



DECEMBER 15, 2016

AMAZON FINE FOODS REVIEWS

SENTIMENT ANALYSIS

AAYUSH AGRAWAL
MEGHA GUPTA
YING PAN
SAMI RUSSELL NOUR




Table of Contents

BUSINESS UNDERSTANDING	3
DATA UNDERSTANDING	3
ANALYTICS	3
GOAL OF THE PROJECT	3
DATA EXPLORATION	4
METHODOLOGY	5
<i>Data Manipulation</i>	<i>5</i>
<i>Text Mining</i>	<i>6</i>
<i>Sentiment Analysis Using Vader sentiment library</i>	<i>7</i>
<i>Modeling and evaluation</i>	<i>8</i>
<i>Deployment.....</i>	<i>10</i>
FINDINGS AND CONCLUDING REMARKS	16

BUSINESS UNDERSTANDING

Amazon is an Ecommerce website based out of Seattle, Washington. It has become the largest internet-based retailer in the world, and sells products in various categories including consumer electronics, beauty products, gourmet food and groceries.

Amazon allows users to view and write reviews about products. As time has progressed, the number of products as well as the number of reviews for each product have increased exponentially. Most interested users like to check the ratings and reviews in order to evaluate the product and learn about potential issues or warranty claims. However, simply looking at the rating score does not provide much information, and one would have to look into the text of the reviews in order to get useful insights. Sifting through thousands of reviews to find the most helpful ones can be a difficult task for any user. Currently, Amazon provides the most helpful positive as well as most helpful critical review for each product. This increases customer experience as it saves a user from scrolling through the vast amount of textual data.

DATA UNDERSTANDING

The dataset was obtained from Kaggle and the Stanford Network Analysis Project. It is in the form of a comma separated(CSV) file, and contains 568,454 reviews from 256,059 users for 74,258 food products. The reviews span from October 1999 - October 2012.

Some critical columns in the dataset are as follows:

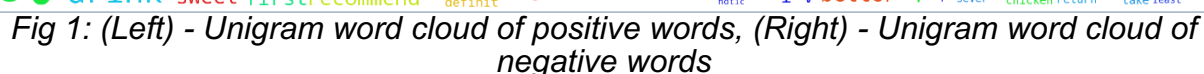
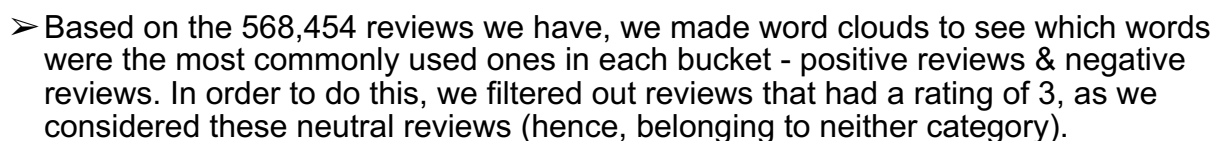
- Helpfulness Numerator: number of users who found the review helpful
- Helpfulness Denominator: number of users who indicated if the review was helpful
- Rating by the user (Scale of 1 - 5)
- Summary of the review
- Text of the review

ANALYTICS

GOAL OF THE PROJECT

Our project focused on doing a sentiment analysis on reviews posted by users for food products on Amazon. Analyzing user reviews on food products on Amazon, we will be classifying a particular review as positive or negative sentiment based on the text. The business use case of the project is to develop a review classification model that will help Amazon classify the vast amount of user reviews into positive and negative review buckets, including those that are not accompanied by a star rating.

- The current distribution of user rating/score over the dataset is as follows:



- We also made one bi-gram word cloud for each bucket, which gives us an intuitive picture of which are the most frequent bi-grams in our original dataset. The idea was to incorporate the word order and then generate the commonly used 2-word phrases.



Fig 2: (Left) - Bigram Word cloud of Positive words, (Right) - Bigram word cloud of negative words

METHODOLOGY

Our project was built entirely in Spark using PySpark and MLib library.

