ST446 Project

Twitter Sentiment Analysis Using Apache Spark

Candidate Number: 14565

Abstract

In this empirical project, we perform sentiment analysis on Stanford's Sentiment140 dataset. The dataset is fairly large and cannot be processed on a single machine. We build scalable machine learning models using Apache Spark and it's DataFrame API and perform sentiment analysis on the dataset using different machine learning models. The models considered are – Elastic Net Logistic Classifier, Decision Tree Classifier, Random Forest Classifier and Gradient Boost Tree Classifier. Spark's DataFrame based machine learning library, SparkML is used to build and evaluate these models.

I. Introduction

Machine Learning techniques have been adopted by many sectors of society with applications ranging from healthcare to marketing. However, more recently there has been a need for platforms and models that can handle the big datasets and data with high velocity. Apache Spark has emerged as a platform that can handle these tasks efficiently and in a fault-tolerant manner. For this project, we implement large scale sentiment analysis using Spark's various tools. Apache Spark is an open-source platform that was first developed at the University of California, Berkley's AMP Lab. Spark can distribute large datasets across a computing cluster and perform processing tasks on these datasets in parallel. These two features make Spark efficient at handling data of large volume or velocity. The main architecture of Spark consists of a driver, which breaks down a task into multiple components and assigns them to different worker nodes and executers that execute these split up tasks in parallel. (Spark, 2018)

At the heart of Spark is the Resilient Distributed Dataset or RDD. A RDD is a schema-less distributed collection of objects. A dataset stored as an RDD is partitioned across different nodes in a computing cluster. As the name "Resilient" suggests, the data is stored in a fault-tolerant manner. RDDs store multiple copies of its partitions across its nodes. Data processing tasks can be applied to RDDs in a manner such that the tasks can be run in parallel and evaluate data in an efficient manner. Data from an RDD can also be made to persist in-memory so that the user can use the data multiple times with reduced computation time. RDDs are particularly suited to big data processing due to their many features. First, RDDs employ lazy evaluation. This means that the results from transformations are only computed once the results are actively required. Second, RDDs are fault tolerant which allows any data lost by failure to be automatically recovered. Third, the immutability of RDDs makes data sharing easier and safer. However, RDDs have certain drawbacks such as being schema-less and lacking input optimization. Due to this we make the decision not to use them for this empirical project. Rather we use Spark DataFrames, which are an abstraction over RDDs and improve on its drawbacks. (Spark, 2018)

Spark DataFrames were first introduced alongside Spark version 1.3 to improve upon the drawbacks of RDDs. The primary difference is that DataFrames are two-dimensional distributed

data structures with a schema. This is similar to DataFrames from Python Pandas library where each column has a name and type. Another additional advantage of using Spark DataFrames is that it uses the Catalyst optimizer to automatically find the best plan to carry out most data processing tasks.

For our empirical application, we conduct large scale sentiment analysis in Spark using DataFrames and Spark's ML package. To provide a brief overview, Spark ML is a DataFrame based API that provides a variety of tools to make implementation of machine learning algorithms in Spark practical and scalable. It contains many common machine learning algorithms, pipelines and feature extraction tools that make it easier to implement machine learning tasks. (Meng et al., 2016)

In this project, we will use pipelines and four classification algorithms - Net Elastic Logistic Classifier, Decision Tree, Random Forest, and Gradient Boost Tree Classifier from the Spark ML API to perform sentiment analysis on tweets from Stanford's Sentiment140 dataset. We implement this project in Google Cloud using a Dataproc computing cluster. The following sections outline the data used, provide an overview of the algorithms, explain the implementation of the code and then discuss final results.

II. Data

We use the Sentiment140 dataset which was developed by graduate students from Stanford's Computer Science department using the Twitter Search API. The dataset contains 1.6 million tweets and is made available in a CSV format. The dataset contains 6 columns - polarity label, tweet id, tweet date, query, username and the text of the tweet. For our project, we will only work with the columns of polarity label and the text of the tweet. Each tweet has a polarity label of 0 if it is a negative tweet and a polarity label of 4 if it is a positive tweet. This data was collected using computed automation. Any tweets with the "☺" emoticon were assumed to be positive and tweets with the "☹" emoticon were assumed to be negative. (Go, Bhayani, & Huang, 2009)

