Table of Contents

1. INTRODUCTION 2

1.1 SENTIMENT ANALYSIS 2

2. EXECUTIVE SUMMARY 2

2.1 PROBLEM STATEMENT 2

2.2 APPROACH 4

2.2.1 Random Split 4

2.2.2 Exploratory Data Analysis 4

2.2.2.1 Case 1 4

2.2.2.2 Case 2 5

2.2.2.3 Case 3 6

2.2.2.4 Case 4 7

3. FEATURE ENGINEERING 8

3.1 Term frequency–inverse document frequency (TF-IDF) 8

3.2 Word2Vec 9

4. MODELS FOR TF - IDF 11

4.1 RANDOM FOREST CLASSIFICATION 11

4.2 NAIVE BAYES 11

4.3 LOGISTIC REGRESSION WITH ML 12

5. MODELS FOR WORD2VEC 14

5.1 LOGISTIC REGRESSION WITH LBFGS 14

5.2 K MEANS CLUSTERING 14

6. COMPARISON 15

6.1 TF - IDF 15

7. CONCLUSION 16

# 1. INTRODUCTION

In this project we will study the basics of sentiment analysis with Natural Language Processing techniques using both spark and gensim. We will try to build a predictive model for a real-world dataset and provide results and recommendations. We begin by looking at the basics of

## 1.1 SENTIMENT ANALYSIS

Sentiment Analysis aims to determine the attitude of the speaker or a writer with respect to some topic using NLP. A basic task in this analysis is classifying the polarity of a given text at the document, sentence or feature/aspect level - whether the expressed opinion in a document, sentence or an entity feature/aspect is positive or negative.

The rise of social media such a blogs and social networks have fueled interest in sentiment analysis. With the proliferation of reviews, ratings, recommendations etc, there is a great opportunity for businesses to identify new opportunities and manage their reputations.

NLP comprises of making the machine understand how humans speak, write and communicate. Humans communicate or convey messages in structured pattern, unstructured pattern, contain regional slangs and idioms. NLP tries to bridge the gap in communication. As our project deals with sentimental analysis of movie reviews we will be using NLP to build different models like TF –IDF and Word2Vec which will predict the polarity of the review.

