

Компьютерная  
лингвистика.  
Статистический анализ  
текстовых данных

## **Теоретические положения**

Зародившись в 2009 году как проект группы аспирантов из Калифорнийского университета Беркли, Apache Spark стал одним из ведущих распределенных фреймворков обработки больших данных в мире. Spark может быть развернут различными способами, предоставляет собственное API для таких языков программирования как Scala, Java, Python, R. А так же поддерживает SQL, потоковые данные, машинное обучение и т.д. Spark используется банками, поисковиками, страховыми компаниями, службами такси, авиакомпаниями, учеными, правительственными учреждениями и такими крупными технологическими гигантами, как Amazon, Yandex, IBM Research, Yahoo!.

Spark представляет из себя фреймворк с открытым исходным кодом, входящий в экосистему большинства дистрибутивов Hadoop. Но из-за двух основных преимуществ Spark стал основой выбора при обработке больших данных, обогнав старую парадигму MapReduce, за счет которой так стал известен Hadoop.

Первое преимущество - скорость. Благодаря подсистеме обработки операций в оперативной памяти, Spark позволяет выполнять задачи до 100 раз быстрее, чем MapReduce в определенных ситуациях, в особенности это касается многоэтапных заданий, требующих постоянной записи состояний на диск между этапами. Даже на операциях, данные для которых не могут целиком содержаться в памяти, Spark показывает результаты в 10 раз быстрее, чем MapReduce.

И второе преимущество - это удобный для разработчиков Spark API. Можно утверждать, что дружелюбный программный интерфейс - это наиболее важное преимущество Spark.

Apache Spark состоит из следующих основных компонент: ядро, Spark SQL, MLlib, GraphX, Streaming. На рисунке 1 представлена экосистема Apache Spark [1].

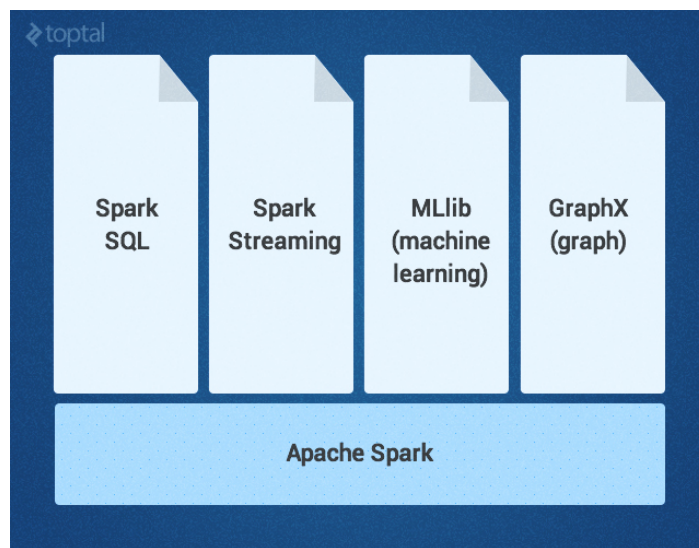


Рисунок 1 - Экосистема фреймворка Apache Spark

Основой фреймворка является Spark Core. Ядро обеспечивает распределенную диспетчеризацию, функции ввода-вывода, а так же планирование.

Spark SQL - одна из библиотек фреймворка, ориентированная на обработку структурированных данных. Используемые структуры данных называются DataFrames, заимствованные из R и Python (в библиотеке Pandas), которые выступают в роли распределенного механизма SQL-запросов.

Spark Streaming - это инструмент обработки потоковых данных в режиме micro-batch, которым можно манипулировать с помощью Apache Spark API.

Spark GraphX - это инструмент для масштабируемой обработки графовых структур. GraphX позволяет выполнять операции над структурой данных DataFrame.

Spark MLlib - это библиотека для применения методов машинного обучения, поставляющаяся с реализациями таких популярных алгоритмов, как: классификация, регрессия, деревья принятия решений, рекомендации и кластеризация [2].

Основная концепция Spark - RDD: Resilient Distributed Dataset. RDD из себя представляет произвольную коллекцию объектов, где удобнее всего работать с кортежами, как в реляционной таблице. RDD может быть распределена в памяти

или на диске, а может быть полностью виртуальной. RDD разбивается на минимальный объем RDD, партии, который будет обработан каждым рабочим узлом, что приводит к быстрой и масштабируемой параллельной обработке [3].

Spark MLlib - это распределенная система машинного обучения поверх Spark Core, которая, во многом благодаря архитектуре Spark, основанной на распределенной памяти, в 9 раз быстрее, чем реализация, используемая Apache Mahout. Библиотека MLlib имеет очень простую архитектуру и философию: благодаря ей над наборами RDD, являющимися распределенными массивами, можно применять разные алгоритмы. В качестве собственных типов данных MLlib использует LabeledPoint и Vector, которые из себя представляют множество функций для обработки наборов RDD.

Vector - это вектор в математическом смысле, который может быть либо плотным, либо разреженным.

LabeledPoint - маркированная точка в пространстве данных. Может использоваться в классификации и регрессии и состоит из вектора признаков и маркера.

Для использования на кластерах MLlib содержит только параллельные алгоритмы. Именно поэтому некоторые классические алгоритмы машинного обучения не нашли реализации в MLlib. В MLlib реализованы новейшие алгоритмы для кластеров, такие как распределенные случайные леса, метод k-средних и метод чередующихся наименьших квадратов. Это говорит о том, что библиотека особенно ярко демонстрирует себя при обучении моделей на огромных распределенных наборах данных [4].

Таким образом, следующие алгоритмы машинного обучения и статистики были реализованы и поставляются с MLlib:

- суммарная статистика, корреляции, стратифицированная выборка, проверка гипотез, генерация случайных данных;

- логистическая регрессия, линейная регрессия, деревья решений, наивная классификация Байеса;
- методы совместной фильтрации, включая чередование наименьших квадратов (ALS);
- методы кластерного анализа, включая k-средние и латентное распределение Дирихле (LDA);
- методы уменьшения размерности, такие как сингулярное разложение (SVD) и анализ главных компонент (PCA);
- функции извлечения и преобразования объектов;
- алгоритмы оптимизации, такие как стохастический градиентный спуск, ограниченная память BFGS (L-BFGS) [5].

**DataFrame** – это распределенный набор данных [6], представленный именованными столбцами. DataFrame – это аналог таблицы в реляционной базе данных или DataFrame'a в Python/R, но с улучшенной оптимизацией для распределенных вычислений. DataFrame может быть создан из большого количества источников: csv-файлов, таблиц в Hive, внешних баз данных или существующих RDD. DataFrame доступен из Scala, Java, Python и R. В Scala API и Java API DataFrame представлен как Dataset[Row] и Dataset<Row> соответственно.

**Токенизация** — это процесс разбиения текста на более мелкие части, называемые токенами.

**Стоп-слова** – это слова, которые должны быть исключены из входных данных, как правило, потому, что стоп-слова имеют низкую значимость, но высокую частоту встречаемости. К стоп-словам относятся местоимения, союзы, предлоги, частицы и т.д..

**TF-IDF.** Частота слова - обратная частота документа (tf-idf - term frequency - inverse document frequency) - это метод векторизации признаков, широко используемый в интеллектуальном анализе текста, который отражает важность термина для документа в корпусе документов (патентном массиве). Пусть  $t$  - терм,  $d$  -

документ,  $D$  – массив документов. Частота слова  $TF(t, d)$  – это количество раз, когда терм появился в документе. Частота документа  $DF(t, D)$  – это количество документов, содержащих терм  $t$ . Если будут использоваться только частоты появления слов, то очень легко переоценить термы, которые появляются очень часто, но несут мало информации о документе, например, предлоги. Если терм часто встречается в массиве документов, это означает, что он не несет особой информации о документе. Обратная частота документа – это числовая мера того, насколько информативно ценным является терм:

$$IDF(t, D) = \log \frac{|D| + 1}{DF(t, D) + 1},$$

где  $t$  – терм;

$D$  – массив документов;

$|D|$  – общее число документов в массиве документов.

Мера TF-IDF – это произведение двух сомножителей:

$$TFIDF(t, d, D) = TF(t, d) \cdot IDF(t, D),$$

где  $t$  – терм;

$D$  – массив документов;

$tf$  – частота слова;

$idf$  – обратная частота документа.

Более высокий показатель TF-IDF получают термы, которые более важны для данного документа, но реже употребляются в других документах массива документов. Это можно интерпретировать как знак того, что терм является важным для данного конкретного документа и может быть использован, чтобы точно описать этот документ.

