Лабораторная работа №6.   
Обработка текста на естественном языке

*Цель работы*: познакомиться с обработкой естественного языка, изучить понятия "токенизация", "стемминг", "лемматизация", модель *Bag* *of* *Words*. Научиться применять полученные знания для классификации текста, сентимент-анализа предложений и тематическое моделирование и реализуем систему идентификации тем в документах.

## Основные понятия. установка пакетов

Обработка естественного языка (Natural Language Processing — NLP) становится важной частью современных систем. Она интенсивно применяется в поисковых системах, речевых интерфейсах, процессорах документов.

Для построения приложений будет использоваться пакет *Python Natural Language Toolkit* (NLTK).

$ pip3 install nltk

Чтобы получить доступ ко всем наборам данных, предоставляемых в пакете *NLTK*, мы должны загрузить их. Откройте оболочку *Python*, введя в окне терминала следующую команду:

$ python3

Введите следующие команды для загрузки данных.

>>> import nltk

>>> nltk.download()

В работе также будет использоваться пакет gensim – библиотека средств семантического моделирования. Установите этот пакет, выполнив в окне терминала следующую команду:

$ pip3 install gensim

Для правильной работы пакета *gensim* вам может понадобиться еще один пакет: *pattern*. Его можно установить с помощью следующей команды:

$ pip3 install pattern

## Токенизация

**Токенизация** — это процесс разбиения входного текста на меньшие элементы, такие как слова или предложения. Эти элементы называются токенами (лексемами). В зависимости от того, что мы хотим сделать, мы можем определить собственные методы для разбиения текста на множество токенов.

Выполним токенизацию текста с помощью пакета NLTK.

1. Создайте новый файл Python и импортируйте следующие пакеты.

from nltk.tokenize import sent\_tokenize, word\_tokenize, WordPunctTokenizer

1. Определим входной текст, который будем использовать для токенизации.

# Определение входного текста

input\_text = "Do you know how tokenization works? Itfs actually

quite interesting! Let ’s analyze a couple of sentences and

figure it out."

1. Выполним разбивку входного текста с помощью токенизатора предложений.

# Токенизация предложений

print("\nSentence tokenizer:")

print(sent\_tokenize(input text))

1. Выполним разбивку входного текста с помощью токенизатора слов.

# Токенизатор слов

print("\nWord tokenizer:")

print(word\_tokenize(input\_text))

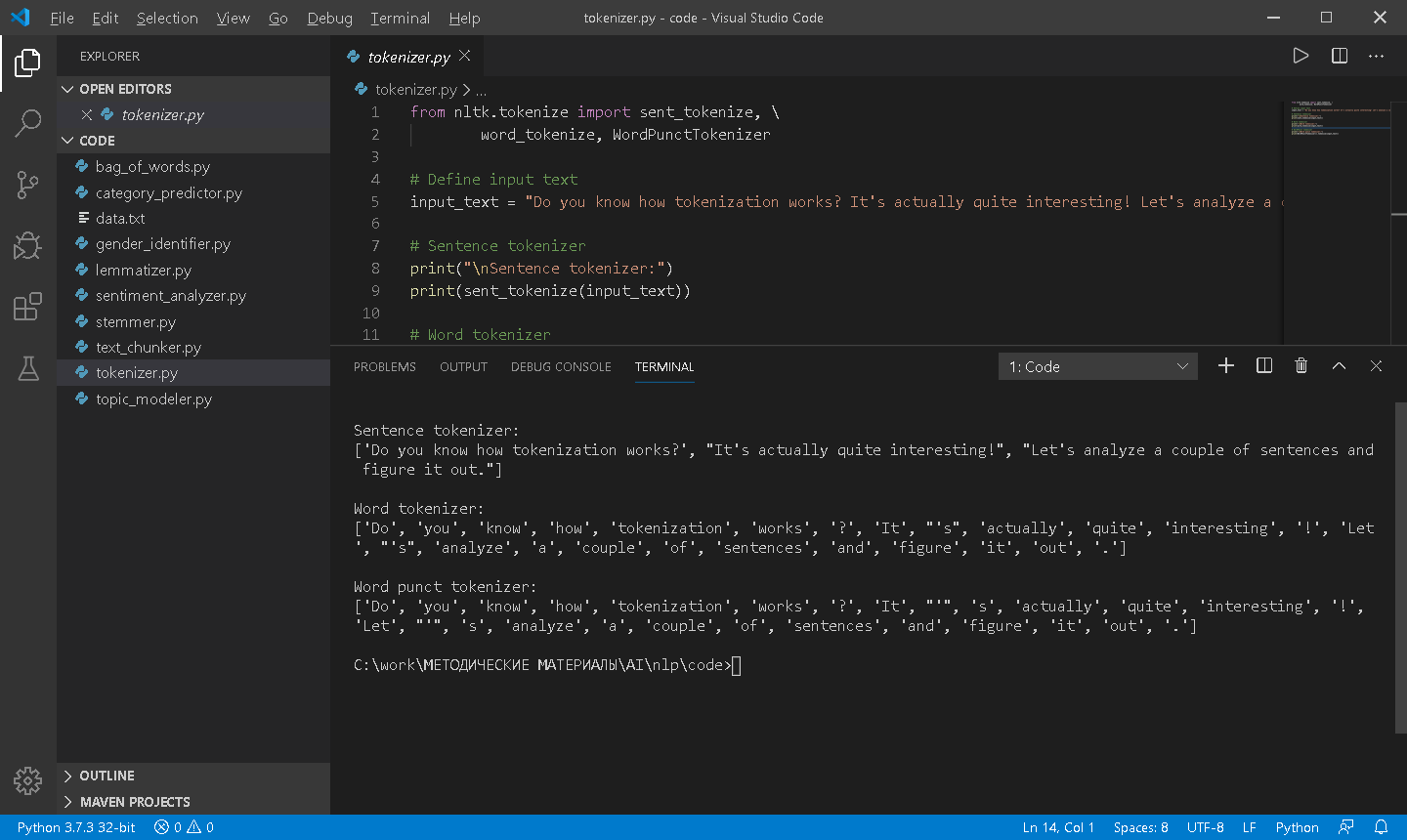
1. Выполним разбивку входного текста с помощью токенизатора слов и токенизатора пунктуации.

# Токенизатор пунктуации

print("\nWord punct tokenizer:")

print(WordPunctTokenizer().tokenize(input\_text))

После выполнения этого кода в окне терминала отобразится следующий вывод:



Токенизатор предложений разбивает входной текст на предложения. Два токенизатора слов ведут себя по-разному в отношении знаков пунктуации. Например, слово "It's" разбивается токенизатором пунктуации иначе, нежели обычным токенизатором.

## Преобразование слов в их базовые формы с помощью стемминга

Мы должны работать с различными формами одного и того же слова и хотим добиться того, чтобы компьютер понимал, что эти различные слова имеют одну и ту же корневую форму. Например, слово sing может встречаться во многих формах, таких как sang, singer, singing, singer и др. Все только что перечисленные слова имеют общий корень.

В процессе анализа текста полезно извлекать такие корневые формы. Это позволяет получать полезные статистики, помогающие анализировать входной текст. Одним из способов обеспечения этого является **стемминг**. Целью стемминга является приведение слов в их различных формах к общему корню. В основном это эвристический процесс, заключающийся в отсечении окончаний слов для выделения их корневых форм. Покажем, как это можно сделать с помощью NLTK.

Создайте новый файл Python и импортируйте следующие пакеты.

from nltk.stem.porter import PorterStemmer

from nltk.stem.lancaster import LancasterStemmer

from nltk.stem.snowball import SnowballStemmer

Определим входные слова.

input\_words = [’writing 1, 'calves', 'be', 'branded', 'horse',

'randomize', 'possibly', 'provision', 'hospital',

'kept', 'scratchy', 'code']

Создадим объекты для стеммеров Портера (Porter), Ланкастера (Lancaster)и Сноуболла(Snowball).

# Создание объектов различных стеммеров

porter = PorterStemmer()

lancaster = LancasterStemmer()

snowball = SnowballStemmer('english')

Создадим список имен стеммеров для их отображения в виде таблицы и соответствующим образом отформатируем выходной текст.

# Создание списка имен стеммеров для отображения

['PORTER', 'LANCASTER', 'SNOWBALL']

formatted\_text = '{:>16}' \* (len(stemmer\_names) + 1)

print('\n', formatted\_text.format('INPUT WORD', \*stemmer\_names),

'\n', '='\*68)

Выполним итеративный стемминг слов, используя три различных стеммера.