We start by importing the dataset as a Spark DataFrame using a specified schema. We print out the first few rows to get a general idea of that the data looks like.

```
#import data as a dataframe

from pyspark.sql.types import *

file_url = "gs://ishita2/training.1600000.processed.noemoticon.csv"

#set schema

schema = StructType([
    StructField("label", StringType(), True),
    StructField("id", StringType(), True),
    StructField("date", StringType(), True),
    StructField("query", StringType(), True),
    StructField("user", StringType(), True),
    StructField("text", StringType(), True)
])

twitter_df = spark.read.csv(file_url, header = False, schema = schema)
```

```
# view first 5 tweet data

twitter_new.take(5)
```

```
[Row(label='0', id='2303322829', date='Tue Jun 23 18:01:57 PDT 2009', que
ry='NO_QUERY', user='HunneysRunnin', text="Good night everybody...&quot;w
hat a day...what a day&quot; I'm headed to the gym to do my evening work
out. I have to meet my trainer this weekend "),
 Row(label='0', id='2286003454', date='Mon Jun 22 15:52:08 PDT 2009', que
ry='NO_QUERY', user='kurdman', text='@RebazQ i feel you bro, its pretty h
ot here too! '),
 Row(label='4', id='1468554776', date='Tue Apr 07 02:23:03 PDT 2009', que
ry='NO_QUERY', user='reneasaurus', text="Won my first game of settlers  .
.and I didn't even realize I could have won the pound before haha"),
 Row(label='0', id='2259978543', date='Sat Jun 20 18:28:27 PDT 2009', que
ry='NO_QUERY', user='stacmasters', text='Hanging out in the lobby of the
hotel at MetroCon. @SnafuComics ditched me for some guest related activit
ies! '),
 Row(label='0', id='2261946759', date='Sat Jun 20 21:37:10 PDT 2009', que
ry='NO_QUERY', user='JackieXTaylor', text="Seriously feel like I'm gonna
pass out...and my bed's so far away ")]
```

As we can see, the tweets contain a lot of non-numerical characters, upper case letters etc. which if left unchanged are going to make text analysis more difficult. Thus, we focus on cleaning the tweets. For this we use the “regex_replace” function which finds a pattern in a string and then replaces it with a specified pattern. We define a function “clean” that takes a tweet as an input. This function then makes all letters lowercase, removes retweets, removes links and removes any

non-alphanumeric characters. We create a new DataFrame of cleaned tweets and take a look at some of them.

```
from pyspark.sql.functions import col, lower, regexp_replace, split

#clean the tweet's words for better analysis
def clean(tweet):
  lower_tweet = lower(tweet)
  clean_tweet = regexp_replace(lower_tweet, "^rt ", "") #remove retweet
  clean_tweet = regexp_replace(clean_tweet, "(https?\://)\S+", "") #remove Links
  clean_tweet = regexp_replace(clean_tweet, "@\w+", "")  # remove @
  clean_tweet = regexp_replace(clean_tweet, "[^a-zA-Z0-9\\s]", "") #remove anything not an alp
habetor number
  return clean_tweet

#create new dataframe to hold the cleaned tweets and polarity scores
clean_tweet_df = twitter_new.select(clean(col("text")).alias("tweet"), col("label").alias("tar
get"))

#print some cleaned tweets

clean_tweet_df.cache()
clean_tweet_df.show(n=5, truncate=False, vertical=True)
```

```
-RECORD 0-----------------------------------------------------------------------------------
----------------------------------
 tweet  | registered for next year individual tax acct systems and control presentational spe
aking and consumer behavior oh fun
 target | 0
-RECORD 1-----------------------------------------------------------------------------------
----------------------------------
 tweet  |  oopsthat was my mistake
 target | 0
-RECORD 2-----------------------------------------------------------------------------------
----------------------------------
 tweet  |  no pandas at the zoo i was going to buy you summat from the gift shop but the chea
pest thing was 100000000
 target | 0
-RECORD 3-----------------------------------------------------------------------------------
----------------------------------
 tweet  | ugh its 406 am i cant sleep cant find my cell phone number im having problems with
my bank and its so stinking hot
 target | 0
-RECORD 4-----------------------------------------------------------------------------------
----------------------------------
 tweet  | dying  mouth surgery tomorrow
 target | 0
only showing top 5 rows
```

One important thing to note is that we have a balanced dataset. This means that out of the 1.6 million tweets, 800,000 are positive and the other 800,000 are negative. We shuffle and split our cleaned data into three sets – 70% for training, 10% for validation and 20% for testing.

```
total_tweets = clean_tweet_df.count()
positive_tweets = clean_tweet_df.filter("label == 4").count()
neutral_tweets = clean_tweet_df.filter("label == 2").count()
negative_tweets = clean_tweet_df.filter("label == 0").count()

print(f"Total Tweets: {total_tweets}")
print(f"Positive Tweets: {positive_tweets}")
print(f"Neutral Tweets: {neutral_tweets}")
print(f"Negative Tweets: {negative_tweets}")
```

```
Total Tweets: 1600000
Positive Tweets: 800000
Neutral Tweets: 0
Negative Tweets: 800000
```

```
#shuffle and split the dataset
(train_data, val_data, test_data) = clean_tweet_df.randomSplit([0.7, 0.1, 0.2], seed = 10)

#cache datasets in memory
train_data.cache()
val_data.cache()
test_data.cache()
```

```
DataFrame[tweet: string, target: string]
```