**Word2Vec** используется для вычисления распределенного векторного представления слов. Основное преимущество распределенных представлений состоит в том, что подобные слова близки в векторном пространстве. Это облегчает обобщение на новые шаблоны и делает оценку модели более надежной.

Распределенное векторное представление полезно во многих приложениях обработки естественного языка, таких как распознавание именованных сущностей, устранение неоднозначности, синтаксический анализ, маркировка и машинный перевод.

В реализации модели Word2Vec в Spark использовалась модель skip-gram. Целью обучения skip-gram является изучение векторных представлений слов, которые хорошо предсказывают его контекст в одном предложении. Математически задана последовательность обучающих слов  $w_1, w_2, \dots, w_T$ . Цель модели skip-gram состоит в том, чтобы максимизировать среднее логарифмическое правдоподобие:

$$\frac{1}{T} \sum_{t=1}^T \sum_{j=-k}^{j=k} \log p(w_{t+j} | w_t),$$

где  $k$  - размер окна обучения.

В модели skip-gram каждое слово  $w$  связано с двумя векторами  $u_w$  и  $v_w$ , которые являются векторными представлениями  $w$  как слова и контекста соответственно. Вероятность правильного предсказания слова  $w_i$  заданного слова  $w_j$  определяется моделью softmax, которая:

$$p(w_i | w_j) = \frac{\exp(u_{w_i}^\top v_{w_j})}{\sum_{l=1}^V \exp(u_l^\top v_{w_j})},$$

где  $V$  — это размер словаря.

Модель skip-gram с softmax достаточно затратна, потому что сложность вычислений  $\log p(w_i/w_j)$  пропорциональна  $V$  и может быть порядка нескольких миллионов. Для ускорения обучения Word2Vec использовался иерархический softmax, который уменьшает сложность вычисления  $\log p(w_i/w_j)$  до  $O(\log(V))$  [7].

## Практическая часть

### Описание развертывания PySpark

Описанный способ установки подходит для дистрибутивов, основанных на Ubuntu 18.04 или выше.

Необходимо убедиться, что в операционной системе уже установлен Python 3.x. Для этого необходимо выполнить в терминале команду:

```
python3 --version
```

Для управления программными пакетами, написанными на Python, необходимо установить систему управления пакетами pip. Для установки системы управления пакетами pip в Ubuntu для Python 3 необходимо выполнить следующую команду в терминале:

```
sudo apt-get update  
sudo apt install python3-pip
```

Для запуска приложения с использованием PySpark в системе должен быть установлен Java JDK. Узнать версию установленной Java можно, выполнив в терминале команду:

```
java -version
```

Если Java не установлена, то можно выбрать одну из двух версий Java – свободную OpenJDK или проприетарную Oracle Java. Свободная версия есть в официальных репозиториях Ubuntu. Во избежание возникновения ошибок рекомендуется устанавливать версию Java 8 OpenJDK. Установить OpenJDK можно следующей командой в терминале:

```
sudo apt install openjdk-8-jdk
```

Можно установить PySpark через pip [8]

```
sudo pip3 --no-cache-dir install pyspark
```

но лучше скачать полную версию PySpark со страницы загрузки Apache Spark [9].

```
mkdir spark245  
cd spark245  
wget http://apache-mirror.rbc.ru/pub/apache/spark/spark-  
2.4.5/spark-2.4.5-bin-hadoop2.7.tgz  
sudo tar -zxvf spark-2.4.5-bin-hadoop2.7.tgz
```



Установим значения переменных среды

```
sudo nano /etc/environment
```

Добавить новую строку

```
JAVA_HOME="/usr/lib/jvm/java-8-openjdk-amd64"
```

Применим изменения

```
source /etc/environment
```

Чтобы PySpark запускался с использованием Python 3.x, необходимо создать следующие переменные окружения для текущего пользователя:

```
source /etc/environment
export PYSPARK_PYTHON=/usr/bin/python3
export PYSPARK_DRIVER_PYTHON=python3
export SPARK_HOME="/home/vagrant/spark245/spark-2.4.5-bin-
hadoop2.7"
export PATH="$SPARK_HOME/bin:$PATH"
```

Чтобы создать новые переменные окружения для текущего пользователя, необходимо добавить в конец скрытого файла .bashrc, находящегося в домашнем каталоге пользователя, приведенные выше строки. На рисунке 2 можно увидеть содержимое файла .bashrc.

```
sudo nano .bashrc
```

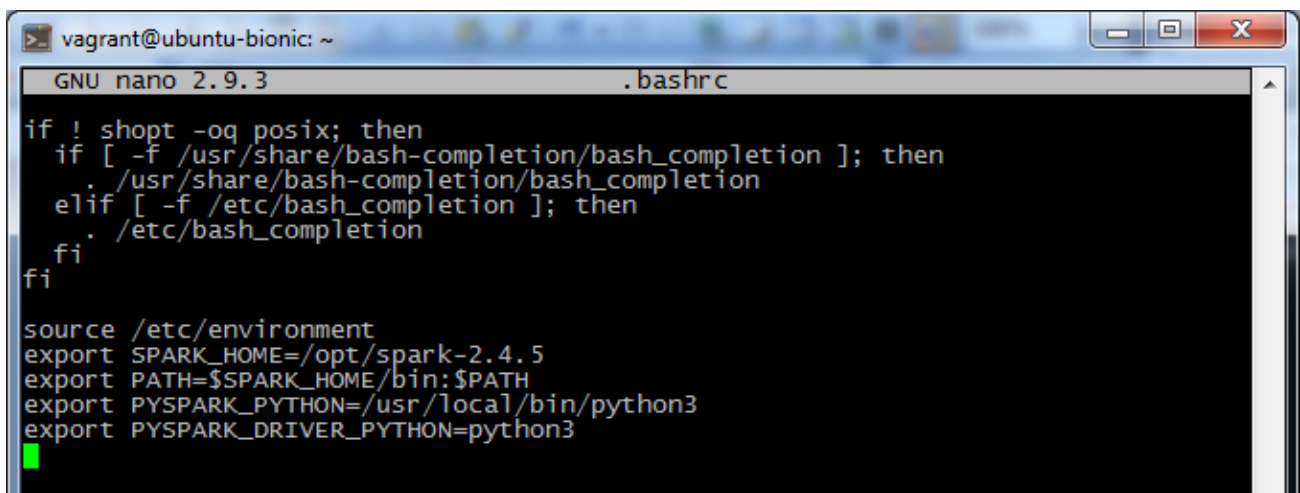


Рисунок 2 – содержимое файла .bashrc



```
mkdir имя_каталога  
cd имя_каталога
```

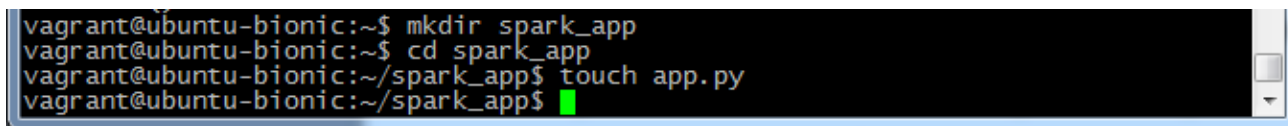
Для создания файла, содержащего код будущей программы, необходимо создать внутри каталога проекта файл с расширением .py. Данное расширение говорит о том, что этот файл содержит текст программы на языке программирования Python. Команда терминала:

```
touch app.py
```

Файл запускается на выполнение следующим образом

```
spark-submit /home/vagrant/spark_app/app.py
```

На рисунке 4 можно увидеть последовательность команд терминала для создания простого приложения на языке Python.



```
vagrant@ubuntu-bionic:~$ mkdir spark_app  
vagrant@ubuntu-bionic:~$ cd spark_app  
vagrant@ubuntu-bionic:~/spark_app$ touch app.py  
vagrant@ubuntu-bionic:~/spark_app$
```

Рисунок 4 – Последовательность команд терминала

Созданный файл можно открыть в любом удобном текстовом редакторе с подсветкой синтаксиса.

Точкой входа в приложение Spark служит Spark Session [10]. Spark Session обеспечивает способ взаимодействия с различными функциональными возможностями Spark с меньшим количеством конструкций. В Spark Session инкапсулируются Spark Context, SQL Context, Hive Context.

Для создания точки входа в коде программы необходимо, во-первых, импортировать модуль SparkSession. Для этого в начале кода программы в блоке импорта модулей необходимо написать:

```
from pyspark.sql import SparkSession
```

Затем необходимо создать непосредственно саму точку входа. С помощью класса Builder создается Spark Session. Подробнее о Builder можно прочитать в официальной документации Spark [11]. Ниже можно увидеть программный код для создания точки входа в приложение Spark:

```
spark = SparkSession\
```

```

        .builder\
        .appName("SimpleApplication")\
        .getOrCreate()

spark.stop()

```

В конце приложения Spark необходимо останавливать Spark Context командой `stop()`.

