Song Year Prediction on Million Song Dataset using Pyspark

Akshat Kumar,Ajinkya Patankar

ak1648@rutgers.edu
aap256@rutgers.edu

December 18, 2018

Abstract

Timbre is the quality of a musical note, sound, or tone that distinguishes different types of sound production, such as voices and musical instruments This project focuses upon prediction of the release year of a particular song by using a song's inherent timbre features. It will serve as a useful topic because people tend to have some affinity to different genres of music throughout their life. This research will also help in determining the long term evolution of music over the years and could be a useful basis for recommending songs. The study could further be extended to generate new songs by learning the audio features of songs over the years. In this research, we have applied four different predictive models on Million Song Dataset and these are built on top of Apache Spark. The models have been compared based on their accuracy to predict the release year as well as their performance on a single node and multi-node Apache Spark cluster environment.

1 Introduction

1.1 Problem Description

In simple terms , timbre is a unique feature of any sound vibration, other than frequency, amplitude and modulation, which makes it different from every other sound. Two sounds can have same frequency and amplitude but certainly not the same timbre. [4]
Example, The sounds played by a violin and a piano at the same note sound different because of timber.
The Million Song Dataset contains 90 timbre features.So, the pertinent question with regard to this data was:
Does there exist a strong link between the audio features of a particular era to the types of songs created then?

So, essentially what we are trying to do is take the timbre features of the songs and trying to establish a link with the release year of these songs.

1.2 Problem Significance and Goal

People tend to have some affinity to the genres of music(such as community events, college and high school), thus predicting the year of any particular song will serve as a useful topic.
Also, an accurate model of the variation in the audio features of songs through the years could be useful in predicting in long terms, the evolution of various music genres.
Prediction of the release years of songs would also be a useful basis for song recommendations.[3]

2 Dataset

The Million Song Dataset, is a freely-available collection of audio features and metadata for a million contemporary popular music tracks available at UCI Machine Learning Repository. Attractive features of the Million Song Database include the range of existing resources to which it is linked, and the fact that it is the largest current research dataset in this field. [1]
The MSD contains metadata and audio analysis for a million songs that were legally available to The Echo Nest. The songs are representative of recent western commercial music. The MSD stands out as the largest currently available for researchers.

2.1 The Timbre Features

Timbre, in literal sense means, Tone Quality that distinguishes different types of sounds.
The first feature is the decision label Year (target), ranging from 1922 to 2011 and then there are 90 attributes[5]
TimbreAverage[1-12]
TimbreCovariance[13-90]
Following is a table, that lists down some of the features of the songs :

Field	Type	Description
Analysis Sample Rate	float	rate audio
danceability	float	algorithmic estimate
energy	float	energy from listener view
key confidence	float	confidence measure

Table 1: Sample Timbre Features Description

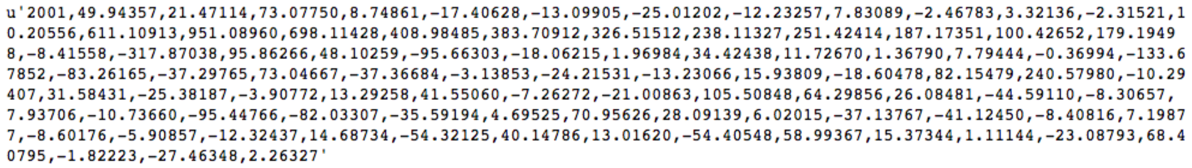
2.2 Dataset Characterstics

The MSD contains audio features and metadata for a million contemporary popular music tracks. It contains: 280 GB of data, 1, 000, 000 songs/files, 44, 745 unique artists, 7, 643 unique terms (Echo Nest tags), 2, 201, 916 asymmetric similarity relationships and 515, 576 dated tracks starting from 1922

3 Approach and Methodology

3.1 Sample Data Analysis

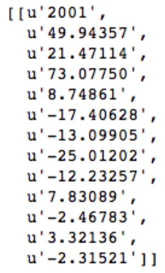
We imported the data set and took first two results of the same. Following was the output.



```
u'2001,49.94357,21.47114,73.07750,8.74861,-17.40628,-13.09905,-25.01202,-12.23257,7.83089,-2.46783,3.32136,-2.31521,1
0.20556,611.10913,951.08960,698.11428,408.98485,383.70912,326.51512,238.11327,251.42414,187.17351,100.42652,179.1949
8,-8.41558,-317.87038,95.86266,48.10259,-95.66303,-18.06215,1.96984,34.42438,11.72670,1.36790,7.79444,-0.36994,-133.6
7852,-83.26165,-37.29765,73.04667,-37.36684,-3.13853,-24.21531,-13.23066,15.93809,-18.60478,82.15479,240.57980,-10.29
407,31.58431,-25.38187,-3.90772,13.29258,41.55060,-7.26272,-21.00863,105.50848,64.29856,26.08481,-44.59110,-8.30657,
7.93706,-10.73660,-95.44766,-82.03307,-35.59194,4.69525,70.95626,28.09139,6.02015,-37.13767,-41.12450,-8.40816,7.1987
7,-8.60176,-5.90857,-12.32437,14.68734,-54.32125,40.14786,13.01620,-54.40548,58.99367,15.37344,1.11144,-23.08793,68.4
0795,-1.82223,-27.46348,2.26327'
```

3.2 Data Cleaning

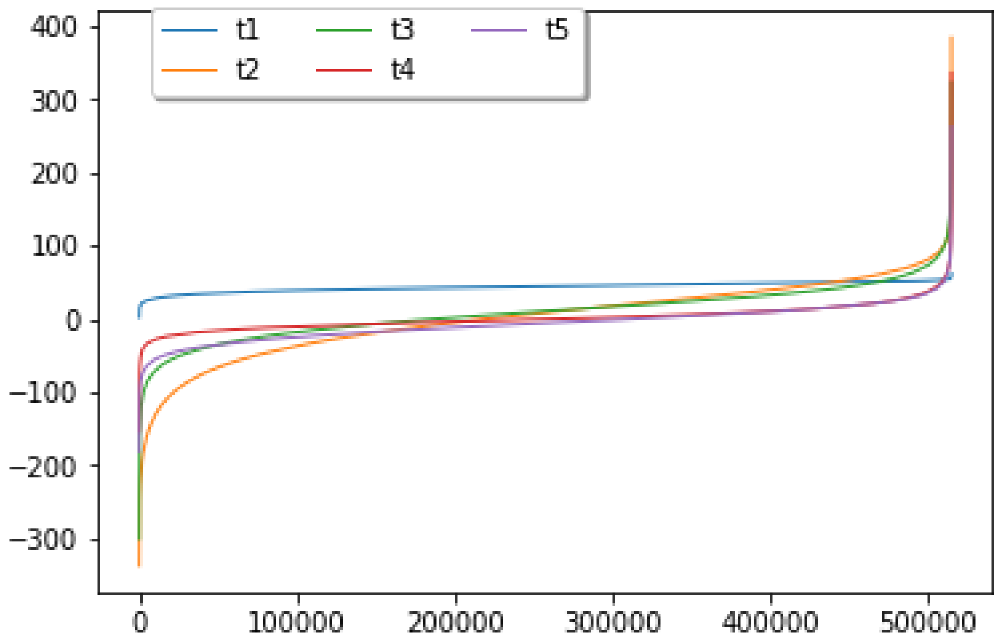
The above cluttered results were represented in a presentable format, by capsuling the timber features with its corresponding label which is year in which the song was released.



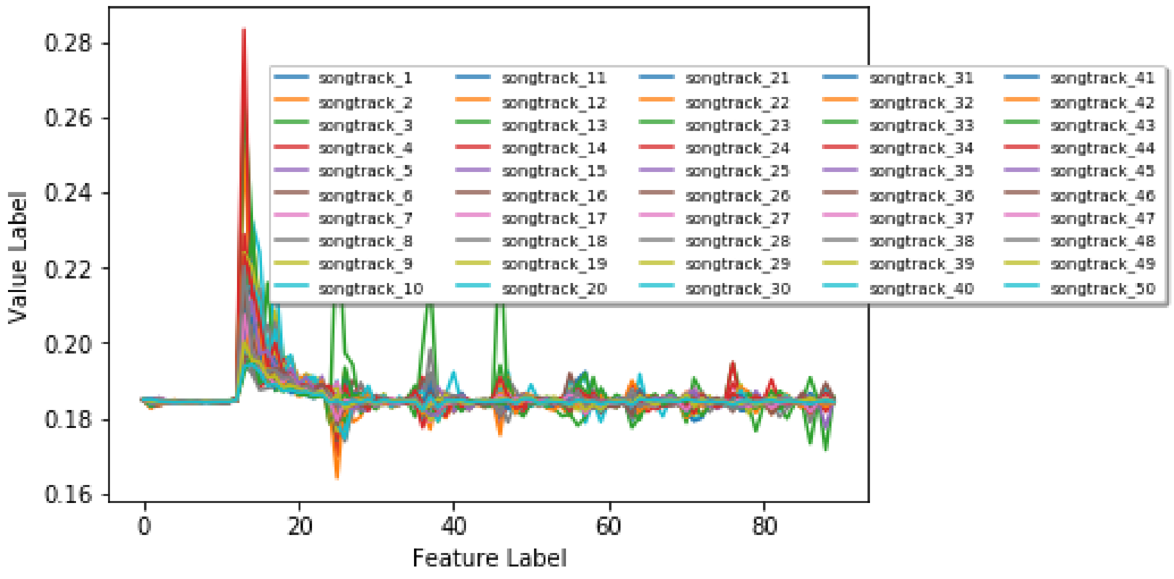
```
[[u'2001',
  u'49.94357',
  u'21.47114',
  u'73.07750',
  u'8.74861',
  u'-17.40628',
  u'-13.09905',
  u'-25.01202',
  u'-12.23257',
  u'7.83089',
  u'-2.46783',
  u'3.32136',
  u'-2.31521']]
```

3.3 Timbre Feature Analysis

A sample of timbre features are selected from the data and their values are being compared against each other to know whether they lie in the same scale or they have different scales and require normalization.
t1,t2,t3,t4,t5 represent sample timbre features.
As you can see from the below graph, value of features vary in a large magnitude and there we will do feature scaling to bring down every feature on a scale of 0 to 1.



We have also plotted the value of each of the dataset features for 50 randomly selected song tracks to get a sense of the difference in the feature values which would be helpful for us in further analysis.



