MidTerm Assignment

Name – Tanmay Ingle, Bhaskar Akula

Case1: Classification and Regression

Our goal here is to predict the year of the song based on multiple features. There are 90 features, 12 = timbre average, 78 = timbre covariance. The first value is the year (target), ranging from 1922 to 2011.

Preprocessing

Label Shifting

We first parse the data and convert it into LabeledPoint format.

In [3]:
```
data = sc.textFile("D://NEU - Big Data and Intelligent Analytics/Midterm/YearPredictionMSD.txt")
```

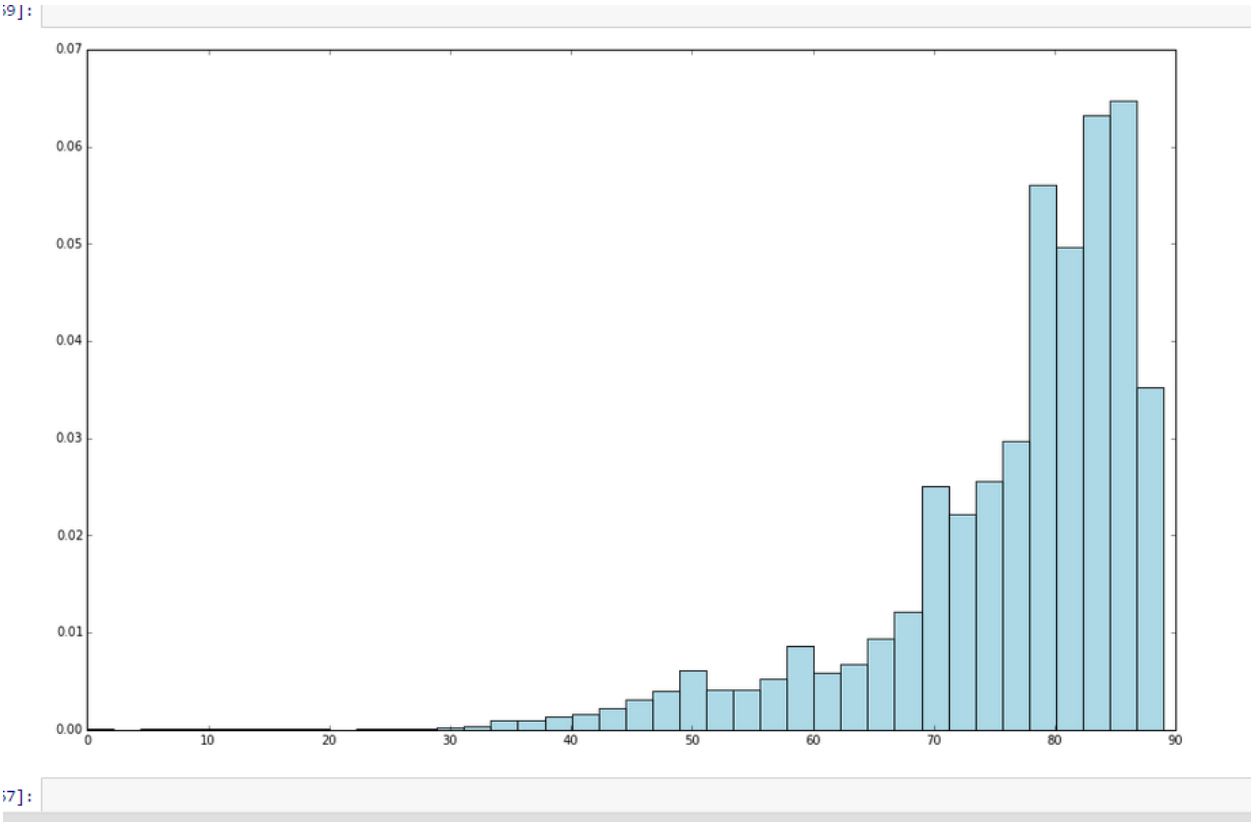
In [4]:
```
#def parsePoint(line):
 #   values = [float(x) for x in line.replace(';', ' ').split(' ')]
  #  return LabeledPoint(values[0], values[1:])

def line_to_fields(line):
    space_separated_line = line.replace(',', ' ')
    string_array = space_separated_line.split(' ')
    float_array = map(float, string_array)
    return LabeledPoint(float_array[0], float_array[1:])
```

In [5]:
```
parsedData = data.map(line_to_fields)
```

We plotted a histogram of the target variable i.e. Year of the Song. As it is evident range of the target variable goes from 1922 to 2011. Thus we need to do Label Shifting for better results. We subtract 1922 from each of the target value.

6]:
```
parsedData1 = parsedData.map(lambda x: LabeledPoint((x.label-1922.0),x.features))
```



Thus we can see that our target variable is skewed towards left.

