BIA – 678E
BIG DATA TECHNOLOGIES

OPINION MINING ON AMAZON PRODUCT REVIEW DATASET

Final Project Report

Submitted to Prof.Venu Guntupalli

by

Akhil Meda Vasudeva Murthy - 2008878

Sushmitha Mohana - 20008501

Gayatri Kondapavuluri - 10477760

INTRODUCTION

Sentiment analysis, often known as opinion mining, is a branch of natural language processing. It elicits people's opinions, including judgments, attitudes, and feelings toward particular people, subjects, and events. People and organizations are using public opinion for decision-making more and more as a result of the rapid expansion of digital platforms in cyberspace like blogs and social networks. A significant study on using opinion mining to mine people's sentiments based on text in cyberspace has been done recently. Numerous opinions increasingly using public opinion for decision-making due to mining techniques, such as machine learning and lexicon-based approaches, have been used by researchers to analyze and categorize people's attitudes based on a text and debate the existing gap.

Data Science and Machine learning tasks have radically changed our thinking and made our lives easy in predicting the future outcomes. Considering there is a lot of information today on the internet and it increases exponentially in a fraction of time we need to store it and generate insights. The main constraint here is time. Sentiment analysis has become an interesting field for both research and industrial domains. The expression sentiment refers to the feelings or thought of the person across some certain issues

In this project, the Apache Spark framework is used to perform sentiment analysis on Amazon reviews from two product categories i.e beauty and apparel, using its Machine Learning Library (MLliB). Preprocessing and machine learning text feature extraction are used for improved sentiment analysis classification outcomes. The project concludes with a performance evaluation of the classification algorithms used such as Naive Bayes, Random Forest and Logistic Regression for the sentiment analysis

OBJECTIVE

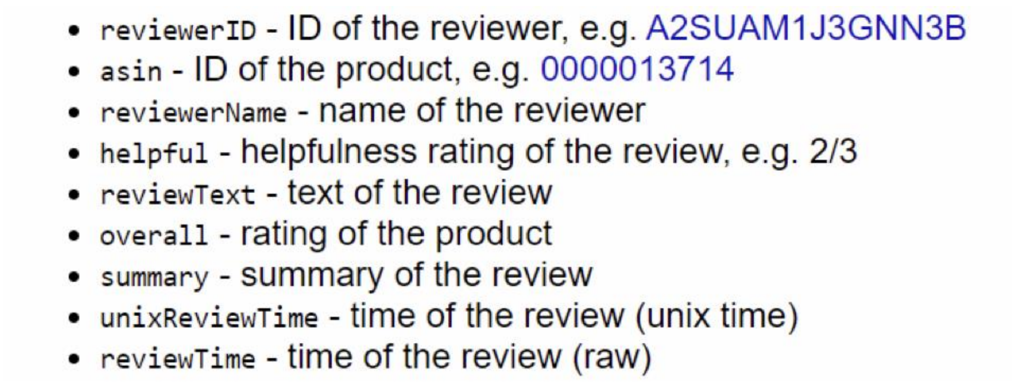
- To perform a sentiment analysis using Apache Spark framework, an open-source distributed data processing platform which utilizes distributed memory abstraction.
- Perform data preprocessing on the given text and apply ML text feature extraction steps in Classification.
- Evaluate performance of classification algorithms used such as Naive Bayes, Random Forest, and Logistic Regression for the sentiment analysis
- The goal of using Apache Spark’s Machine learning library (MLlib) is to handle an extraordinary amount of data effectively using Pyspark, an interface for Apache Spark in Python

DATASET

The source website from where the dataset was taken, consists of links to product metadata (descriptions, category information, price, brand, and picture features) and product reviews (ratings, text, helpfulness votes) for products belonging to the 30+ product categories sold on Amazon.

The product review datasets for product categories Beauty and Apparel from the year 2018 were chosen. These datasets were combined to get a total of 400,000+ records, which was used for the purpose of this project.

The dataset consists of the following columns:

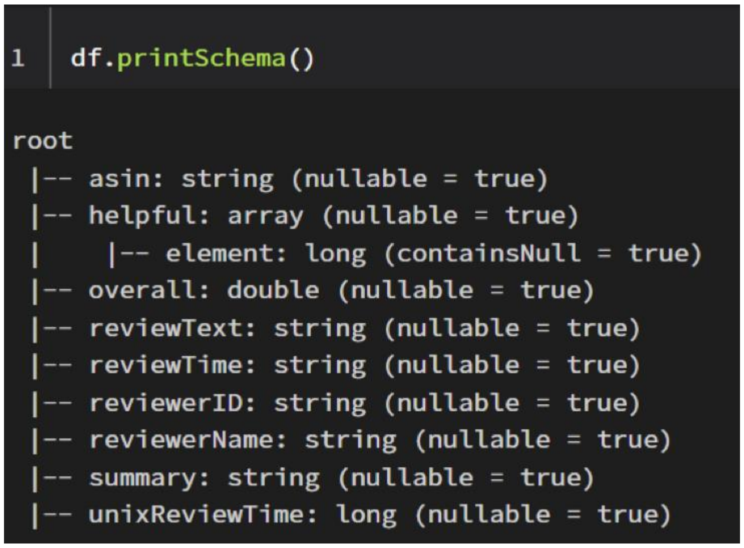


- `reviewerID` - ID of the reviewer, e.g. A2SUAM1J3GNN3B
- `asin` - ID of the product, e.g. 0000013714
- `reviewerName` - name of the reviewer
- `helpful` - helpfulness rating of the review, e.g. 2/3
- `reviewText` - text of the review
- `overall` - rating of the product
- `summary` - summary of the review
- `unixReviewTime` - time of the review (unix time)
- `reviewTime` - time of the review (raw)

The columns most relevant for our project are ‘overall’, ‘Text’, and ‘summary’.

Note that an ‘overall’ value of 1 is the least positive rating , while 3 is a neutral rating and 5 is the most positive rating that a product can receive.

We have Further used printSchema() to observe the structure of the dataframe, which is each column with its corresponding type as shown below.



```
df.printSchema()

root
 |-- asin: string (nullable = true)
 |-- helpful: array (nullable = true)
 |    |-- element: long (containsNull = true)
 |-- overall: double (nullable = true)
 |-- reviewText: string (nullable = true)
 |-- reviewTime: string (nullable = true)
 |-- reviewerID: string (nullable = true)
 |-- reviewerName: string (nullable = true)
 |-- summary: string (nullable = true)
 |-- unixReviewTime: long (nullable = true)
```

Fig 1: Structure of the dataframe

PROJECT FLOWCHART:

The below figures are intended to give a brief idea about the process flow of the project

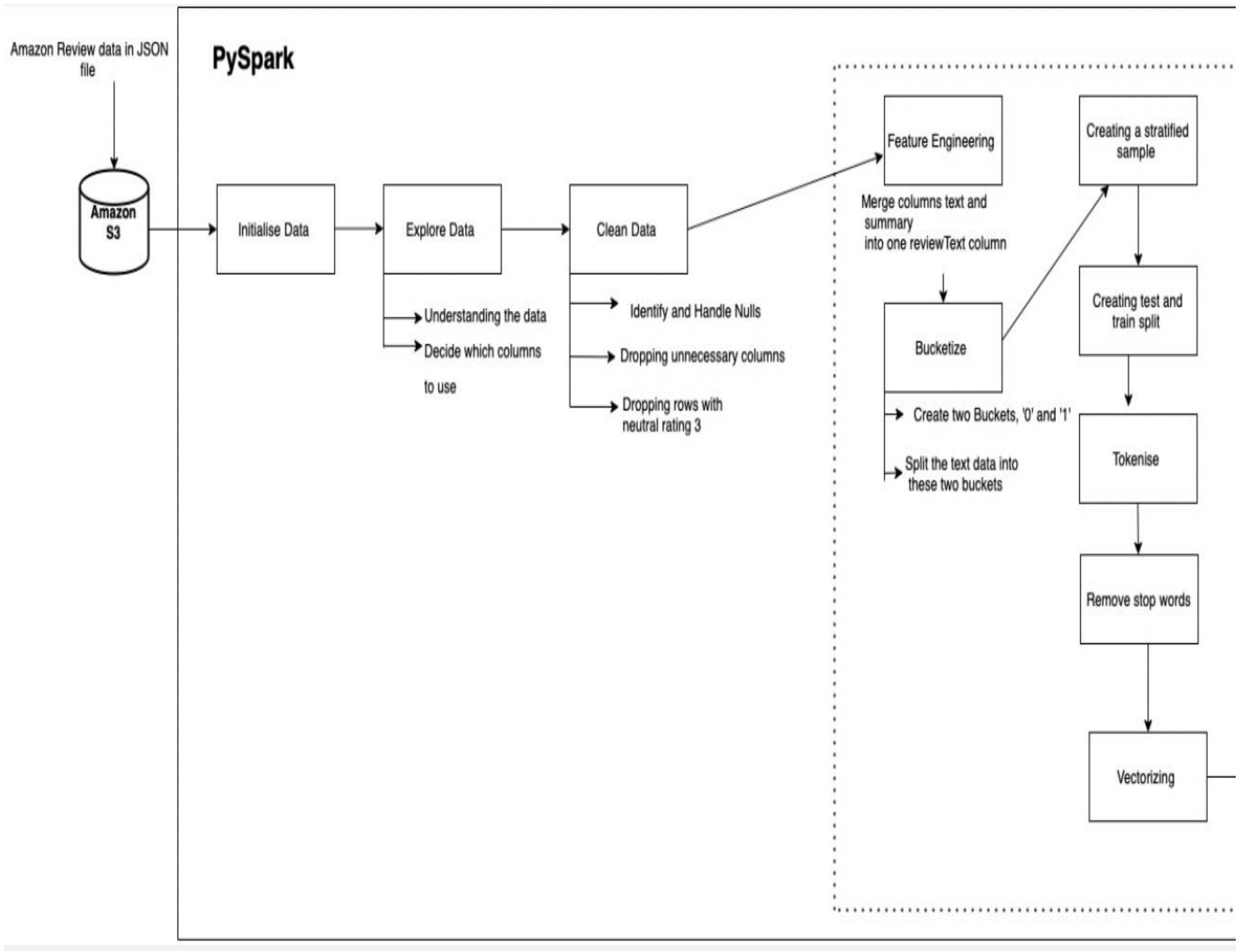

Amazon Review data in JSON file
PySpark
Amazon S3
Initialise Data
Explore Data
Clean Data
Understanding the data
Decide which columns to use
Identify and Handle Nulls
Dropping unnecessary columns
Dropping rows with neutral rating 3
Feature Engineering
Merge columns text and summary into one reviewText column
Bucketize
Create two Buckets, '0' and '1'
Split the text data into these two buckets
Creating a stratified sample
Creating test and train split
Tokenise
Remove stop words
Vectorizing

Fig 2: The project flowchart from loading data into Amazon S3 to data preprocessing step- Vectorizing

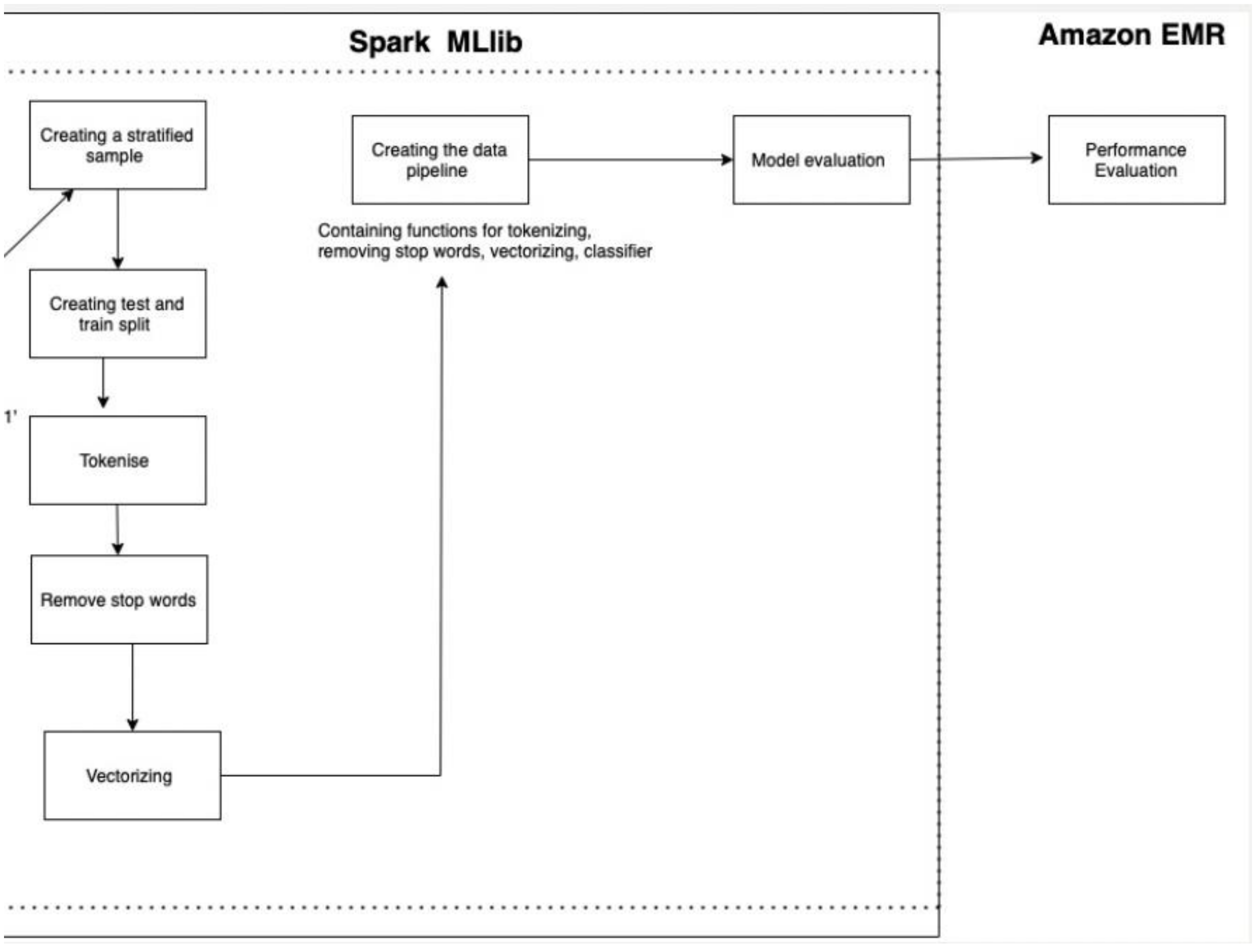

Spark MLlib
Amazon EMR
Creating a stratified sample
Creating the data pipeline
Model evaluation
Performance Evaluation
Containing functions for tokenizing, removing stop words, vectorizing, classifier
Creating test and train split
1'
Tokenise
Remove stop words
Vectorizing

