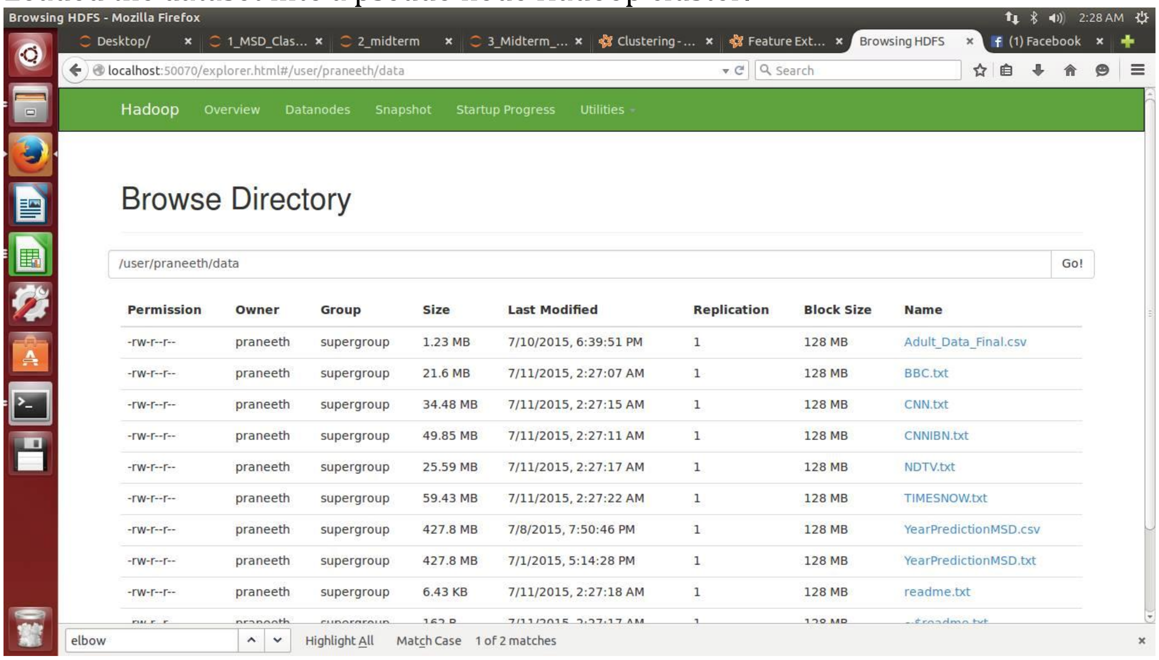
Submitted by: Praneeth Krishna, Prateek Gangwal, Sakshi Arora

MillionSongData

Preprocessing Steps:

- Loaded the dataset into a pseudo node Hadoop cluster.



- Created a parse function, which parses through every line separated by delimiter (,) and written labeled points of features and labels.
- Data Caching is done using .cache to increase the performance.
- Data Exploration is done using mllib.stat library and performed function like mean, numberOfZeroes, variance, count etc among the data points.
- Calculated the Max and Min year to shift the labels according to our data requirements using the following code:

```
parsedDataInit = rawData.map(parsePoint) onlyLabels = parsedDataInit.map(lambda a: a.label)
minYear = onlyLabels.min() maxYear = onlyLabels.max() print maxYear, minYear

#Shifting labels
parsedData = NewData.map(lambda a: LabeledPoint((a[0] - 1922),a[1:]))
print type(parsedData.take(1)[0])
# View the first point
print '\n{0}'.format(parsedData.take(1))
```

Min -> 1922

Max -> 2011

- Normalized the data using the normalizer in the mllib.feature using the following code:

```
from pyspark.mllib.feature
import VectorTransformer from pyspark.mllib.feature
```

```
import Normalizer
```

- Parsed the map function and created a binary variable for <1965 as 0 and >=1965 as 1 using the following code:

```
parsedData = NewData.map(lambda a: LabeledPoint((1 if a[0] >=1922 else 0),a[1:]))
```