Data Manipulation

We filtered out the data which had the score 3 and converted ratings 1 and 2 as 0 i.e. negative rating and 4 and 5 as 1 i.e. positive rating.

```
#Importing data set and making a SparkSQLDataFrame
sc = SparkContext("local", "test")
sqlContext = SQLContext(sc)
data = sqlContext.read.csv('file:///D:/Carlson/Fall/6330 Harvesting Big Data/FinalProject/Reviews.csv', header = True )

#Filtering out records with rating = 3
messages = data.filter(data.Score != "3").select('Score', 'Text')
messages = messages.withColumn("Score", messages["Score"].cast(DoubleType()))

#Converting score to a binary positive - 1 and negative - 0 feature
def toBinary(score):
    if score >= 3: return 1
    else: return 0
udfScoretoBinary=udf(toBinary, StringType())

messages = messages.withColumn("Target", udfScoretoBinary("Score")).select('Text', 'Target')
messages.show(2)
```

Text	Target
I have bought sev...	1
"Product arrived ...	0

Text Mining

Case Normalization

To do text mining, our first step was to normalize the case of our reviews to make sure every term is in lowercase. The example output is 'i have bought several of the vitality canned dog food'.

```
#Lower casing the text
def lower_text(line):
    return line.lower()
udflower_text=udf(lower_text, StringType())
messages_lower = messages.withColumn("text_lower", udflower_text("Text")).select('text_lower', 'Target')
```

Tokenization

Next, we tokenized the lowercase text into words. The list of tokens becomes input for further processing. The example output is 'i', 'have', 'bought', 'several', 'of', 'the', 'vitality', 'canned', 'dog', 'food'.

```
#Tokenizing the document
tokenizer = Tokenizer(inputCol="text_lower", outputCol="words")
wordsDataFrame = tokenizer.transform(messages_lower)
```

Stop words removal

Stop words are basically a set of commonly used words in any language, not just English. The reason why stop words are critical to many applications is that, if we remove the words that are very commonly used in a given language, we can focus on the important words instead. So here in our case, we removed those stop words before calculating TF-IDF. The example output of our first review after removing stop words is 'have', 'bought', 'several', 'vitality', 'canned', 'dog', 'food'.

```
# Removing stopwords
remover = StopWordsRemover(inputCol="words", outputCol="words_filtered")
wordsDataFrame1 = remover.transform(wordsDataFrame).select("Target", "words_filtered")
```

Stemming

A stemming algorithm is a process of linguistic normalization, in which the variant forms of a word are reduced to a common form. The further output is 'bought' 'sever' 'vital' 'can' 'dog' 'food' for the first review.

```
# Stemming the text
stemmer = PorterStemmer()
def stem_text(tokens):
    stems = stem_tokens(tokens, stemmer)
    return ' '.join(stems)

udfstem_text=udf(stem_text, StringType())
wordsDataFrame2 = wordsDataFrame1.withColumn("final_text", udfstem_text("words_filtered")).select('final_text', 'Target')
```

TF-IDF

After our data preparation is done, we did TF-IDF in pyspark using sparse metrics since there are null values in the input dataset. Term frequency is just the number of times the word occurs in a particular comment in a review. While IDF is calculated as:

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

D - Number of reviews in the dataset (corpus)

DF(t,D) - Number of documents in which the word occurs.

```
#Creating pipeline for Tokenizing, TF - IDF and Logistic Regression Model
tokenizer = Tokenizer(inputCol="text", outputCol="words")
hashingTF = HashingTF(inputCol=tokenizer.getOutputCol(), outputCol="hashing")
idf = IDF(inputCol=hashingTF.getOutputCol(), outputCol="features")
lr = LogisticRegression(maxIter=10, regParam=0.01)
pipeline = Pipeline(stages=[tokenizer, hashingTF, idf, lr])
# Training the model
model = pipeline.fit(training)
```

TF - IDF creates a sparse metrics format RDD which is highly efficient in storage and computation as it only stores the places where there is a value hence in case of data sparsity sparse metric RDD is efficient.