Fig 3: The project flowchart from data preprocessing step-Vectorizing to running the Spark jobs on an Amazon EMR cluster

HARDWARE REQUIREMENTS TO EXECUTE SPARK JOBS

- Even though Spark performs much of its computation in memory, it still uses local disks, which do not fit in RAM, to store data and preserve intermediate output between stages.
- Having 4-8 disks per node configured without RAID is better.
- Spark runs well on memory ranging from 8 Gigabytes to hundreds of Gigabytes per machine. It is better to allocate at most 75% of the memory for Spark in all cases, and the rest should be assigned to the operating system and buffer cache.
- Numerous Spark applications are network bound when the data is found inside the memory. These applications can be made faster using a 10 Gigabit or higher network.
- Spark performs minimal sharing between threads; therefore, it scales well to tens of CPU cores per machine. At least 8-16 cores per machine must be provisioned.

TECHNOLOGIES USED

Amazon S3 :

Amazon Simple Storage Service (Amazon S3) is an object storage service that offers industry-leading scalability, data availability, security, and performance. A range of use cases can be handled by Amazon S3, such as data lakes, websites, mobile applications, backup and restore, archiving, enterprise applications, IoT devices, and big data analytics. Amazon S3 data can be managed by optimizing, organizing, and configuring access based on your organization's business and compliance needs.

For our project, our downloaded data files were in the JSON format. We performed the following steps for uploading the data file to an Amazon S3 bucket:

1. Create a bucket in Amazon S3.
    a. Sign into the AWS Management Console and open the Amazon S3 console
    b. Click Create Bucket.
    c. In the Bucket Name box of the Create a Bucket dialog box, type a bucket name. The bucket name you choose must be unique among all existing bucket names in Amazon S3. One way to help ensure uniqueness is to prefix your bucket names with the name of your organization. Bucket names must comply with certain rules.
    d. Select a Region. Create the bucket in the same Region as your cluster. If your cluster is in the US West (Oregon) Region, choose US West (Oregon) Region (us-west-2).
    e. Choose Create. When Amazon S3 successfully creates your bucket, the console displays your empty bucket in the Buckets panel.
2. Create a folder.
    a. Choose the name of the new bucket.
    b. Choose the Actions button and click Create Folder in the drop-down list.
    c. Name the new folder ‘load’

Apache Spark:

Apache Spark is one of the newer open-source, lightning-fast big data distributed processing platforms based on the same principles as Hadoop, which is developed in a manner to evolve the computational speed. Apache Spark is a distributed processing system used for big data workloads. It utilizes in memory caching, and optimized query execution for fast analytic queries against data of any size.

Spark Ecosystem consists of several components, including Spark Core and upper-level libraries such as Spark SQL, Spark Streaming, Spark Mllib, GraphX, and SparkR. Spark Core manages cluster managers such as Standalone, Hadoop YARN, Mesos, and Kubernetes.

The Spark components which are of utmost important in the context of this project are described as follows:

1. ***Spark Core***: All processes of Apache Spark are handled by the Spark Core.It provides a vast range of APIs as well as applications for programming languages such as Scala, Java, and Python APIs to facilitate the ease of development. In-memory computation is implemented in the Spark core to deliver speed and to solve the issue of MapReduce.
2. ***Spark SQL***: Spark SQL is a data processing framework in Apache Spark that is built on top of Spark Core. Both structured and semi-structured data can be accessed via Spark SQL. Spark SQL can read the data from different formats such as text, CSV, JSON, Avro, etc. It can create powerful, interactive, and analytical applications through both streaming and historical data.
3. ***Spark MLlib***: It is a package of Apache Spark which accommodates different types of machine learning algorithms (classification, regression, and clustering) on top of Spark. It performs data processing with large datasets to recognize the patterns and makes decisions based on them. Machine learning algorithms run with many iterations for the desired objective in an adaptable manner.

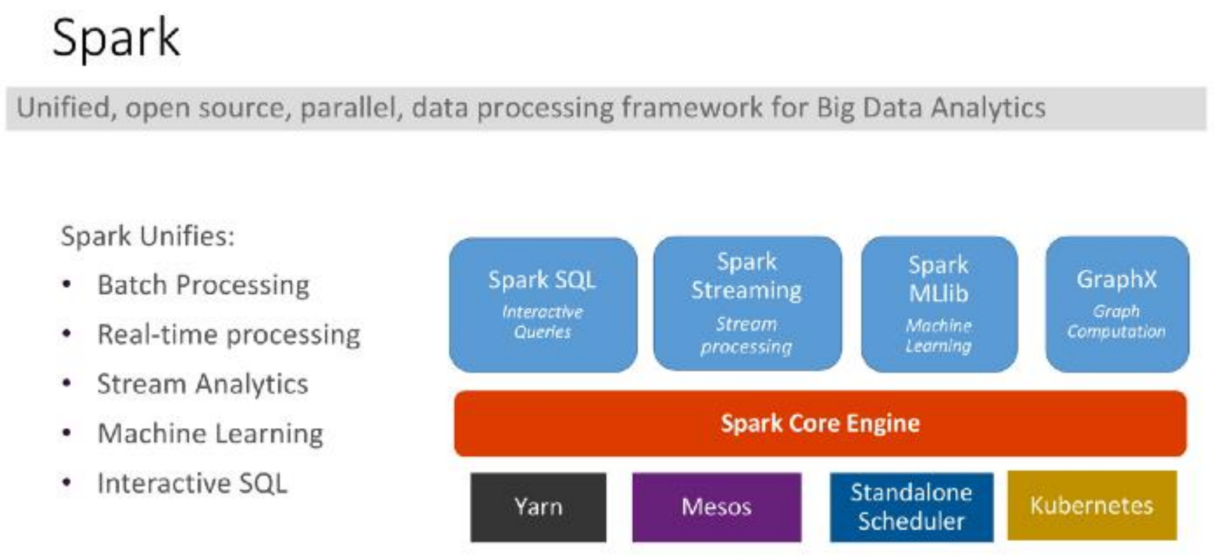


Fig 4: Components of Spark

Amazon EMR:

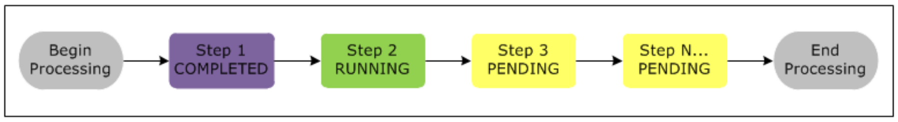
Amazon EMR (previously called Amazon Elastic MapReduce) is a managed cluster platform that simplifies running big data frameworks, such as Apache Spark and Apache Hadoop, on AWS to process and analyze vast amounts of data. Using these frameworks and related open-source projects, you can process data for analytics purposes and business intelligence workloads. Amazon EMR also lets you transform and move large amounts of data into and out of other AWS data stores and databases, such as Amazon Simple Storage Service (Amazon S3) and Amazon DynamoDB.

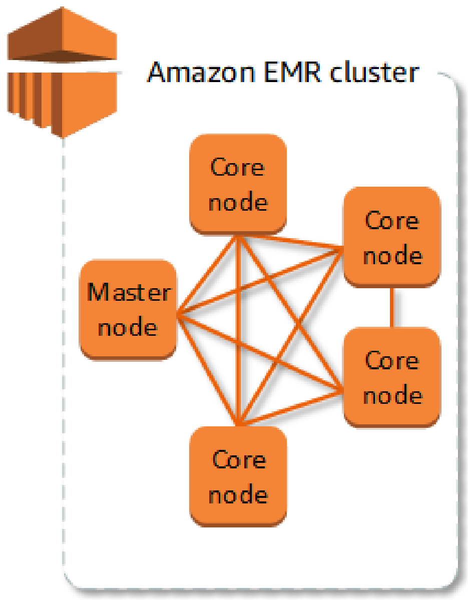
EMR Notebooks: An EMR notebook is a "serverless" notebook that you can use to run queries and code. Unlike a traditional notebook, the contents of an EMR notebook itself—the equations, queries, models, code, and narrative text within notebook cells—run in a client. The commands are executed using a kernel on the EMR cluster. Notebook contents are also saved to Amazon S3 separately from cluster data for durability and flexible re-use.

We used Amazon EMR Notebooks along with Amazon EMR clusters running Apache Spark to create and open Jupyter Notebook and JupyterLab interfaces within the Amazon EMR console.

In short, EMR allows us to store data in Amazon S3 and run compute as we need to process that data.

The following diagram represents the step sequence and change of state for the steps as they are processed.





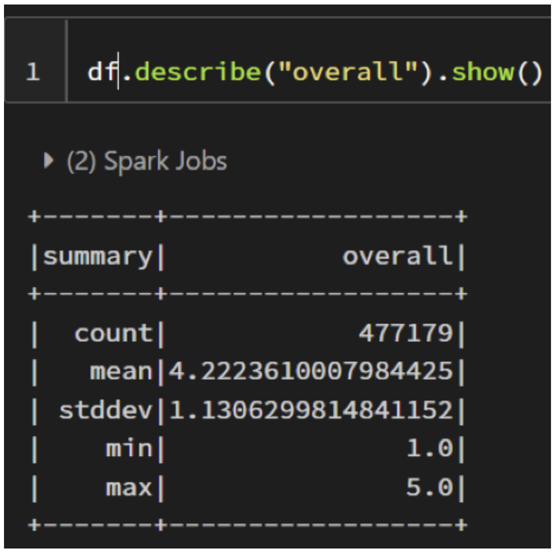
Using the steps mentioned above we ran our Pyspark code on EMR notebooks that are running on Spark Cluster / Pyspark Kernel. We set the file path to the URI / source location on S3 to fetch the datasets and then processed the same on the notebooks.

DATASET EXPLORATION

After setting up the Machine Learning Environment we proceeded to explore our dataset. Data Exploration allows for deeper understanding of a dataset, making it easier to navigate and use the data.

The DataFrame.describe() Pyspark function is used to calculate basic statistics such as count, mean, stddev, min, and max. for the numeric column 'overall'. In statistics the mean, mode and the median are called the "Central Tendency."

The mean is considered the most reliable measure of central tendency for making assumptions about a population from a single sample[6]. Therefore, by observing the mean we can conclude that most product ratings (displayed in the column 'overall') are positive.