# 2. EXECUTIVE SUMMARY

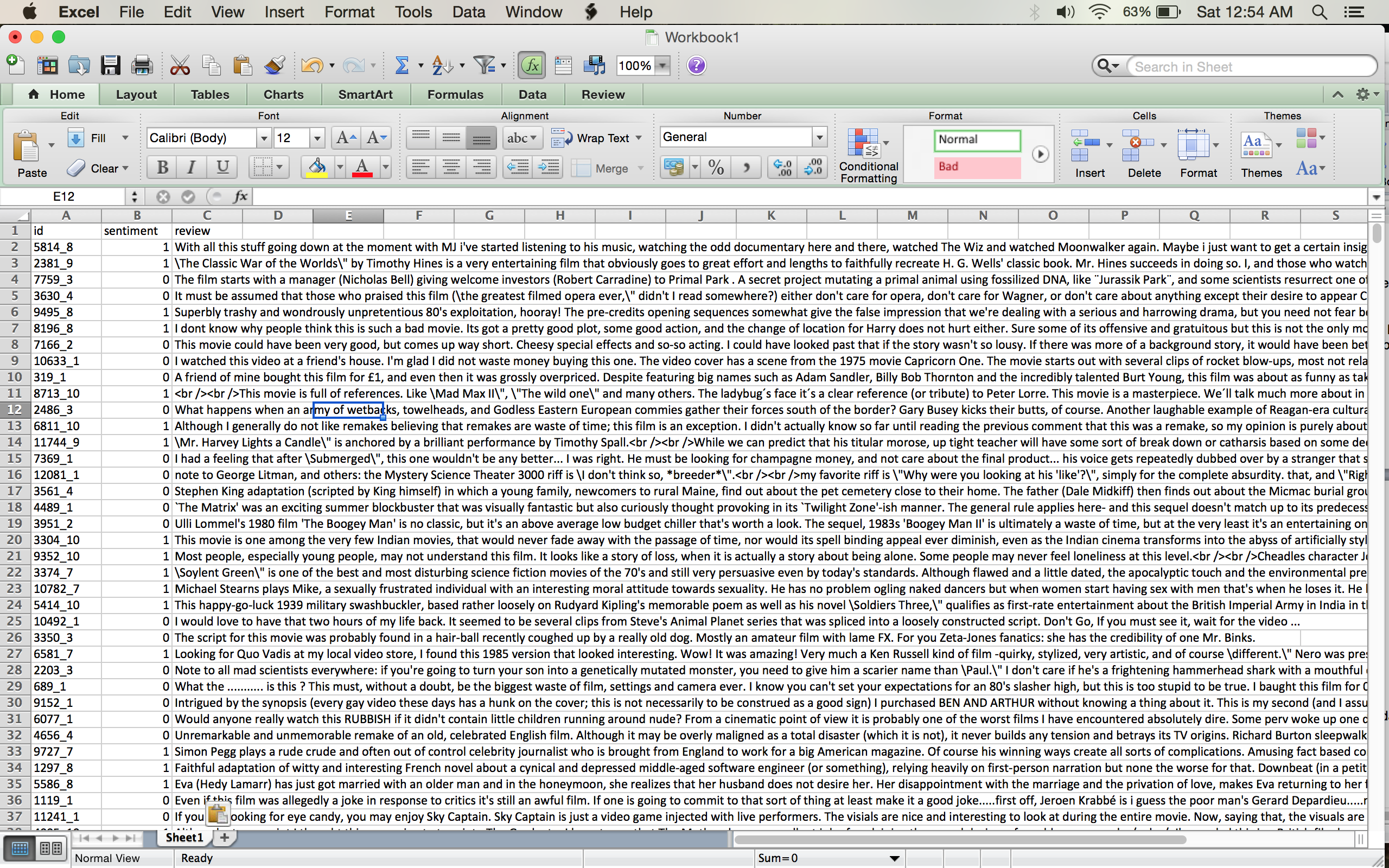
## 2.1 PROBLEM STATEMENT

The given dataset provides the 25000 reviews of movies on IMDB. The sentiment is associated with each review as 0(thumbs down) or 1(thumbs up).