III. Elastic Net Logistic Classifier

i. Introduction

Logistic regression is a parametric machine learning algorithm used for binary classification. Even though Logistic regression can be used from multi-class classification, for this project we will be using it to classify tweets into two categories- negative and positive. Logistic regression works by fitting a S shaped curve to our input data using the sigmoid function. For this, we first have to convert our vector of inputs into their weighted sum. Thus, given input vector x and a weight vector w^T we have the weighted sum:

$$z = w^T x$$

The weighted sum is then classified to a category using the sigmoid function. The sigmoid function is well suited for binary classification because it has a range between 0 and 1. Thus, it provides us with the probability of the input belonging to one of the categories. If the output of the sigmoid function is greater than 0.5, we assign the observation to the positive category and if not then the observation is assigned to the negative category. The sigmoid function is as follows:

$$f(z) = \frac{1}{1 + e^{-z}}$$

Now we need to find the best weights for our models. This can be done by fitting a probability distribution over our training dataset such that the parameters of the probability distribution maximize the likelihood that the dataset occurs from the distribution. For example, using the Bernoulli distribution. This can be done by minimizing the logistic loss function.

$$L_{log} = \log(1 + \exp(-y\, w^T x))$$

To prevent the model from overfitting we add regularization to this loss function. It adds bias to the model and decreases variance to improve model performance on unseen datasets. Regularization works by adding penalty terms to the loss function which shrinks weights towards zero. There are two main types of penalties – the L1 penalty and the L2 penalty. The L2 penalty can be defined as the sum of the squared weights from the weights vector. The λ parameter controls the degree with which the weights are shrunk. Larger values of λ will shrink the weights more and vice-versa. This penalty shrinks the weights towards zero. However, due to its specification it never shrinks a weight to zero. This is not ideal if we are trying to select only the most important input features. To overcome this drawback, we can use the L1 penalty. The L1 penalty is defined as the sum of absolute weights. However, the L1 penalty has its own shortcomings in that it tends to focus on only one input feature when we have a number of different correlated input features. Thus, given that both the penalties have their own drawbacks and advantages, we use the elastic net regularization which incorporates both L1 and L2 penalties. Elastic net employs an additional feature α which lies between 0 and 1 and decides how much weight is given to each penalty.

$$L2 = L_{log} + \lambda \sum_{i=1}^{n} w_i^2$$

$$L1 = L_{log} + \lambda \sum_{i=1}^{n} |w_i|$$

$$Elastic\ Net\ Loss = L_{log} + \lambda \sum_{i=1}^{n} (\alpha w_i^2 + (1 - \alpha)|w_i|)$$

We use grid search to select the optimum values for the α and λ parameters.

ii. Grid Search

We first build the ML pipeline that allows us to chain multiple steps together and make grid search and logistic classification possible. We start by specifying the different components to be included in our pipeline. First, we need to tokenize the tweets. For this, we use the regex tokenizer which breaks the tweets down into words. Keep in mind that these tweets have already been cleaned. We then load common English words into a variable named "stop_words" and remove these stopwords from our tweets using a stopwords remover. This is done because stopwords do not provide any information to the model for making its final classification. Once stopwords have been removed, the leftover tokens now need to be encoded as fixed length vectors. We use the CountVectorizer function for this purpose. The CountVectorizer transforms each tweet into a vector of token count. In this process, it generates a vocabulary of the words present in the tweets. In our implementation, we limit the vocabulary size to 2^{17} most frequently occurring words. We also use the parameter minDF to discard any words that appear in less than 5 tweets. We do this as words that appear in only a small number of tweets will not contribute much information towards the classification process. We then use IDF to compute inverse document frequency. IDF takes the vectorized outputs from CountVectorizer and scales the features. It decreases the weights of the features that appear in the tweets with high frequency. This takes care of common words that the stop words remover might have missed. So, if a feature appears in almost all tweets its IDF is close to zero. The next step for constructing the pipeline is to encode our class labels into a column of label indices using the StringIndexer.

```
#building a pipeline for validation and model fit
from pyspark.ml.classification import LogisticRegression

tokenizer = RegexTokenizer(inputCol="tweet",
                           outputCol="words",
                           pattern="\\W")


stopwords_rm = StopWordsRemover(stopWords=stop_words,
                                caseSensitive=False,
                                inputCol="words",
                                outputCol="filtered")

count_vec = CountVectorizer(minDF=5., vocabSize=2**17,
                           inputCol="filtered",
                           outputCol="vec")


sparse_vec = IDF(inputCol='vec', outputCol="features", minDocFreq=10)

label_index = StringIndexer(inputCol = "target", outputCol = "label")
```