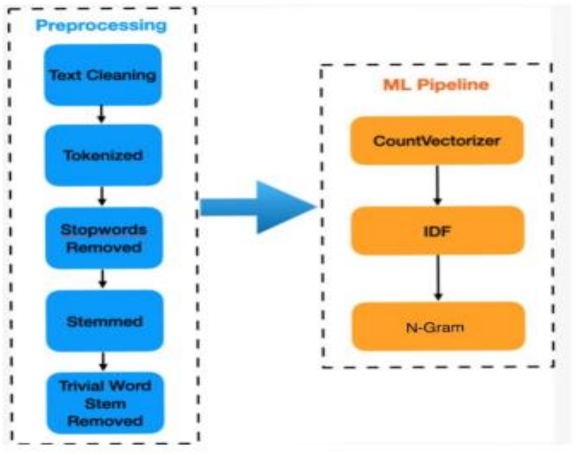


```
df.describe("overall").show()
```

```
+-------+------------------+
|summary|           overall|
+-------+------------------+
|  count|            477179|
|   mean|4.2223610007984425|
| stddev|1.1306299814841152|
|    min|               1.0|
|    max|               5.0|
+-------+------------------+
```

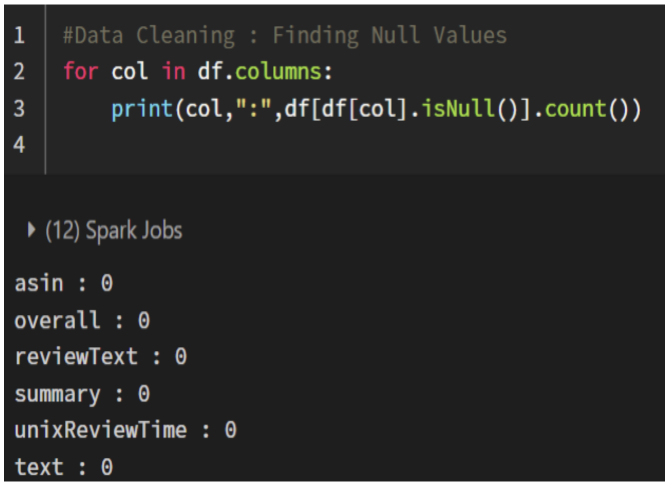
Fig 5: Basic statistical values of column 'overall'

DATA PREPROCESSING



Data Cleaning

Identifying and Removing Null /NA values: Handling Null/NA values is important because most of the machine learning algorithms don't support data with these values. These values in the data are to be handled properly. If not, it leads to drawing inaccurate inferences. Therefore we looked to identify and eliminate NA/Null values, but there were no such values found in the dataset.



```
#Data Cleaning : Finding Null Values
for col in df.columns:
    print(col,":",df[df[col].isNull()].count())

```

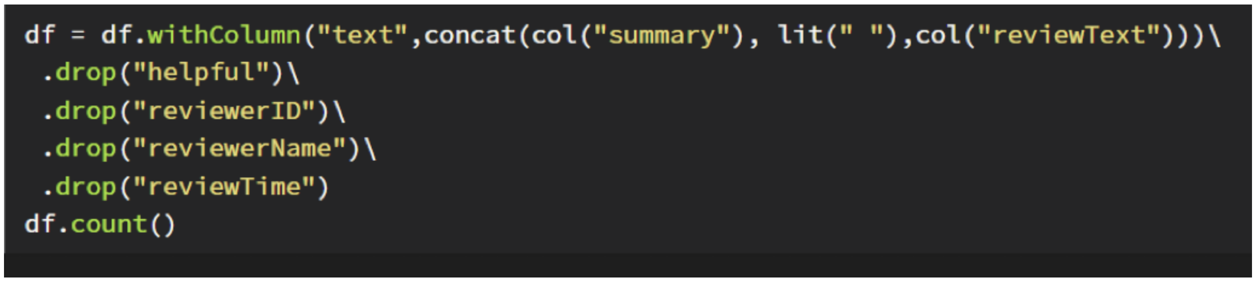
```
(12) Spark Jobs
asin : 0
overall : 0
reviewText : 0
summary : 0
unixReviewTime : 0
text : 0
```

Fig 6: Checking for Null values in the dataset

FEATURING ENGINEERING

Feature engineering is formulating useful features from existing data following the target to be learned and the machine learning model used. It involves transforming data to forms that better relate to the underlying target to be learned. In layman terms, it is the addition, deletion, combination, mutation — of your dataset to improve machine learning model training, leading to better performance and greater accuracy.

- Deleting unnecessary columns: Since we are performing sentiment analysis on product review data, the columns ‘summary’ , ‘reviewText’ and ‘overall’ are the only columns of use to fulfill the objective of the project. The remaining columns in the dataset are deleted.
- As a part of this step the two columns with relevant data for sentiment analysis- ‘reviewText’ and ‘summary’ are concatenated into one column called ‘text.’



```
df = df.withColumn("text",concat(col("summary"), lit(" "),col("reviewText")))\
 .drop("helpful")\
 .drop("reviewerID")\
 .drop("reviewerName")\
 .drop("reviewTime")
df.count()
```

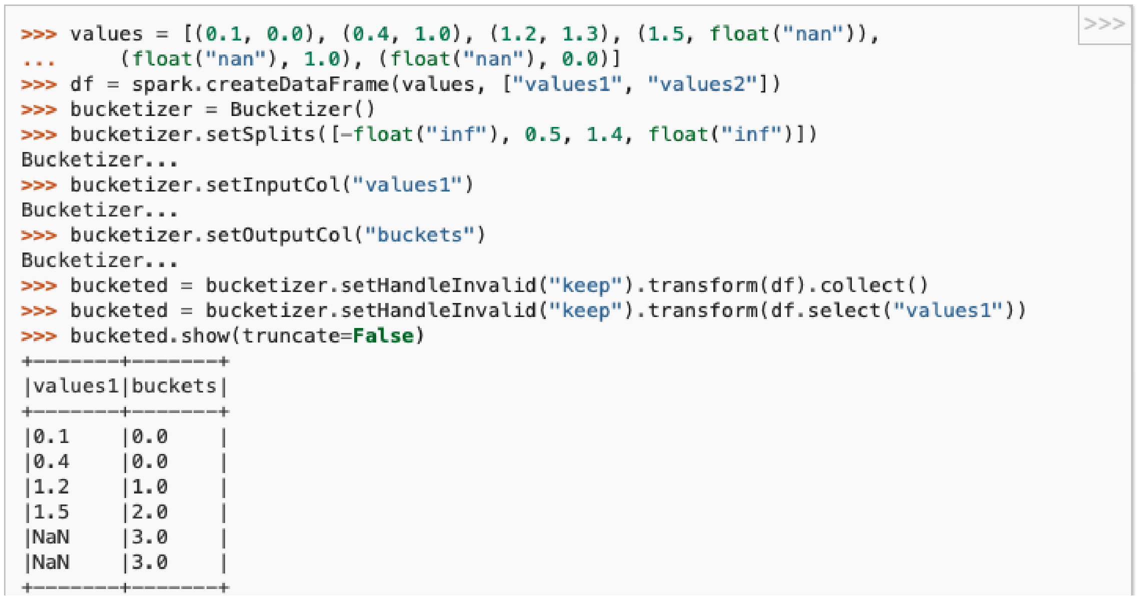
Fig 7: Concatenating columns ‘summary’ and ‘reviewText’ into column ’text’ while also dropping other irrelevant columns

BUCKETIZING

The Bucketizer() functions available in the Spark MLlib, map a column of continuous features to a column of feature buckets. In simple terms it assigns a bucket value (string) to a numerical input (column), based on the array boundaries provided.

The dataset we collected from amazon reviews on beauty and lifestyle did not come with any predefined labels. Bucketizer helps us set the labels on the dataset by choosing 0 which stands for negative sentiment , 1 for positive sentiment which are assigned as labels in the process.

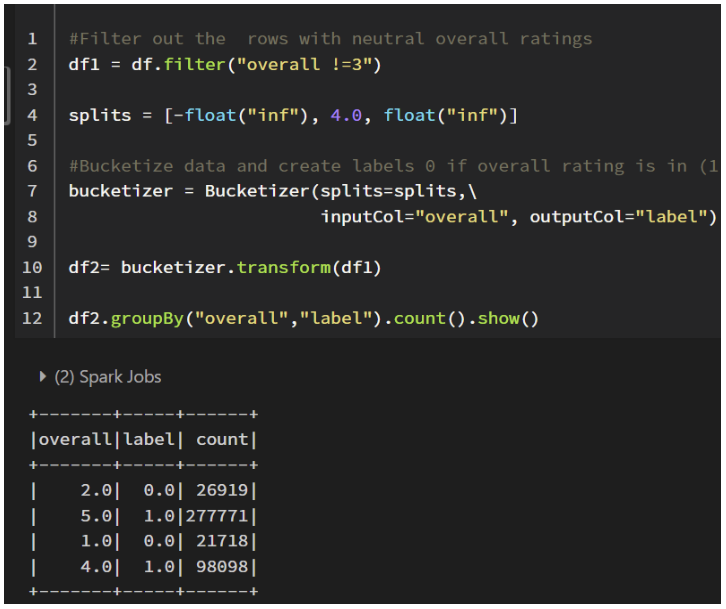
Given below is a simple example to illustrate the same:



```
>>> values = [(0.1, 0.0), (0.4, 1.0), (1.2, 1.3), (1.5, float("nan")),
...     (float("nan"), 1.0), (float("nan"), 0.0)]
>>> df = spark.createDataFrame(values, ["values1", "values2"])
>>> bucketizer = Bucketizer()
>>> bucketizer.setSplits([-float("inf"), 0.5, 1.4, float("inf")])
Bucketizer...
>>> bucketizer.setInputCol("values1")
Bucketizer...
>>> bucketizer.setOutputCol("buckets")
Bucketizer...
>>> bucketed = bucketizer.setHandleInvalid("keep").transform(df).collect()
>>> bucketed = bucketizer.setHandleInvalid("keep").transform(df.select("values1"))
>>> bucketed.show(truncate=False)
+-------+-------+
|values1|buckets|
+-------+-------+
|0.1    |0.0    |
|0.4    |0.0    |
|1.2    |1.0    |
|1.5    |2.0    |
|NaN    |3.0    |
|NaN    |3.0    |
+-------+-------+
```

Fig 8: Example to illustrate Bucketization

In the above example, all continuous ‘values’ in between -infinity and 0.5 are put in bucket 0, while between 0.5 and 1.4 are put in bucket 1 and so on. Similarly, we create ‘labels’, ‘0’: For overall ratings 1&2 and ‘1’: For overall ratings 4&5



```
#Filter out the  rows with neutral overall ratings
df1 = df.filter("overall !=3")

splits = [-float("inf"), 4.0, float("inf")]

#Bucketize data and create labels 0 if overall rating is in (1
bucketizer = Bucketizer(splits=splits,\
                        inputCol="overall", outputCol="label")

df2= bucketizer.transform(df1)

df2.groupBy("overall","label").count().show()
```

```
(2) Spark Jobs
+-------+-----+------+
|overall|label| count|
+-------+-----+------+
|    2.0|  0.0| 26919|
|    5.0|  1.0|277771|
|    1.0|  0.0| 21718|
|    4.0|  1.0| 98098|
+-------+-----+------+
```

Fig 9: Using Bucketizer() to label ‘overall’ values as either ‘0’ or ‘1’

All ‘text’ with ‘overall’ value 3 were filtered out before this step was performed.

SAMPLING

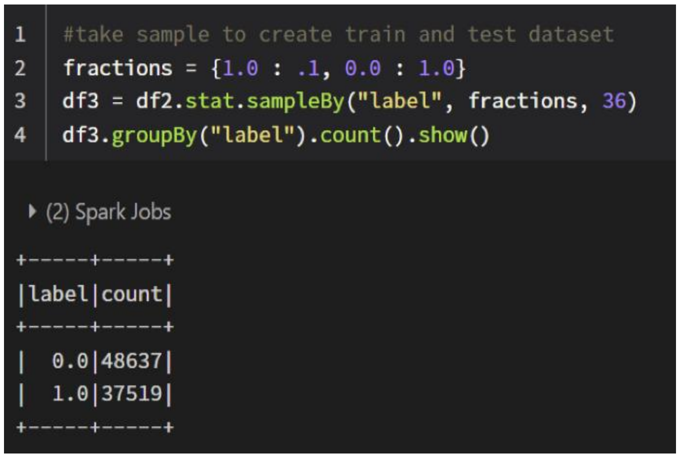
A classification dataset with skewed class proportions is called imbalanced. Classes that make up a large proportion of the data set are called majority classes. Those that make up a smaller proportion are a minority class. As already discussed in the previous sections of this report, a large part of the chosen dataset is mostly positive (with mean ‘overall’ value 4.0) and so we have an imbalanced dataset at hand.

The problem with training the model with an imbalanced dataset is that the model will be biased towards the majority class only. This causes a problem when we are interested in the prediction of the minority class. For example, for a Cancer dataset problem, predictions of having cancer are more important over not having cancer.

Machine learning techniques will fail or give misleadingly optimistic performance on such datasets. Data sampling provides a collection of techniques that transform a training dataset in order to balance or better balance the class distribution. Once balanced, standard machine learning algorithms can be trained directly on the transformed dataset without any modification. This allows the challenge of imbalanced classification, even with severely imbalanced class distributions, to be addressed with a data preparation method.

For the purpose of this project, we have used the ‘stratified sampling without replacement’ method to create our sample which will be further used to create the training and testing datasets. It is done by dividing the dataset into subgroups or strata, and an almost equal number of instances are sampled from each stratum; where once an observation is selected, it cannot be selected again.

In PySpark, dataframe.stat.sampleBy() returns a stratified sample without replacement based on the fraction given on each stratum. Therefore, the resulting sample has almost equal number of observations belonging to each class/'label'



```
#take sample to create train and test dataset
fractions = {1.0 : .1, 0.0 : 1.0}
df3 = df2.stat.sampleBy("label", fractions, 36)
df3.groupBy("label").count().show()
```

```
+-----+-----+
|label|count|
+-----+-----+
|  0.0|48637|
|  1.0|37519|
+-----+-----+
```

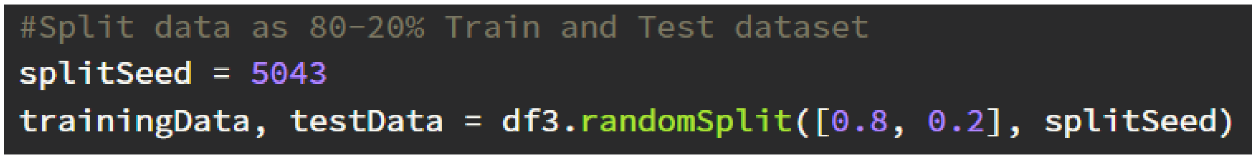
Fig 10: Creating a stratified sample

CREATING THE TRAIN-TEST SPLIT

Training and Testing split is used to train the model and testing subset to validate the coefficient. Train test split is a model validation procedure that allows you to simulate how a model would perform on new/unseen data.

- Arrange: Make sure your data is arranged into a format acceptable for train test split. This consists of separating your full data set into "Features" and "Target."
- Split The Data : Split the data set into two pieces — a training set and a testing set. This consists of random sampling without replacement about 80 percent of the rows and putting them into your training set. The remaining 25 percent is put into your test.
- Train the Model : Train the model on the training set.
- Test the Model: Test the model on the testing set and evaluate the performance.

