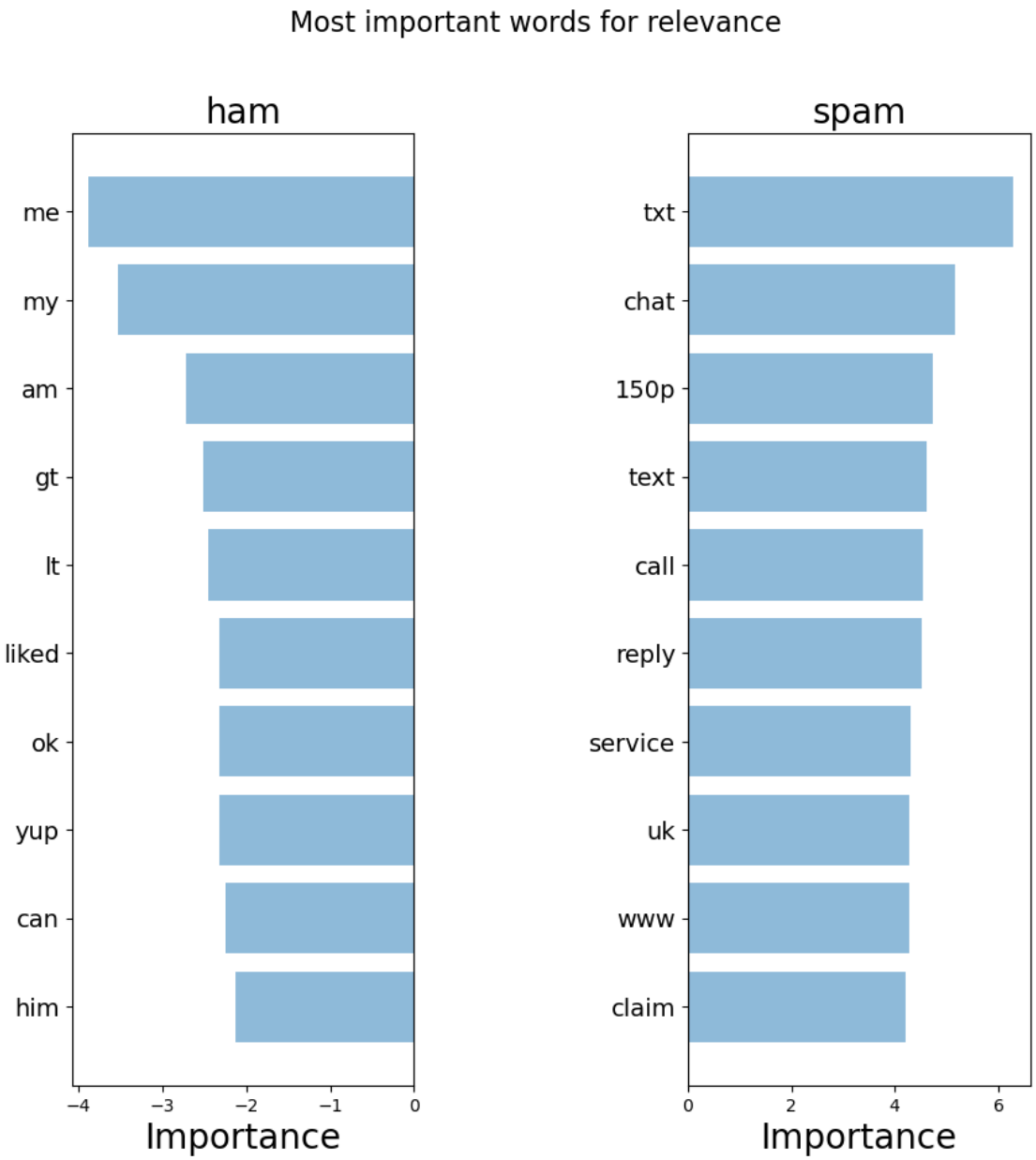

Рисунок 13. График важных слов.

Выбранные моделью слова действительно выглядят гораздо более релевантными. Несмотря на то, что метрики на нашем тестовом множестве увеличились совсем незначительно, у нас теперь гораздо больше уверенности в использовании модели в реальной системе, которая будет взаимодействовать с клиентами.

3.3. Применение семантики: Word2Vec

Наша последняя модель смогла «выхватить» слова, несущие наибольшее значение, однако в реальной жизни она может столкнуться со

словами, которые не встречались в обучающей выборке — и не сможет точно классифицировать сообщение, даже если она видела весьма похожие слова во время обучения.

Чтобы решить данную проблему, нам потребуется захватить семантическое (смысловое) значение слов — это означает, что для нас важно понимать, что слова «хороший» и «позитивный» ближе друг к другу, чем слова «абрикос» и «континент». Мы воспользуемся инструментом Word2Vec, который поможет нам сопоставить значения слов.

Word2Vec — это техника для поиска непрерывных отображений для слов. Word2Vec обучается на прочтении огромного количества текста с последующим запоминанием того, какое слово возникает в схожих контекстах. После обучения на достаточном количестве данных, Word2Vec генерирует вектор из 300 измерений для каждого слова в словаре, в котором слова со схожим значением располагаются ближе друг к другу.

Для того, чтобы внести знания о семантическом значении слов, мы воспользовались готовой моделью из статьи, находящейся в отрытом доступе, на тему непрерывных векторных представлений слов. Данная модель была предварительно обучена на очень большом объеме информации.

Загрузим модель Word2Vec и, как на предыдущим этапе, выполним понижение размерности данных с помощью TruncatedSVD и построим точечный график (scatter plot) для визуализации разделения классов.

```
    # Load Google's pre-trained Word2Vec model.
    word2vec_path     =     "~/gensim-data/word2vec-google-news-
300/word2vec-google-news-300.gz"
    word2vec                                                    =
gensim.models.KeyedVectors.load_word2vec_format(word2vec_path,
binary=True)
    def        get_average_word2vec(tokens_list,        vector,
generate_missing=False, k=300):
        if len(tokens_list)<1:
            return np.zeros(k)
        if generate_missing:
            vectorized = [vector[word] if word in vector else
np.random.rand(k) for word in tokens_list]
        else:
```

```
            vectorized = [vector[word] if word in vector else
np.zeros(k) for word in tokens_list]
        length = len(vectorized)
        summed = np.sum(vectorized, axis=0)
        averaged = np.divide(summed, length)
        return averaged

    def get_word2vec_embeddings(vectors, clean_questions,
generate_missing=False):
        embeddings = clean_questions['tokens'].apply(lambda x:
get_average_word2vec(x, vectors,

generate_missing=generate_missing))
        return list(embeddings)
    embeddings = get_word2vec_embeddings(word2vec,
clean_questions)
    X_train_word2vec, X_test_word2vec, y_train_word2vec,
y_test_word2vec = train_test_split(embeddings, list_labels,

test_size=0.2, random_state=40)
    fig = plt.figure(figsize=(16, 16))
    plot_LSA(embeddings, list_labels)
    plt.show()
```