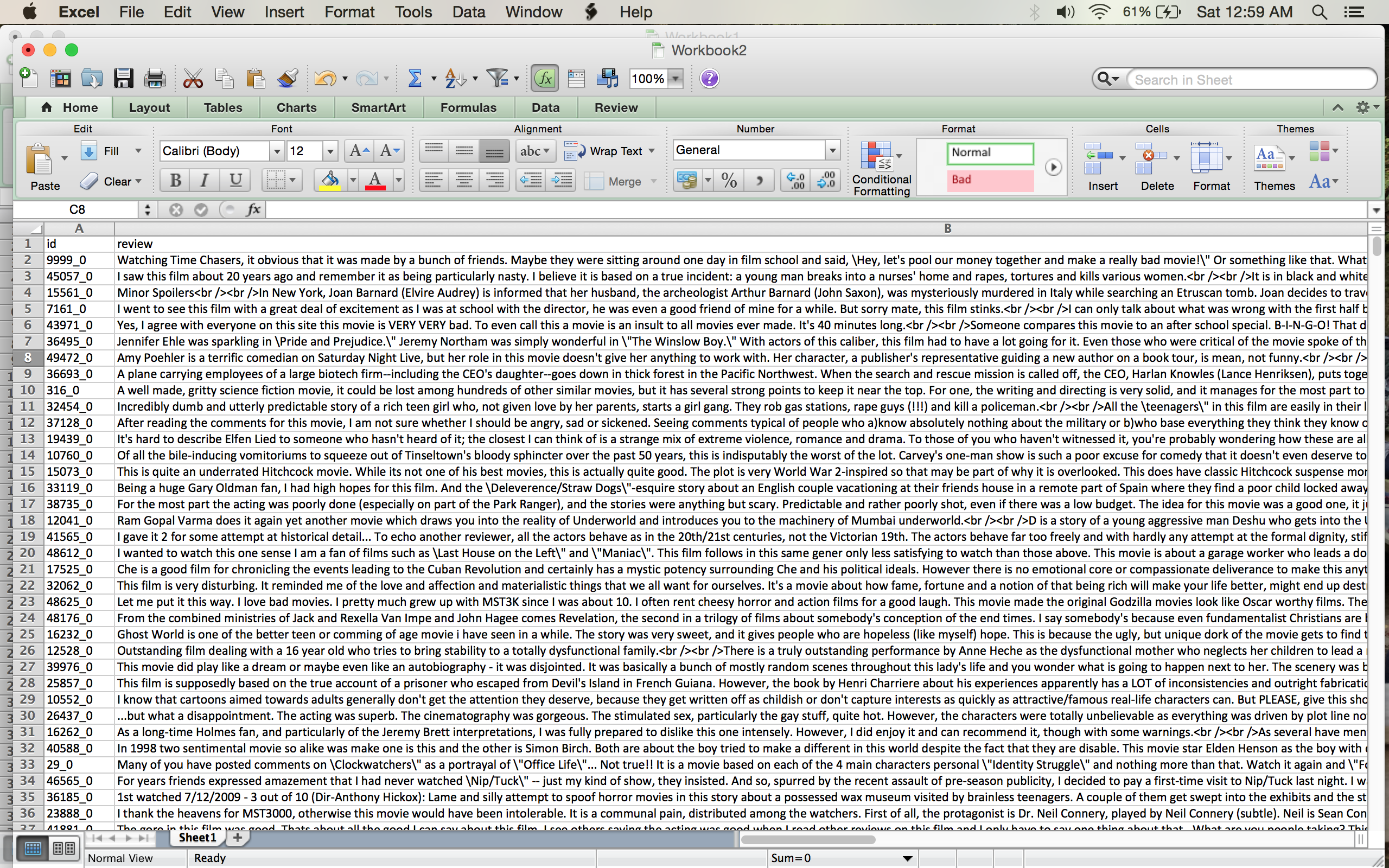
Our approach aims to apply simple TF-IDF as well as Word2Vec models to try and build a system, which can predict the polarity of the review as thumbs down (0) or thumbs up (1).