Created the 80-20 training and test set split from the sample dataset to train and test the model.



```
#Split data as 80-20% Train and Test dataset
splitSeed = 5043
trainingData, testData = df3.randomSplit([0.8, 0.2], splitSeed)
```

Fig 11: Creating the Test-Train Split

TOKENIZING:

Tokenization is a simple process that takes raw data and converts it into a useful data string. While tokenization is well known for its use in cybersecurity and in the creation of NFTs, tokenization is also an important part of the NLP process. Tokenization is used in natural language processing to split paragraphs and sentences into smaller units that can be more easily assigned meaning.

The first step of the NLP process is gathering the data (a sentence) and breaking it into understandable parts (words).. This step determines the vocabulary of the dataset (set of unique tokens in the dataset).

Here's an example of a string of data:

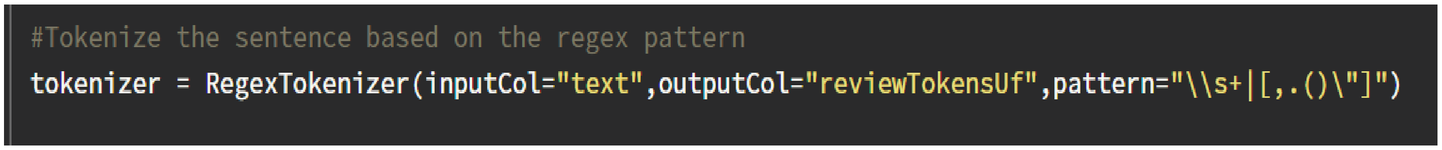
"What restaurants are nearby?"

In order for this sentence to be understood by a machine, tokenization is performed on the string to break it into individual parts. With tokenization, we'd get something like this:

'what' 'restaurants' 'are' 'nearby' .

There are types of Tokenization Word/sentence, character and sub word in which we have implemented word tokenization using Regextokenizer() function.

The Regextokenizer() function in Pyspark converts the input string into lowercase, divides it by whitespaces, and does not tokenize any non-character. An index value is assigned to each token



```
#Tokenize the sentence based on the regex pattern
tokenizer = RegexTokenizer(inputCol="text",outputCol="reviewTokensUf",pattern="\\s+|[,.()\"]")
```

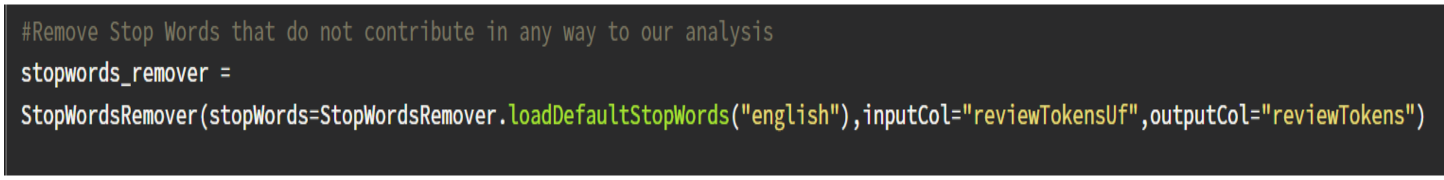
Fig 12: Using RegexTokenizer() to divide 'text' into smaller sub- texts

REMOVING STOPWORDS

Stop words are words which should be excluded from the input, typically because the words appear frequently and don't carry as much meaning.

Examples of stop words in English are "a", "the", "is", "are" and etc.

StopWordsRemover takes as input a sequence of strings (e.g. the output of a Tokenizer) and drops all the stop words from the input sequences. The list of stopwords is specified by the stopWords parameter. Default stop words for some languages are accessible by calling StopWordsRemover.loadDefaultStopWords(language), for which available options are "danish", "dutch", "english", "finnish", "french", "german", "hungarian", "italian", "norwegian", "portuguese", "russian", "spanish", "swedish" and "turkish". A boolean parameter caseSensitive indicates if the matches should be case sensitive (false by default).



```
#Remove Stop Words that do not contribute in any way to our analysis
stopwords_remover =
StopWordsRemover(stopWords=StopWordsRemover.loadDefaultStopWords("english"),inputCol="reviewTokensUf",outputCol="reviewTokens")
```

Fig 13: Using StopWordsRemover() to remove stopwords

- Remove the stop words using the function StopWordsRemover as per the English dictionary.
- A StopWordsRemover() function in MLLib is a feature transformer that filters out stop words from

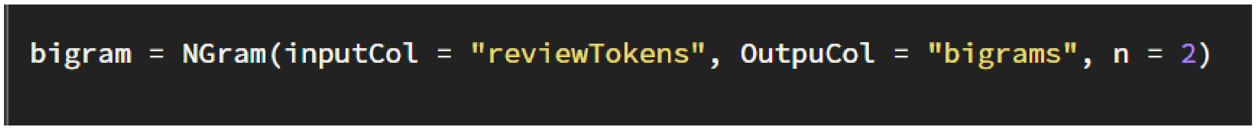
GENERATING N-GRAMS:

SL.No.	Type of n-gram	Generated n-grams
1	Unigram	[“I”,”reside”,”in”, “Bengaluru”]
2	Bigram	[“I reside”,”reside in”,”in Bengaluru”]
3	Trigram	[“I reside in”, “reside in Bengaluru”]

A feature transformer that converts the input array of strings into an array of n-grams.

It returns an array of n-grams where each n-gram is represented by a space-separated string of words. When the input is empty, an empty array is returned. The above example illustrates this concept.

This technique is used to capture combinations of words that affect the sentiment of the document



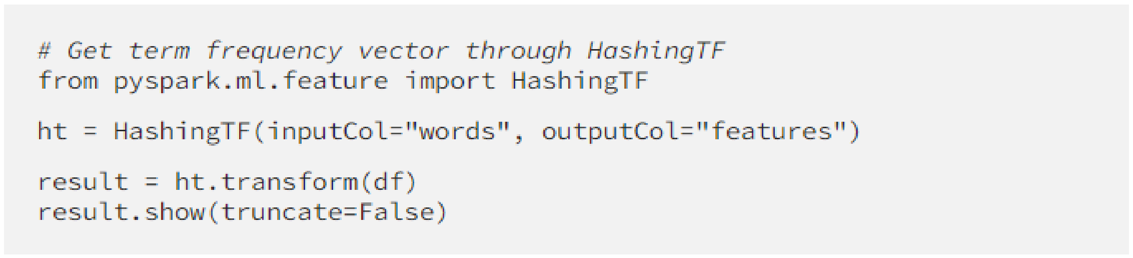
```
bigram = NGram(inputCol = "reviewTokens", OutpuCol = "bigrams", n = 2)
```

Fig 14: Using NGarm() to generate bigrams

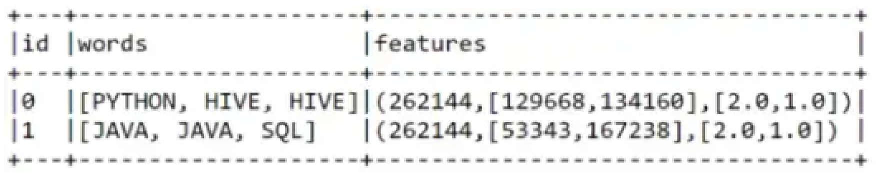
In this project we have considered Bigrams which take two words at a time as an input where n = 2.

HASHING: It maps a sequence of terms to their term frequencies using the hashing trick. CountVectorizer and HashingTF estimators are used to generate term frequency vectors. They basically convert documents into a numerical representation which can be fed directly or with further processing into other algorithms like LDA, MinHash for Jaccard Distance, Cosine Distance to name a few.

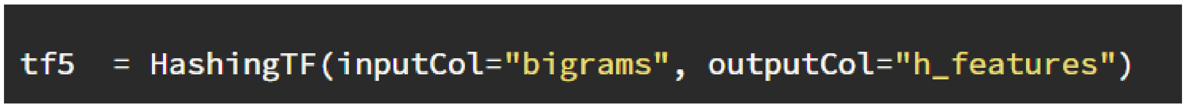
HashingTF converts documents to vectors of fixed size. The default feature dimension is 262,144. The terms are mapped to indices using a Hash Function. The hash function used is MurmurHash 3. The term frequencies are computed with respect to the mapped indices.



```
# Get term frequency vector through HashingTF
from pyspark.ml.feature import HashingTF

ht = HashingTF(inputCol="words", outputCol="features")

result = ht.transform(df)
result.show(truncate=False)
```



```
+---+--------------------+--------------------------------+
|id |words               |features                        |
+---+--------------------+--------------------------------+
|0  |[PYTHON, HIVE, HIVE]|(262144,[129668,134160],[2.0,1.0])|
|1  |[JAVA, JAVA, SQL]   |(262144,[53343,167238],[2.0,1.0]) |
+---+--------------------+--------------------------------+
```



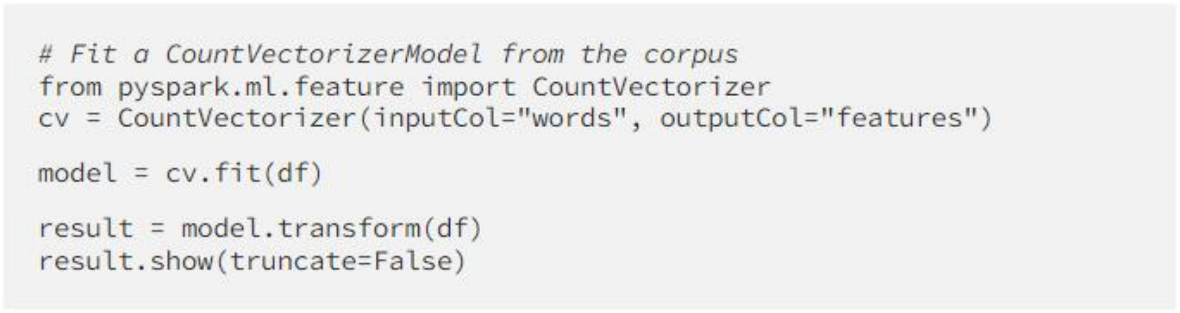
```
tf5  = HashingTF(inputCol="bigrams", outputCol="h_features")
```

VECTORIZING:

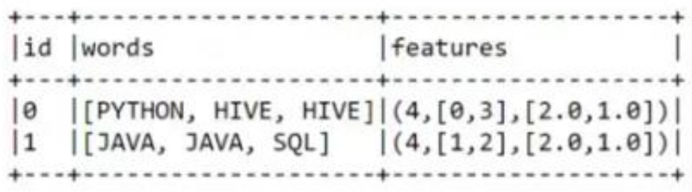
CountVectorizer

Once the text is split into n-grams, it is turned into numerical vectors that only ML algorithms can process.

In simple terms, Vectorization is defining a numerical measure to characterize the text that was tokenized in the previous step. CountVectoriser() converts text documents to vectors which gives information of token counts.



```
# Fit a CountVectorizerModel from the corpus
from pyspark.ml.feature import CountVectorizer
cv = CountVectorizer(inputCol="words", outputCol="features")

model = cv.fit(df)

result = model.transform(df)
result.show(truncate=False)
```



```
+---+--------------------+-------------------+
|id |words               |features           |
+---+--------------------+-------------------+
|0  |[PYTHON, HIVE, HIVE]|(4,[0,3],[2.0,1.0])|
|1  |[JAVA, JAVA, SQL]   |(4,[1,2],[2.0,1.0])|
+---+--------------------+-------------------+
```

Fig16 : Example to show the working of CountVectorizer()

We vectorize the n-grams using the TF-IDF method, once indexes have been assigned. TF-IDF stands for Term Frequency -Inverse Document Frequency
In the case of our project we already created the term frequency vectors using CountVectorizer(). We proceed to find the Inverse Document Frequency using the MLlib function IDF().
The standard formulation is used i.e idf = log((m + 1) / (d(t) + 1)), where m is the total number of documents and d(t) is the number of documents that contain the term t.

This implementation supports filtering out terms which do not appear in a minimum number of documents.

The TF-IDF technique is further explained as follows :

It is a method to convert documents into vectors such that the vector reflects the importance of a term to a document in the corpus.if we denote a term by t, a document by d, and the corpus by D.

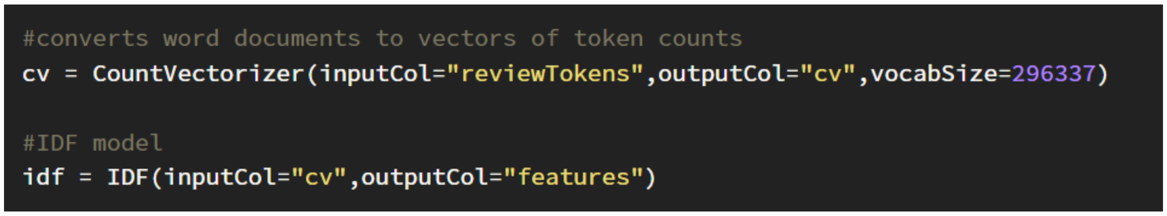
Term Frequency (TF(t,d)): Number of times term t appears in document d.

Document Frequency (DF(t,D)): Number of documents that contain the term t.