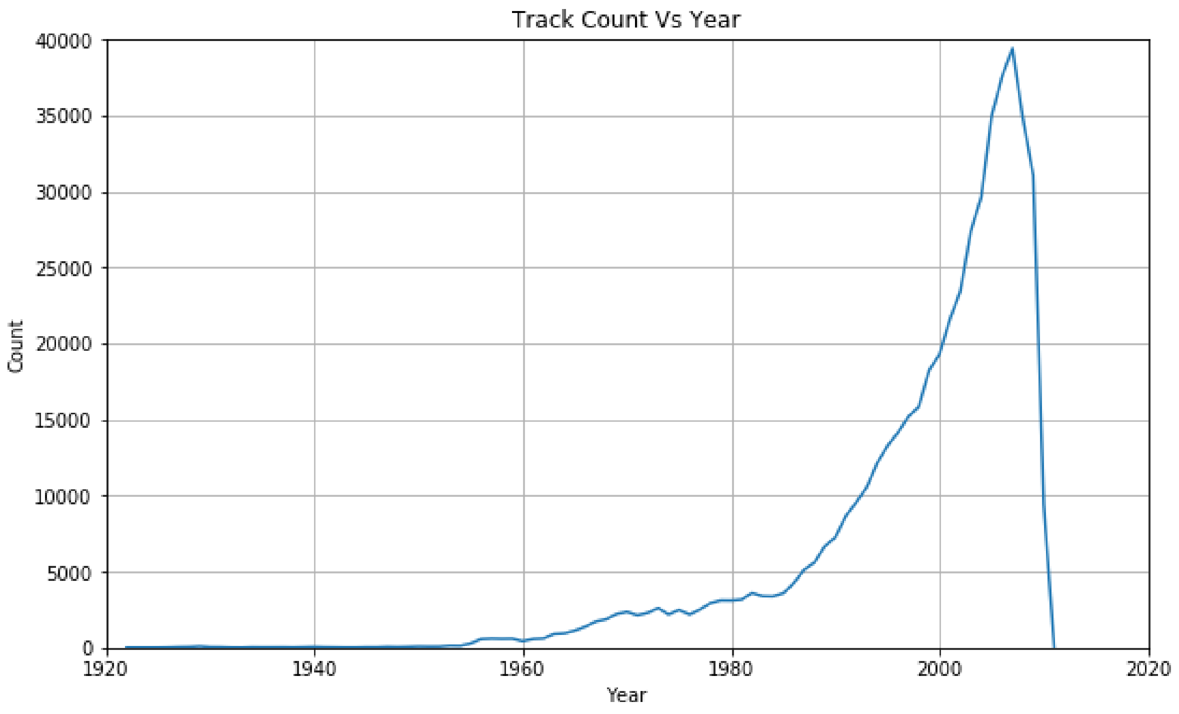
3.4 Rescaling and Normalization of Timber Features

After the timbre feature analysis, we know that feature scaling and normalization has to be done before we could move forward.
The formula below is calculated by subtracting the maximum value of a particular feature from each of it's values and dividing the result with the difference in the maximum and minimum value of that feature.
Formula used for rescaling and normalization is:

$$scaledFeature[i] = \frac{feature[i]-max}{max-min} \tag{1}$$

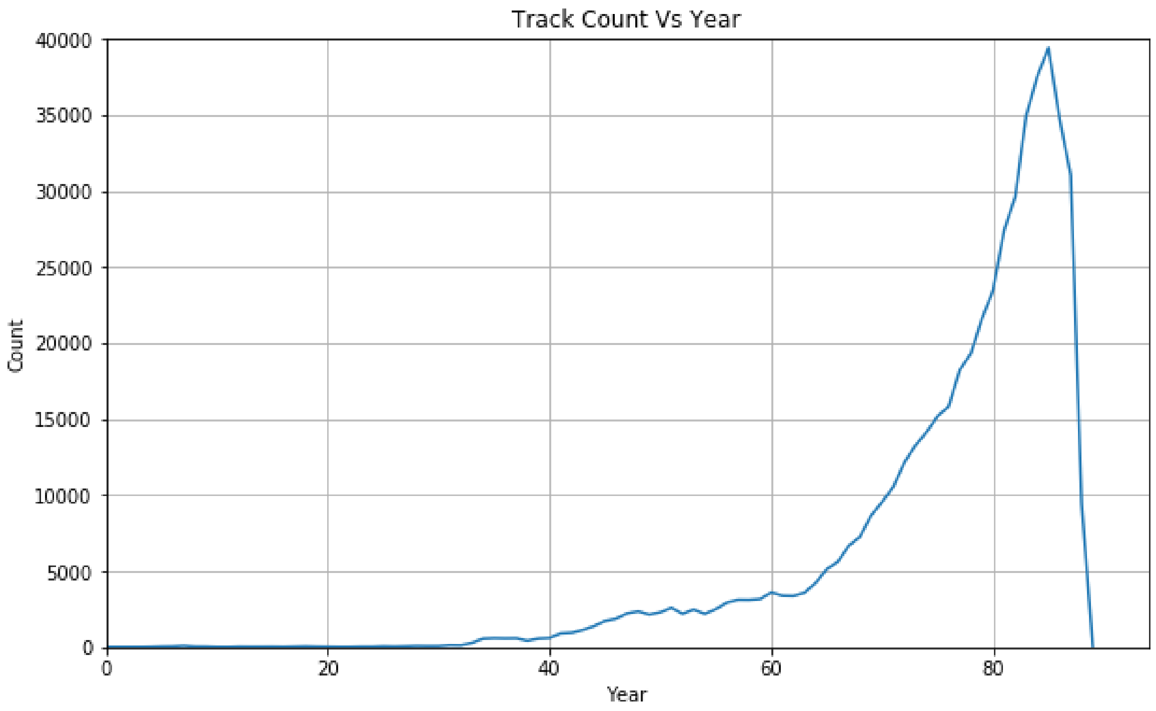
3.5 Finding Track Count per year

We found out the smallest as well as the largest value of the years, corresponding to each of the songs in our data set. Minimum value was 1922, maximum was 2011.
The number of songs was then plotted against the year in which it was released.After a little more refinement of the corresponding labels, the training data set was ready.



3.6 Shifting of Labels

In order to simplify the complexity and enhance the efficiency of the predictive model, all the values of the labels will be shifted, in order to start from 0. This means that the Value of the first label, ie 1922 will be shifted to the value 0 1923 :1 , 1924: 2 so on.

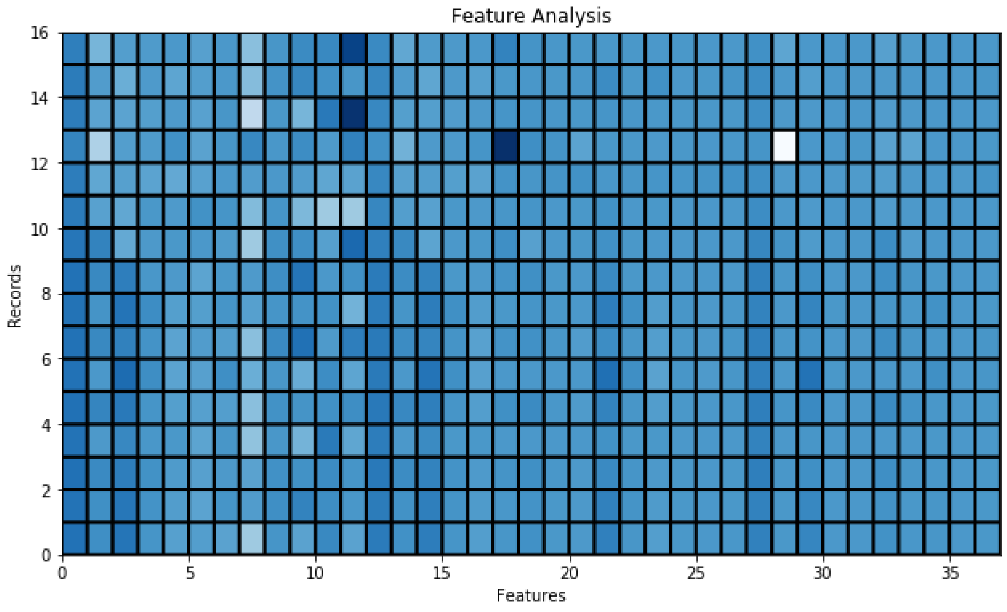


3.7 Feature addition

Till now, we have been using 12 timbre features, however accuracy of the system enhances rapidly, if more number of relevant features are added. For that, we used the concept of 2-way interaction among the features. Suppose there are 3 features a,b and c. Then apart from these three features, we can add some more features like a*a, a*b, a*c, b*b, b*c etc.

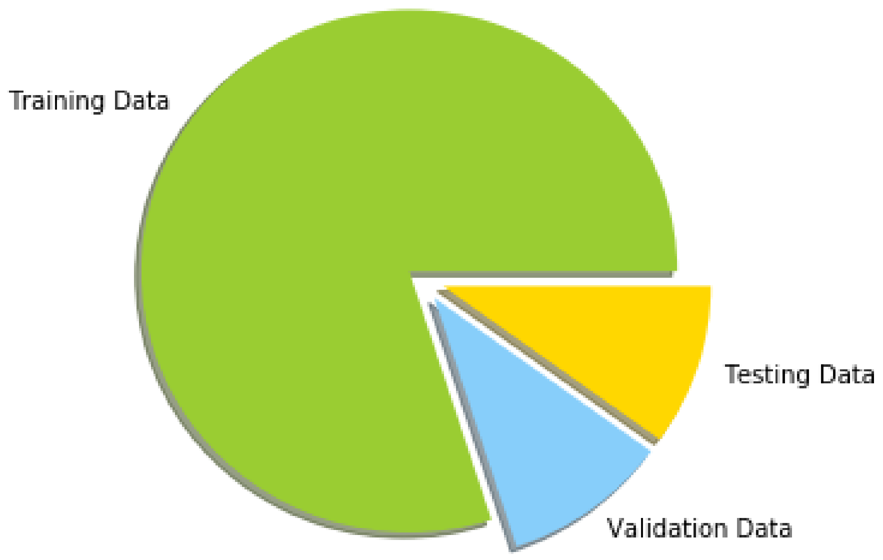
3.8 Visualization of Normalized Data

Data has been visualized in the form of a heat-map. This heat map clearly illustrates the features whose value is approaching 1 after rescaling and normalization.
line-graph has also been generated, which represents the number of songs per year, for all the years in the range 1922-2011.



3.9 Dataset Division for Model Development

We have also created a pie-chart, that represents the breakdown of the amount of data we would be using in the future for Training/Validation/Testing sets.