**Dataset Screenshot:**

**Labeled TrainData**



**UnLabeled TrainData**

****

## 

## 2.2 APPROACH

## 2.2.1 Random Split

We used random split to split the data into train data and test data so that the split data could be used for training the model and for testing the model.

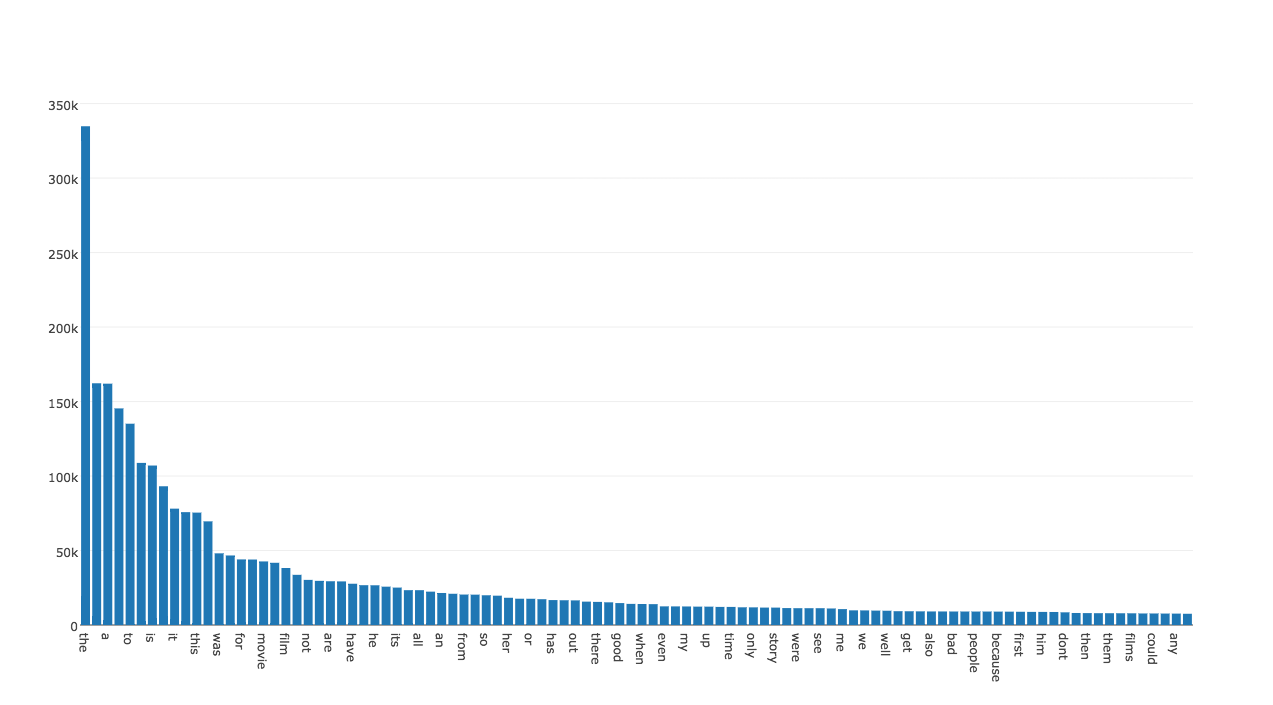
## 2.2.2 Exploratory Data Analysis

First the dataset is looked for irregularities. In our dataset the reviews will have special characters and html tags that needs to be removed as a part of data preprocessing step. We remove those html tags using The Beautiful Soup library. To remove the punctuation, Numbers we used NLTK.

Following are the plots of few data analysis

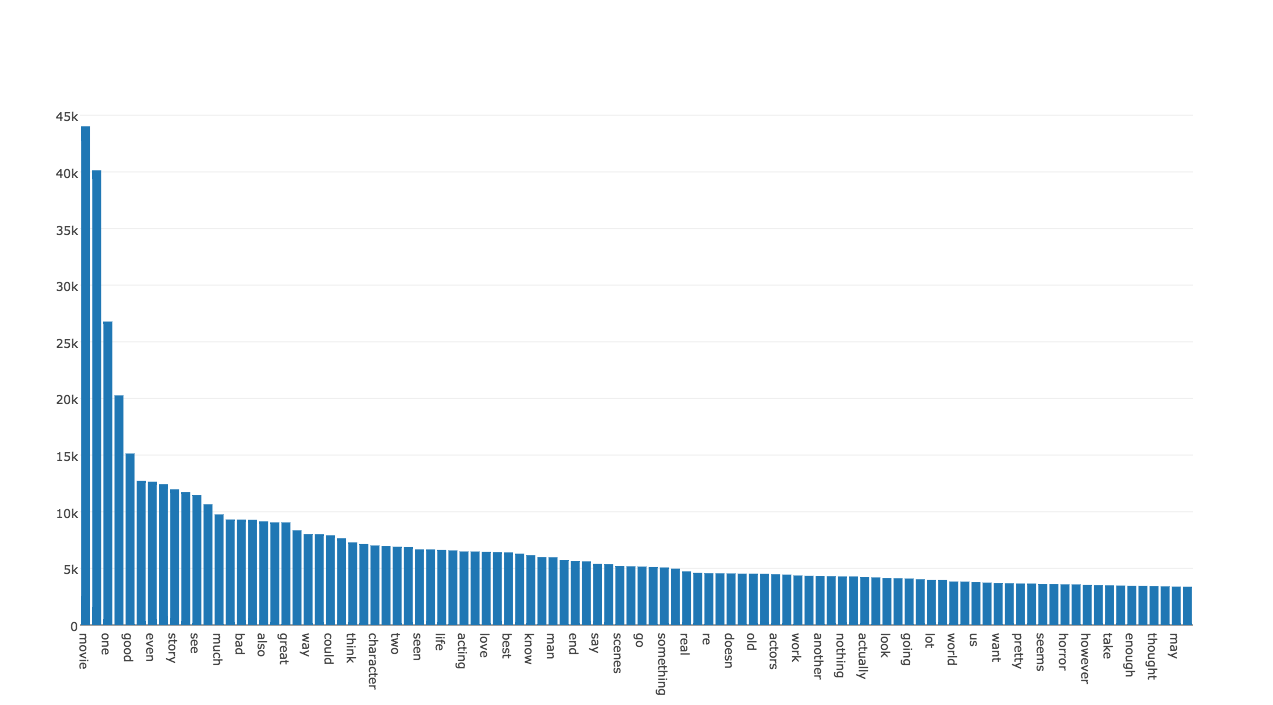
## 2.2.2.1 Case 1

The following plot tells about the word count in the dataset that is a very naive word count:



## 2.2.2.2 Case 2

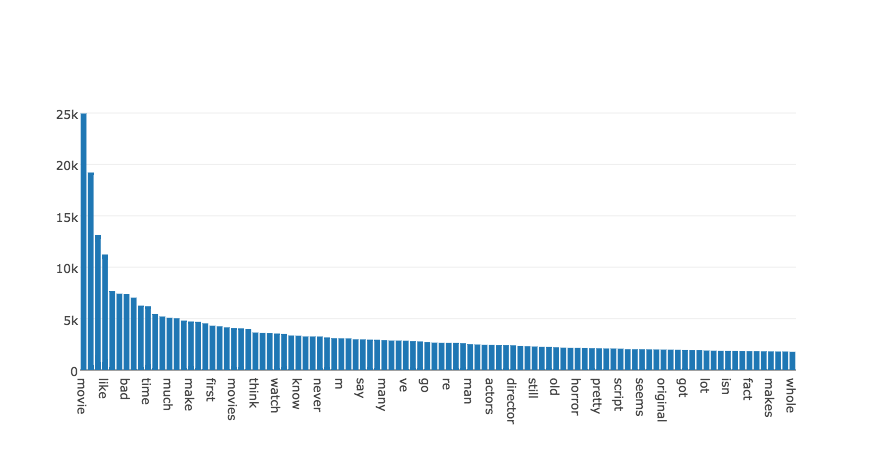
In the previous plot the words like: the, of, is, I etc show up, which are not really useful. So we are using NLTK (Natural Language ToolKit) which will remove words like the, of, is and keep words that don’t repeat often. After using NLTK the data analysis of the reviews is as follows:



The plot has the following words - movie, like, even, really, much, people, great, make, think, watch

## 2.2.2.3 Case 3

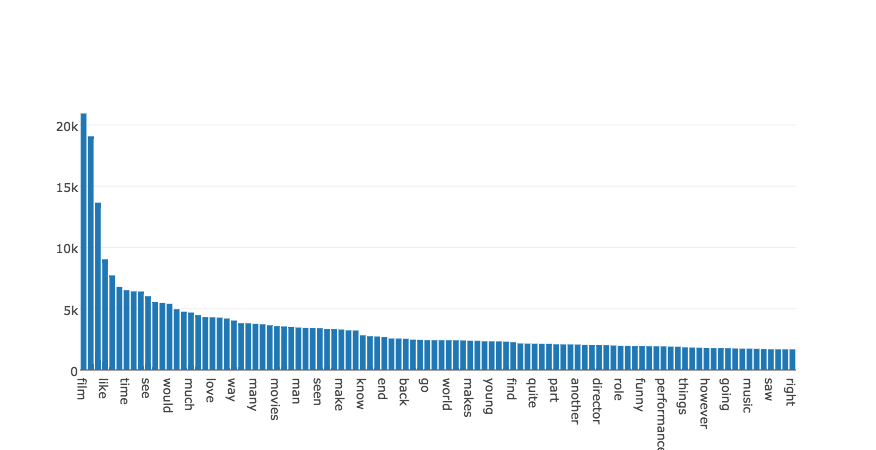
Below is the plot of negative reviews which tells about the negative word count



And the plot has words like bad, never, old, horror etc.

## 2.2.2.4 Case 4

Below is the plot of positive reviews which tells about the positive word count



And the plot has words like love, funny, right etc.

# 3. FEATURE ENGINEERING

Feature engineering is the process of creating ‘features’ that could be used in machine learning algorithms. For the purpose of sentiment analysis, the text that we use to train and predict has to be converted to usable and efficient numerical vectors to be used in the classification algorithms. Unlike other process, for text analysis, the assigning of vectors cannot be done manually and natural language processing (NLP) has to be used. For this specific problem, we used two different techniques for feature engineering, with varied results.

