МИНИСТЕРСТВО ОБРАЗОВАНИЯ РЕСПУБЛИКИ БЕЛАРУСЬ

БЕЛОРУССКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ

МЕХАНИКО-МАТЕМАТИЧЕСКИЙ ФАКУЛЬТЕТ

Кафедра дифференциальных уравнений и системного анализа

ЖДАНОВИЧ

Евгения Сергеевна

СТАТИСТИЧЕСКИЙ АНАЛИЗ ТЕКСТА И ЗАДАЧА ОПРЕДЕЛЕНИЯ АВТОРСТВА

Дипломная работа

Научный руководитель:
кандидат физ.-мат. наук,
доцент Е. М. Радыно

Допущена к защите

«___» _____ 2018 г.

Зав. кафедрой дифференциальных уравнений и системного анализа

доктор физ.-мат. наук, профессор В. И. Громак

Минск, 2018

ОГЛАВЛЕНИЕ

РЕФЕРАТ

В дипломной работе 45 страниц, 10 рисунков, 2 таблицы, 10 источников, одно приложение.

МАШИННОЕ ОБУЧЕНИЕ, СТАТИСТИЧЕСКИЙ АНАЛИЗ ТЕКСТА, ОПРЕДЕЛЕНИЕ АВТОРСТВА, АНАЛИЗ ДАННЫХ, ОБРАБОТКА ЕСТЕСТВЕННОГО ЯЗЫКА

В дипломной работе производится отбор статистических признаков текста, классификация текстов, принадлежащих различным авторам, и исследование динамики точности классификации в зависимости от длины текстовых фрагментов.

Задача решалась в программной среде Jupyter Notebook дистрибутива Anaconda, который позволяет сразу установить Python и необходимые библиотеки.

Для решения поставленной задачи использовались:

- Методы обработки естественного языка;
- Статистические характеристики текстов;
- Методы машинного обучения;
- Методы понижения размерности для возможности визуализации.

На основе полученной динамики изменения точности классификации в зависимости от длин текстовых фрагментов были сделаны соответствующие выводы об оптимальной длине текстов, используемых для обучения и тестирования моделей.

Дипломная работа выполнена автором самостоятельно.

ABSTRACT

The project contains 45 pages, 10 pictures, 3 tables, 10 sources, one appendix.

MACHINE LEARNING, STATISTICAL TEXT ANALYSIS, AUTHORSHIP DETECTION, DATA ANALYSIS, NATURAL LANGUAGE PROCESSING

In this graduate work the selection of statistical text features, classification of texts belonging to different authors was made. Dynamics of the classification accuracy according to the text length was investigated.

The problem was solved in an interactive computing environment Jupyter Notebook belonging to the Anaconda distributive that allows Python installation with the necessary packages.

For the methods were used:

- Natural language processing methods;
- Generating statistical text characteristics;
- Machine learning;
- Dimensionality reduction for purpose of visualization.

The conclusion about the optimal text length that is used for model training and testing was made on the basis of the dynamics of the classification accuracy that depends on the length of text fragments.

The thesis project was done solely by the author.

РЭФЕРАТ

У дыпломнай працы 45 старонак, 10 рысункаў, 2 табліцы, 10 крыніц, адзін дадатак.

МАШЫННАЕ НАВУЧАННЕ, СТАТЫСТЫЧНЫ АНАЛІЗ ТЭКСТА, ВЫЗНАЧЭННЕ АЎТАРСТВА, АНАЛІЗ ДАДЗЕНЫХ, АПРАЦОЎКА НАТУРАЛЬНАЙ МОВЫ

У дыпломнай працы робіцца адбор статыстычных прыкмет тэкста, класіфікацыя тэкстаў, якія належаць розным аўтарам, і даследаванне дынамікі акуратнасці класіфікацыі ў залежнасці ад даўжыні тэкставых фрагментаў.

Задача вырашалася ў праграмным асяроддзі Jupyter Notebook дыстрыбутыва Anaconda, які дазваляе адразу устанавіць Python і неабходныя бібліятэкі.

Для вырашэння пастаўленай задачы выкарыстоўваліся:

- Метады апрацоўкі натуральнай мовы;
- Статыстычныя характарыстыкі тэкстаў;
- Метады машиннага навучання;
- Метады паніжэння размернасці для магчымасці візуалізацыі.

На основе атрымленай дынамікі змены акуратнасці класіфікацыі ў залежнасці ад дліны тэкставых фрагментаў былі зроблемы адпаведныя вывады аб аптымальнай даўжыні тэкстаў, якія выкарыстоўваюцца для абучэння і тэста маделяў.

Дыпломная праца выканана аўтарам самастойна.

ВВЕДЕНИЕ

В последнее время можно наблюдать тенденцию поиска и определения структур, которые характерны для текстов, принадлежащих различным авторам. Печатный текст делает невозможным проведение почерковедческой экспертизы, которая позволила бы определить принадлежность текста конкретному человеку. Поэтому встал вопрос о возможности определения авторского стиля при отсутствии возможности изучения почерка, опираясь на сам текст и его содержание. В связи с этим необходимо было выявить признаки, которые смогут решить задачу определения авторства для нерукописного текста. Были проведены исследования, в которых применялись статистические, формально-количественные методы, позволяющие выявить характерные для авторов черты. Одними из первых данной задачей занимались Н.А. Морозов [1] и А.А. Марков [2]. Помимо этого, выделялись и совершенствовались новые методики определения авторского стиля.

Следует отметить, что все данные исследования ставили перед собой одну цель: установление авторства неизвестных текстов, полагаясь на имеющуюся выборку авторов. Для этого осуществлялось выделение признаков, на основе которых происходило обучение моделей, осуществляющих классификацию текстов. Но не исследовалась динамика точности классификации текстов при изменении длины классифицируемого текста. В рамках данной дипломной работы производился отбор признаков, частотный анализ текстов различной длины и обучение модели для получения описанной выше динамики.

ГЛАВА 1
ОБРАБОТКА ЕСТЕССТВЕННОГО ЯЗЫКА МЕТОДАМИ МАШИННОГО ОБУЧЕНИЯ

1.1 Векторная модель текста

Для решения различных задач, где в качестве исходных данных выступают тексты, требуется преобразование этих текстов к единому виду. Для этого используется векторная модель текста (vector space model), которая ставит текстам в соответствие вектор из общего для всей коллекции текстов векторного пространства. Данная модель используется в решении задач классификации, кластеризации документов, поиска в них.

Текст состоит не только из слов, но и из других элементов, таких как знаки пунктуации, числа, специальные обозначения. Все выше перечисленные единицы текста называются термами. Изначально указывается, каким образом определяется вес терма в тексте. Упорядочение весов всех термов есть вектор, который представляет текст в векторном пространстве. При этом в данном векторе может находиться информация о термах, которые отсутствуют в конкретном документе (тексте), но присутствуют во всем корпусе документов. Таким образом, получаем набор векторов одинакового размера, представляющих все тексты коллекции.

1.2 Предобработка исходных данных

Текст, представленный в привычном для нас виде, является непригодным для извлечения из него каких-либо признаков и построения моделей. Он представляет собой строку, содержащую пунктуационные символы, которые не несут

смысловой нагрузки; может состоять из слов, представленных во всевозможных своих формах. В связи с этим мы имеем большое разнообразие отдельных единиц текста и отсутствие единого представления, единой структуры, из-за чего исходный текст не позволяет выявить различные признаки, не подвергаясь предварительной обработке.

1.2.1 Токенизация

К двум важным этапам предварительной обработки текста относятся токенизация и нормализация. Очевидно, что единицей текста является слово. Под токенизацией понимается разбиение текста на отдельные единицы-слова, которые носят название токенов. В результате токенизации исходная строка текста преобразовывается в список слов, из которых она состояла. Токенизация текста выполняется в несколько этапов:

1. Приведение исходного текста к нижнему регистру
2. Замена пунктуационных символов пробелами
3. Объявление слов отдельными токенами

Данный метод предобработки достаточно прост и понятен, но несмотря на это на каждом из этапов могут возникать различные нюансы, а именно:

1. Приведение аббревиатур к нижнему регистру может быть спутано с выражением эмоций. Например, «ООО» (Общество с ограниченной ответственностью) после обработки будет эквивалентно «ооо», которое, вероятнее всего, могло так же использоваться и для выражения удивления.
2. При замене всех знаков препинания на пробелы могут происходить потери исходных слов. Например, существуют сложные слова, которые пишутся через дефис (красно-синий, где-то). После удаления дефиса

мы получим два слова вместо исходного одного, который несет иную нагрузку. Текст, написанный в свободном стиле, может содержать смайлы, которые играют особую роль при семантическом анализе текста. При удалении знаков препинания пропадает возможность использования этого признака.

3. Имена собственные могут идти в парах. Например, названия отдельных стран, городов, организаций (Саудовская Аравия, Нижний Новгород). При токенизации они будут разделены на два отдельных слова. Но с точки зрения логики было бы полезно рассматривать их как одно целое. Аналогичный вопрос встает с сокращениями. Существуют принятые сокращения, которые при токенизации потеряются, и, более того, внесут ложную информацию в полученный набор.

Помимо этого можно выделить проблему, которая может возникнуть при токенизации текстов определенного типа. А именно, в китайском языке редко пользуются пробелами, вследствие чего текст представляет собой сплошной набор иероглифов, к которому нельзя так просто применить алгоритм токенизации. Помимо этого, в немецком языке часто используется практика образования сложных слов путем конкатенации нескольких. Для обработки подобных текстов требуется сегментация текста на слова.

1.2.2 Нормализация

После осуществления токенизации можно переходить к нормализации текста. Нормализация подразумевает под собой приведение слов к нормальной форме, где нормальная форма слова – это каноническая форма слова. Например, для существительных начальная форма слова – это форма единственного числа, стоящая в именительном падеже, для прилагательных – прилагательное мужского рода, единственного числа и стоящее в именительном падеже без предлога.

Данное преобразование не влечет особых потерь, так как конкретная форма слова редко обладает полезной информацией (смысл слова остается тем же). Часто встречаются задачи с небольшим объемом исходных данных, в связи с чем желательно уменьшить число признаков. Приводя же слова к начальной форме, мы уменьшаем количество уникальных слов.

При осуществлении нормализации можно выделить два подхода: стэмминг и лемматизация. Стэмминг занимается поиском основы слова, учитывая морфологию исходного слова. Таким образом, находя общую для всех грамматических форм слова основу, отсекая суффиксы и окончания, стэмминг осуществляет морфологический разбор слова. Алгоритм стэмминга – это конкретный способ решения задачи поиска основы слов, а стэммер – конкретная реализация. Стэмминг обладает следующим недостатком: различные формы слова могут иметь разные основы. Поэтому после стэмминга результаты обработки этих слов будут различны. В связи с этим применяется иная техника – лемматизация. В отличие от стэмминга она работает на основе словаря, где и хранятся данные о словах и их начальных формах. Это позволяет избегать ошибок, описанных выше. В случае если слова нет в словаре, то происходит построение гипотезы о способе изменения слова.

1.3 Понижение размерности

Описанные ниже алгоритмы используются для понижения размерности, что позволяет эффективно визуализировать многомерные компоненты. Из многомерной переменной мы пытаемся получить новую переменную, которая будет лежать в двух- или трехмерном пространстве и сохранять закономерности исходной переменной.

1.3.1 Алгоритм t-SNE

Название алгоритма есть сокращение с полного варианта t-distributed stochastic neighbor embedding. На русский язык можно его попробовать перевести как «t-распределенное внедрение соседей». Данный метод относится к методам множественного обучения признаков. Классическое представление алгоритма SNE было изложено в 2002 году Ровейсом и Джеффри Хинтоном, а расширение t-SNE представлено в 2008 году Джеффри Хинтоном и Лоуренсом ван дер Маатеном.

1.3.1.1 Математическое описание алгоритма

Будем строить биективное отображение, то есть каждая точка будет представлять один экземпляр (строку) исходного набора текстов. Чтобы визуализация могла отображать разделение на классы, мы хотим, чтобы точки, принадлежащие одному классу, располагались рядом, что и покажет формула (1.1):

$$p_{j|i} = \frac{\exp\left(\frac{-|x_i - x_j|^2}{2\sigma_i^2}\right)}{\sum_{k \neq j}^{n} \exp\left(\frac{-|x_i - x_k|^2}{2\sigma_i^2}\right)} \qquad (1.1)$$

Формула (1.1), основанная на гауссовском распределении вокруг точки x_i с заданной дисперсией σ_i определяет условное сходство двух точек. Дисперсии вычисляются таким образом, чтобы точки, расположенные в областях с малой плотностью, имели большую дисперсию, чем точки, расположенные в областях с большой плотностью. Для этого используется оценка перплексии (1.2). Обычно эта оценка используется для сравнения вероятностных моделей, при этом низкое значение перплексии означает, что распределение вероятностей (закон, описывающий область значений случайной величины и вероятности их исхода) хорошо работают на этапе предсказания. В нашем случае можно ее

интерпретировать как сглаженную оценку эффективного количества «соседей» для конкретной точки.

$$Perp(P_i) = 2^{H(P_i)} \tag{1.2}$$

Для этого вычисляется энтропия Шеннона в битах (1.3):

$$H(P_i) = -\sum_j p_{j|i} \log_2 p_{j|i} \tag{1.3}$$

Сходство двух точек будет определяться через формулу (1.4) условного сходства двух точек:

$$p_{j|i} = \frac{p_{j|i} + p_{i|j}}{2\mathrm{N}} \tag{1.4}$$

На основе данной формулы вычисляется матрица сходства для исходных данных. Данная матрица является постоянной.

В отличие от матрицы сходства исходных данных, матрица сходства для отображения зависит от точек отображения. Близость этих двух матриц будет нам доказывать, что похожие исходные точки отображаются в также похожие точки. При определении матрицы сходства для точек отображения (1.5) вместо гауссовского распределения используется распределение Стьюдента с одной степенью свободы или распределение Коши. Причина данной замены заключается в следующем: расстояние между парой точек в пространстве отображения, которые соответствуют паре среднеудаленных точек в исходном пространстве, должно быть намного больше, чем расстояние, которое можно получить при помощи гауссовского распределения. А метод пытается воспроизвести одинаковые расстояния в обоих пространствах, и возникает проблема

«скученности». Благодаря тяжелым «хвостам» распределения Стьюдента дисбаланс в распределении расстояний для соседей точек скомпенсирован (1.5).

$$q_{ij} = \frac{f(|x_i - x_j|)}{\sum_{k \neq i} f(|x_i - x_k|)}, \text{где f(y)} = \frac{1}{1 + y^2} \quad (1.5)$$

В ходе алгоритма происходит минимизация расстояния между матрицами сходства, осуществляемая за счет минимизации расстояния Кульбака-Лейблера между распределениями p_{ij} и q_{ij} (1.6):

$$\text{KL(P||Q)} = \sum_{i,j} p_{ij} log \frac{p_{ij}}{q_{ij}} \quad (1.6)$$

Для минимизации данного расстояния применяется градиентный спуск. Под градиентом можно понимать сумму всех сил, которые приложены к точке отображения i. В формуле (1.7) u_{ij} обозначает единичный вектор, который идет от y_j к y_i.

$$\frac{\partial \text{KL(P||Q)}}{\partial y_i} = 4 \sum_j (p_{ij} - q_{ij}) g(|x_i - x_j|) u_{ij}, \text{где g(y)} = \frac{y}{1 + y^2} \quad (1.7)$$

1.3.1.1 Физическая аналогия

Данный алгоритм можно интерпретировать с физической точки зрения следующим образом. Считаем, что все точки полученного отображения соединены между собой пружинами, обладающими разными жесткостями. Жесткости пружин зависят от величины p_{ij} - q_{ij}, которая обозначает разность сходства пары точек исходных данных и сходства пары точек отображения. Точки отображения притягиваются, если расстояние между точками данных малое, а между точками отображения большое. Отталкиваются в противном случае. Градиент является результирующей силой, которая действует на точку в пространстве отображения. Мы отпускаем систему, и она будет изменяться до

достижения равновесного состояния. Отображение на данном шаге и будет являться тем, к которому мы и стремимся.

1.3.2 Метод главных компонент

Метод главных компонент (PCA – Principal Components Analysis) был изобретен Пирсоном в 1901 году. В нем происходит вычисление собственных векторов и собственных значений ковариационной матрицы исходных данных или находится сингулярное разложение матрицы.

1.3.2.1 Описание метода

Изначально мы работаем с матрицей X, которая имеет размерность I×J, I обозначает количество образцов, J – количество независимых переменных. Матрица X раскладывается в произведение двух матриц T и P (1.8):

$$X = TP^t + E = \sum_i^A t_i p_i^t + E \tag{1.8}$$

Матрица T представляет собой переменные, являющиеся комбинацией исходных переменных x (1.9), а именно:

$$t_i = p_{i_1} x_1 + \cdots + p_{i_J} x_J, \text{где i} = 1,..\text{A} \tag{1.9}$$

Матрица T, имеющая размерность I×A, называется матрицей счетов. Эта матрица показывает структуру исходных данных. Она заключает в себе проекции исходных образцов (векторы $x_1,\ldots x_I$ размерности J) на A-мерное подпространство главных компонент. Строки матрицы представляют собой координаты образцов в новой системе координат, а столбцы – проекции всех образцов на новую координатную ось. Если отобразить матрицу счетов на графике, то близкие точки будут являться схожими между собой, то есть обладать положительной

корреляцией; диаметрально противоположные точки будут обладать отрицательной корреляцией; а точки, расположенные под прямым углом, будут иметь нулевую корреляцию. Собственные значения матрицы T^tT показывают важность соответствующих им главных компонент.

Матрица P перехода из исходного пространства (J-мерное) в пространство главных компонент (A-мерное), имеющая размерность J×A, есть матрица нагрузок. График матрицы показывает зависимость переменных между собой. Строка матрицы нагрузок представляет собой проекцию переменных $x_1 \dots x_J$ на соответствующую ось главных компонент. Столбец же матрицы есть проекция соответствующей переменной x_j на новую систему координат. Для матрицы P верно (1.10):

$$P^tP = I \tag{1.10}$$

Матрица E размерности I×J есть матрица остатков. Визуально разложение можно представить следующим образом:

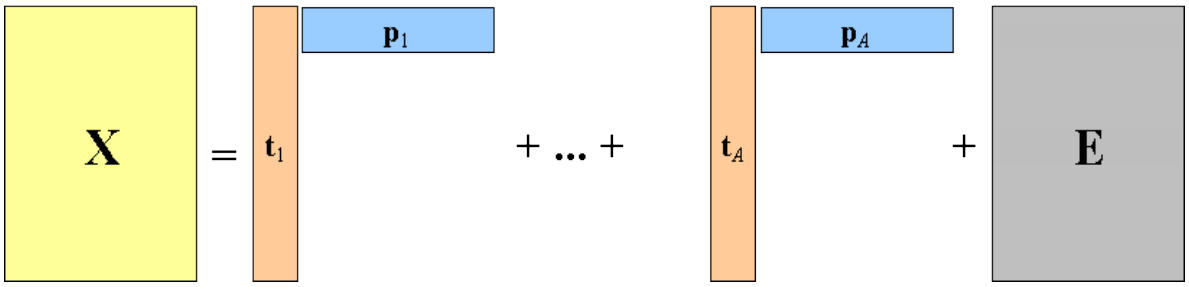

Рисунок 1.1 Сингулярное разложение

Главными компонентами будут являться новые переменные t_i. Количество главных компонент равно числу строк в матрице T и числу столбцов в матрице P, и равно A. Так как мы понижаем размерность, это число будет меньше количества образцов I и числа переменных J.

Понижая размерность, мы хотим выделить признаки, которые будут независимы друг от друга. Поэтому данный метод обладает свойством ортогональности. Если мы увеличиваем число главных компонент, то к матрице счетов T добавляется столбец, соответствующий новой компоненте (новому направлению), при этом имеющаяся часть матрицы T не изменяется. Матрица нагрузок P аналогично матрице T не перестраивается.