IDF(t,D): Numerical measure of how much information a terms provide

$$IDF(t,D) = \log \frac{|D|+1}{DF(t,D)+1}, \quad TFIDF(t,d,D) = TF(t,d) \cdot IDF(t,D).$$

TF-IDF therefore ensures that terms with high frequency in the document will have high TF but if a term has high frequency across the corpus then its importance is reduced by IDF. A term present in all documents in the corpus will have TF-IDF equal to 0. Both HashingTF and CountVectorizer can be used to generate the term frequency vector



```
#converts word documents to vectors of token counts
cv = CountVectorizer(inputCol="reviewTokens",outputCol="cv",vocabSize=296337)

#IDF model
idf = IDF(inputCol="cv",outputCol="features")
```

Fig 17: Creating the TF-IDF model

For example:

Consider three documents

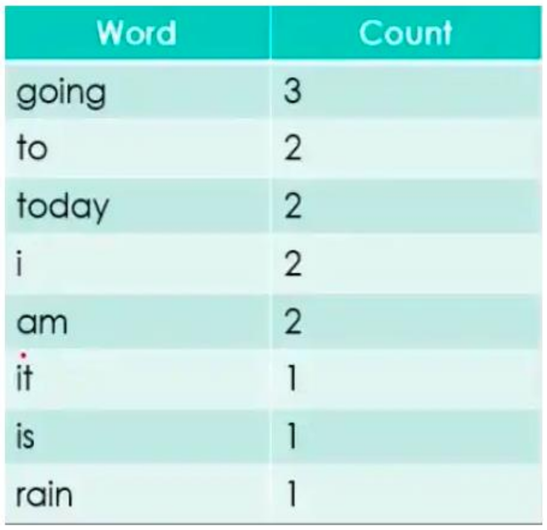
Document 1 It is going to rain today.

Document 2 Today I am not going outside.

Document 3 I am going to watch the season premiere.

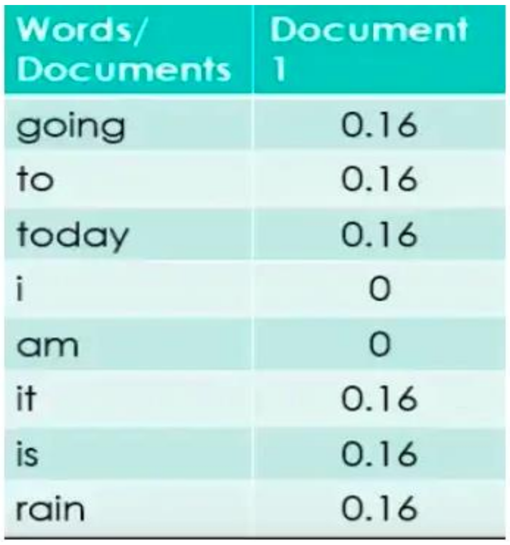
After we clean and tokenize the data, and find the vocabulary from all three documents we proceed to find the term frequency of Document 1 using

TF = (Number of repetitions of word in a document) / (# of words in a document)



Word	Count
going	3
to	2
today	2
i	2
am	2
it	1
is	1
rain	1

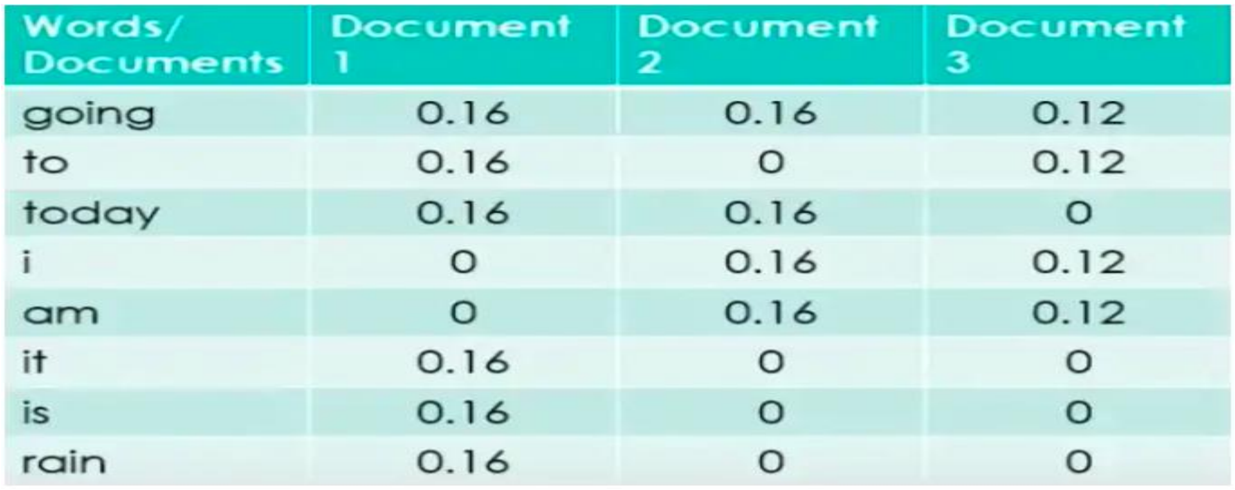
Fig 18: Vocab of documents



Words/ Documents	Document 1
going	0.16
to	0.16
today	0.16
i	0
am	0
it	0.16
is	0.16
rain	0.16

Fig 19: Term frequency of document1

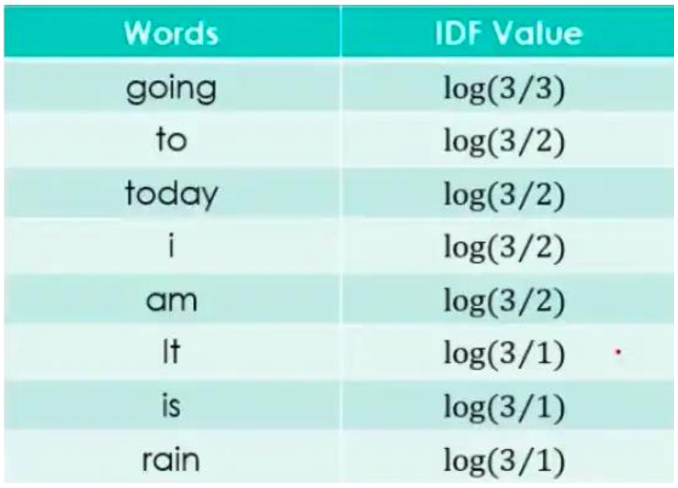
Continuing the same step for the rest of the sentences:



Words/ Documents	Document 1	Document 2	Document 3
going	0.16	0.16	0.12
to	0.16	0	0.12
today	0.16	0.16	0
i	0	0.16	0.12
am	0	0.16	0.12
it	0.16	0	0
is	0.16	0	0
rain	0.16	0	0

Fig 20: Term frequencies for each document for the vocabulary

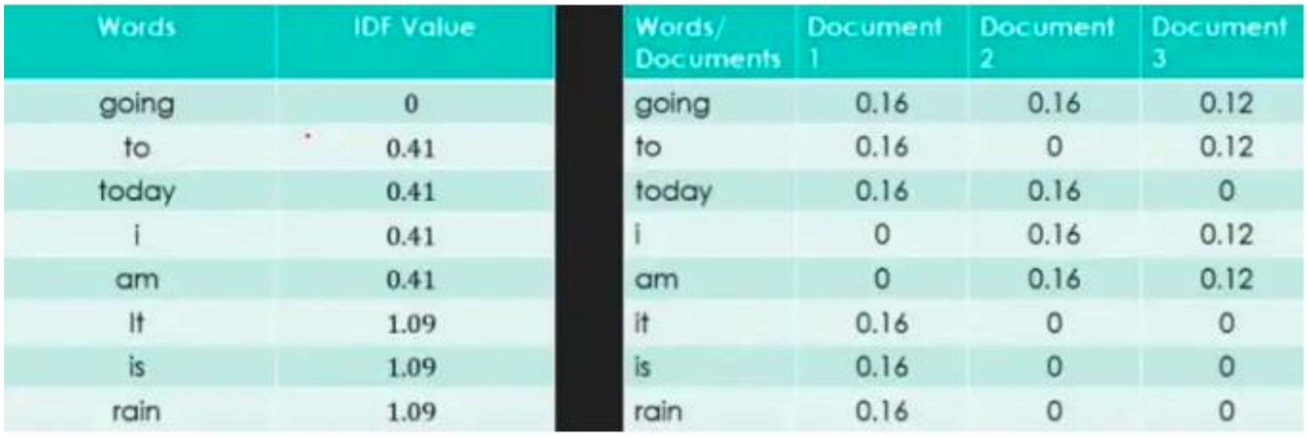
Now we calculate the IDF for the vocab. The results of the calculation are illustrated in the following figure



Words	IDF Value
going	log(3/3)
to	log(3/2)
today	log(3/2)
i	log(3/2)
am	log(3/2)
It	log(3/1)
is	log(3/1)
rain	log(3/1)

Fig 21: IDF for the vocabulary

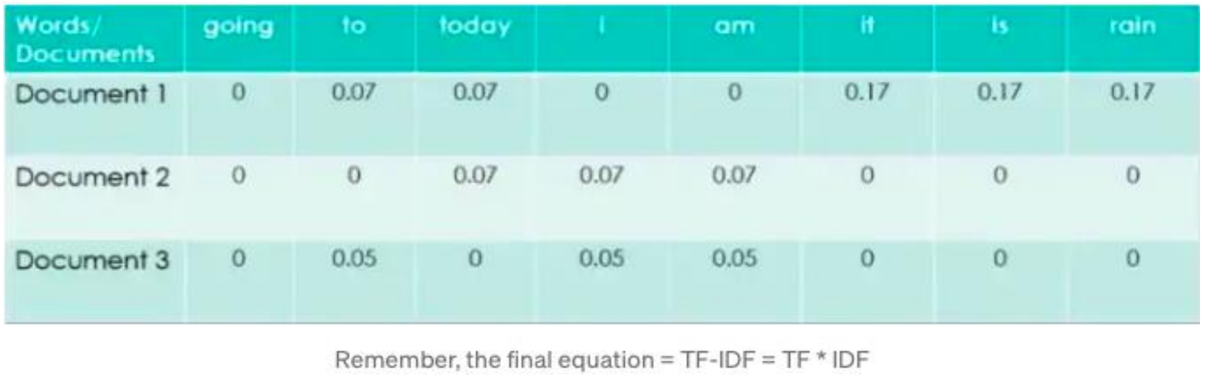
Finally we build the model i.e stack all the words next to each other and compare the results. We us the table in Fig 23 to ask questions



Words	IDF Value
going	0
to	0.41
today	0.41
i	0.41
am	0.41
It	1.09
is	1.09
rain	1.09

Words/ Documents	Document 1	Document 2	Document 3
going	0.16	0.16	0.12
to	0.16	0	0.12
today	0.16	0.16	0
i	0	0.16	0.12
am	0	0.16	0.12
it	0.16	0	0
is	0.16	0	0
rain	0.16	0	0

Fig 22: TF and IDF value of all three documents



Words/ Documents	going	to	today	i	am	it	is	rain
Document 1	0	0.07	0.07	0	0	0.17	0.17	0.17
Document 2	0	0	0.07	0.07	0.07	0	0	0
Document 3	0	0.05	0	0.05	0.05	0	0	0

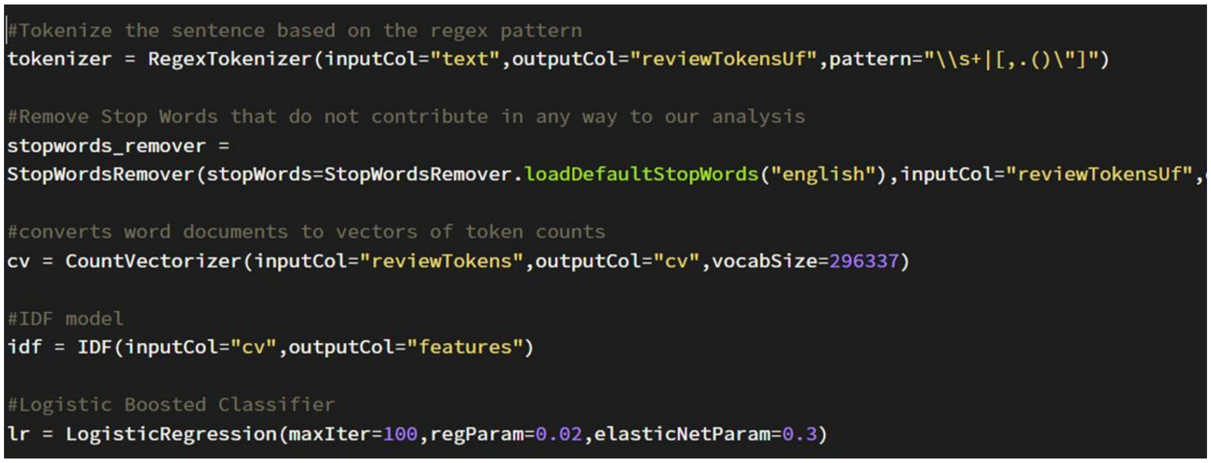
Remember, the final equation = TF-IDF = TF * IDF

Fig 23: Comparing TF-IDF results of the three documents

You can easily see using this table that words like **'it','is','rain'** are important for document 1 but not for document 2 and document 3 which means Document 1 and 2 & 3 are different w.r.t talking about rain. You can also say that Document 1 and 2 talk about something **'today'**, and document 2 and 3 discuss something about the writer because of the word **'I'.** This table helps you find similarities and non similarities btw documents, words and more much much better than BOW.