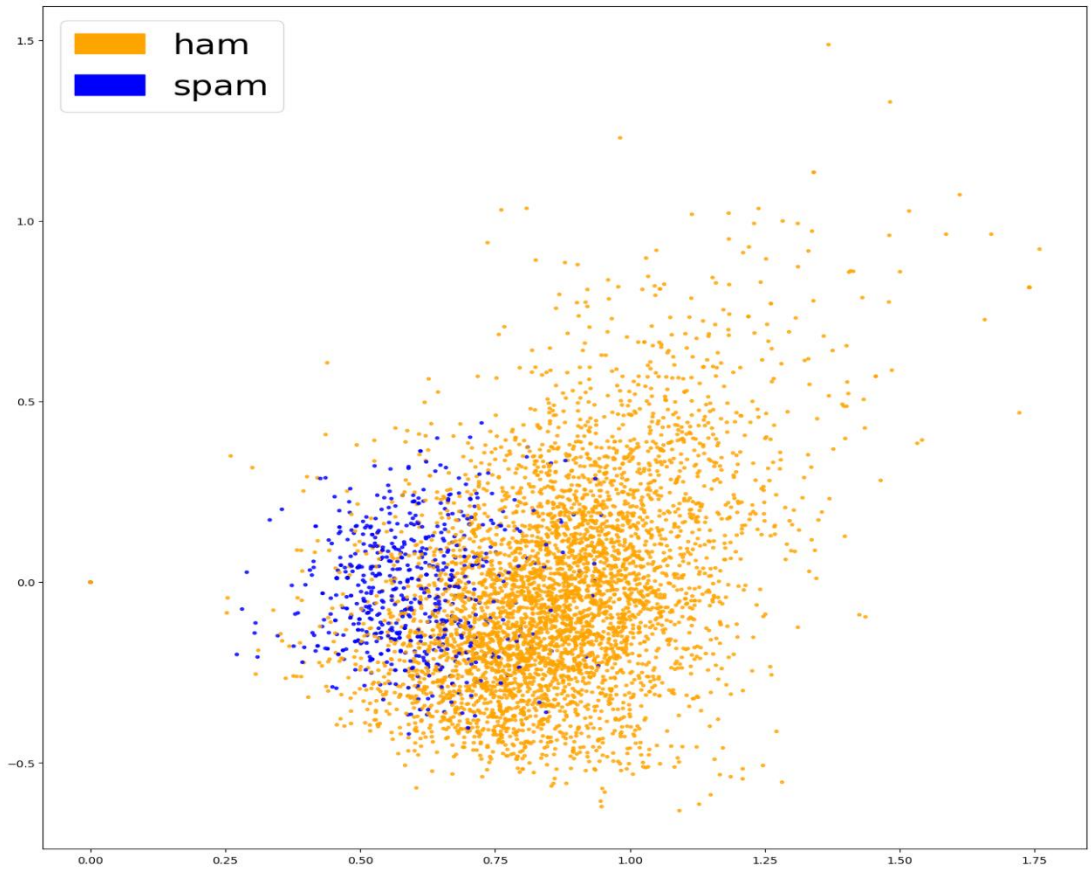

Рисунок 14. Разделение классов с использованием Word2Vec

Разделение очень четкое.

Обучим модель логистической регрессии на данных, представленных векторными эмбеддингами Word2Vec, и оценим её производительность.

```
    clf_w2v = LogisticRegression(C=30.0, class_weight='balanced', solver='newton-cg',
                                 multi_class='multinomial', random_state=40)
    clf_w2v.fit(X_train_word2vec, y_train_word2vec)
    y_predicted_word2vec = clf_w2v.predict(X_test_word2vec)
accuracy_word2vec, precision_word2vec, recall_word2vec, f1_word2vec, r2_word2vec = get_metrics(y_test_word2vec, y_predicted_word2vec)
    print("accuracy = %.3f, precision = %.3f, recall = %.3f, f1 = %.3f, r2 = %.3f" % (accuracy_word2vec, precision_word2vec, recall_word2vec, f1_word2vec, r2_word2vec))
```

Результат: accuracy = 0.949, precision = 0.959, recall = 0.949, f1 = 0.952, r2 = 0.493

Несмотря на то, что метрики ухудшились, у нас получилась хорошая обученная модель, которая может верно классифицировать сообщения.

Построим нормализованную матрицу ошибок (confusion matrix) для классификации.

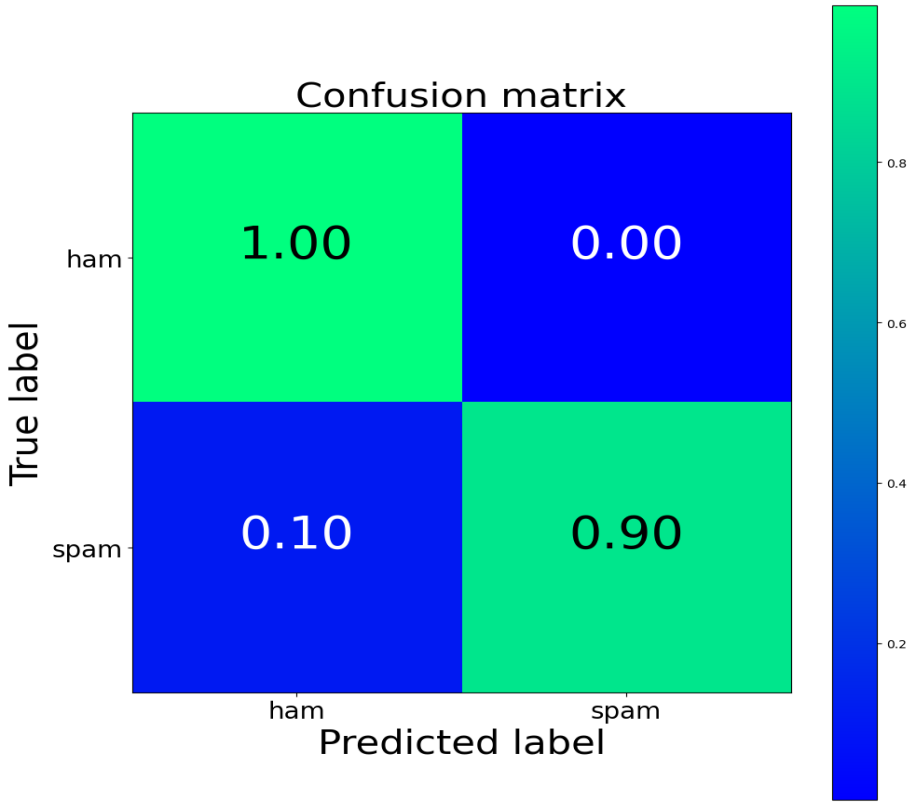

Рисунок 15. Матрица ошибок с использованием Word2Vec

Поскольку наши векторные представления более не в виде вектора с одним измерением на слово, как было в предыдущих моделях, теперь тяжелее понять, какие слова наиболее релевантны для нашей классификации. Несмотря на то, что мы по-прежнему обладаем доступом к коэффициентам нашей логистической регрессии, они относятся к 300 измерениям наших вложений, а не к индексам слов.

3.4. LIME

Для столь небольшого прироста точности, полная потеря возможности объяснить работу модели — это слишком жесткий компромисс. К счастью, при работе с более сложными моделями мы можем использовать интерпретаторы наподобие LIME, которые применяются для того, чтобы получить некоторое представление о том, как работает классификатор.

Напишем код, который выполняет объяснение предсказаний классификатора на основе модели Word2Vec, используя библиотеку Lime.

```
    train_data,   X_test_data,   y_train_data,   y_test_data   =
train_test_split(list_corpus, list_labels, test_size=0.2,

random_state=40)
    vector_store = word2vec
    def word2vec_pipeline(examples):
        global vector_store
        tokenizer = RegexpTokenizer(r'\w+')
        tokenized_list = []
        for example in examples:
            example_tokens = tokenizer.tokenize(example)
            vectorized_example                                    =
get_average_word2vec(example_tokens,                vector_store,
generate_missing=False, k=300)
            tokenized_list.append(vectorized_example)
        return clf_w2v.predict_proba(tokenized_list)

    c = make_pipeline(count_vectorizer, clf)
    def explain_one_instance(instance, class_names):
        explainer = LimeTextExplainer(class_names=class_names)
        exp        =        explainer.explain_instance(instance,
word2vec_pipeline, num_features=6)
        return exp

    def visualize_one_exp(features, labels, index, class_names =
["ham","spam"]):
        exp  =  explain_one_instance(features[index],  class_names
= class_names)
```

```
print('Index: %d' % index)
print('True class: %s' % class_names[labels[index]])
exp.show_in_notebook(text=True)
```