1.3.2.2 Сингулярное разложение

При разложении по сингулярным значениям (1.11) матрица X разлагается в произведение трех матриц:

$$X = USV^T \tag{1.11}$$

Матрица U образована ортонормированными собственными векторами u_i матрицы XX^t, которые соответствуют собственным значениям λ_i , то есть $XX^t u_i = \lambda_i u_i$. Матрица V образована ортонормированными собственными векторами v_i матрицы $X^t X$, а именно: $X^t X v_i = \lambda_i v_i$. Матрица S – положительно определенная диагональная матрица. Элементы матрицы S – сингулярные значения $\sigma_1 \geq \cdots \geq \sigma_i \geq 0$, где $\sigma_i = \sqrt{\lambda_i}$.

Метод главных компонент и сингулярное разложение связаны между собой согласно формуле (1.12):

$$T = US \text{ и } P = V \tag{1.12}$$

ГЛАВА 2
СТАТИСТИЧЕСКИЙ АНАЛИЗ ТЕКСТА

2.1 Извлечение признаков из предобработанного текста

Для решения определенной задачи исходный текст либо предобработанный не является достаточным для обучения модели. Для более глубокого изучения исходного текста необходимо обратиться к методам, которые позволят выявить некоторые признаки, характерные для имеющихся данных.

2.1.1 Мешок слов

Название модели текста «мешок слов» есть перевод с английского «bag-of-words». Она была предложена в 1975 году Солтоном. Данная модель текста представляет собой суммативное единство слов, составляющих исходный текст. Единицы «мешка слов» – слова, каждое из которых имеет атрибут, а именно: количество встреч данного слова в тексте.

Важными особенностями данной модели является отсутствие учета порядка слов в документе и морфологических форм представления слов.

Описать работу «мешка слов» можно следующим образом:

1. Дана выборка, состоящая из n различных слов: $w_1, \dots, w_n$.
2. Текст кодируется при помощи n признаков, где i-тый признак обозначает долю вхождений слова w_i среди всех вхождений слов в текст.

Обычно «мешок слов» применяется уже к предобработанному тексту, из которого были исключены все стоп-слова. Данные слова не несут в себе

смысловую нагрузку, поэтому часто доля их вхождений в текст не учитывается. Аналогично предлагается не учитывать в данной модели редкие слова, так как они, имея слишком малый вес, не смогут внести вклад в построенную модель.

2.1.2 Мешок термов

«Мешок термов» считается обобщением модели «мешок слов», названия происходит от английского «bag-of-terms». В отличие от мешка слов, его элементом является терм, который характеризуется частотой встречаемости в тексте. Под термом понимается символьное выражение объекта формальной модели (системы, языка). В качестве них могут использоваться всевозможные символьные выражения текста.

2.1.3 TF-IDF метод векторизации текстовых данных

Метод TF-IDF Vectorizer используется с целью назначения весов словам исходных текстов. Он основан на двух предположениях, а именно:

1. Слова, которые часто встречаются в тексте, важны для данного текста.
2. С другой же стороны, слово, которое редко встречается в других текстах (документах), оно важно для текущего. Иными словами, можно сказать, что данное слово может быть признаком, по которому можно идентифицировать данный документ среди остальных.

Данные предположения можно представить следующими формулами:

$$TDF(d, w) = n_{d\omega} \tag{2.1}$$

где n_{dw} обозначает долю вхождений слова w в текст (документ) d.

$$IDF(d,w)=log\frac{l}{n_\omega} \quad (2.2)$$

Где l – общее количество текстов, а n_w обозначает количество текстов, в которых встречалось слово w. Можно проанализировать, что если данное слово можно встретить в каждом тексте, то есть $l = n_\omega$, то тогда значение признака $IDF(d,w)=log1 = 0$. Это может свидетельствовать о том, что если слово встречается очень часто на всем корпусе документов, то оно вряд ли важно.

Тогда общая формула мне метода TF-IDF примет вид:

$$TDF - IDF(d,w)= n_{d\omega}log\frac{l}{n_\omega} \quad (2.3)$$

На основе данных формул можно сделать вывод, что результат выражения будет максимальным, если слово много раз встречается в тексте d, и число вхождений его в остальные документы минимально.

2.1.4 N-граммы

Достоинство n-грамм заключаем в том, что они позволяют учитывать порядок слов, в отличие от «мешка слов». Помимо этого, использование n-грамм расширяет признаковое пространство благодаря учету словосочетаний. Поэтому простые модели могут служить для поиска более сложных закономерностей по сравнению с «мешком слов».

N-граммы можно разделить на буквенные и словесные. Под словесными n-граммами понимаются наборы из n идущих подряд токенов. Среди n-грамм можно выделить:

- Униграммы: наборы, состоящие из одного токена;
- Биграммы: наборы, состоящие из двух подряд идущих токенов;

- Триграммы: наборы, состоящие из трех подряд идущих токенов и т.д.

Рассмотрим их построение на примере фразы «обработка естественного языка методами машинного обучения». Тогда:

- Униграммы: обработка, естественного, языка, методами, машинного, обучения;
- Биграммы: обработка естественного, естественного языка, языка методами, методами машинного, машинного обучения, и т.д.

В зависимости от выбора параметра n можно получить как огромное число признаков, так и свести всё к тому, что каждый текст будет сам по себе являться отдельным признаком (что будет означать подгонку под исходную выборку).

В буквенных n-граммах в качестве токенов рассматриваются буквы. Остальной процесс выделения униграмм, биграмм и т.д. аналогичен словесным биграммам.

Также можно обратить внимание на расширенную версию n-грамм, а именно k-skip-n-граммы. Это наборы из n токенов, где между соседними токенами должно быть не более чем k токенов. В качестве примера можно рассмотреть 1-skip-2-граммы. Тогда для приведенной ранее фразы получим следующий результат: обработка языка, естественного методами, языка машинного, методами обучения.

2.1.5 Хэширование

Перед рассмотрением данного метода следует привести определение хэш-функции. Поэтому h(x) – хэш-функция, которая принимает 2^n возможных значений. Данный метод подразумевает замену всех слов x на их хэши h(x) и использование этих хэшей как токенов. После этого можно применять описанные ранее методы, как «мешок слов», TF-IDF векторизатор и т.д.

Применение хэширования обладает рядом преимуществ. В первую очередь, это упрощение хранения модели. В данном случае значение хэша можно отождествить с индексом конкретного слова (токена). Например, при использовании «мешка слов» необходимо сохранять соответствие между словами и признаками. Также при использовании данного метода происходит сокращение числа признаков (возможность объединять исходные слова с одинаковыми хэшами).

2.1.6 Стоп-слова

Стоп-слова – это слова, которые встречаются в большом объеме текстов и не несут особой смысловой нагрузки. Обычно к ним относят междометия, предлоги, частицы, некоторые местоимения, прилагательные и другие части речи. Поэтому чаще всего предобработанный текст очищают от стоп-слов.

В качестве примеров можно отметить следующие группы:

- Междометия: ах, ух, ну, уж, ой;
- Местоимения: я, мы, мой, вы, ваш;
- Вводные конструкции: скажем, допустим, например, в общем, на самом деле;
- Обобщения и неточные определения: всего, примерно, около, где-то, порядка и другие.

2.2 Выделение характеристик

Очевидно, что из текста можно выявить бесконечное число характеристик. Можно начать с подсчета числа предложений и дойти до количества запятых на абзац текста. Главный вопрос заключается в том, какие характеристики можно будет считать релевантными. Поэтому в данном вопросе можно опираться на

логику, и исходя из нее совершать отбор признаков. Признаки, извлекаемые из текста, можно, как минимум, разделить на три группы:

1. Признаки, основанные на знаках препинания;
2. Признаки, базирующиеся на словах как единицах текста;
3. Признаки, где единицей является отдельный символ – буква.

Первая группа может исследовать распределение знаков препинания по тексту, взаимосвязь между знаками препинания и общим числом символов, слов в тексте.

Вторая группа дает возможность получить признаки, основанные на длинах слов, частотах употребления различных частей речи, длинах предложений, важности слов, «стоп-словах» либо уникальных словах.

Третья группа, например, позволит исследовать отношение между прописными и строчными буквами, строить биграммы и др.

ГЛАВА 3
ЗАДАЧА ОПРЕДЕЛЕНИЯ АВТОРСТВА

3.1 Постановка задачи

Задача определения авторства состоит в следующем: перед нами есть набор текстов (фрагменты произведений авторов) на русском или другом языке. Имеется неидентифицированный текст, но известно, что он принадлежит кому-то из перечисленных выше авторов. Необходимо определить автора данного текста. Будем исследовать динамику точности классификации текстов при изменении длины классифицируемого текста. Для решения данной задачи будут использованы методы, описанные выше, а также будут отбираться признаки и использоваться в сочетании с другими подходами для выявления сочетания, которое позволит получить наилучший результат.

3.2 Используемые технологии

Для реализации поставленной задачи в качестве программной среды использовался дистрибутив Anaconda, который позволяет сразу установить Python и необходимые библиотеки. Средой для выполнения задачи был выбран Jupyter Notebook. Основные пакеты, используемые в ходе исследования:

- Numpy, pandas, scikit-learn для анализа и обучения моделей;
- Matplotlib, seaborn для визуализации.

3.3 Построение тренировочного и тестового множеств

В сборе и построении данных, используемых для обучения, можно выделить следующие основные этапы:

1. Сбор текстов авторов;
2. Загрузка текстов;
3. Разбиение текстов на равные участки, предварительная обработка;
4. Конструирование тренировочного и текстового наборов.

Сначала производился сбор произведений для отобранных авторов и загрузка их из электронной библиотеки. Затем из текстов удалялись служебные символы, препятствующие дальнейшему анализу.

Все произведения для отдельного автора объединялись в единую строку, из которой затем нарезались строки определенной длины. Динамика точности классификации текстов исследовалась в зависимости из изменения длины данных строк. Во избежание пересечения тестового и тренировочного наборов, отрезки для тренировочного набора отбирались из первой половины строки, объединяющей произведения конкретного авторов, а тестового – из второй соответственно.

3.4 Решение задачи

Как говорилось ранее, изначально весь текст собирался по группам для каждого автора и подвергался предобработке, где удалялись служебные символы, не несущие никакой смысловой нагрузки. Так как нам важно исследовать динамику точности классификации текстов в зависимости от изменения длины текстов, генерировались множественные тестовые и тренировочные выборки для

различных длин текста L, где L принимало значения из множества {20000, 10000, 5000, 1000, 500, 200}.

Для обучения модели необходима векторизация текста. Для полученных текстов рассчитываются различные характеристики, которые затем представляются в качестве единого вектора. Длины векторов для всех элементов получаются равными между собой. Очевидно, что признаки могут быть разными, иметь разные единицы измерения и лежать в разных интервалах. Поэтому при построении вектора необходимо производить нормировку координат. Нормировка координат производится глобально и основывается на векторах, рассчитанных для тренировочной выборки. Производилось центрирование выборки: линейно сдвигалась выборка так, чтобы средние значения признаков были равны нулю. Дисперсия чувствительна к масштабированию, то есть сильно зависит от порядков случайной величины. Поэтому рекомендуется стандартизировать признаки, если их единицы измерения сильно отличаются своими порядками. Стандартизация обычно требуется для обучающих алгоритмов: качество обучения заметно снижается, если отдельный признак не обладает нормальным распределением. Многие алгоритмы (например, метод опорных векторов с RBF ядром, L1 или L2 регуляризация линейных моделей) ожидают на входе центрированные признаки с центром в нуле и с одинаковым распределением. Если какой-то признак имеет дисперсию, которая на несколько порядков больше, чем дисперсия других признаков, она может доминировать над целевой функцией. В связи с этим модель не сможет обучаться на других признаках так, как предполагалось.

Для решения задачи определения авторства необходимо из огромного числа характеристик (признаков) выбрать те, которые будут использоваться для анализа текстов. На определенных этапах отбора признаков текст приводился к нижнему регистру, осуществлялась токенизация и лемматизация.

Вставал вопрос, какие признаки могли бы охарактеризовать стиль и манеру письма отдельного автора. В качестве основных были отобраны следующие статистические характеристики текста:

- Отношение количества заглавных букв к количеству строчных букв;
- Распределение различных знаков препинания по тексту. Производился подсчет вхождения каждого символа из заданного набора в текстовые строки.
- Распределение длин предложений. В качестве длины предложения бралось количество слов в нем. Для этого изначально проводилась токенизация и очистка предложений от знаков препинания. Полученные длины предложений подвергались нормализации;
- Распределение длин слов. Определялись длины слов, входящих в текст, и изучалось распределение подсчитанных длин.
- Определялась водность текста. Для этого рассчитывалось отношение количества слов после очистки текста от стоп-слов к количеству слов в исходном тексте. Разность единицы и данной величины и определяет водность текста. Например, текст, состоящий полностью из стоп-слов, будет иметь водность, равную единице; а текст, не содержащий стоп-слов, будет определяться нулевым значением водности. Считается, что неестественные тексты могут обладать повышенной водностью.
- Разнообразие речи. Под ней понимается отношение количества уникальных слов к общему количеству слов в тексте.
- Распределение частей речи в тексте. Для этого были отобраны наиболее часто встречающиеся части речи, осуществлялась нормализация текста, после чего производился подсчет встреч каждой части речи в предобработанном тексте.

Помимо этих признаков, рассчитывались буквенные биграммы для корпусов текстов. На основе тренировочной выборки определялись наиболее часто встречающиеся биграммы, их частоты использовались в качестве признаков. При тестировании подсчитывались частоты отобранных на этапе обучения наиболее часто встречающихся биграмм.

3.5 Критерии качества в задачах классификации

Для оценки качества моделей в задачах машинного обучения используются различные метрики. Для начала следует обратиться к понятию матрицы ошибок (confusion matrix). Выделяется два вида ошибок классификации: False Negative (FN) и False Positive (FP). Под ошибкой False Negative, или ошибкой второго рода, понимается неотнесение объекта к классу, которому он на самом деле принадлежит. Ошибка False Positive, или же ошибка первого рода, предполагает отнесение к данному классу объекта, который в реальности ему не принадлежит. Обозначим $\hat{y}$ответы модели, а y – истинная метка класса. Ошибки классификации для двух классов можно представить в следующей таблице:

	y = 1	y = 0
$\hat{y} = 1$	True Positive (TP)	False Positive (FP)
$\hat{y} = 0$	False Negative (FN)	True Negative (TN)

Таблица 3.1 Ошибки классификации

Метрика accuracy, или же аккуратность обозначает долю правильных ответов алгоритма и рассчитывается по формуле (3.1):

$$Accuracy = \frac{TP + TN}{TP + TN + FP + FN} \quad (3.1)$$

Определим метрики точность (precision) по формуле (3.2) и полноту (recall) по формуле (3.3):

$$Precision = \frac{TP}{TP + FP} \quad (3.2)$$

$$Recall = \frac{TP}{TP + FN} \quad (3.3)$$

Точность определяется как отношение объектов, верно классифицированных как принадлежащих классу, по отношению ко всем объектам. Полнота показывает отношение верно положительно классифицированных объектов ко всем объектам положительного класса, которые были найдены алгоритмом. Последние две метрики не зависят от соотношения классов, что является преимуществом по сравнению с accuracy, которая будет показывать нехорошие результаты для несбалансированных выборок.

Для определения точности классификации удобно иметь единую метрику, поэтому применяются различные комбинации метрик precision и recall. Чтобы можно было определять вес каждой метрики в итоговой, добавляется параметр β, который будет отвечать за вес метрики precision. Логично, что при метриках, близких к единице, итоговая метрика должна так же достигать максимума, и, наоборот, при малом значении какой-либо из метрик F_β должна стремиться к нулевому значению. В качестве примера можно рассмотреть F-меру (3.4) как среднее гармоническое этих двух метрик:

$$F_\beta = (1 + \beta^2)\frac{precision * recall}{(\beta^2 * precision) + recall} \quad (3.4)$$

Еще одной возможностью оценить качество модели является площадь (Area Under Curve) под кривой ошибок (Receiver Operating Characteristic curve) – ROC AUC. Кривая ошибок изображается в координатах True Positive Rate (TPR) и FPR (False Positive Rate), значения в точках которой вычисляются по формулам (3.5) и (3.6):

$$TPR = \frac{TP}{TP + FN} \quad (3.5)$$

$$FPR = \frac{FP}{FP + TN} \quad (3.6)$$

True Positive Rate эквивалентно полноте, а False Positive Rate отображает, какая доля объектов, не принадлежащих данному классу, была предсказана неверно. Площадь под полученной кривой будет являться метрикой, показывающей качество алгоритма. Логично, что наилучшим случаем являются значения TPR = 1 и FPR = 0, при котором площадь под кривой будет равна единице.

Следует обратить внимание на логистическую функцию потерь (logistic loss). Здесь используются ранее введенные обозначения: $\hat{y}$ – ответы модели, y – истинная метка класса, – l определяет размер выборки. Она задается формулой (3.7):

$$logloss = -\frac{1}{l}\sum_{i=1}^{l}(y_i \log(\hat{y}_i) + (1 - y_i)\log(1 - \hat{y}_i)) \quad (3.7)$$

Данная метрика может использоваться в задачах максимизации точности предсказания путем штрафования за неверно предсказанные метки классов.

3.6 Классификация текстов

3.6.1 Матрица ошибок

Для оценки полученных классификаторов строилась матрица ошибок, которая основана на таблице сопряженности. Представляя совместное распределение двух переменных, таблица сопряженности предназначена для исследования связи между ними. По одной оси располагаются оригинальные метки классов, по другой – предсказанные. Главная диагональ матрицы показывает количество верно предсказанных значений для каждого из классов. Неверно предсказанные элементы будут располагаться вне главной диагонали. Соответственно, сумма диагональных элементов есть все верно классифицированные объекты. Тогда общую точность классификации можно выразить как отношение суммы диагональных элементов (d_i) к общему количеству элементов (N) формулой (3.8).

$$Overall\ Accuracy = \frac{\sum d_i}{N} \tag{3.8}$$

Аналогичным образом можно посчитать точность определения реального (рассчитанного) класса: разделив число верно классифицированных объектов на общее количество объектов этого класса. Формула для первого класса будет иметь вид (3.9):

$$Producer's\ Accuracy = \frac{d_1}{a_{1i}} \tag{3.9}$$

Матрица ошибок строилась для более глубокого анализа результатов классификации. На рисунках (3.1) – (3.6) приведены данные матрицы, построенные на основе классификаторов, обученных на текстах различной длины.