# Стемминг слов и отображение результатов

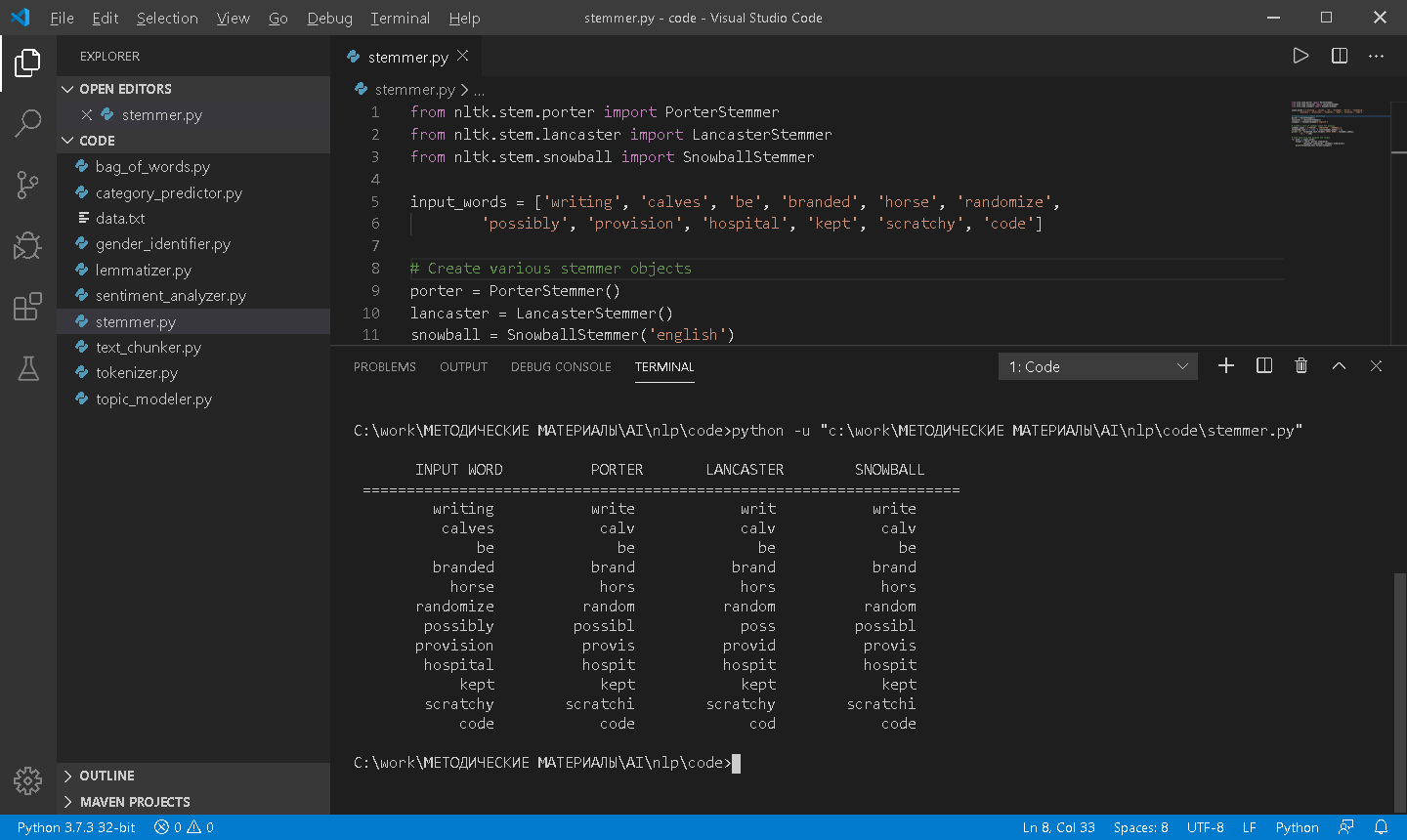
for word in input\_words:

output = [word, porter.stem(word),

lancaster.stem(word), snowball.stem(word)]

print(formatted\_text.format(\*output))

Выполнив этот код, вы увидите в окне терминала следующий вывод:



Стеммеры ведут себя по-разному в отношении слов наподобие possibly или provision. Наименее строгим из них является стеммер Портера, а наиболее строгим — стеммер Ланскастера. Результаты, полученные с помощью стеммера Ланкастера, немного сбивают с толку, поскольку он слишком урезает слова. В то же время этот алгоритм демонстрирует высокую скорость.

## Преобразование слов в их корневые формы с помощью лемматизации

Лемматизация — еще один способ редукции слов к их корневым формам.

Как вы могли заметить в предыдущем разделе, корневые формы, полученные с помощью стиммеров, не имели смысла. Например, в качестве корня слова *calves* все три стеммера указали форму *calv*, в действительности не являющуюся словом. Лемматизация предпринимает более структурированный подход для разрешения этой проблемы.

Процеес лемматизации использует словарь и морфологический анализ слов. В нем корневые формы получают путем удаления окончаний, таких как *ing* или *ed*, из форм с окончаниями. Такую базовую форму любого слова называют леммой. Если вы лемматизируете слово *calves*,-то получите *calf* в качестве результата. Следует отметить, что конечный результат зависит от того, является слово глаголом или существительным. Рассмотрим пример лемматизации с помощью пакета *NLTK*.

Создайте новый файл *Python* и импортируйте следующие пакеты:

from nltk.stem import WordNetLemmatizer

Определим входные слова. Мы будем использовать тот же набор слов, что и в предыдущем разделе, чтобы можно было сравнить результаты.

input\_words = ['writing \*, 'calves', 'be', 'branded', 'horse',

'randomize', 'possibly', 'provision', 'hospital',

'kept', 'scratchy', 'code']

Создадим объект lemmatizer.

# Создание объекта лемматизатора

lemmatizer = WordNetLemmatizer()

Создадим список имен лемматизаторов для их отображения в виде таблицы и соответствующим образом отформатируем выходной текст.

# Создание списка имен лемматизаторов для их отображения

lemmatizer\_names = ['NOUN LEMMATIZER', 'VERB LEMMATIZER']

formatted\_text = '{:>24}' \* (len(lemmatizer\_names) + 1)

print('\n', formatted\_text.format('INPUT WORD',

\*lemmatizer\_names), '\n',

Выполним итеративную лемматизацию слов, используя лемматизаторы Noun (существительное) и Verb (глагол).

# Лемматизация слов и отображение результатов

for word in input\_words:

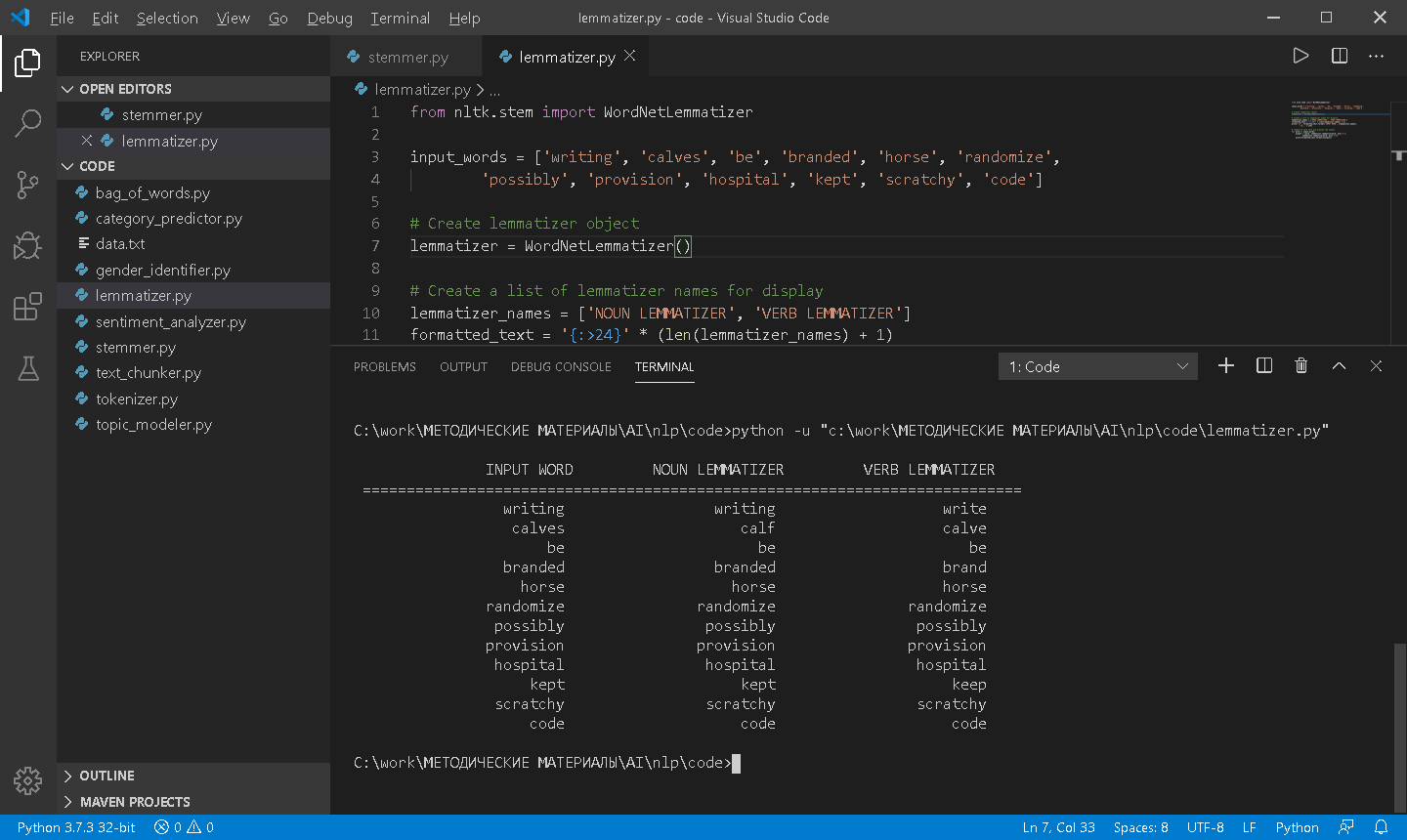
output = [word, lemmatizer.lemmatize(word, pos='n'),

lemmatizer.lemmatize(word, pos='v')]

print(formatted\_text.format(\*output))

Выполнив этот код, вы увидите в окне своего терминала следующий вывод.