Для чтения и записи в RDD содержимого файлов в Spark имеется два способа.

`textFile(path)` – эта функция читает содержимое текстового файла из HDFS или локальной файловой системы и возвращает RDD, состоящую из строк этого файла.

Ниже в качестве примера представлен программный код для создания RDD с названием `input_file`, состоящей из строк файла `1.txt`:

```

input_file = spark.sparkContext.textFile('/home/vagrant/spark_app/Samples/1.txt')

```

На рисунке 5 можно увидеть содержимое RDD с названием `input_file`.

```

>>> print(input_file.collect())
['Apache Spark (от англ. spark – искра, вспышка) – фреймворк с открытым исходным кодом для реализации распределённой обработки неструктурированных и слабоструктурированных данных, входящий в экосистему проектов Hadoop. В отличие от классического обработчика из ядра Hadoop, реализующего двухуровневую концепцию MapReduce с дисковым хранилищем, Spark использует специализированные примитивы для рекуррентной обработки в оперативной памяти, благодаря чему позволяет получать значительный выигрыш в скорости работы для некоторых классов задач[5], в частности, возможность многократного доступа к загруженным в память пользовательским данным делает библиотеку привлекательной для алгоритмов машинного обучения[6].', 'ключевой автор – румынско-канадский учёный в области информатики Матей Захария (англ. Matei Zaharia), начал работу над проектом в 2009 году, будучи аспирантом Университета Калифорнии в Беркли. В 2010 году проект опубликован под лицензией BSD, в 2013 году передан фонду Apache и переведён на лицензию Apache 2.0, в 2014 году принят в число проектов верхнего уровня Apache.']
>>>

```

Рисунок 5 – Содержимое RDD с названием `input_file`.

Как можно заметить из рисунка, получившаяся RDD состоит из 2 строк, потому что во входном файле было только 2 строки. Каждая строка взята в одинарные кавычки.

`wholeTextFiles(path)` – эта функция читает содержимое каталога текстовых файлов из HDFS или локальной файловой системы и возвращает RDD с парами ключ-значение, где ключом выступает путь к файлу, а значением – содержимое файла целиком.

Ниже в качестве примера представлен программный код для создания RDD с названием `input_files`, в которой ключом выступает путь к файлу на локальной файловой системе, а значением – его содержимое:

```
input_files = spark.sparkContext.wholeTextFiles('/home/vagrant/spark_app/Samples/*.txt')
```

Стоит обратить внимание, что из каталога с названием `Samples` читаются только те файлы, что имеют расширение `".txt"`. На рисунке 6 можно увидеть строки RDD с названием `input_files`.

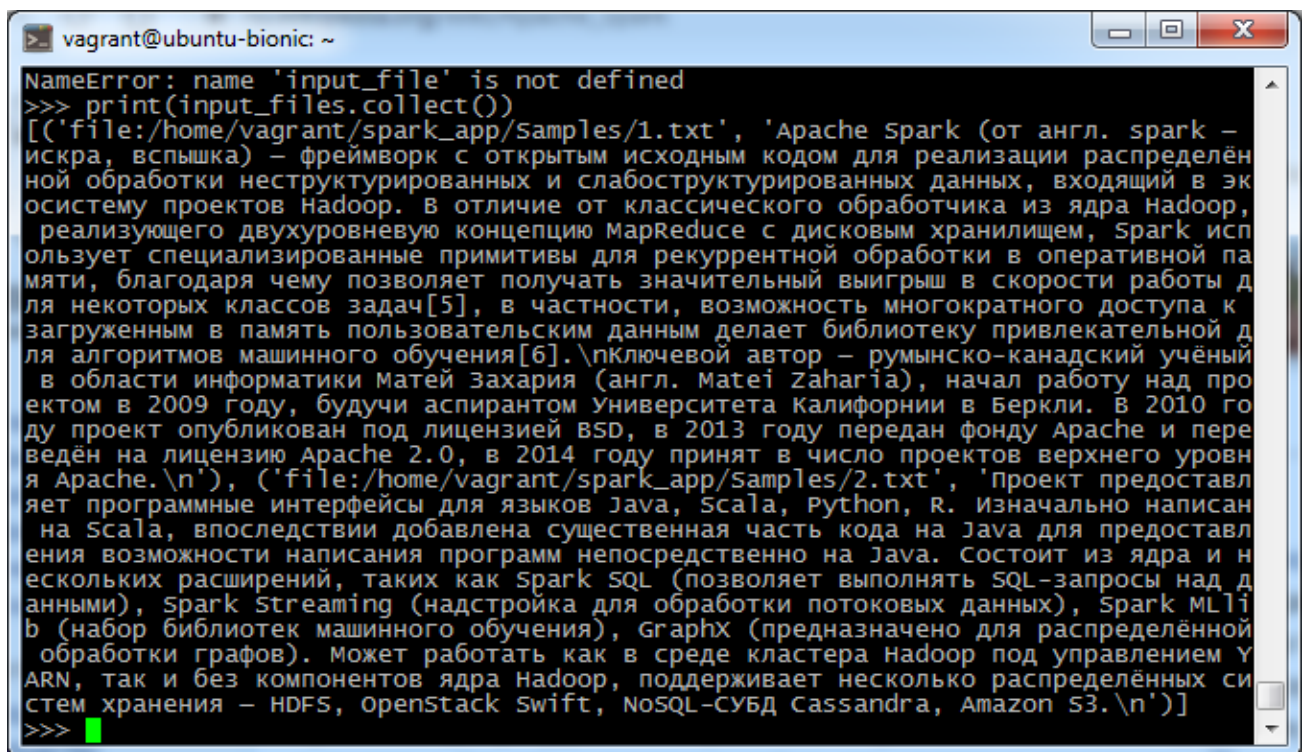


Рисунок 6 – Содержимое строк RDD с названием `input_files`.

Для вычисления значения TF-IDF необходимо предварительно разбить данные в RDD на токены. Для этого используется `Tokenizer`.

Перед этим надо установить дополнительную библиотеку

```
pip3 install numpy
```


Но Tokenizer применяется для данных типа DataFrame. Поэтому исходную RDD необходимо преобразовать в DataFrame. RDD, созданная при вызове функции `textFile` содержит строки типа `str`. Из такой RDD нельзя создать DataFrame. Поэтому необходимо применить `map`-операцию, в ходе которой строки исходной RDD будут заменены на списки, состоящие из одного элемента – первоначальной строки. Программный код для преобразования исходной RDD:

```
prepared = input_file.map(lambda x: ([x]))
```

```
df = prepared.toDF()
```

```
df.show()
```

На рисунке 7 можно увидеть вид полученной DataFrame.



```
>>> df.show()
+-----+
|_1|
+-----+
|Аpache spark (от ...|
|Ключевой автор - ...|
+-----+

>>> prepared_df.show()
+-----+
|text|
+-----+
|Аpache spark (от ...|
|Ключевой автор - ...|
+-----+

>>>
```

Рисунок 7 – Вид полученной DataFrame

Как можно заметить, название столбца таблицы имеет вид `"_1"`, такое название не является информативным. Изменить название столбца таблицы на `"text"` можно командой:

```
prepared_df = df.selectExpr('_1 as text')
```

Входными данными для Tokenizer служит название столбца таблицы, данные которой будут разбиваться на токены, а выходными – название столбца с полученными токенами.



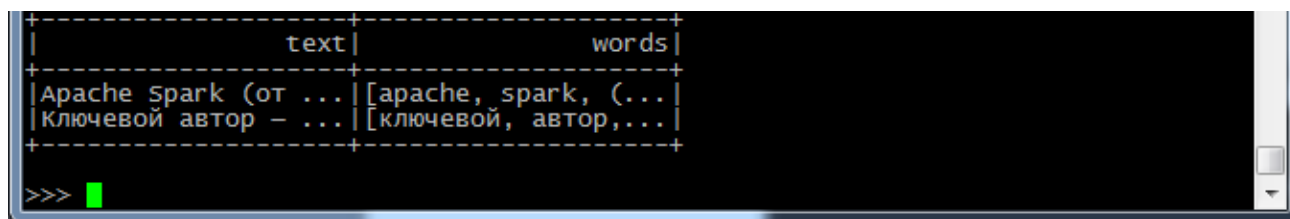
Для использования `Tokenizer` необходимо импортировать `Tokenizer` из `Mllib`, добавив в начало кода программы в блок импорта модулей строку:

```
from pyspark.ml.feature import Tokenizer
```

Программный код для получения таблицы `words`, содержащей токены:

```
tokenizer = Tokenizer(inputCol='text', outputCol='words')
words = tokenizer.transform(prepared_df)
words.show()
```

На рисунке 8 можно увидеть таблицу `words`, содержащую токены.



text	words
Apache spark (от ...)	[apache, spark, (...]
ключевой автор - ...	[ключевой, автор, ...]