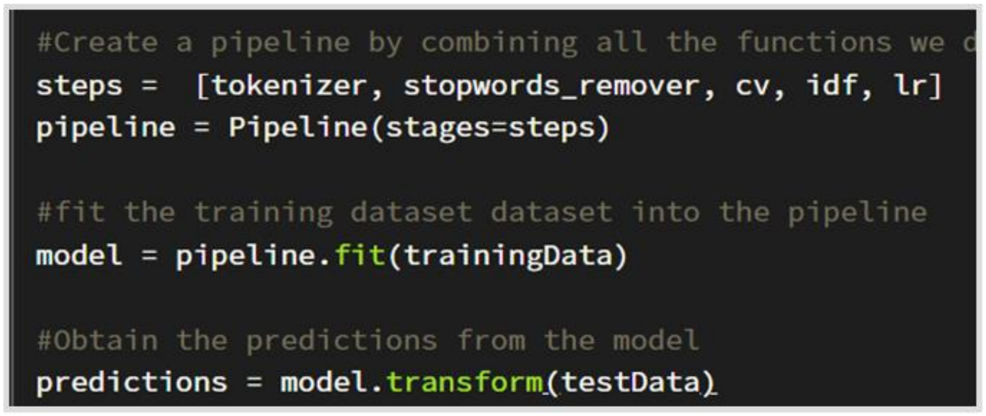
PIPELINES:

When you call NLP on a text or voice, it converts the whole data into strings, and then the prime string undergoes multiple steps (the process called processing pipeline.) It uses trained pipelines to supervise your input data and reconstruct the whole string depending on voice tone or sentence length.
For each pipeline, the component returns to the main string. Then passes on to the next components. The capabilities and efficiencies depend upon the components, their models, and training.



```
#Tokenize the sentence based on the regex pattern
tokenizer = RegexTokenizer(inputCol="text",outputCol="reviewTokensUf",pattern="\\s+|[,.()\"]")

#Remove Stop Words that do not contribute in any way to our analysis
stopwords_remover =
StopWordsRemover(stopWords=StopWordsRemover.loadDefaultStopWords("english"),inputCol="reviewTokensUf",

#converts word documents to vectors of token counts
cv = CountVectorizer(inputCol="reviewTokens",outputCol="cv",vocabSize=296337)

#IDF model
idf = IDF(inputCol="cv",outputCol="features")

#Logistic Boosted Classifier
lr = LogisticRegression(maxIter=100,regParam=0.02,elasticNetParam=0.3)
```

Fig 24: Functions that are a part of the data pipeline



```
#Create a pipeline by combining all the functions we d
steps =  [tokenizer, stopwords_remover, cv, idf, lr]
pipeline = Pipeline(stages=steps)

#fit the training dataset dataset into the pipeline
model = pipeline.fit(trainingData)

#Obtain the predictions from the model
predictions = model.transform(testData)
```

Fig 25: Building the data pipeline

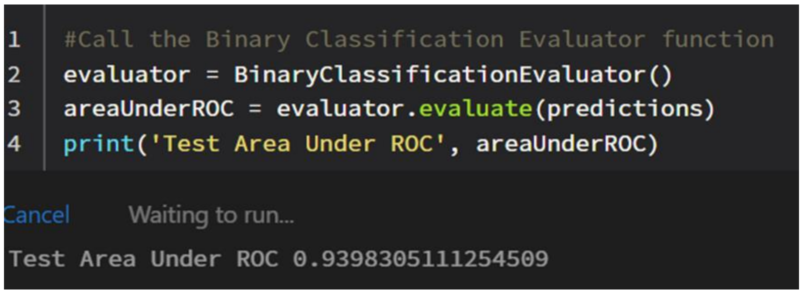
This is an example of a pipeline where we assign each step in the data preprocessing stage and the model to be fit as stages and pass it as a parameter into the pipeline. Pipeline in sparks is an estimator and transformer function where we fit the training data first and later transform the test data to evaluate the model.

MODEL EVALUATION AND METRICS

BinaryClassificationEvaluator :

- Evaluator for binary classification, which expects input columns rawPrediction, label and an optional weight column.
- The rawPrediction column can be of type double (binary 0/1 prediction, or probability of label 1) or of type vector (length-2 vector of raw predictions, scores, or label probabilities).

- We use this BinaryClassificationEvaluator function on predictions to determine the Area Under RoC and other metrics such as accuracy_score, precision, f1_score, recall, and so on.



```
#Call the Binary Classification Evaluator function
evaluator = BinaryClassificationEvaluator()
areaUnderROC = evaluator.evaluate(predictions)
print('Test Area Under ROC', areaUnderROC)
```

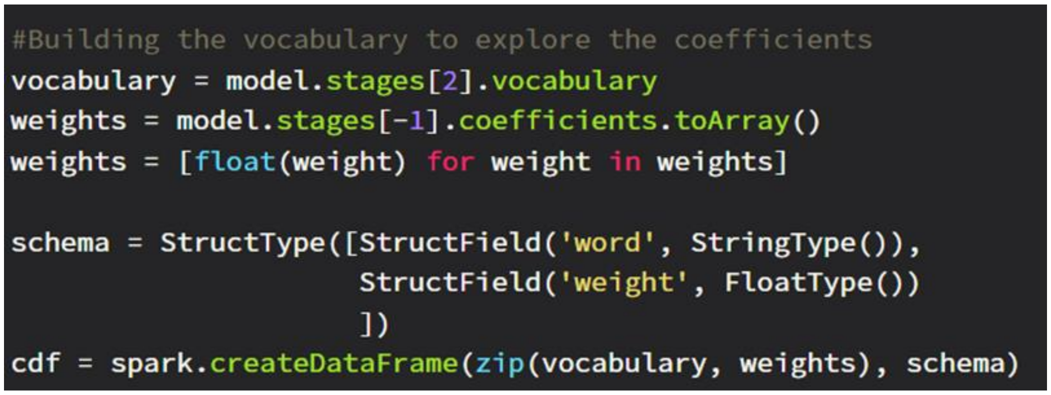
```
Test Area Under ROC 0.9398305111254509
```

Fig 25: calling the classifier

<u>VOCABULARY DATAFRAME</u>:

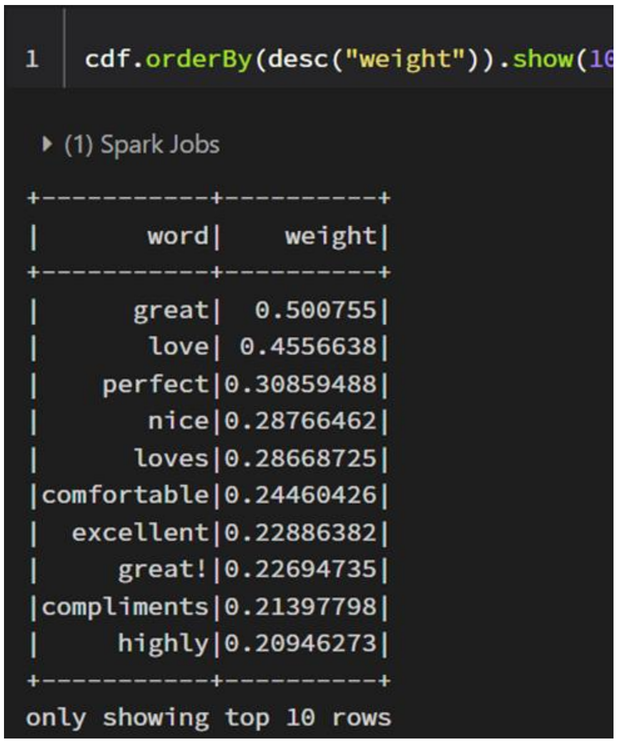
We built a dataframe of words in the vocabulary and their corresponding weights to explore the weights and impact on the target variable.

The words that have a positive coefficient or weight contribute to the positive sentiment and negative coefficients will have a adverbial score of the target variable.



```
#Building the vocabulary to explore the coefficients
vocabulary = model.stages[2].vocabulary
weights = model.stages[-1].coefficients.toArray()
weights = [float(weight) for weight in weights]

schema = StructType([StructField('word', StringType()),
                     StructField('weight', FloatType())
                     ])
cdf = spark.createDataFrame(zip(vocabulary, weights), schema)
```

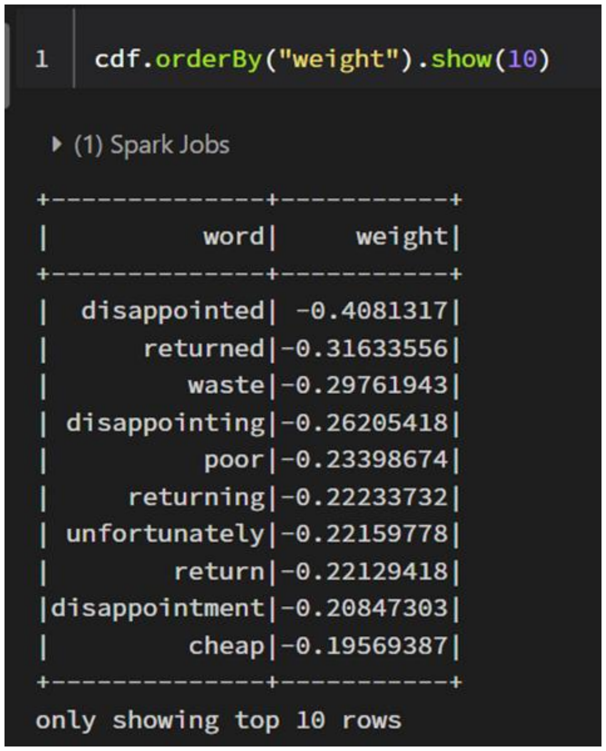
Fig 26: Building the vocabulary



```
cdf.orderBy(desc("weight")).show(1
```

▸ (1) Spark Jobs

```
+-----------+----------+
|       word|    weight|
+-----------+----------+
|      great|  0.500755|
|       love| 0.4556638|
|    perfect|0.30859488|
|       nice|0.28766462|
|      loves|0.28668725|
|comfortable|0.24460426|
|  excellent|0.22886382|
|     great!|0.22694735|
|compliments|0.21397798|
|     highly|0.20946273|
+-----------+----------+
only showing top 10 rows
```



```
cdf.orderBy("weight").show(10)
```

▸ (1) Spark Jobs

```
+-------------+-----------+
|         word|     weight|
+-------------+-----------+
|  disappointed| -0.4081317|
|     returned|-0.31633556|
|        waste|-0.29761943|
| disappointing|-0.26205418|
|         poor|-0.23398674|
|    returning|-0.22233732|
| unfortunately|-0.22159778|
|       return|-0.22129418|
|disappointment|-0.20847303|
|        cheap|-0.19569387|
+-------------+-----------+
only showing top 10 rows
```

Fig 27: Positive words with their weights Fig 28:Negative words with their weights

Important metrics used to assess the performance of each model are:

Precision: is defined as the number of true positives divided by the number of true positives plus the number of false positives as indicated below.

$$\tau_\pi \mid (\tau_\pi + \phi_\nu)$$

Accuracy: is the fraction of predictions our model got right. Formally, accuracy has the following definition:

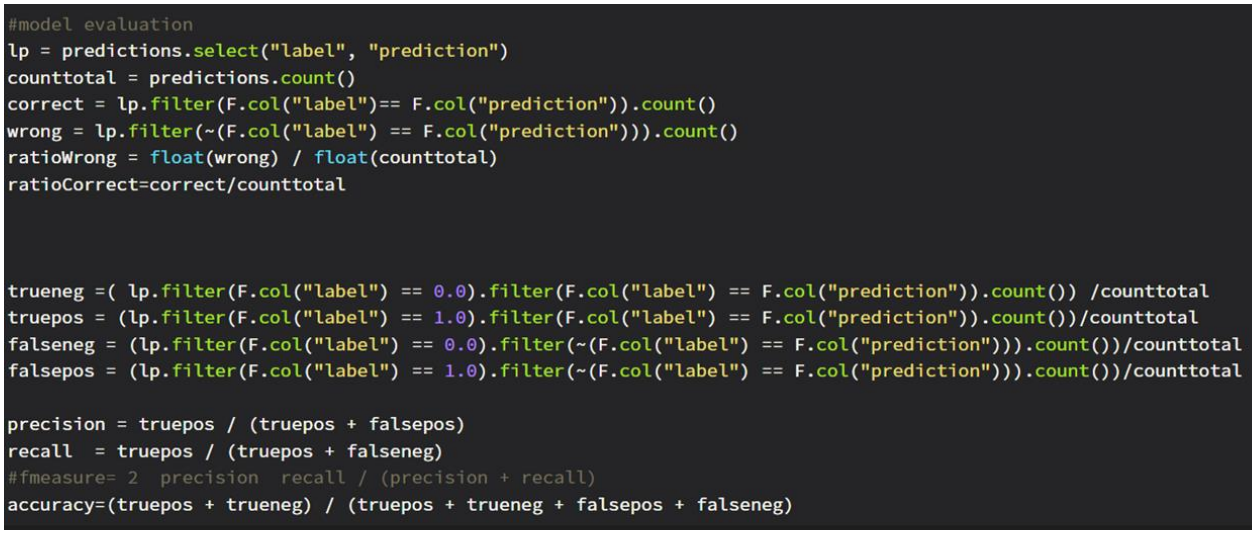
where TP = True positive; FP = False positive, TN = True Negative, FN = False Negative

$$\text{Accuracy} = \frac{TP + TN}{TP + TN + FP + FN}$$

AUC: **AUC** stands for "Area under the ROC Curve." That is, AUC measures the entire two-dimensional area underneath the entire ROC curve from (0,0) to (1,1).
AUC provides an aggregate measure of performance across all possible classification thresholds. One way of interpreting AUC is as the probability that the model ranks a random positive example more highly than a random negative

Logistic Regression; implementation and Results:

Logistic regression is a popular method to predict a categorical response. It is a special case of Generalized Linear models that predicts the probability of the outcomes. In spark.ml logistic regression can be used to predict a binary outcome by using binomial logistic regression, or it can be used to predict a multiclass outcome by using multinomial logistic regression. Use the family parameter to select between these two algorithms, or leave it unset and Spark will infer the correct variant.