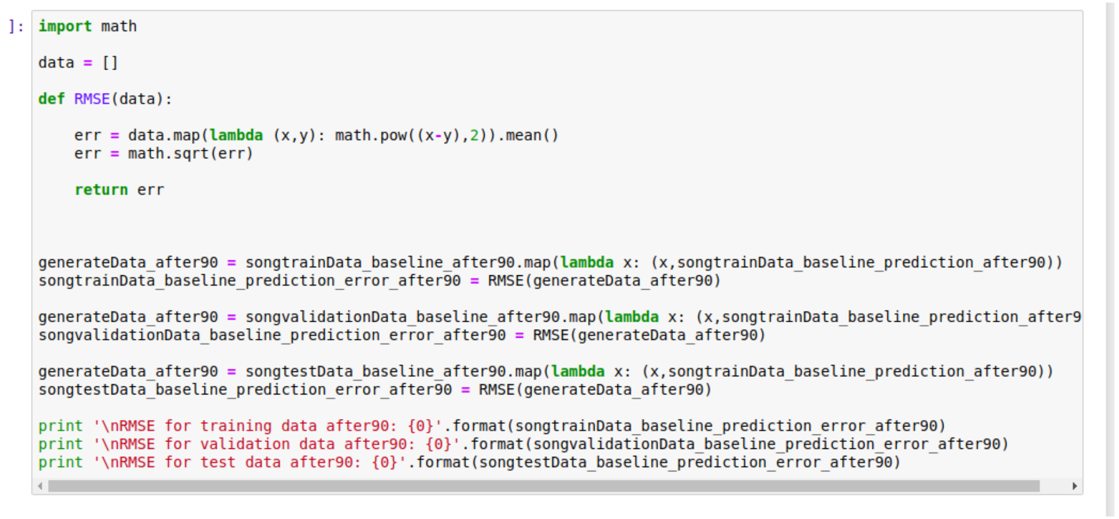


4 Model Framework Implementation

The Model Development Stage is the next stage in our project. We have built four machine learning models namely **Baseline Model, Linear Regression Model, Random Forest Model and Gradient Boosted Trees model** on top of Apache Spark on the transformed data for song year prediction. In the end, we will be using the comparison between **Regression, Random Forest and Gradient Boosted** models as the yardstick of knowing which is better in terms of both accuracy and computational time.[2]

4.1 Development of Baseline Model

A baseline model has been developed that predicts the average of all the years in our dataset. In our case, the average comes out to be 76.4 (after shifting the year label)

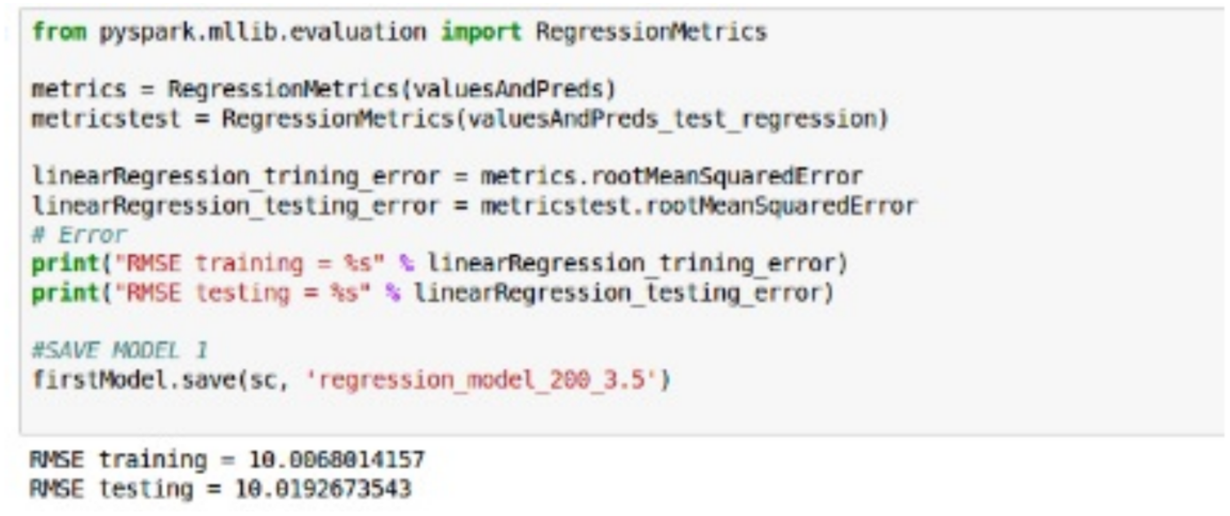


```
]: import math

data = []

def RMSE(data):

    err = data.map(lambda (x,y): math.pow((x-y),2)).mean()
    err = math.sqrt(err)

    return err


generateData_after90 = songtrainData_baseline_after90.map(lambda x: (x,songtrainData_baseline_prediction_after90))
songtrainData_baseline_prediction_error_after90 = RMSE(generateData_after90)

generateData_after90 = songvalidationData_baseline_after90.map(lambda x: (x,songtrainData_baseline_prediction_after9
songvalidationData_baseline_prediction_error_after90 = RMSE(generateData_after90)

generateData_after90 = songtestData_baseline_after90.map(lambda x: (x,songtrainData_baseline_prediction_after90))
songtestData_baseline_prediction_error_after90 = RMSE(generateData_after90)

print '\nRMSE for training data after90: {0}'.format(songtrainData_baseline_prediction_error_after90)
print '\nRMSE for validation data after90: {0}'.format(songvalidationData_baseline_prediction_error_after90)
print '\nRMSE for test data after90: {0}'.format(songtestData_baseline_prediction_error_after90)
```

We calculated the Root Mean Square Error (RMSE) on training, test and validation datasets and have found the RMSE for training dataset to be 10.93 approx, for testing dataset to be 10.96 approx and for validation dataset to be 10.897 approx.

4.2 Development of Linear Regression based model

A linear regression based model has been implemented to predict the year of a particular song. Comparing the results so obtained with those of **Baseline Model**, we can see that the **Linear Regression Model** has outperformed the former at first decimal place itself.



```
from pyspark.mllib.evaluation import RegressionMetrics

metrics = RegressionMetrics(valuesAndPreds)
metricstest = RegressionMetrics(valuesAndPreds_test_regression)

linearRegression_trining_error = metrics.rootMeanSquaredError
linearRegression_testing_error = metricstest.rootMeanSquaredError
# Error
print("RMSE training = %s" % linearRegression_trining_error)
print("RMSE testing = %s" % linearRegression_testing_error)

#SAVE MODEL 1
firstModel.save(sc, 'regression_model_200_3.5')
```

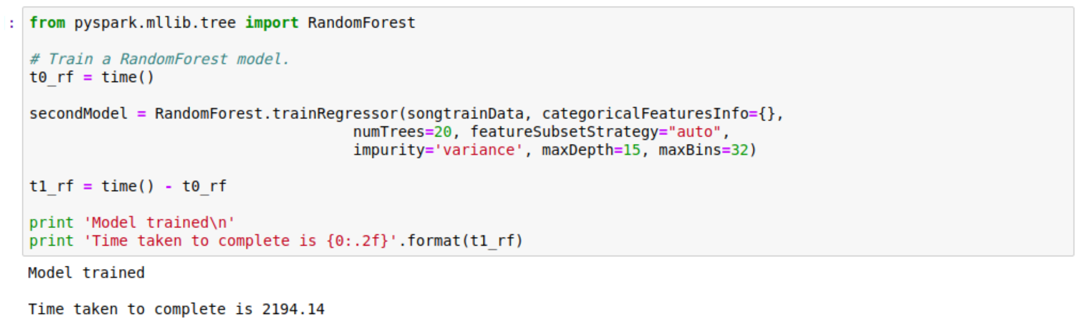
```
RMSE training = 10.0068014157
RMSE testing = 10.0192673543
```

Here, we find that the Root Mean Square Error for training dataset is 10.006 approx, while that of testing dataset is 10.019 approx.

4.3 Random Forest Implementation

4.3.1 Training model with dataset

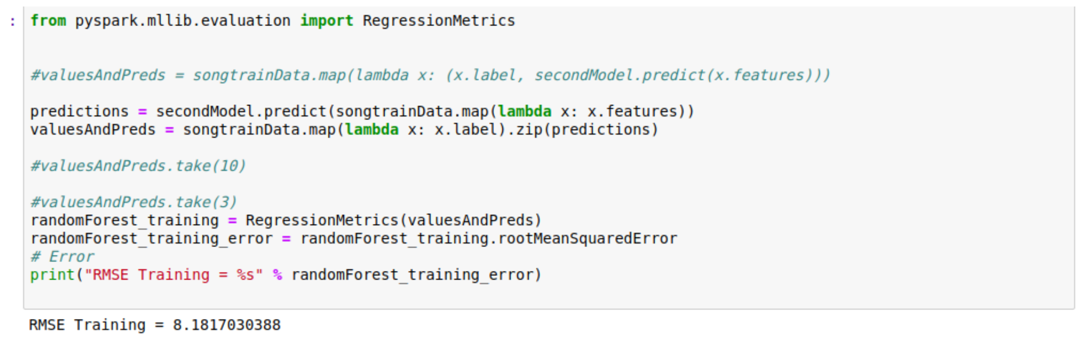
Random forests train a set of decision trees separately, so the training can be done in parallel. Instead of searching for the most important feature, it searches for the best feature amongst random subset of features.



```
from pyspark.mllib.tree import RandomForest

# Train a RandomForest model.
t0_rf = time()

secondModel = RandomForest.trainRegressor(songtrainData, categoricalFeaturesInfo={},
                                    numTrees=20, featureSubsetStrategy="auto",
                                    impurity='variance', maxDepth=15, maxBins=32)

t1_rf = time() - t0_rf

print 'Model trained\n'
print 'Time taken to complete is {0:.2f}'.format(t1_rf)
```

```
Model trained

Time taken to complete is 2194.14
```

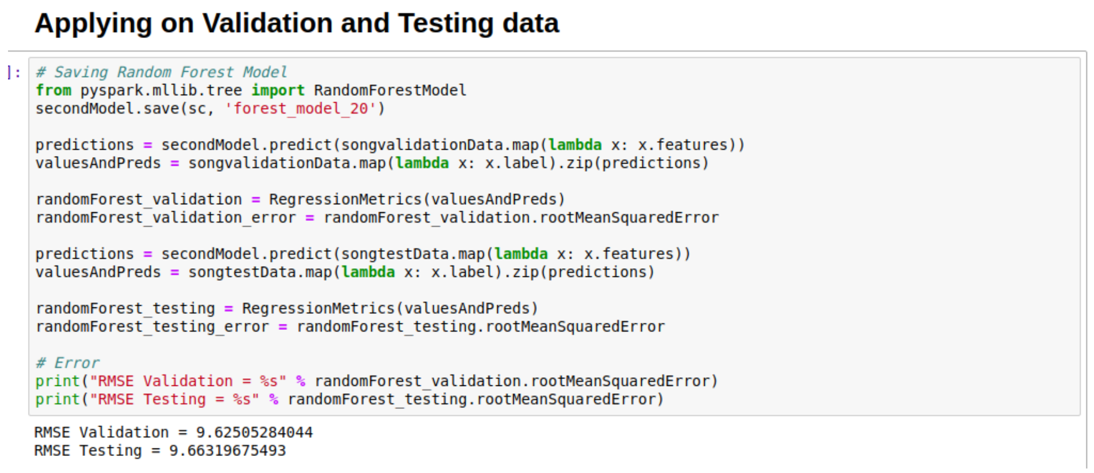
The algorithm injects randomness into the training process so that each decision tree is a bit different. Combining the predictions from each tree reduces the variance of the predictions, improving the performance on test data. The time taken by Rondom Forest trainig process is 2194.14



```
from pyspark.mllib.evaluation import RegressionMetrics


#valuesAndPreds = songtrainData.map(lambda x: (x.label, secondModel.predict(x.features)))

predictions = secondModel.predict(songtrainData.map(lambda x: x.features))
valuesAndPreds = songtrainData.map(lambda x: x.label).zip(predictions)

#valuesAndPreds.take(10)

#valuesAndPreds.take(3)
randomForest_training = RegressionMetrics(valuesAndPreds)
randomForest_training_error = randomForest_training.rootMeanSquaredError
# Error
print("RMSE Training = %s" % randomForest_training_error)
```

```
RMSE Training = 8.1817030388
```

4.3.2 Testing and Validation dataset application



Applying on Validation and Testing data

```
# Saving Random Forest Model
from pyspark.mllib.tree import RandomForestModel
secondModel.save(sc, 'forest_model_20')

predictions = secondModel.predict(songvalidationData.map(lambda x: x.features))
valuesAndPreds = songvalidationData.map(lambda x: x.label).zip(predictions)

randomForest_validation = RegressionMetrics(valuesAndPreds)
randomForest_validation_error = randomForest_validation.rootMeanSquaredError

predictions = secondModel.predict(songtestData.map(lambda x: x.features))
valuesAndPreds = songtestData.map(lambda x: x.label).zip(predictions)

randomForest_testing = RegressionMetrics(valuesAndPreds)
randomForest_testing_error = randomForest_testing.rootMeanSquaredError

# Error
print("RMSE Validation = %s" % randomForest_validation.rootMeanSquaredError)
print("RMSE Testing = %s" % randomForest_testing.rootMeanSquaredError)
```

```
RMSE Validation = 9.62505284044
RMSE Testing = 9.66319675493
```

4.4 Implementation of Gradient Boosted Trees

Gradient boosting iteratively trains a sequence of decision trees. On each iteration, the algorithm uses the current ensemble to predict the label of each training instance and then compares the prediction with the true label. The dataset is re-labeled to put more emphasis on training instances with poor predictions. Thus, in the next iteration, the decision tree will help correct for previous mistakes.