## 3.1 Term frequency–inverse document frequency (TF-IDF)

Term frequency–inverse document frequency (TF-IDF), is a numerical statistic that is intended to reflect how important a word is to a document in a collection or corpus. TF-IDF is available as part of pyspark.ml.features package and helps in converting the text to vectors depending on the occurrence and repetition of the word. Before using TF-IDF, The data has to be cleaned. The first step in cleaning the text is the removal of HTML tags from the text. For this purpose, we use a package called Beautiful Soup available in python. The next step would be removed the punctuations and non-alphabetical characters, these characters will affect the efficiency of the created vectors. This is done using regular expressions by replacing anything that is not alphabetical with a space.

Every sentence in a spoken language has words, which repeat frequently. In the English language it would be the preposition, articles, pronouns etc. These words are called stopwords in natural language processing. It is important we remove these words before the features could be created. We use the package called nltk for this purpose. We download the stop words and remove those words from our text by looping through it.

Once we have reached this state, the next step would be to tokenize the text into words. This is done using the tokenizer function available in pyspark.ml.features. Once the tokenization is done, we use a hashing function to convert the words into numerical values for the creation of TF-IDF model. The converted vectors are then fed to the IDF class for fitting. Once the model is fitted, it can be used for transforming our hashed vales to vectors. The IDF model on transformation gives a SparseVector of features. These features can be used in the ml algorithm. The IDF model also gives us options to choose the number of features we would want to use.

We tried the IDF model for 50 features and 300 features. On evaluation of the classification model, the model with 300 features gave an area under ROC value of about 80% whereas the model with 50 features performed under an area under ROC of 67%.

The steps from text to vectors using TF-IDF are given below.

## 3.2 Word2Vec

The Word2Vec also converts the words to vectors, but taking into consideration the semantics and grammar of the language used. Word2Vec is also available in the pyspark.ml.feature package. The Word2Vec works by creating a model with the language training data to learn the semantics of the language used in processing. This requires a lot of text for training and it is to be noted that this training data need not be labeled for the sentiment it represents. So, we used the unlabeled reviews that were available for this process. The unlabeled data also goes through the process of cleaning. Beautiful Soup and regular expressions are used to remove html tags and non-alphabetic characters in the text.

For the word2vec model to be fitted it is important that this data has to be in the form of sentences instead of words. English language has various ways in which its sentences would be ended. To create sentences from paragraphs, we make use of yet another feature from the nltk package called punkt. This punkt is fed to a tokenizer and the reviews are tokenized into sentences. This is then fed to the Word2Vec model for fitting. Once our model is fitted, we can use it to train our actual training data for the machine leaning algorithm. This data is also cleaned for html tags and non-alphabets, but in both the word2vec training data and the ml training data is not treated for stop words, as the stop words are integral part of language semantics. We were able to attain an area under ROC of about 89% with the vectors created using the word2Vec model (300 features)

The process diagram is as follows

# 4. MODELS FOR TF - IDF

## 4.1 RANDOM FOREST CLASSIFICATION

Random forests train a set of decision trees separately, so the training can be done in parallel. The algorithm injects randomness into the training process so that each decision tree is a bit different. Combining the predictions from each tree reduces the variance of the predictions, improving the performance on test data.

The randomness injected into the training process includes:

* Subsampling the original dataset on each iteration to get a different training set (a.k.a. bootstrapping).
* Considering different random subsets of features to split on at each tree node.

Apart from these randomizations, decision tree training is done in the same way as for individual decision trees.

**Random Forest Classification Algorithm**

stringIndexer = StringIndexer(inputCol="label", outputCol="indexed")

si\_model = stringIndexer.fit(selectData)

td = si\_model.transform(selectData)

rfc = RandomForestClassifier(maxDepth=2, labelCol="indexed")

pipeline = Pipeline(stages=[tokenizer, hashingTF,idf,stringIndexer , rfc])

paramGrid = ParamGridBuilder().addGrid(hashingTF.numFeatures, [300, 400]).addGrid(rfc.maxDepth, [2, 5, 10]).build()

cv = CrossValidator().setNumFolds(3).setEstimator(pipeline).setEstimatorParamMaps(paramGrid).setEvaluator(BinaryClassificationEvaluator())

cvModel = cv.fit(pipelineTrainingData)

testTransform = cvModel.transform(pipelineTestData)

predictions = testTransform.select('review', 'label', 'prediction')

predictionsAndLabels = predictions.map(lambda x : (x[1], x[2]))

trainErr = predictionsAndLabels.filter(lambda r : r[0] != r[1]).count() / float(testData.count())

print("TrainErr: "+str(trainErr))