Давайте взглянем на пару объяснений для предложений из нашего датасета.

```
visualize_one_exp(X_test_data, y_test_data, 22)
```

Результат:

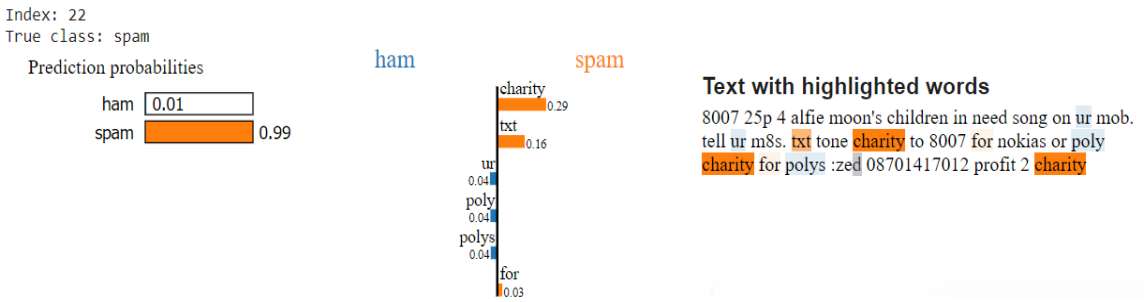

Рисунок 16. Объяснений предсказаний

```
visualize_one_exp(X_test_data, y_test_data, 60)
```

Результат:

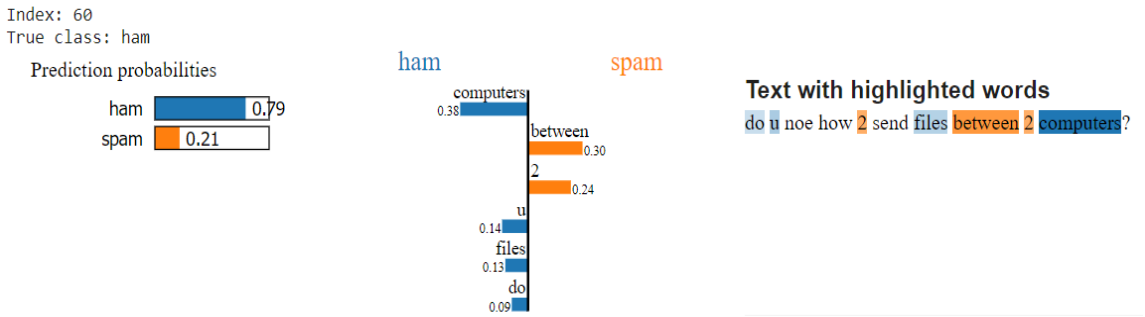

Рисунок 17. Объяснений предсказаний

Запустим LIME на репрезентативной выборке тестовых данных, и посмотрим, какие слова встречаются регулярно и вносят наибольший вклад в конечный результат. Используя данный подход, мы можем получить оценки значимости слов аналогично тому, как мы делали это для предыдущих моделей, и валидировать предсказания нашей модели.

```
random.seed(40)
```

```
def get_statistical_explanation(test_set, sample_size, word2vec_pipeline, label_dict):
```

```
        sample_sentences = random.sample(test_set, sample_size)
        explainer = LimeTextExplainer()

        labels_to_sentences = defaultdict(list)
        contributors = defaultdict(dict)

        # First, find contributing words to each class
        for sentence in sample_sentences:
            probabilities = word2vec_pipeline([sentence])
            curr_label = probabilities[0].argmax()
            labels_to_sentences[curr_label].append(sentence)
            exp      =      explainer.explain_instance(sentence,
word2vec_pipeline, num_features=6, labels=[curr_label])
            listed_explanation = exp.as_list(label=curr_label)

            for word,contributing_weight in listed_explanation:
                if word in contributors[curr_label]:

contributors[curr_label][word].append(contributing_weight)
                else:
                    contributors[curr_label][word]             =
[contributing_weight]

        # average each word's contribution to a class, and sort
them by impact
        average_contributions = {}
        sorted_contributions = {}
        for label,lexica in contributors.items():
            curr_label = label
            curr_lexica = lexica
            average_contributions[curr_label]                  =
pd.Series(index=curr_lexica.keys())
            for word,scores in curr_lexica.items():
                average_contributions[curr_label].loc[word]    =
np.sum(np.array(scores))/sample_size
            detractors                                         =
average_contributions[curr_label].sort_values()
            supporters                                         =
average_contributions[curr_label].sort_values(ascending=False)
            sorted_contributions[label_dict[curr_label]] = {
                'detractors':detractors,
                 'supporters': supporters
            }
        return sorted_contributions

    label_to_text = {
        0: 'ham',
        1: 'spam',
    }
    sorted_contributions                                       =
get_statistical_explanation(X_test_data, 100, word2vec_pipeline,
label_to_text)
    # First index is the class (Disaster)
```

```
    # Second index is 0 for detractors, 1 for supporters
    # Third is how many words we sample
    top_words                                                    =
sorted_contributions['spam']['supporters'][:15].index.tolist()
    top_scores                                                   =
sorted_contributions['ham']['supporters'][:15].tolist()
    bottom_words                                                 =
sorted_contributions['spam']['detractors'][:15].index.tolist()
    bottom_scores                                                =
sorted_contributions['ham']['detractors'][:15].tolist()

    plot_important_words(top_scores,  top_words,  bottom_scores,
bottom_words,"Most important words for relevance")
```

