Predicting the variety of wine using Natural Language Processing in Big Data Environment

MIE 1628H: Big Data Science

Course Instructor: Prof. Yanina Shevchenko

Shabari Girish

Student Number: 1005627644

University of Toronto

School of Mechanical and Industrial Engineering

1. Executive Summary

The objective of this project was to predict the wine variety using Natural Language Processing (NLP). Kaggle Wine Dataset was used by the supervised machine learning algorithms to predict the target variable- wine variety, based on the engineered features. A comprehensive comparative analysis was done by implementing the multi-class classification machine learning algorithms such as the Logistic Regression (one-vs-rest), Decision Trees, Random Forest. Furthermore, each model was tuned to obtain the best parameters using cross-validation and hyper-parameter tuning. The models were evaluated using multi-classification evaluators such as the accuracy, weighted precision, recall and f1-score, ROC curve, AUC as well as a confusion matrix. A winning model was chosen based on the accuracy of each model. Lastly, challenges, recommendation and future scope of work are highlighted.

2. Literature Review

Natural Language Processing in the field of AI that aids computers understand, interpret and manipulate human language [1]. While NLP is not a new domain, several studies have focused on improving the systems' ability to process language. This gets even better and lucrative with the researchers armed with the Big Data tools which enable processing in parallel environments. In the 1950s and 1960s, the direct word-for-word replacement was popular for machine translation. Later, researchers started experimenting with different text vectorization techniques such as the Term Frequency (TF), n-grams, Term Frequency-Inverse Document Frequency and Words2Vec etc. as well stemming and lemmatization techniques to accurately process natural language. Brochet and Dubourdieu (2001) conducted a lexical analysis of four corpora of wine reviews from a cognitive linguistic perspective and concluded that wine reviews are not only describing sensory properties of the wine, bit also includes idealistic and hedonistic information from the wine prototypes based on previous experience [3]. However, most of the work to date consisted of processing of the language on a single edge node, which poses constraints in terms of scalability and processing. Thus, in this project, the NLP was implemented in Scala and PySpark and deployed on Microsoft Azure to ensure parallel processing in a big data environment without any scalability constraints.

3. Business Problem

There are lots of unstructured textual data produced on the web pertinent to wine. The ability to process this data and predict the wine variety can help businesses stay ahead of their competitors by understanding the trend in the market and acting accordingly to cater to the market’s demand.

4. Dataset

The dataset used for this problem was the ‘Kaggle Wine Dataset’. which consisted of about 130,000 observations with 702 wine varieties. However, 610 wine varieties had less than 100 wine reviews. Thus, the dataset was imbalanced as shown in figure 2:

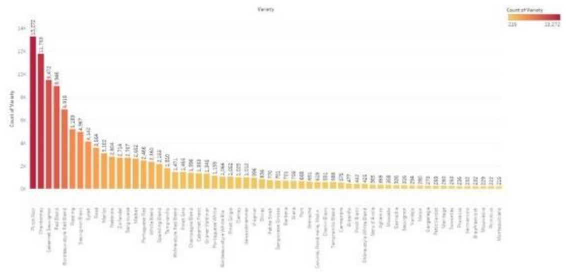

Figure 1: Number of Records per wine variety (65)

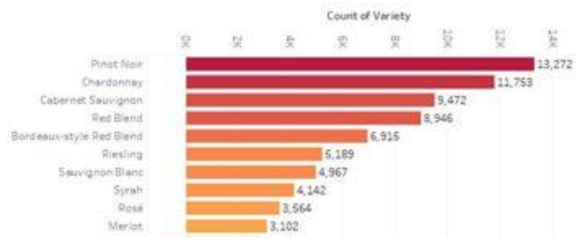

Figure 2: Number of Records per top 10 wine variety

Figure 3: Word cloud of records among countries

Figure 4: Distribution of wine varieties among countries

As per the business problem elucidated above, only the top wine varieties are of interest to the businesses. Accordingly, in this project, a sample consisting of top 10 wine varieties was considered which accounted for a total of about 71,000 observations as shown in figure 1.

5. Target Variable and Machine Learning approach

The dataset used for this problem was labeled. Thus, supervised machine learning models were used to train them based on the training data. Since the business problem was to be able to predict the wine variety based on the description, wine variety was chosen as the target variable. Given that there were multiple classes of wine to be predicted, this becomes a multi-class classification problem.

6. Methodology- Machine Learning Workflow

The Machine Learning workflow architecture used for this problem is illustrated in figure 5:

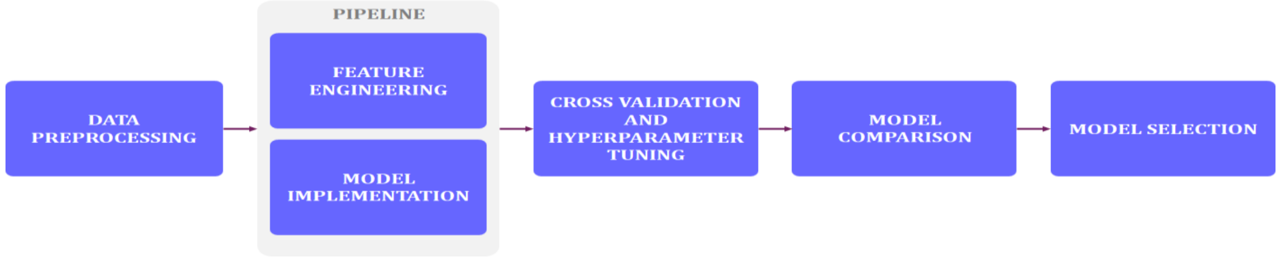

Figure 5: Machine Learning Workflow

The data was preprocessed to get rid of the null values in the dataset. Later, a pipeline consisting of feature engineering as well as the model implementation was developed. In order to ensure the optimal performance of each model, they were tuned using cross-validation and hyperparameter tuning. Eventually, a comparative analysis was performed to obtain a winning model. Each of these milestones is mentioned in detail below.

6.1. Data Pre-processing and Exploratory Analysis

Data pre-processing in the imperative to the model as it involves cleaning and transformation of the data required by the machine learning model. The dataset used for this problem had null values in sundry features as shown in table 1.

Country	Description	Designation	Points	Price	Province	Region 1	Region 2	Taster	Taster Twitter	Title	Variety	Winery
26	0	22214	0	4088	26	10092	34659	14781	18264	0	0	0

Table 1: Count of the Null Values in the Dataset

The correlation matrix was used to identify the important features in predicting the wine variety. Accordingly, it was obtained that the features such as the designation, twitter handle, and title were deleted. An understanding of the data is required to accurately fill the null values. Thus, we resorted to exploratory analysis for the same.

6.2 Pipeline Development

The pipeline framework was adopted to ensure the step-by-step implementation of the ML workflow with the input data. It consisted of two major components: Feature Engineering and Model Implementation.