Sentiment Analysis Using Vader sentiment library

To analyze sentiment by popular sentiment analysis library in Python, we used Vader Sentiment analyzer library. Vader Sentiment library provides four scores for a given text mentioned below:

1. Compound: It represents the polarity of the statement/text given ranging from 0 to 1
2. Negative: Negative sentiment score of the text given ranging from 0 to 1
3. Positive: Positive sentiment score of the text given ranging from 0 to 1
4. Neutral: Neutral sentiment score of the text given ranging from 0 to 1

We took a random sample of 20,000 instances from data and calculated sentiment score for each text on 4 parameters.

```
# importing the data
sentiment_pandas = pd.read_csv('D:/Carlson/Fall/6330 Harvesting Big Data/FinalProject/Reviews.csv')

#Sampling
sentiment_pandas = sentiment_pandas.sample(n = 20000)

#Calculating sentiments
sentiment_pandas_1 = pd.DataFrame(list(sentiment_pandas.Text.apply(lambda x: vaderSentiment.SentimentIntensityAnalyzer().polarity_scores(x))))

#Ratings processing
sentiment_pandas_1['Target'] = list(sentiment_pandas_1['Score'])
sentiment_pandas_1 = sentiment_pandas_1.ix[sentiment_pandas_1.Target != 3,]
sentiment_pandas_1.Target = sentiment_pandas_1.Target.apply(lambda x: 1 if x>3 else 0)

# Plotting the positive and negative sentiments average values from Vader Sentiment Library
plt.rcParams['figure.figsize'] = (10.0, 5.0)
sentiment_pandas_1.groupby('Target').mean().plot(kind='bar',alpha=0.75, rot=0,color = ['b','r','y','g']);
plt.suptitle('Vader Sentiment Analysis', fontsize=20)
plt.axvline(0.5, color='black');
```

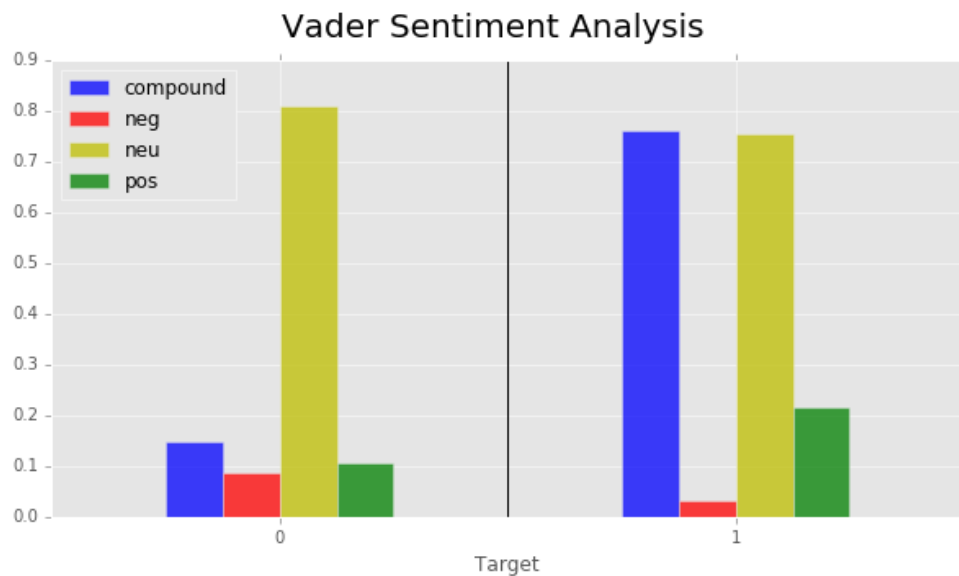


Fig 3: Vader aggregated score on different components for positive and negative class comments

Modeling and evaluation

The model to predict positive or negative review based on TF-IDF feature to predict the positive and negative spin words were built using three different algorithms and were evaluated over average accuracy in a 5 - fold cross-validation framework.

Logistic Regression

We used grid search parameter search to find optimal value of regularization parameter to optimize the 5 - fold cross validation average accuracy


```
#Creating pipeline for Tokenizing, TF - IDF and Logistic Regression Model
tokenizer = Tokenizer(inputCol="text", outputCol="words")
hashingTF = HashingTF(inputCol=tokenizer.getOutputCol(), outputCol="hashing")
idf = IDF(inputCol=hashingTF.getOutputCol(), outputCol="features")
lr = LogisticRegression(maxIter=10, regParam=0.01)
pipeline = Pipeline(stages=[tokenizer, hashingTF, idf, lr])
# Training the model
paramGrid = ParamGridBuilder()\
    .addGrid(lr.regParam, [0.1, 0.01, .001, 1, 100, 200]) \
    .build()
crossval = CrossValidator(estimator=pipeline,
                          estimatorParamMaps=paramGrid,
                          evaluator=BinaryClassificationEvaluator() ,
                          numFolds=5)
model = crossval.fit(training)
model.avgMetrics
```

The result of that was as follows -

Regularization Parameter	Accuracy
0.001	94.01%
0.01	94.72%
0.1	95.61%
1	95.20%
100	90.63%
200	90.16%

Logistic Regression with 0.1 regularization parameter got us the highest result of 95.61% accuracy.