Now that we have our terms defined, we build a grid search pipeline to find the optimal α and λ values to use for implementing the elastic net. We first initialise a logistic regression model with an initial value of α and λ and build the grid search pipeline. We then specify a grid map with different sets of parameters for α and λ. These sets are selected based on the commonly used range for the parameters. For each set of these parameters, we fit the logistic model to our validation data and then evaluate its accuracy using the Multi Class Classification Evaluator. When fitting the model over validation data, we use 80% of the data for training and leave the rest for evaluation.

```
#grid search for the best elastic net parameters

#select some initial parameters
lam = 0.02
alpha = 0.3

#build the initial grid model
grid_model = LogisticRegression(labelCol="label", featuresCol="features",
                           maxIter=100, regParam=lam,
                            elasticNetParam=alpha)

#create grid pipeline
grid_pipline = Pipeline(stages=[tokenizer, stopwords_rm, count_vec, sparse_vec, label_index, g
rid_model])

#specify a grid map with different model parameters
grid_builder = ParamGridBuilder().\
    addGrid(grid_model.regParam, [0., 0.01, 0.02, 0.03, 0.04]).\
    addGrid(grid_model.elasticNetParam, [0.,0.1, 0.2, 0.3,0.4,0.5]).\
    build()

#select the evaluator
evaluator = MulticlassClassificationEvaluator()

#put all the parameters together under the validation model
validation = TrainValidationSplit(estimator=grid_pipline,
                            estimatorParamMaps=grid_builder,
                            evaluator=evaluator,
                            trainRatio=0.8)
```

```
#fit the validation model to validation data
search_model = validation.fit(val_data)
```

The optimal parameters that the grid search found are 0.01 for λ and 0.1 for α .

```
#print the parameters of the best fit model

print("Validated Elastic Net Parameters:",search_model.bestModel.stages[-1]._java_obj.parent()
.getElasticNetParam())
print("Validated Regular Parameters:",search_model.bestModel.stages[-1]._java_obj.parent().get
RegParam())
```

```
Validated Elastic Net Parameters: 0.1
Validated Regular Parameters: 0.01
```

iii. Model Fit and Results

Now that we have our optimal parameters, we specify a new logistic regression model using these parameters. We construct a new pipeline that applies all the necessary transformations and logistic classification to the inputs and fit this pipeline to our training data to train the model.

```
#set the hyper parameters to the hyperparameters selected by validation
best_lambda = search_model.bestModel.stages[-1]._java_obj.parent().getRegParam()
best_alpha = search_model.bestModel.stages[-1]._java_obj.parent().getElasticNetParam()

en_logistic = LogisticRegression(labelCol="label", featuresCol="features",
                         maxIter=100, regParam=best_lambda,
                          elasticNetParam=best_alpha)

pipeline = Pipeline(stages=[tokenizer, stopwords_rm, count_vec, sparse_vec, label_index, en_lo
gistic])
```

```
# fit the model to train data
en_logistic_model = pipeline.fit(train_data)
```

We use this trained model to predict the classification labels for test data. The final testing metrics are calculated using the Multi Class Classification Evaluator. We present two test metrics- test accuracy and test F1. The test accuracy is the percentage of correctly predicted test labels and the test F1 is the weighted average of precision and recall. In the case of the Elastic Net Logistic Classifier both the metrics are close to 76%.

```
#make predictions on test data
en_logistic_predictions = en_logistic_model.transform(test_data)
```

```
#accuracy measurement
en_logistic_accuracy = MulticlassClassificationEvaluator(
    labelCol="label", predictionCol="prediction", metricName="accuracy")

en_logistic_f1 = MulticlassClassificationEvaluator(
    labelCol="label", predictionCol="prediction", metricName="f1")

logistic_acc = en_logistic_accuracy.evaluate(en_logistic_predictions)
logistic_f1 = en_logistic_f1.evaluate(en_logistic_predictions)

print("Test Accuracy = %g " % (logistic_acc))
print("Test F1 = %g " % (logistic_f1))
```