6.2.1 Feature Engineering

The features were engineered for the input to the ML models as follows:

1. **Concatenation:** String fields such as the description, country, taster name, region was concatenated to be vectorized later.
2. **Tokenization:** Conversion of all strings into words called tokens
3. **Stemming**: Reduce the tokens (words) to its root stem
4. **Stop words:** Filter out words that have no statistical significance
5. **TF-IDF Vectorizer:** Rate the importance of a word to the document. **TF-IDF has better prediction capabilities as compared to that of the models based on the frequency of the words or n-gram model in the NLP domain.**
6. **Normalizing:** The number features such as price and point were normalized to remove bias
7. **Vector Assembler:** Combine normalized features with the TF-IDF vectorized features
8. **Principal Component Analysis:** Reduce dimensionality to point in the direction with maximum variance for the model to discern clearly between the classes

6.2.2 Model Implementation

All the multi-class classification ML models available in the ML.lib package was implemented to ensure working in the big data environment and avoid running the models on a single edge node as it does with sklearn in python. This **governed the choice of our model**. Below are the ML multi-class classification models available in Scala:

1. Logistic Regression (one-vs-rest)
2. Decision Tree
3. Random Forest

6.3 Cross-Validation and Hyperparameter tuning

With the aim of obtaining optimal performance of each model three-fold cross-validation with hyperparameter tuning was implemented.

6.4 Model Comparison and Results Analysis

6.4.1 Selecting Evaluation Metrics

The selection of apt evaluation metrics plays a vital role in benchmarking the ML models. Given the business problem, there was no cost associated with false positive and false negatives. Moreover, after filtering for the top 10 wine varieties, the dataset was balanced which made **accuracy** as the rightful choice for the evaluation metrics. However, other metrics such as the **weighted precision, recall and f1-score, ROC curve, AUC as well as confusion matrix.**

6.4.2 Results Analysis and Discussion:

Initially, a conservative approach was used wherein the dimensionality was reduced to 50 features. The models showed an improvement in the accuracy on an average by 8-12% after cross-validation and hyperparameter tuning as shown in Tables 2 & 3.

		PCA 100	PCA 500
Logistic Regression	Train	0.74332	0.76827
	Test	0.73691	0.79733
Decision Trees	Train	0.53322	0.53282
	Test	0.53252	0.53176
Random Forest	Train	0.56427	0.48387
	Test	0.56413	0.47945

	Cross-Validation		Parameters
Logistic Regression	Train	0.79733	regParam (0,0.1,0.5) elasticNetParam(0.1,0.3,0.5) maxIter (3,5)
	Test	0.77128	
Decision Trees	Train	0.57416	maxBins (25,35) max depth (5,7) impurity (Gini, Entropy)
	Test	0.53176	
Random Forest	Train	0.56716	maxBins (25,35) max depth (5,7) impurity (Gini, Entropy)
	Test	0.55289	

Tables 2 & 3: Model Results Comparison

Naïve Bayes algorithm was implemented and found that it gave the worst results because it assumes features as independent to predict the outcome. Thus, it was not considered further.

The dimensionality was increased from 100 to 500 for two reasons:

i. The optimal regularization parameter for OVR was 0, thus the model had no overfitting
ii. The risk of over-fitting was low given 71,000 observations

The results for PCA 500 showed over-fitting with decision tree as its training accuracy increased but the testing accuracy was compromised. OVR was the best performance with no regularization, thus OVR was chosen and optimized further.

6.4.3 Model Feature Importance:

Since PCA was used in the model pipeline, the model transparency was lost. PCA results in the new vectors pointing in the direction with the highest variance and these vectors are the result of all the features combined. Thus, the downside of using PCA is that model feature importance (i.e. the words in this case) cannot be determined.

6. 5 Model Selection

Since the accuracy of the Logistic Regression (OVR) was just 0.4% less than the ensemble, OVR was selected as the final model weighing model accuracy, complexity and computation time. Since OVR was selected as the best model, it was scrutinized further with the evaluation metrics such as the weighted precision, recall and f1-score, ROC curve, AUC as well as confusion matrix. These results are highlighted below.

Metrics	PCA 500
Train Accuracy	0.79733
Test Accuracy	0.76827
Precision	0.76751
Recall	0.76827
F1 Score	0.76526

Table 4: OVR Weighted Precision, Recall, F1 score, and Accuracy

The performance of the one class as compared to that of the other can be obtained from the confusion matrix and the ROC curve which is shown in Figures 7 and 8.