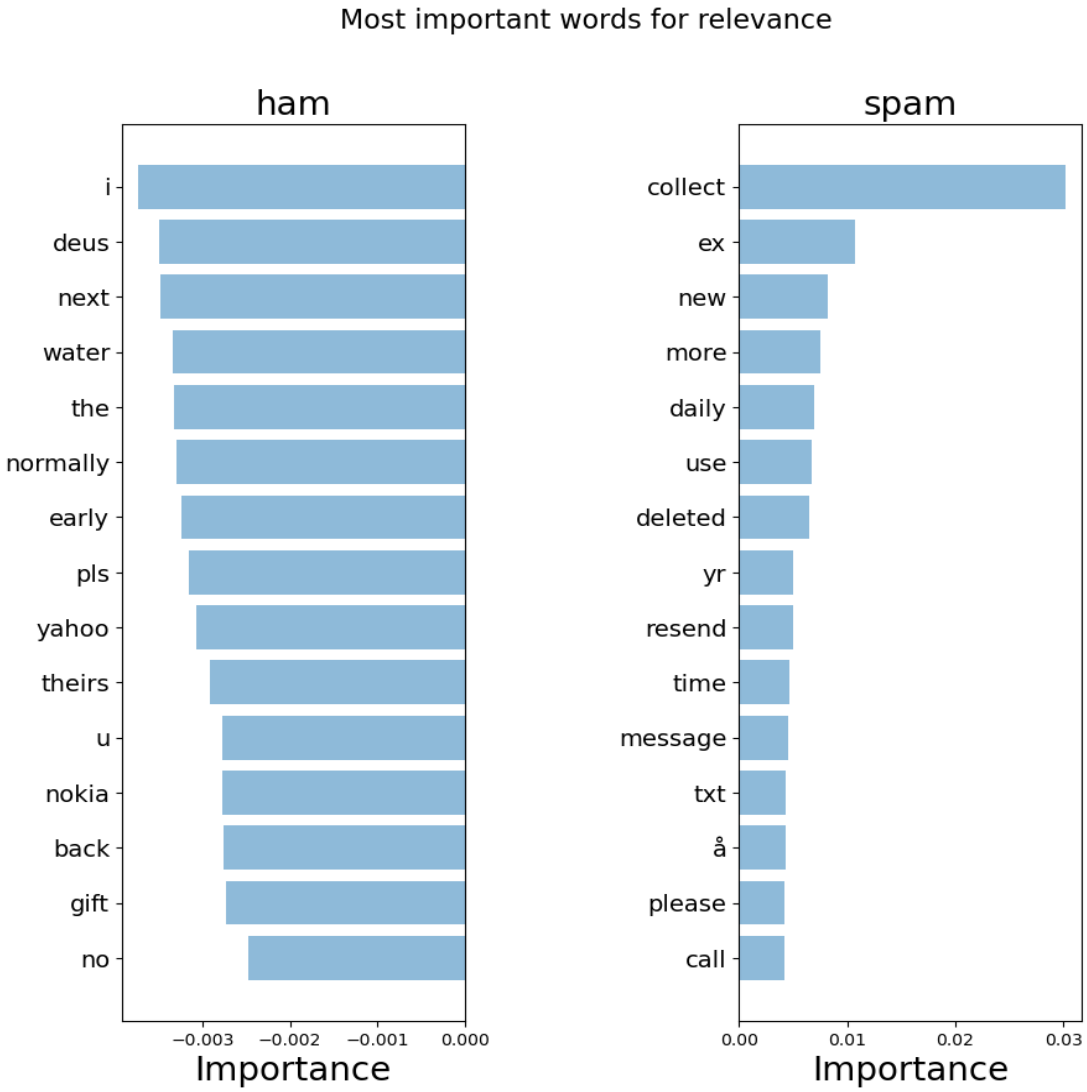

Рисунок 18. График важных слов с использованием Word2Vec и Lime

3.5. Использование синтаксиса при применении end-to-end подходов

Мы рассмотрели быстрые и эффективные подходы для генерации компактных векторных представлений предложений. Однако, опуская порядок слов, мы отбрасываем всю синтаксическую информацию из наших предложений. Для решение данной проблемы можно использовать более сложную модель, которая принимает целые выражения в качестве ввода и предсказывает метки, без необходимости построения промежуточного представления. Распространенный для этого способ состоит в рассмотрении предложения как последовательности индивидуальных векторов слов с использованием или Word2Vec, или более свежих подходов вроде GloVe или CoVe. Именно этим мы и займемся далее.

CNN для NLP

Сверточные нейронные сети для классификации предложений (CNN for Sentence Classification) обучаются очень быстро и могут сослужить отличную службу в качестве входного уровня в архитектуре глубокого обучения. Несмотря на то, что сверточные нейронные сети (CNN) в основном известны своей высокой производительностью на данных-изображениях, они показывают превосходные результаты при работе с текстовыми данными, и обычно гораздо быстрее обучаются, чем большинство сложных подходов NLP (например, LSTM-сети и архитектуры Encoder/Decoder). Эта модель сохраняет порядок слов и обучается ценной информации о том, какие последовательности слов служат предсказанием наших целевых классов. В отличии от предыдущих моделей, она в курсе существования разницы между фразами «Лёша ест растения» и «Растения едят Лёшу».

Выполним предобработку текстовых данных для использования в сверточной нейронной сети (CNN).

```
EMBEDDING_DIM = 300
MAX_SEQUENCE_LENGTH = 35
VOCAB_SIZE = len(VOCAB)

VALIDATION_SPLIT=.2
tokenizer = Tokenizer(num_words=VOCAB_SIZE)
tokenizer.fit_on_texts(clean_questions["text"].tolist())
```

```
    sequences                                                    =
tokenizer.texts_to_sequences(clean_questions["text"].tolist())

    word_index = tokenizer.word_index
    print('Found %s unique tokens.' % len(word_index))

    cnn_data             =             pad_sequences(sequences,
maxlen=MAX_SEQUENCE_LENGTH)
    labels                                                       =
to_categorical(np.asarray(clean_questions["target"]))

    indices = np.arange(cnn_data.shape[0])
    np.random.shuffle(indices)
    cnn_data = cnn_data[indices]
    labels = labels[indices]
    num_validation_samples     =     int(VALIDATION_SPLIT     *
cnn_data.shape[0])

    embedding_weights       =       np.zeros((len(word_index)+1,
EMBEDDING_DIM))
    for word,index in word_index.items():
        embedding_weights[index,:] = word2vec[word] if word in
word2vec else np.random.rand(EMBEDDING_DIM)
    print(embedding_weights.shape)
```

В результате получили 8920 уникальных значений.

Используем функцю ConvNet, которая создает модель CNN для классификации текстов, используя предобученные эмбеддинги.

```
    def   ConvNet(embeddings,   max_sequence_length,   num_words,
embedding_dim, labels_index, trainable=False, extra_conv=True):

        embedding_layer = Embedding(num_words,
                                    embedding_dim,
                                    weights=[embeddings],

input_length=max_sequence_length,
                                    trainable=trainable)

        sequence_input   =   Input(shape=(max_sequence_length,),
dtype='int32')
        embedded_sequences = embedding_layer(sequence_input)

        # Yoon Kim model (https://arxiv.org/abs/1408.5882)
        convs = []
        filter_sizes = [3,4,5]

        for filter_size in filter_sizes:
```

```
            l_conv              =              Conv1D(filters=128,
kernel_size=filter_size, activation='relu')(embedded_sequences)
            l_pool = MaxPooling1D(pool_size=3)(l_conv)
            convs.append(l_pool)

        l_merge = Concatenate(axis=1)(convs)

        # add a 1D convnet with global maxpooling, instead of
Yoon Kim model
        conv      =      Conv1D(filters=128,      kernel_size=3,
activation='relu')(embedded_sequences)
        pool = MaxPooling1D(pool_size=3)(conv)

        if extra_conv==True:
            x = Dropout(0.5)(l_merge)
        else:
            # Original Yoon Kim model
            x = Dropout(0.5)(pool)
        x = Flatten()(x)
        x = Dense(128, activation='relu')(x)
        #x = Dropout(0.5)(x)

        preds = Dense(labels_index, activation='softmax')(x)

        model = Model(sequence_input, preds)
        model.compile(loss='categorical_crossentropy',
                      optimizer='adam',
                      metrics=['acc'])

        return model
```

Теперь потренируем нашу нейросеть.

```
    x_train = cnn_data[:-num_validation_samples]
    y_train = labels[:-num_validation_samples]
    x_val = cnn_data[-num_validation_samples:]
    y_val = labels[-num_validation_samples:]

    model   =   ConvNet(embedding_weights,   MAX_SEQUENCE_LENGTH,
len(word_index)+1, EMBEDDING_DIM,

len(list(clean_questions["target"].unique())), False)
    model.fit(x_train, y_train, validation_data=(x_val, y_val),
epochs=3, batch_size=128)
```

В результате получили метрики качества модели.

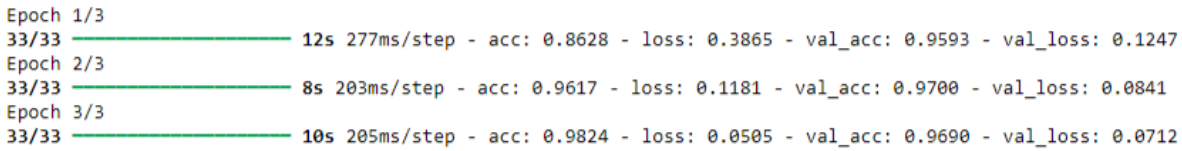

Рисунок 19. Точность модели CNN.