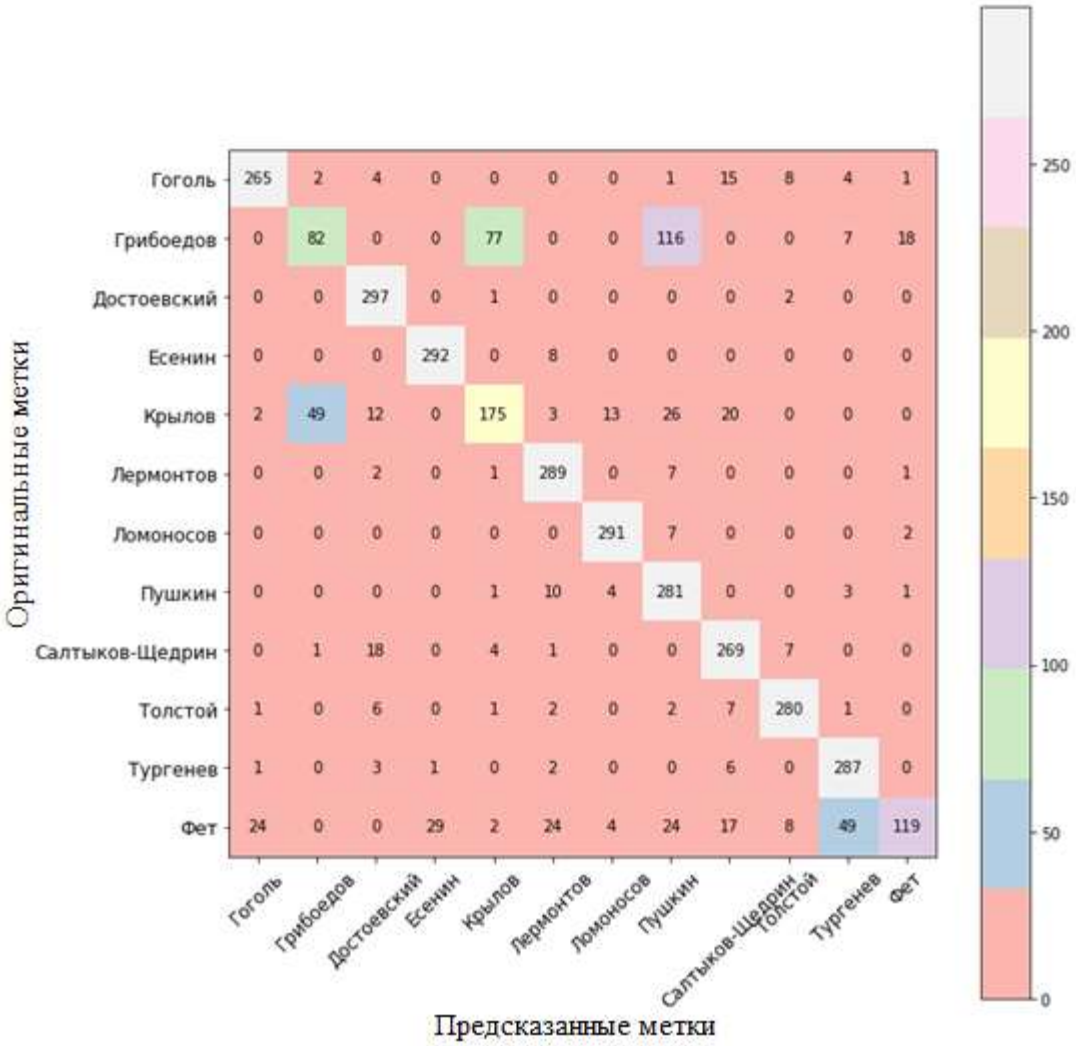

Рисунок 3.1 Матрица ошибок на текстах длиной 20000 символов

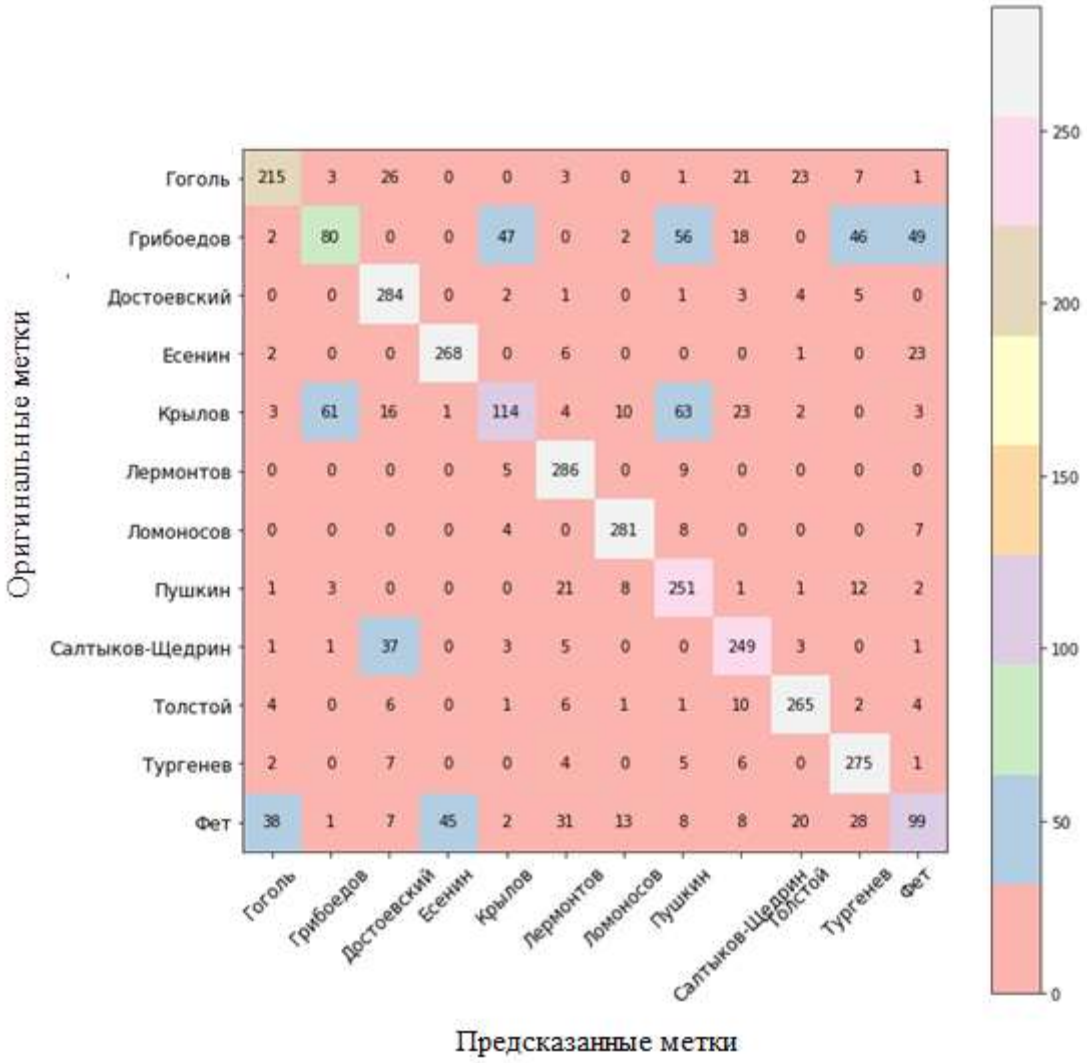

Рисунок 3.2 Матрица ошибок на текстах длиной 10000 символов

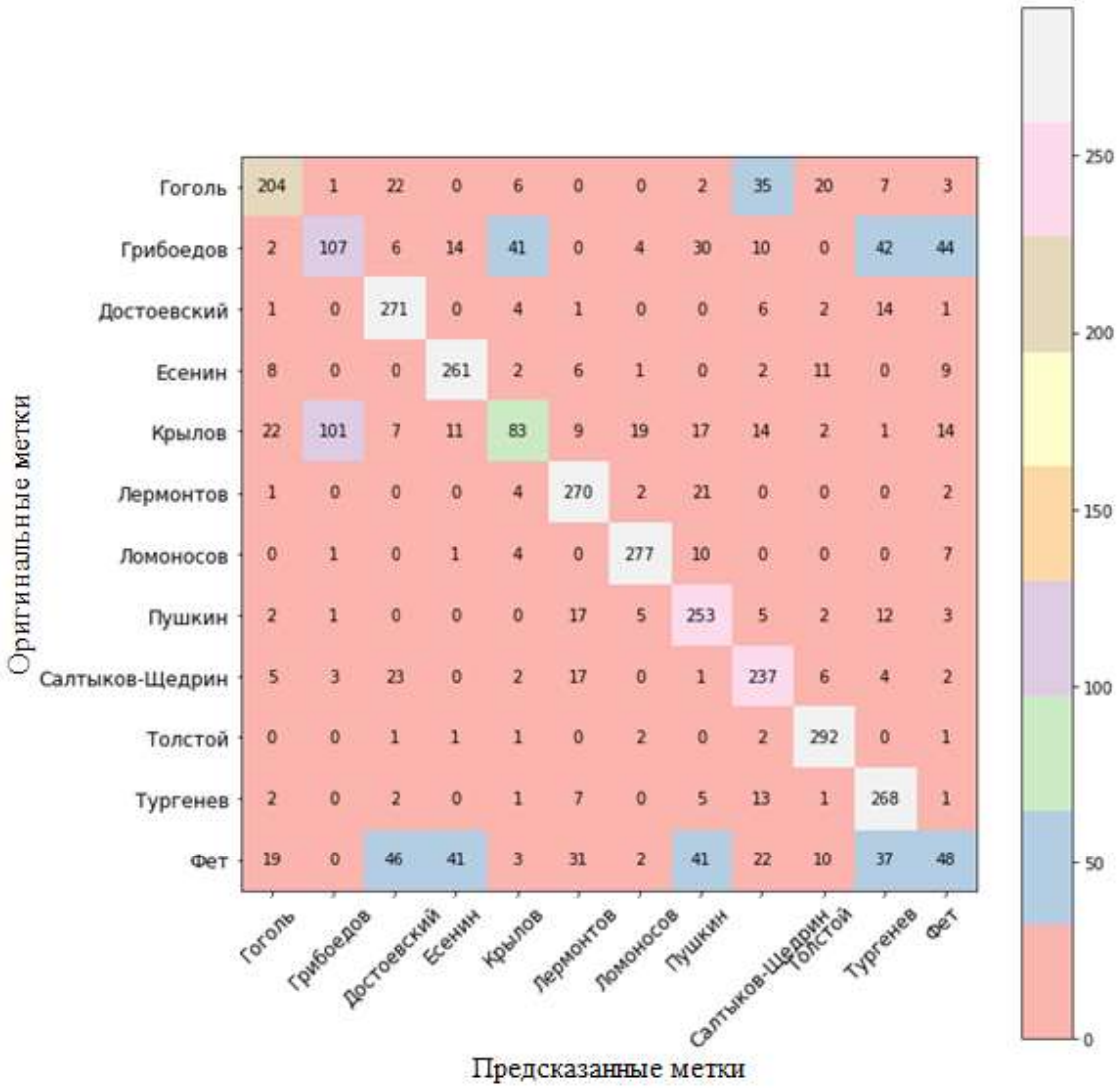

Рисунок 3.3 Матрица ошибок на текстах длиной 5000 символов

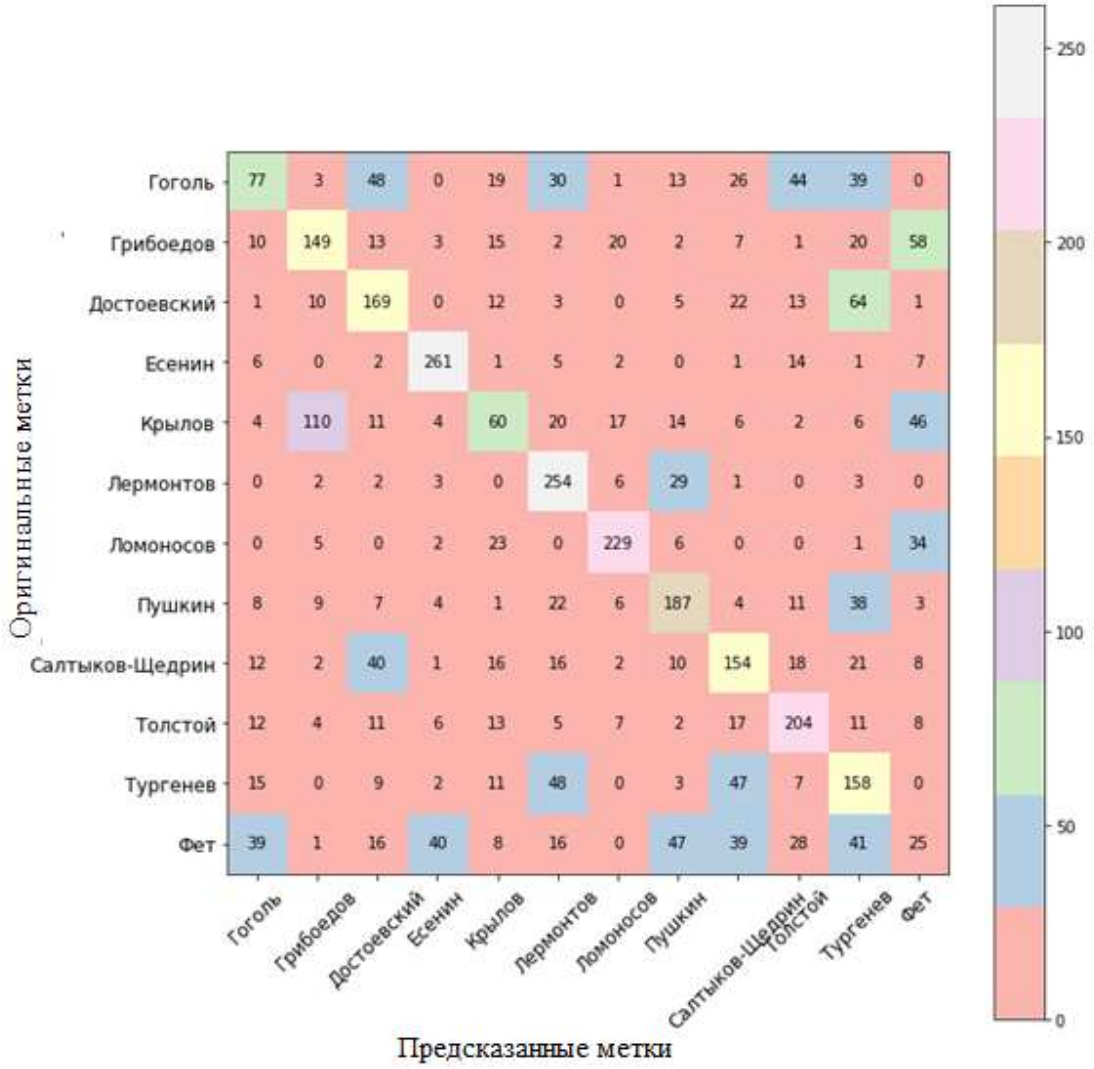

Рисунок 3.4 Матрица ошибок на текстах длиной 1000 символов

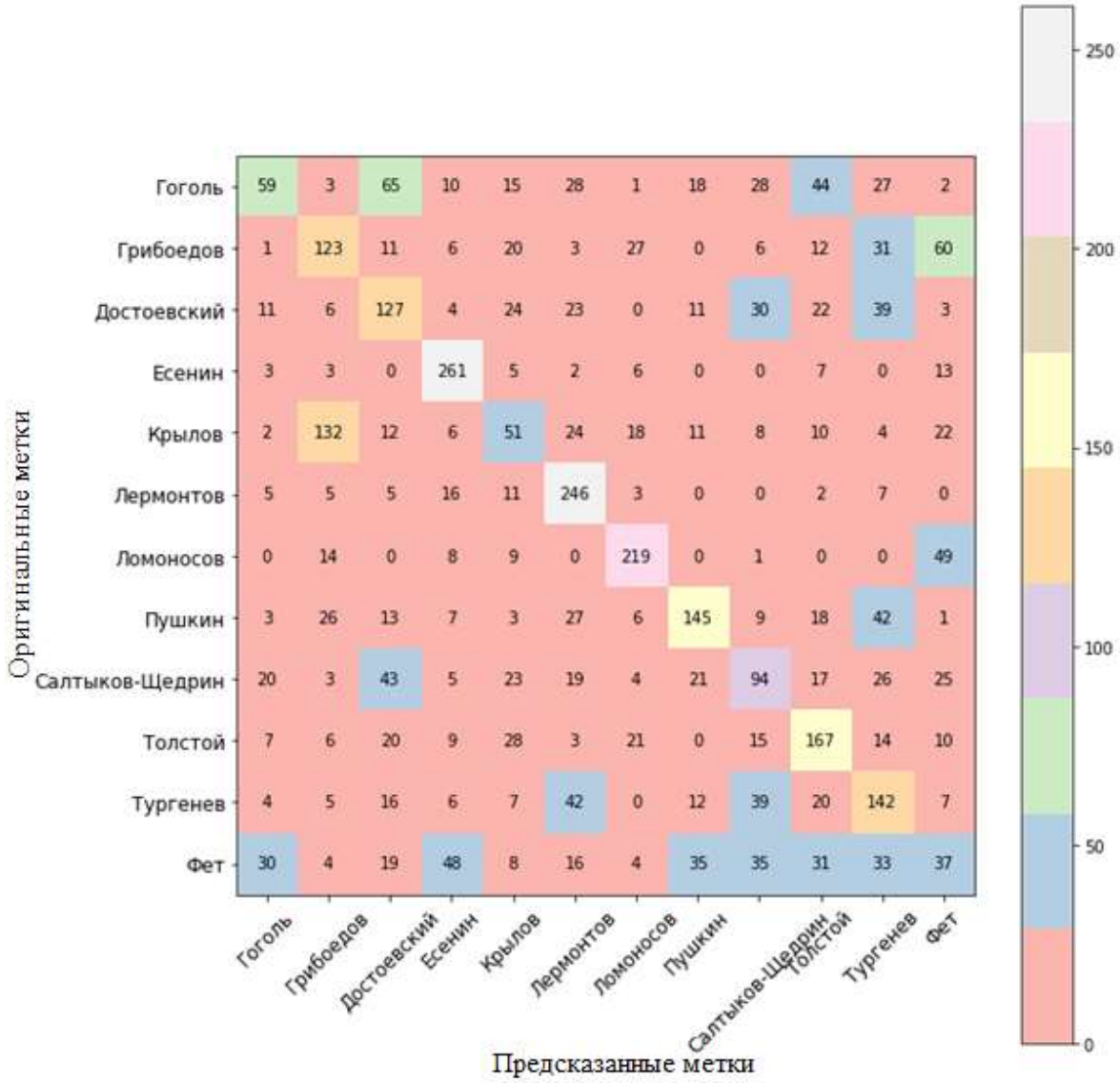

Рисунок 3.5 Матрица ошибок на текстах длиной 500 символов

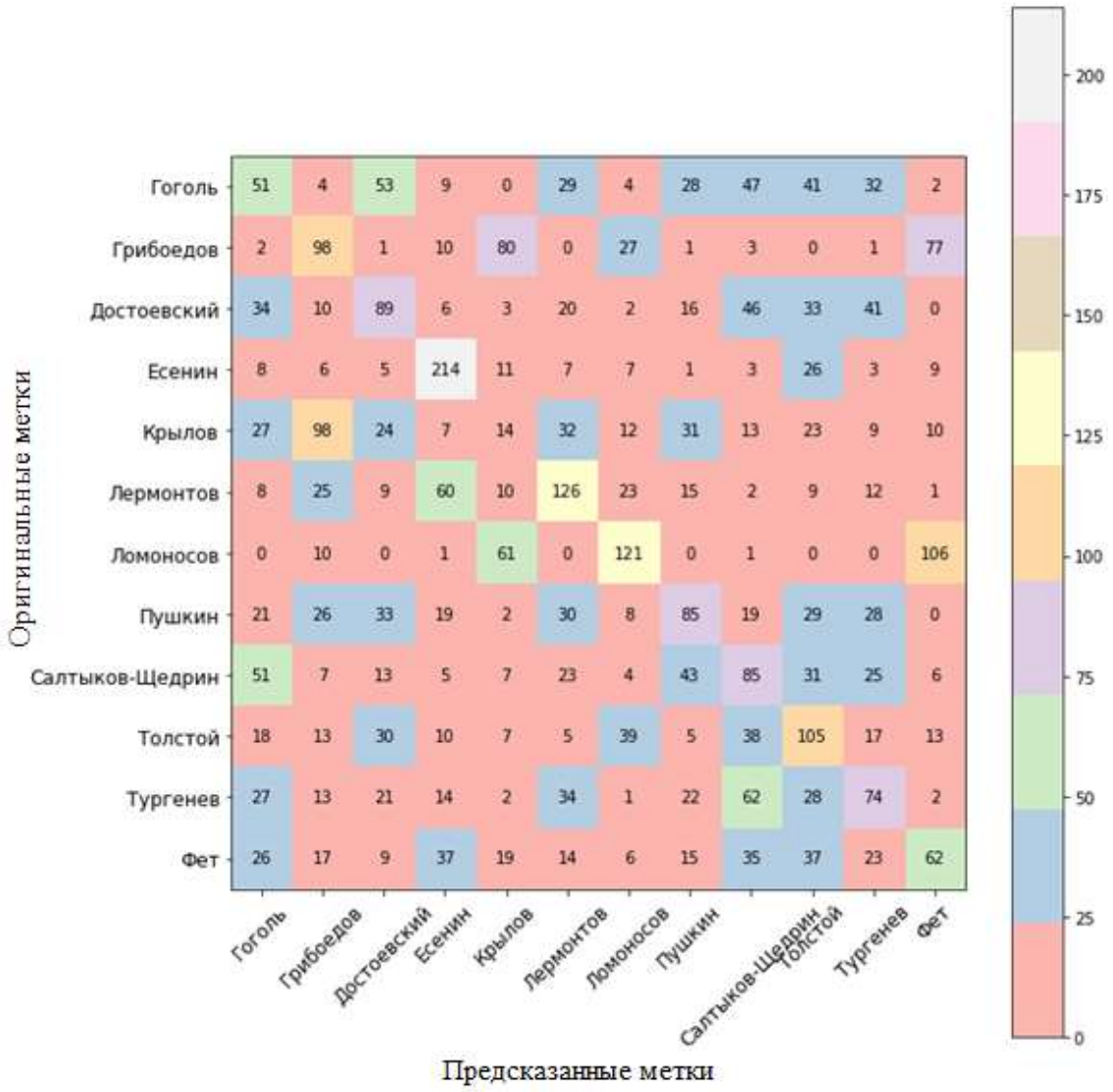

Рисунок 3.6 Матрица ошибок на текстах длиной 200 символов

3.6.2 Динамика точности классификации

Классификация текстов осуществлялась на основе векторов, построенных на статистических признаках, извлеченных из текстов, и посчитанных частотах биграмм. Для классификации использовалась различные алгоритмы: логистическая регрессия, метод k-ближайших соседей, метод опорных векторов с различными ядрами, стохастический градиентный спуск, стохастический градиентный спуск на векторизованных текстах методом TF-IDF. В качестве

основной метрики использовалась «аккуратность» (accuracy). Наилучшие результаты показали логистическая регрессия и метод опорных векторов. Для достижения наилучшей точности происходил подбор параметров оптимальных параметров с использованием GridSearchCV.