Naive Bayes

We followed similar process of 5 - fold cross validation method in Naive bayes with grid search on smoothing function. We couldn't get accuracy above 57-58%, this may be caused by violation of conditional independence assumption.

```
#Creating pipeline for Tokenizing, TF - IDF and Logistic Regression Model
tokenizer = Tokenizer(inputCol="text", outputCol="words")
hashingTF = HashingTF(inputCol=tokenizer.getOutputCol(), outputCol="hashing")
idf = IDF(inputCol=hashingTF.getOutputCol(), outputCol="features")
nb = NaiveBayes(smoothing=1.0, modelType="multinomial")
pipeline = Pipeline(stages=[tokenizer, hashingTF, idf, nb])
# Training the model
paramGrid = ParamGridBuilder().addGrid(nb.smoothing, [1,2]).build()
crossval = CrossValidator(estimator=pipeline,
                          estimatorParamMaps=paramGrid,
                          evaluator=BinaryClassificationEvaluator() ,
                          numFolds=5)
model = crossval.fit(training)
model.avgMetrics
```

Gradient boosted trees

We also try to use GBT algorithm in spark using a 5 - fold cross validation using Grid search on number of trees but now able to get required accuracy and with increasing the number of trees we went into memory issues.

```
#Creating pipeline for Tokenizing, TF - IDF and Logistic Regression Model
tokenizer = Tokenizer(inputCol="text", outputCol="words")
hashingTF = HashingTF(inputCol=tokenizer.getOutputCol(), outputCol="hashing")
idf = IDF(inputCol=hashingTF.getOutputCol(), outputCol="features")
gbt = GBTClassifier(labelCol="indexedLabel", featuresCol="indexedFeatures", maxIter=10)
pipeline = Pipeline(stages=[tokenizer, hashingTF, idf, nb])
# Training the model
paramGrid = ParamGridBuilder().addGrid(gbt.maxIter, [5,10,15]).build()
crossval = CrossValidator(estimator=pipeline,
                          estimatorParamMaps=paramGrid,
                          evaluator=BinaryClassificationEvaluator(),
                          numFolds=5)
model = crossval.fit(training)
model.avgMetrics
```

Deployment

The PySpark application was deployed using Amazon Web Services (AWS) in order to test scalability in terms of the amount of data to be analyzed. An AWS deployment will be able to handle much larger datasets efficiently. Therefore, if the application works on AWS with the current data set containing a little over half a million records, it will be scalable up to datasets with millions or more records.

There are two components to AWS used during deployment: S3 for storage and EMR for a Spark cluster. Data and code can be uploaded into an S3 bucket and processed by Spark on the cluster.

First, the PySpark application must be adapted for running on a Spark cluster. If created in an iPython notebook, it must be converted into a ".py" file. Most Python libraries are already imported on an AWS cluster, so they are not necessary to include. Two versions of the application must be created - one for running the application in the normal cluster shell and another one for running the application in the PySpark shell. The main difference between the two code files is that using the normal shell requires the code to explicitly define the SparkContext, whereas in the PySpark shell a SparkContext is already running (if this line is not removed, it will result in an error because you can only have one SparkContext at a time).

The next step is to configure the AWS environment. The dataset and both Python files are uploaded to an S3 bucket on AWS. Then, a Spark cluster is created with the EMR service on AWS. Default settings are used, except for choosing an m3.xlarge cluster (this creates a cluster with up to 16 gigabytes of memory, which was necessary for this exercise). Once the cluster is ready, the cluster name node is connected to through Putty SSH. Once in the cluster, the data and Python files are downloaded onto the cluster from

the AWS S3 bucket. AWS S3 gives a URL for each file, so it is easiest to use the “wget” command and the provided URL to download the files onto the cluster.