Выведем предсказания на валидационных данных и так же метрики качества.

```
y_pred = model.predict(x_val)
y_pred_cnn = np.argmax(y_pred, axis=1)
y_val_cnn = np.argmax(y_val, axis=1)
accuracy_cnn, precision_cnn, recall_cnn, f1_cnn, r2_cnn = get_metrics(y_val_cnn, y_pred_cnn)
print("accuracy = %.3f, precision = %.3f, recall = %.3f, f1 = %.3f, r2 = %.3f" % (accuracy_cnn, precision_cnn, recall_cnn, f1_cnn, r2_cnn))
```

Результат: accuracy = 0.969, precision = 0.969, recall = 0.969, f1 = 0.967, r2 = 0.731

Все показатели высокие, особенно точность, полнота и отзыв, которые практически идентичны и составляют около 0,969. Значение R2 равное 0,731 является достаточно хорошим показателем качества подгонки модели.

Для визуализации результатов построим матрицу ошибок.

```
cm_cnn = confusion_matrix(y_val_cnn, y_pred_cnn)
fig = plt.figure(figsize=(10, 10))
plot = plot_confusion_matrix(cm, classes=['ham','spam'], normalize=False, title='Confusion matrix')
plt.show()

print("CNN confusion matrix")
print(cm_cnn)
print("Word2Vec confusion matrix")
print(cm_w2v)
print("TFIDF confusion matrix")
print(cm2)
print("BoW confusion matrix")
print(cm)
```

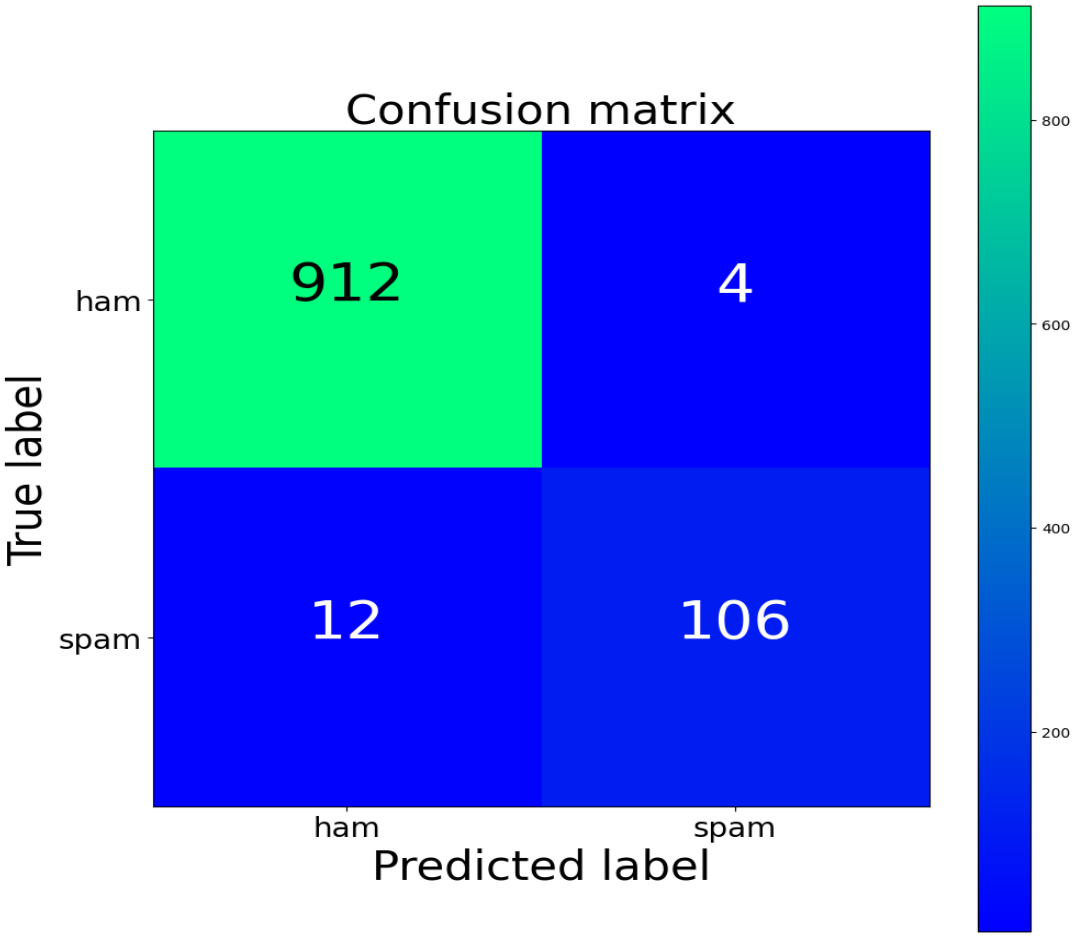

Рисунок 20. Точность модели CNN.

Заключение

В ходе работы были построены две модели: модель логистической регрессии и CNN.

Модель логистической регрессии была улучшена с помощью методов TF-IDF, Word2Vec и LIME. Высокая производительность (точность (accuracy) = 0.949 , полнота (precision) = 0.959, отзыв (recall) = 0.949 ,F1-мера = 0.952 ,R2-мера = 0.493) демонстрирует успешное применение модели логистической регрессии с векторными эмбеддингами Word2Vec для решения задачи многоклассовой классификации. Модель достигла высокой точности и хорошо обобщается на новых данных. Это свидетельствует как о качественных данных, так и о правильном выборе модели и метода представления данных.

С помощью модели CNN, так же удалось достичь высоких значений точности (accuracy = 0.969), полноты (precision = 0.969), отзыва(recall = 0.969) и меры(f1 = 0.967), r2 = 0.731, а также приемлемое значение R2 указывают на то, что модель хорошо обобщается и способна точно предсказывать результаты на новых, ранее невидимых данных. Это говорит о том, что модель успешно обучена и готова к применению на реальных данных.

Обе модели, логистическая регрессия и CNN, показали высокую точность классификации спама. Тем не менее, модель CNN обладает значительным преимуществом: в отличие от логистической регрессии, которая опирается на статистические связи между словами, CNN учитывает синтаксическую информацию, более эффективно распознавая спам-сообщения. Логистическая регрессия менее эффективна при распознавании спама, замаскированного сложными грамматическими конструкциями или завуалированным текстом. Поэтому модель CNN рекомендована для реальной классификации спама.

Таким образом, поставленная цель была достигнута. Были разработаны и внедрены алгоритмы, способные автоматически классифицировать входящие сообщения по двум категориям: "спам" и "не спам".

Для достижения цели были решены следующие задачи:

1. Выполнен анализ проблемы, обоснована ее актуальность.
2. Осуществлена загрузка, очистка и обработка текстового набора данных.
3. Подготовлены данные, проведен анализ длин и распределение символов в сообщениях.
4. Построена модель с использованием Bag of Words, с помощью логистической регрессии классифицированы сообщения, визуализированы векторные представления для анализа распределения классов, сделана оценка модели.
5. Обучена модель Word2Vec на данных для получения векторных представлений слов, построена свёрточная нейронная сеть (CNN) для анализа текстов.
6. Сравнены результаты CNN с логистической регрессией, чтобы выявить наиболее подходящую модель для задачи классификации.
7. Выполнена интерпретация полученных результатов и сделаны выводы о достижении цели.

Список использованных источников и литературы

1.РИА НОВОСТИ. URL: https://ria.ru/20090220/162696753.html (дата обращения: 9.12.2024).

2. Исследовательская работа по теме "Спам - проблема номер один в глобальной Сети". URL: https://nsportal.ru/ap/library/drugoe/2012/10/28/issledovatelskaya-rabota-po-teme-spam-problema-nomer-odin-v-globalnoy-seti (дата обращения: 10.12.2024).

3. Основы Natural Language Processing для текста: Блог компании Voximplant Python/Машинное обучение/Программирование. URL: https://habr.com/ru/companies/Voximplant/articles/446738/(дата обращения: 10.12.2024).

4. scikit - learn: Machine Learning in Python. URL: https://scikit- learn.org/stable/user_guide.html(дата обращения: 6.12.2024).