BinaryClassificationEvaluator().evaluate(testTransform)

## 4.2 NAIVE BAYES

[Naive Bayes](http://en.wikipedia.org/wiki/Naive_Bayes_classifier) is a simple multiclass classification algorithm with the assumption of independence between every pair of features. Naive Bayes can be trained very efficiently. Within a single pass to the training data, it computes the conditional probability distribution of each feature given label, and then it applies Bayes’ theorem to compute the conditional probability distribution of label given an observation and use it for prediction.

These models are typically used for [document classification](http://nlp.stanford.edu/IR-book/html/htmledition/naive-bayes-text-classification-1.html). Within that context, each observation is a document and each feature represents a term whose value is the frequency of the term (in multinomial naive Bayes) or a zero or one indicating whether the term was found in the document (in Bernoulli naive Bayes). Feature values must be nonnegative. The model type is selected with an optional parameter “multinomial” or “bernoulli” with “multinomial” as the default. [Additive smoothing](http://en.wikipedia.org/wiki/Lidstone_smoothing) can be used by setting the parameter λ (default to 1.0). For document classification, the input feature vectors are usually sparse, and sparse vectors should be supplied as input to take advantage of sparsity. Since the training data is only used once, it is not necessary to cache it.

**Naive Bayes Algorithm**

lp = selectData.map(lambda x : LabeledPoint(x.label,x.features))

(trainingData, testData) = lp.randomSplit([0.6, 0.4])

nb = NaiveBayes.train(trainingData,1.0)

pipeline = Pipeline(stages=[tokenizer, hashingTF,idf, nb])

model = pipeline.fit(trainingData)

selected = model.transform(testData).select('review', 'label', 'prediction')

# Build a parameter grid.

paramGrid = ParamGridBuilder().addGrid(hashingTF.numFeatures, [300, 400]).addGrid(nb.regParam, [0.01, 0.1, 1.0]).build()

#Set up cross-validation.

cv = CrossValidator().setNumFolds(3).setEstimator(pipeline).setEstimatorParamMaps(paramGrid).setEvaluator(BinaryClassificationEvaluator())

#Fit a model with cross-validation.

cvModel = cv.fit(trainingData)

testTransform = cvModel.transform(testData)

predictions = testTransform.select("review", "prediction", "label")

predictionsAndLabels = predictions.map(lambda x : (x[1], x[2]))

trainErr = predictionsAndLabels.filter(lambda r : r[0] != r[1]).count() / float(testData.count())

print("TrainErr: "+str(trainErr))

BinaryClassificationEvaluator().evaluate(testTransform)

## 4.3 LOGISTIC REGRESSION WITH ML

Logistic Regression is a regression model where the nominal variable is categorical. Use Binary Logistic regression when you have one nominal variable with two values like male/female or good/bad and one measurement variable, which is an independent variable. Use Multiple Logistic regression when you have one nominal variable and more than one measurement variable. The goal of logistic regression is to predict the probability of getting the desired nominal value given the measurement value and also to predict that getting a particular nominal value is dependent on the measurement value.

**Logistic Regression Algorithm**

(trainingData, testData) = selectData.randomSplit([0.6, 0.4])

lr = LogisticRegression(maxIter=5, regParam=0.01)

model = lr.fit(trainingData)

result = model.transform(testData)

evaluator = BinaryClassificationEvaluator()

evaluator.evaluate(result, {evaluator.metricName: "areaUnderPR"})

evaluator.evaluate(result, {evaluator.metricName: "areaUnderROC"})

Here we do a random split of the data that is available to us. We train the model with 60 percent of data and test the model with 40 percent of the data. We set the maximum no of iterations as 5 and the regularization parameter to 0.01. We fit the model according to the training data. Then the test data is transformed based on the model. We predict whether the sentiment of the review is 1 (good) or 0 (Bad). AreaUnderROC values plots false positive rate against true positive rate. So higher the auROC value better the model.

**CROSS VALIDATOR**

An important task in ML is *model selection*, or using data to find the best model or parameters for a given task. This is also called *tuning*.Pipelines facilitate model selection by making it easy to tune an entire Pipeline at once, rather than tuning each element in the Pipeline separately.