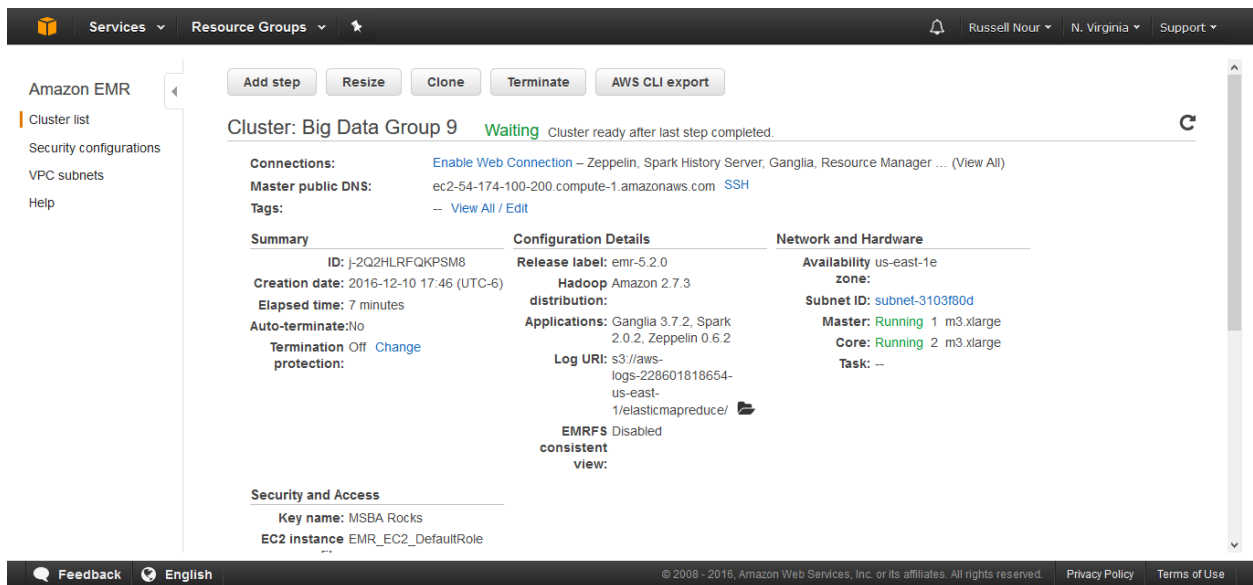


Fig 4: Spark cluster on AWS

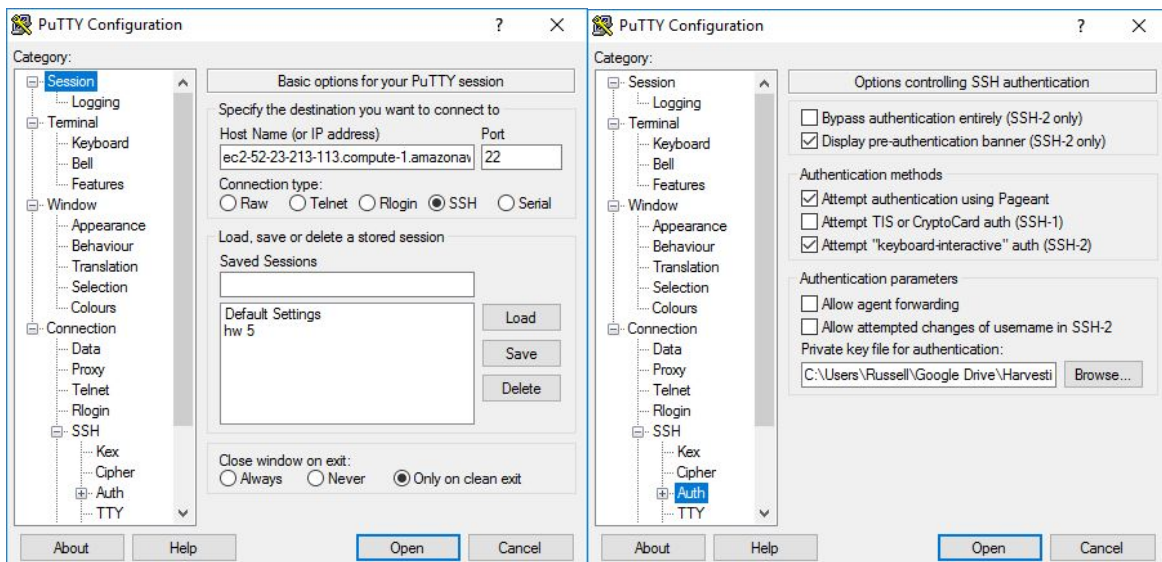


Fig 5: Connecting to AWS cluster through Putty SSH

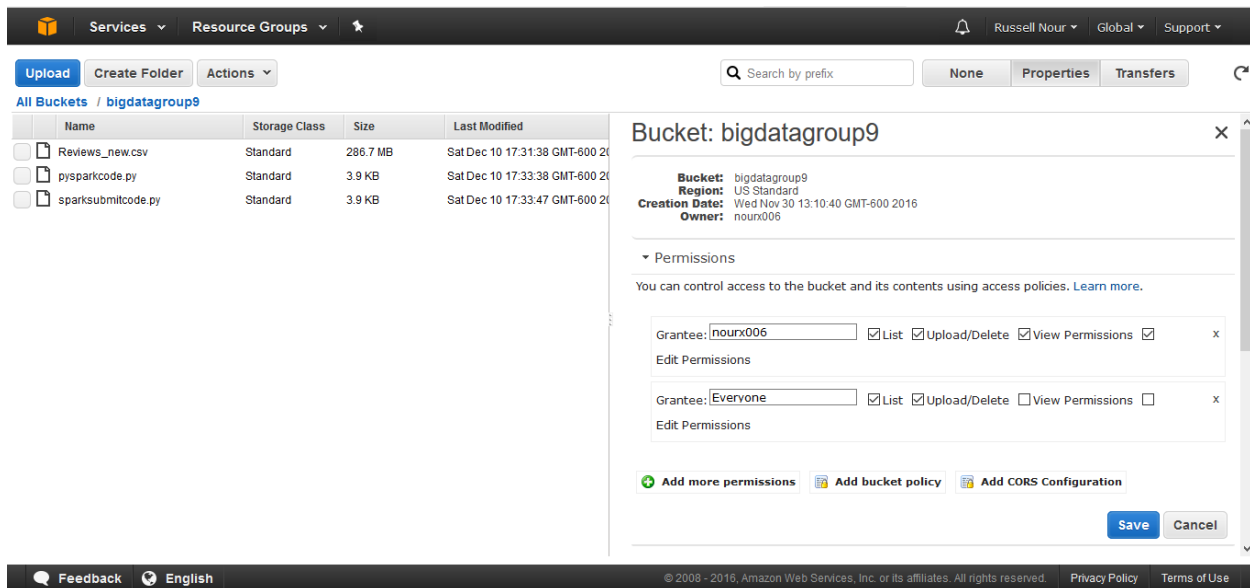


Fig 6: AWS S3 bucket with data and Python files



Fig 7: S3 bucket file URL for use with “wget” command in cluster shell

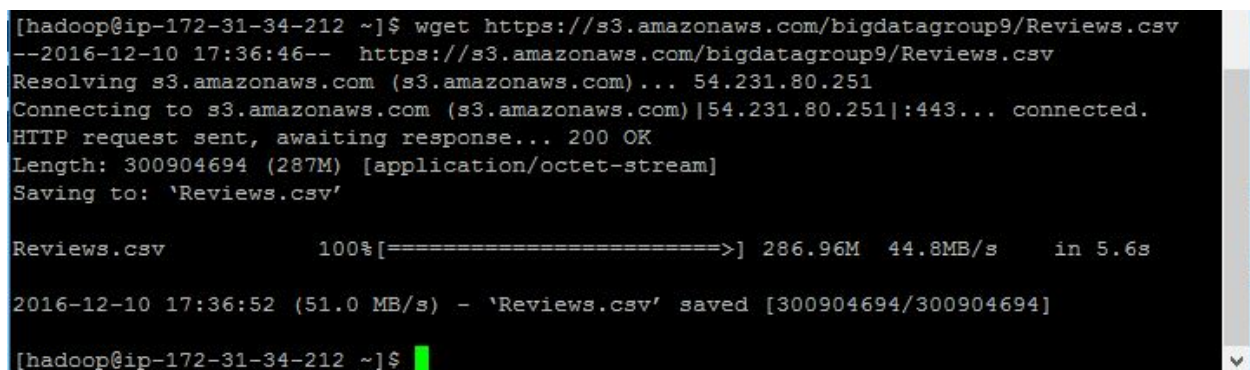


Fig 8: Downloading the dataset onto the cluster name node

Once the data and the Python scripts are on the cluster name node, it is important to make sure that the data is copied onto the actual cluster nodes themselves so that either script can be run on the data.

```
[hadoop@ip-172-31-34-212 ~]$ ls
pysparkcode.py  Reviews.csv  sparksubmitcode.py
[hadoop@ip-172-31-34-212 ~]$
```