Приложения

Приложение 1

Программный код

```
     #pip install bleach
     #pip install keras
     #pip install tensorflow
     import pandas as pd
     import numpy as np
     import sklearn
     import keras
     import nltk
     import re
     import codecs
     import matplotlib.pyplot as plt
     from wordcloud import WordCloud
     from nltk.tokenize import RegexpTokenizer
     from tensorflow.keras.preprocessing.text import Tokenizer
     from  keras.preprocessing.sequence import pad_sequences
     from keras.utils import to_categorical
     from sklearn.model_selection import train_test_split
     from sklearn.feature_extraction.text import CountVectorizer,
TfidfVectorizer
     from sklearn.decomposition import PCA, TruncatedSVD
     import matplotlib
     import seaborn as sns
     import matplotlib.patches as mpatches
     from sklearn.linear_model import LogisticRegression
     from sklearn.metrics import accuracy_score, f1_score,
precision_score, recall_score, r2_score, classification_report
     import itertools
     from sklearn.metrics import confusion_matrix
     !python -m gensim.downloader --download word2vec-google-
news-300
     import gensim
     !pip install lime
     from lime import lime_text
     from sklearn.pipeline import make_pipeline
     from lime.lime_text import LimeTextExplainer
     import random
     from collections import defaultdict
     from tensorflow.keras.preprocessing.text  import Tokenizer
     from keras.preprocessing.sequence import pad_sequences
     from keras.utils import to_categorical
     from keras.layers import Dense, Input, Flatten, Dropout,
Concatenate
     from keras.layers import Conv1D, MaxPooling1D, Embedding
     from keras.layers import LSTM, Bidirectional
     from keras.models import Model
     url = 'spam.csv'
```

```
    data = pd.read_csv(url,encoding="latin1")
    data.drop(columns = ['Unnamed: 2', 'Unnamed: 3', 'Unnamed:
4'], inplace = True)
    data.rename(columns = {'v1': 'target', 'v2': 'text'},
inplace = True)
    display(data)
    data.tail()
    data.describe()
    data.duplicated().sum()
    data = data.drop_duplicates(keep = 'first')
    data.shape
    def standardize_text(df, text_field):
        df[text_field] = df[text_field].str.lower()
        return df

    questions = standardize_text(data, "text")

    questions.to_csv("clean_df.csv")
    display(questions)
    clean_questions = pd.read_csv("clean_df.csv")
    clean_questions.tail()
    clean_questions.groupby("target").count()
    # Group data by 'target' and plot
    clean_questions.groupby('target').size().plot(
        kind='barh',
        color=sns.color_palette('coolwarm', n_colors=2)  # Apply
'coolwarm' palette with 2 colors
    )

    # Remove top and right spines
    plt.gca().spines[['top', 'right']].set_visible(False)

    # Add labels and title
    plt.xlabel('Count')
    plt.ylabel('Category')
    plt.title('Category Distribution')

    # Display the plot
    plt.show()
    a = " ".join(clean_questions["text"])
    en_cloud = WordCloud(max_words=100,
background_color='white',
                         width=800, height=400,
                         collocations=False).generate(a)
    fig = plt.figure(figsize=(20, 10))
    plt.imshow(en_cloud);
    tokenizer = RegexpTokenizer(r'\w+')
    clean_questions["tokens"] =
clean_questions["text"].apply(tokenizer.tokenize)
    clean_questions.head()
    all_words = [word for tokens in clean_questions["tokens"]
for word in tokens]
```

```
    sentence_lengths = [len(tokens) for tokens in
clean_questions["tokens"]]
    VOCAB = sorted(list(set(all_words)))
    print("%s words total, with a vocabulary size of %s" %
(len(all_words), len(VOCAB)))
    print("Max sentence length is %s" % max(sentence_lengths))
    fig = plt.figure(figsize=(10, 10))
    plt.xlabel('Sentence length')
    plt.ylabel('Number of sentences')
    plt.hist(sentence_lengths)
    plt.show()
    def cv(data):
        count_vectorizer = CountVectorizer()
        emb = count_vectorizer.fit_transform(data)
        return emb, count_vectorizer

    from sklearn.preprocessing import LabelEncoder
    #Замена категориальных признаков на качественные
    label_enc = LabelEncoder()
    clean_questions["target"] =
label_enc.fit_transform(clean_questions["target"])
    list_corpus = clean_questions["text"].tolist()
    list_labels = clean_questions["target"].tolist()

    X_train, X_test, y_train, y_test =
train_test_split(list_corpus, list_labels, test_size=0.2,

random_state=40)

    X_train_counts, count_vectorizer = cv(X_train)
    X_test_counts = count_vectorizer.transform(X_test)
    def plot_LSA(test_data, test_labels,
savepath="PCA_demo.csv", plot=True):
            lsa = TruncatedSVD(n_components=2)
            lsa.fit(test_data)
            lsa_scores = lsa.transform(test_data)
            color_mapper = {label:idx for idx,label in
enumerate(set(test_labels))}
            color_column = [color_mapper[label] for label in
test_labels]
            colors = ['orange','blue']
            if plot:
                plt.scatter(lsa_scores[:,0], lsa_scores[:,1],
s=8, alpha=.8, c=test_labels,
cmap=matplotlib.colors.ListedColormap(colors))
                red_patch = mpatches.Patch(color='orange',
label='ham')
                green_patch = mpatches.Patch(color='blue',
label='spam')
                plt.legend(handles=[red_patch, green_patch],
prop={'size': 30})
```

```
    fig = plt.figure(figsize=(16, 16))
    plot_LSA(X_train_counts, y_train)
    plt.show()
    clf = LogisticRegression(C=30.0, class_weight='balanced',
solver='newton-cg', n_jobs=-1, random_state=40)
    clf.fit(X_train_counts, y_train)

    y_predicted_counts = clf.predict(X_test_counts)
    from sklearn.metrics import classification_report,
confusion_matrix
    print(pd.Series(y_train).value_counts(normalize=True)) #
Доля каждого класса в тренировочных данных
    print(pd.Series(y_test).value_counts(normalize=True))  #
Доля каждого класса в тестовых данных
    def get_metrics(y_test, y_predicted):
        # true positives / (true positives+false positives)
        precision = precision_score(y_test, y_predicted,
pos_label=None,
                                                average='weighted')
        # true positives / (true positives + false negatives)
        recall = recall_score(y_test, y_predicted,
pos_label=None,
                                    average='weighted')

        # harmonic mean of precision and recall
        f1 = f1_score(y_test, y_predicted, pos_label=None,
average='weighted')

        r2 = r2_score(y_test, y_predicted)
        # true positives + true negatives/ total
        accuracy = accuracy_score(y_test, y_predicted)
        return accuracy, precision, recall, f1, r2

    accuracy, precision, recall, f1, r2 = get_metrics(y_test,
y_predicted_counts)
    print("accuracy = %.3f, precision = %.3f, recall = %.3f, f1
= %.3f, r2 = %.3f" % (accuracy, precision, recall, f1,r2))
    def plot_confusion_matrix(cm, classes,
                              normalize=False,
                              title='Confusion matrix',
                              cmap=plt.cm.winter):
        if normalize:
            cm = cm.astype('float') / cm.sum(axis=1)[:,
np.newaxis]
        plt.imshow(cm, interpolation='nearest', cmap=cmap)
        plt.title(title, fontsize=30)
        plt.colorbar()
        tick_marks = np.arange(len(classes))
        plt.xticks(tick_marks, classes, fontsize=20)