Результаты работы алгоритмов представлены в табл.3.2.

	L = 20000	L = 10000	L = 5000	L = 1000	L = 500	L = 200
Logistic Regression	0.813	0.751	0.725	0.536	0.467	0.317
KNN	0.774	0.718	0.665	0.332	0.236	0.133
SVM	0.776	0.743	0.702	0.503	0.475	0.280
Tfidf + SGDClassifier	0.709	0.721	0.733	0.606	0.534	0.332
SGDClassifier	0.753	0.721	0.664	0.492	0.395	0.255

Таблица 3.2 Точность классификации (метрика accuracy)

Из данных, представленных в таблице, можно сделать вывод, что качество классификации значительно снижается при использовании для обучения текстов длины менее 5000 символов. При этом качество классификации при увеличении длины текстов с 5000 до 20000 символов повышается незначительно. Векторизация исходных текстов - достаточно трудоемкая задача. Поэтому при отсутствии достаточных ресурсов, можно не использовать тексты наибольшей длины, не ощущая при этом особых потерь качества.

3.7 Визуализация

Мы получили длинные векторы, содержащие в себе множество признаков, вычисленных на созданной выборке. Было бы интересно посмотреть на эти данные и попытаться найти некоторые закономерности между ними. Если бы визуально полученные векторы разделялись на кластеры, соответствующие разным авторам, это бы означало, что признаки были выбрано достаточно успешно. Но возникает проблема, как можно визуализировать эти векторы, учитывая, что они имеют большую размерность. Данную проблему решают методы понижения размерности пространства. Следует отметить, что данные методы чувствительны к выбору единиц измерения. Величины, имеющие разброс, на порядок больший по сравнению с остальными признаками, вносили бы наибольший вклад при понижении размерности. Поэтому сначала производится нормировка выборки, после этого же применяется один из методов понижения размерности.

В рамках данной работы методы главных компонент и t-SNE применялись для получения наглядного представления полученных признаков. Векторы, которые состоят из статистических признаков и частот биграмм отображались на плоскости. Цвета точкам присваивались исходя из оригинальных меток (оригинальной принадлежности какому-либо автору), каждая точка обозначает конкретный текст, а именно вектор, который был получен после отображения длинного вектора (признаков текста) в пространство меньшей размерности. В связи с этим можно было наблюдать полученные распределения текстов.

Для текстов длины 20000 были получены графики, изображенные на рис.3.7, рис.3.8, рис.3.9. Каждая точка обладает цветом (номер), что означает ее принадлежность конкретному автору. Видно, что понижение размерности при помощи метода t-SNE для текстов длины 20000 помогает явно выделить разные

классы авторов. Первые два графика (рис.3.7, рис.3.8) были получены при понижении размерности до пространства $\mathcal{R}^2$, а третий график (рис.3.9) изображает компоненты в пространстве $\mathcal{R}^3$.

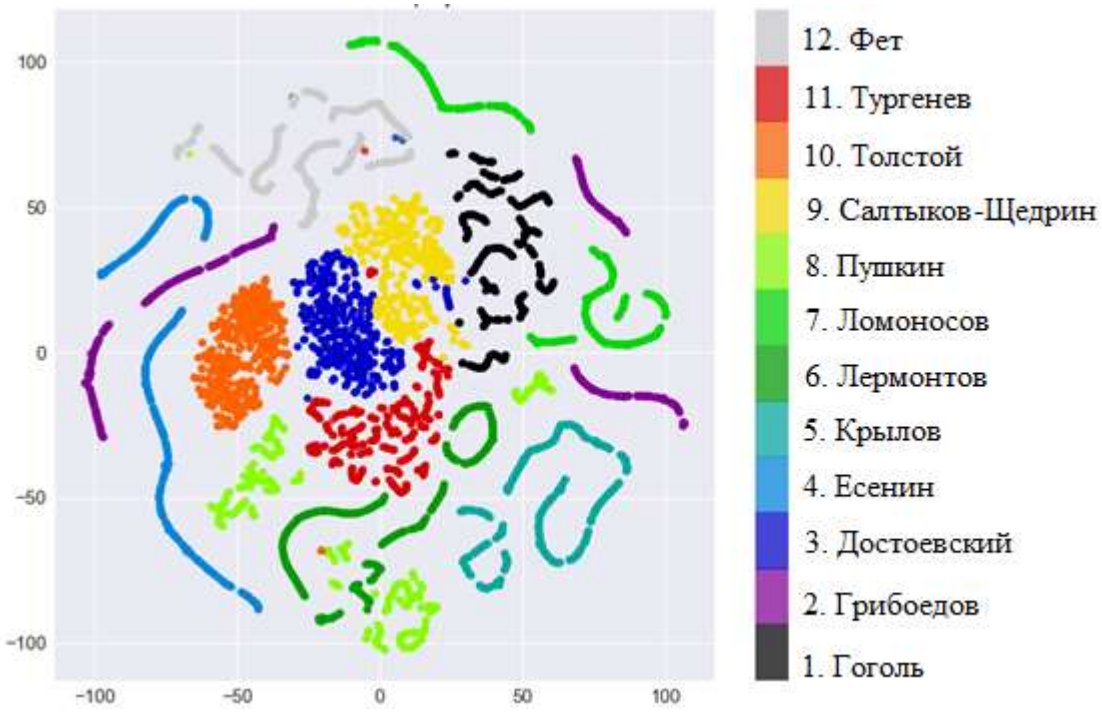

Рисунок 3.7 Алгоритм t-SNE для текстов длины 20000 символов тренировочного набора. Библиотека matplotlib.

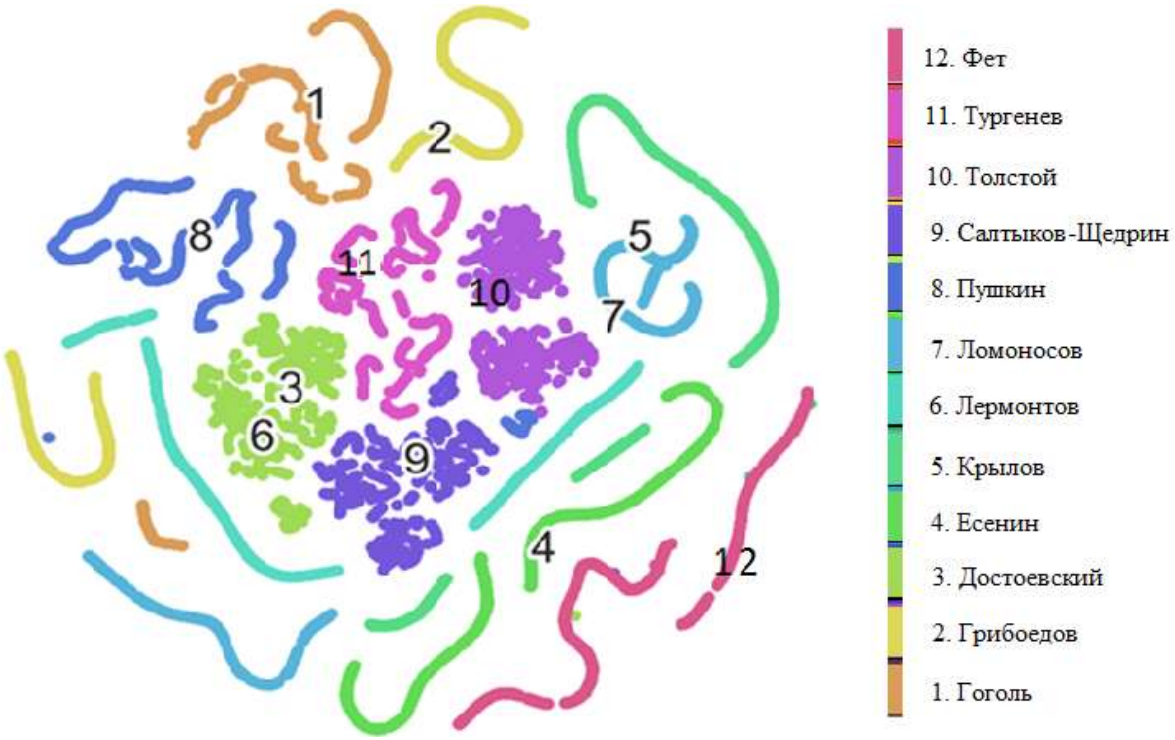

Рисунок 3.8 Алгоритм t-SNE для текстов длины 20000 символов тренировочного набора. Библиотека seaborn.

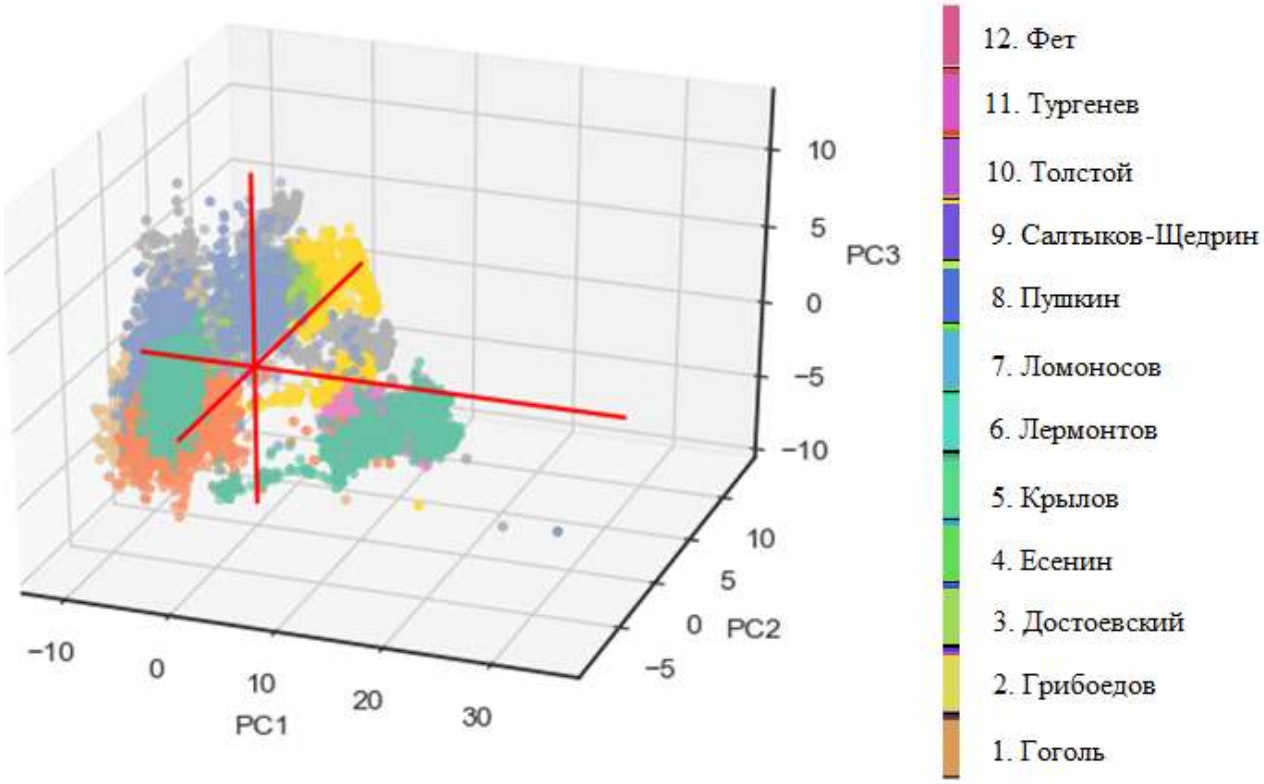

Рисунок 3.9 Метод главных компонент для текстов длины 20000 символов тренировочного набора. Библиотека seaborn.

ЗАКЛЮЧЕНИЕ

В рамках дипломной работы решалась задача классификации текстов известных авторов. Помимо этого исследовалась зависимость точности классификации от длины текста, которая не изучалась в работах, описывающих проблематику и первые подходы к решению исходной задачи. Для решения данной проблемы были поставлены и решены задачи, включающие в себя:

- Подготовку текста к анализу;
- Исследование текста как последовательности символов;
- Поиск и выделение закономерностей, способных охарактеризовать текст;
- Получение векторного представления текста;
- Визуальное представление полученных признаков;
- Построение и обучение моделей, осуществляющих классификацию текстов;
- Сравнение построенных моделей для разных корпусов текстов и определение пороговых длин текстов, при которых точность классификации изменяется незначительно либо, наоборот, претерпевает спад или подъем точности классификации.

СПИСОК ИСПОЛЬЗОВАННОЙ ЛИТЕРАТУРЫ

1. Морозов Н.А. Лингвистические спектры: средство для отличения плагиатов от истинных произведений того или иного известного автора. Стилеметрический этюд. // Известия отд. русского языка и словестности Имп.Акад.наук, Т.XX, кн.4, 1915.
2. Марков А.А. Пример статистического исследования над текстом "Евгения Онегина", иллюстрирующий связь испытаний в цепь. // Известия Имп.Акад.наук, серия VI, Т.X, N3, 1913, с.153.
3. Марков А.А. Об одном применении статистического метода. // Известия Имп.Акад.наук, серия VI, Т.X, N4, 1916, с.239.
4. T. Hastie, R. Tibshirani, J. Friedman. The Elements of Statistical Learning. Data Mining, Inference, and Prediction. 2nd Edition. - Springer, 2013.
5. Мартыненко Г. Я . Основы стилеметрии . — Л.: Изд-во ЛГУ, 1988 . — 176 с.
6. Хмелев Д.В. Распознавание автора текста с использованием цепей А.А. Маркова //Вести. МГУ. Сер. 9. Филология. 2000. №2. С. 115-126.
7. Фоменко В.П., Фоменко Т.Г. Авторский инвариант русских литературных текстов // Методы количественного анализа текстов нарративных источников. М.: Ин-т истории СССР, 1983. С. 86-109.
8. Шитиков В. К., Мастицкий С. Э. Классификация, регрессия и другие алгоритмы Data Mining с использованием R. 2017.
9. От Нестора до Фонвизина. Новые методы определения авторства. М.: Издат. группа «Прогресс», 1994.

ПРИЛОЖЕНИЕ А.

Код программы

Ниже приведен код алгоритма на Python, созданный в среде Jupyter Notebook.

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import os
import glob
import nltk, re
import collections
import random
import operator
from functools import reduce
import time
from collections import Counter
from itertools import islice
from sklearn import preprocessing
```

In [2]:

```
from nltk import word_tokenize
from nltk.stem.wordnet import WordNetLemmatizer
from nltk.corpus import stopwords
import pymorphy2
morph = pymorphy2.MorphAnalyzer()
```

Import data

In [11]:

```
FILE_PATH = "E:/0UNI/7 сем/Диплом/Выборка/Русский/"
```

In [12]:

```
folders = os.listdir(path=FILE_PATH)
folders
```

Out[12]:

```
['Гоголь',
 'Грибоедов',
 'Достоевский',
 'Есенин',
 'Крылов',
 'Лермонтов',
 'Ломоносов',
 'Пушкин',
 'Салтыков-Щедрин',
 'Толстой',
 'Тургенев',
 'Фет']
```

In [13]:

```
def read_file(path):
    """
    Read txt file.
    :param path: path to the text-file
    :return: string
    """
    with open(path, 'r') as myfile:
        data = myfile.read()
    return data
```

In [14]:

```
def pure_text(text):
    """
    Remove some punctuation
    """
    norm_text = text
    # Replace breaks with spaces
    for s in ['<br />', '\n', '&#46;']:
        norm_text = norm_text.replace(s, " ")
    # Replace punctuation with spaces
```

```
        for char in ['\\', '/']:
            norm_text = norm_text.replace(char, " ")
        return norm_text
```

In [15]:

```
    def read_files_concat(folder_path):
        """
        Read all txt-files in the given folder path.
        :param folder_path: path to a folder with txt-files.
        :return: list of strings.
        """
        file_names = [name for name in glob.glob(FILE_PATH + folder_path +'/*.*txt
')]
        read_files = [read_file(file_name) for file_name in file_names]
        text = " ".join(read_files)
        return text
```

In [16]:

```
    def save_to_txt(file_name, data):
        thefile = open(file_name, 'w')
        for item in data:
            thefile.write("%s\n" % item)
```

Train, Test

Authors encoding

In [17]:

```
    def create_author_dict(list_names):
        """
        Create a dictionary, where each author corresponds to the natural number
        """
        author_coded = {}
        ind = 1
        for folder in list_names:
            author_coded[folder] = ind
            ind += 1
        return author_coded
```

In [18]:

```
author_coded = create_author_dict(folders)
author_coded
```

Out[18]:

```
{'Гоголь': 1,
 'Грибоедов': 2,
 'Достоевский': 3,
 'Есенин': 4,
 'Крылов': 5,
 'Лермонтов': 6,
 'Ломоносов': 7,
 'Пушкин': 8,
 'Салтыков-Щедрин': 9,
 'Толстой': 10,
 'Тургенев': 11,
 'Фет': 12}
```

In [19]:

```
# author_splitted = sum(author, [])
# y = [author_coded[name] for name in author_splitted]
```

For train, test dataset constructing

In [20]:

```
def extract_substrings(s, L, N):
    """
    :param s: string
    :param n: len of the splitted strings
    :param N: number of strings to obtain
    """
    chunks, chunk_size = len(s), L
    splitted = [ s[i:i+chunk_size] for i in range(0, chunks, chunk_size) ]
    while len(splitted) < N:
        pos_start = random.randint(0, len(s) - chunk_size - 1)
        splitted.append(s[pos_start:pos_start + chunk_size])
```

```
    if len(splitted) > N:
        return splitted[:N]
    return splitted
```

In [21]:

```
def string_train_test(s, L, N, M):
    """
    :param s: string
    :param L: len
    :param N: number of strings for train
    :param M: number os strings for test
    """
    length = len(s)
    train = extract_substrings(s[: int(len(s) / 2)], L, N)
    test = extract_substrings(s[int(len(s) / 2):], L, M)
    return train, test
```

In [22]:

```
def get_author_texts(author_folder, L, N, M):
    s = pure_text(read_files_concat(author_folder))
    train, test = string_train_test(s, L, N, M)
    return train, test
```

In [23]:

```
def build_train_test_sets(folders_path, L, N, M):
    """
    :param folders_path: path to all folders
    :param L: length of strings
    :param N: number of strings for the train set from one author
    :param M: number of strings for the test set from one author
    """
    X_train, y_train = [], []
    X_test, y_test = [], []

    for folder in folders:
        cur_train, cur_test = get_author_texts(folder, L, N, M)
```

```
            X_train.append(cur_train)
            X_test.append(cur_test)
            y_train.append([author_coded[folder]] * len(cur_train))
            y_test.append([author_coded[folder]] * len(cur_test))
            # y_train.append([folder] * len(cur_train))
            # y_test.append([folder] * len(cur_test))
    return sum(X_train, []), sum(y_train, []), sum(X_test,[]), sum(y_test, [])
```

Text preprocessing

Feature Selection

Посчитать распределение различных знаков препинания в тексте

In [24]:

```
def punctuation_freq(lower_text):
    """
    Count the number of occurencies of punctuation characters in a text
    :param text: string
    :return: list with occurencies
    """
    symbols = [',', '.', '?', '!', ':', ';', '(', '-', '"', "'"]
    punct_dict = dict.fromkeys(symbols, 0) # ',.?!:;(-"\''
    if len(lower_text):
        for symbol in symbols:
            punct_dict[symbol] = lower_text.count(symbol)
        punct_dict['"'] /= 2
        punct_dict["'"] /= 2
    else:
        return list(punct_dict.values())

    return list(punct_dict.values())
```

In [25]:

```
punctuation_freq('fewepofkwkm;-()ef')
```

Out[25]:

```
[0, 0, 0, 0, 0, 1, 1, 0, 0.0, 0.0]
```

In [26]:

```
def remove_punctuation(text):
    """
    Strip punctuation/symbols from words
    :param text: string
    :return: modified string
    """
    norm_text = text
    symbols = [',', '.', '?', '!', ':', ';', '(', ')', '-', '"', "'"]
    # Replace punctuation with spaces
    for char in  symbols:
        norm_text = norm_text.replace(char, ' ')
    return norm_text
```

In [27]:

```
def word_norm(word):
    """
    Convert word to its first form
    """
    return morph.parse(word)[0].normal_form
```

In [28]:

```
POS = ['NOUN', 'ADJF', 'ADJS','COMP','VERB','INFN','PRTF','PRTS','GRND', 'ADVB
','NPRO','PRED','PREP','CONJ','PRCL','INTJ']
```

In [29]:

```
def POS_distribution(tokenized):
    """
    Распределение частей речи в  тексте
    """
    # normalized = [word_norm(word) for word in tokenized]

    # Переходим к частям речи
    words_POS = [morph.parse(word)[0].tag.POS for word in tokenized]
    POS_distr = dict.fromkeys(POS, 0) # инициализация dict для всех частей реч
и
```

```
        POS_counted = dict(collections.Counter(words_POS))
        for word in POS:
            if word in POS_counted.keys():
                POS_distr[word] = POS_counted[word]

        return list(POS_distr.values())
```

In [31]:

```
def numpy_concatenate(a):
    """
    Flatten the list with numbers
    """
    return list(np.concatenate(a))
```

In [32]:

```
STOP_WORDS = set(stopwords.words('russian'))
```

In [33]:

```
def sent_length_distribution(lower_text):
    """
    Sentence lengths distribution
    """
    # Количество предложений
    sentences = nltk.sent_tokenize(lower_text)
    sentence_num = len(sentences)

    # FEATURE Среднее количество слов в предложениях
    # удаление пунктуации
    sentences_no_punct = [remove_punctuation(sentence) for sentence in sentenc
es]
    # список из токенизированных предложений
    sent_tokenized = [word_tokenize(sent) for sent in sentences_no_punct]
    # длины предложений
    sentence_lengths = [len(sent) for sent in sent_tokenized]

    # интервалы для разбиения длин предложений
```

```
    bins=[1, 4, 7, 10, 15, 100]
    distr = list(np.histogram(sentence_lengths, bins=bins)[0])

    return distr
```

In [34]:

```
def word_length_distribution(tokenized):
    """
    Распределение длин слов
    """

    # длины слов
    word_lengths = [len(word) for word in tokenized]

    # интервалы для разбиения длин слов
    bins=[1, 4, 7, 10, 100]
    distr = list(np.histogram(word_lengths, bins=bins)[0])

    return distr
```

In [35]:

```
def capital_to_lower_count(text):
    """
    Number of Uppercase letters to the number of lowercase
    """

    lowercase_char_num = sum([i.isalpha() and i.islower() for i in text]) # ко
личество строчных символов
    uppercase_char_num = sum([i.isalpha() and i.isupper() for i in text]) # ко
личество заглавных символов
    try:
        upper_to_lower = uppercase_char_num / lowercase_char_num # отношение з
аглавных букв к строчным
    except ZeroDivisionError:
        upper_to_lower = 0
    return upper_to_lower
```

In [36]:

```
def non_stopwords(normalized):
    """
    Водность текста -
    Разница между единицей и отношением «количество слов после очистки стоп-слов/количество слов в исходном тексте».
    """

    # не стоп-слова
    filtered_words = [word for word in normalized if word not in STOP_WORDS]
    try:
        stop_words_to_full_length = 1 - len(filtered_words) / len(normalized)
    except ZeroDivisionError:
        return 0
    return stop_words_to_full_length
```

In [37]:

```
def lexical_diversity(normalized):
    """
    Unique words to the whole text length
    """

    try:
        lexical_div = len(set(normalized)) / len(normalized)
    except ZeroDivisionError:
        return 0

    return lexical_div
```

In [38]:

```
# text1 = X_test[0]
# res1 = []
# upper_to_lower1 = capital_to_lower_count(text1)
# res1.append([upper_to_lower1])
# lower_text1 = str(text1).lower()
```

```
# punct_occur_distr1 = punctuation_freq(lower_text1)
# sent_len_distr1 = sent_length_distribution(lower_text1)
# pure_text1 = remove_punctuation(lower_text1)
# tokenized1 = word_tokenize(pure_text1)
# pos_distr1 = POS_distribution(tokenized1)
# word_len_distr1 = word_length_distribution(tokenized1)
# normalized1 = [word_norm(word) for word in tokenized1]
# lexical_divers1 = lexical_diversity(normalized1)
# nonstop_to_volume1 = non_stopwords(normalized1)
```

In [39]:

```
def text_features(input_text):

    text = input_text
    res = []

    #____FEATURE____Отношение количества заглавных букв к строчным
    upper_to_lower = capital_to_lower_count(text)
    res.append([upper_to_lower])

    # Приведение к нижнему регистру
    lower_text = str(text).lower()

    #____FEATURE____Распределение знаков препинания (',.?!:;(-"\'') по тексту
    punct_occur_distr = punctuation_freq(lower_text)
    res.append(punct_occur_distr)

    #____FEATURE____Распределение длин предложений
    sent_len_distr = sent_length_distribution(lower_text)
    res.append(sent_len_distr)

    # Токенизация
    pure_text = remove_punctuation(lower_text)
    tokenized = word_tokenize(pure_text)
```

```
        #____FEATURE____Распределение частей речи в тексте
        pos_distr = POS_distribution(tokenized)
        res.append(pos_distr)

        #____FEATURE____Распределение длин слов
        word_len_distr = word_length_distribution(tokenized)
        res.append(word_len_distr)

        # Нормализация
        normalized = [word_norm(word) for word in tokenized]

        #____FEATURE____Разнообразие речи: Количество уникальных слов ко всему объ
ему текста
        lexical_divers = lexical_diversity(normalized)
        res.append([lexical_divers])

        #____FEATURE____Водность речи
        nonstop_to_volume = non_stopwords(normalized)
        res.append([nonstop_to_volume])

        return numpy_concatenate(res)
```

Bigrams

In [40]:

```
ALPHABET_RU = 'абвгдеёжзийклмнопрстуфхцчшщъыьэюя'
```

In [41]:

```
russian_bigrams = []

for i in ALPHABET_RU:
    for j in ALPHABET_RU:
        russian_bigrams.append(i+j)
```

In [42]:

```
print(russian_bigrams[:10])
```

```
print('Количество биграм: ', len(russian_bigrams))
```

```
['аа', 'аб', 'ав', 'аг', 'ад', 'ае', 'аё', 'аж', 'аз', 'аи']
Количество биграм:  1089
```

In [43]:

```
def count_bigrams(text):


    text_modified = pure_text(str(text).lower())
    bigrams = Counter(x+y for x, y in zip(*[text_modified[i:] for i in range(2
)]) if x.isalpha() and y.isalpha())
    return bigrams
```

In [44]:

```
count_bigrams('СТАТИСТИЧЕСКИЙ АНАЛИЗ ТЕКСТА')
```

Out[44]:

```
Counter({'ал': 1,
         'ан': 1,
         'ат': 1,
         'ек': 1,
         'ес': 1,
         'из': 1,
         'ий': 1,
         'ис': 1,
         'ич': 1,
         'ки': 1,
         'кс': 1,
         'ли': 1,
         'на': 1,
         'ск': 1,
         'ст': 3,
         'та': 2,
         'те': 1,
         'ти': 2,
         'че': 1})
```

Construct TRAIN/TEST

L = 20000

In [45]:

```
X_train, y_train, X_test, y_test = build_train_test_sets(folders, 20000, 1000, 300)
```

In [212]:

```
print('Количество строк в трейне: ', len(X_train))
print('Длина строки: ', len(X_train[0]))
print('Количество строк в test: ', len(X_test))
```

```
Количество строк в трейне:  12000
Длина строки:  20000
Количество строк в test:  3600
```

Apply features to train/test

In [235]:

```
start_time = time.time()
text_features(X_train[0])
print("--- %s seconds ---" % (time.time() - start_time))
```

```
--- 6.630380392074585 seconds ---
```

In [291]:

```
start_time = time.time()
X_train_features = []
i = 0
for text in X_train:
    X_train_features.append(text_features(text))
    i += 1
    if i % 1000 == 0:
        print('Выполнено ', i, ' итераций.')
print("--- %s seconds ---" % (time.time() - start_time))
```

```
Выполнено  1000  итераций.
Выполнено  2000  итераций.
Выполнено  3000  итераций.
Выполнено  4000  итераций.
Выполнено  5000  итераций.
Выполнено  6000  итераций.
```

```
Выполнено  7000  итераций.
Выполнено  8000  итераций.
Выполнено  9000  итераций.
Выполнено  10000  итераций.
Выполнено  11000  итераций.
Выполнено  12000  итераций.
--- 51494.5682182312 seconds ---
```

Common function

In [46]:

```
def apply_features(data, print_iterations = 1000):
    start_time = time.time()
    data_features = []
    i = 0
    for text in data:
        data_features.append(text_features(text))
        i += 1
        if i % print_iterations == 0:
            print('Выполнено ', i, ' итераций.')
    print("--- %s seconds ---" % (time.time() - start_time))
    return data_features
```

In [ ]:

```
X_train_20000, y_train_20000, X_test_20000, y_test_20000 = build_train_test_sets(folders, 20000, 1000, 300)
X_train_10000, y_train_10000, X_test_10000, y_test_10000 = build_train_test_sets(folders, 10000, 1000, 300)
X_train_5000, y_train_5000, X_test_5000, y_test_5000 = build_train_test_sets(folders, 5000, 1000, 300)
X_train_1000, y_train_1000, X_test_1000, y_test_1000 = build_train_test_sets(folders, 1000, 1000, 300)
```

In [566]:

```
X_test_features = apply_features(X_test, 400)
```

```
Выполнено  400  итераций.
Выполнено  800  итераций.
Выполнено  1200  итераций.
```

```
Выполнено  1600  итераций.
Выполнено  2000  итераций.
Выполнено  2400  итераций.
Выполнено  2800  итераций.
Выполнено  3200  итераций.
Выполнено  3600  итераций.
--- 22765.061748743057 seconds ---
```

In [513]:

```
X_test_features_n500 = apply_features(X_test[:500], 100)
```

```
--- 2678.3511624336243 seconds ---
```

In [516]:

```
save_to_txt('X_test_features_20000_n500.txt' , X_test_features_n500)
```

In [567]:

```
save_to_txt('X_test_features_20000.txt', X_test_features)
```

DOWNLOAD FEATURES FROM TEXT FILE

In [47]:

```
TXT_FILE_PATH = 'C:/Users/Yauheniya/Documents/Diploma'
```

In [48]:

```
import pickle
```

In [50]:

```
def read_features_file(path):
    list_of_lists = []
    with open(path) as f:
        for line in f:
            line = line.replace('[', '')
            line = line.replace(']', '')
            inner_list = [float(elt.strip()) for elt in line.split(',')]
            list_of_lists.append(inner_list)
    return list_of_lists
```

In [51]:

```
X_test_features_1000 = read_features_file('C:/Users/Yauheniya/Documents/Diplom
a\\X_test_features_1000.txt')
```

```
X_train_features_1000 = read_features_file('C:/Users/Yauheniya/Documents/Diplo
ma\\X_train_features_1000.txt')


X_test_features_5000 = read_features_file('C:/Users/Yauheniya/Documents/Diplom
a\\X_test_features_5000.txt')

X_train_features_5000 = read_features_file('C:/Users/Yauheniya/Documents/Diplo
ma\\X_train_features_5000.txt')


X_test_features_10000 = read_features_file('C:/Users/Yauheniya/Documents/Diplo
ma\\X_test_features_10000.txt')

X_train_features_10000 = read_features_file('C:/Users/Yauheniya/Documents/Dipl
oma\\X_train_features_10000.txt')


X_test_features_20000 = read_features_file('C:/Users/Yauheniya/Documents/Diplo
ma\\X_test_features_20000.txt')

X_train_features_20000 = read_features_file('C:/Users/Yauheniya/Documents/Dipl
oma\\X_train_features_20000.txt')
```

In [56]:

```
def read_bigrams_file(path):
    with open(path, 'r') as myfile:
        data=myfile.read().replace('\n', '')


    data = data.replace(']','')
    data = data.split('[')
    list_lines = []
    data = [i for i in data if i]
    for line in data:
        inner_list = [float(elt.strip()) for elt in line.split(' ') if elt]
        list_lines.append(inner_list)
    return list_lines
```

In [57]:

```
# X_test_bigrams_1000 = read_features_file('C:/Users/Yauheniya/Documents/Diplo
ma\\X_test_bigrams_1000.txt')

# X_train_bigrams_1000 = read_features_file('C:/Users/Yauheniya/Documents/Dipl
oma\\X_train_bigrams_1000.txt')
```

```
    # X_test_bigrams_5000 = read_features_file('C:/Users/Yauheniya/Documents/Diplo
ma\\X_test_bigrams_5000.txt')

    # X_train_bigrams_5000 = read_features_file('C:/Users/Yauheniya/Documents/Dipl
oma\\X_train_bigrams_5000.txt')


    # X_test_bigrams_10000 = read_features_file('C:/Users/Yauheniya/Documents/Dipl
oma\\X_test_bigrams_10000.txt')

    # X_train_bigrams_10000 = read_features_file('C:/Users/Yauheniya/Documents/Dip
loma\\X_train_bigrams_10000.txt')


    X_test_bigrams_20000 = read_bigrams_file('C:/Users/Yauheniya/Documents/Diploma
\\X_test_bigrams_20000.txt')

    X_train_bigrams_20000 = read_bigrams_file('C:/Users/Yauheniya/Documents/Diplom
a\\X_train_bigrams_20000.txt')
```

L = 10000

In [58]:

```
    X_train_10000, y_train_10000, X_test_10000, y_test_10000 = build_train_test_se
ts(folders, 10000, 1000, 300)
```

In [615]:

```
    X_train_10000, y_train_10000, X_test_10000, y_test_10000 = build_train_test_se
ts(folders, 10000, 1000, 300)

    print('Apply features to train: ')

    X_train_10000_features = apply_features(X_train_10000, 1000)

    print('Apply features to test: ')

    X_test_10000_features = apply_features(X_test_10000, 300)

    Apply features to train:

    Выполнено  1000  итераций.

    Выполнено  2000  итераций.

    Выполнено  3000  итераций.

    Выполнено  4000  итераций.

    Выполнено  5000  итераций.

    Выполнено  6000  итераций.

    Выполнено  7000  итераций.

    Выполнено  8000  итераций.
```

```
Выполнено  9000  итераций.
Выполнено  10000  итераций.
Выполнено  11000  итераций.
Выполнено  12000  итераций.
--- 71220.50283145905 seconds ---
Apply features to test:
Выполнено  300  итераций.
Выполнено  600  итераций.
Выполнено  900  итераций.
Выполнено  1200  итераций.
Выполнено  1500  итераций.
Выполнено  1800  итераций.
Выполнено  2100  итераций.
Выполнено  2400  итераций.
Выполнено  2700  итераций.
Выполнено  3000  итераций.
Выполнено  3300  итераций.
Выполнено  3600  итераций.
--- 12608.028089284897 seconds ---
```

In [616]:

```
save_to_txt('X_test_features_10000.txt', X_test_10000_features)
save_to_txt('X_train_features_10000.txt', X_train_10000_features)
```

L = 5000

In [59]:

```
X_train_5000, y_train_5000, X_test_5000, y_test_5000 = build_train_test_sets(folders, 5000, 1000, 300)
```

In [617]:

```
X_train_5000, y_train_5000, X_test_5000, y_test_5000 = build_train_test_sets(folders, 5000, 1000, 300)
X_train_5000_features = apply_features(X_train_5000, 1000)
X_test_5000_features = apply_features(X_test_5000, 300)
```

```
Выполнено  1000  итераций.
Выполнено  2000  итераций.
```

```
Выполнено  3000  итераций.
Выполнено  4000  итераций.
Выполнено  5000  итераций.
Выполнено  6000  итераций.
Выполнено  7000  итераций.
Выполнено  8000  итераций.
Выполнено  9000  итераций.
Выполнено  10000  итераций.
Выполнено  11000  итераций.
Выполнено  12000  итераций.
--- 16093.26831483841 seconds ---
Выполнено  300  итераций.
Выполнено  600  итераций.
Выполнено  900  итераций.
Выполнено  1200  итераций.
Выполнено  1500  итераций.
Выполнено  1800  итераций.
Выполнено  2100  итераций.
Выполнено  2400  итераций.
Выполнено  2700  итераций.
Выполнено  3000  итераций.
Выполнено  3300  итераций.
Выполнено  3600  итераций.
--- 4962.587314367294 seconds ---
```

In [620]:

```
save_to_txt('X_test_features_5000.txt', X_test_5000_features)
save_to_txt('X_train_features_5000.txt', X_train_5000_features)
```

L = 1000

In [60]:

```
X_train_1000, y_train_1000, X_test_1000, y_test_1000 = build_train_test_sets(folders, 1000, 1000, 300)
```

In [619]:

```
X_train_1000, y_train_1000, X_test_1000, y_test_1000 = build_train_test_sets(f
olders, 1000, 1000, 300)
X_train_1000_features = apply_features(X_train_1000, 1000)
X_test_1000_features = apply_features(X_test_1000, 300)
```

```
Выполнено  1000  итераций.
Выполнено  2000  итераций.
Выполнено  3000  итераций.
Выполнено  4000  итераций.
Выполнено  5000  итераций.
Выполнено  6000  итераций.
Выполнено  7000  итераций.
Выполнено  8000  итераций.
Выполнено  9000  итераций.
Выполнено  10000  итераций.
Выполнено  11000  итераций.
Выполнено  12000  итераций.
--- 3074.2685482501984 seconds ---
Выполнено  300  итераций.
Выполнено  600  итераций.
Выполнено  900  итераций.
Выполнено  1200  итераций.
Выполнено  1500  итераций.
Выполнено  1800  итераций.
Выполнено  2100  итераций.
Выполнено  2400  итераций.
Выполнено  2700  итераций.
Выполнено  3000  итераций.
Выполнено  3300  итераций.
Выполнено  3600  итераций.
--- 893.1362199783325 seconds ---
```

In [621]:

```
save_to_txt('X_test_features_1000.txt', X_test_1000_features)
save_to_txt('X_train_features_1000.txt', X_train_1000_features)
```

Add L 500, 200, 100

In [53]:

```
X_test_features_500 = read_features_file('C:/Users/Yauheniya/Documents/Diploma\\X_test_features_500.txt')

X_train_features_200 = read_features_file('C:/Users/Yauheniya/Documents/Diploma\\X_train_features_200.txt')

X_train_features_100 = read_features_file('C:/Users/Yauheniya/Documents/Diploma\\X_train_features_100.txt')


X_train_500, y_train_500, X_test_500, y_test_500 = build_train_test_sets(folders, 500, 1000, 300)

X_train_200, y_train_200, X_test_200, y_test_200 = build_train_test_sets(folders, 200, 1000, 300)

X_train_100, y_train_100, X_test_100, y_test_100 = build_train_test_sets(folders, 100, 1000, 300)


print('Apply features to train_500: ')

X_train_features_500 = apply_features(X_train_500, 1000)

save_to_txt('X_train_features_500.txt', X_train_features_500)


print('Apply features to test_200: ')

X_test_features_200 = apply_features(X_test_200, 300)

save_to_txt('X_test_features_200.txt', X_test_features_200)


print('Apply features to test_100: ')

X_test_features_100 = apply_features(X_test_100, 300)

save_to_txt('X_test_features_100.txt', X_test_features_100)
Apply features to train_500:
Выполнено  1000  итераций.
Выполнено  2000  итераций.
Выполнено  3000  итераций.
Выполнено  4000  итераций.
Выполнено  5000  итераций.
Выполнено  6000  итераций.
```

```
Выполнено  7000  итераций.
Выполнено  8000  итераций.
Выполнено  9000  итераций.
Выполнено  10000  итераций.
Выполнено  11000  итераций.
Выполнено  12000  итераций.
--- 1906.083199262619 seconds ---
Apply features to test_200:
Выполнено  300  итераций.
Выполнено  600  итераций.
Выполнено  900  итераций.
Выполнено  1200  итераций.
Выполнено  1500  итераций.
Выполнено  1800  итераций.
Выполнено  2100  итераций.
Выполнено  2400  итераций.
Выполнено  2700  итераций.
Выполнено  3000  итераций.
Выполнено  3300  итераций.
Выполнено  3600  итераций.
--- 238.18416953086853 seconds ---
Apply features to test_100:
Выполнено  300  итераций.
Выполнено  600  итераций.
Выполнено  900  итераций.
Выполнено  1200  итераций.
Выполнено  1500  итераций.
Выполнено  1800  итераций.
Выполнено  2100  итераций.
Выполнено  2400  итераций.
Выполнено  2700  итераций.
Выполнено  3000  итераций.
Выполнено  3300  итераций.
```

```
Выполнено  3600  итераций.
--- 126.03307580947876 seconds ---
```

In [193]:

```
X_train_500, y_train_500, X_test_500, y_test_500 = build_train_test_sets(folde
rs, 500, 1000, 300)
print('Apply features to train_500: ')
X_train_500_features = apply_features(X_train_500, 1000)
print('Apply features to test_500: ')
X_test_500_features = apply_features(X_test_500, 300)
save_to_txt('X_test_features_500.txt', X_test_500_features)

X_train_200, y_train_200, X_test_200, y_test_200 = build_train_test_sets(folde
rs, 200, 1000, 300)
print('Apply features to train_200: ')
X_train_200_features = apply_features(X_train_200, 1000)
print('Apply features to test_200: ')
X_test_200_features = apply_features(X_test_200, 300)
save_to_txt('X_train_features_200.txt', X_train_200_features)

X_train_100, y_train_100, X_test_100, y_test_100 = build_train_test_sets(folde
rs, 100, 1000, 300)
print('Apply features to train_100: ')
X_train_100_features = apply_features(X_train_100, 1000)
print('Apply features to test_100: ')
X_test_100_features = apply_features(X_test_100, 300)
save_to_txt('X_train_features_100.txt', X_train_100_features)
```

```
Apply features to train_500:
Выполнено  1000  итераций.
Выполнено  2000  итераций.
Выполнено  3000  итераций.
Выполнено  4000  итераций.
Выполнено  5000  итераций.
Выполнено  6000  итераций.
Выполнено  7000  итераций.
```

```
Выполнено  8000  итераций.
Выполнено  9000  итераций.
Выполнено  10000  итераций.
Выполнено  11000  итераций.
Выполнено  12000  итераций.
--- 1637.513708114624 seconds ---
Apply features to test_500:
Выполнено  300  итераций.
Выполнено  600  итераций.
Выполнено  900  итераций.
Выполнено  1200  итераций.
Выполнено  1500  итераций.
Выполнено  1800  итераций.
Выполнено  2100  итераций.
Выполнено  2400  итераций.
Выполнено  2700  итераций.
Выполнено  3000  итераций.
Выполнено  3300  итераций.
Выполнено  3600  итераций.
--- 502.9348421096802 seconds ---
Apply features to train_200:
Выполнено  1000  итераций.
Выполнено  2000  итераций.
Выполнено  3000  итераций.
Выполнено  4000  итераций.
Выполнено  5000  итераций.
Выполнено  6000  итераций.
Выполнено  7000  итераций.
Выполнено  8000  итераций.
Выполнено  9000  итераций.
Выполнено  10000  итераций.
Выполнено  11000  итераций.
Выполнено  12000  итераций.
```

```
--- 679.055750131607 seconds ---

Apply features to test_200:

Выполнено  300  итераций.

Выполнено  600  итераций.

Выполнено  900  итераций.

Выполнено  1200  итераций.

Выполнено  1500  итераций.

Выполнено  1800  итераций.

Выполнено  2100  итераций.

Выполнено  2400  итераций.

Выполнено  2700  итераций.

Выполнено  3000  итераций.

Выполнено  3300  итераций.

Выполнено  3600  итераций.

--- 19852.91144132614 seconds ---

Apply features to train_100:

Выполнено  1000  итераций.

Выполнено  2000  итераций.
Выполнено  3000  итераций.
Выполнено  4000  итераций.
Выполнено  5000  итераций.
Выполнено  6000  итераций.
Выполнено  7000  итераций.
Выполнено  8000  итераций.
Выполнено  9000  итераций.
Выполнено  10000  итераций.
Выполнено  11000  итераций.
Выполнено  12000  итераций.
--- 375.43145871162415 seconds ---
Apply features to test_100:
Выполнено  300  итераций.
Выполнено  600  итераций.
Выполнено  900  итераций.
Выполнено  1200  итераций.
Выполнено  1500  итераций.
Выполнено  1800  итераций.
Выполнено  2100  итераций.
Выполнено  2400  итераций.
Выполнено  2700  итераций.
Выполнено  3000  итераций.
Выполнено  3300  итераций.
Выполнено  3600  итераций.
--- 110.28601455688477 seconds ---
```

Count bigrams for train/test

In [253]:

```
start_time = time.time()
text_bigrams = []
for text in X_train[:1]:
    text_bigrams.append(count_bigrams(text))
print("--- %s seconds ---" % (time.time() - start_time))
```

```
--- 0.8690032958984375 seconds ---
```

In [61]:

```
def count_bigrams_distr(data, label, max_freq):
    total_bigrams = Counter()
    for text in data:
        total_bigrams.update(count_bigrams(text))
    most_common_bigrams_freq = total_bigrams.most_common(200)
    most_common_bigrams = [key for key, val in most_common_bigrams_freq]

    all_freqs = []
    for text in data:
        cur_frequencies = []
        cur_bigrams = count_bigrams(text)
        for bigram in most_common_bigrams:
            cur_frequencies.append(cur_bigrams[bigram])
        all_freqs.append(cur_frequencies)

    if label == 'train':
        largest_freq = np.max(all_freqs)
        return largest_freq, all_freqs / largest_freq
    else:
        return max_freq, all_freqs / max_freq
```

In [277]:

```
start_time = time.time()
bigrams_largest_freq, X_train_bigrams = count_bigrams_distr(X_train, 'train',
0)
```

```
print("--- %s seconds ---" % (time.time() - start_time))
```

```
--- 608.1697566509247 seconds ---
```

In [280]:

```
start_time = time.time()
bigrams_largest_freq, X_test_bigrams = count_bigrams_distr(X_test, 'test', bigrams_largest_freq)
print("--- %s seconds ---" % (time.time() - start_time))
```

```
--- 178.92333936691284 seconds ---
```

In [294]:

```
save_to_txt('X_train_bigrams_20000.txt', X_train_bigrams)
```

In [295]:

```
start_time = time.time()
save_to_txt('X_test_bigrams_20000.txt', X_test_bigrams)
print("--- %s seconds ---" % (time.time() - start_time))
```

```
--- 25.53329849243164 seconds ---
```

In [296]:

```
start_time = time.time()
save_to_txt('X_train_features_20000.txt', X_train_features)
print("--- %s seconds ---" % (time.time() - start_time))
```

```
--- 0.7941687107086182 seconds ---
```

Bigrams with scaling

In [62]:

```
def count_total_bigrams(X, list_bigrams = russian_bigrams):
    total_bigrams = dict.fromkeys(list_bigrams, 0)

    for text in X:
        total_bigrams.update(count_bigrams(text))

    total_bigrams1 = dict.fromkeys(list_bigrams, 0)
    for bigram in list_bigrams:
        total_bigrams1[bigram] = total_bigrams[bigram]

    return total_bigrams1
```

In [187]:

```
count_total_bigrams(['пипиор', 'неуаан'], ['аб', 'бв', 'пи', 'не'])
```

Out[187]:

```
{'аб': 0, 'бв': 0, 'не': 1, 'пи': 2}
```

In [63]:

```
def separate_bigrams(data, list_bigrams):
    """
    Return frequencies only for bigrams mentioned in list_bigrams
    """
    train_bigrams_freq = []
    for text in data:
        cur_frequencies = []
        cur_bigrams = count_bigrams(text)
        for bigram in list_bigrams:
            cur_frequencies.append(cur_bigrams[bigram])
        train_bigrams_freq.append(cur_frequencies)
    return train_bigrams_freq
```

In [64]:

```
def dataset_bigrams(X_train, X_test):
    """
    Count bigrams frequencies for train and test
    """
    train_bigrams = count_total_bigrams(X_train)
    most_common_bigrams_freq = Counter(train_bigrams).most_common(200)
    train_list_bigrams = [key for key, val in most_common_bigrams_freq]

    # train_bigrams_freq = [val for key, val in most_common_bigrams_freq]
    train_bigrams_freq = separate_bigrams(X_train, train_list_bigrams)
    test_bigrams_freq = separate_bigrams(X_test, train_list_bigrams)

#     train_list_bigrams = [key for key, val in most_common_bigrams_freq]
#     test_bigrams = count_total_bigrams(X_test, train_list_bigrams)
#     test_bigrams_freq = [val for key, val in list(test_bigrams.items())]
```

```
    return train_bigrams_freq, test_bigrams_freq
```

In [58]:

```
X_train_bigrams_freq_20000, X_test_bigrams_freq_20000 = dataset_bigrams(X_train_20000, X_test_20000)

X_train_bigrams_freq_10000, X_test_bigrams_freq_10000 = dataset_bigrams(X_train_10000, X_test_10000)

X_train_bigrams_freq_5000, X_test_bigrams_freq_5000 = dataset_bigrams(X_train_5000, X_test_5000)

X_train_bigrams_freq_1000, X_test_bigrams_freq_1000 = dataset_bigrams(X_train_1000, X_test_1000)
```

In [59]:

```
X_train_bigrams_freq_500, X_test_bigrams_freq_500 = dataset_bigrams(X_train_500, X_test_500)

X_train_bigrams_freq_200, X_test_bigrams_freq_200 = dataset_bigrams(X_train_200, X_test_200)

X_train_bigrams_freq_100, X_test_bigrams_freq_100 = dataset_bigrams(X_train_100, X_test_100)
```

Нормализация

An alternative standardization is scaling features to lie between a given minimum and maximum value, often between zero and one, or so that the maximum absolute value of each feature is scaled to unit size. This can be achieved using MinMaxScaler or MaxAbsScaler, respectively.

The motivation to use this scaling include robustness to very small standard deviations of features and preserving zero entries in sparse data.

Vectorized features normalize

In [ ]:

```
# min_max_scaler = preprocessing.MinMaxScaler()

# X_train_minmax = min_max_scaler.fit_transform(X_train)

# X_test_minmax = min_max_scaler.transform(X_test)
```

In [596]:

```
min_max_scaler_20000 = preprocessing.MinMaxScaler()

X_train_features_norm = min_max_scaler_20000.fit_transform(X_train_features)

X_test_features_norm = min_max_scaler_20000.transform(X_test_features)
```

Bigrams normalize

In [199]:

```
def bigrams_normalize(X_train_bigrams_freq, X_test_bigrams_freq):
    """
    Нормализация биграммных частот
    """
    max_value = np.max(X_train_bigrams_freq)
    return X_train_bigrams_freq / max_value, X_test_bigrams_freq / max_value
```

In [486]:

```
X_train_bigrams_freq, X_test_bigrams_freq = bigrams_normalize(X_train_bigrams_freq, X_test_bigrams_freq)
```

Dataset with L = 20000

In [524]:

```
print('Количество строк в X_train_features: ', len(X_train_features))
print('Количество строк в X_test_features_n500: ', len(X_test_features_n500))
```

```
Количество строк в X_train_features:  12000
Количество строк в X_test_features_n500:  500
```

In [523]:

```
print('Количество строк в X_train_bigrams_freq: ', len(X_train_bigrams_freq))
print('Количество строк в X_test_bigrams_freq: ', len(X_test_bigrams_freq))
```

```
Количество строк в X_train_bigrams_freq:  12000
Количество строк в X_test_bigrams_freq:  3600
```

In [200]:

```
def concat_features_bigrams(features, bigrams):
    """
    Concatenates two matrices (multi-dim-arrays): with features and bigrams frequencies
    """
    concat_features = []
    for i, j in zip(features, bigrams):
        concat_features.append(list(i) + list(j))
    return concat_features
```

In [568]:

```
X_train_20000_concat = concat_features_bigrams(X_train_features, X_train_bigrams_freq)

X_test_20000_concat = concat_features_bigrams(X_test_features, X_test_bigrams_freq)
```

In [603]:

```
X_train_20000_norm = concat_features_bigrams(X_train_features_norm, X_train_bigrams_freq)

X_test_20000_norm = concat_features_bigrams(X_test_features_norm, X_test_bigrams_freq)
```

Train

In [1]:

```
from sklearn.svm import SVC

from sklearn import metrics

from sklearn.linear_model import LogisticRegression

from sklearn.naive_bayes import GaussianNB

from sklearn.ensemble import GradientBoostingClassifier

from sklearn.tree import DecisionTreeClassifier

from sklearn.metrics import accuracy_score, roc_auc_score

from sklearn.model_selection import validation_curve

from sklearn.neighbors import KNeighborsClassifier
```

In [2]:

```
from sklearn.model_selection import RandomizedSearchCV

from sklearn.model_selection import GridSearchCV
```

In [ ]:

```
y_train = y_train_20000

y_test = y_test_20000
```

In [569]:

```
model = SVC(kernel='linear')

model.fit(X_train_20000_concat, y_train)

print(model)


print(metrics.classification_report(y_test, model.predict(X_test_20000_concat)))

# print(metrics.confusion_matrix(expected, predicted))
```

```
SVC(C=1.0, cache_size=200, class_weight=None, coef0=0.0,
  decision_function_shape='ovr', degree=3, gamma='auto', kernel='linear',
  max_iter=-1, probability=False, random_state=None, shrinking=True,
  tol=0.001, verbose=False)
```

	precision	recall	f1-score	support
1	0.52	0.48	0.50	300
2	0.26	0.24	0.25	300
3	0.66	0.96	0.78	300
4	1.00	0.97	0.98	300
5	1.00	0.17	0.29	300
6	0.72	0.94	0.81	300
7	0.96	0.86	0.91	300
8	0.58	0.89	0.70	300
9	0.79	0.73	0.76	300
10	0.79	0.91	0.85	300
11	0.58	0.91	0.70	300
12	0.07	0.01	0.01	300
avg / total	0.66	0.67	0.63	3600

In [570]:

```
print(metrics.confusion_matrix(y_test, model.predict(X_test_20000_concat)))
```

```
[[145   0  27   0   0   6   0   0  39  26  57   0]
 [ 23  73   6   0   0   0  11  72   0   0 113   2]
 [  0   0 288   0   0   0   0   0   8   2   1   1]
 [  0   0   0 291   0   0   0   0   0   9   0   0]
 [  0 209   2   0  50  23   0   6   0   4   6   0]
 [  0   0   0   1   0 283   0  16   0   0   0   0]
 [  0   0   0   0   0   0 258  17   0   0   0  25]
 [  1   1   0   0   0  24   0 268   0   0   6   0]
 [ 10   0  68   0   0   0   0   0 220   0   2   0]
 [  0   0  12   0   0   3   0   3   3 274   5   0]
```

```
 [  0   0  16   0   0   4   0   0   8   0 272   0]
 [101   0  20   0   0  52   0  81   0  33  11   2]]
```

In [585]:

```
svc = SVC(kernel='rbf').fit(X_train_20000_concat, y_train)
svc.score(X_test_20000_concat, y_test)
```

Out[585]:

```
0.08333333333333333
```

In [586]:

```
svc = SVC(kernel='linear').fit(X_train_20000_concat, y_train)
svc.score(X_test_20000_concat, y_test)
```

Out[586]:

```
0.6733333333333333
```

In [584]:

```
svc = SVC(kernel='sigmoid').fit(X_train_20000_concat, y_train)
svc.score(X_test_20000_concat, y_test)
```

Out[584]:

```
0.08333333333333333
```

In [ ]:

```
# X_train_features, X_train_bigrams_freq
```

In [587]:

```
svc = SVC(kernel='linear').fit(X_train_features, y_train)
svc.score(X_test_features, y_test)
```

Out[587]:

```
0.6741666666666667
```

In [588]:

```
svc = SVC(kernel='linear').fit(X_train_bigrams_freq, y_train)
svc.score(X_test_bigrams_freq, y_test)
```

Out[588]:

```
0.6797222222222222
```

In [598]:

```
svc = SVC(kernel='linear').fit(X_train_features_norm, y_train)
svc.score(X_test_features_norm, y_test)
```

Out[598]:

```
0.7066666666666667
```

In [604]:

```
svc = SVC(kernel='linear').fit(X_train_20000_norm, y_train)
svc.score(X_test_20000_norm, y_test)
```

Out[604]:

```
0.7436111111111111
```

LOGRegression

In [211]:

```
from sklearn.linear_model import LogisticRegression
```

In [212]:

```
model_log = LogisticRegression()
model_log.fit(X_train_20000_norm, y_train)
# predicted = model_log.predict(X_test_20000_norm)
model_log.score(X_test_20000_norm, y_test)
# print(metrics.classification_report(y_test, predicted))
```

Naive Bayes

In [210]:

```
from sklearn.naive_bayes import GaussianNB
```

In [612]:

```
model_gauss = GaussianNB()
model_gauss.fit(X_train_20000_norm, y_train)
model_gauss.score(X_test_20000_norm, y_test)
```

Out[612]:

```
0.5966666666666667
```

Not normalized data

In [647]:

```
svc = SVC(kernel='linear').fit(X_train_20000_concat, y_train)
svc.score(X_test_20000_concat, y_test)
```

Out[647]:

```
0.6733333333333333
```

Common function

In [208]:

```
    def count_score(X_train_features, X_test_features, X_train_bigrams_freq, X_tes
t_bigrams_freq):
        # объединяем признаки и биграммы
        X_train_concat = concat_features_bigrams(X_train_features, X_train_bigrams
_freq)
        X_test_concat = concat_features_bigrams(X_test_features, X_test_bigrams_fr
eq)