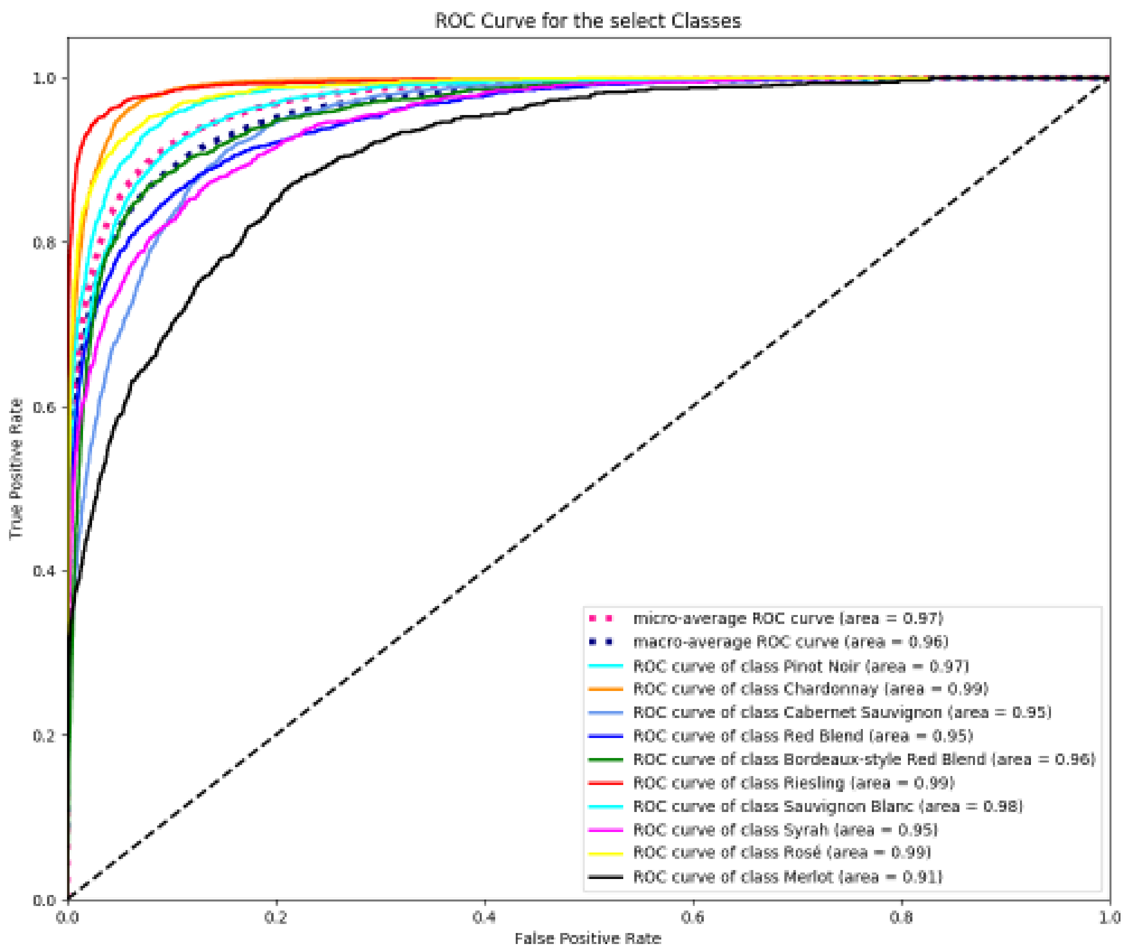

Figure 6: ROC Curve

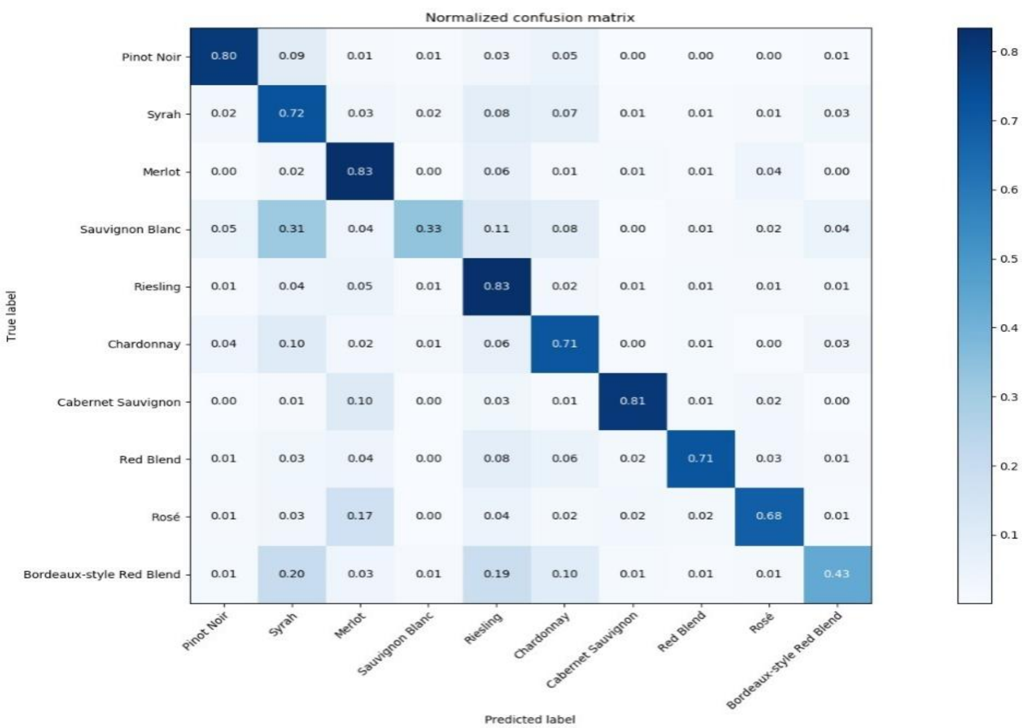

Figure 7: Confusion Matrix

The model has a very high rate of true positive with an average AUC for each class of about 0.9, this dictates the performance of the model as seen in figure 9. Moreover, from the confusion matrix in figure 10, most of the classes were predicted with about 85% accuracy and only two classes which were predicted with the accuracy of about 40%.

7. Challenges

i. Limited library in Scala for multi-class classification- This is inevitable as we wanted to run our models in Big Data environment and didn't want to end up using sklearn, running models on a single edge node
ii. Resources available with the Databricks community edition as well as Queen's cluster were not enough to train the model. We deployed our model on Microsoft Azure to scale the cluster to 54Gb for the worker node
iii. Runtime errors which were tackled with the support documentation for spark

9. Further Scope of Improvement

The model can be improved further as follows:

i. Using **Bag of Words** to improve the vocabulary of the machine. The current model used unigram, with bigram additional features can be introduced to improve the model accuracy
ii. Using **Multilayer Perceptron** Model- Neural Nets with Scala

iii. Avoiding dimensionality reduction and using **feature importance** function in trees
iv. Attempting **non-linear transformation** for dimensionality reduction such as TSNE algorithm
v. Using **Stratified Splitting** to increase homogeneity in the population
vi. Using different vectorizers such as Word2Vec and Count Vectorizer

10. Contribution

With the previous experience in Data Analytics and Machine Learning, I was able to contribute to Data Cleaning, Feature Engineering, Pipeline Development, Model implementation - Decision Trees and Random Forest Model. Our team discussed the model results, compared them and brainstormed apt model improvement scope and eventually selecting a winning model.

11. References

1. Natural Language Computing, CSC401/2511, University of Toronto
2. Frederic Brochet and Denis Duourdieu, 2001. Wine descriptive language supports the specificity of chemical senses. *Brain and Language,* 77:187-196
3. https://monkeylearn.com/text-classification/
4. https://www.kaggle.com/zynicide/wine-reviews
5. https://towardsdatascience.com/multi-class-text-classification-with-scikit-learn-12f1e60e0a9f
6. https://spark.apache.org/docs/latest/mllib-dimensionality-reduction
7. https://medium.com/rahasak/random-forest-classifier-with-apache-spark-c63b4a23a7cc
8. https://www.programcreek.com/scala/org.apache.spark.ml.feature.PCA

Visualizations : Tableau, Microsoft PowerBI, Scala

12. Appendix

Mentioned is the link to access the final code on Data Bricks:

https://databricks-prod-cloudfront.cloud.databricks.com/public/4027ec902e239c93eaaa8714f173bcfc/4644463000288664/60553195985253/6893616237043114/latest.html

Code:

```
import org.apache.spark._

import org.apache.spark.sql.expressions.Window


//import libs

import org.apache.spark.ml.Pipeline

import org.apache.spark.ml.tuning.{CrossValidator, ParamGridBuilder}

import org.apache.spark.sql.Row

import org.apache.spark.mllib.evaluation.BinaryClassificationMetrics

import org.apache.spark.ml.linalg._

import org.apache.spark.sql.expressions.Window

import org.apache.spark.ml.feature._

import spark.sqlContext.implicits._

import org.apache.spark.sql.functions._

import org.apache.spark.ml.tuning._

import org.apache.spark.ml.param.ParamMap

import org.apache.spark.ml.evaluation.BinaryClassificationEvaluator

import org.apache.spark.mllib.evaluation.MulticlassMetrics

import org.apache.spark.ml.classification._

import org.apache.spark.ml.evaluation._

import org.apache.spark.mllib.classification.LogisticRegressionWithLBFGS

import org.apache.spark.mllib.util.MLUtils
```