Рисунок 8 – Таблица `words` с токенами

Прежде чем приступить к вычислению TF-IDF, нужно еще удалить все стоп-слова. Для этого предназначен `StopWordsRemover`. `StopWordsRemover` принимает на вход последовательность строк (например, выходные значения после применения `Tokenizer`) и удаляет все стоп-слова из входных данных. Список стоп-слов определяется параметром `stopWords`. Для текстов на английском языке этот параметр можно пропустить. Чтобы получить список стоп-слов для других доступных языков, необходимо вызвать

```
StopWordsRemover.loadDefaultStopWords (language)
```

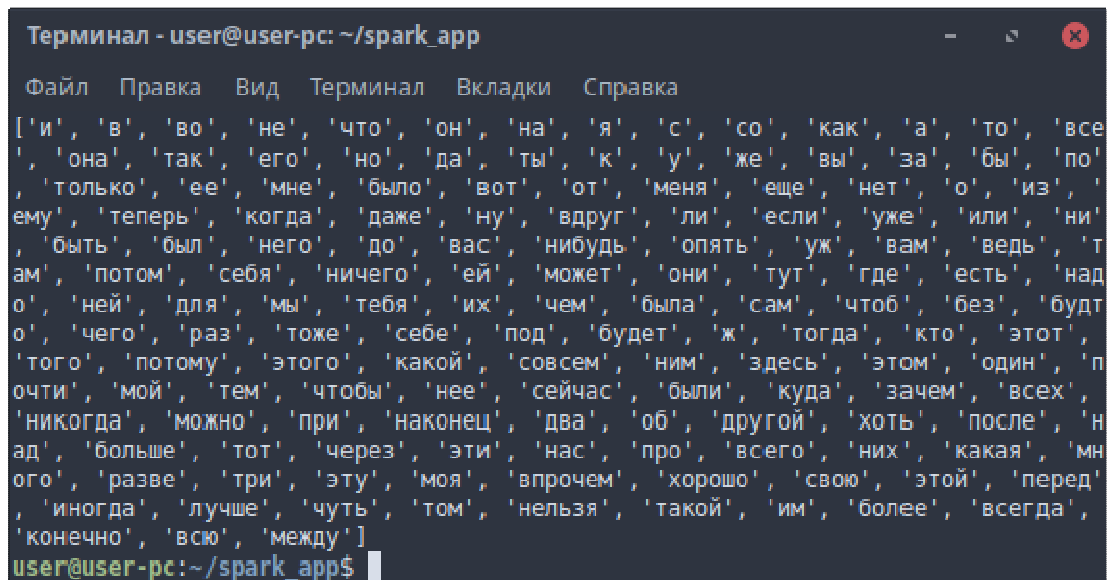
а затем передать получившийся список стоп-слов как параметр `stopWords` в `StopWordsRemover`. К доступным языкам относятся датский, голландский, английский, финский, французский, немецкий, венгерский, итальянский, норвежский, португальский, русский, испанский, шведский и турецкий. Также можно задавать свой собственный список стоп-слов и передавать его как параметр `stopWords`. Для использования `StopWordsRemover` в начале кода программы в блоке импорта модулей необходимо добавить строку:

```
from pyspark.ml.feature import StopWordsRemover
```

Программный код для создания списка стоп-слов для русского языка:

```
stop_words = StopWordsRemover.loadDefaultStopWords('russian')
```

На рисунке 9 можно увидеть список стоп-слов для русского языка в Spark



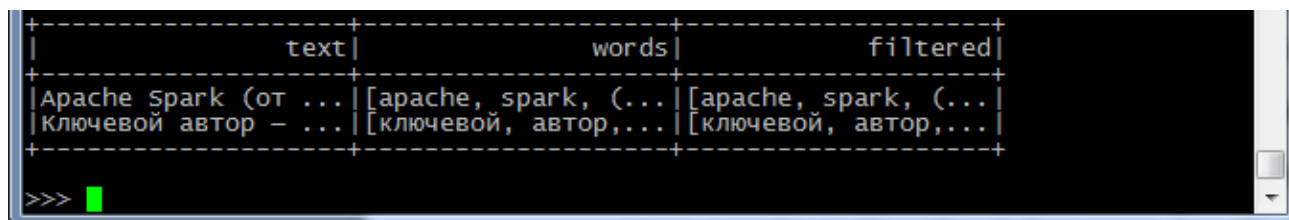
```
Терминал - user@user-pc: ~/spark_app
Файл  Правка  Вид  Терминал  Вкладки  Справка
['и', 'в', 'во', 'не', 'что', 'он', 'на', 'я', 'с', 'со', 'как', 'а', 'то', 'все',
', 'она', 'так', 'его', 'но', 'да', 'ты', 'к', 'у', 'же', 'вы', 'за', 'бы', 'по',
', 'только', 'ее', 'мне', 'было', 'вот', 'от', 'меня', 'еще', 'нет', 'о', 'из', '
ему', 'теперь', 'когда', 'даже', 'ну', 'вдруг', 'ли', 'если', 'уже', 'или', 'ни',
', 'быть', 'был', 'него', 'до', 'вас', 'нибудь', 'опять', 'уж', 'вам', 'ведь', 'т
ам', 'потом', 'себя', 'ничего', 'ей', 'может', 'они', 'тут', 'где', 'есть', 'над
о', 'ней', 'для', 'мы', 'тебя', 'их', 'чем', 'была', 'сам', 'чтоб', 'без', 'будт
о', 'чего', 'раз', 'тоже', 'себе', 'под', 'будет', 'ж', 'тогда', 'кто', 'этот',
', 'того', 'потому', 'этого', 'какой', 'совсем', 'ним', 'здесь', 'этом', 'один', 'п
очти', 'мой', 'тем', 'чтобы', 'нее', 'сейчас', 'были', 'куда', 'зачем', 'всех',
', 'никогда', 'можно', 'при', 'наконец', 'два', 'об', 'другой', 'хоть', 'после', 'н
ад', 'больше', 'тот', 'через', 'эти', 'нас', 'про', 'всего', 'них', 'какая', 'мн
ого', 'разве', 'три', 'эту', 'моя', 'впрочем', 'хорошо', 'свою', 'этой', 'перед'
', 'иногда', 'лучше', 'чуть', 'том', 'нельзя', 'такой', 'им', 'более', 'всегда',
', 'конечно', 'всю', 'между']
user@user-pc:~/spark_app$
```

Рисунок 9 – Список стоп-слов для русского языка в Spark

Программный код для удаления стоп-слов из таблицы words, полученной ранее, и записи полученных данных в таблицу filtered:

```
stop_words = StopWordsRemover.loadDefaultStopWords('russian')
remover = StopWordsRemover(inputCol="words",
outputCol="filtered", stopWords=stop_words)
filtered = remover.transform(words)
filtered.show()
```

На рисунке 10 можно увидеть таблицу filtered:



text	words	filtered
Apache spark (от ...	[apache, spark, (...]	[apache, spark, (...]
Ключевой автор – ...	[ключевой, автор,...]	[ключевой, автор,...]