4.4.1 Training model with Gradient Booster

]:
```python
from pyspark.mllib.tree import GradientBoostedTrees, GradientBoostedTreesModel

t0 = time()

thirdModel = GradientBoostedTrees.trainRegressor(songtrainData,
    categoricalFeaturesInfo={}, numIterations=60)

t1 = time() - t0

print 'Model Trained'
print 'Time :{0:.2f}'.format(t1)
print '\n'
```

```
Model Trained
Time :930.64
```

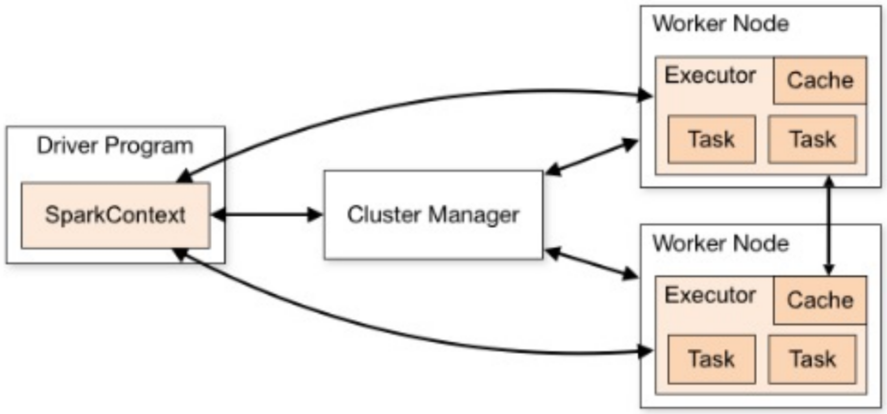
]:
```python
predictions = thirdModel.predict(songtrainData.map(lambda x: x.features))
labelsAndPredictions = songtrainData.map(lambda lp: lp.label).zip(predictions)
metrics = RegressionMetrics(labelsAndPredictions)
gb_training_error = metrics.rootMeanSquaredError
print("RMSE training= %s" % metrics.rootMeanSquaredError)

predictions = thirdModel.predict(songtestData.map(lambda x: x.features))
valuesAndPreds = songtestData.map(lambda x: x.label).zip(predictions)

#Save Model
#thirdModel.save(sc, 'boosted_model_60')
#gb_testing = RegressionMetrics(valuesAndPreds)
#gb_testing_error = gb_testing.rootMeanSquaredError
#print("RMSE testing= %s" % gb_testing.rootMeanSquaredError)
```

```
RMSE training= 9.7528682448
```

Once, we realize that the model has reached a stage that residuals cannot be remodelled further, or it has perfected the model, we can stop the iterations.

The specific mechanism for re-labeling instances is defined by a loss function. With each iteration, GBTs further reduce this loss function on the training data.

5 Single vs Multi-node Performance Comparison



Single node V/S Multi-node Spark cluster environment:
We implemented the three models, namely Linear regression, random forest and gradient boosted trees and tested them on a single node and multi-node Spark environment.

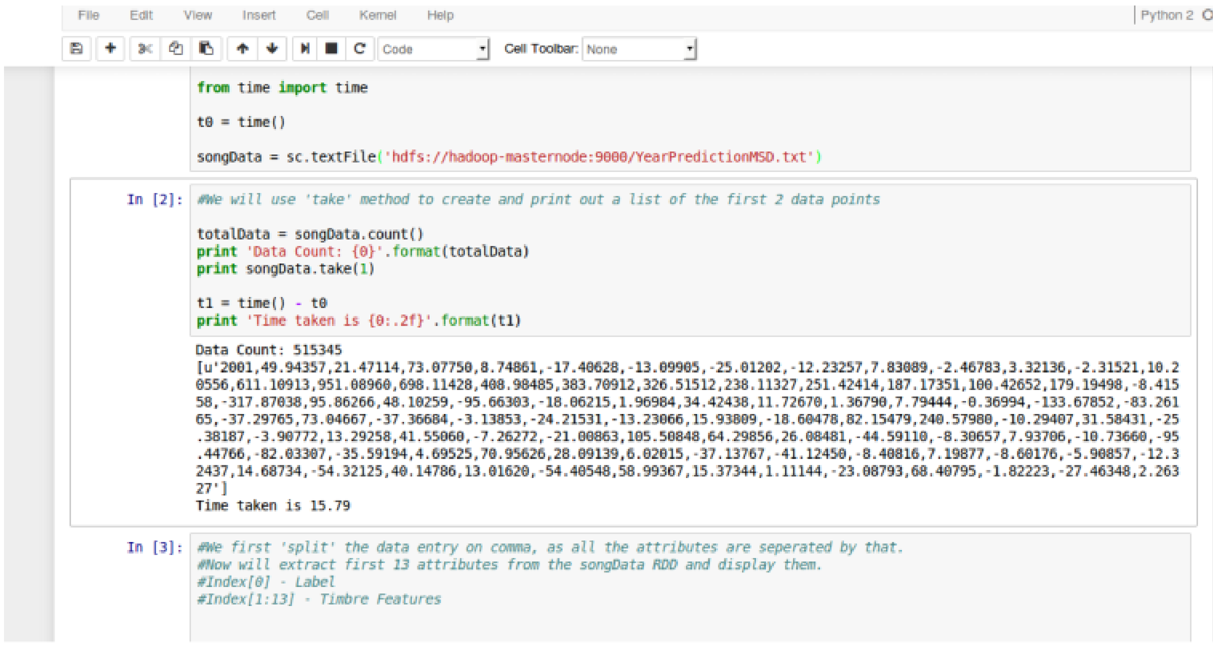
5.1 Configuration Specification

Configuration of Single Node system :
The overall single node processing has been done on a i5 system , with 16 GB RAM, exclusively allocated to the virtual machine. The processing is distributed across 4 cores.

Configuration of Multi node system :
The computation is done with the help of 5 nodes. Out of which 1 is the master node and 4 of them are slaves. Each of the four slaves have i5 processors with 4GB RAM , having 1 core each.

5.2 Performance Benchmarking

- **Data Count - 1 node :**



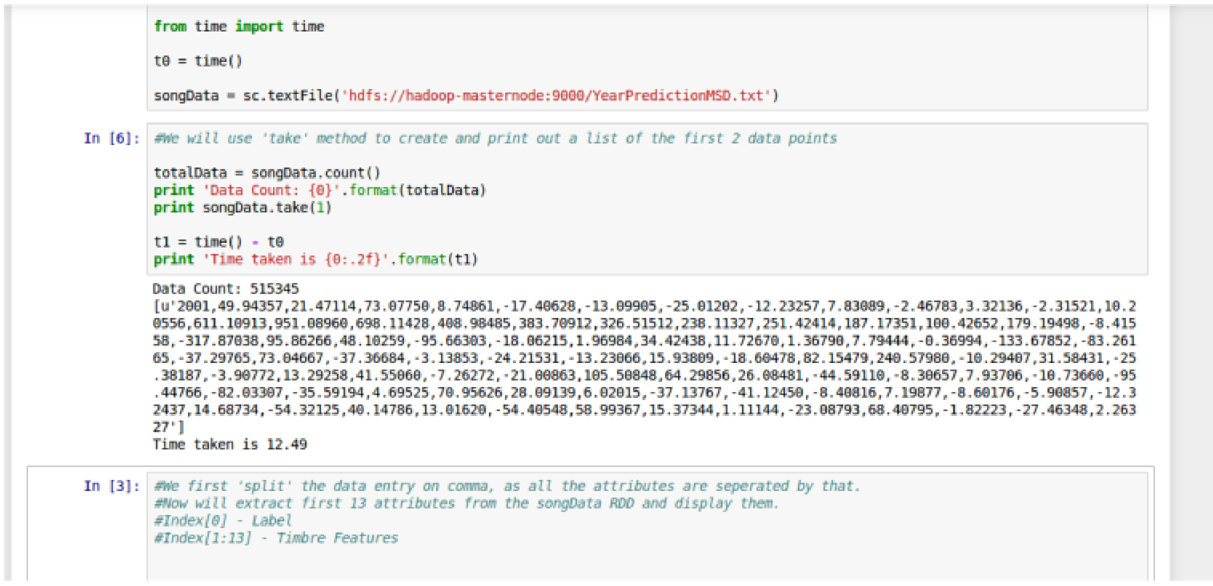
```
from time import time

t0 = time()

songData = sc.textFile('hdfs://hadoop-masternode:9000/YearPredictionMSD.txt')
```

```
In [2]: #We will use 'take' method to create and print out a list of the first 2 data points

        totalData = songData.count()
        print 'Data Count: {0}'.format(totalData)
        print songData.take(1)

        t1 = time() - t0
        print 'Time taken is {0:.2f}'.format(t1)
```

```
Data Count: 515345
[u'2001,49.94357,21.47114,73.07750,8.74861,-17.40628,-13.09905,-25.01202,-12.23257,7.83089,-2.46783,3.32136,-2.31521,10.2
0556,611.10913,951.08960,698.11428,408.98485,383.70912,326.51512,238.11327,251.42414,187.17351,100.42652,179.19498,-8.415
58,-317.87038,95.86266,48.10259,-95.66303,-18.06215,1.96984,34.42438,11.72670,1.36790,7.79444,-0.36994,-133.67852,-83.261
65,-37.29765,73.04667,-37.36684,-3.13853,-24.21531,-13.23066,15.93809,-18.60478,82.15479,240.57980,-10.29407,31.58431,-25
.38187,-3.90772,13.29258,41.55060,-7.26272,-21.00863,105.50848,64.29856,26.08481,-44.59110,-8.30657,7.93706,-10.73660,-95
.44766,-82.03307,-35.59194,4.69525,70.95626,28.09139,6.02015,-37.13767,-41.12450,-8.40816,7.19877,-8.60176,-5.90857,-12.3
2437,14.68734,-54.32125,40.14786,13.01620,-54.40548,58.99367,15.37344,1.11144,-23.08793,68.40795,-1.82223,-27.46348,2.263
27']
Time taken is 15.79
```

```
In [3]: #We first 'split' the data entry on comma, as all the attributes are seperated by that.
        #Now will extract first 13 attributes from the songData RDD and display them.
        #Index[0] - Label
        #Index[1:13] - Timbre Features
```