Show result

```
%python


!pip install nltk

import nltk

nltk.download("all")

from nltk.stem import SnowballStemmer

import pyspark.sql.functions as F

from pyspark.sql import types

from pyspark.sql import Row
```

Show result

Select Most Relevant and Top Classes

```
val DF = spark.sql("select * from winemag")

val window = Window.partitionBy("variety")

val WineDF = DF.withColumn("frequency", count("variety").over(window))
        .orderBy(desc("frequency"))
        .where(col("frequency")>3000) //Threshold set at 2000(need to
justify)

//WineDF.count()
```

```
DF: org.apache.spark.sql.DataFrame = [_c0: string, country: string ... 12 mor
e fields] window: org.apache.spark.sql.expressions.WindowSpec = org.apache.sp
ark.sql.expressions.WindowSpec@4c2a0466 WineDF: org.apache.spark.sql.Dataset[
org.apache.spark.sql.Row] = [_c0: string, country: string ... 13 more fields]
```

```
//Total classes
val wine_class = WineDF.select("variety").distinct()
wine_class.show
wine_class.count()
```

```
+--------------------+ | variety| +--------------------+ | Chardonnay| |Borde
aux-style Re...| | Rosé| | Syrah| | Merlot| | Red Blend| | Pinot Noir| | Cabe
rnet Sauvignon| | Sauvignon Blanc| | Riesling| +--------------------+ wine_cl
ass: org.apache.spark.sql.Dataset[org.apache.spark.sql.Row] = [variety: strin
g] res1: Long = 10
```

Drop Nulls and Columns

```
//Remove unecessary columns and strip records with null values

//val concatDF =
WineDF.withColumn("description",concat(WineDF("description"),lit("
"),WineDF("taster_name")))
//                                        ,lit(" "),WineDF("country")))

val dropcolDF = WineDF.drop("title").drop("region_1").drop("region_2")
               .drop("taster_name")
```

```
                .drop("taster_twitter_handle")

.drop("winery").drop("designation").drop("frequency").drop("_c0")
                .na.drop()    //Strip null values
                .withColumn("id",monotonically_increasing_id()) //re-index the
data

//Strip Special Characters
val cleanedDF = dropcolDF.select(dropcolDF.columns.map(c =>
regexp_replace(dropcolDF(c), """[^A-Za-z0-9\s]+""", "").alias(c)): _*)


//cleanedDF.count()
//cleanedDF.show()
```

```
dropcolDF: org.apache.spark.sql.DataFrame = [country: string, description: st
ring ... 5 more fields] cleanedDF: org.apache.spark.sql.DataFrame = [country:
string, description: string ... 5 more fields]
```

```
// display(cleanedDF.select("concatenate"))
```

Set Tokenizer and Strip Stopwords

```
//Tokenize
val Tokenize = new Tokenizer()
                .setInputCol("description")
                .setOutputCol("description_token")


//Remove stop words
val Stopwordsremover = new StopWordsRemover()
                .setInputCol(Tokenize.getOutputCol)
                .setOutputCol("filtered")


val pipeline1 = new Pipeline()
              .setStages(Array(Tokenize, Stopwordsremover))


val Stopwords = pipeline1.fit(cleanedDF).transform(cleanedDF)
```

```
//Create Tempview
Stopwords.createTempView("Stopwords")

//Stopwords.show()
//Stopwords.count()
```

```
Tokenize: org.apache.spark.ml.feature.Tokenizer = tok_ebad034e8b7e Stopwordsr
emover: org.apache.spark.ml.feature.StopWordsRemover = stopWords_7a991a48dfb9
pipeline1: org.apache.spark.ml.Pipeline = pipeline_32038b79b145 Stopwords: or
g.apache.spark.sql.DataFrame = [country: string, description: string ... 7 mo
re fields]
```

Engage Stemmer

```
%python
from pyspark.sql.types import *

#Call table to python
Stopwords = spark.table("Stopwords")

#Use snowball
stemmer = SnowballStemmer('english')

#UDF to stem each token
udf1 = udf(lambda tokens: [stemmer.stem(token) for token in tokens],
ArrayType(StringType()))
Stemmed = Stopwords.withColumn("stemmed", udf1("filtered"))

#Cast datatypes
Stemmed = Stemmed.withColumn("points",
Stemmed["points"].cast("int")).withColumn("price",
Stemmed["price"].cast("int"))

Stemmed.createTempView("Stemmed")
```

Feature Extraction

```
//Call table in scala
```

```
val Stemmed = spark.table("Stemmed")

//Indexing Country (Categorical Feature)
val countryIndex = new StringIndexer()
              .setInputCol("country")
              .setOutputCol("countryIndex")

// val tasterhandle = new StringIndexer()
//                .setInputCol("taster_twitter_handle")
//                .setOutputCol("tasterhandle")

//Feature Extractors
//CountVectorizer Model - - (TF followed by IDF)
val countVec = new CountVectorizer()
                .setInputCol("stemmed")
                .setOutputCol("countvec")
              //  .setMinDF(5)
              //  .setMinTF(5)
                  .setVocabSize(4000)

//HashTF Model - (TF followed by IDF)
val hashTF = new HashingTF()
             .setInputCol("stemmed")
             .setOutputCol("hashtf")
             .setNumFeatures(4000)

val IDF = new IDF()
           .setInputCol(countVec.getOutputCol)
           .setOutputCol("tfidf")

//Word2Vec Model - (Uses word similarity)

// val word2Vec = new Word2Vec()
```

```
//                  .setInputCol("stemmed")
//                  .setOutputCol("word2vec")

//Choose which feature extractor to use and add it to pipeline
val pipeline2 = new Pipeline()
              .setStages(Array(countryIndex, countVec, IDF))

val FeatureExtractor = pipeline2.fit(Stemmed).transform(Stemmed)

//FeatureExtractor.show()
```

```
Stemmed: org.apache.spark.sql.DataFrame = [country: string, description: stri
ng ... 8 more fields] countryIndex: org.apache.spark.ml.feature.StringIndexer
= strIdx_67dffcea4690 countVec: org.apache.spark.ml.feature.CountVectorizer =
cntVec_92bb152144f8 hashTF: org.apache.spark.ml.feature.HashingTF = hashingTF
_215d2a22a1a6 IDF: org.apache.spark.ml.feature.IDF = idf_c1f8e7693a32 pipelin
e2: org.apache.spark.ml.Pipeline = pipeline_24c7cfba8ef5 FeatureExtractor: or
g.apache.spark.sql.DataFrame = [country: string, description: string ... 11 m
ore fields]
```