Рисунок 10 – Таблица filtered

На рисунке 11 можно увидеть столбец таблицы words с токенами до удаления стоп-слов, а на рисунке 12 – после удаления стоп-слов.



```
Терминал - user@user-pc: ~/spark_app
Файл  Правка  Вид  Терминал  Вкладки  Справка
-----
words | [apache, spark, (от, англ., spark, -, искра,, вспышка), -, фреймворк, с, открытым, исходным, кодом, для, реали-
зации, распределённой, обработки, неструктурированных, и, слабоструктурированных, данных,, входящий, в, экосистему, про-
ектов, hadoop., в, отличие, от, классического, обработчика, из, ядра, hadoop,, реализующего, двухуровневую, концепцию,
mapreduce, с, дисковым, хранилищем,, spark, использует, специализированные, примитивы, для, рекуррентной, обработки, в,
оперативной, памяти,, благодаря, чему, позволяет, получать, значительный, выигрыш, в, скорости, работы, для, некоторых
, классов, задач,, в, частности,, возможность, многократного, доступа, к, загруженным, в, память, пользовательским, дан-
ным, делает, библиотеку, привлекательной, для, алгоритмов, машинного, обучения.]
```

Рисунок 11 – Столбец "words" таблицы words с токенами до удаления стоп-слов

```
Терминал - user@user-pc: ~/spark_app
Файл  Правка  Вид  Терминал  Вкладки  Справка
-----
filtered | [apache, spark, (от, англ., spark, -, искра,, вспышка), -, фреймворк, открытым, исходным, кодом, реализации
, распределённой, обработки, неструктурированных, слабоструктурированных, данных,, входящий, экосистему, проектов, hadoo
r., отличие, классического, обработчика, ядра, hadoop,, реализующего, двухуровневую, концепцию, mapreduce, дисковым, х
ранилищем,, spark, использует, специализированные, примитивы, рекуррентной, обработки, оперативной, памяти,, благодаря,
чему, позволяет, получать, значительный, выигрыш, скорости, работы, некоторых, классов, задач,, частности,, возможность
, многократного, доступа, загруженным, память, пользовательским, данным, делает, библиотеку, привлекательной, алгоритм
ов, машинного, обучения.]
```

Рисунок 12 – Столбец "filtered" таблицы filtered после удаления стоп-слов

Для получения вектора частоты встречаемости токенов (термов) используется CountVectorizer, предоставляемый MLlib, вместо HashingTF. HashingTF работает в 2 раза быстрее, но не дает возможности обратиться к словам в списке токенов по индексу. Далее инструментами MLlib вычисляются меры IDF и TF-IDF. Входными данными для CountVectorizer является название столбца с термами, частоту встречаемости которых необходимо найти, выходными – столбец с векторами частоты встречаемости термов. Для использования CountVectorizer в начале кода программы в блоке импорта модулей необходимо добавить строку:

```
from pyspark.ml.feature import CountVectorizer
```

Программный код для получения частоты встречаемости термов (TF):

```
vectorizer = CountVectorizer(inputCol="filtered",
outputCol="raw_features").fit(filtered)
featurized_data = vectorizer.transform(filtered)
featurized_data.cache()
```

В fit() передается таблица, частоту встречаемости термов которой необходимо найти.

На рисунке 13 можно увидеть таблицу со значениями частоты встречаемости термов.

```
>>> featurized_data.show()
+-----+-----+-----+-----+
|          text|          words|          filtered|          raw_feature
+-----+-----+-----+-----+
|Apache spark (от ...|[apache, spark, (...|[apache, spark, (...|(102,[0,2,3,4,5,6..
|. |
|ключевой автор - ...|[ключевой, автор,...|[ключевой, автор,...|(102,[1,2,3,5,10,..
|. |
+-----+-----+-----+-----+
>>>
```

Рисунок 13 – Таблица со значениями частоты встречаемости термов.

На рисунке 14 можно увидеть содержимое столбца `raw_features` для первой записи таблицы `featurized_data`.

```
>>> featurized_data.select('raw_features').show(truncate=False, vertical=True)
-RECORD 0-----
-----
raw_features | (102,[0,2,3,4,5,6,7,8,9,11,12,13,16,17,19,20,21,24,27,28,29,31,32,3
3,34,37,38,40,42,46,47,48,50,51,52,54,56,57,58,59,63,64,65,67,75,77,78,79,80,81,82,
83,84,86,87,89,91,92,93,95,96,97,101],[3.0,2.0,1.0,2.0,1.0,1.0,1.0,1.0,1.0,1.0,
1.0,1.0,1.0,1.0,1.0,1.0,1.0,1.0,1.0,1.0,1.0,1.0,1.0,1.0,1.0,1.0,1.0,1.0,1.0,1.0,
1.0,1.0,1.0,1.0,1.0,1.0,1.0,1.0,1.0,1.0,1.0,1.0,1.0,1.0,1.0,1.0,1.0,1.0,1.0,1.0,
1.0,1.0,1.0,1.0,1.0,1.0,1.0,1.0,1.0,1.0,1.0,1.0,1.0,1.0,1.0,1.0,1.0,1.0,1.0,1.0,
1.0,1.0,1.0,1.0,1.0,1.0,1.0,1.0,1.0,1.0,1.0,1.0,1.0,1.0])
```

Рисунок 14 – Содержимое столбца raw\_features для записи таблицы featurized\_data

В столбце `raw_features` хранятся `sparseVector`'а. Первое значение вектора – количество термов в словаре, полученном после использования `CountVectorizer`, второе значение – список индексов термов данной записи таблицы в словаре, третье значение – список частоты встречаемости термов данной записи таблицы. Получить список термов словаря для `vectorizer` можно с помощью его свойства `vocabulary`. Программный код для получения списка термов:

```
vocabulary = vectorizer.vocabulary
```

На рисунке 15 можно увидеть список термов в словаре.

```
>>> print(vocabulary)
['spark', 'году', '-', 'apache', 'обработки', 'проектов', 'данным', 'машинного', 'п
ривлекательной', 'англ.', 'аспирантом', 'hadoop', 'библиотеку', 'хранилищем', 'bs
d', '2010', 'искра', 'ядра', 'число', 'неструктурированных', 'hadoop.', 'реализац
ии', 'беркли.', 'начал', 'задач[5]', 'верхнего', 'захария', 'реализующего', 'конце
пцию', 'отличие', 'опубликован', 'позволяет', 'специализированные', 'классов', 'эко
систему', 'университета', 'zaharia)', '(от', 'делает', '(англ.', 'загруженным', 'п
ринят', 'чему', '2014', 'области', '2009', 'классического', 'обработчика', 'возможн
ость', 'информатики', 'доступа', 'выигрыш', 'распределённой', 'уровня', 'вспышка)',
'матей', 'получать', 'оперативной', 'памяти', 'mapreduce', 'переведён', 'учёный',
'калифорнии', 'частности', 'использует', 'данных', '2013', 'значительный', 'mate
i', 'будучи', 'лицензию', 'румынско-канадский', 'ключевой', 'лицензией', 'проект',
'открытым', '2.0', 'входящий', 'исходным', 'примитивы', 'кодом', 'фреймворк', 'мно
гократного', 'двухуровневую', 'дисковым', 'работу', 'слабоструктурированных', 'реку
ррентной', 'передан', 'некоторых', 'автор', 'обучения[6].', 'благодаря', 'скорости',
'apache.', 'алгоритмов', 'работы', 'пользовательским', 'году', 'проектом', 'фонд
у', 'память']
>>>
```

Рисунок 15 – список термов в словаре

Для поиска значений обратной частоты документа IDF необходимо использовать IDFModel. Входными значениями для IDFModel являются вектора с данными о частоте встречаемости термов, выходными – вектор значений TF-IDF. Первое значение вектора – количество термов в словаре, полученном после использования CountVectorizer, второе значение – список индексов термов данной записи таблицы в словаре, третье значение – список мер значимости TF-IDF для каждого терма. Для использования IDF в начале кода программы в блоке импорта модулей необходимо добавить строку:

```
from pyspark.ml.feature import IDF
```

Программный код для поиска TF-IDF:

```
idf = IDF(inputCol='raw_features', outputCol='features')
idf_model = idf.fit(featurized_data)
rescaled_data = idf_model.transform(featurized_data)
```

На рисунке 16 можно увидеть результат поиска значений TF-IDF

```
>>> rescaled_data.show()
+-----+-----+-----+-----+-----+
| text | words | filtered | raw_features | features |
+-----+-----+-----+-----+-----+
| Apache Spark (от ... | [apache, spark, (... | [apache, spark, (... | (102,[0,2,3,4,5,6,... | (102,[0,2,3,4,5,6,... |
| Ключевой автор - ... | [ключевой, автор,... | [ключевой, автор,... | (102,[1,2,3,5,10,... | (102,[1,2,3,5,10,... |
+-----+-----+-----+-----+-----+
```