∗ **Data Count - 5 nodes :**



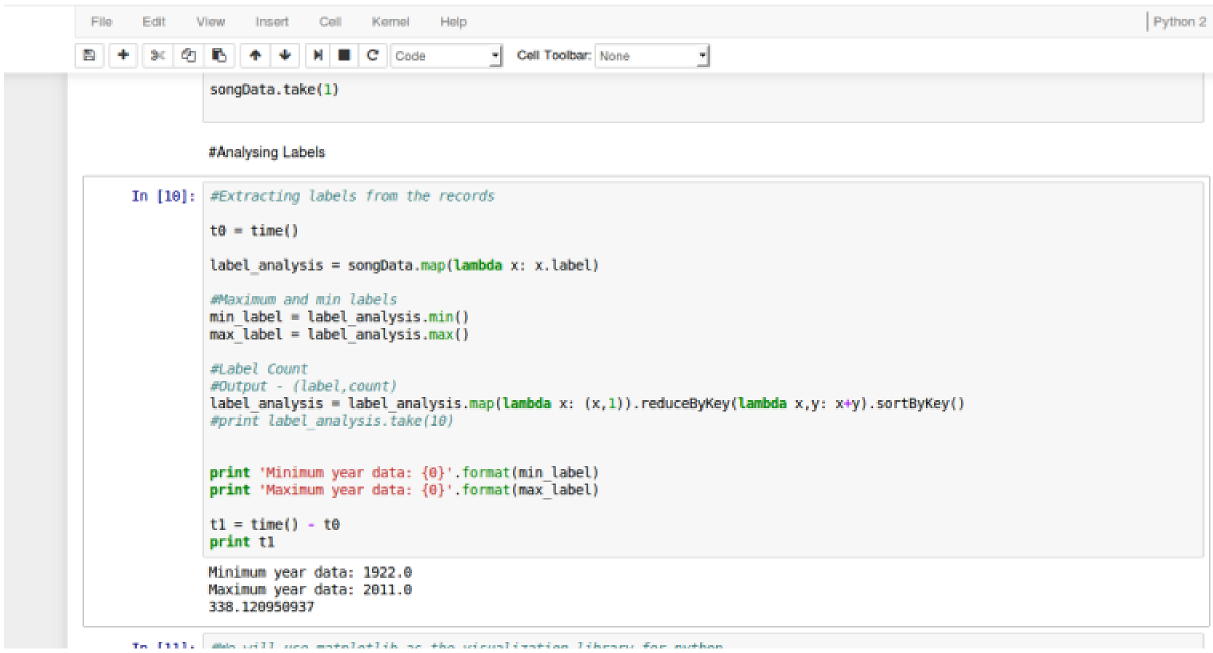
```
from time import time

t0 = time()

songData = sc.textFile('hdfs://hadoop-masternode:9000/YearPredictionMSD.txt')
```

```
In [6]: #We will use 'take' method to create and print out a list of the first 2 data points

        totalData = songData.count()
        print 'Data Count: {0}'.format(totalData)
        print songData.take(1)

        t1 = time() - t0
        print 'Time taken is {0:.2f}'.format(t1)
```

```
Data Count: 515345
[u'2001,49.94357,21.47114,73.07750,8.74861,-17.40628,-13.09905,-25.01202,-12.23257,7.83089,-2.46783,3.32136,-2.31521,10.2
0556,611.10913,951.08960,698.11428,408.98485,383.70912,326.51512,238.11327,251.42414,187.17351,100.42652,179.19498,-8.415
58,-317.87038,95.86266,48.10259,-95.66303,-18.06215,1.96984,34.42438,11.72670,1.36790,7.79444,-0.36994,-133.67852,-83.261
65,-37.29765,73.04667,-37.36684,-3.13853,-24.21531,-13.23066,15.93809,-18.60478,82.15479,240.57980,-10.29407,31.58431,-25
.38187,-3.90772,13.29258,41.55060,-7.26272,-21.00863,105.50848,64.29856,26.08481,-44.59110,-8.30657,7.93706,-10.73660,-95
.44766,-82.03307,-35.59194,4.69525,70.95626,28.09139,6.02015,-37.13767,-41.12450,-8.40816,7.19877,-8.60176,-5.90857,-12.3
2437,14.68734,-54.32125,40.14786,13.01620,-54.40548,58.99367,15.37344,1.11144,-23.08793,68.40795,-1.82223,-27.46348,2.263
27']
Time taken is 12.49
```

```
In [3]: #We first 'split' the data entry on comma, as all the attributes are seperated by that.
        #Now will extract first 13 attributes from the songData RDD and display them.
        #Index[0] - Label
        #Index[1:13] - Timbre Features
```

∗ **Finding the minimum and maximum year - 1 node :**



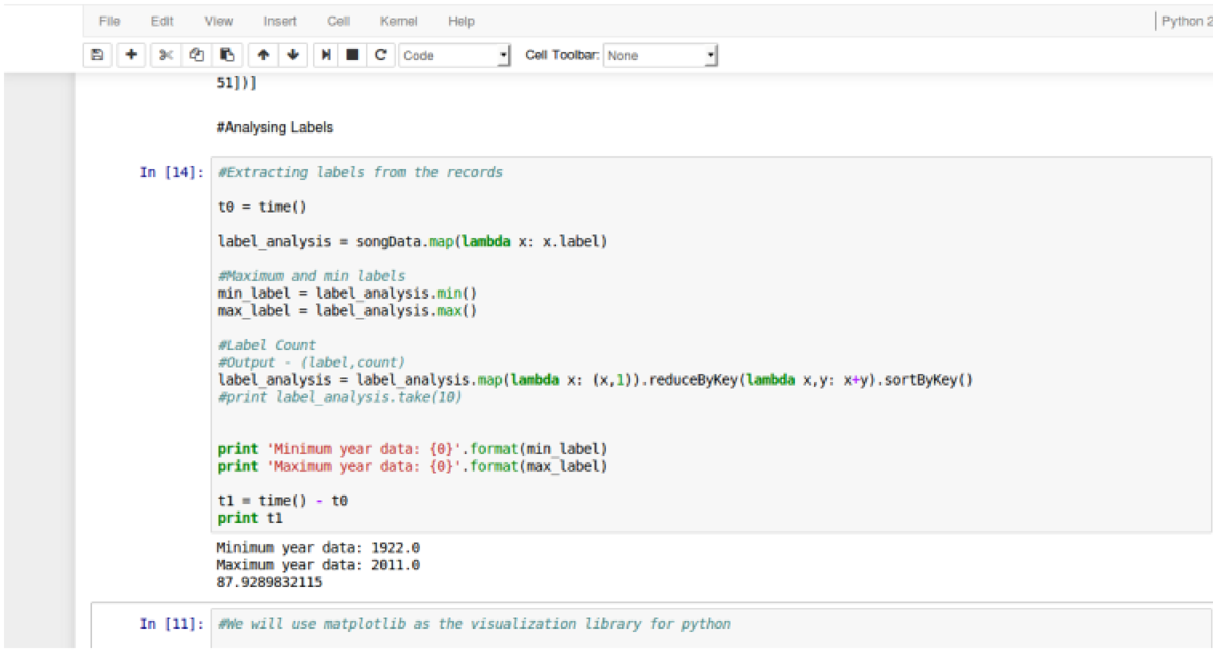
```
songData.take(1)
```

#Analysing Labels

In [10]:
```
#Extracting labels from the records

t0 = time()

label_analysis = songData.map(lambda x: x.label)

#Maximum and min labels
min_label = label_analysis.min()
max_label = label_analysis.max()

#Label Count
#Output - (label,count)
label_analysis = label_analysis.map(lambda x: (x,1)).reduceByKey(lambda x,y: x+y).sortByKey()
#print label_analysis.take(10)


print 'Minimum year data: {0}'.format(min_label)
print 'Maximum year data: {0}'.format(max_label)

t1 = time() - t0
print t1
```

```
Minimum year data: 1922.0
Maximum year data: 2011.0
338.120950937
```

∗ **Finding the minimum and maximum year - 5 nodes :**



```
51])]
```

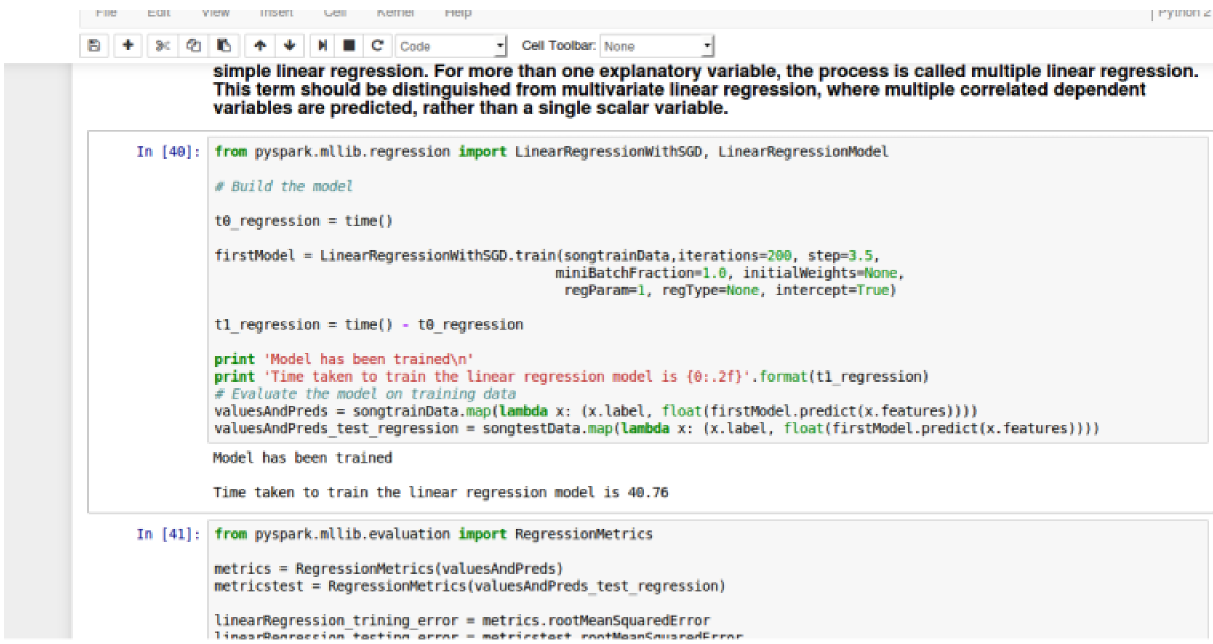
#Analysing Labels

In [14]:
```
#Extracting labels from the records

t0 = time()

label_analysis = songData.map(lambda x: x.label)

#Maximum and min labels
min_label = label_analysis.min()
max_label = label_analysis.max()

#Label Count
#Output - (label,count)
label_analysis = label_analysis.map(lambda x: (x,1)).reduceByKey(lambda x,y: x+y).sortByKey()
#print label_analysis.take(10)


print 'Minimum year data: {0}'.format(min_label)
print 'Maximum year data: {0}'.format(max_label)

t1 = time() - t0
print t1
```

```
Minimum year data: 1922.0
Maximum year data: 2011.0
87.9289832115
```