В отношении таких слов, как *writing* или *calves*, лемматизатор существительных работает иначе, чем лемматизатор глаголов. Если вы сравните эти результаты с теми, которые были получены с помощью стеммеров, то увидите, что здесь также имеются различия. Все результаты, полученные с помощью лемматизаторов, имеют смысл, тогда как результаты, полученные с помощью стеммеров, могут как иметь смысл, так и не иметь.



## Разбиение текстовых данных на информационные блоки

Обычно текстовые данные приходится разбивать на отдельные информа ционные блоки (чанки) для последующего анализа. Этот процесс называется чанкингом, и он часто используется при анализе текста. Условия разбиения текста на блоки могут меняться в зависимости от конкретной задачи. Чанкинг не следует смешивать с токенизацией, которая также связана с разбие нием текста на отдельные части. В процессе чанкинга мы не придерживаемся никаких ограничений, а результирующие блоки обязаны иметь смысл.

При работе с большими документами важно разбивать текст на блоки для извлечения смысловой информации. В этом разделе будет показано, как раз бить входной текст на ряд блоков.

Создайте новый файл *Python* и импортируйте следующие пакеты.

import numpy as np

from nltk. Corpus import brown

Определим функцию, разбивающую входной текст на блоки. Первым параметром является текст, вторым — количество слов в каждом блоке.

# Разбиение входного текста на блоки,

# причем каждый блок содержит N слов

def chunkier(input\_data, N):

input\_words = input\_data.split(\*output = [])

#Выполним цикл по словам и разобьем их на блоки,

#используя входной параметр

cur chunk = []

count = О

for word in input\_words:

cur\_chunk.append(word)

count += 1

if count == N:

output.append(' '.join(cur\_chunk))

count, cur chunk = 0, []

output.append(' '.join(cur\_chunk))

return output

Определим основную функцию и прочитаем входные данные, используя коллекцию текстов Brown. В данном случае мы читаем 12 тысяч слов, но вы можете задать для этого параметра любое другое желаемое значение.

if \_\_name\_\_ == '\_\_main\_\_':

# Чтение первых 12000 слов из коллекции Brown

input\_data = ' '.join(brown.words()[:12000])

Определим количество слов в каждом блоке.

# Определение количества слов в каждом блоке

chunk size = 700

Разобьем текст на блоки и отобразим результат.

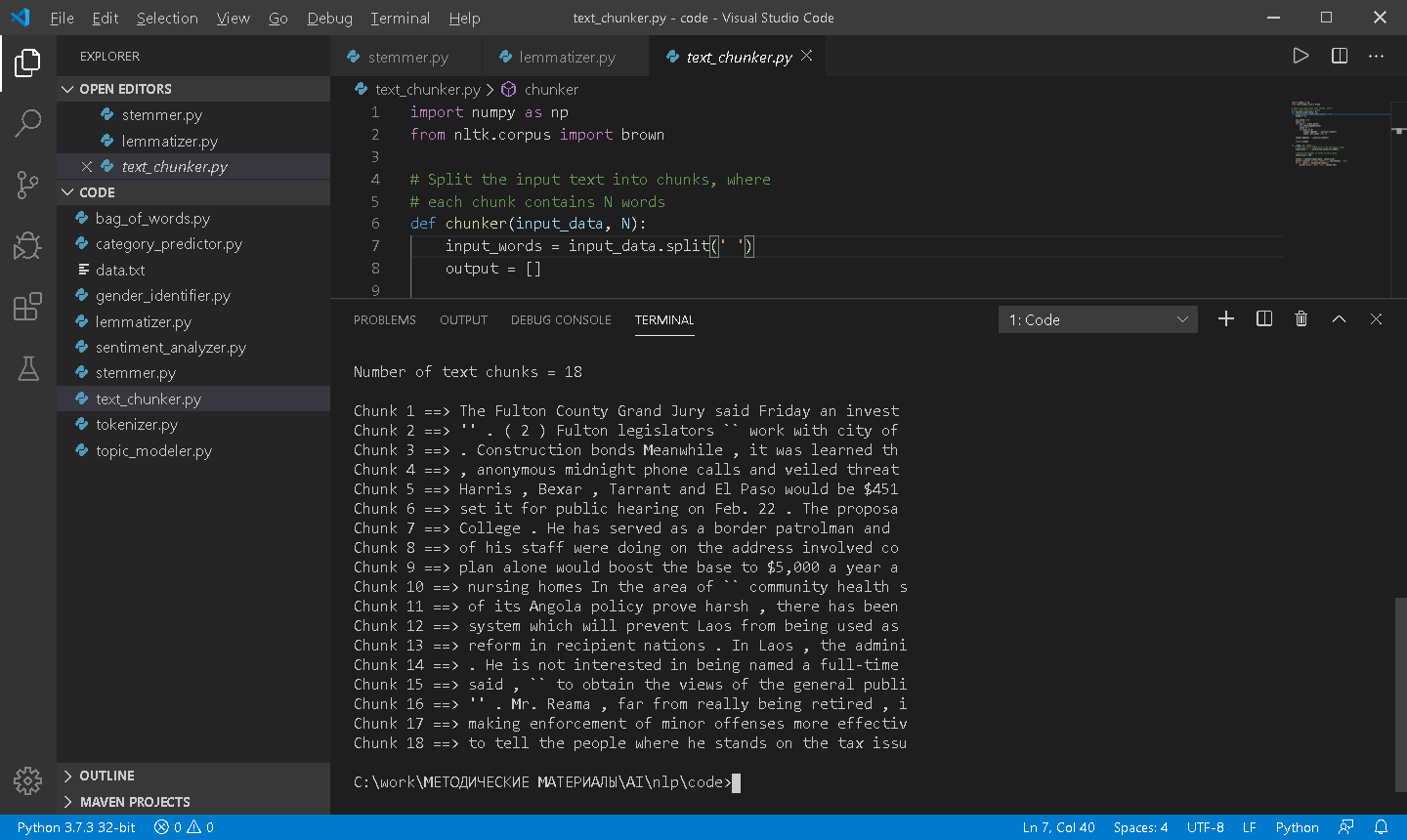
chunks = chunker(input\_data, chunk\_size)

print('\nNumber of text chunks =', len(chunks)')

for i, chunk in enumerate(chunks):

print('Chunk \*, i+1, '==>', chunk[:50])

Выполнив этот код, вы увидите в окне терминала следующий вывод:



## Извлечение частотности слов с помощью модели Bag of Words

Одной из основных задач текстового анализа является преобразование текста в числовую форму, к которой можно было бы применить машинное обучение. Рассмотрим случай текстовых документов, содержащих миллионы слов. Анализ таких документов предполагает извлечение текста и преобразование его в числовое представление.

Чтобы алгоритмы машинного обучения могли анализировать данные и извлекать содержательную информацию, они должны получать эти данные в числовом виде. И здесь на помощь приходит модель *Bag of Words* (мешок слов). Эта модель извлекает словарь, состоящий из всех слов, которые встречаются в документе, и строит модель с помощью терм-документной матрицы. Это позволяет нам представить любой документ в виде ''мешка слов".

При этом мы поддерживаем лишь счетчик слов, игнорируя их порядок следования и грамматические детали.

**Терм-документная матрица** (document term matrix) – это таблица, которая содержит значения счетчиков различных слов, встречающихся в документе.

Поэтому текстовый документ можно представить в виде взвешенной комбинации различных слов. Мы можем устанавливать пороговые значения и выбирать наиболее значимые слова. В некотором смысле мы строим гистограмму всех слов в документе, совокупность которых будет служить вектором признаков. Этот вектор признаков используется для классификации текста.

Рассмотрим следующие предложения.

* Предложение 1: The children are playing in the hall.
* Предложение 2: The hall has a lot of space.
* Предложение 3: Lots of children like playing in an open space.