        plt.yticks(tick_marks, classes, fontsize=20)

        fmt = '.2f' if normalize else 'd'
        thresh = cm.max() / 3.
```

```
        for i, j in itertools.product(range(cm.shape[0]),
range(cm.shape[1])):
            plt.text(j, i, format(cm[i, j], fmt),
horizontalalignment="center",
                     color="white" if cm[i, j] < thresh else
"black", fontsize=40)

        plt.tight_layout()
        plt.ylabel('True label', fontsize=30)
        plt.xlabel('Predicted label', fontsize=30)

        return plt
    cm = confusion_matrix(y_test, y_predicted_counts)
    fig = plt.figure(figsize=(10, 10))
    plot = plot_confusion_matrix(cm, classes=['ham', 'spam'],
normalize=False, title='Confusion matrix')
    plt.show()
    print(cm)
    def get_most_important_features(vectorizer, model, n=5):
      index_to_word = {v:k for k,v in
vectorizer.vocabulary_.items()}

      # loop for each class
      classes ={}
      for class_index in range(model.coef_.shape[0]):
        word_importances = [(el, index_to_word[i]) for i,el in
enumerate(model.coef_[class_index])]
        sorted_coeff = sorted(word_importances, key = lambda x :
x[0], reverse=True)
        tops = sorted(sorted_coeff[:n], key = lambda x : x[0])
        bottom = sorted_coeff[-n:]
        classes[class_index] = {
          'tops':tops,
          'bottom':bottom,
        }
      return classes

    importance = get_most_important_features(count_vectorizer,
clf, 10)
    def plot_important_words(top_scores, top_words,
bottom_scores, bottom_words, name):
        y_pos = np.arange(len(top_words))
        top_pairs = [(a,b) for a,b in zip(top_words,
top_scores)]
        top_pairs = sorted(top_pairs, key=lambda x: x[1])

        bottom_pairs = [(a,b) for a,b in zip(bottom_words,
bottom_scores)]
        bottom_pairs = sorted(bottom_pairs, key=lambda x: x[1],
reverse=True)

        top_words = [a[0] for a in top_pairs]
```

```
        top_scores = [a[1] for a in top_pairs]

        bottom_words = [a[0] for a in bottom_pairs]
        bottom_scores = [a[1] for a in bottom_pairs]

        fig = plt.figure(figsize=(10, 10))

        plt.subplot(121)
        plt.barh(y_pos,bottom_scores, align='center', alpha=0.5)
        plt.title('ham', fontsize=20)
        plt.yticks(y_pos, bottom_words, fontsize=14)
        plt.suptitle('Key words', fontsize=16)
        plt.xlabel('Importance', fontsize=20)

        plt.subplot(122)
        plt.barh(y_pos,top_scores, align='center', alpha=0.5)
        plt.title('spam', fontsize=20)
        plt.yticks(y_pos, top_words, fontsize=14)
        plt.suptitle(name, fontsize=16)
        plt.xlabel('Importance', fontsize=20)

        plt.subplots_adjust(wspace=0.8)
        plt.show()

    top_scores = [a[0] for a in importance[0]['tops']]
    top_words = [a[1] for a in importance[0]['tops']]
    bottom_scores = [a[0] for a in importance[0]['bottom']]
    bottom_words = [a[1] for a in importance[0]['bottom']]
    plot_important_words(top_scores, top_words, bottom_scores,
bottom_words, "Most important words for relevance")
    def tfidf(data):
        tfidf_vectorizer = TfidfVectorizer()

        train = tfidf_vectorizer.fit_transform(data)

        return train, tfidf_vectorizer

    X_train_tfidf, tfidf_vectorizer = tfidf(X_train)
    X_test_tfidf = tfidf_vectorizer.transform(X_test)
    fig = plt.figure(figsize=(16, 16))
    plot_LSA(X_train_tfidf, y_train)
    plt.show()
    clf_tfidf = LogisticRegression(C=30.0,
class_weight='balanced', solver='newton-cg',
                               multi_class='multinomial', n_jobs=-
1, random_state=40)
    clf_tfidf.fit(X_train_tfidf, y_train)

    y_predicted_tfidf = clf_tfidf.predict(X_test_tfidf)
    accuracy_tfidf, precision_tfidf, recall_tfidf, f1_tfidf,
r2_tfidf = get_metrics(y_test, y_predicted_tfidf)
```

```
     print("accuracy = %.3f, precision = %.3f, recall = %.3f, f1
= %.3f, r2 = %.3f"  % (accuracy_tfidf, precision_tfidf,
recall_tfidf, f1_tfidf, r2_tfidf))
     cm2 = confusion_matrix(y_test, y_predicted_tfidf)
     fig = plt.figure(figsize=(10, 10))
     plot = plot_confusion_matrix(cm2, classes=['spam','ham'],
normalize=False, title='Confusion matrix')
     plt.show()
     print("TFIDF confusion matrix")
     print(cm2)
     print("BoW confusion matrix")
     print(cm)
     importance_tfidf =
get_most_important_features(tfidf_vectorizer, clf_tfidf, 10)
     top_scores = [a[0] for a in importance_tfidf[0]['tops']]
     top_words = [a[1] for a in importance_tfidf[0]['tops']]
     bottom_scores = [a[0] for a in
importance_tfidf[0]['bottom']]
     bottom_words = [a[1] for a in importance_tfidf[0]['bottom']]

     plot_important_words(top_scores, top_words, bottom_scores,
bottom_words, "Most important words for relevance")
     # Load Google's pre-trained Word2Vec model.
     word2vec_path = "~/gensim-data/word2vec-google-news-
300/word2vec-google-news-300.gz"
     word2vec =
gensim.models.KeyedVectors.load_word2vec_format(word2vec_path,
binary=True)
     def get_average_word2vec(tokens_list, vector,
generate_missing=False, k=300):
         if len(tokens_list)<1:
             return np.zeros(k)
         if generate_missing:
             vectorized = [vector[word] if word in vector else
np.random.rand(k) for word in tokens_list]
         else:
             vectorized = [vector[word] if word in vector else
np.zeros(k) for word in tokens_list]
         length = len(vectorized)
         summed = np.sum(vectorized, axis=0)
         averaged = np.divide(summed, length)
         return averaged

     def get_word2vec_embeddings(vectors, clean_questions,
generate_missing=False):
         embeddings = clean_questions['tokens'].apply(lambda x:
get_average_word2vec(x, vectors,

generate_missing=generate_missing))
         return list(embeddings)
     embeddings = get_word2vec_embeddings(word2vec,
clean_questions)
```

```
     X_train_word2vec, X_test_word2vec, y_train_word2vec,
y_test_word2vec = train_test_split(embeddings, list_labels,

test_size=0.2, random_state=40)
     fig = plt.figure(figsize=(16, 16))
     plot_LSA(embeddings, list_labels)
     plt.show()
     clf_w2v = LogisticRegression(C=30.0,
class_weight='balanced', solver='newton-cg',
                                  multi_class='multinomial',
random_state=40)
     clf_w2v.fit(X_train_word2vec, y_train_word2vec)
     y_predicted_word2vec = clf_w2v.predict(X_test_word2vec)
     accuracy_word2vec, precision_word2vec, recall_word2vec,
f1_word2vec, r2_word2vec = get_metrics(y_test_word2vec,
y_predicted_word2vec)
     print("accuracy = %.3f, precision = %.3f, recall = %.3f, f1
= %.3f, r2 = %.3f" % (accuracy_word2vec, precision_word2vec,
recall_word2vec, f1_word2vec, r2_word2vec))
     cm_w2v = confusion_matrix(y_test_word2vec,
y_predicted_word2vec)
     fig = plt.figure(figsize=(10, 10))
     plot = plot_confusion_matrix(cm, classes=['ham','spam'],
normalize=True, title='Confusion matrix')
     plt.show()
     print("Word2Vec confusion matrix")
     print(cm_w2v)
     print("TFIDF confusion matrix")
     print(cm2)
     print("BoW confusion matrix")
     print(cm)
     _train_data, X_test_data, y_train_data, y_test_data =
train_test_split(list_corpus, list_labels, test_size=0.2,

random_state=40)
     vector_store = word2vec
     def word2vec_pipeline(examples):
         global vector_store
         tokenizer = RegexpTokenizer(r'\w+')
         tokenized_list = []
         for example in examples:
             example_tokens = tokenizer.tokenize(example)
             vectorized_example =
get_average_word2vec(example_tokens, vector_store,
generate_missing=False, k=300)
             tokenized_list.append(vectorized_example)
         return clf_w2v.predict_proba(tokenized_list)

     c = make_pipeline(count_vectorizer, clf)
     def explain_one_instance(instance, class_names):
         explainer = LimeTextExplainer(class_names=class_names)
         exp = explainer.explain_instance(instance,
word2vec_pipeline, num_features=6)
```

```
        return exp

    def visualize_one_exp(features, labels, index, class_names =
["ham","spam"]):
        exp = explain_one_instance(features[index], class_names
= class_names)
        print('Index: %d' % index)
        print('True class: %s' % class_names[labels[index]])
        exp.show_in_notebook(text=True)
    visualize_one_exp(X_test_data, y_test_data, 22)
    visualize_one_exp(X_test_data, y_test_data, 60)
    random.seed(40)

    def get_statistical_explanation(test_set, sample_size,
word2vec_pipeline, label_dict):
        sample_sentences = random.sample(test_set, sample_size)
        explainer = LimeTextExplainer()

        labels_to_sentences = defaultdict(list)
        contributors = defaultdict(dict)

        # First, find contributing words to each class
        for sentence in sample_sentences:
            probabilities = word2vec_pipeline([sentence])
            curr_label = probabilities[0].argmax()
            labels_to_sentences[curr_label].append(sentence)
            exp = explainer.explain_instance(sentence,
word2vec_pipeline, num_features=6, labels=[curr_label])
            listed_explanation = exp.as_list(label=curr_label)

            for word,contributing_weight in listed_explanation:
                if word in contributors[curr_label]:

contributors[curr_label][word].append(contributing_weight)
                else:
                    contributors[curr_label][word] =
[contributing_weight]

        # average each word's contribution to a class, and sort
them by impact
        average_contributions = {}
        sorted_contributions = {}
        for label,lexica in contributors.items():
            curr_label = label
            curr_lexica = lexica
            average_contributions[curr_label] =
pd.Series(index=curr_lexica.keys())
            for word,scores in curr_lexica.items():
                average_contributions[curr_label].loc[word] =
np.sum(np.array(scores))/sample_size
            detractors =
average_contributions[curr_label].sort_values()
```

```
            supporters =
average_contributions[curr_label].sort_values(ascending=False)
            sorted_contributions[label_dict[curr_label]] = {
                'detractors':detractors,
                 'supporters': supporters
            }
        return sorted_contributions

    label_to_text = {
        0: 'ham',
        1: 'spam',
    }
    sorted_contributions =
get_statistical_explanation(X_test_data, 100, word2vec_pipeline,
label_to_text)
    # First index is the class (Disaster)
    # Second index is 0 for detractors, 1 for supporters
    # Third is how many words we sample
    top_words =
sorted_contributions['spam']['supporters'][:15].index.tolist()
    top_scores =
sorted_contributions['ham']['supporters'][:15].tolist()
    bottom_words =
sorted_contributions['spam']['detractors'][:15].index.tolist()
    bottom_scores =
sorted_contributions['ham']['detractors'][:15].tolist()

    plot_important_words(top_scores, top_words, bottom_scores,
bottom_words,"Most important words for relevance")
    EMBEDDING_DIM = 300
    MAX_SEQUENCE_LENGTH = 35
    VOCAB_SIZE = len(VOCAB)

    VALIDATION_SPLIT=.2
    tokenizer = Tokenizer(num_words=VOCAB_SIZE)
    tokenizer.fit_on_texts(clean_questions["text"].tolist())
    sequences =
tokenizer.texts_to_sequences(clean_questions["text"].tolist())

    word_index = tokenizer.word_index
    print('Found %s unique tokens.' % len(word_index))

    cnn_data = pad_sequences(sequences,
maxlen=MAX_SEQUENCE_LENGTH)
    labels =
to_categorical(np.asarray(clean_questions["target"]))

    indices = np.arange(cnn_data.shape[0])
    np.random.shuffle(indices)
    cnn_data = cnn_data[indices]
    labels = labels[indices]
    num_validation_samples = int(VALIDATION_SPLIT *
cnn_data.shape[0])
```

```
    embedding_weights = np.zeros((len(word_index)+1,
EMBEDDING_DIM))
    for word,index in word_index.items():
        embedding_weights[index,:] = word2vec[word] if word in
word2vec else np.random.rand(EMBEDDING_DIM)
    print(embedding_weights.shape)
    def ConvNet(embeddings, max_sequence_length, num_words,
embedding_dim, labels_index, trainable=False, extra_conv=True):

        embedding_layer = Embedding(num_words,
                                    embedding_dim,
                                    weights=[embeddings],

input_length=max_sequence_length,
                                    trainable=trainable)

        sequence_input = Input(shape=(max_sequence_length,),
dtype='int32')
        embedded_sequences = embedding_layer(sequence_input)

        # Yoon Kim model (https://arxiv.org/abs/1408.5882)
        convs = []
        filter_sizes = [3,4,5]

        for filter_size in filter_sizes:
            l_conv = Conv1D(filters=128,
kernel_size=filter_size, activation='relu')(embedded_sequences)
            l_pool = MaxPooling1D(pool_size=3)(l_conv)
            convs.append(l_pool)

        l_merge = Concatenate(axis=1)(convs)

        # add a 1D convnet with global maxpooling, instead of
Yoon Kim model
        conv = Conv1D(filters=128, kernel_size=3,
activation='relu')(embedded_sequences)
        pool = MaxPooling1D(pool_size=3)(conv)

        if extra_conv==True:
            x = Dropout(0.5)(l_merge)
        else:
            # Original Yoon Kim model
            x = Dropout(0.5)(pool)
        x = Flatten()(x)
        x = Dense(128, activation='relu')(x)
        #x = Dropout(0.5)(x)

        preds = Dense(labels_index, activation='softmax')(x)

        model = Model(sequence_input, preds)
        model.compile(loss='categorical_crossentropy',
                      optimizer='adam',
```

```
                    metrics=['acc'])

        return model
    x_train = cnn_data[:-num_validation_samples]
    y_train = labels[:-num_validation_samples]
    x_val = cnn_data[-num_validation_samples:]
    y_val = labels[-num_validation_samples:]

    model = ConvNet(embedding_weights, MAX_SEQUENCE_LENGTH,
len(word_index)+1, EMBEDDING_DIM,

len(list(clean_questions["target"].unique())), False)
    model.fit(x_train, y_train, validation_data=(x_val, y_val),
epochs=3, batch_size=128)
    y_pred = model.predict(x_val)
    y_pred_cnn = np.argmax(y_pred, axis=1)
    y_val_cnn = np.argmax(y_val, axis=1)
    accuracy_cnn, precision_cnn, recall_cnn, f1_cnn, r2_cnn =
get_metrics(y_val_cnn, y_pred_cnn)
    print("accuracy = %.3f, precision = %.3f, recall = %.3f, f1
= %.3f, r2 = %.3f" % (accuracy_cnn, precision_cnn, recall_cnn,
f1_cnn, r2_cnn))
    cm_cnn = confusion_matrix(y_val_cnn, y_pred_cnn)
    fig = plt.figure(figsize=(10, 10))
    plot = plot_confusion_matrix(cm, classes=['ham','spam'],
normalize=False, title='Confusion matrix')
    plt.show()

    print("CNN confusion matrix")
    print(cm_cnn)
    print("Word2Vec confusion matrix")
    print(cm_w2v)
    print("TFIDF confusion matrix")
    print(cm2)
    print("BoW confusion matrix")
    print(cm)
```