No Feature Reduction Regression

- Data is spitted into training, validation and test data sets using randomSplit() using the following code.

```
weights = [.6, .2, .2]
seed = 42
parsedTrainData, parsedValData, parsedTestData = parsedData.randomSplit(weights,seed)
parsedTrainData.cache()
parsedValData.cache()
parsedTestData.cache()
```

- Created a baseline model by taking average from all the years.
- Evaluated baseline model using Evaluation matrix RMSE using the following code:

```
labelsAndPredsTrain = parsedTrainData.map(lambda x:(x.label,76.39463))
rmseTrainBase = calcRMSE(labelsAndPredsTrain)
```

- Evaluated the model Base Line Model, LinearRegressionwithSGD, RidgeRegressionwithSGD, LassowithSGD.
- Compared the models using RMSE.

Analysis: LinearRegregressionwithSGD outperforms other models as it has the least RMSE of 15.71

No Feature Reduction Classification

- Evaluated the model SVMwithSGD, LogisticRegressionwithLBFGS, LogisticRegressionwithSGD.
- Compared the models using following metrics:

```
AreaUnderCurver
ConfusionMatrix
Error Evaluation
```

Feature Engineering PCA

- Implemented PCA in scala as Pyspark doesn't support PCA.
- Top 20 variables were generated as output from PCA analysis.

- Applied the Regression and Classification models after doing feature engineering with PCA.

Analysis: Regression and Classification models after PCA outperforms other models without feature reduction.

Income Classification Problem

Preprocessing Steps:

- The categorical variables were converted into numeric variables using the following steps:
    - Factorized the data using scala using the following code:

```
def cleanData(i : String): String = {
val a1 = i.split(",") val b1 : List[AnyVal] =
List(a1(0).toDouble,convertCol1toInt(a1(1)),a1(2).toDouble,convertCol4toInt(a1(3)),a1(4).t
oDouble,convertCol6toInt(a1(5)),convertCol7toInt(a1(6)),convertCol8toInt(a1(7)),convertC
ol9toInt(a1(8)),convertCol10toInt(a1(9)),a1(10).toDouble,a1(11).toDouble,a1(12).toDouble
, convertCol14toInt(a1(13)),convertCol15toInt(a1(14)))
b1.mkString(",")
}
```

    - The function *convertColtoInt* function is defined as follows:

```
def convertCol15toInt(s : String): Double =
{
 val a2 = s match { case " <=50K" => 0 case " >50K" => 1 } a2
}
```

    - The categorical missing values were replaced by Mode function.
    - Created dummy variables using the following code:

```
features_dummies = pd.get_dummies(features1)
```

Applying SVMwithSGD Classification:

- Applied the model using the following code:

```
from pyspark.mllib.classification
import SVMWithSGD, SVMModel
# Build the model model = SVMWithSGD.train(parsedTrainData,
iterations=1000,step=0.001,regParam=0.01)
```

- Evaluated the model on training data using the following code.

```
# Evaluating the model on training data
```

```
labelsAndPreds = parsedTrainData.map(lambda p: (p.label, float(model.predict(p.features))))
trainErr = labelsAndPreds.filter(lambda (v, p): v != p).count() /
float(parsedTrainData.count()) print("Training Error = " + str(trainErr))
```

- Performed model evaluation by observing *area under curve, confusion matrix and validation error.*

```
metrics = BinaryClassificationMetrics(labelsAndPreds) AUC = metrics.areaUnderROC APR =
metrics.areaUnderPR print("train AreaUnderCurve = " + str(AUC))
```

```
p = np.array(labelsAndPredsval).collect() confusion_matrix(p[:,0],p[:,1])
```

Applying LogicsticRegressionwithLBFGS Classification:

- Applied the model using the following code:

```
model = LogisticRegressionWithLBFGS.train(parsedTrainData,iterations=1000,regParam=0.01)
```

- Evaluated the model on training data using the following code.

```
# Evaluating the model on training data
labelsAndPreds = parsedTrainData.map(lambda p: (p.label, float(model.predict(p.features))))
trainErr = labelsAndPreds.filter(lambda (v, p): v != p).count() /
float(parsedTrainData.count())
print("Training Error = " + str(trainErr))
```