Vector Assemble

```
//Vector Assembler
//Choose the require features and iterate multiple times

val assembler =  new VectorAssembler()
                .setInputCols(Array("tfidf","countryIndex","points"))  //or
use word2vec
                .setOutputCol("features")

val AssembledDF = assembler.transform(FeatureExtractor).drop("description")
```

```
assembler: org.apache.spark.ml.feature.VectorAssembler = vecAssembler_7331d13
71798 AssembledDF: org.apache.spark.sql.DataFrame = [country: string, points:
int ... 11 more fields]
```

Dimensionality Reduction

```
//Dimensionality Reduction
```

```
val pcafeatures = new PCA()
                  .setInputCol("features")
                  .setOutputCol("pcafeatures")
                  .setK(500)  //Choose appropriate number of features

val PCADF = pcafeatures.fit(AssembledDF).transform(AssembledDF)
pcafeatures: org.apache.spark.ml.feature.PCA = pca_0ffb47aee80e PCADF: org.ap
ache.spark.sql.DataFrame = [country: string, points: int ... 12 more fields]
```

Prepare Label, Features and Test/Train Split

```
//Test-Train Split

val FeatureDF = PCADF.select("pcafeatures","variety")
val Array(train_data,test_data) = FeatureDF.randomSplit(Array(0.7, 0.3), seed
= 12345)

//FeatureDF.show()

// Prep Index labels and features
val labelIndexer = new StringIndexer()
                     .setInputCol("variety")
                     .setOutputCol("varietyIndex")
                     .fit(FeatureDF)

val featureIndexer = new VectorIndexer()
                      .setInputCol("pcafeatures")
                      .setOutputCol("featureIndex")
                      .fit(FeatureDF)

//Scale Feature for Naive Bayes
val featureIndexer1 = new MinMaxScaler()
                  .setInputCol("pcafeatures")
                  .setOutputCol("featureIndex1")
```

```
                    .setMax(1)
                    .setMin(0)

val labelConverter = new IndexToString()
                       .setInputCol("prediction")
                       .setOutputCol("predictedLabel")
                       .setLabels(labelIndexer.labels)
```

```
FeatureDF: org.apache.spark.sql.DataFrame = [pcafeatures: vector, variety: st
ring] train_data: org.apache.spark.sql.Dataset[org.apache.spark.sql.Row] = [p
cafeatures: vector, variety: string] test_data: org.apache.spark.sql.Dataset[
org.apache.spark.sql.Row] = [pcafeatures: vector, variety: string] labelIndex
er: org.apache.spark.ml.feature.StringIndexerModel = strIdx_619773713a20 feat
ureIndexer: org.apache.spark.ml.feature.VectorIndexerModel = vecIdx_99be9e1ce
78e featureIndexer1: org.apache.spark.ml.feature.MinMaxScaler = minMaxScal_e0
bbb04e2705 labelConverter: org.apache.spark.ml.feature.IndexToString = idxToS
tr_89dc3a6f0c5d
```

Model 1 - Logistic Regression

```
//Logistic Regression Model

val LR = new LogisticRegression()
         .setFeaturesCol("featureIndex")   //setting features column
         .setLabelCol("varietyIndex")

val LR_pipeline = new Pipeline()
                 .setStages(Array(labelIndexer, featureIndexer, LR,
labelConverter))

val LR_model = LR_pipeline.fit(train_data)

val LR_predictions = LR_model.transform(test_data)
```

```
LR: org.apache.spark.ml.classification.LogisticRegression = logreg_9a4e6d67cd
85 LR_pipeline: org.apache.spark.ml.Pipeline = pipeline_f9be57b292e9 LR_model
: org.apache.spark.ml.PipelineModel = pipeline_f9be57b292e9 LR_predictions: o
rg.apache.spark.sql.DataFrame = [pcafeatures: vector, variety: string ... 6 m
ore fields]
```

Logistic Regression Evaluation

```
val LR_evaluator = new MulticlassClassificationEvaluator()
                  .setLabelCol("varietyIndex")
                  .setPredictionCol("prediction")
                  .setMetricName("accuracy")

val LR_testaccuracy = LR_evaluator.evaluate(LR_predictions)

println("Test Error for Log Regression = " + (1.0 - LR_testaccuracy))
```

```
Test Error for Log Regression = 0.23202305819211844 LR_evaluator: org.apache.
spark.ml.evaluation.MulticlassClassificationEvaluator = mcEval_a181363a88c8 L
R_testaccuracy: Double = 0.7679769418078816
```

Logistic Regression - Metrics

```
val LR_predictions1 = LR_predictions.select("prediction", "varietyIndex")

val LR_RDD = LR_predictions1.rdd.map{x=>(x.getAs[Double](0),
x.getAs[Double](1))}

val LR_metrics= new MulticlassMetrics(LR_RDD)

println(s"Weighted precision: ${LR_metrics.weightedPrecision}")
println(s"Weighted recall: ${LR_metrics.weightedRecall}")
println(s"Weighted F1 score: ${LR_metrics.weightedFMeasure}")
println(s"Accuracy: ${LR_metrics.accuracy}")
```

```
Weighted precision: 0.7665200224388106 Weighted recall: 0.7679769418078816 We
ighted F1 score: 0.7649928837997194 Accuracy: 0.7679769418078816 LR_predictio
ns1: org.apache.spark.sql.DataFrame = [prediction: double, varietyIndex: doub
le] LR_RDD: org.apache.spark.rdd.RDD[(Double, Double)] = MapPartitionsRDD[757
] at map at command-60553195985273:3 LR_metrics: org.apache.spark.mllib.evalu
ation.MulticlassMetrics = org.apache.spark.mllib.evaluation.MulticlassMetrics
@350373a4
```

Logistic Regression - Check for Overfit

```
val LR_train = LR_model.transform(train_data)
```

```
val LR_trainaccuracy = LR_evaluator.evaluate(LR_train)

println("Train Error for Log Regression = " + (1.0 - LR_trainaccuracy))
```

```
Train Error for Log Regression = 0.20209399621976343 LR_train: org.apache.spa
rk.sql.DataFrame = [pcafeatures: vector, variety: string ... 6 more fields] L
R_trainaccuracy: Double = 0.7979060037802366
```