Рисунок 16 – результат поиска значений TF-IDF

На рисунке 19 можно увидеть содержимое столбца features со значениями TF-IDF для первой записи таблицы rescaled\_data

```
Терминал - user@user-pc: ~/spark_app
Файл  Правка  Вид  Терминал  Окладки  Справка
-----
features | (102,[0,2,3,4,5,6,7,8,9,11,12,13,16,17,19,20,21,26,27,28,30,31,32,33,36,39,41,45,46,47,49,50,51,53,54,55,57
,58,59,63,65,66,68,69,71,73,75,76,77,78,79,80,81,82,83,84,87,90,92,93,96,97,101],[1.2163953243244932,0.0,0.0,0.81093021
62163283,0.0,0.4054651081081644,0.4054651081081644,0.4054651081081644,0.4054651081081644,0.405465108
1081644,0.4054651081081644,0.4054651081081644,0.4054651081081644,0.4054651081081644,0.40546510810816
44,0.4054651081081644,0.4054651081081644,0.4054651081081644,0.4054651081081644,0.4054651081081644,0.
4054651081081644,0.4054651081081644,0.4054651081081644,0.4054651081081644,0.4054651081081644,0.40546
51081081644,0.4054651081081644,0.4054651081081644,0.4054651081081644,0.4054651081081644,0.4054651081
081644,0.4054651081081644,0.4054651081081644,0.4054651081081644,0.4054651081081644,0.405465108108164
4,0.4054651081081644,0.4054651081081644,0.4054651081081644,0.4054651081081644,0.4054651081081644,0.4
054651081081644,0.4054651081081644,0.4054651081081644,0.4054651081081644,0.4054651081081644,0.405465
1081081644,0.4054651081081644,0.4054651081081644,0.4054651081081644,0.4054651081081644,0.40546510810
81081644,0.4054651081081644,0.4054651081081644,0.4054651081081644,0.4054651081081644,0.40546510810810
```

Рисунок 17 – Содержимое столбца features со значениями TF-IDF для первой записи таблицы rescaled\_data

Для использования модели Word2Vec в начале кода программы в блоке импорта модулей необходимо добавить строку:

```
from pyspark.ml.feature import Word2Vec
```

Программный код для построения модели Word2Vec:

```
word2Vec = Word2Vec(vectorSize=3, minCount=0, inputCol='words',
outputCol='result')
model = word2Vec.fit(words)
w2v_df = model.transform(words)
w2v_df.show()
```

На рисунке 18 можно увидеть результат построения модели Word2Vec

```
vagrant@ubuntu-bionic: ~/spark_app
20/02/14 15:11:44 INFO TaskSetManager: Starting task 0.0 in stage 27.0 (TID 34, localhost, executor driver, part
ition 1, PROCESS_LOCAL, 7902 bytes)
20/02/14 15:11:44 INFO Executor: Running task 0.0 in stage 27.0 (TID 34)
20/02/14 15:11:44 INFO HadoopRDD: Input split: file:/home/vagrant/spark_app/samples/1.txt:951+951
20/02/14 15:11:44 INFO PythonRunner: Times: total = 46, boot = -493, init = 539, finish = 0
20/02/14 15:11:44 INFO Executor: Finished task 0.0 in stage 27.0 (TID 34). 2727 bytes result sent to driver
20/02/14 15:11:44 INFO TaskSetManager: Finished task 0.0 in stage 27.0 (TID 34) in 63 ms on localhost (executor
driver) (1/1)
20/02/14 15:11:44 INFO TaskSchedulerImpl: Removed TaskSet 27.0, whose tasks have all completed, from pool
20/02/14 15:11:44 INFO DAGScheduler: ResultStage 27 (showString at NativeMethodAccessorImpl.java:0) finished in
0.071 s
20/02/14 15:11:44 INFO DAGScheduler: Job 22 finished: showString at NativeMethodAccessorImpl.java:0, took 0.0749
97 s
+-----+-----+-----+
|      text      |      words      |      result      |
+-----+-----+-----+
| Apache spark (от ... | [apache, spark, (... | [0.00107842319678... |
| ключевой автор - ... | [ключевой, автор,... | [0.06058470056658... |
+-----+-----+-----+
```

Рисунок 18 – Результат построения модели Word2Vec

Полный код программы приведен в приложении А.

Чтобы запустить программу на выполнение

```
spark-submit app.py
```

## **Пример решения прикладной задачи поиска наиболее значимых признаков в тексте патентных документов с использованием алгоритма TF-IDF**

Патентный документ - это документ, выдающийся уполномоченным органом государственной власти, подтверждающий исключительное право патентообладателя на изобретение, полезную модель либо на промышленный образец.

Патентные документы представляют собой XML-файлы. XML - это язык разметки, который осуществляет представление документов любого типа, сортировку, поиск информации, фильтрацию и представление информации в иерархическом виде. Информация содержится в тегах, причем теги не определяются никакими стандартами XML. На рисунке 19 можно увидеть пример патентного документа USPTO (<https://www.google.com/googlebooks/uspto-patents-grants-text.html>).

Для поиска наиболее значимых признаков патентов производится анализ тегов <claim>. В этих тегах содержится формула изобретения, и именно эти теги наиболее пригодны для обработки с целью извлечения информации. На рисунке 20 можно увидеть пример тегов <claim> в патентном документе.

Для анализа XML-файлов необходимо установить пакет Python с названием lxml. Установить lxml можно следующей командой в терминале:

```
sudo pip3 install lxml
```

Для использования lxml необходимо добавить в начало кода программы в блок импорта модулей строку:

```
from lxml import etree
```

```
/home/user/spark_app/Samples/US10205770.xml - Mousepad
Файл Правка Поиск Вид Документ Справка
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE us-patent-grant SYSTEM "us-patent-grant-v45-2014-04-03.dtd" [ ]>
<us-patent-grant lang="EN" dtd-version="v4.5 2014-04-03" file="US10205770-20190212.XML" status="PRODUCTION" id="us-pat
<us-bibliographic-data-grant>
<publication-reference>
<document-id>
<country>US</country>
<doc-number>10205770</doc-number>
<kind>B2</kind>
<date>20190212</date>
</document-id>
</publication-reference>
<application-reference appl-type="utility">
<document-id>
<country>US</country>
<doc-number>14327399</doc-number>
<date>20130108</date>
</document-id>
</application-reference>
<us-application-series-code>14</us-application-series-code>
<priority-claims>
<priority-claim sequence="01" kind="national">
<country>IN</country>
<doc-number>79/MUM/2012</doc-number>
<date>20120109</date>
</priority-claim>
</priority-claims>
<us-term-of-grant>
<us-term-extension>930</us-term-extension>
</us-term-of-grant>
<classifications-ipc>
<classification-ipc>
<ipc-version-indicator><date>20060101</date></ipc-version-indicator>
<classification-level>A</classification-level>
<section>G</section>
<class>06</class>
<subclass>F</subclass>
<main-group>15</main-group>
<subgroup>173</subgroup>
<symbol-position>F</symbol-position>
<classification-value>I</classification-value>
<action-date><date>20190212</date></action-date>
```

Рисунок 19 – Патентный документ USPTO

```
/home/user/spark_app/Samples/US10205770.xml - Mousepad
Файл Правка Поиск Вид Документ Справка
<?DOCTYPE description="Detailed Description" end="tail"/>
</description>
<us-claim-statement>What is claimed is:</us-claim-statement>
<claims id="claims">
<claim id="CLM-00001" num="00001">
<claim-text>1. A system for integrating a feature rich application platform operating on a mobile device and a vehicle
<claim-text>a head unit configured to control access to a user input device in the vehicle infotainment system, the head
<claim-text>a content consumer application operating under control of the processor in the head unit and configured to
<claim-text>receive from the content consumer application a selection of a first sub-application included in a plurali
<claim-text>launch the first sub-application for execution on the mobile device, wherein first sub-application starts
<claim-text>bind to the service started by the first sub-application to ensure that the commands received from the con
</claim-text>
</claim>
<claim id="CLM-00002" num="00002">
<claim-text>2. The system of <claim-ref idref="CLM-00001">claim 1</claim-ref> where:
<claim-text>the head unit further includes a user output device, and</claim-text>
<claim-text>the content consumer application is configured to receive messages from the first sub-application over the
</claim-text>
</claim>
<claim id="CLM-00003" num="00003">
<claim-text>3. The system of <claim-ref idref="CLM-00001">claim 1</claim-ref> where:
<claim-text>the user output device includes a display having space sufficient to display at least one sub-application
<claim-text>the content consumer application is configured to receive a sub-application name associated with the first
</claim-text>
</claim>
<claim id="CLM-00004" num="00004">
<claim-text>4. The system of <claim-ref idref="CLM-00001">claim 1</claim-ref> where:
<claim-text>the user-generated commands include a first command where the mobile device launches the first sub-applica
</claim-text>
</claim>
<claim id="CLM-00005" num="00005">
<claim-text>5. The system of <claim-ref idref="CLM-00004">claim 4</claim-ref> where:
<claim-text>the content consumer application is configured to receive messages generated by the first sub-application
</claim-text>
</claim>
<claim id="CLM-00006" num="00006">
<claim-text>6. The system of <claim-ref idref="CLM-00001">claim 1</claim-ref> where:
<claim-text>the user-generated commands include a second command where a next sub-application name in a list of sub-ap
</claim-text>
</claim>
<claim id="CLM-00007" num="00007">
```