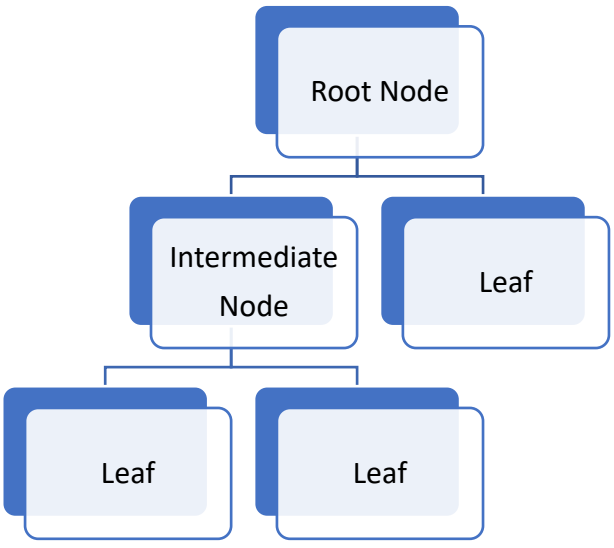
```
Test Accuracy = 0.768668
Test F1 = 0.768118
```

IV. Decision Tree Classifier

i. Introduction

Decision Trees are popular non-parametric classification models. A decision tree works by filtering the observations on a series of different input features. At the end of the filtration, we assign the observation to a category. A decision tree is made up of nodes and branches. Nodes, filter observations based on a rule relating to a single input feature. There are special nodes called leaves, where a category is assigned to an observation. Branches are connections from nodes to other nodes or leaves. This can be better explained with the help of the following diagram:

Decision Tree



Each node creates bifurcations which gives rise to the treelike structure. Decision trees are built recursively. This means we evaluate different input features at each node and then select the feature that filters observations the best. To find the best feature filters for our decision nodes we use the "Gini Impurity" measure. Thus, the split is chosen at each node based on which split maximizes the information gain, IG, when the split, s, is applied to dataset D.This can be mathematically represented in the following way:

$$argmax\ IG(D, s)$$

The information gain is computed as the Gini impurity at the parent node minus the weighted sum of the Gini impurity at the child nodes. The Gini impurity at a node for a binary classification problem is defined as:

$$Gini\ Impurity = f_1(1 - f_1)$$

Where f_1 is the frequency of label 1 at the node. When deciding on split candidates for different input features we have different approaches for categorical and continuous features. For categorical input features, if the feature can take on M possible values, then we have $2^M - 1$ potential split candidates. In binary classification, we can decrease the number of potential split candidates to $M - 1$ by ordering them by their average values. For continuous input features, we make the features discrete by splitting the values of the feature into different "bins". This is done by performing a quantile calculation on a sample of the data. (Akalin, 2020)

In Spark, the tree construction ends when one of three criteria is met. First, the node depth has reached the specified maximum depth. Second, no split candidate has information gain more that the specified minimum information gain. Third, no split candidate has children nodes that receive enough training instances to meet the minimum requirement. This prevents trees from adding more depth and potentially overfitting.

ii. Model Fit and Results

We start by building a pipeline to implement the Decision Tree Classifier. The pipeline contains a tokenizer, stopwords remover, feature extractor, inverse document frequency, and label indexer. These are implemented in exactly the same manner as for the Elastic Net Logistic Classifier. The final part of the pipeline is the Decision Tree Classifier Model. We constrain the maximum depth of the tree to 20 as deeper trees are more prone to overfitting the data. We fit the pipeline to training data and train the decision tree.

```
from pyspark.ml.classification import DecisionTreeClassifier

#preprocess the text
tokenizer = RegexTokenizer(inputCol="tweet",
                           outputCol="words",
                           pattern="\\W")

stopwords_rm = StopWordsRemover(stopWords=stop_words,
                                caseSensitive=False,
                                inputCol="words",
                                outputCol="filtered")

count_vec = CountVectorizer(minDF=5., vocabSize=2**17,
                           inputCol="filtered",
                           outputCol="vec")

sparse_vec = IDF(inputCol='vec', outputCol="features", minDocFreq=10)

label_index = StringIndexer(inputCol = "target", outputCol = "label")

dt = DecisionTreeClassifier(labelCol="label", featuresCol="features",
                           maxDepth = 20)

#create a pipeline
pipeline = Pipeline(stages=[tokenizer, stopwords_rm, count_vec, sparse_vec, label_index, dt])
```

```
#fit the model to validation data
dt_model = pipeline.fit(train_data)
```