Logistic Regression - HyperParameter Tuning and Cross Validation

```
//Logistic Regression HyperParameter Tuning and Cross Validation

val LR_paramGrid = new ParamGridBuilder()
                .addGrid(LR.regParam, Array(0,0.1))
                .addGrid(LR.elasticNetParam, Array(0,0.1,0.3))
                .addGrid(LR.maxIter, Array(70,100))
                .build()

val LR_CrossValidation = new CrossValidator()
           .setEstimator(LR_pipeline)
           .setEvaluator(LR_evaluator)
           .setEstimatorParamMaps(LR_paramGrid)
           .setNumFolds(3)
```

```
LR_paramGrid: Array[org.apache.spark.ml.param.ParamMap] = Array({ logreg_9a4e
6d67cd85-elasticNetParam: 0.0, logreg_9a4e6d67cd85-maxIter: 70, logreg_9a4e6d
67cd85-regParam: 0.0 }, { logreg_9a4e6d67cd85-elasticNetParam: 0.1, logreg_9a
4e6d67cd85-maxIter: 70, logreg_9a4e6d67cd85-regParam: 0.0 }, { logreg_9a4e6d6
7cd85-elasticNetParam: 0.3, logreg_9a4e6d67cd85-maxIter: 70, logreg_9a4e6d67c
d85-regParam: 0.0 }, { logreg_9a4e6d67cd85-elasticNetParam: 0.0, logreg_9a4e6
d67cd85-maxIter: 70, logreg_9a4e6d67cd85-regParam: 0.1 }, { logreg_9a4e6d67cd
85-elasticNetParam: 0.1, logreg_9a4e6d67cd85-maxIter: 70, logreg_9a4e6d67cd85
-regParam: 0.1 }, { logreg_9a4e6d67cd85-elasticNetParam: 0.3, logreg_9a4e6d67
cd85-maxIter: 70, logreg_9a4e6d67cd85-regParam: 0.1 }, { logreg_9a4e6d67cd85-
elasticNetParam: 0.0, logreg_9a4e6d67cd85-maxIter: 100, logreg_9a4e6d67cd85-r
egParam: 0.0 }, { logreg_9a4e6d67cd85-elasticNetParam: 0.1, logreg_9a4e6d67cd
85-maxIter: 100, logreg_9a4e6d67cd85-regParam: 0.0 }, { logreg_9a4e6d67cd85-e
lasticNetParam: 0.3, logreg_9a4e6d67cd85-maxIter: 100, logreg_9a4e6d67cd85-re
gParam: 0.0 }, { logreg_9a4e6d67cd85-elasticNetParam: 0.0, logreg_9a4e6d67cd8
5-maxIter: 100, logreg_9a4e6d67cd85-regParam: 0.1 }, { logreg_9a4e6d67cd85-el
```

```
asticNetParam: 0.1, logreg_9a4e6d67cd85-maxIter: 100, logreg_9a4e6d67cd85-reg
Param: 0.1 }, { logreg_9a4e6d67cd85-elasticNetParam: 0.3, logreg_9a4e6d67cd85
-maxIter: 100, logreg_9a4e6d67cd85-regParam: 0.1 }) LR_CrossValidation: org.a
pache.spark.ml.tuning.CrossValidator = cv_d15eb896e522
```

```
val LR_CVmodel = LR_CrossValidation.fit(train_data)
val LR_CVpredictions = LR_CVmodel.transform(test_data)
val LR_CVaccuracy = LR_evaluator.evaluate(LR_CVpredictions)

println("Cross Validated Test Error for Logistic Regression = " + (1.0 -
LR_CVaccuracy))
```

Logistic Regression CV - Check for Overfit

```
val LR_CVtrain = LR_model.transform(train_data)

val LR_CVtrainaccuracy = LR_evaluator.evaluate(LR_CVtrain)

println("Cross Validated Train Error for Logistic Regression = " + (1.0 -
LR_CVtrainaccuracy))
```

Logistic Regression CV - Metrics

```
val LR_CVpredictions1 = LR_CVpredictions.select("prediction", "varietyIndex")

val LR_CVRDD = LR_CVpredictions1.rdd.map{x=>(x.getAs[Double](0),
x.getAs[Double](1))}

val LR_CVmetrics= new MulticlassMetrics(LR_CVRDD)

println(s"Weighted precision: ${LR_CVmetrics.weightedPrecision}")
println(s"Weighted recall: ${LR_CVmetrics.weightedRecall}")
println(s"Weighted F1 score: ${LR_CVmetrics.weightedFMeasure}")
println(s"Accuracy: ${LR_CVmetrics.accuracy}")
```

Logistic Regression ROC

```
// Curve Plotting
```

```
val LR1 =
LR_predictions.select(col("featureIndex"),col("prediction"),col("predictedLabel"),col("variety"),col("varietyIndex"),col("probability").as("prob"))


import org.apache.spark.ml.linalg.DenseVector


//labelConverter.getLabels


val toArr: Any => Array[Double] = _.asInstanceOf[DenseVector].toArray
val toArrUdf = udf(toArr)
val Table1 = LR1.withColumn("probability",toArrUdf('prob))
Table1.createTempView("Table")
```

```
LR1: org.apache.spark.sql.DataFrame = [featureIndex: vector, prediction: double ... 4 more fields] import org.apache.spark.ml.linalg.DenseVector toArr: Any => Array[Double] = <function1> toArrUdf: org.apache.spark.sql.expressions.UserDefinedFunction = UserDefinedFunction(<function1>,ArrayType(DoubleType,false),None) Table1: org.apache.spark.sql.DataFrame = [featureIndex: vector, prediction: double ... 5 more fields]
```

```
%python
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
from itertools import cycle
from sklearn import svm, datasets
from sklearn.metrics import roc_curve, auc
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import label_binarize
from sklearn.multiclass import OneVsRestClassifier
from scipy import interp


Table = spark.table("Table")
ovr_prob = Table.toPandas()


labels=ovr_prob["varietyIndex"].to_frame()
```

```
one_hot = pd.get_dummies(labels['varietyIndex'])

# Drop column B as it is now encoded
labels=labels.drop('varietyIndex',axis=1)

# Join the encoded df
labels=labels.join(one_hot)
y_score=ovr_prob["probability"].values
y_test=labels.values
y_score2=np.array([np.array(i) for i in y_score])


fpr=dict()

tpr=dict()

roc_auc=dict()

for i in range(10):
  fpr[i], tpr[i], _ =roc_curve(y_test[:, i], y_score2[:, i])
  roc_auc[i] =auc(fpr[i], tpr[i])

# Compute micro-average ROC curve and ROC area
fpr["micro"], tpr["micro"], _=roc_curve(y_test.ravel(), y_score2.ravel())
roc_auc["micro"] =auc(fpr["micro"], tpr["micro"])

all_fpr = np.unique(np.concatenate([fpr[i] for i in range(10)]))

# Then interpolate all ROC curves at this points
mean_tpr=np.zeros_like(all_fpr)
for i in range(10):
  mean_tpr+=interp(all_fpr, fpr[i], tpr[i])
```

```
 # Finally average it and compute AUC
mean_tpr/=10
fpr["macro"] =all_fpr
tpr["macro"] =mean_tpr
roc_auc["macro"] =auc(fpr["macro"], tpr["macro"])

# Plot all ROC curves
abc=plt.figure(figsize=(12,10))
lw=2
plt.plot(fpr["micro"], tpr["micro"],
         label='micro-average ROC curve (area = {0:0.2f})'
         .format(roc_auc["micro"]),
         color='deeppink', linestyle=':', linewidth=4)

plt.plot(fpr["macro"], tpr["macro"],
         label='macro-average ROC curve (area = {0:0.2f})'
         .format(roc_auc["macro"]),color='navy', linestyle=':', linewidth=4)



colors=cycle(['aqua', 'darkorange', 'cornflowerblue', 'blue', 'green', 'red',
'cyan', 'magenta', 'yellow', 'black'])

labels=['Pinot Noir', 'Chardonnay', 'Cabernet Sauvignon', 'Red Blend',
'Bordeaux-style Red Blend', 'Riesling', 'Sauvignon Blanc', 'Syrah', 'Rosé',
'Merlot']

for i,j, color in zip(range(10), labels, colors):
  plt.plot(fpr[i], tpr[i], color=color, lw=lw,
           label='ROC curve of class {0} (area = {1:0.2f})'
           .format(j, roc_auc[i]))



plt.plot([0, 1], [0, 1], 'k--', lw=lw)
```

```
plt.xlim([0.0, 1.0])
plt.ylim([0.0, 1.05])
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')
plt.title('ROC Curve for the select Classes')
plt.legend(loc="lower right")
display(abc)
```