        # скалируем
        scaler = preprocessing.StandardScaler()
        X_train_standard = scaler.fit_transform(X_train_concat)
        X_test_standard = scaler.fit_transform(X_test_concat)

        # svc = SVC(kernel='linear').fit(X_train_standard, y_train)
        svc = LogisticRegression().fit(X_train_standard, y_train)
        return svc.score(X_test_standard, y_test)
```

In [704]:

```
from sklearn.ensemble import GradientBoostingClassifier as gbc
```

In [214]:

```
# log
count_score(X_train_features_20000, X_test_features_20000, X_train_bigrams_fre
q, X_test_bigrams_freq)
```

Out[214]:

```
0.7722222222222223
```

In [674]:

```
# логистическая без стандартизация
count_score(X_train_features, X_test_features, X_train_bigrams_freq, X_test_bi
grams_freq)
```

Out[674]:

```
0.6411111111111111
```

In [665]:

```
# без нормализации
# count_score(X_train_features, X_test_features, X_train_bigrams_freq, X_test_
bigrams_freq)
```

Out[665]:

```
0.6733333333333333
```

In [701]:

```
count_score(X_train_features, X_test_features, X_train_bigrams_freq, X_test_bi
grams_freq)
```

Out[701]:

```
0.6672222222222223
```

In [667]:

```
count_score(X_train_features, X_test_features, X_train_bigrams_freq, X_test_bi
grams_freq)
```

Out[667]:

```
0.7475
```

In [654]:

```
count_score(X_train_10000_features, X_test_10000_features, X_train_bigrams_fre
q_10000, X_test_bigrams_freq_10000)
```

Out[654]:

```
0.6966666666666667
```

In [640]:

```
count_score(X_train_features, X_test_features, X_train_bigrams_freq, X_test_bi
grams_freq)
```

Out[640]:

```
0.7452777777777778
```

In [641]:

```
# c MinMaxScaler 0.707
count_score(X_train_10000_features, X_test_10000_features, X_train_bigrams_fre
q_10000, X_test_bigrams_freq_10000)
```

Out[641]:

```
0.6994444444444444
```

In [642]:

```
# c MinMaxScaler 0.6325
count_score(X_train_5000_features, X_test_5000_features, X_train_bigrams_freq_
5000, X_test_bigrams_freq_5000)
```

Out[642]:

```
0.6936111111111111
```

In [643]:

```
# c MinMaxScaler 0.47916
count_score(X_train_1000_features, X_test_1000_features, X_train_bigrams_freq_1000, X_test_bigrams_freq_1000)
```

Out[643]:

```
0.5077777777777778
```

Confusion Matrix

In [81]:

```
import seaborn as sns
```

In [82]:

```
# cm = metrics.confusion_matrix(y_test, predictions)
def draw_comfusion_matrix_sns(cm):
    plt.figure(figsize=(9,9))
    sns.heatmap(cm, annot=True, fmt=".3f", linewidths=.5, square = True, cmap = 'Blues_r');
    plt.ylabel('Actual label');
    plt.xlabel('Predicted label');
    all_sample_title = 'Accuracy Score: {0}'.format(score)
    plt.title(all_sample_title, size = 15);
```

In [83]:

```
def draw_cm_matplot(cm):
    plt.figure(figsize=(10,10))
    plt.imshow(cm, interpolation='nearest', cmap='Pastel1')
    plt.title('Confusion matrix', size = 15)
    plt.colorbar()
    tick_marks = np.arange(len(folders))
    plt.xticks(tick_marks, folders, rotation=45, size = len(folders))
    plt.yticks(tick_marks, folders, size = len(folders))
    plt.tight_layout()
    plt.ylabel('Actual label', size = 15)
    plt.xlabel('Predicted label', size = 15)
    width, height = cm.shape
```

```
    for x in range(width):
        for y in range(height):
            plt.annotate(str(cm[x][y]), xy=(y, x),
            horizontalalignment='center',
            verticalalignment='center')
```

In [185]:

```
lr = LogisticRegression().fit(X_train_scale_20000, y_train_20000)
cm_20000 = metrics.confusion_matrix(y_test, lr.predict(X_test_scale_20000))
```

In [186]:

```
lr.score(X_test_scale_20000, y_test_20000)
```

Out[186]:

```
0.8130555555555555
```

In [187]:

```
draw_cm_matplot(cm_20000)
```

In [190]:

```
lr = LogisticRegression().fit(X_train_scale_10000, y_train_10000)
print('Score = ', lr.score(X_test_scale_10000, y_test_10000))
cm_10000 = metrics.confusion_matrix(y_test, lr.predict(X_test_scale_10000))
draw_cm_matplot(cm_10000)
```

```
Score =  0.7408333333333333
```

In [191]:

```
lr = LogisticRegression().fit(X_train_scale_5000, y_train_5000)
print('Score = ', lr.score(X_test_scale_5000, y_test_5000))
cm_5000 = metrics.confusion_matrix(y_test, lr.predict(X_test_scale_5000))
draw_cm_matplot(cm_5000)
```

```
Score =  0.7141666666666666
```

In [192]:

```
lr = LogisticRegression().fit(X_train_scale_1000, y_train_1000)
print('Score = ', lr.score(X_test_scale_1000, y_test_1000))
cm_1000 = metrics.confusion_matrix(y_test, lr.predict(X_test_scale_1000))
```

```
draw_cm_matplot(cm_1000)
```

```
Score =  0.5352777777777777
```

In [207]:

```
lr = LogisticRegression().fit(X_train_scale_500, y_train_500)
print('Score = ', lr.score(X_test_scale_500, y_test_500))
cm_500 = metrics.confusion_matrix(y_test, lr.predict(X_test_scale_500))
draw_cm_matplot(cm_500)
```

```
Score =  0.46416666666666667
```

In [208]:

```
lr = LogisticRegression().fit(X_train_scale_200, y_train_200)
print('Score = ', lr.score(X_test_scale_200, y_test_200))
cm_200 = metrics.confusion_matrix(y_test, lr.predict(X_test_scale_200))
draw_cm_matplot(cm_200)
```

```
Score =  0.31222222222222223
```

In [209]:

```
lr = LogisticRegression().fit(X_train_scale_100, y_train_100)
print('Score = ', lr.score(X_test_scale_100, y_test_100))
cm_100 = metrics.confusion_matrix(y_test, lr.predict(X_test_scale_100))
draw_cm_matplot(cm_100)
```

```
Score =  0.25
```

Tune logistic regression

In [212]:

```
def tune_param_logistic(X_train, X_test, y_train, y_test):

    start_time = time.time()

    C_param_range = [0.0001, 0.001, 0.01, 0.1, 1, 10]

    acc_table = pd.DataFrame(columns = ['C_parameter','Accuracy'])
```

```
    acc_table['C_parameter'] = C_param_range


    j = 0
    for i in C_param_range:
        lr = LogisticRegression(C = i)
        lr.fit(X_train, y_train)
        # print ('Score = ', lr.score(X_test, y_test), ' . C = ', i)
        acc_table.iloc[j,1] = accuracy_score(y_test, lr.predict(X_test))
        j += 1


    print("--- %s seconds ---" % (time.time() - start_time))
    return acc_table
```

In [214]:

```
tune_param_logistic(X_train_scale_20000, X_test_scale_20000, y_train, y_test)
```

```
--- 135.60317873954773 seconds ---
```

Out[214]:

	C_parameter	Accuracy
0	0.0001	0.781389
1	0.0010	0.804167
2	0.0100	0.801667
3	0.1000	0.811111
4	1.0000	0.813056

	C_parameter	Accuracy
5	10.0000	0.805

In [215]:

```
tune_param_logistic(X_train_scale_10000, X_test_scale_10000, y_train, y_test)
```

```
--- 144.0762186050415 seconds ---
```

Out[215]:

	C_parameter	Accuracy
0	0.0001	0.726111
1	0.0010	0.750556
2	0.0100	0.745833
3	0.1000	0.748611
4	1.0000	0.740833
5	10.0000	0.733333

In [216]:

```
tune_param_logistic(X_train_scale_5000, X_test_scale_5000, y_train, y_test)
```

```
--- 151.885760307312 seconds ---
```

Out[216]:

	C_parameter	Accuracy
0	0.0001	0.705278
1	0.0010	0.725278
2	0.0100	0.716111
3	0.1000	0.710278
4	1.0000	0.714167
5	10.0000	0.695833

In [217]:

```
tune_param_logistic(X_train_scale_1000, X_test_scale_1000, y_train, y_test)
--- 188.87707138061523 seconds ---
```

Out[217]:

	C_parameter	Accuracy
0	0.0001	0.506389
1	0.0010	0.5

	C_parameter	Accuracy
		18889
2	0.0100	0.520556
3	0.1000	0.536389
4	1.0000	0.535278
5	10.0000	0.534444

In [218]:

```
tune_param_logistic(X_train_scale_500, X_test_scale_500, y_train, y_test)
```

```
--- 183.8874614238739 seconds ---
```

Out[218]:

	C_parameter	Accuracy
0	0.0001	0.441389
1	0.0010	0.458333
2	0.0100	0.4625

	C_parameter	Accuracy
3	0.1000	0.466944
4	1.0000	0.464167
5	10.0000	0.462778

In [219]:

```
tune_param_logistic(X_train_scale_200, X_test_scale_200, y_train, y_test)
```

```
--- 202.77686429023743 seconds ---
```

Out[219]:

	C_parameter	Accuracy
0	0.0001	0.256944
1	0.0010	0.276667
2	0.0100	0.292222
3	0.1000	0.306389
4	1.0000	0.312222

	C_parameter	Accuracy
5	10.0000	0.312222

k-NN

In [84]:

```
knc = KNeighborsClassifier(n_neighbors = 10, metric = "cityblock")
knc.fit(X_train_scale_20000, y_train)
knc.score(X_test_scale_20000, y_test)
```

Out[84]:

```
0.7686111111111111
```

In [89]:

```
knc = KNeighborsClassifier(n_neighbors = 12, metric = "euclidean")
knc.fit(X_train_scale_20000, y_train)
knc.score(X_test_scale_20000, y_test)
```

Out[89]:

```
0.7733333333333333
```

In [88]:

```
knc = KNeighborsClassifier(n_neighbors = 11, metric = "euclidean")
knc.fit(X_train_scale_20000, y_train)
knc.score(X_test_scale_20000, y_test)
```

Out[88]:

```
0.7744444444444445
```

In [172]:

```
params = {"n_neighbors": range(3,50,10)} # "metric": ["euclidean", "cityblock"
]
model = KNeighborsClassifier()
grid = GridSearchCV(model, params)
start = time.time()
grid.fit(X_train_scale_20000, y_train)
```

```
# evaluate the best grid searched model on the testing data
print("[INFO] grid search took {:.2f} seconds".format(time.time() - start))
acc = grid.score(X_test_scale_20000, y_test)
print("[INFO] grid search accuracy: {:.2f}%".format(acc * 100))
print("[INFO] grid search best parameters: {}".format(grid.best_params_))
```

```
[INFO] grid search took 2614.65 seconds
[INFO] grid search accuracy: 72.83%
[INFO] grid search best parameters: {'n_neighbors': 3}
```

In [90]:

```
knc = KNeighborsClassifier(n_neighbors = 11, metric = "euclidean")
knc.fit(X_train_scale_10000, y_train)
knc.score(X_test_scale_10000, y_test)
```

Out[90]:

```
0.7180555555555556
```

In [91]:

```
knc = KNeighborsClassifier(n_neighbors = 11, metric = "euclidean")
knc.fit(X_train_scale_5000, y_train_5000)
knc.score(X_test_scale_5000, y_test_5000)
```

Out[91]:

```
0.6647222222222222
```

In [92]:

```
knc = KNeighborsClassifier(n_neighbors = 11, metric = "euclidean")
knc.fit(X_train_scale_1000, y_train_1000)
knc.score(X_test_scale_1000, y_test_1000)
```

Out[92]:

```
0.3322222222222222
```

In [95]:

```
knc = KNeighborsClassifier(n_neighbors = 11, metric = "cityblock")
knc.fit(X_train_scale_1000, y_train_1000)
knc.score(X_test_scale_1000, y_test_1000)
```

Out[95]:

```
0.30444444444444446
```

In [96]:

```
knc = KNeighborsClassifier(n_neighbors = 11, metric = "euclidean")
knc.fit(X_train_scale_500, y_train_500)
knc.score(X_test_scale_500, y_test_500)
```

Out[96]:

```
0.2361111111111111
```

In [97]:

```
knc = KNeighborsClassifier(n_neighbors = 11, metric = "euclidean")
knc.fit(X_train_scale_200, y_train_200)
knc.score(X_test_scale_200, y_test_200)
```

Out[97]:

```
0.13277777777777777
```

SVM

In [104]:

```
svm = SVC(kernel='linear', C=0.0001, gamma = 0.0001)
svm.fit(X_train_scale_20000, y_train)
svm.score(X_test_scale_20000, y_test)
```

Out[104]:

```
0.7705555555555555
```

In [105]:

```
svm = SVC(kernel='linear', C=0.0001, gamma = 0.001)
svm.fit(X_train_scale_10000, y_train)
svm.score(X_test_scale_10000, y_test)
```

Out[105]:

```
0.7430555555555556
```

In [106]:

```
svm = SVC(kernel='linear', C=0.0001, gamma = 0.001)
svm.fit(X_train_scale_5000, y_train)
svm.score(X_test_scale_5000, y_test)
```

Out[106]:

```
0.7016666666666667
```

In [107]:

```
svm = SVC(kernel='linear', C=0.0001, gamma = 0.001)
```

```
svm.fit(X_train_scale_1000, y_train)
svm.score(X_test_scale_1000, y_test)
```

Out[107]:

```
0.5025
```

In [108]:

```
svm = SVC(kernel='linear')
svm.fit(X_train_scale_500, y_train)
svm.score(X_test_scale_500, y_test)
```

Out[108]:

```
0.46194444444444444
```

In [109]:

```
svm = SVC(kernel='linear', C=0.0001, gamma = 0.001)
svm.fit(X_train_scale_500, y_train)
svm.score(X_test_scale_500, y_test)
```

Out[109]:

```
0.4747222222222222
```

In [110]:

```
svm = SVC(kernel='linear', C=0.0001, gamma = 0.001)
svm.fit(X_train_scale_200, y_train_200)
svm.score(X_test_scale_200, y_test_200)
```

Out[110]:

```
0.2797222222222222
```

In [177]:

```
svm = SVC()
svm.fit(X_train_scale_20000, y_train)
svm.score(X_test_scale_20000, y_test)
```

Out[177]:

```
0.7052777777777778
```

In [185]:

```
def svc_param_selection(X, y, nfolds=3):
    Cs = [0.001, 0.01, 0.1, 1, 10]
    gammas = [0.001, 0.01, 0.1, 1]
    param_grid = {'C': Cs, 'gamma' : gammas}
```

```
    grid_search = GridSearchCV(SVC(kernel='linear'), param_grid, cv=nfolds)
    grid_search.fit(X, y)
    grid_search.best_params_

    return grid_search.best_params_
```

In [186]:

```
svc_param_selection(X_train_scale_20000, y_train, nfolds=3)
```

Out[186]:

```
{'C': 0.01, 'gamma': 0.001}
```

In [190]:

```
svm = SVC(kernel = 'linear', C=0.01, gamma = 0.001)
svm.fit(X_train_scale_20000, y_train)
svm.score(X_test_scale_20000, y_test)
```

Out[190]:

```
0.7494444444444445
```

Best params = 0.776944

In [193]:

```
svm = SVC(kernel = 'linear', C=0.001, gamma = 0.0001)
svm.fit(X_train_scale_20000, y_train)
svm.score(X_test_scale_20000, y_test)
```

Out[193]:

```
0.7769444444444444
```

In [205]:

```
svm = SVC(kernel = 'linear', C=100, gamma = 0.0001)
svm.fit(X_train_scale_20000, y_train)
svm.score(X_test_scale_20000, y_test)
```

Out[205]:

```
0.72
```

In [206]:

```
def svc_param_selection1(X, y, nfolds=3):
    Cs = [0.001, 0.01]
    gammas = [0.00001, 0.0001, 0.001, 0.01, 0.1, 1]
    kernels = ['linear']
```

```
    param_grid = {'C': Cs, 'gamma' : gammas}
    grid_search = GridSearchCV(SVC(), param_grid, cv=nfolds)
    grid_search.fit(X, y)
    grid_search.best_params_

    return grid_search.best_params_
```

In [207]:

```
svc_param_selection1(X_train_scale_20000, y_train, nfolds=3)
```

Out[207]:

```
{'C': 0.01, 'gamma': 0.01}
```

In [208]:

```
svm = SVC(kernel = 'linear', C=0.01, gamma = 0.01)
svm.fit(X_train_scale_20000, y_train)
svm.score(X_test_scale_20000, y_test)
```

Out[208]:

```
0.7494444444444445
```

In [ ]:

```
svc_param_selection(X_train_scale_10000, y_train, nfolds=3)
```

In [ ]:

```
svc_param_selection(X_train_scale_5000, y_train, nfolds=3)
```

In [ ]:

```
svc_param_selection(X_train_scale_1000, y_train, nfolds=3)
```

GradientBoostingClassifier

In [111]:

```
gbc = GradientBoostingClassifier()
gbc.fit(X_train_scale_20000, y_train)
gbc.score(X_test_scale_20000, y_test)
```

Out[111]:

```
0.6913888888888889
```

In [ ]:

```
param_test1 = {'n_estimators':range(20,81,10)}
gsearch1 = GridSearchCV(
    estimator = GradientBoostingClassifier(
```

```
            learning_rate=0.1, min_samples_split=500,min_samples_leaf=50,max_depth
=8,max_features='sqrt',subsample=0.8,random_state=10),
        param_grid = param_test1, scoring='roc_auc',n_jobs=4, cv=5)
    gsearch1.fit(X_train_scale_20000,y_train)
```

In [ ]:

```
    def gbc_param_selection(X, y, nfolds=5):
        estimators = range(20,81,10)

        param_grid = {'n_estimators': estimators}
        grid_search = GridSearchCV(GradientBoostingClassifier(learning_rate=0.1, m
in_samples_split=500,
                                                              min_samples_leaf=50,
max_depth=8,
                                                              max_features='sqrt',
subsample=0.8,random_state=10), param_grid, cv=nfolds)
        grid_search.fit(X, y)
        grid_search.best_params_

        return grid_search.best_params_, grid_search.best_params_, grid_search.bes
t_score_
```

In [ ]:

```
    gsearch1.grid_scores_, gsearch1.best_params_, gsearch1.best_score_
```

DecisionTreeClassifier

In [180]:

```
    dt = DecisionTreeClassifier()
    dt.fit(X_train_scale_20000, y_train)
    dt.score(X_test_scale_20000, y_test)
```

Out[180]:

```
    0.49
```

Понижение размерности

In [140]:

```
    from sklearn import decomposition
```

In [141]:

```
    colors = ['#990033', '#CC00FF', '#330066', '#00FF33', '#FF9900',
```

```
                '#3366FF', '#333300', '#999999', '#336666', '#FF33CC', '#009900', '#
FF9999']
```

In [142]:

In [215]:

```
# That's an impressive list of imports.
import numpy as np
from numpy import linalg
from numpy.linalg import norm
from scipy.spatial.distance import squareform, pdist

# We import sklearn.
import sklearn
from sklearn.manifold import TSNE
from sklearn.datasets import load_digits
from sklearn.preprocessing import scale

# We'll hack a bit with the t-SNE code in sklearn 0.15.2.
from sklearn.metrics.pairwise import pairwise_distances
from sklearn.manifold.t_sne import (_joint_probabilities,
_kl_divergence)
# from sklearn.utils.extmath import _ravel
# Random state.
RS = 20150101

# We'll use matplotlib for graphics.
import matplotlib.pyplot as plt
import matplotlib.patheffects as PathEffects
import matplotlib
%matplotlib inline