Fig 9: Dataset and Python files on the cluster name node

```
[hadoop@ip-172-31-38-202 ~]$ ls
finefoods-ps.py  finefoods-ss.py  Reviews_new.csv
[hadoop@ip-172-31-38-202 ~]$ hadoop fs -ls
[hadoop@ip-172-31-38-202 ~]$ hadoop fs -put Reviews_new.csv
[hadoop@ip-172-31-38-202 ~]$ hadoop fs -ls
Found 1 items
-rw-r--r--  1 hadoop hadoop  300687716 2016-12-13 22:25 Reviews_new.csv
[hadoop@ip-172-31-38-202 ~]$
```

Fig 10: Putting the data onto the Hadoop cluster nodes (names of files have changed slightly due to using a new cluster and new file versions)

There are two ways to run the scripts: using the “spark-submit” command in the cluster terminal and using the “execfile” command within the PySpark shell. The only difference between the scripts for these two methods is that when in the PySpark shell a SparkContext is already defined, so the line of code defining the SparkContext must be removed (otherwise an error will occur because more than one SparkContext cannot be running at the same time).

```
pysparkcode.py  Reviews.csv  sparksubmitcode.py
[hadoop@ip-172-31-34-212 ~]$ spark-submit sparksubmitcode.py
16/12/10 17:42:13 INFO SparkContext: Running Spark version 2.0.2
16/12/10 17:42:14 INFO SecurityManager: Changing view acls to: hadoop
16/12/10 17:42:14 INFO SecurityManager: Changing modify acls to: hadoop
16/12/10 17:42:14 INFO SecurityManager: Changing view acls groups to:
16/12/10 17:42:14 INFO SecurityManager: Changing modify acls groups to:
16/12/10 17:42:14 INFO SecurityManager: SecurityManager: authentication disabled; ui acls disabled; users with view permissions: Set(hadoop); groups with view permissions: Set(); users with modify permissions: Set(hadoop); groups with modify permissions: Set()
16/12/10 17:42:14 INFO Utils: Successfully started service 'sparkDriver' on port 35078.
16/12/10 17:42:14 INFO SparkEnv: Registering MapOutputTracker
16/12/10 17:42:14 INFO SparkEnv: Registering BlockManagerMaster
16/12/10 17:42:14 INFO DiskBlockManager: Created local directory at /mnt/tmp/blockmgr-6aa823e0-a98e-4aeb-b020-d16c68c4abb2
```

Fig 11: Python file starting to run using the spark-submit command in the cluster terminal

[illegible]

Fig 12: Executing the Python file within the PySpark shell

```
hadoop@ip-172-31-34-212:~  
cala:1103) finished in 0.212 s  
16/12/10 17:48:03 INFO DAGScheduler: Job 21 finished: treeAggregate at LogisticRegression.  
scala:1103, took 1.516566 s  
16/12/10 17:48:03 INFO LBFGS: Step Size: 1.000  
16/12/10 17:48:03 INFO LBFGS: Val and Grad Norm: 0.107979 (rel: 0.000732) 0.0292308  
16/12/10 17:48:03 WARN LogisticRegression: LogisticRegression training finished but the re  
sult is not converged because: max iterations reached  
16/12/10 17:48:03 INFO MapPartitionsRDD: Removing RDD 67 from persistence list  
16/12/10 17:48:03 INFO BlockManager: Removing RDD 67  
16/12/10 17:48:03 INFO CodeGenerator: Code generated in 28.29394 ms  
16/12/10 17:48:03 INFO Instrumentation: LogisticRegression-LogisticRegression_4fb590d51216  
cd617d06-307025862-1: training finished  
root  
|-- text: string (nullable = true)  
|-- label: double (nullable = true)  
|-- words: array (nullable = true)  
|   |-- element: string (containsNull = true)  
|-- hashing: vector (nullable = true)  
|-- features: vector (nullable = true)  
|-- rawPrediction: vector (nullable = true)  
|-- probability: vector (nullable = true)  
|-- prediction: double (nullable = true)  
  
16/12/10 17:48:04 INFO CodeGenerator: Code generated in 101.362659 ms  
16/12/10 17:48:04 INFO SparkContext: Starting job: reduce at /home/hadoop/sparksubmitcode.  
py:186  
16/12/10 17:48:04 INFO DAGScheduler: Got job 22 (reduce at /home/hadoop/sparksubmitcode.py  
:186) with 32 output partitions  
16/12/10 17:48:04 INFO DAGScheduler: Final stage: ResultStage 36 (reduce at /home/hadoop/s  
parksubmitcode.py:186)  
16/12/10 17:48:04 INFO DAGScheduler: Parents of final stage: List()  
16/12/10 17:48:04 INFO DAGScheduler: Missing parents: List()  
16/12/10 17:48:04 INFO DAGScheduler: Submitting ResultStage 36 (PythonRDD[125] at reduce a  
t /home/hadoop/sparksubmitcode.py:186), which has no missing parents  
16/12/10 17:48:04 INFO MemoryStore: Block broadcast_45 stored as values in memory (estim  
ed size 4.1 MB, free 5.8 GB)  
16/12/10 17:48:04 INFO MemoryStore: Block broadcast_45_piece0 stored as bytes in memory (e  
stimated size 2.5 MB, free 5.8 GB)  
16/12/10 17:48:04 INFO BlockManagerInfo: Added broadcast_45_piece0 in memory on 172.31.34.  
212:45139 (size: 2.5 MB, free: 5.8 GB)
```