In [11]:
```
#We will use matplotlib as the visualization library for python
```

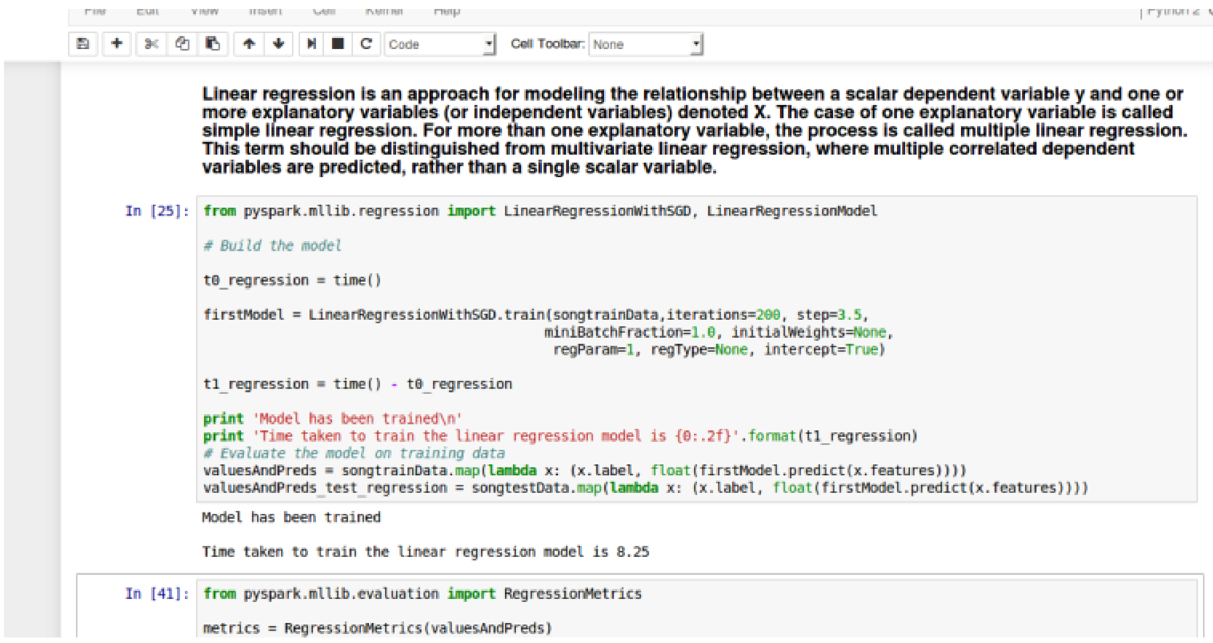
* **Linear Regression Model - 1 node :**



simple linear regression. For more than one explanatory variable, the process is called multiple linear regression. This term should be distinguished from multivariate linear regression, where multiple correlated dependent variables are predicted, rather than a single scalar variable.

```
In [40]: from pyspark.mllib.regression import LinearRegressionWithSGD, LinearRegressionModel

         # Build the model

         t0_regression = time()

         firstModel = LinearRegressionWithSGD.train(songtrainData,iterations=200, step=3.5,
                                                    miniBatchFraction=1.0, initialWeights=None,
                                                     regParam=1, regType=None, intercept=True)

         t1_regression = time() - t0_regression

         print 'Model has been trained\n'
         print 'Time taken to train the linear regression model is {0:.2f}'.format(t1_regression)
         # Evaluate the model on training data
         valuesAndPreds = songtrainData.map(lambda x: (x.label, float(firstModel.predict(x.features))))
         valuesAndPreds_test_regression = songtestData.map(lambda x: (x.label, float(firstModel.predict(x.features))))

         Model has been trained

         Time taken to train the linear regression model is 40.76

In [41]: from pyspark.mllib.evaluation import RegressionMetrics

         metrics = RegressionMetrics(valuesAndPreds)
         metricstest = RegressionMetrics(valuesAndPreds_test_regression)

         linearRegression_trining_error = metrics.rootMeanSquaredError
```

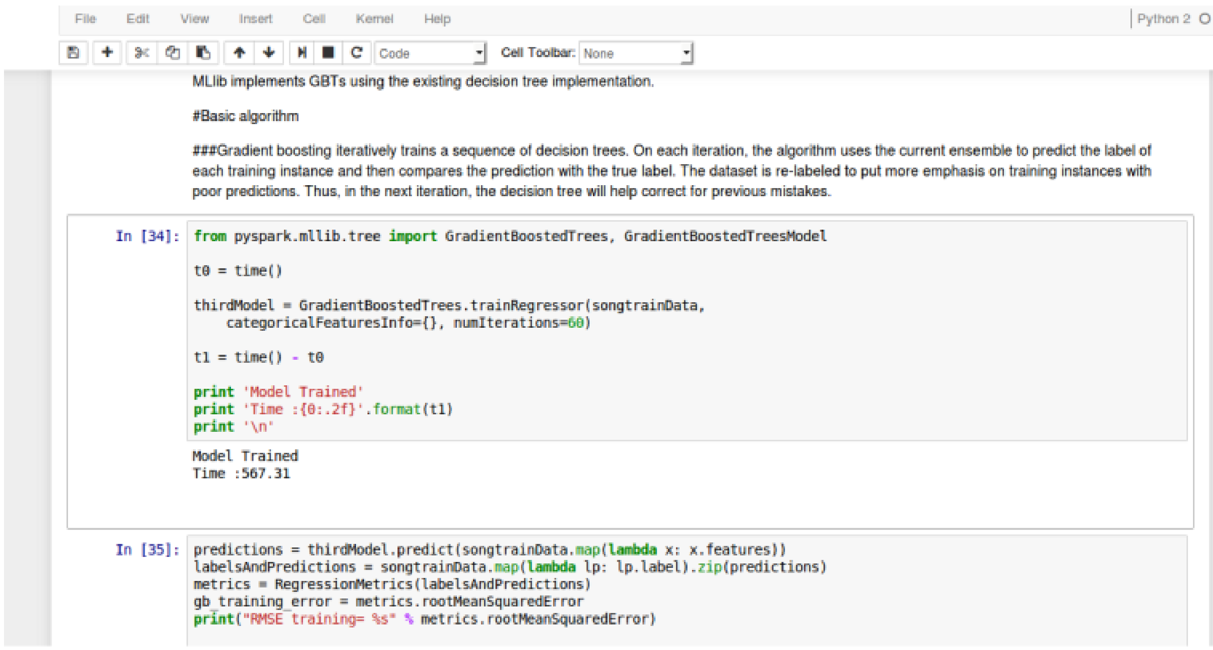
* **Linear Regression Model - 5 nodes :**



Linear regression is an approach for modeling the relationship between a scalar dependent variable y and one or more explanatory variables (or independent variables) denoted X. The case of one explanatory variable is called simple linear regression. For more than one explanatory variable, the process is called multiple linear regression. This term should be distinguished from multivariate linear regression, where multiple correlated dependent variables are predicted, rather than a single scalar variable.

```
In [25]: from pyspark.mllib.regression import LinearRegressionWithSGD, LinearRegressionModel

         # Build the model

         t0_regression = time()

         firstModel = LinearRegressionWithSGD.train(songtrainData,iterations=200, step=3.5,
                                                    miniBatchFraction=1.0, initialWeights=None,
                                                     regParam=1, regType=None, intercept=True)

         t1_regression = time() - t0_regression

         print 'Model has been trained\n'
         print 'Time taken to train the linear regression model is {0:.2f}'.format(t1_regression)
         # Evaluate the model on training data
         valuesAndPreds = songtrainData.map(lambda x: (x.label, float(firstModel.predict(x.features))))
         valuesAndPreds_test_regression = songtestData.map(lambda x: (x.label, float(firstModel.predict(x.features))))

         Model has been trained

         Time taken to train the linear regression model is 8.25

In [41]: from pyspark.mllib.evaluation import RegressionMetrics

         metrics = RegressionMetrics(valuesAndPreds)
```

* **Gradient Boosted Model - 1 node :**



MLlib implements GBTs using the existing decision tree implementation.

#Basic algorithm

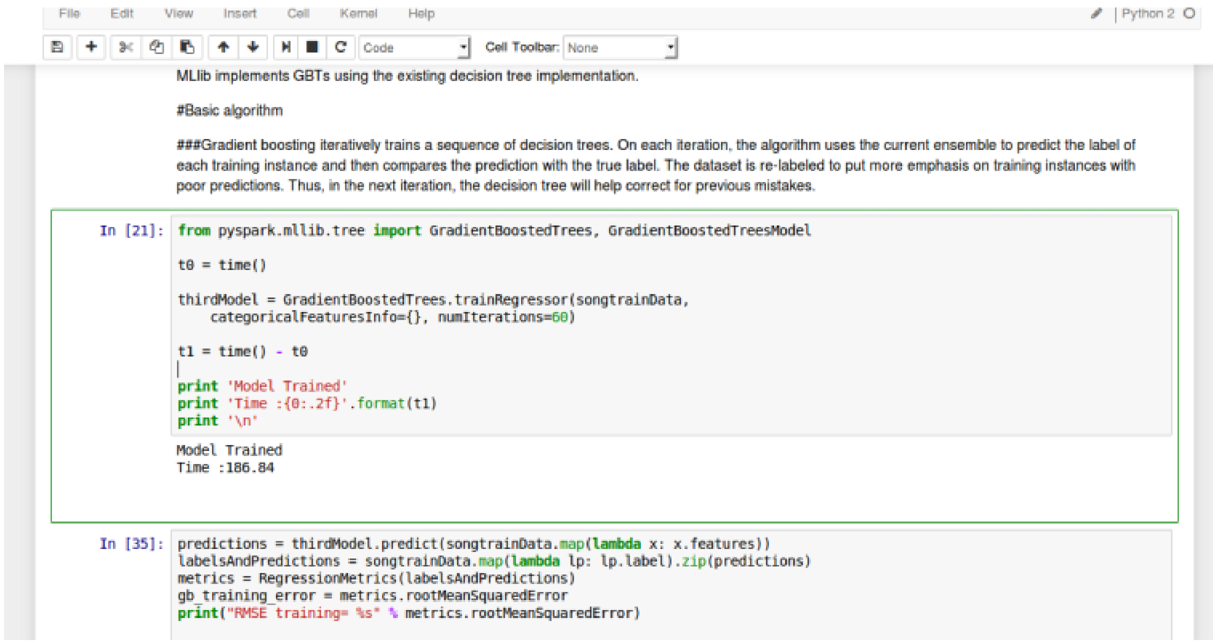
###Gradient boosting iteratively trains a sequence of decision trees. On each iteration, the algorithm uses the current ensemble to predict the label of each training instance and then compares the prediction with the true label. The dataset is re-labeled to put more emphasis on training instances with poor predictions. Thus, in the next iteration, the decision tree will help correct for previous mistakes.