We then use the trained Decision Tree to predict the labels of the test data tweets using the transform function. The test accuracy is 61% and the test F1 score is 57%. The trained Decision Tree has a depth of 20 with 4177 nodes.

```
#accuracy measurement
dt_accuracy = MulticlassClassificationEvaluator(
    labelCol="label", predictionCol="prediction", metricName="accuracy")

dt_f1 = MulticlassClassificationEvaluator(
    labelCol="label", predictionCol="prediction", metricName="f1")

d_acc = dt_accuracy.evaluate(dt_predictions)
d_f1 = dt_f1.evaluate(dt_predictions)

print("Test Accuracy = %g " % (d_acc))
print("Test F1 = %g " % (d_f1))
```

```
Test Accuracy = 0.613927
Test F1 = 0.572088
```

```
tree = dt_model.stages[5]
print(tree)
```

```
DecisionTreeClassificationModel (uid=DecisionTreeClassifier_f7499507803d) of depth 20 with 41
77 nodes
```

V. Random Forest Classifier

i. Introduction

The Random Forest classifier is based on Decision Trees. However, it makes many important modifications to overcome the drawbacks associated with Decision Trees. A Random Forest is basically an ensemble of Decision Trees. For a binary classification problem, each tree in the forest provides its own predicted class to classify an input and the predicted class with the most votes is chosen for final classification.

The reason that Random Forests work better than Decision Trees is because they use a large number of uncorrelated Decision Trees for classification, and this prevents overfitting. To reduce correlation between different trees, each tree in the forest in trained separately on a random split of the training dataset and with randomly selected input features. By adding this element of randomness to the construction of the decision tree, we prevent the model from overfitting. However, this leads to a small loss in interpretability.

To prevent correlation between trees, Random Forests use two different methods. First, they use a subsample of the training data to train each decision tree. This is also known as bootstrapping. The structure of a decision tree is greatly influenced by the data points that it is trained on. By

training different trees on different samples from the training data we end up with trees with different structures.

Second, they use a random subset of input features for splitting at nodes. As we talked about earlier, when constructing Decision Trees, we pick the best features from the whole set of input features to split a node. However, if we were to follow the same approach when constructing a Random Forest then the strongest input features would be selected by a majority of the trees. This would cause the trees to become correlated. Instead, when constructing a Random Forest, a tree can split from only a random subset of the input features. This means that the strongest input features are not always selected and it forces different trees to learn different aspects of the data. Other than this, the training process remains the same as for Decision Tree classifiers. (Akalin, 2020)

ii. Grid Search

We start by constructing a Random Forest pipeline to perform a grid search for the optimal values of number of trees and maximum depth. The pipeline contains the standard specification of tokenizer, stop words remover, count vectorizer, inverse document frequency, and label indexer. These are implemented in the same way as has been defined in the Elastic Net Logistic Classifier section above. The pipeline also contains a Random Forest Classifier model with an initial maximum depth and number of trees set to 10. We then specify a grid map with different sets of values for the number of trees and maximum depth parameters. The set of values for number of trees ranges from 50 to 250 with a step size of 50. The set of values for the maximum depth parameter ranges from 10 to 30 with a step size of 10. For each set of these parameters, we fit the Random Forest model to our validation data and then evaluate its accuracy using the Multi Class Classification Evaluator. When fitting the model over validation data, we use 80% of the data for training and leave the rest for evaluation.

```
#grid search for the best parameters

#select some initial parameters
num_tree = 10
max_depth = 10

#build the initial grid model
grid_model = RandomForestClassifier(labelCol="label",
                                    featuresCol="features",
                                    numTrees=num_tree,
                                     maxDepth=max_depth)
#create grid pipeline
grid_pipline = Pipeline(stages=[tokenizer, stopwords_rm,
                                count_vec, sparse_vec,
                                label_index, grid_model])
#specify a grid map with different model parameters
grid_builder = ParamGridBuilder().\
    addGrid(grid_model.numTrees, [50, 100, 150, 200, 250]).\
    addGrid(grid_model.maxDepth, [10, 20, 30]).\
    build()

#select the evaluator
evaluator = MulticlassClassificationEvaluator()

#put all the parameters together under the validation model
validation = TrainValidationSplit(estimator=grid_pipline,
                            estimatorParamMaps=grid_builder,
                            evaluator=evaluator,
                            trainRatio=0.8)
```

```
#fit the model to validation data
search_model = validation.fit(val_data)
```