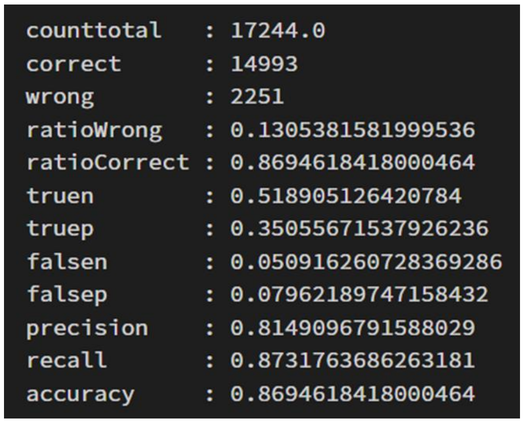


```
#model evaluation
lp = predictions.select("label", "prediction")
counttotal = predictions.count()
correct = lp.filter(F.col("label")== F.col("prediction")).count()
wrong = lp.filter(~(F.col("label") == F.col("prediction"))).count()
ratioWrong = float(wrong) / float(counttotal)
ratioCorrect=correct/counttotal



trueneg =( lp.filter(F.col("label") == 0.0).filter(F.col("label") == F.col("prediction")).count()) /counttotal
truepos = (lp.filter(F.col("label") == 1.0).filter(F.col("label") == F.col("prediction")).count())/counttotal
falseneg = (lp.filter(F.col("label") == 0.0).filter(~(F.col("label") == F.col("prediction"))).count())/counttotal
falsepos = (lp.filter(F.col("label") == 1.0).filter(~(F.col("label") == F.col("prediction"))).count())/counttotal

precision = truepos / (truepos + falsepos)
recall  = truepos / (truepos + falseneg)
#fmeasure= 2  precision  recall / (precision + recall)
accuracy=(truepos + trueneg) / (truepos + trueneg + falsepos + falseneg)
```

We calculated the Accuracy Score, Precision and Recall to measure the performance of logistic Regression.

Pipeline built with steps = [tokenizer, stopwords_remover, cv, idf, lr]

Accuracy is 87% , Precision = 81%, AUC is 93%.



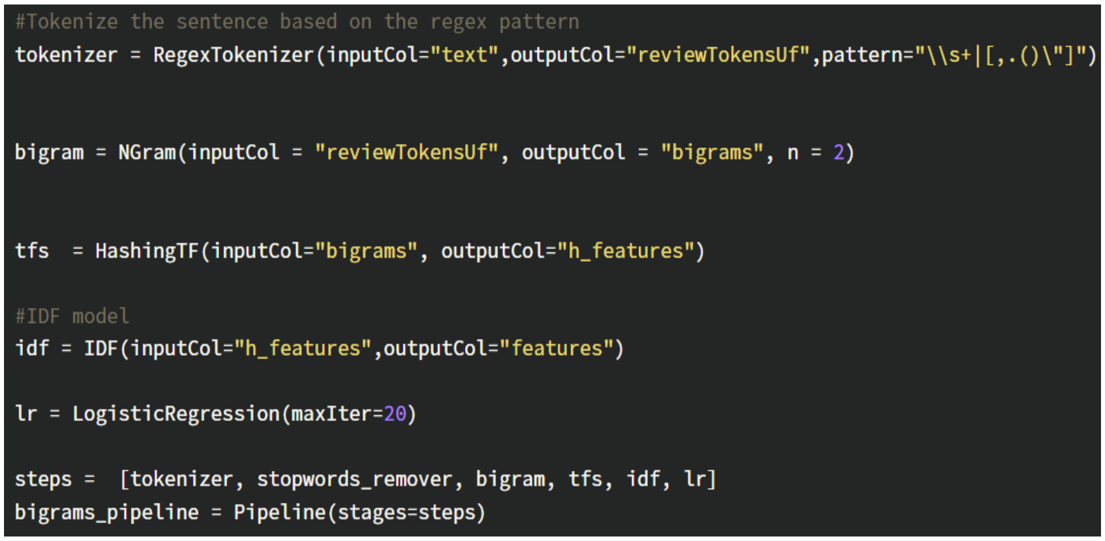
```
counttotal   : 17244.0
correct      : 14993
wrong        : 2251
ratioWrong   : 0.1305381581999536
ratioCorrect : 0.8694618418000464
truen        : 0.518905126420784
truep        : 0.35055671537926236
falsen       : 0.050916260728369286
falsep       : 0.07962189747158432
precision    : 0.8149096791588029
recall       : 0.8731763686263181
accuracy     : 0.8694618418000464
```

Fig 29: Performance metrics of Logistic Regression

Logistic Regression with Bigrams; implementation and Results :

We calculated the Accuracy Score, Precision, Auc to measure the performance of logistic Regression with Bigrams, HashingTF, IDF and logistic Regression.

Pipeline used with steps = [tokenizer, bigrams, tfs, idf, lr]


```
#Tokenize the sentence based on the regex pattern
tokenizer = RegexTokenizer(inputCol="text",outputCol="reviewTokensUf",pattern="\\s+|[,.()\"]")


bigram = NGram(inputCol = "reviewTokensUf", outputCol = "bigrams", n = 2)


tfs  = HashingTF(inputCol="bigrams", outputCol="h_features")

#IDF model
idf = IDF(inputCol="h_features",outputCol="features")

lr = LogisticRegression(maxIter=20)

steps =  [tokenizer, stopwords_remover, bigram, tfs, idf, lr]
bigrams_pipeline = Pipeline(stages=steps)
```

Accuracy is 89% , Precision = 89%, AUC is 94.7%.

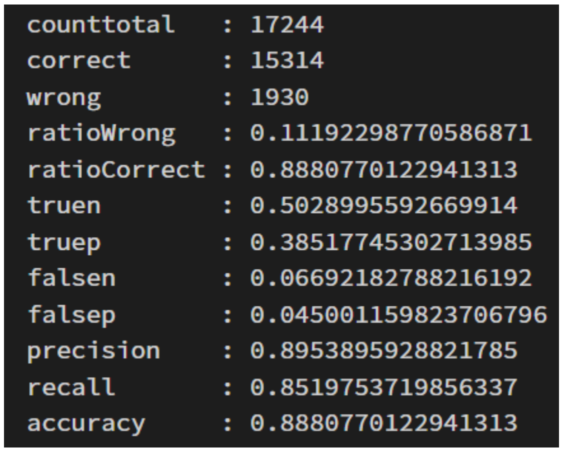

```
counttotal   : 17244
correct      : 15314
wrong        : 1930
ratioWrong   : 0.11192298770586871
ratioCorrect : 0.8880770122941313
truen        : 0.5028995592669914
truep        : 0.38517745302713985
falsen       : 0.06692182788216192
falsep       : 0.045001159823706796
precision    : 0.8953895928821785
recall       : 0.8519753719856337
accuracy     : 0.8880770122941313
```

Fig 30: Performance metrics of Logistic Regression with bigrams

Random Forest; Implementation and Results :

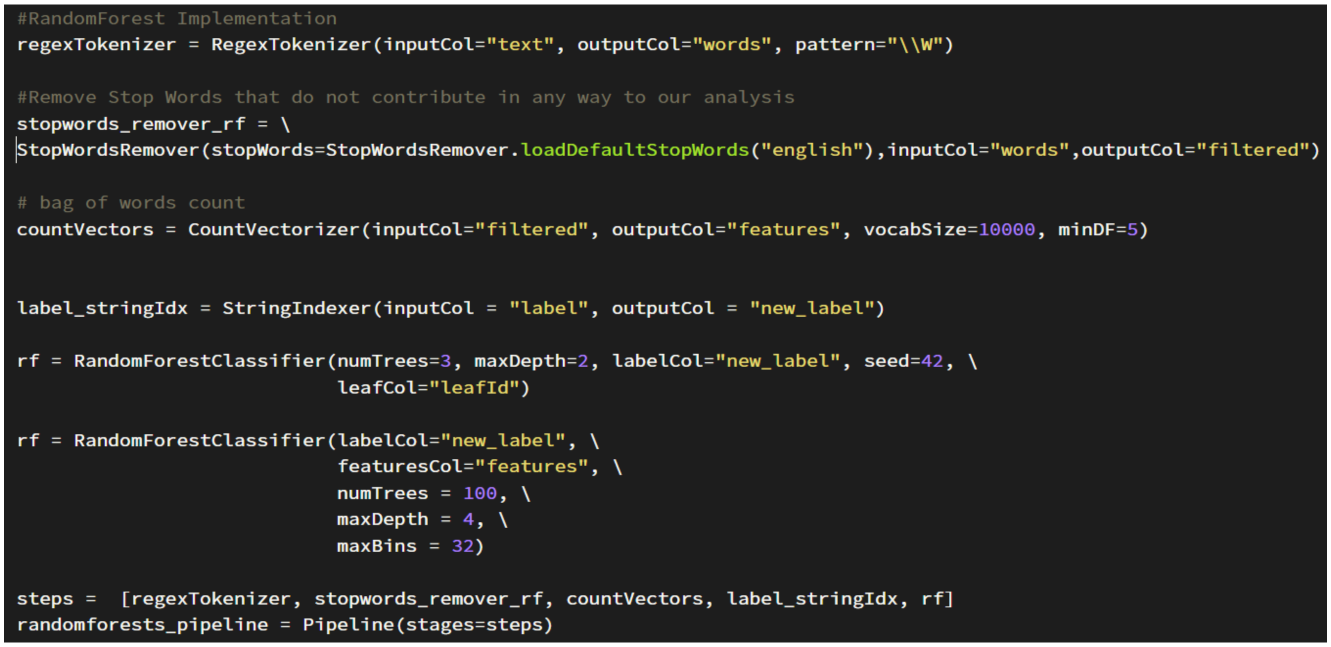
Random forests or random decision forests is an ensemble learning method for classification, regression and other tasks that operates by constructing a multitude of decision trees at training time. For classification tasks, the output of the random forest is the class selected by most trees.

Random forests generally outperform decision trees, but their accuracy is lower than gradient boosted trees. However, data characteristics can affect their performance.

We calculated the Accuracy Score, Precision, AUC to measure the performance of the Random Forest Classifier.

Pipeline used with steps = [tokenizer, stopwordsremover, CountVectors, rf]

Accuracy is 58% , Precision = 2% for positive as major class , AUC is 94.7%.



```
#RandomForest Implementation
regexTokenizer = RegexTokenizer(inputCol="text", outputCol="words", pattern="\\W")

#Remove Stop Words that do not contribute in any way to our analysis
stopwords_remover_rf = \
StopWordsRemover(stopWords=StopWordsRemover.loadDefaultStopWords("english"),inputCol="words",outputCol="filtered")

# bag of words count
countVectors = CountVectorizer(inputCol="filtered", outputCol="features", vocabSize=10000, minDF=5)


label_stringIdx = StringIndexer(inputCol = "label", outputCol = "new_label")

rf = RandomForestClassifier(numTrees=3, maxDepth=2, labelCol="new_label", seed=42, \
                            leafCol="leafId")

rf = RandomForestClassifier(labelCol="new_label", \
                            featuresCol="features", \
                            numTrees = 100, \
                            maxDepth = 4, \
                            maxBins = 32)

steps =  [regexTokenizer, stopwords_remover_rf, countVectors, label_stringIdx, rf]
randomforests_pipeline = Pipeline(stages=steps)
```

Fig 31: Performance metrics of Logistic Regression with bigrams

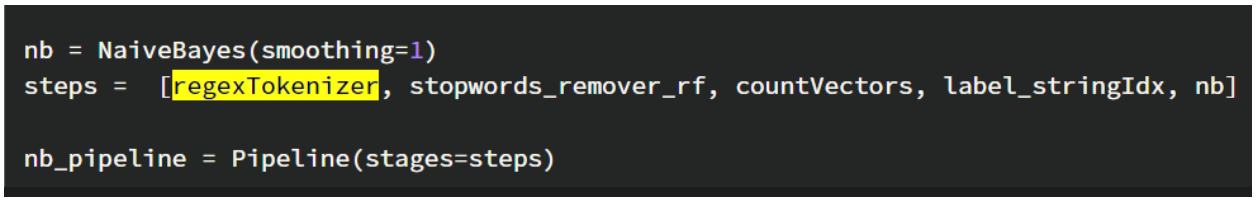
Naïve Bayes Classifier; Implementation and Results :

A Naive Bayes classifier is a probabilistic machine learning model that’s used for classification tasks. The crux of the classifier is based on the Bayes theorem.