Рисунок 20 – пример тегов <claim> в патентном документе

Загрузка патентов в RDD производится с использованием команды `wholeTextFiles()`, которая принимает в качестве параметра директорию с патентами. Чтобы читать из директории только XML-файлы, необходимо задать в пути шаблон, указав тип файла XML. Программный код для загрузки патентов в RDD:

```
input_data =
spark.sparkContext.wholeTextFiles(' /home/user/spark_app/Samples/*.xml
')
```

RDD с патентами имеет вид, который можно увидеть на рисунке 21. В первом столбце путь к файлу на жестком диске, во втором – содержимое патентного документа. Для удобства демонстрации RDD была приведена только одна строка RDD.

```

Терминал - user@user-pc: ~/spark_app
Файл Правка Вид Терминал Вкладки Справка
[('file:/home/user/spark_app/Samples/US10201534.xml', '<?xml version="1.0" encoding="UTF-8"?><!DOCTYPE us-patent-grant SYSTEM "us-patent-grant-v45-2014-04-03.dtd" [ ]><us-patent-grant lang="EN" dtd-version="v4.5 2014-04-03" file="US10201534-20190212.XML" status="PRODUCTION" id="us-patent-grant" country="US" date-produced="20190128" date-publ="20190212"><us-bibliographic-data-grant><publication-reference><document-id><country>US</country><doc-number>10201534</doc-number><kind>B2</kind><date>20190212</date></document-id></publication-reference><application-reference appl-type="utility"><document-id><country>US</country><doc-number>15895933</doc-number><date>20180213</date></document-id></application-reference><us application series code>15</us application series code><us term of grant></disclaimer><text>This patent is subject to a terminal disclaimer.</text></disclaimer></us-term-of-grant><classification-ipc><classification-ipc><ipc-version-indicator><date>20060101</date></ipc-version-indicator><classification-level>A</classification-level><section>A</section><class>61</class><subclass>K</subclass><main-group>31</main-group><subgroup>46</subgroup><symbol-position>F</symbol-position><classification-value>I</classification-value><action-date><date>20190212</date></action-date><generating-office><country>US</country></generating-office><classification-status>B</classification-status><classification-data-source>H</classification-data-source></classification-ipc><classification-ipc><ipc-version-indicator><date>20060101</date></ipc-version-indicator><classification-level>A</classification-level><section>A</section><class>61</class><subclass>K</subclass><main-group>9</main-group><subgroup>00</subgroup><symbol-position>L</symbol-position><classification-value>I</classification-value><action-date><date>20190212</date></action-date><generating-office><country>US</country></generating-office><classification-status>B</classification-status><classification-data-source>H</classification-data-source></classification-ipc><classification-ipc><ipc-version-indicator><date>20060101</date></ipc-version-indicator><classification-level>A</classification-level><section>A</section><class>61</class><subclass>K</subclass><main-group>47</main-group><subgroup>02</subgroup><symbol-position>L</symbol-position><classification-value>I</classification-value>

```

Рисунок 21 – строка RDD с загруженными патентами

Поскольку такое содержимое не пригодно для применения алгоритма TF-IDF, необходимо применить предобработку текстовых данных. К изначальной RDD с парами ключ-значение, где ключ – путь к файлу на жестком диске, а значение – содержимое патента, применяется 3 последовательных map-операции. Первая map-операция изменяют ключ RDD и изменяет значение с XML-текстом патента на текст, извлеченный из тегов <claim>. Ключом становится номер патента и код страны. Для получения номера патента и кода страны необходимо извлечь значения из тегов <doc-number> и <country>. Программный код для получения номера патента и кода страны:

```

def get_patent_name(patent_data):
    root = etree.fromstring(patent_data.encode('utf-8'))
    doc_numers = root.findall("./doc-number")
    countries = root.findall("./country")
    patent_name = countries[0].text + doc_numers[0].text

```



```
return patent_name
```

Программный код функции для получения текста, заключенного в теги

<claim>:

```
def get_claims(patent_data):
    root = etree.fromstring(patent_data.encode('utf-8'))
    claims = root.findall("./claim")
    if(claims):
        claims_without_tags = [''.join(claim.itertext()) for
claim in claims]
        return ''.join(claims_without_tags)

    return ''
```

Вторая map-операция изменяет только значение, не затрагивая ключи RDD. Из полученных claim'ом удаляется пунктуация. Для использования punctuation модуля string необходимо добавить в начало кода программы в блок импорта модулей строку:

```
import string
```

Программный код функции для удаления пунктуации:

```
def remove_punctuation(text):
    return text.translate(str.maketrans('', '',
string.punctuation))
```

Третья map-операция удаляет все разрывы строк. Для реализации этой функции использовался модуль регулярных выражений, поэтому необходимо добавить в начало кода программы в блок импорта модулей строку:

```
import re
```

Программный код функции для удаления разрыва строк:

```
def remove_linebreaks(text):
    return text.strip()
```

Программный код для предобработки загруженных данных:

```
prepared_data = input_data.map(lambda x:
(get_patent_name(x[1]), get_claims(x[1]))) \
```

```
.map(lambda x: (x[0], remove_punctuation(x[1]))) \
.map(lambda x: (x[0], remove_linebreaks(x[1])))
```

Для дальнейшего решения поставленной задачи необходимо разбить текст claim'ов на токены. Для этого используется Tokenizer:

```
prepared_df = prepared_data.toDF().selectExpr('_1 as
patent_name', '_2 as patent_claims')
tokenizer = Tokenizer(inputCol="patent_claims",
outputCol="words")
words_data = tokenizer.transform(prepared_df)
```

Поскольку полученные токены могут содержать не только слова, но цифры и другие символы, непригодные для обработки, необходимо оставить только токены, являющиеся словами. Программный код функции для получения токенов, являющихся только словами с использованием модуля регулярных выражений:

```
def get_only_words(tokens):
    return list(filter(lambda x: re.match('[a-zA-Z]+', x),
tokens))
```

Программный код для фильтрации токенов:

```
filtered_words_data = words_data.rdd.map(lambda x: (x[0], x[1],
get_only_words(x[2])))
filtered_df = filtered_words_data.toDF().selectExpr('_1 as
patent_name', '_2 as patent_claims', '_3 as words')
```

Поскольку обрабатываемый патентный массив содержит патенты на английском языке, в StopWordsRemover нет необходимости передавать список стоп-слов, так как параметр stopWords по умолчанию установлен для англоязычных текстов и его можно опустить. Программный код для удаления стоп-слов из текста патентных документов:

```
remover = StopWordsRemover(inputCol='words',
outputCol='filtered')
filtered = remover.transform(filtered_df)
```

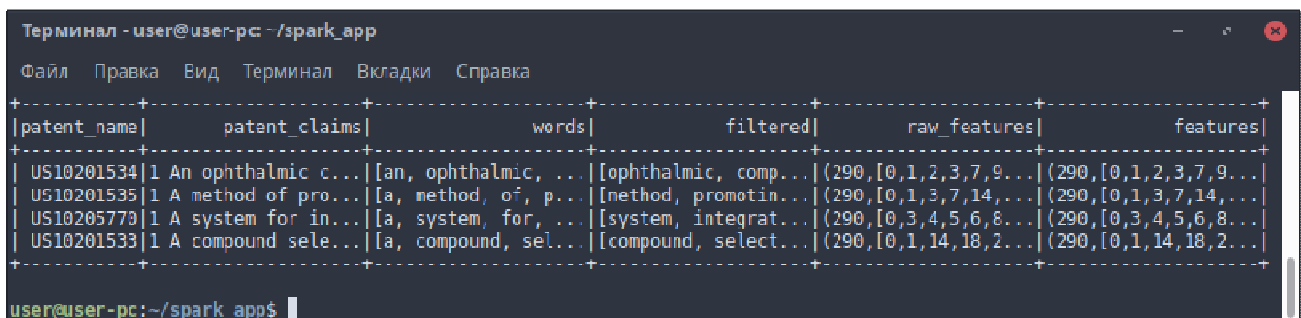
Программный код для поиска наиболее значимых признаков в тексте патентных документов:

```

vectorizer          =          CountVectorizer(inputCol='filtered',
outputCol='raw_features').fit(filtered)
featurized_data = vectorizer.transform(filtered)
featurized_data.cache()
idf = IDF(inputCol='raw_features', outputCol='features')
idf_model = idf.fit(featurized_data)
rescaled_data = idf_model.transform(featurized_data)

```

На рисунке 22 можно увидеть таблицу с результатом выполнения программы по извлечению наиболее значимых признаков из патентных документов.



patent_name	patent_claims	words	filtered	raw_features	features
US10201534	1 An ophthalmic c...	[an, ophthalmic, ...]	[ophthalmic, comp...	(290,[0,1,2,3,7,9...	(290,[0,1,2,3,7,9...
US10201535	1 A method of pro...	[a, method, of, p...	[method, promotin...	(290,[0,1,3,7,14,...]	(290,[0,1,3,7,14,...]
US10205770	1 A system for in...	[a, system, for, ...]	[system, integrat...	(290,[0,3,4,5,6,8...	(290,[0,3,4,5,6,8...
US10201533	1 A compound sele...	[a, compound, sel...	[compound, select...	(290,[0,1,14,18,2...	(290,[0,1,14,18,2...