Среди всех слов, входящих в состав всех трех предложений, часть являются уникальными: the, children, are, playing, in, hall, has, a, lot, of, space, like, an, open

Всего здесь имеется 14 различных слов. Построим гистограмму для каждого предложения, используя их счетчики слов. Каждый вектор признаков будет 14-мерным, поскольку общее количество слов составляет 14.

* Предложение 1: [2,1,1,1,1,1, 0, 0, 0, 0, 0, 0, 0, 0]
* Предложение 2: [1, 0, 0, 0, 0, 1,1,1,1,1,1, 0, 0, 0]
* Предложение 3: [0,1, 0,1,1, 0, 0, 0, 1,1,1,1,1,1]

Теперь, когда векторы признаков извлечены, мы можем использовать алгоритмы машинного обучения для анализа данных.

Покажем, как создать модель Bag of Words в NLTK. Создайте новый файл Python и импортируйте следующие пакеты.

import numpy as np

from sklearn.feature\_extraction.text import CountVectorizer

from nltk.corpus import brown

from text\_chunker import chunker

Прочитаем входные данные из коллекции Brown. Мы прочитаем 5400 слов, однако вы можете задать любое другое их количество.

# Чтение данных из коллекции Brown

input\_data = .join(brown.words()[:5400])

Определим количество слов в каждом блоке.

# Количество слов в каждом блоке

chunk size = 800

Разобьем входной текст на блоки.

text chunks = chunker(input\_data, chunk\_size)

Преобразуем блоки в элементы словаря.

# Преобразование в элементы словаря

chunks = []

for count, chunk in enumerate(text\_chunks):

d = {'index': count, 'text': chunk}

chunks.append(d)

Извлечем терм-документную матрицу, из которой мы получаем значения счетчиков каждого слова. Для этого используем метод *CountVectorizer*, который имеет два входных параметра. Первый параметр — минимальная, а второй — максимальная частота документа. Здесь термин "частота" относится к количеству вхождений слова в тексте.

# Извлечение терм-документной матрицы

count\_vectorizer - = CountVectorizer(min\_df=7, max\_df=20)

document\_term\_matrix = count\_vectorizer.fit\_transform([chunk['text \*]

for chunk in chunks])

Извлечем словарь и отобразим его. Здесь термин "словарь" относится к списку отдельных слов, извлеченных на предыдущем шаге.

# Извлечение и отображение словаря

vocabulary = np.array(count\_vectorizer.get\_feature\_names())

print("\nVocabulary:\n", vocabulary)

Сгенерируем отображаемые имена.

# Генерация имен блоков

chunk\_names = []

for i in range(len(text\_chunks)):

chunk names.append('Chunk-' + str(i+l))

Выведем терм-документную матрицу.

# Вывод терм-документной матрицы

print("\nDocument term m a t r i x )

formatted\_text = '{:>12}' \*(len(chunk\_names) + 1)

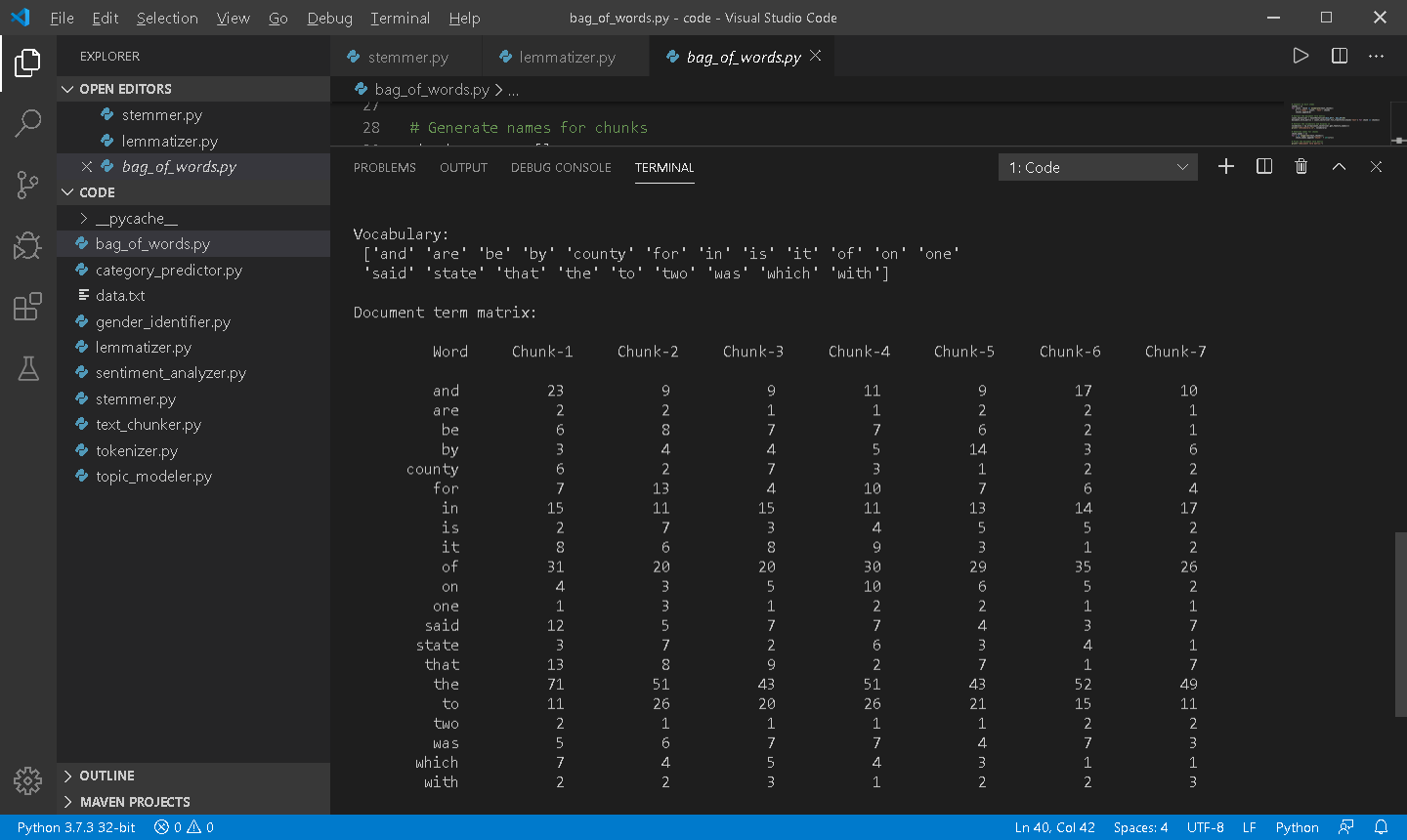
print('\n', formatted\_text.format(’Word', \*chunk\_names), '\n')

for word, item in zip(vocabulary, document\_term\_matrix.T):

output = [word] + [str(freq) for freq in item.data]

print(formatted\_text.format(\*output))

Выполнив этот код, вы увидите в окне терминала следующий вывод.



Этот вывод представляет все слова, содержащиеся в терм-документной матрице, и значения соответствующих счетчиков для каждого блока.

## Создание прогнозатора категорий

Прогнозаторы категорий используются для предсказания категории, к которой принадлежит заданный элемент текста. Их часто используют при классификации текста для категоризации текстовых документов. Поисковые механизмы часто используют этот инструмент для поиска результатов по их релевантности.

Предположим, мы хотим спрогнозировать, относится ли данное предложение к спорту, политике или науке. Для этого мы создаем массив данных и тренируем алгоритм. Далее этот алгоритм может быть использован для вынесения суждений о неизвестных данных.

Для построения такого прогнозатора мы будем использовать статистику tf-idf (Term Frequency — InverseDocument Frequency — "частота слова — обратная частота документа"). Мы должны понимать важность каждого слова в наборе документов. Статистика tf-idf помогает оценить, насколько важным является данное слово в документе из набора.

Рассмотрим первую часть этой статистики, tf (term frequency— частота слова). Это мера частотности каждого слова, встречающегося в документе.

Поскольку общее количество слов в разных документах разное, абсолютные числа в гистограмме будут меняться. Чтобы уравнять условия, гистограмму необходимо нормализовать. Поэтому мы делим значение счетчика каждого слова на общее количество слов в данном документе для получения значения частотности.

Второй частью данной статистики является idf (inverse document frequency — обратная частота документа) — мера уникальности слова в данном документе коллекции. Вычисляя частотность, мы полагаем, что все слова одинаково важны. Но сама по себе частотность слов не может служить надежной характеристикой, поскольку, например, такие общеупотребительные слова, как like и the, встречаются намного чаще других. Чтобы сбалансировать частотность таких общеупотребительных слов, мы должны уменьшить их вес и одновременно увеличить вес редко используемых слов.

Благодаря этому мы получаем возможность идентифицировать слова, являющиеся уникальными для всех документов коллекции, что, в свою очередь, позволяет формулировать более точные векторы признаков, учитывающие характеристичность слов.