Model 2 - Decision Tree Classifier

```
//Decision Tree Model

val DT = new DecisionTreeClassifier()
            .setLabelCol("varietyIndex")
            .setFeaturesCol("featureIndex")

val DT_pipeline = new Pipeline()
                  .setStages(Array(labelIndexer, featureIndexer, DT,
labelConverter))

val DT_model = DT_pipeline.fit(train_data)

// Make predictions
val DT_predictions = DT_model.transform(test_data)
```

```
DT: org.apache.spark.ml.classification.DecisionTreeClassifier = dtc_a6c49c051
2e8 DT_pipeline: org.apache.spark.ml.Pipeline = pipeline_e68723882548 DT_mode
l: org.apache.spark.ml.PipelineModel = pipeline_e68723882548 DT_predictions:
org.apache.spark.sql.DataFrame = [pcafeatures: vector, variety: string ... 6
more fields]
```

Decision Tree Evaluation

```
val DT_evaluator = new MulticlassClassificationEvaluator()
                  .setLabelCol("varietyIndex")
                  .setPredictionCol("prediction")
                  .setMetricName("accuracy")
```

```
val DT_testaccuracy = DT_evaluator.evaluate(DT_predictions)

println("Test Error for Decision Tree Classifier " + (1.0 - DT_testaccuracy))
```

```
Test Error for Decision Tree Classifier 0.46747502857426826 DT_evaluator: org
.apache.spark.ml.evaluation.MulticlassClassificationEvaluator = mcEval_b5dddb
e16e99 DT_testaccuracy: Double = 0.5325249714257317
```

Decision Tree - Check for Overfit

```
val DT_train = DT_model.transform(train_data)

val DT_trainaccuracy = DT_evaluator.evaluate(DT_train)

println("Train Error for Decision Tree Classifier = " + (1.0 -
DT_trainaccuracy))
```

```
Train Error for Decision Tree Classifier = 0.4667742689064922 DT_train: org.a
pache.spark.sql.DataFrame = [pcafeatures: vector, variety: string ... 6 more
fields] DT_trainaccuracy: Double = 0.5332257310935078
```

Decision Trees - Metrics

```
val DT_predict = DT_predictions.select("prediction", "varietyIndex")

val DT_RDD = DT_predict.rdd.map{x=>(x.getAs[Double](0), x.getAs[Double](1))}

val DT_metrics= new MulticlassMetrics(DT_RDD)

println(s"Weighted precision: ${DT_metrics.weightedPrecision}")
println(s"Weighted recall: ${DT_metrics.weightedRecall}")
println(s"Weighted F1 score: ${DT_metrics.weightedFMeasure}")
println(s"Accuracy: ${DT_metrics.accuracy}")
```

```
Weighted precision: 0.4771987495671818 Weighted recall: 0.5325249714257319 We
ighted F1 score: 0.486462906188746 Accuracy: 0.5325249714257317 DT_predict: o
rg.apache.spark.sql.DataFrame = [prediction: double, varietyIndex: double] DT
_RDD: org.apache.spark.rdd.RDD[(Double, Double)] = MapPartitionsRDD[14367] at
map at command-4095632453112007:3 DT_metrics: org.apache.spark.mllib.evaluati
```

```
on.MulticlassMetrics = org.apache.spark.mllib.evaluation.MulticlassMetrics@45
2da24b
```

Decision Tree - HyperParameter Tuning and Cross Validation

```
//Decision Tree HyperParameter Tuning and Cross Validation

val DT_paramGrid = new ParamGridBuilder()
                .addGrid(DT.maxDepth, Array(5,7))
                .addGrid(DT.impurity, Array("entropy","gini"))
                .addGrid(DT.maxBins, Array(25,35))
                .build()

val DT_CrossValidation = new CrossValidator()
           .setEstimator(DT_pipeline)
           .setEvaluator(DT_evaluator)
           .setEstimatorParamMaps(DT_paramGrid)
           .setNumFolds(3)
val DT_CVmodel = DT_CrossValidation.fit(train_data)
val DT_CVpredictions = DT_CVmodel.transform(test_data)
val DT_CVaccuracy = DT_evaluator.evaluate(DT_CVprediction)

println("Cross Validated Test Error for Decision Tree Classifier = " + (1.0 -
DT_CVaccuracy))
```

Decision Tree CV - Check for Overfit

```
val DT_CVtrain = DT_model.transform(train_data)

val DT_CVtrainaccuracy = DT_evaluator.evaluate(DT_CVtrain)

println("Cross Validated Train Error for Decision Tree Classifier = " + (1.0
-  DT_CVtrainaccuracy))
```

Decision Trees CV - Metrics

```
val DT_CVpredictions1 = DT_CVpredictions.select("prediction", "varietyIndex")
```

```
val DT_CVRDD = DT_CVpredictions1.rdd.map{x=>(x.getAs[Double](0),
x.getAs[Double](1))}

val DT_CVmetrics= new MulticlassMetrics(DT_CVRDD)

println(s"Weighted precision: ${DT_CVmetrics.weightedPrecision}")
println(s"Weighted recall: ${DT_CVmetrics.weightedRecall}")
println(s"Weighted F1 score: ${DT_CVmetrics.weightedFMeasure}")
println(s"Accuracy: ${DT_CVmetrics.accuracy}")
```

Model 3 - Random Forest Classifier

```
//RandomForest

val RF = new RandomForestClassifier()
        .setLabelCol("varietyIndex")
        .setFeaturesCol("featureIndex")
        .setNumTrees(20)

val RF_pipeline = new Pipeline()
                  .setStages(Array(labelIndexer, featureIndexer, RF,
labelConverter))

val RF_model = RF_pipeline.fit(train_data)

// Make predictions
val RF_predictions = RF_model.transform(test_data)
```

```
RF: org.apache.spark.ml.classification.RandomForestClassifier = rfc_1b6a2e483
412 RF_pipeline: org.apache.spark.ml.Pipeline = pipeline_224a80171f93 RF_mode
l: org.apache.spark.ml.PipelineModel = pipeline_224a80171f93 RF_predictions:
org.apache.spark.sql.DataFrame = [pcafeatures: vector, variety: string ... 6
more fields]
```