- Performed model evaluation by observing *area under curve, confusion matrix and validation error.*

```
metrics = BinaryClassificationMetrics(labelsAndPreds)
AUC = metrics.areaUnderROC
APR = metrics.areaUnderPR
print("train AreaUnderCurve = " + str(AUC))
```

```
p = np.array(labelsAndPreds.collect()) confusion_matrix(p[:,0],p[:,1])
```

Applying Decision Tree Classification:

- Applied the model using the following code:

```
model = GradientBoostedTrees.trainClassifier(parsedTrainData, categoricalFeaturesInfo={},
numIterations=3)
```

- Evaluated the model on training data using the following code.

```
# Evaluate model on train instances and compute test error predictions =
model.predict(parsedTrainData.map(lambda x: x.features)) labelsAndPredictions =
parsedTrainData.map(lambda lp: lp.label).zip(predictions) testErr =
labelsAndPredictions.filter(lambda (v, p): v != p).count() / float(parsedTrainData.count()) print('Test
Error = ' + str(testErr)) print('Learned classification GBT model:') print(model.toDebugString())
```

- Performed model evaluation by observing *area under curve, confusion matrix and validation error.*

```
metrics = BinaryClassificationMetrics(labelsAndPredictions) AUC = metrics.areaUnderROC APR =
metrics.areaUnderPR print("train AreaUnderCurve = " + str(AUC))
```

```
p = np.array(labelsAndPredictions.collect()) confusion_matrix(p[:,0],p[:,1])
```

TV commercial Clustering

Preprocessing Steps:

- Combining the 5 Lib svm files into 1 Lib svm file using the following code:

```
cnn = "/home/praneeth/Downloads/Case3_TvNews/CNN.txt"
cnn1 = MLUtils.loadLibSVMFile(sc, cnn) x = points.union(cnn1)
cnnibn = "/home/praneeth/Downloads/Case3_TvNews/CNNIBN.txt"
cnnibn1 = MLUtils.loadLibSVMFile(sc, cnnibn) y = x.union(cnnibn1)
ndtv = "/home/praneeth/Downloads/Case3_TvNews/NDTV.txt"
ndtv1 = MLUtils.loadLibSVMFile(sc, ndtv) z = y.union(ndtv1)
tn = "/home/praneeth/Downloads/Case3_TvNews/TIMESNOW.txt"
tn1 = MLUtils.loadLibSVMFile(sc, tn) data = z.union(tn1)
```

- Normalized the data using the following code:

```
scaler2 = StandardScaler(withMean=True, withStd=True).fit(features)
```

Applying K means:

- The union and normalized file is used for the k means algorithm.

- The model is built using the in-built k means algorithm by specifying the max iterations and run.

```
clusters = KMeans.train(data2, 2, maxIterations=10, runs=10, initializationMode="random”)
clusters = KMeans.train(data2, 4, maxIterations=10, runs=10, initializationMode="random”)
clusters = KMeans.train(data2, 6, maxIterations=10, runs=10, initializationMode="random”)
clusters = KMeans.train(data2, 8, maxIterations=10, runs=10, initializationMode="random”
clusters = KMeans.train(data2, 10, maxIterations=10, runs=10, initializationMode="random”
```

- Evaluating the cluster by computing within set sum of squared errors.

```
from math import sqrt from numpy import array def error(point): center =
clusters.centers[clusters.predict(point)] return sqrt(sum([x**2 for x in (point - center)])) WSSSE =
data2.map(lambda point: error(point)).reduce(lambda x, y: x + y) print("Within Set Sum of Squared
Error = " + str(WSSSE))
```

Analysis:

- **Within Set Sum of Squared Error (2) = 1605485.841**
- **Within Set Sum of Squared Error (4) = 1549285.453**
- **Within Set Sum of Squared Error (6) = 1460567.116**
- **Within Set Sum of Squared Error (8) = 1438714.297**
- **Within Set Sum of Squared Error (10) = 1425614.813**