Рисунок 22 – Результат выполнения программы по извлечению наиболее значимых признаков из патентных документов

Полный код программы приведен в приложении Б.

### Задание

Взять произвольные 100 патентов из хранилища USPTO (<https://www.google.com/googlebooks/uspto-patents-grants-text.html>).

Для каждого из патентов определить его значимые признаки.

На данном множестве патентов построить модель Word2Vec. На основе данной модели предоставить пользователю получать список контекстных синонимов для слова.

### Список использованных источников

1 Pointer, I. What is Apache Spark? The big data analytics platform explained [Электронный ресурс] / I. Pointer. - 2017. - Режим доступа : <https://www.infoworld.com/article/3236869/analytics/what-is-apache-spark-the-big-data-analytics-platform-explained.html>.

2 Apache Spark: из open source в индустрию [Электронный ресурс]. - 2017. - Режим доступа : <https://habr.com/company/netologyru/blog/331728>.

3 Apache Spark: что там под капотом? [Электронный ресурс]. - 2015. - Режим доступа : <https://habr.com/post/251507>.

4 Изучаем Spark: молниеносный анализ данных / Х. Карау, Э. Конвински, П. Венделл, М. Захария. - Москва: ДМК Пресс, 2015. - 304 с.

5 MLlib is Apache Spark's scalable machine learning library [Электронный ресурс]. - 2018. - Режим доступа : <http://spark.apache.org/mllib>.

6 Spark SQL [Электронный ресурс]. - Режим доступа : [https://ru.bmstu.wiki/Spark\\_SQL#DataFrame](https://ru.bmstu.wiki/Spark_SQL#DataFrame).

7 Feature Extraction and Transformation - RDD-based API [Электронный ресурс]. - Режим доступа : <https://spark.apache.org/docs/2.2.0/mllib-feature-extraction>.

8 pyspark 2.4.3 [Электронный ресурс]. - Режим доступа : <https://pypi.org/project/pyspark/>

9 Downloads | Apache Spark [Электронный ресурс]. - Режим доступа : <https://spark.apache.org/downloads.html>.

10 A tale of Spark Session and Spark Context [Электронный ресурс]. - Режим доступа : <https://medium.com/@achilleus/spark-session-10d0d66d1d24>.

11 pyspark.sql module [Электронный ресурс]. - Режим доступа : <https://spark.apache.org/docs/preview/api/python/pyspark.sql.html>.

## Приложение А. Код приложения с использованием PySpark

```
from pyspark.sql import SparkSession
from pyspark.ml.feature import Tokenizer
from pyspark.ml.feature import StopWordsRemover
from pyspark.ml.feature import CountVectorizer
from pyspark.ml.feature import IDF
from pyspark.ml.feature import Word2Vec

spark = SparkSession\
    .builder\
    .appName("SimpleApplication")\
    .getOrCreate()

# Построчная загрузка файла в RDD
input_file =
spark.sparkContext.textFile('/home/vagrant/spark_app/Samples/1.txt')

print(input_file.collect())
prepared = input_file.map(lambda x: ([x]))
df = prepared.toDF()
prepared_df = df.selectExpr('_1 as text')

# Разбить на токены
tokenizer = Tokenizer(inputCol='text', outputCol='words')
words = tokenizer.transform(prepared_df)

# Удалить стоп-слова
stop_words = StopWordsRemover.loadDefaultStopWords('russian')
remover = StopWordsRemover(inputCol='words',
outputCol='filtered', stopWords=stop_words)
filtered = remover.transform(words)
```

```
# Вывести стоп-слова для русского языка
print(stop_words)

# Вывести таблицу filtered
filtered.show()

# Вывести столбец таблицы words с токенами до удаления стоп-
слов
words.select('words').show(truncate=False, vertical=True)

# Вывести столбец "filtered" таблицы filtered с токенами после
удаления стоп-слов
filtered.select('filtered').show(truncate=False, vertical=True)

# Посчитать значения TF
vectorizer = CountVectorizer(inputCol='filtered',
outputCol='raw_features').fit(filtered)
featurized_data = vectorizer.transform(filtered)
featurized_data.cache()
vocabulary = vectorizer.vocabulary

# Вывести таблицу со значениями частоты встречаемости термов.
featurized_data.show()

# Вывести столбец "raw_features" таблицы featurized_data
featurized_data.select('raw_features').show(truncate=False,
vertical=True)

# Вывести список термов в словаре
print(vocabulary)

# Посчитать значения DF
```

```
idf = IDF(inputCol='raw_features', outputCol='features')
idf_model = idf.fit(featurized_data)
rescaled_data = idf_model.transform(featurized_data)

# Вывести таблицу rescaled_data
rescaled_data.show()

# Вывести столбец "features" таблицы featurized_data
rescaled_data.select('features').show(truncate=False,
vertical=True)

# Построить модель Word2Vec
word2Vec = Word2Vec(vectorSize=3, minCount=0, inputCol='words',
outputCol='result')
model = word2Vec.fit(words)
w2v_df = model.transform(words)
w2v_df.show()

spark.stop()
```

## Приложение Б. Программа по поиску наиболее значимых признаков в тексте патентных документов с использованием алгоритма TF-IDF

```
from pyspark.sql import SparkSession
from pyspark.ml.feature import Tokenizer
from pyspark.ml.feature import StopWordsRemover
from pyspark.ml.feature import CountVectorizer
from pyspark.ml.feature import IDF
from lxml import etree
import re
import string

def get_patent_name(patent_data):

    """
    Получение названия патента
    """

    root = etree.fromstring(patent_data.encode('utf-8'))
    doc_numers = root.findall("./doc-number")
    countries = root.findall("./country")
    patent_name = countries[0].text + doc_numers[0].text
    return patent_name

def get_claims(patent_data):

    """
    Проверка claim'ов патента
    """

    root = etree.fromstring(patent_data.encode('utf-8'))
    claims = root.findall("./claim")
    if(claims):
        claims_without_tags = [''.join(claim.itertext()) for
claim in claims]
        return ''.join(claims_without_tags)
```



```

        return ''

def remove_punctuation(text):

    """
    Удаление пунктуации из текста
    """

    return text.translate(str.maketrans('', '',
string.punctuation))

def remove_linebreaks(text):

    """
    Удаление разрыва строк из текста
    """

    return text.strip()

def get_only_words(tokens):

    """
    Получение списка токенов, содержащих только слова
    """

    return list(filter(lambda x: re.match('[a-zA-Z]+', x),
tokens))

spark = SparkSession\
    .builder\
    .appName("SimpleApplication")\
    .getOrCreate()

```

```

input_data =
spark.sparkContext.wholeTextFiles('/home/user/spark_app/Samples/*.xml
')

print(input_data.take(4))

prepared_data = input_data.map(lambda x:
(get_patent_name(x[1]), get_claims(x[1]))) \
    .map(lambda x: (x[0], remove_punctuation(x[1]))) \
    .map(lambda x: (x[0], remove_linebreaks(x[1])))

prepared_df = prepared_data.toDF().selectExpr('_1 as
patent_name', '_2 as patent_claims')

# Разбить claims на токены
tokenizer = Tokenizer(inputCol="patent_claims",
outputCol="words")
words_data = tokenizer.transform(prepared_df)

# Отфильтровать токены, оставив только слова
filtered_words_data = words_data.rdd.map(lambda x: (x[0], x[1],
get_only_words(x[2])))
filtered_df = filtered_words_data.toDF().selectExpr('_1 as
patent_name', '_2 as patent_claims', '_3 as words')

# Удалить стоп-слова (союзы, предлоги, местоимения и т.д.)
remover = StopWordsRemover(inputCol='words',
outputCol='filtered')
filtered = remover.transform(filtered_df)

vectorizer = CountVectorizer(inputCol='filtered',
outputCol='raw_features').fit(filtered)
featurized_data = vectorizer.transform(filtered)

```

```
featurized_data.cache()

idf = IDF(inputCol='raw_features', outputCol='features')
idf_model = idf.fit(featurized_data)
rescaled_data = idf_model.transform(featurized_data)

# Вывести таблицу rescaled_data
rescaled_data.show()

spark.stop()
```