Чтобы вычислить эту статистику, нужно вычислить отношение количества документов, содержащих данное слово, к общему количеству документов в коллекции. По сути, это отношение характеризует долю документов, содержащих данное слово. Обратная частота документа рассчитывается далее как логарифм этого отношения, взятый со знаком "минус".

После этого мы объединяем частоту слов и обратную частоту документа, чтобы сформировать вектор признаков для категоризации документов. Рассмотрим, как создать прогнозатор категорий.

Создайте новый файл Python и импортируйте следующие пакеты.

from sklearn.datasets import fetch\_20newsgroups

from sklearn.naive\_bayes import MultinomialNB

from sklearn.feature\_extraction.text import TfidfTransformer

from sklearn.feature\_extraction.text import CountVectorizer

Определим карту категорий, которые будут использоваться для тренировки. В данном случае у нас будет пять категорий. В этом словаре ключи ссылаются на имена в пакете scikit-learn.

# Определение карты категорий

category\_map = {'talk.politics.misc': ’Politics',

'rec.autos': 'Autos', 'rec.sport.hockey': 'Hockey',

'sci.electronics': 'Electronics', 'sci.med': 'Medicine'}

Получим тренировочный набор, используя вызов fetch\_20newsgroups.

# Получение тренировочного набора данных

training\_data = fetch\_20newsgroups(subset='train',

categories=category\_map.keys(), shuffle=True,

random state=5)

Извлечем значения счетчиков слов с помощью объекта CountVectorizer.

# Создание векторизатора и извлечение счетчиков слов

count\_vectorizer = CountVectorizer()

train\_tc = count\_vectorizer.fit\_transform(training\_data.data)

print("XnDimensions of training data:", train\_tc.shape)

Создадим преобразователь "Term Frequency Inverse Document Frequency" (tf-idf ) и обучим его, используя имеющиеся данные.

# Создание преобразователя tf-idf

tfidf = TfidfTransformer()

train tfidf = tfidf.fit\_transform(train\_tc)

Определим выборку входных предложений, которые будут использованы для тестирования.

# Определение тестовых данных

input\_data = ['You need to be careful with cars when you are driving on slippery roads', 'A lot of devices can be operated wirelessly', 'Players need to be careful when they are close to goal posts’, 'Political debates help us understand the perspectives of both sides']

Обучим мультиномиальный байесовский классификатор, используя тренировочные данные.

# Обучение байесовского классификатора

classifier = MultinomialNB().fit(train\_tfidf, training\_data.target)

Преобразуем входные данные, используя векторизатор счетчиков.

# Преобразование входных данных с помощью

# векторизатора счетчиков

input\_tc = count\_vectorizer.transform(input\_data)

Преобразуем векторизованные данные с помощью преобразователя tf-idf, чтобы их можно было пропустить через модель выводов.

# Преобразование векторизованных данных с помощью

# преобразователя tf-idf

input tfidf = tfidf.transform(input\_tc)

Предскажем результат, используя вектор, полученный с помощью преобразователя tf-idf

# Прогнозирование результирующих категорий

predictions = classifier.predict(input\_tfidf)

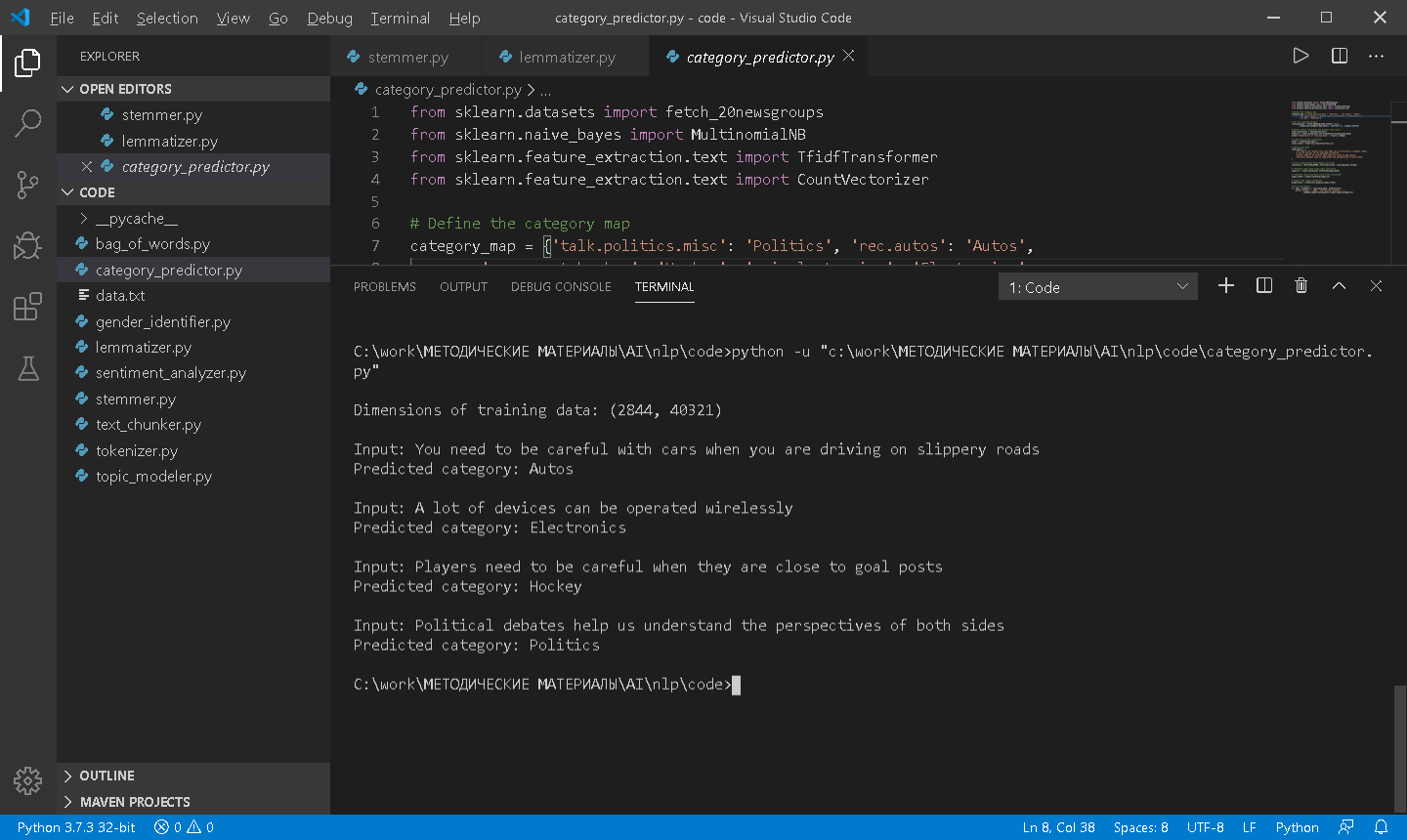
Выведем результирующие категории для каждого из образцов во входных тестовых данных.

# Вывод результатов

for sent, category in zip(input\_\_data, predictions):

print('\nlnput:', sent, '\nPredicted category:',

category\_\_map[training\_data.target\_names[category]])



## Создание анализатора грамматических родов

Одной из задач, представляющих интерес в связи с обработкой текста, является идентификация грамматического рода. В данном случае мы создадим вектор признаков с помощью эвристики и используем его для обучения классификатора. В качестве эвристики выберем последние N букв заданного имени. Например, если имя заканчивается буквами ш, то, вероятнее всего, это женское имя, такое как Amelia или Genelia. С другой стороны, если имя заканчивается буквами r/с, то оно, вероятно, является мужским именем, таким как Mark или Clark.Поскольку точное количество букв, которые при этом следует учитывать, нам неизвестно, мы проведем эксперименты, в ходе которых выясним, какое количество букв обеспечивает наилучшие ответы. Рассмотрим, как это делается.

Создайте новый файл *Python* и импортируйте следующие пакеты.

import random

from nltk import NaiveBayesClassifier

from nltk.classify import accuracy as nltk\_accuracy

from nltk.corpus import names

Определим функцию, извлекающую последние N букв из входного слова.

# Извлечение последних N букв из входного слова

# и возврат значения, выступающего в качестве "признака"

def extract\_features(word, N=2):

last\_n\_letters = word[-N:]

return {'feature': last\_n\_letters.lower()}

Определим основную функцию и извлечем обучающие данные из пакета scikit-learn. Эти данные содержат маркированные мужские и женские имена

if \_\_name\_\_ == '\_\_main\_\_':

# Создание обучающих данных с использованием

# помеченных имен, доступных в NLTK

male\_list = [(name, 'male') for name in

names.words('male.txt')]

female\_list = [(паше, 'female') for name in

names.words('female.txt')]

data = (male list + female list)

Зададим затравочное значение для генератора случайных чисел и перемешаем данные.

# Затравочное значение для генератора случайных чисел

random.seed(5)

# Перемешивание данных

random.shuffle(data)

Создадим выборку имен, которые будут использоваться для тестирования.