Currently, spark.ml supports model selection using the [CrossValidator](http://spark.apache.org/docs/latest/api/scala/index.html#org.apache.spark.ml.tuning.CrossValidator) class, which takes an Estimator, a set of ParamMaps, and an [Evaluator](http://spark.apache.org/docs/latest/api/scala/index.html#org.apache.spark.ml.Evaluator). CrossValidator begins by splitting the dataset into a set of *folds* which are used as separate training and test datasets; e.g., with k=3 folds,CrossValidator will generate 3 (training, test) dataset pairs, each of which uses 2/3 of the data for training and 1/3 for testing. CrossValidator iterates through the set of ParamMaps. For each ParamMap, it trains the given Estimator and evaluates it using the given Evaluator. The ParamMap that produces the best evaluation metric (averaged over the k folds) is selected as the best model. CrossValidator finally fits the Estimator using the best ParamMap and the entire dataset.

**CROSS VALIDATOR CODE**

# Build a parameter grid.

paramGrid=ParamGridBuilder().addGrid(hashingTF.numFeatures,[300, 400])

.addGrid(lr.regParam, [0.01, 0.1, 1.0]).build()

#Set up cross-validation.

cv=CrossValidator().setNumFolds(3).setEstimator(pipeline).setEstimatorParamMaps(paramGrid).setEvaluator(BinaryClassificationEvaluator())

#Fit a model with cross-validation.

cvModel = cv.fit(trainingData)

testTransform = cvModel.transform(testData)

predictions = testTransform.select("review", "prediction", "label")

predictionsAndLabels = predictions.map(lambda x : (x[1], x[2]))

trainErr = predictionsAndLabels.filter(lambda r : r[0] != r[1]).count() / float(testData.count())

print("TrainErr: "+str(trainErr))

BinaryClassificationEvaluator().evaluate(testTransform)

For our model we have decided to go with 300 features to run the cross validator class.

# 5. MODELS FOR WORD2VEC

## 5.1 LOGISTIC REGRESSION WITH LBFGS

Logistic Regression is a regression model where the nominal variable is categorical. Use Binary Logistic regression when you have one nominal variable with two values like male/female or good/bad and one measurement variable, which is an independent variable. Use Multiple Logistic regression when you have one nominal variable and more than one measurement variable. The goal of logistic regression is to predict the probability of getting the desired nominal value given the measurement value and also to predict that getting a particular nominal value is dependent on the measurement value.

**Logistic Regression Algorithm Code**

**RESULT:**

## 5.2 K MEANS CLUSTERING

[k-means](http://en.wikipedia.org/wiki/K-means_clustering) is one of the most commonly used clustering algorithms that clusters the data points into a predefined number of clusters. The MLlib implementation includes a parallelized variant of the [k-means++](http://en.wikipedia.org/wiki/K-means%2B%2B) method called kmeans. The implementation in MLlib has the following parameters:

* k is the number of desired clusters.
* Max iterations are the maximum number of iterations to run.
* Initialization Mode specifies either random initialization or initialization via k-means||.
* Runs are the number of times to run the k-means algorithm (k-means is not guaranteed to find a globally optimal solution, and when run multiple times on a given dataset, the algorithm returns the best clustering result).
* Initialization Steps determines the number of steps in the k-means|| algorithm.
* Epsilon determines the distance threshold within which we consider k-means to have converged.

CODE

RESULT

# 6. COMPARISON

## 6.1 TF - IDF

We did run the above said three models for TF – IDF and the results are listed below

Logistic Regression was better of all three because of the ML implementation with cross validator.

**FEATURE SELECTION:**

Based on the results we decided to run the Logistic Regression With LBFGS of mllib because the results were better in Logistic Regression with ML implementation.

CODE for LBFGS

Result

# 7. CONCLUSION

Logistic Regression with ML gave better result in TF – IDF. So we decided to implement Logistic Regression with LBFGS in ml. Word2Vec is a deep learning model that is better than TF – IDF. Since we got better result for Logistic regression with LBFGS in TF – IDF we decided to implement Logistic Regression with LBFGS in Word2Vec, which gave better result as expected than every other model.