Fig 13: Output while Python script running through cluster terminal (spark-submit)

```
hadoop@ip-172-31-34-212:~  
intended', u'to', u'represent', u'the', u'product', u'as', u'""jumbo""'], Target=u'0')  
Row(words=[u'"this', u'is', u'a', u'confection', u'that', u'has', u'been', u'around',  
u'a', u'few', u'centuries.', u'', u'it', u'is', u'a', u'light', u'pillowy', u'citru  
s', u'gelatin', u'with', u'nuts', u'-' , u'in', u'this', u'case', u'filberts.', u'and'  
, u'it', u'is', u'cut', u'into', u'tiny', u'squares', u'and', u'then', u'liberally',  
u'coated', u'with', u'powdered', u'sugar.', u'', u'and', u'it', u'is', u'a', u'tiny',  
u'mouthful', u'of', u'heaven.', u'', u'not', u'too', u'chewy', u'and', u'very', u'f  
lavorful.', u'', u'i', u'highly', u'recommend', u'this', u'yummy', u'treat.', u'', u'  
if', u'you', u'are', u'familiar', u'with', u'the', u'story', u'of', u'c.s.', u'lewis'  
", u'""the', u'lion'], Target=u'1')  
+-----+-----+  
|Target|      words_filtered|  
+-----+-----+  
|      1|[bought, several,...|  
|      0|["product, arrive...|  
+-----+-----+  
only showing top 2 rows  
  
16/12/10 18:23:18 WARN BLAS: Failed to load implementation from: com.github.fommil.ne  
tlib.NativeSystemBLAS  
16/12/10 18:23:18 WARN BLAS: Failed to load implementation from: com.github.fommil.ne  
tlib.NativeRefBLAS  
16/12/10 18:23:27 WARN LogisticRegression: LogisticRegression training finished but t  
he result is not converged because: max iterations reached  
root  
|-- text: string (nullable = true)  
|-- label: double (nullable = true)  
|-- words: array (nullable = true)  
|   |-- element: string (containsNull = true)  
|-- hashing: vector (nullable = true)  
|-- features: vector (nullable = true)  
|-- rawPrediction: vector (nullable = true)  
|-- probability: vector (nullable = true)  
|-- prediction: double (nullable = true)  
[Stage 36:> (0 + 16) / 32]
```

Fig 14: Output while Python script running within PySpark shell (cleaner output due to default log settings)

FINDINGS AND CONCLUDING REMARKS

Spark's capabilities in each step of a typical predictive modeling exercise were demonstrated. Specifically, it was shown that Spark is fully capable of:

- Data processing and manipulation
- Text mining
- Building predictive models
- Evaluating predictive models
- Deployment for distributed computing

Therefore, Spark can be used for any aspect of a "Big Data" analysis without the need for additional packages, tools, or libraries.

Codes

We have made a github repository to showcase our code. Please see the link below:
<https://github.com/aayushmnit/big-data-project>