# Создание тестовых данных

input\_names = ['Alexander', 'Danielle', 'David', 'Cheryl']

Определим процентные доли данных, которые будут использоваться для тренировки и тестирования.

# Определение количеств образцов, используемых

# для тренировки и тестирования

num\_train = int(0.8 \* len(data))

Для прогнозирования рода мы будем использовать в качестве вектора признаков последние N символов имени. Мы будем варьировать этот параметр, чтобы выяснить, как при этом меняются результаты. В данном случае мы исследуем интервал значений от 1 до 5.

# Итерирование по различным длинам конечного

# фрагмента для сравнения точности

for i in range(1, 6):

print('\nNumber of end letters:', i)

features = [(extract\_features(n, i), gender) for (n, gender) in data]

Разделим данные на тренировочный и тестовый наборы.

train\_data, test\_data = features[:num\_train], features[num\_train:]

Создадим наивный байесовский классификатор, используя тренировочные данные.

classifier = NaiveBayesClassifier.train(train\_data)

Вычислим точность классификатора, используя встроенный метод, оступный в библиотекеNLTK.

# Вычисление точности классификатора

accuracy = round(100 \* nltk\_accuracy(classifier, test\_data), 2)

print('Accuracy = ' + str(accuracy) + '%')

Предскажем результатдля каждого имени из входного тестового списка.

# Предсказание результатов для входных имен

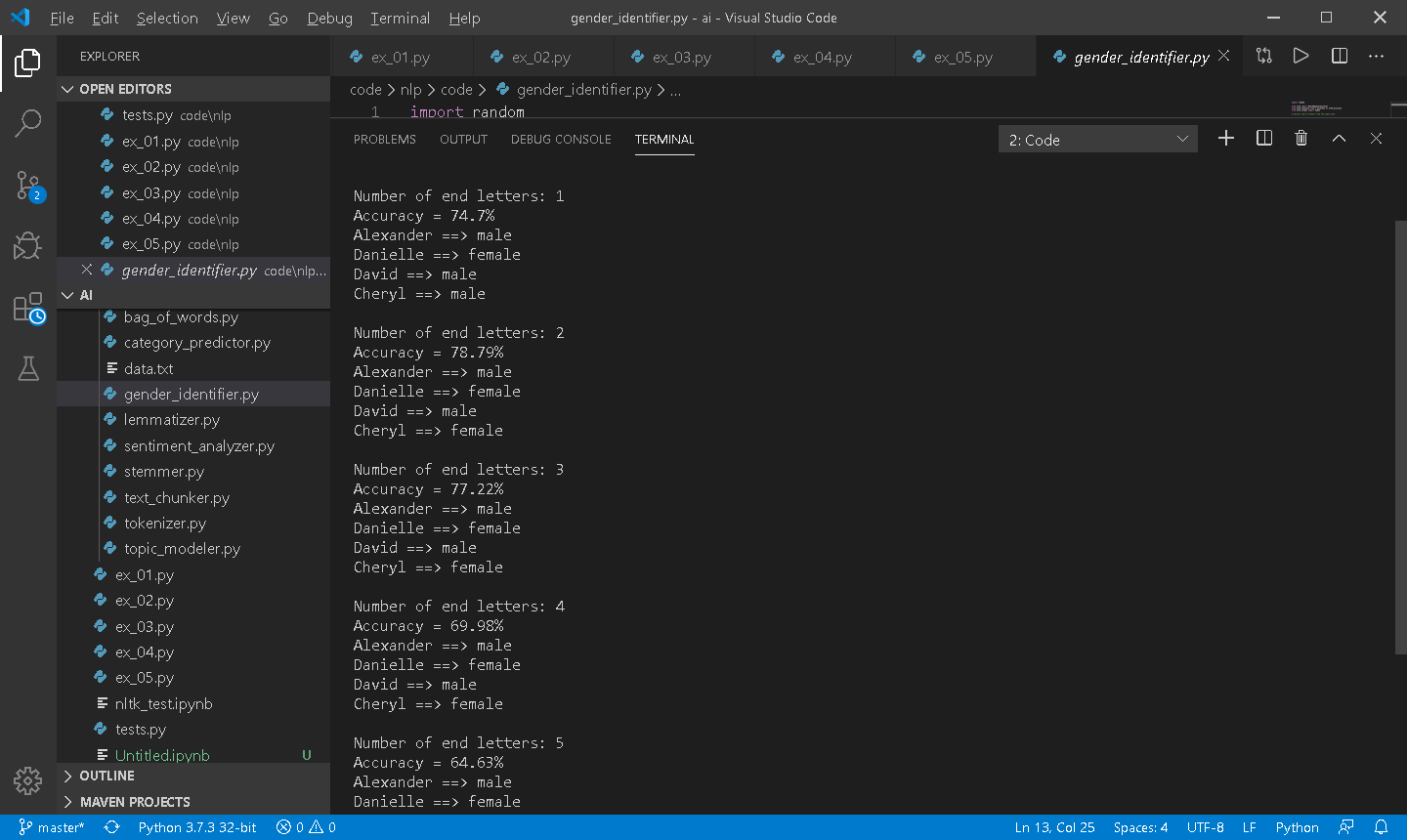
# с использованием обученной модели классификатора

for name in input\_names:

print(name, '==>',

classifier.classify(extract\_features(name, i)))

Выполнив этот код, вы получите следующий вывод в окне своего терминала.



На этом снимке экрана представлены точность и предсказанные результаты для тестовых данных. Посмотрим, как обстоят дела с большими значениями N.

Нетрудно заметить, что точность достигает своего максимума при N = 2, а затем начинает снижаться.

## Создание сентимент-анализатора

Сентимент-анализ — это процесс определения тональности заданного фрагмента текста. Нго, например, можно использовать для того, чтобы определить, является ли отзыв о кинофильме положительным или отрицательным. Это одно из наиболее популярных применений обработки естественного языка. В зависимости от конкретной задачи мы можем расширять количество категорий. Обычно эту методику используют для того, чтобы определить отношение людей к конкретному продукту, торговой марке или теме. Ее часто применяют для анализа результатов маркетинговых компаний, опросов общественного мнения, присутствия в социальных сетях, составления обзоров продуктов на торговых сайтах т.п.

Рассмотрим, как определить тональность рецензии на фильм. Для создания данного анализатора мы используем наивный байесовский классификатор. Сначала нам предстоит извлечь все уникальные слова из текста. Для работы NLTK-классификатора требуется, чтобы данные были подготовлены в виде словаря. Как только текстовые данные будут разделены на тренировочный и тестовый наборы, мы обучим наивный байесовский классификатор распределять обзоры на основании того, являются ли они положительными и отрицательными. Мы также выведем наиболее информативные слова, указывающие на положительные или отрицательные отзывы.

Эта информация представляет для нас интерес, поскольку позволяет понять, какие именно слова используются для выражения той или иной реакции.

Создайте новый файл Python и импортируйте следующие пакеты.

from nitk.corpus import movie\_reviews

from nltk.classify import NaiveBayesClassifier

from nltk.classify.util import accuracy as nltk\_accuracy

Определим функцию, которая создает объект словаря на основании входных слов и возвращает его.

# Извлечение признаков из входного списка слов

def extract\_features(words):

return diet([(word, True) for word in words])

Определим основную функцию и загрузим помеченные отзывы о фильмах.

if \_\_name\_\_ == '\_\_main\_\_' :

# Загрузка отзывов из коллекции

fileids\_pos = movie\_reviews.fileids('pos')

fileids\_neg = movie\_reviews.fileids('neg')

Извлечем признаки из отзывов о фильмах и снабдим их соответствующими метками.

# Извлечение признаков из отзывов

features\_pos = [(extract\_features(movie\_reviews.words(

fileids=[f])), 'Positive') for f in fileids\_pos]

features\_neg = [(extract\_features(movie\_reviews.words(

fileids=[f])), 'Negative') for f in fileids\_neg]

Определим разбиение данных на тренировочный и тестовый наборы. В нашем случае 80% данных будут тренировочными, а 20% — тестовыми.

# Определение относительных долей тренировочного

# и тестового наборов (80% и 20%)

threshold = 0.8

num\_pos = int(threshold \* len(features\_pos))

num neg = int(threshold \* len(features\_neg))

Разделим векторы признаков для тренировочного и тестового наборов.

# Создание тренировочного и тестового наборов

features\_train = features\_pos[:num\_pos] + features\_neg[:num\_neg]

features\_test = features\_pos[num\_pos:] + features neg[num\_neg:]

Выведем количество точек данных, используемых для тренировки и тестирования.

# Вывод количества используемых точек данных

print('\nNumber of training datapoints:', len(features\_train))

print('Number of test datapoints:\*, len(features\_test))

Обучим наивный байесовский классификатор, используя тренировочные данные, и вычислим точность, используя встроенный метод, доступный в NLTK.

# Обучение наивного байесовского классификатора

classifier = NaiveBayesClassifier.train(features\_train)

print('\nAccuracy of the classifier: ', nltk\_accuracy(

classifier, features\_test))

Выведем первые N наиболее информативных слов.