Random Forest - Evaluation

```
val RF_evaluator = new MulticlassClassificationEvaluator()
                  .setLabelCol("varietyIndex")
```

```
                  .setPredictionCol("prediction")
                  .setMetricName("accuracy")

val RF_testaccuracy = RF_evaluator.evaluate(RF_predictions)

println("Test Error for Random Forest Classifier " + (1.0 - RF_testaccuracy))
```

```
Test Error for Random Forest Classifier 0.4358694031705014 RF_evaluator: org.
apache.spark.ml.evaluation.MulticlassClassificationEvaluator = mcEval_cd06b07
23b5e RF_testaccuracy: Double = 0.5641305968294986
```

```
//RF_predictions.select("variety","varietyIndex","probability","prediction","
predictedLabel").show()
```

Random Forest - Check for Overfit

```
val RF_train = RF_model.transform(train_data)

val RF_trainaccuracy = RF_evaluator.evaluate(RF_train)

println("Train Error for Random Forest Classifier = " + (1.0 -
RF_trainaccuracy))
```

```
Train Error for Random Forest Classifier = 0.43572535944103463 RF_train: org.
apache.spark.sql.DataFrame = [pcafeatures: vector, variety: string ... 6 more
fields] RF_trainaccuracy: Double = 0.5642746405589654
```

```
RF_predictions.select("variety","varietyIndex","probability","prediction","pr
edictedLabel").show()
```

Random Forest - Metrics

```
val RF_predict = RF_predictions.select("prediction", "varietyIndex")

val RF_RDD = RF_predict.rdd.map{x=>(x.getAs[Double](0), x.getAs[Double](1))}

val RF_metrics= new MulticlassMetrics(RF_RDD)

println(s"Weighted precision: ${RF_metrics.weightedPrecision}")
println(s"Weighted recall: ${RF_metrics.weightedRecall}")
println(s"Weighted F1 score: ${RF_metrics.weightedFMeasure}")
```

```
println(s"Accuracy: ${RF_metrics.accuracy}")
```

```
Weighted precision: 0.616038031369321 Weighted recall: 0.5641305968294986 Wei
ghted F1 score: 0.5094228893418442 Accuracy: 0.5641305968294986 RF_predict: o
rg.apache.spark.sql.DataFrame = [prediction: double, varietyIndex: double] RF
_RDD: org.apache.spark.rdd.RDD[(Double, Double)] = MapPartitionsRDD[14519] at
map at command-4095632453112005:3 RF_metrics: org.apache.spark.mllib.evaluati
on.MulticlassMetrics = org.apache.spark.mllib.evaluation.MulticlassMetrics@76
8382fd
```

Random Forest - HyperParameter Tuning and Cross Validation

```
//Random Forest - HyperParameter Tuning and Cross Validation

val RF_ParamGrid = new ParamGridBuilder()
  .addGrid(RF.maxBins, Array(25,35))
  .addGrid(RF.maxDepth, Array(5,7))
  .addGrid(RF.impurity, Array("entropy", "gini"))
  .build()

// define cross validation stage to search through the parameters
// K-Fold cross validation with ClassificationEvaluator
val RF_CrossValidation = new CrossValidator()
  .setEstimator(RF_pipeline)
  .setEvaluator(RF_evaluator)
  .setEstimatorParamMaps(RF_ParamGrid)
  .setNumFolds(3)

val RF_CVmodel = RF_CrossValidation.fit(train_data)
val RF_CVpredictions = RF_CVmodel.transform(test_data)
val RF_CVaccuracy = RF_evaluator.evaluate(RF_CVpredictions)

println("Cross Validated Test Error for Random Forest Classifier = " + (1.0 -
RF_CVaccuracy))
```

Random Forest CV - Check for Overfit

```
val RF_CVtrainprediction = RF_CVmodel.transform(train_data)
```

```
val RF_CVtrainaccuracy = RF_evaluator.evaluate(RF_CVtrainprediction)

println("Cross Validated Train Error for Random Forest Classifier = " + (1.0
- RF_CVtrainaccuracy))
```

Random Forest CV - Metrics

```
val RF_CVpredictions1 = RF_CVpredictions.select("prediction", "varietyIndex")

val RF_CVRDD = RF_CVpredictions1.rdd.map{x=>(x.getAs[Double](0),
x.getAs[Double](1))}

val RF_CVmetrics= new MulticlassMetrics(RF_CVRDD)

println(s"Weighted precision: ${RF_CVmetrics.weightedPrecision}")
println(s"Weighted recall: ${RF_CVmetrics.weightedRecall}")
println(s"Weighted F1 score: ${RF_CVmetrics.weightedFMeasure}")
println(s"Accuracy: ${RF_CVmetrics.accuracy}")
```

Model 4 - Naive Bayes Classifier

```
// Naive Bayes Classifier

val NB = new NaiveBayes()
         .setFeaturesCol("featureIndex1")   //setting features column
         .setLabelCol("varietyIndex")

val NB_pipeline = new Pipeline()
                .setStages(Array(labelIndexer, featureIndexer1, NB,
labelConverter))

val NB_model = NB_pipeline.fit(train_data)

val NB_predictions = NB_model.transform(test_data)
```

Naive Bayes - Evaluation

```
val NB_evaluator = new MulticlassClassificationEvaluator()
                  .setLabelCol("varietyIndex")
                  .setPredictionCol("prediction")
                  .setMetricName("accuracy")

val NB_testaccuracy = NB_evaluator.evaluate(NB_predictions)

println("Test Error for Naive Bayes Classifier " + (1.0 - NB_testaccuracy))
```

Naive Bayes - Check for Overfit

```
val NB_train = NB_model.transform(train_data)

val NB_trainaccuracy = NB_evaluator.evaluate(NB_train)

println("Train Error for Naive Bayes Classifier = " + (1.0 -
NB_trainaccuracy))
```

Naive Bayes - Metrics

```
val NB_predict = NB_predictions.select("prediction", "varietyIndex")

val NB_RDD = RF_predict.rdd.map{x=>(x.getAs[Double](0), x.getAs[Double](1))}

val NB_metrics= new MulticlassMetrics(NB_RDD)

println(s"Weighted precision: ${NB_metrics.weightedPrecision}")
println(s"Weighted recall: ${NB_metrics.weightedRecall}")
println(s"Weighted F1 score: ${NB_metrics.weightedFMeasure}")
println(s"Accuracy: ${NB_metrics.accuracy}")
```