The optimal number of trees is 200 and the optimal maximum depth is 30.

```
#print the parameters of the best fit model

print("Validated number of trees:",search_model.bestModel.stages[-1]._java
_obj.parent().getNumTrees())
print("Validated maximum depth of trees:",search_model.bestModel.stages[-1
]._java_obj.parent().getMaxDepth())
```

```
Validated number of trees: 200
Validated maximum depth of trees: 30
```

iii. Model Fit and Results

Now that we have obtained the optimal parameters using grid search, we use these parameters to fit a Random Forest model to our training data. We build the Random Forest pipeline using the optimal values of number of trees and maximum depth.

```
#set the hyper parameters to the hyperparameters selected by validation

best_numTree = search_model.bestModel.\
               stages[-1]._java_obj.parent().getNumTrees()
best_maxDepth = search_model.bestModel.\
                stages[-1]._java_obj.parent().getMaxDepth()

rd_forest = RandomForestClassifier(labelCol="label",
                                   featuresCol="features",
                                   numTrees=best_numTree,
                                    maxDepth=best_maxDepth)

pipeline = Pipeline(stages=[tokenizer, stopwords_rm,
                            count_vec, sparse_vec,
                            label_index, rd_forest])
```

```
# fit the model to train data
rd_forest_model = pipeline.fit(train_data)
```

We use the fitted Random Forest pipeline to predict the classification labels for our test set tweets and evaluate them. The test accuracy and test F1 score are both around 73%.

```
#accuracy measurement
rd_forest_accuracy = MulticlassClassificationEvaluator(
    labelCol="label", predictionCol="prediction", metricName="accuracy")

rd_forest_f1 = MulticlassClassificationEvaluator(
    labelCol="label", predictionCol="prediction", metricName="f1")

forest_acc = rd_forest_accuracy.evaluate(rd_forest_predictions)
forest_f1 = rd_forest_f1.evaluate(rd_forest_predictions)

print("Test Accuracy = %g " % (forest_acc))
print("Test F1 = %g " % (forest_f1))
```

```
Test Accuracy = 0.732724
Test F1 = 0.73118
```

```
forest = rd_forest_model.stages[5]
print(forest)
```

```
RandomForestClassificationModel (uid=RandomForestClassifier_c2c1b2705c17)
with 250 trees
```

VI. Gradient Boost Tree Classifier

i. Introduction

Gradient Boost Tree Classifier is similar to Random Forests in that it classifies using a group of Decision Trees. However, in case of Gradient Boost the trees are added one-by-one and are not trained at the same time. Due to this, the model is also referred to as the "Multiple Additive Regression model".

This technique is based on the concept of "Boosting". Which means that the model follows an iterative approach where each new tree focuses on classifying the datapoints misclassified by the previous tree. At each iteration the model uses the current set of trees to predict the correct labels for the training data and calculates the log loss function:

$$2\sum_{i=1}^{N}\log\left(1+\exp\left(-2y_iF(x_i)\right)\right)$$

$$where\ N = number\ of\ inputs$$

$$y_i = classification\ label\ for\ input\ i$$

$$x_i = features\ of\ input\ i$$

$$F(x_i) = predicted\ classification\ label\ for\ input\ i$$

Based on the value of this loss function, we perform gradient descent by adding a new tree to the model which reduces the loss. New trees are added till we reach the specified number of maximum iterations. The final classification is made from the weighted sum of individual tree classifications. The weights are shrunk for each tree in the sequence using a learning rate parameter.

ii. Model Fit and Results

We start by constructing a Gradient Boost Tree pipeline to build the model to fit to the training data. The pipeline contains the standard specification of tokenizer, stop words remover, count vectorizer, inverse document frequency, and label indexer. These are implemented in the same way as described in the Elastic Net Logistic Classifier section above. The pipeline also contains a Gradient Boost Tree Classifier (GBT) model with number of iterations/trees set to 20. Gradient Boost Tree classifiers take longer to train compared to Random Forest Classifiers. This is due to two main reasons. First, unlike Random Forests which train different trees in parallel, GBTs build trees sequentially so they can train only one tree at a time. Moreover, Random Forests selects split candidates from a subset of input features and uses a subsample of the training data when training trees. This is in stark contrast to GBT's, who select split candidates from the entire set of input features. As a result of this, it takes a significantly longer time to train GBTs. Over multiple trials we discovered that we were only able to build a GBT with 20 trees within a reasonable computational time.

```
# GBT classifier
from pyspark.ml.classification import GBTClassifier

tokenizer = RegexTokenizer(inputCol="tweet",
                           outputCol="words",
                           pattern="\\W")

stopwords_rm = StopWordsRemover(stopWords=stop_words,
                                caseSensitive=False,
                                inputCol="words",
                                outputCol="filtered")

count_vec = CountVectorizer(minDF=5., vocabSize=2**17,
                            inputCol="filtered",
                            outputCol="vec")

sparse_vec = IDF(inputCol='vec', outputCol="features", minDocFreq=10)

label_index = StringIndexer(inputCol = "target", outputCol = "label")
```

```
#build the pipeline to fit the model
num_tree = 20

gbt_cl = GBTClassifier(labelCol="label", featuresCol="features",
                             maxIter=num_tree)

pipeline = Pipeline(stages=[tokenizer, stopwords_rm,
                             count_vec, sparse_vec,
                             label_index, gbt_cl])
```

```
# fit the model to train data
gbt_cl_model = pipeline.fit(train_data)
```