In [34]:
```
from pyspark.mllib.tree import GradientBoostedTrees, GradientBoostedTreesModel

t0 = time()

thirdModel = GradientBoostedTrees.trainRegressor(songtrainData,
    categoricalFeaturesInfo={}, numIterations=60)

t1 = time() - t0

print 'Model Trained'
print 'Time :{0:.2f}'.format(t1)
print '\n'
```

```
Model Trained
Time :567.31
```

In [35]:
```
predictions = thirdModel.predict(songtrainData.map(lambda x: x.features))
labelsAndPredictions = songtrainData.map(lambda lp: lp.label).zip(predictions)
metrics = RegressionMetrics(labelsAndPredictions)
gb_training_error = metrics.rootMeanSquaredError
print("RMSE training= %s" % metrics.rootMeanSquaredError)
```

* **Gradient Boosted Model - 5 nodes :**



MLlib implements GBTs using the existing decision tree implementation.

#Basic algorithm

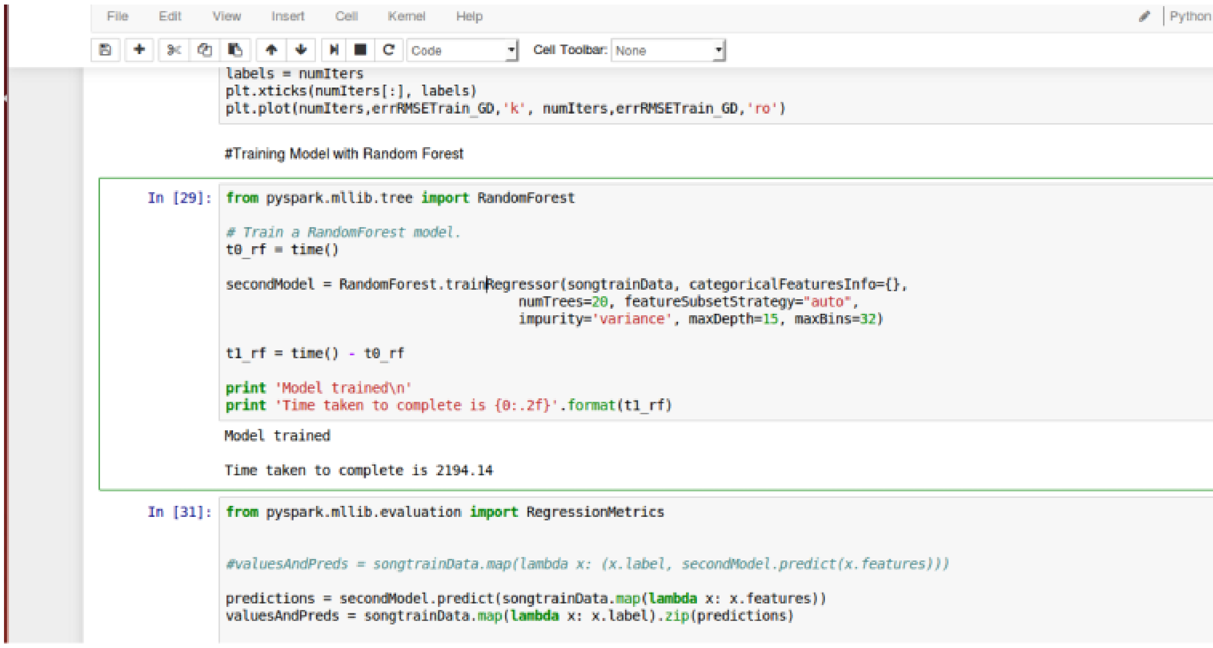
###Gradient boosting iteratively trains a sequence of decision trees. On each iteration, the algorithm uses the current ensemble to predict the label of each training instance and then compares the prediction with the true label. The dataset is re-labeled to put more emphasis on training instances with poor predictions. Thus, in the next iteration, the decision tree will help correct for previous mistakes.

In [21]:
```
from pyspark.mllib.tree import GradientBoostedTrees, GradientBoostedTreesModel

t0 = time()

thirdModel = GradientBoostedTrees.trainRegressor(songtrainData,
    categoricalFeaturesInfo={}, numIterations=60)

t1 = time() - t0

print 'Model Trained'
print 'Time :{0:.2f}'.format(t1)
print '\n'
```

```
Model Trained
Time :186.84
```

In [35]:
```
predictions = thirdModel.predict(songtrainData.map(lambda x: x.features))
labelsAndPredictions = songtrainData.map(lambda lp: lp.label).zip(predictions)
metrics = RegressionMetrics(labelsAndPredictions)
gb_training_error = metrics.rootMeanSquaredError
print("RMSE training= %s" % metrics.rootMeanSquaredError)
```

* **Random Forest Model - 1 node :**



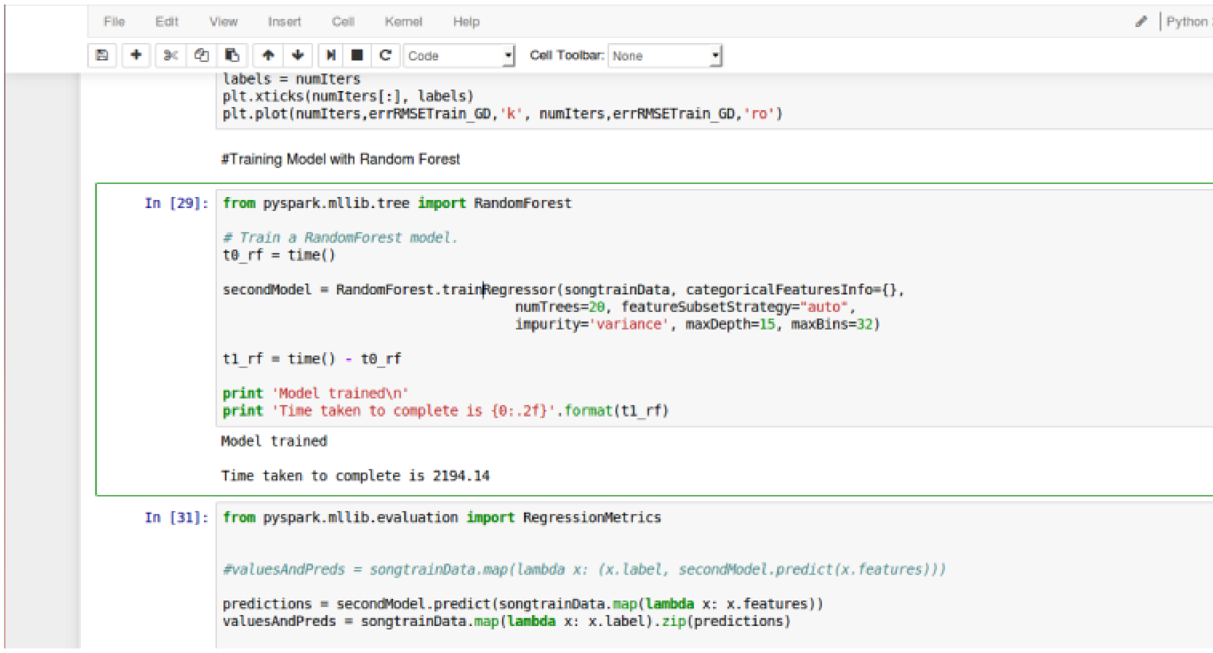
```
labels = numIters
plt.xticks(numIters[:], labels)
plt.plot(numIters,errRMSETrain_GD,'k', numIters,errRMSETrain_GD,'ro')
```

#Training Model with Random Forest

```
In [29]: from pyspark.mllib.tree import RandomForest

         # Train a RandomForest model.
         t0_rf = time()

         secondModel = RandomForest.trainRegressor(songtrainData, categoricalFeaturesInfo={},
                                             numTrees=20, featureSubsetStrategy="auto",
                                             impurity='variance', maxDepth=15, maxBins=32)

         t1_rf = time() - t0_rf

         print 'Model trained\n'
         print 'Time taken to complete is {0:.2f}'.format(t1_rf)

         Model trained

         Time taken to complete is 2194.14
```

```
In [31]: from pyspark.mllib.evaluation import RegressionMetrics

         #valuesAndPreds = songtrainData.map(lambda x: (x.label, secondModel.predict(x.features)))

         predictions = secondModel.predict(songtrainData.map(lambda x: x.features))
         valuesAndPreds = songtrainData.map(lambda x: x.label).zip(predictions)
```

* **Random Forest Model - 5 node :**



```
labels = numIters
plt.xticks(numIters[:], labels)
plt.plot(numIters,errRMSETrain_GD,'k', numIters,errRMSETrain_GD,'ro')
```

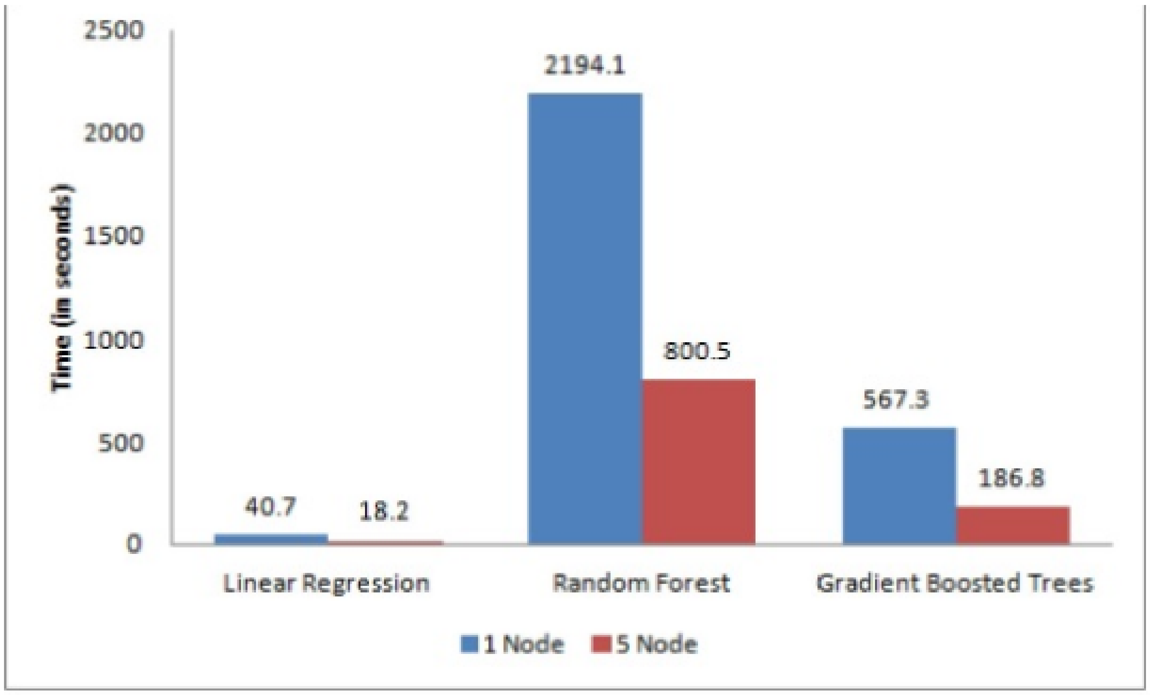
#Training Model with Random Forest

```
In [29]: from pyspark.mllib.tree import RandomForest

         # Train a RandomForest model.
         t0_rf = time()

         secondModel = RandomForest.trainRegressor(songtrainData, categoricalFeaturesInfo={},
                                             numTrees=20, featureSubsetStrategy="auto",
                                             impurity='variance', maxDepth=15, maxBins=32)

         t1_rf = time() - t0_rf

         print 'Model trained\n'
         print 'Time taken to complete is {0:.2f}'.format(t1_rf)

         Model trained

         Time taken to complete is 2194.14
```