N = 15

print('\nTop ' + str(N) + ' most informative words:')

for i, item in

enumerate(classifier.most\_informative\_features()):

print(str(i+1) + ' + item[0])

if i == N - 1:

break

Определим выборку предложений, используемых для тестирования.

# Тестирование входных отзывов о фильмах

input\_reviews = [

'The costumes in this movie were great',

'I think the story was terrible and the characters were very

weak',

'People say that the director of the movie is amazing',

'This is such an idiotic movie. I will not recommend it to

anyone.

]

Выполним итерации по выборочным данным и спрогнозируем результаты.

print("\nMovie review predictions:")

for review in input\_reviews:

print("\nReview:", review)

Вычислим вероятности для каждого класса.

# Вычисление вероятностей

probabilities =

classifier.prob\_classify(extract\_features(review.split()))

Выберем вероятность с наибольшим значением.

# Выбор максимального значения

predicted\_sentiment = probabilities.max()

Выберем предсказанный выходной класс (положительная или отрицательная тональность).

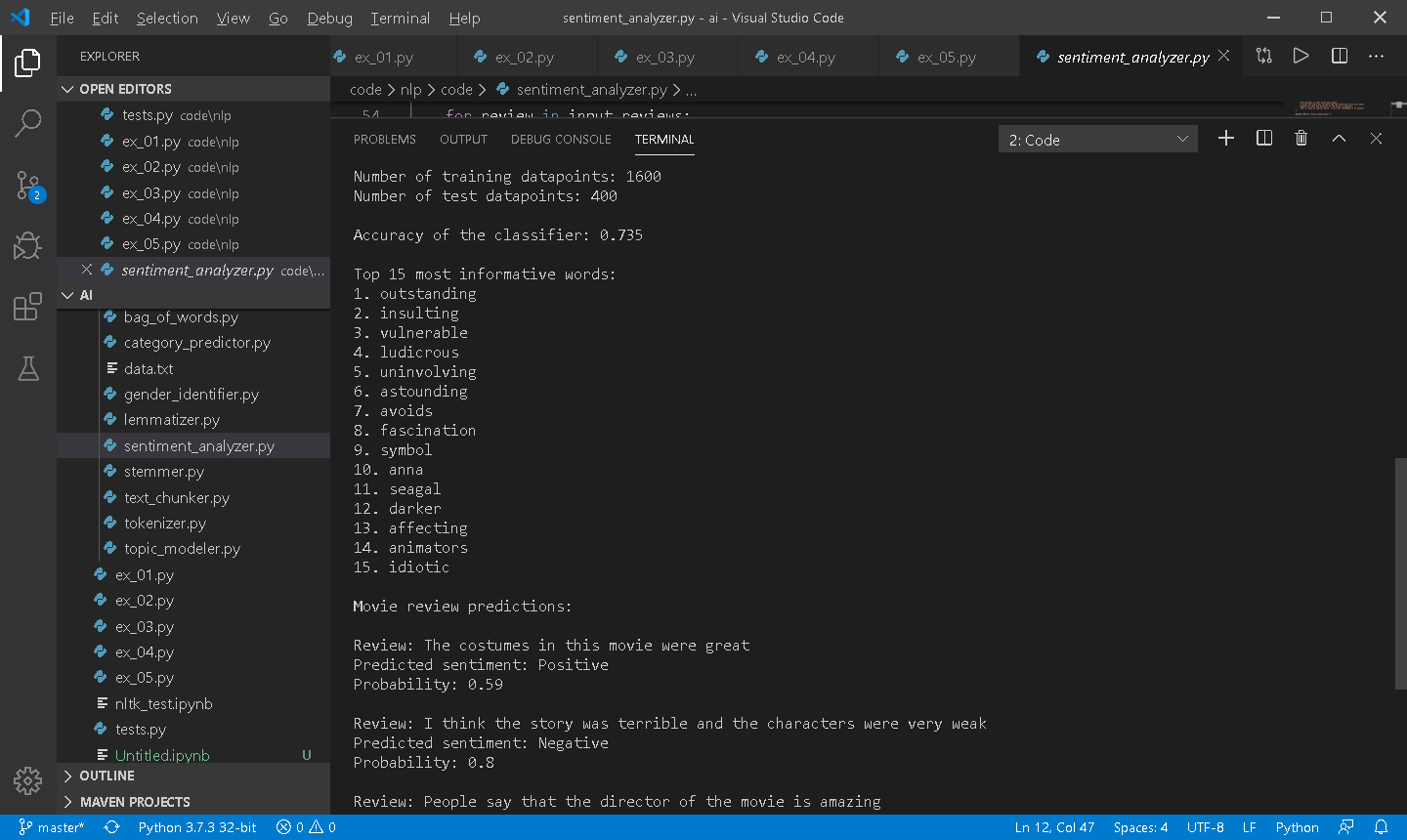
# Вывод результатов

print("Predicted sentiment:", predicted\_sentiment)

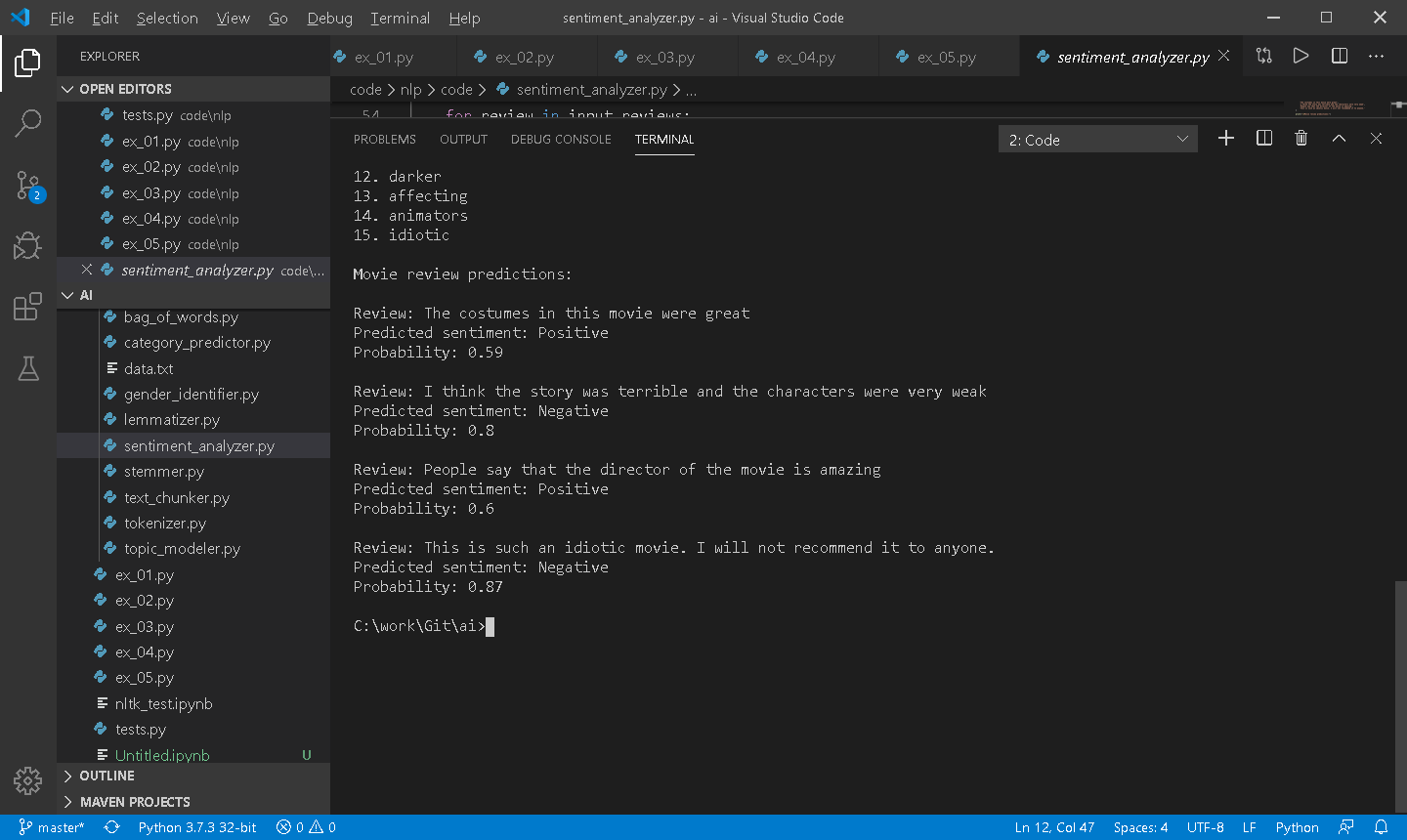
print("Probability:",

round(probabilities.prob(predicted\_sentiment), 2))

Выполнив этот код, вы увидите следующий вывод в окне терминала. На этом экранном снимке отображены первые 15 наиболее информативных слов.