We then checked summary statistics for columns and check various metrics like mean variance of columns.

Next we used the function take() and count() and explored the data further to get acquainted with it.

We observed that the range of various features differ.Thus the data is scaled so that all the variables have equal mean = 0 and variance = 1.

```
In [7]: from pyspark.mllib.util import MLUtils
        from pyspark.mllib.linalg import Vectors
        from pyspark.mllib.feature import StandardScaler

        label1 = parsedData1.map(lambda x: x.label)
        features1 = parsedData1.map(lambda x: x.features)

        scaler2 = StandardScaler(withMean=True, withStd=True).fit(features1)
        data2 = label1.zip(scaler2.transform(features1.map(lambda x: Vectors.dense(x.toArray()))))
```

```
In [79]: #data2.take(3)
```

```
Out[79]: [(79.0,
           DenseVector([1.0806, 0.3913, 1.8265, 0.4647, -0.4747, -0.2782, -1.5524, -1.3108, 0.3877, -0.6662, 0.7934, -0.5843, -1.0561,
          -1.0451, -0.8059, -0.7474, -1.0553, -0.8588, -0.8731, -0.9034, -0.666, -0.8362, -1.008, -0.7348, -0.4237, -0.5046, 0.2612, 0.34
          7, -0.6778, -0.4639, -0.0319, 0.1447, 0.0299, 0.1036, 0.1717, -0.6767, -0.1981, -0.4437, 0.5854, 0.2428, -0.3011, -0.1776, 0.376
          8, -0.429, 0.4195, -0.4536, 0.0073, 0.3731, 0.3638, 0.0519, -0.3395, -0.4291, 0.0074, 0.4785, 0.0509, -0.3108, 0.0022, 0.2411,
          -0.0745, -0.1152, -0.1953, 0.1551, -0.2723, 0.1388, -0.3661, -0.2796, 0.0154, 0.3712, -0.0351, 0.1863, -0.1121, -0.2007, 0.1156,
          0.3024, 0.2005, -0.0126, 0.0409, -0.1139, 0.2518, 0.1065, -0.0853, 0.1085, 0.1428, -0.2374, 0.0492, -0.3562, 0.5445, -0.4706,
          -0.256, 0.0423])),
           (79.0,
           DenseVector([0.8809, 0.3323, 1.7485, 0.7218, -0.1649, -1.1912, 0.7657, 0.1096, 1.4209, 0.4149, 0.5413, -0.2651, 0.4796, -0.218
          6, -1.0799, -0.9676, -0.2813, -0.8035, 0.4496, -0.4885, -0.3557, -0.3178, -0.7869, 0.1654, -0.1939, -0.1517, 0.3839, 0.745, -0.6
          816, -0.9741, 0.1236, 0.3046, 0.3091, -0.5358, 1.0499, -0.4744, -0.6051, -0.3521, 0.1958, 0.8705, -0.2684, 1.4761, 0.4758, -0.11
```

Model Building(Regression)

We split the data into 70% training data and 30%test data.

```
]: #Spliting data into training and test data
   (trainingData, testData) = z.randomSplit([0.7, 0.3])
```

We built our models by using three types of regression techniques namely Linear Regression with SGD, Ridge Regression and Lasso Regression. The metric which we used was to compare the result was Root Mean Square Error. We computed the same for training and test data

We got following results for RMSE:-

Linear Regression with SGD – 10.5

RIdgeRegression with SGD – 9.55

Lasso Regression with SGD – 10

Interpretation

Thus as we can see, we are off from our target available on an average about 10 years. RidgeRegression gave the best error. However Lasso Regression gave a similar error but with less number of features since it assigns 0 as weight to less important features, thereby reducing the cost to collect them in future.

Model Building(Classification)

We split the target variable into less than 1965 and greater than 1965, while parsing itself.

```
n [4]: #def parsePoint(line):
        #   values = [float(x) for x in line.replace(';', ' ').split(' ')]
         #  return LabeledPoint(values[0], values[1:])

       def line_to_fields(line):
           space_separated_line = line.replace(',', ' ')
           string_array = space_separated_line.split(' ')
           float_array = map(float, string_array)
           if float_array[0]<1965.0:
               return LabeledPoint(0.0, float_array[1:])
           else:
               return LabeledPoint(1.0, float_array[1:])
```

```
n [5]: parsedData = data.map(line_to_fields)
```

This was followed by same preprocessing like that of regression.

Then we tried building Logisitic Regression with SGD ,SVM with SGD and Logistic Regression with LBFGS. We got the best results for SVM with SGD.

The metrics which we used were Accuracy for training and test data and most importantly the Area under ROC curve and Area under PR curve since they portray how good our classifier is. We also computed the confusion matrix to understand our results.