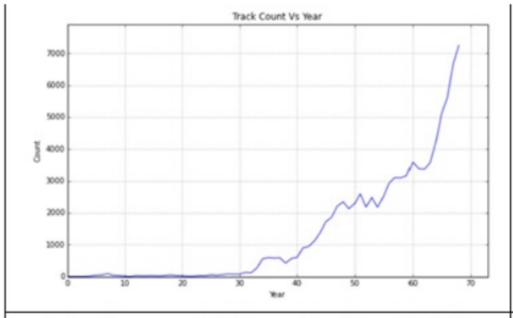
```
In [31]: from pyspark.mllib.evaluation import RegressionMetrics

         #valuesAndPreds = songtrainData.map(lambda x: (x.label, secondModel.predict(x.features)))

         predictions = secondModel.predict(songtrainData.map(lambda x: x.features))
         valuesAndPreds = songtrainData.map(lambda x: x.label).zip(predictions)
```

6 Results

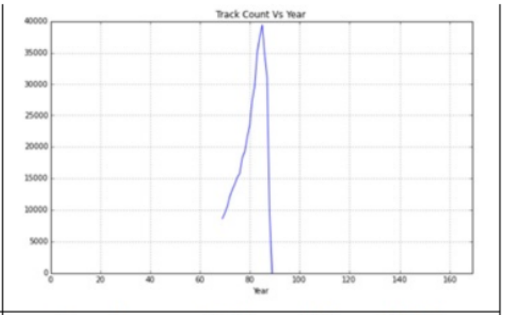
For the purpose of implementation, Apache 2.3.0 single node cluster with a master and worker set was used and for multinode, a set of 5 nodes i.e one master and 4 workers, was used.



The above figure shows the time comparison of single node and multiple nodes during training of predictive models. Given below is the diagrammed representation of the number of records per year, after the segregation of datasets.

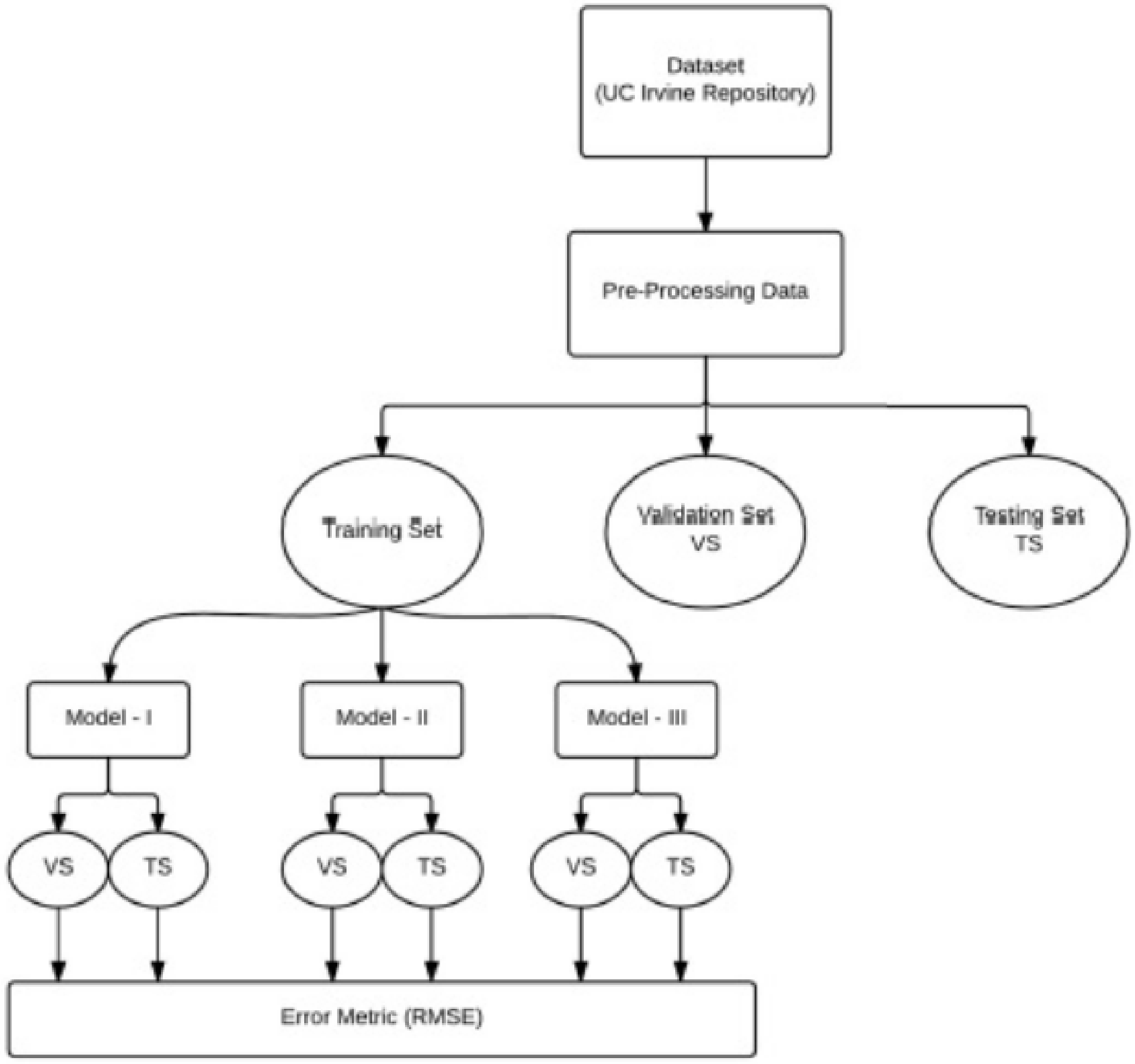


Track count VS Year (before 1990)

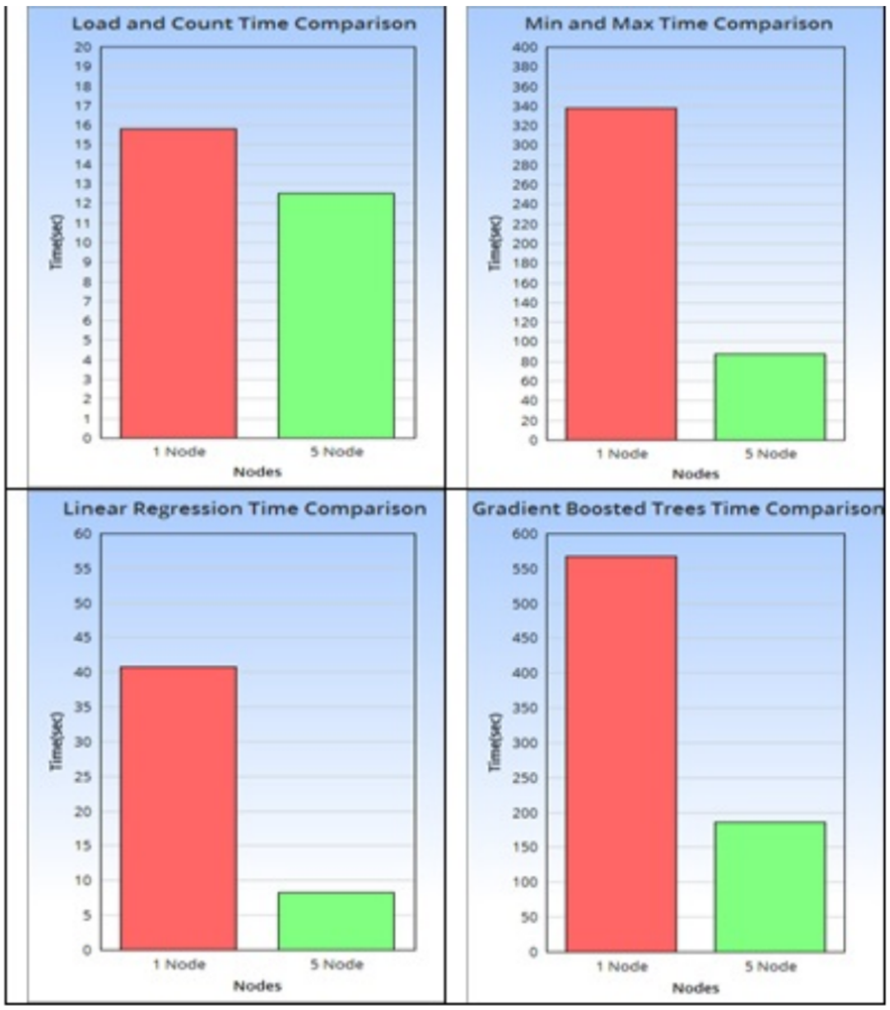


Track count VS Year (after 1990)

Following is the algorithmic view of the work done



Graphical Representation of Time comparison of computation



Following is the table for comparing the accuracy of the models through RMSE

MODEL	RMSE
Baseline	10.93
Linear Regression	10.01
Random Forest	9.66
Gradient Boosted Trees	9.77

Table 2: Model RMSE Comparison

7 Conclusions

From results, it is clear that, distributed computing greatly reduces the computation time of training our models. Thus, we highly recommend the use of Apache Spark framework, for faster results. Apart from that, we have been also able to establish which algorithm, can be best used for the prediction.
Random Forest performed the best in terms of accuracy calculated in terms of RMSE. Linear Regression performed well both in single and multi-node cluster with the least training time. On an average, multi-node cluster ran the the built models 2.5X faster in comparison to a single node cluster.
By creating a system that predicts the release year of a particular song, we can develop a more robust model for a song recommender system. Conventional recommender systems, generally take into account just the user ratings, but integrating year prediction, we can also include audio features.

8 Future Scope

In our proposed work, we used a sub-set of the million song dataset. In future, we can scale up the system for all one million data samples using the Amazon Web Services (AWS). We can also investigate the trend of time complexity by increasing the number of nodes in the cluster step by step.
Furthermore, relationships can also be established between the instruments used some time back and now, by further studying the timbre features of instruments as well.

References

[1] https://archive.ics.uci.edu/ml/datasets/yearpredictionmsd.

[2] https://cseweb.ucsd.edu/classes/wi17/cse258 a/reports/a028.pdf.

[3] https://vdocuments.mx/music-similarity-measures-whats-the use.html.

[4] https://www.ee.columbia.edu/ dpwe/pubs/BertEWL11 msd.pdf.

[5] https://www.kaggle.com/vinayshanbhag/predict-release-timeframe-from-audio features/data.