Следующая информация убеждает нас в том, что все прогнозы оказались корректными.



## Тематическое моделирование с использованием латентного размещения Дирихле

Тематическое моделирование — это процесс идентификации шаблонов в текстовых данных, которые соответствуют некоторой теме. Если текст содержит несколько тем, то эта методика может быть использована для идентификации и разделения тем в пределах входного текста. Это делается для обнаружения скрытой тематической структуры в данном наборе документов.

Тематическое моделирование помогает оптимально организовать документы, которые впоследствии могут быть использованы для анализа. Следует отметить одну особенность алгоритмов тематического моделирования, которая заключается в том, что в этом случае мы не нуждаемся ни в каких помеченных данных. Это напоминает ситуацию с обучением без учителя, когда идентификация шаблонов осуществляется без использования маркеров.

С учетом огромных объемов текстовых данных, генерируемых в Интернете, тематическое моделирование приобретает особую важность, поскольку обеспечивает возможность суммаризации (summerization) всех этих данных, что иначе было бы невозможным.

Латентное размещение Дирихле (Latent Dirichlet Allocation) — это метод тематического моделирования, базовая предпосылка которого состоит в том, что каждый заданный фрагмент документа представляет собой смесь нескольких тем.

В качестве примера рассмотрим следующее предложение: "Data visualization is an important tool in financial analysis". В этом предложении затрагиваются несколько тем, таких как данные, визуализация, финансы и др. Такое конкретное сочетание помогает нам идентифицировать данный текст в большом документе.

В сущности, это статистическая модель, которая пытается воспользоваться идеей сочетания тем и построить модель на ее основе. В данной модели предполагается, что документы генерируются в ходе случайного процесса, основанного на этих темах. При таком подходе тема — это распределение слов из фиксированного словаря.

Рассмотрим, как выполняется тематическое моделирование в *Python*.

В этом разделе мы будем использовать библиотеку *gensim*. Создайте новый файл Python и импортируйте следующие файлы.

from nltk.tokenize import RegexpTokenizer

from nltk.corpus import stopwords

from nltk.stem.snowball import SnowballStemmer

from gensim import models, corpora

Определим функцию, загружающую входные данные. Входной файл содержит 10 предложений, разделенных символами новой строки.

# Загрузка входных данных

def load\_data(input\_file):

data = []

with open(input\_file, 'rf) as f:

for line in f.readlines():

data.append(line[:—1])

return data

Определим функцию, обрабатывающую входной текст. Первый шаг состоит в токенизации текста. Нам нужно выполнить стемминг токенизированного текста.

# Функция обработки, предназначенная для токенизации текста,

# удаления стоп-слов и выполнения стемминга

def process(input\_text):

# Создание регулярного выражения для токенизатора

tokenizer = RegexpTokenizer(г'\w+')

# Создание стеммера Сноуболла

stemmer = SnowballStemmer('english')

Кроме того, мы должны удалить стоп-слова из входного текста, поскольку они не добавляют никакой информации. Получим список стоп-слов.

# Получение списка стоп-слов

stop\_words = stopwords.words(’english \*)

Токенизируем входную строку.

# Токенизация входной строки

tokens = tokenizer.tokenize(input\_text.lower())

Удалим стоп-слова.

# Удаление стоп-слов

tokens = [х for х in tokens if not x in stop\_words]

Выполним стемминг токенизированных слов и вернем список.

# Выполнение стемминга токенизированных слов

tokens\_stemmed = [stemmer.stem(х) for х in tokens]

return tokens stemmed

Определим основную функцию и загрузим входные данные из файла data.txt.

if \_\_name\_\_ == '\_\_main\_\_':

# Загрузка входных данных

data = load data('data.txt')

# Создание списка токенов предложений

tokens = [process(х) for х in data]

# Создание словаря на основе токенизированных предложений

dict\_tokens = corpora.Dictionary(tokens)

# Создание терм-документной матрицы

doc\_term\_mat = [diet\_tokens.doc2bow(token) for token in tokens]

В качестве входного параметра мы должны предоставить количество тем. В данном случае нам известно, что входной текст содержит две различные темы. Укажем это.

# Определим количество тем для LDA-модели

num\_topics = 2

Сгенерируем латентную модель Дирихле.

# Генерирование LDA-модели

ldamodel = models.ldamodel.LdaModel(doc\_term\_mat,

num topics=num\_topics, id2word=dict\_tokens, passes=25)

Выведем для каждой темы первые 5 представительных слов.

num\_words = 5

print('\nTop ' + str(num words) + ' contributing words to each topic:')

for item in ldamodel.print\_topics(num\_topics=num\_topics,

num words=num words):

print('\nTopic', item[0])

# Вывод представительных слов вместе с их

# относительными вкладами

list\_of\_strings = item[l].split(' + ')

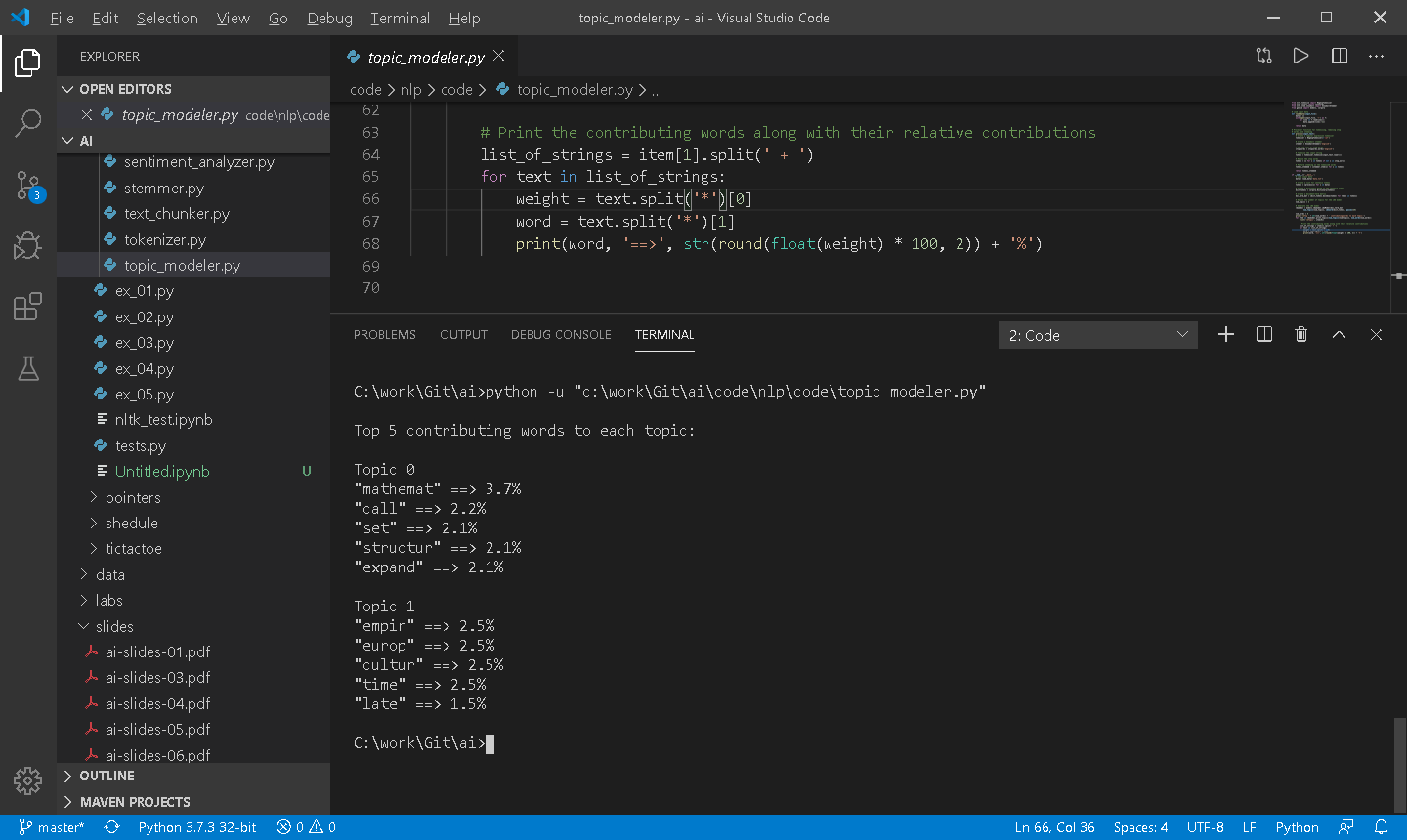
for text in list\_of\_\_strings:

weight = text.split('\*')[0]

word = text.split('\*')[1]

print(word, '==>’, str(round(float(weight)\*100, 2))+%')

Выполнив код, вы увидите в окне терминала следующий вывод.



Как видите, программа неплохо справилась с задачей разделения двух тем: математики и истории. Если вы обратитесь к тексту, то убедитесь в том, что речь в нем идет либо о математике, либо об истории.