We use the fitted GBT model to predict classification labels of the test set tweets. The test accuracy is 65% and the test F1 score is 64%.

```
#accuracy measurement
gbt_cl_accuracy = MulticlassClassificationEvaluator(
    labelCol="label", predictionCol="prediction", metricName="accuracy")

gbt_cl_f1 = MulticlassClassificationEvaluator(
    labelCol="label", predictionCol="prediction", metricName="f1")

gbt_acc = gbt_cl_accuracy.evaluate(gbt_cl_predictions)
gbt_f1 = gbt_cl_f1.evaluate(gbt_cl_predictions)

print("Test Accuracy = %g " % (gbt_acc))
print("Test F1 = %g " % (gbt_f1))
```

```
Test Accuracy = 0.659628
Test F1 = 0.646213
```

```
gbt_forest = gbt_cl_model.stages[5]
print(gbt_forest)
```

```
GBTClassificationModel (uid=GBTClassifier_a246aac7b644) with 20 trees
```

VII. Final Results

Overall, we find that Elastic Net Logistic Classifier achieves the highest test accuracy and F1 score at around 76% for both. This is closely followed by the Random Forest Classifier with test accuracy and F1 score of around 73% for both. We notice that, both Random Forest Classifiers and Gradient Boost Tree Classifiers improve testing accuracy as compared to Decision Trees. This indicates that using multiple uncorrelated Decision Trees for classification allows for the model to understand different aspects of the data better which leads to better performance on unseen datasets. GBTs are only able to achieve a test accuracy of 65%. The low performance of GBTs on test data suggests that the specified parameters likely overfit the model. It is possible that the performance can be improved by using grid search to select the appropriate parameter for number of trees. However, this would require either greater computational power or a longer amount of time to train the model (close to a couple of days) on the standard Google cloud dataproc cluster used.

The use of Spark DataFrames allowed us to perform sentiment analysis on a dataset with 1.6 millions tweets relatively fast. Training took the longest time for the Gradient Boost Model (close to 4 hours) while training for the rest of the models took a couple of hours each. This is possible due to the architecture of Apache Spark and its DataFrames API which makes scalable machine learning easy to implement.

```
print("Test Elastic Net Logistic Classifier Accuracy = %g " % (logistic_acc))
print("Test Elastic Net Logistic Classifier F1 = %g " % (logistic_f1))

print("Test Decision Tree Classifier Accuracy = %g " % (d_acc))
print("Test Decision Tree Classifier F1 = %g " % (d_f1))

print("Test Random Forest Classifier Accuracy = %g " % (forest_acc))
print("Test Random Forest Classifier F1 = %g " % (forest_f1))
```

```
Test Elastic Net Logistic Classifier Accuracy = 0.768668
Test Elastic Net Logistic Classifier F1 = 0.768118
Test Decision Tree Classifier Accuracy = 0.613927
Test Decision Tree Classifier F1 = 0.572088
Test Random Forest Classifier Accuracy = 0.732724
Test Random Forest Classifier F1 = 0.73118
```

```
print("Test Gradient Boosted Tree Classifier Accuracy = %g " % (gbt_acc))
print("Test Gradient Boosted Tree F1 = %g " % (gbt_f1))
```

```
Test Gradient Boosted Tree Classifier Accuracy = 0.659628
Test Gradient Boosted Tree F1 = 0.646213
```

References

- Go, A., Bhayani, R., & Huang, L. (2009). Twitter sentiment classification using distant supervision. *CS224N project report, Stanford*, *1*(12), 2009.

- Meng, X., Bradley, J., Yavuz, B., Sparks, E., Venkataraman, S., Liu, D., ... & Talwalkar, A. (2016). Mllib: Machine learning in apache spark. *The Journal of Machine Learning Research*, *17*(1), 1235-1241.

- Spark, A. (2018). Apache spark. *Retrieved January*, *17*, 2018.

- Akalin, A. (2020). *Computational Genomics with R*. CRC Press.