# We import seaborn to make nice plots.
import seaborn as sns
```

```
sns.set_style('darkgrid')
sns.set_palette('muted')
sns.set_context("notebook", font_scale=1.5,
rc={"lines.linewidth": 2.5})
```

In [216]:

```
def scatter(x, colors):
    # We choose a color palette with seaborn.
    palette = np.array(sns.color_palette("hls", 13))

    # We create a scatter plot.
    f = plt.figure(figsize=(8, 8))
    ax = plt.subplot(aspect='equal')
    sc = ax.scatter(x[:,0], x[:,1], lw=0, s=40,
                    c=palette[colors.astype(np.int)])
    plt.xlim(-25, 25)
    plt.ylim(-25, 25)
    ax.axis('off')
    ax.axis('tight')

    # We add the labels for each digit.
    txts = []
    for i in range(10):
        # Position of each label.
        xtext, ytext = np.median(x[colors == i, :], axis=0)
        txt = ax.text(xtext, ytext, str(i), fontsize=24)
        txt.set_path_effects([
            PathEffects.Stroke(linewidth=5, foreground="w"),
            PathEffects.Normal()])
        txts.append(txt)

    return f, ax, sc, txts
```

In [217]:

```
y = np.hstack(y_train)
```

In [692]:

```
digits_proj = TSNE(random_state=RS).fit_transform(X_train_10000_concat)
scatter(digits_proj, y)
plt.savefig('digits_tsne-generated.png', dpi=120)
```

In [693]:

```
proj = TSNE(random_state=RS).fit_transform(X_train_features_norm)
scatter(proj, y)
plt.savefig('X_train_features_norm.png', dpi=120)
```

In [694]:

```
proj = TSNE(random_state=RS).fit_transform(X_train_bigrams_freq_10000_norm)
scatter(proj, y)
plt.savefig('Bigrams_freq_10000_norm.png', dpi=120)
```

In [695]:

```
proj = TSNE(random_state=RS).fit_transform(X_train_10000_features)
scatter(proj, y)
plt.savefig('Features_10000_no_norm.png', dpi=120)
```

In [218]:

```
def count_tsne(X_train_features, X_test_features, X_train_bigrams_freq, X_test
_bigrams_freq, name_pic):
        # объединяем признаки и биграммы
        X_train_concat = concat_features_bigrams(X_train_features, X_train_bigrams
_freq)
        X_test_concat = concat_features_bigrams(X_test_features, X_test_bigrams_fr
eq)


        # скалируем
        scaler = preprocessing.StandardScaler()
        X_train_standard = scaler.fit_transform(X_train_concat)
        X_test_standard = scaler.fit_transform(X_test_concat)



        proj = TSNE(random_state=RS).fit_transform(X_train_standard)
        scatter(proj, np.hstack(y_train))
        plt.savefig(name_pic + '_train_stand.png', dpi=120)
```

```
    proj = TSNE(random_state=RS).fit_transform(X_test_standard)
    scatter(proj, np.hstack(y_test))
    plt.savefig(name_pic + '_test_stand.png', dpi=120)
```

In [697]:

```
count_tsne(X_train_features, X_test_features, X_train_bigrams_freq, X_test_big
rams_freq, 'L_20000_concat')
```

In [219]:

```
count_tsne(X_train_features_1000, X_test_features_1000, X_train_bigrams_freq_1
000, X_test_bigrams_freq_1000, 'L_1000_concat')
```

PCA

In [220]:

```
from sklearn import decomposition
```

In [223]:

```
colors = ['#990033', '#CC00FF', '#330066', '#00FF33', '#FF9900',
          '#3366FF', '#333300', '#999999', '#336666', '#FF33CC', '#009900', '#
FF9999']
```

In [716]:

```
# Прогоним встроенный в sklearn PCA
pca = decomposition.PCA(n_components=2)
pca.fit(X_train_features_norm)
X_pca = pca.transform(X_train_features_norm)

colors = ['#990033', '#CC00FF', '#330066', '#00FF33', '#FF9900',
          '#3366FF', '#333300', '#999999', '#336666', '#FF33CC', '#009900', '#
FF9999']
# И нарисуем получившиеся точки в нашем новом пространстве
for i, color, author in zip(range(0, 13), colors, author_coded.keys()):
    plt.plot(X_pca[y == i+1, 0], X_pca[y == i+1, 1], color, label=str(author))

plt.legend(loc=0);
```

In [222]:

```
import matplotlib.pyplot as plt
```

```
import seaborn as sns; sns.set(style='white')
%matplotlib inline
from sklearn import decomposition
from sklearn import datasets
from mpl_toolkits.mplot3d import Axes3D
```

In [236]:

```
pca = decomposition.PCA(n_components=3)
XX = pca.fit_transform(X_train_scale)
XX[y == 1, 2].mean()
```

Out[236]:

```
2.8819675671186733
```

In [230]:

```
def plot_pca_3d(X_in, y):

    pca = decomposition.PCA(n_components=3)
    X = pca.fit_transform(X_in)

    print('Projecting %d-dimensional data to 3D')

    # Заведём красивую трёхмерную картинку
    fig = plt.figure(1, figsize=(6, 5))
    plt.clf()
    ax = Axes3D(fig, rect=[0, 0, .95, 1], elev=48, azim=134)

    plt.cla()

    for name, label in zip(colors, range(1,13)):
        ax.text3D(X[y == label, 0].mean(),
                  X[y == label, 1].mean() + 1.5,
                  X[y == label, 2].mean(), name,
                  horizontalalignment='center',
                  bbox=dict(alpha=.5, edgecolor='w', facecolor='w'))
    # Поменяем порядок цветов меток, чтобы они соответствовали правильному
```

```
    # y_clr = np.choose(y, [1, 2, 0]).astype(np.float)
    ax.scatter(X[:, 0], X[:, 1], X[:, 2], c=y, cmap=plt.cm.spectral)

    ax.w_xaxis.set_ticklabels([])
    ax.w_yaxis.set_ticklabels([])
    ax.w_zaxis.set_ticklabels([])
```

In [238]:

```
plot_pca_3d(X_train_scale, y_train)
```

```
Projecting %d-dimensional data to 3D
```

In [221]:

```
def plot_pca(X, y):
    pca = decomposition.PCA(n_components=2)
    X_reduced = pca.fit_transform(X)

    print('Projecting %d-dimensional data to 2D' % X.shape[1])

    plt.figure(figsize=(12,10))
    plt.scatter(X_reduced[:, 0], X_reduced[:, 1], c=y,
                edgecolor='none', alpha=0.7, s=40,
                cmap=plt.cm.get_cmap('nipy_spectral', 12))
    plt.colorbar()
    plt.title('Authors. PCA projection')
```

In [732]:

```
plot_pca(X_test_features_norm, y_test)
```

```
Projecting 38-dimensional data to 2D
```

In [734]:

```
from sklearn.manifold import TSNE
tsne1 = TSNE(random_state=17)

X_tsne = tsne1.fit_transform(X_train_features_norm)

plt.figure(figsize=(12,10))
```

```
    plt.scatter(X_tsne[:, 0], X_tsne[:, 1], c=y_train,
                edgecolor='none', alpha=0.7, s=40,
                cmap=plt.cm.get_cmap('nipy_spectral', 12))
    plt.colorbar()
    plt.title('Authors. t-SNE projection')
```

Out[734]:

```
    Text(0.5,1,'Authors. t-SNE projection')
```

In [735]:

```
    pca = decomposition.PCA(n_components=2)
    pca.fit(X_train_features_norm)
    X_pca = pca.transform(X_train_features_norm)

    pca = decomposition.PCA(n_components=2)
    pca.fit(X_test_features_norm)
    X_test_pca = pca.transform(X_test_features_norm)
```

In [749]:

```
    def count_pca(data):
        pca = decomposition.PCA(n_components=2)
        pca.fit(data)
        X_pca = pca.transform(data)
        return X_pca
```

In [739]:

```
    from sklearn.tree import DecisionTreeClassifier
    from sklearn.metrics import accuracy_score, roc_auc_score
```

In [745]:

```
    logr = LogisticRegression().fit(X_pca, y_train)
    logr.score(X_test_pca, y_test)
```

Out[745]:

```
    0.22583333333333333
```

In [226]:

```
    def scaler(X_train_features, X_test_features, X_train_bigrams_freq, X_test_big
rams_freq):
```

```
        # объединяем признаки и биграммы
        X_train_concat = concat_features_bigrams(X_train_features, X_train_bigrams
_freq)
        X_test_concat = concat_features_bigrams(X_test_features, X_test_bigrams_fr
eq)


        # скалируем
        scaler = preprocessing.StandardScaler()
        X_train_standard = scaler.fit_transform(X_train_concat)
        X_test_standard = scaler.fit_transform(X_test_concat)


        # svc = SVC(kernel='linear').fit(X_train_standard, y_train)
        return X_train_standard, X_test_standard
```

In [228]:

```
X_train_scale, X_test_scale = scaler(X_train_features_20000, X_test_features_2
0000, X_train_bigrams_freq, X_test_bigrams_freq)
```

In [128]:

```
X_train_scale_20000, X_test_scale_20000 = scaler(X_train_features_20000, X_tes
t_features_20000, X_train_bigrams_freq_20000, X_test_bigrams_freq_20000)
X_train_scale_10000, X_test_scale_10000 = scaler(X_train_features_10000, X_tes
t_features_10000, X_train_bigrams_freq_10000, X_test_bigrams_freq_10000)
X_train_scale_5000, X_test_scale_5000 = scaler(X_train_features_5000, X_test_f
eatures_5000, X_train_bigrams_freq_5000, X_test_bigrams_freq_5000)
X_train_scale_1000, X_test_scale_1000 = scaler(X_train_features_1000, X_test_f
eatures_1000, X_train_bigrams_freq_1000, X_test_bigrams_freq_1000)
```

In [129]:

```
X_train_scale_500, X_test_scale_500 = scaler(X_train_features_500, X_test_feat
ures_500, X_train_bigrams_freq_500, X_test_bigrams_freq_500)
X_train_scale_200, X_test_scale_200 = scaler(X_train_features_200, X_test_feat
ures_200, X_train_bigrams_freq_200, X_test_bigrams_freq_200)
X_train_scale_100, X_test_scale_100 = scaler(X_train_features_100, X_test_feat
ures_100, X_train_bigrams_freq_100, X_test_bigrams_freq_100)
```

In [748]:

```
logr = LogisticRegression().fit(X_train_scale, y_train)
logr.score(X_test_scale, y_test)
```

Out[748]:

```
0.7744444444444445
```

In [750]:

```
logr = LogisticRegression().fit(count_pca(X_train_scale), y_train)
logr.score(count_pca(X_test_scale), y_test)
```

Out[750]:

```
0.2847222222222222
```

In [751]:

```
svc = SVC(kernel='linear').fit(count_pca(X_train_scale), y_train)
svc.score(count_pca(X_test_scale), y_test)
```

Out[751]:

```
0.2980555555555556
```

In [752]:

```
logr = SVC(kernel='linear').fit(X_train_scale, y_train)
logr.score(X_test_scale, y_test)
```

Out[752]:

```
0.7475
```

In [755]:

```
def plot_decision_regions(X,y,classifier,test_idx=None,resolution=0.02):

    # Initialise the marker types and colors
    markers = ('s','x','o','^','v', '!', '*', '-', '.', '$', '#', '@')
    colors = ('#990033', '#CC00FF', '#330066', '#00FF33', '#FF9900',
          '#3366FF', '#333300', '#999999', '#336666', '#FF33CC', '#009900', '#
FF9999')
    color_Map = ListedColormap(colors[:len(np.unique(y))]) #we take the color
mapping correspoding to the
                                                           #amount of classes
 in the target data

    # Parameters for the graph and decision surface
    x1_min = X[:,0].min() - 1
    x1_max = X[:,0].max() + 1
    x2_min = X[:,1].min() - 1
    x2_max = X[:,1].max() + 1
```

```
    xx1, xx2 = np.meshgrid(np.arange(x1_min,x1_max,resolution),
                           np.arange(x2_min,x2_max,resolution))

    Z = classifier.predict(np.array([xx1.ravel(),xx2.ravel()]).T)
    Z = Z.reshape(xx1.shape)

    plt.contour(xx1,xx2,Z,alpha=0.4,cmap = color_Map)
    plt.xlim(xx1.min(),xx1.max())
    plt.ylim(xx2.min(),xx2.max())

    # Plot samples
    X_test, Y_test = X[test_idx,:], y[test_idx]

    for idx, cl in enumerate(np.unique(y)):
        plt.scatter(x = X[y == cl, 0], y = X[y == cl, 1],
                    alpha = 0.8, c = color_Map(idx),
                    marker = markers[idx], label = cl
                   )
```

In [759]:

```
X_combined_sepal_standard = np.vstack((X_train_scale,X_test_scale))
Y_combined_sepal = np.hstack((y_train, y_test))
```

In [761]:

```
from matplotlib.colors import ListedColormap
```

In [764]:

```
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import accuracy_score
from sklearn.learning_curve import validation_curve

C_param_range = [0.001,0.01,0.1,1,10,100]

j = 0
for i in C_param_range:
```

```
    # Apply logistic regression model to training data
    lr = LogisticRegression(penalty = 'l2', C = i,random_state = 0)
    lr.fit(X_train_scale, y_train)


    print ('Score = ', lr.score(X_test_scale, y_test), ' . C = ', i)
```

```
Score =  0.7661111111111111  . C =  0.001
Score =  0.7663888888888889  . C =  0.01
Score =  0.7755555555555556  . C =  0.1
Score =  0.7744444444444445  . C =  1
Score =  0.7727777777777778  . C =  10
Score =  0.7658333333333334  . C =  100
```

In [765]:

```
tsne1 = TSNE(random_state=17)


X_train_tsne = tsne1.fit_transform(X_train_features_norm)
X_test_tsne = tsne1.fit_transform(X_test_features_norm)
```

Создание конвейерной обработки

In [112]:

```
from sklearn.pipeline import Pipeline
```

In [113]:

```
from sklearn.feature_extraction.text import CountVectorizer
from sklearn.feature_extraction.text import TfidfTransformer
from sklearn.naive_bayes import MultinomialNB
```

In [114]:

```
text_clf = Pipeline([('vect', CountVectorizer()),
                     ('tfidf', TfidfTransformer()),
                     ('clf', MultinomialNB()),
])
```

In [116]:

```
start_time = time.time()


text_clf = text_clf.fit(X_train_20000, y_train)
```

```
predicted = text_clf.predict(X_test_20000)
print('Acc: ', np.mean(predicted == y_test))


print("--- %s seconds ---" % (time.time() - start_time))
```

```
Acc:  0.6883333333333334
--- 170.39190793037415 seconds ---
```

In [117]:

```
from sklearn.linear_model import SGDClassifier
```

In [118]:

```
text_clf = Pipeline([('vect', CountVectorizer()),
                     ('tfidf', TfidfTransformer()),
                     ('clf', SGDClassifier(loss='hinge', penalty='l2',
                                           alpha=1e-3, n_iter=5, random_state=
42)),
])
_ = text_clf.fit(X_train_20000, y_train)
predicted = text_clf.predict(X_test_20000)
np.mean(predicted == y_test)
```

```
C:\Users\Yauheniya\Anaconda3\lib\site-packages\sklearn\linear_model\stochastic
_gradient.py:117: DeprecationWarning: n_iter parameter is deprecated in 0.19 and wi
ll be removed in 0.21. Use max_iter and tol instead.
  DeprecationWarning)
```

Out[118]:

```
0.7091666666666666
```

In [123]:

```
text_clf = Pipeline([('vect', CountVectorizer()),
                     ('tfidf', TfidfTransformer()),
                     ('clf', SGDClassifier(loss='hinge', penalty='l2',
                                           alpha=1e-3, n_iter=5, random_state=
42)),
])
_ = text_clf.fit(X_train_200, y_train)
predicted = text_clf.predict(X_test_200)
np.mean(predicted == y_test)
```

```
C:\Users\Yauheniya\Anaconda3\lib\site-packages\sklearn\linear_model\stochastic
_gradient.py:117: DeprecationWarning: n_iter parameter is deprecated in 0.19 and wi
ll be removed in 0.21. Use max_iter and tol instead.
  DeprecationWarning)
```

Out[123]:

```
0.33194444444444443
```

In [129]:

```
clf = SGDClassifier(loss='log', penalty='l2', alpha=1e-10, max_iter=1000, rand
om_state=42)
clf.fit(X_train_scale_20000, y_train)
clf.score(X_test_scale_20000, y_test)
```

Out[129]:

```
0.7322222222222222
```

In [130]:

```
clf = SGDClassifier(loss='log', penalty='l2', alpha=1e-10, max_iter=1000, rand
om_state=42)
clf.fit(X_train_scale_10000, y_train)
clf.score(X_test_scale_10000, y_test)
```

Out[130]:

```
0.7208333333333333
```

In [131]:

```
clf = SGDClassifier(loss='log', penalty='l2', alpha=1e-10, max_iter=1000, rand
om_state=42)
clf.fit(X_train_scale_5000, y_train)
clf.score(X_test_scale_5000, y_test)
```

Out[131]:

```
0.6636111111111112
```

In [137]:

```
clf = SGDClassifier(loss='log', penalty='l2', alpha=1e-10, max_iter=50, random
_state=42)
clf.fit(X_train_scale_1000, y_train)
clf.score(X_test_scale_1000, y_test)
```

Out[137]:

```
0.46555555555555556
```

In [136]:

```
clf = SGDClassifier(loss='log', penalty='l2', alpha=1e-10, max_iter=50, random
_state=42)
clf.fit(X_train_scale_500, y_train)
clf.score(X_test_scale_500, y_test)
```

Out[136]:

```
0.3947222222222222
```

In [135]:

```
clf = SGDClassifier(loss='log', penalty='l2', alpha=1e-10, max_iter=50, random
_state=42)
clf.fit(X_train_scale_200, y_train)
clf.score(X_test_scale_200, y_test)
```

Out[135]:

```
0.25472222222222224
```

In [124]:

```
clf = SGDClassifier(loss='log', penalty='l2', alpha=1e-3, max_iter=500, random
_state=42)
clf.fit(X_train_scale_20000, y_train)
clf.score(X_test_scale_20000, y_test)
```

Out[124]:

```
0.7477777777777778
```

In [238]:

```
clf = SGDClassifier(loss='log', penalty='l2', alpha=1e-3, max_iter=3, random_s
tate=42)
clf.fit(X_train_scale_20000, y_train)
clf.score(X_test_scale_20000, y_test)
```

Out[238]:

```
0.7588888888888888
```

In [125]:

```
clf = SGDClassifier(loss='log', penalty='l2', alpha=1e-10, max_iter=5, random_
state=42)
clf.fit(X_train_scale_20000, y_train)
clf.score(X_test_scale_20000, y_test)
```

Out[125]:

```
0.725
```

In [241]:

```
clf = SGDClassifier(loss='log', penalty='l2', alpha=1e-10, max_iter=5, random_
state=42)
clf.fit(X_train_scale_20000, y_train)
clf.score(X_test_scale_20000, y_test)
```

Out[241]:

```
0.7663888888888889
```

In [244]:

```
clf = SGDClassifier(loss='log', penalty='l2', alpha=1e-25, max_iter=3, random_
state=42)
clf.fit(X_train_scale_20000, y_train)
clf.score(X_test_scale_20000, y_test)
```

Out[244]:

```
0.7444444444444445
```

In [243]:

```
clf = SGDClassifier(loss='log', penalty='l2', alpha=1e-25, max_iter=5, random_
state=42)
clf.fit(X_train_scale_20000, y_train)
clf.score(X_test_scale_20000, y_test)
```

Out[243]:

```
0.7627777777777778
```

In [246]:

```
text_clf = Pipeline([('vect', CountVectorizer()),
                     ('tfidf', TfidfTransformer()),
                     ('clf', SVC(kernel = 'linear', C=0.01, gamma = 0.01)),
])
_ = text_clf.fit(X_train, y_train)
predicted = text_clf.predict(X_test)
np.mean(predicted == y_test)
```

Out[246]:

```
0.5311111111111111
```