The metrics for SVM with SGD were as follows:-

Accuracy(training data) – 0.65

Accuracy (test data) – 0.66

Area Under the curve = 0.84

Area under Precision Recall Curve – 0.91

We used PCA and SVD in Scala since it is not yet offered in IPYTHON. We implemented it in Intellij IDEA. However we faced challenges with computation time since it is not interactive. Thus every time we changed a parameter we had to run it from scratch and do preprocessing everytime. Thus it took around 40 -45 mins to run one model. Tuning was very tough because of this issue. Solution to this according to us is as follows:-

1) Instead of running locally, it should be run on a cluster with distributed computation.
2) ML pipelines should be used to create pipelines of preprocessing and parsing and then building model sepearately.

Case:3 Clustering

We used a dataset of TV commericals for exploring clustering. There are 5 files in the dataset namely :- BBC, CNN,CNNIBN, NDTV and TIMESNOW. All the three files are in LIBSVM format instead of regular csv format. LIBSVM format is most useful for sparse data where most of the values are zeroes.

Preprocessing

Since we wanted to perform clustering on all the files, first step was to combine all the data using union function.

```
cnn = "D://NEU - Big Data and Intelligent Analytics/midterm-clustering/CNN.txt"
cnn1 = MLUtils.loadLibSVMFile(sc, cnn)
```

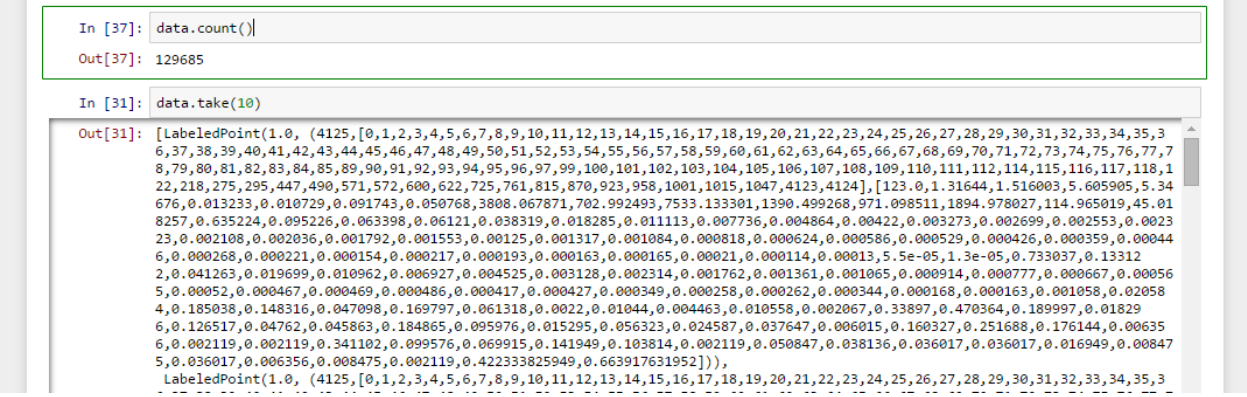
```
x = points.union(cnn1)
```

```
cnnibn = "D://NEU - Big Data and Intelligent Analytics/midterm-clustering/CNNIBN.txt"
cnnibn1 = MLUtils.loadLibSVMFile(sc, cnnibn)
```

```
y = x.union(cnnibn1)
```

This way we combined all 5 files into one RDD.

We counted the lines in the data and explored it using take(10) function.



As we can see the count of data is equal to addition of count of all the five files which we used.

One of the important things which needs to be care of in clustering is standardization of data. And as it can be seen from first values that different features have different range of values. Thus we performed standardization next.

```
In [16]: label = data.map(lambda x: x.label)
         features = data.map(lambda x: x.features)
```

```
In [26]: scaler2 = StandardScaler(withMean=True, withStd=True).fit(features)
         data2 = scaler2.transform(features.map(lambda x: Vectors.dense(x.toArray())))
```

Model Building

We used K means clustering for this purpose. We built a model with max iterations = 10 and K = 5,6,8,10,15.

```
In [38]: # Build the model (cluster the data)
         clusters = KMeans.train(data2, 6, maxIterations=10,
                 runs=10, initializationMode="random")
```

The metric which we used for selecting the optimal K was Within Set Sum of Squared Error (WSSSE). This is basically sum of squared of Euclidean distance between every datapoint and the centroid of the cluster to which it belongs.

```
In [39]: def error(point):
             center = clusters.centers[clusters.predict(point)]
             return sqrt(sum([x**2 for x in (point - center)]))
```

```
In [40]: WSSSE = data2.map(lambda point: error(point)).reduce(lambda x, y: x + y)
         print("Within Set Sum of Squared Error = " + str(WSSSE))
```

```
Within Set Sum of Squared Error = 1469175.06205
```

K values and WSSSE values are as follows-

K	WSSSE
5	1498927.71402
6	1469175.06205
8	1456573.80642
10	1426056
15	1403674.79345

Thus k = 15 is optimal value.