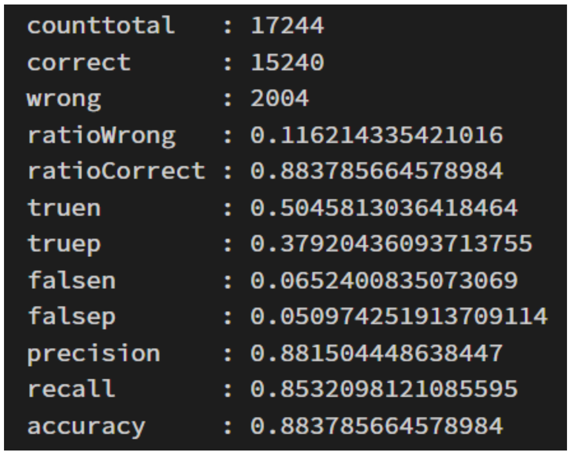
Using Bayes theorem, we can find the probability of A happening, given that B has occurred. Here, B is the evidence and A is the hypothesis. The assumption made here is that the predictors/features are independent. That is, the presence of one particular feature does not affect the other. Hence it is called naive.

Naive Bayes algorithms are mostly used in sentiment analysis, spam filtering, recommendation systems etc. They are fast and easy to implement but their biggest disadvantage is that the requirement of predictors to be independent. In most of the real life cases, the predictors are dependent, this hinders the performance of the classifier.

$$P(A|B) = \frac{P(B|A)P(A)}{P(B)}$$



```
nb = NaiveBayes(smoothing=1)
steps =  [regexTokenizer, stopwords_remover_rf, countVectors, label_stringIdx, nb]

nb_pipeline = Pipeline(stages=steps)
```

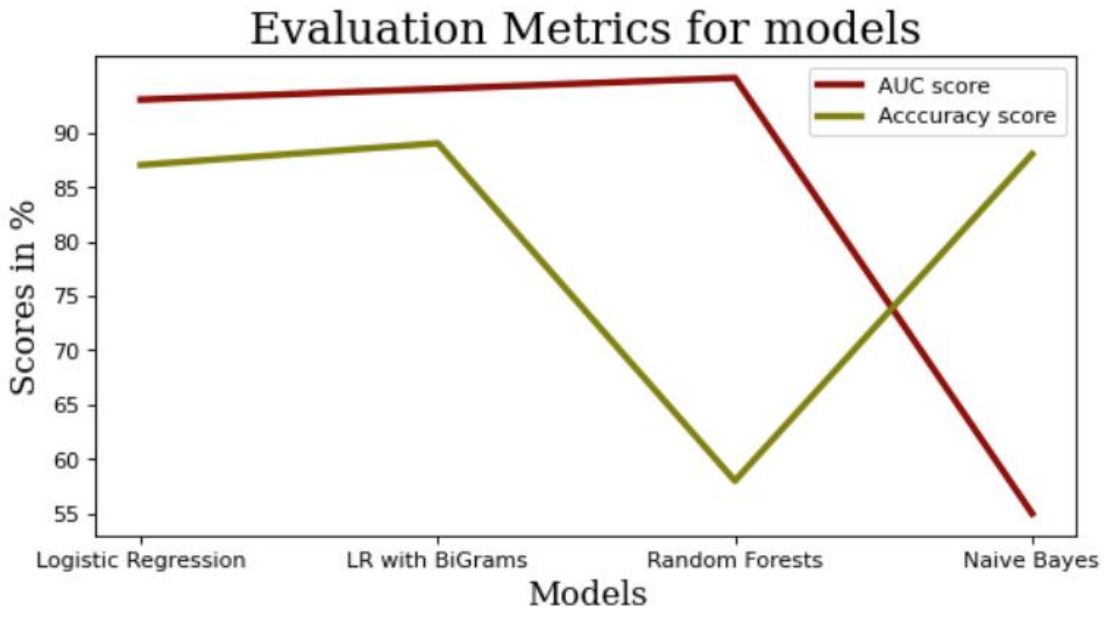


```
counttotal   : 17244
correct      : 15240
wrong        : 2004
ratioWrong   : 0.116214335421016
ratioCorrect : 0.883785664578984
truen        : 0.5045813036418464
truep        : 0.37920436093713755
falsen       : 0.0652400835073069
falsep       : 0.050974251913709114
precision    : 0.881504448638447
recall       : 0.8532098121085595
accuracy     : 0.883785664578984
```

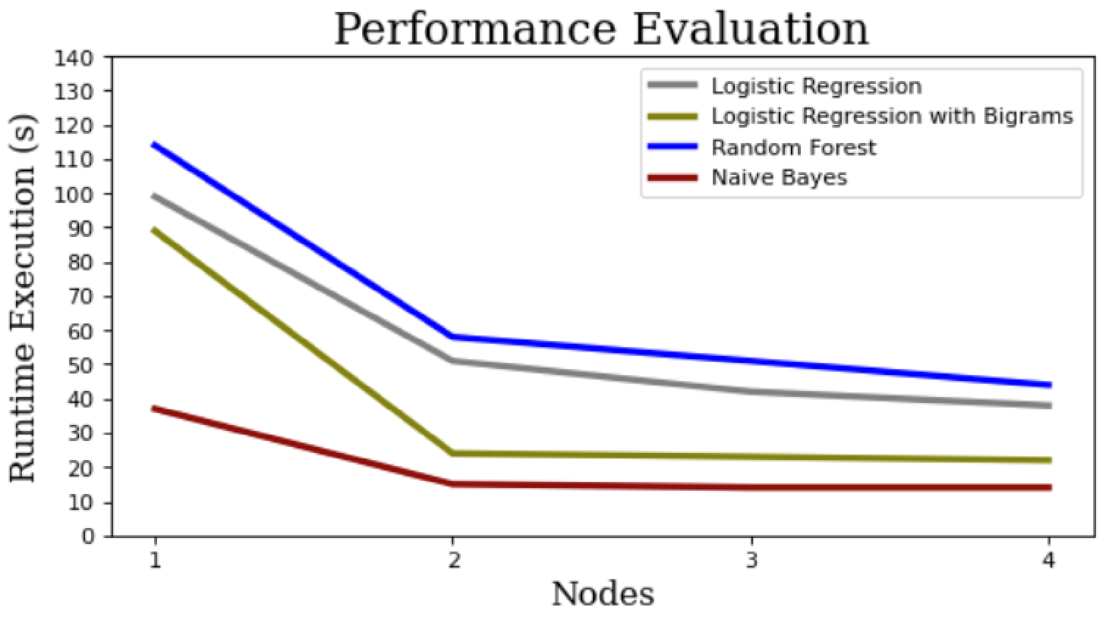
Fig 32: Performance metrics of Logistic Regression with bigrams

Accuracy is 88.3% , Precision = 88%, AUC is 54%.

EVALUATION METRICS COMPARING AUC AND ACCURACY SCORES



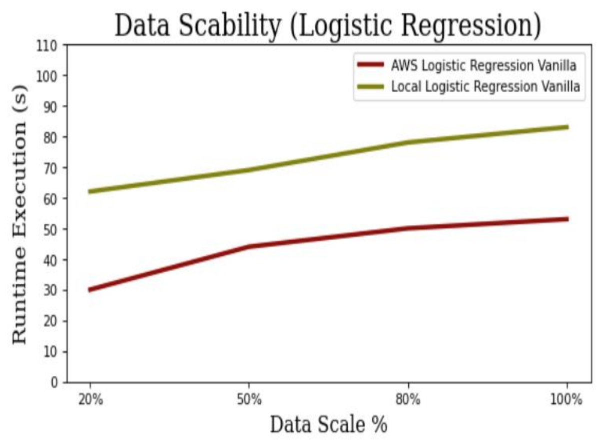
PERFORMANCE EVALUATION BY VARYING THE NUMBER OF NODES ON AMAZON EMR

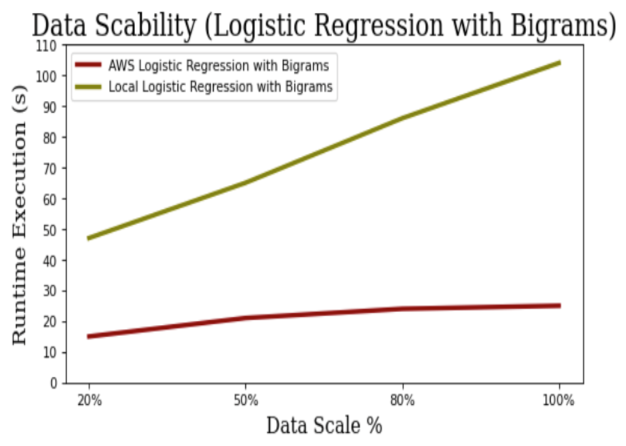


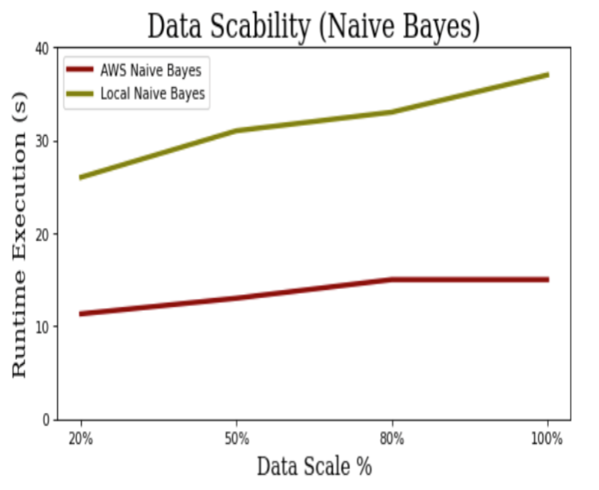
We ran our model using AWS EMR cluster and on Databricks. The above graph shows the time taken vs the cluster size. As we increased the cluster size the time taken decreased. The above graph is based on the model trained using data used for training 80:20.

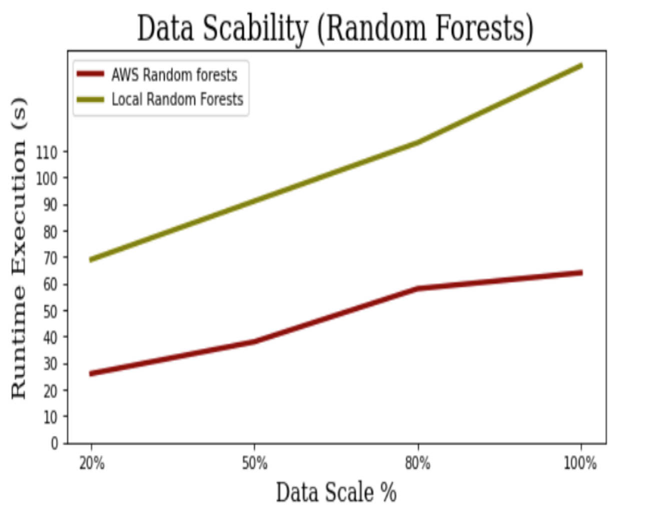
<u>Data Scalability by varying the scale of the data from 20%, 50%, 80% and 100%:</u>

We tried to run our model using AWS EMR cluster and on Databricks. The below graphs show the time taken vs the data scale. As we increased the data from 20% to 100% the time taken increased. The below graphs are based on the models trained using data with varying data scale.









CONCLUSION AND FUTURE WORK

Amazon product reviews in the form of product feedback, and product rating for distinct product categories are treated as big data and which cannot be interpreted directly for the purpose of sentiment analysis; It is therefore preprocessed in order to be suitable for mining tasks. We have used Amazon EMR clusters to analyze our data stored on Amazon S3 bucket and run our models on Pyspark Kernel as offered by EMR notebooks. In this project, we propose an efficient sentiment prediction technique, utilizing the Apache Spark's Machine Learning library to execute different classification algorithms using NLP data preprocessing tasks such as Tokenization, CountVectors, Bigrams, TF-IDF model, HashingTF and so on.

We compare the performance of the classifiers (Naive Bayes, Logistic Regression, Random Forest) using metrics: Accuracy, Precision and AUC. At first, we ran the models on our local machine using Databricks and created checkpoints accordingly. During the process we found the benchmark point on a local machine which is Databricks that runs on a single spark node where the compute node was broken during the execution of Gradient Boosted Trees model. We later moved the computation by instantiating 3 nodes (master and 2 slave) on the EMR cluster which ran within an hour of exec time.

We also performed K fold cross validation technique and tuned our model hyperparameters to boost performance. We even dribbled with ensemble methods such as Gradient Boosted trees and XGBoost, however that did not give any better results as compared to traditional logistic Regression with Bigrams, HashingTF and IDF Vectorization.

In our analysis we found that Logistic Regression with Bigram features has the highest accuracy. The results indicate that the runtime was the least for Naive Bayes and Logistic Regression with Bigrams when the number of cluster nodes was increased, and data scale kept the same. It is also found that the scale of data is directly proportional to the runtime for the spark job irrespective of the number of nodes in the cluster. In the future, we aim to examine the influence of adding features on the input vector and usage of larger product review datasets.

REFERENCES

[1] https://jmcauley.ucsd.edu/data/amazon/

[2] Sentiment Analysis on Twitter Data using Apache Spark Framework, Hossam Elzayady Khaled M. Badran Gouda I. Salama; IEEE 2017

[3] A Study of Big Data Analytics using Apache Spark with Python and Scala, Yogesh Kumar Gupta, Surabhi Kumari, IEEE 2021.

[4]https://docs.aws.amazon.com/emr/latest/ManagementGuide/emr-what-is-emr.html

[5] https://builtin.com/data-science/intro-descriptive-statistics

[6]https://spark.apache.org/docs/latest/api/python/reference/api/pyspark.ml.feature.Bucketizer.html

[7]https://machinelearningmastery.com/data-sampling-methods-for-imbalanced-classification/

[8]https://medium.com/analytics-vidhya/stratified-sampling-in-machine-learning-f5112b5b9cfe