Case2: Determine Income Based on characteristics:

Our goal is to use classification methods and determine whether a person has income >50k or <= 50k.

The dataset we have used for this Case Study is the “Adult” dataset hosted on UCI’s Machine Learning Repository. It contains approximately 32000 observations, with 15 variables. The dependent variable that in all cases we will be trying to predict is whether or not an “individual” has an income greater than $50,000 a year.

Preprocessing

Handling Missing or corrupted data: As it was mentioned in the description of the dataset that missing values are replaced by ‘?’. So we need to handle it accordingly. Since most of the features which are present are categorical we decided that best way to handle this missing data is to replace it with mode of that column i.e. the value which is present most number of times in that column.

Next we observed that one of the columns named education_nums was actually portraying same information as that in education columns. Thus we dropped that column.

Third step was to convert categorical variables into numerical variables and then convert them to dummy variables.

After preprocessing we had 32531 rows and 105 columns to work on.

Model Building

Then we converted the last feature >50k or <= 50k to 1 and 0 respectively and parse the function.

```
def line_to_fields(line):
    space_separated_line = line.replace(',', ' ')
    string_array = space_separated_line.split(' ')
#    float_array = map(float, string_array)
#    return LabeledPoint(float_array[104],float_array[0:103])
    if string_array[-1] == '<=50K':
        return LabeledPoint(0.0, string_array[1:105])
    else:
        return LabeledPoint(1.0, string_array[1:105])
```

Then we standardize the data using StandardScalar and split the data into 70% training data and 30% test data.

We compute 5 models on this data and tune each one of them to their best metrics. Metrics of these 5 models are as follows:-

Results			
	AUC-ROC(test)	AUC-PR(test)	Accuracy(test)
LogReg(L1)	0.896	0.751	0.804
LogReg(L2)	0.896	0.753	0.803
LogReg(LBFGS)	0.888	0.712	0.831
SVM(L1)	0.869	0.695	0.818
SVM(L2)	0.875	0.699	0.829

We have used three metrics for classification – Accuracy, ROC-Area Under curve and Area under Precision-Recall curve.

Accuracy is the number of correctly classified examples divided by the total examples. Since the distribution of categories in target variable is uneven, accuracy is not the only metric to be looked at. (E.g. if there are 100 records and 80 of them are 1s and 20 are 0s. Then even if our model has 80% accuracy, it is still a useless model since it is predicting 1 for all records). Thus to calculate how good a classifier a model is, we use the other two metrics.

The area under the ROC curve (commonly referred to as AUC) represents an average value. An AUC of 1.0 will represent a perfect classifier. An area of 0.5 is referred to as the random score. Thus, a model that achieves an AUC of 0.5 is no better than randomly guessing. Thus we tune our models to have maximum AUC even though we have to compromise on accuracy.

The area under PR curve is referred to as the average precision. An area under the PR curve of 1.0 will equate to a perfect classifier that will achieve 100 percent in both precision and recall. Thus it should as close to 1 as possible.

As it can be seen from the results that LogRegwith LBFGS has almost same AUC score as others but has the maximum Accuracy on test data.

The tuned parameters for 5 models are as follows:-

```
In [725]: # Build the model
          model = LogisticRegressionWithSGD.train(trainingData, iterations=250,step=5,regParam= 0.001,regType='l2')
```

```
# Build the model
model = LogisticRegressionWithSGD.train(trainingData, iterations=250,step=5,regParam= 0.001,regType='l1')
```

```
5]: # Build the model
    model = LogisticRegressionWithLBFGS.train(training, iterations=250,regParam= 0.00001)
```

```
903]: # Build the model and passing the parameter regParam as L1 for L1 regularisation
      model = SVMWithSGD.train(training, iterations=250,step=5,regType="l1", regParam=0.01,)
```

```
[935]: # Build the model
       model = SVMWithSGD.train(training, iterations=250,step=10,regParam= 0.001,regType='l2')
```

Proper regularization was used and optimized regParams.

Lessons learnt and challenges faced

- Using Scala for such a huge a data size was very challenging since it is not intereactive. Thus debugging and tuning parameters is tough and time consuming when used locally.
- Solution to this problem as mentioned in the report is to use ML pipelines if possible and/or use distributed cluster for faster computation.
- PCA and SVD not offered in IPYTHON. It will be convenient when that happens.
- We faced preprocessing issues in IPYTHON Notebook when we tried to use filter() with distinct and count() functions of spark. The error of ‘list index out of range’ is something which took most of our time to debug but eventually didn’t resolve.
- Since it the datasize was large it wasn’t caching in memory leading to its computation every time an action was called.
- We could use K-means properly however with the same parsed data, Gaussian mixture gave an error of “Java heap space”.