Final Project
CS6240
Himanshi Bhardwaj and Charles Heisler
12/10/2016

Tools and Model

We used Apache Spark and its MLlib machine learning library's implementation of random forests.

Data

Excluded Features

All ID features (e.g. sampling event ID's) were excluded. All categorical features were also excluded because they were such a small portion of the overall data set and many of them contained a very large number of categories (e.g counties and ecoregions).

Missing Values

Missing values for features representing bird counts were replaced with 0, others missing values were replaced with the mean for that feature.

Parsing

Data from the archives was represented sparsely, since most most features corresponded to counts of bird species sightings and most such features had values of 0. Each line of the original CSV was loaded into an RDD, and then each value in the RDD was mapped to a sparse representation as an array of pairs `(feature,value)` for each non-zero value present.

```
def map(line):
      sparseValues = []
      values = line.split(“,”)
      for (value,feature) in fields.zipWithIndex:
            if value != 0 and not excluded(value):
                  sparseValues += (feature,value)
      return sparseValues
```

Means

For each feature, the mean of its values was calculated. These means were used both for replacing missing values and for calculating correlation scores between features and the label feature. This was done by flat mapping each entry to a tuple `(feature(1,value))` for each non-missing value, reducing by key to sum the count of values and the value themselves, and then then mapping the count and sum values to means. For each entry in the data RDD, its values were keyed by the feature indices, then each feature key was reduced to a final result.

```
function findMeans(data):
      data.flatMap({values:[(feature(1,value)) for feature,value in values]})
      .reduceByKey({(count1,sum1),(count2,sum2):(count1+count2:sum1+sum2)})
      .mapValues({(count,sum):sum/count})
```

Correlation

For each feature a correlation coefficient $\rho(x,y) = \text{cov}(x,y) / \sigma(x)\sigma(y)$ between its values and the values of the label feature was calculated. The absolute values of these coefficients were then used to filter out a smaller subset of candidate features and to rank them. For each entry in the data RDD, its values were keyed by the feature indices, then each feature key was reduced to a final result.

```
function correlate(data):
      data.flatMap({values:[
            (feature(1,value*value,value.label*value.label,value*value.label))
            for feature,value in values]
      })
      .reduceByKey({sums1,sums2:sums1+sums2})
      .map({(feature,(count,valSqrSum,lblSqrSum,coprodSum)):
            valMean = means[feature]
            lblMean = means[labelFeature]
            valStd = sqrt(valSqrSum/count-valMean*valMean)
            lblStd = sqrt(lblSqrSum/count-lblMean*lblMean)
            covar = coprodSum/count-valMean*lblMean
            correlation = covar/(valStd*lblStd)
            if correlation == NaN or correlation == infinity:
                  return (feature,0)
            return(feature,abs(correlation))
      })
```

Imbalanced data

The labeled data set was notably imbalanced, and contained roughly three times as many false entries as it did true entries. To compensate for this, the true data points were undersampled to create a balanced data set.

```
def undersample(data):
      falseData = data.filter({values:values.label==0})
      trueData = data.filter({values:values.label>0})
      falseCount = falseData.count()
      ratio = trueData.size / falseData.size
      return trueData.union(falseData.sample(false, ratio))
```

Partitioning

Spark documentation’s recommendation of 2 to 3 partitions per worker core was followed. For training and prediction, 16 machines (1 core, 15 workers) and 60 partitions for RDD’s holding the parsed data were used.

Training

Tuning

To improve the performance of the model, 1% of the data was held out and used for feature selection and tuning. Two models were considered for how to partition the work of tuning the parameters in parallel. The first model was to test multiple configurations in parallel, but with each such

configuration training and evaluating its model sequentially. The second model was to test configurations sequentially, but to build and evaluate the model for each configuration in parallel. We chose the second model because this was the approach most amenable to how Spark partitions data and to MLlib's built in functionality for training models in parallel using partitioned RDD's. Models were built in parallel using MLlib, and evaluation was done by mapping each data point to its label and a prediction using the trained model, and then counting how many entries had matching labels and predictions.

```
function test(data,model):
      predictions = data.map({datum:datum.label,model.predict(datum)})
      numCorrect = predictions.filter({(label,prediction):label==prediction})
            .count()
      return numCorrect/data.count()
```

For the final run training run 70% of the data was used for training a random forest using the parameters tuned on 1% of the data, and the remaining 30% was used to check the accuracy of the resulting model.

Feature Selection

All features below a certain threshold of correlation were discarded from consideration. Features were then iteratively selected greedily based on what improved performance the most until improvements in accuracy fell below 0.0001. The threshold used for filtering candidate features was 0.01.

```
epsilon = 0.0001
function selectFeatures(trainData,testData,scores,threshold):
      candidates = scores.filter({(feature,score):score>threshold})
            .map({(feature,score):feature})
      bestAccr = 0.5
      bestFeatures = []
      while (candidates.size > 0):
            bestCandidate = null
            for candidate in candidates:
                  features = bestFeatures + candidate
                  model = train(trainData,features)
                  accr = test(testData,model)
                  if accr – bestAccr > epsilon:
                        bestAccr = accr
                        bestCandidate = candidate
            if bestCandidate == null:
                  break
            candidates -= bestCandidate
            bestFeatures += bestCandidate
      return bestFeatures
```

A total of 14 features were selected with a test accuracy peaking at between 78% and 79%.

Number Features	**Next Feature**	**Accuracy**
1	*Turdus migratorius* (American robin)	0.6689246401354784
2	*Ardea herodias* (great blue heron)	0.6947502116850127
3	MONTH	0.7154953429297206

4	*Melospiza melodia* (song sparrow)	0.7286198137171889
5	*Molothrus ater* (brown-headed cowbird)	0.7425910245554614
6	CAUS_TEMP_MAX	0.7489415749364945
7	NLCD2006_FS_C82_7500_LPI	0.7544453852667231
8	*Anas platyrhynchos* (mallard)	0.7667231160033869
9	*Sturnus vulgaris* (common starling)	0.77307366638442
10	CAUS_TEMP_MIN06	0.781541066892464
11	DIST_FROM_WET_VEG_FRESH	0.7840812870448772
12	*Troglodytes hiemalis* (winter wren)	0.7849280270956817
13	EFFORT_HRS	785774767146486
14	NLCD2011_FS_C95_75_LPI	0.7883149872988993

Tree parameters

The number of trees and the maximum depth of each tree were explored for the model. For each parameter, a range of values was iteratively tried, training a learner in parallel for each set of parameters and then checking if it improved the accuracy. Noticeable improvements in accuracy stopped between 40 and 50 trees, after which accuracy fluctuated slightly up and down as the number of trees increased. Improvements in accuracy from increasing the maximum depth of trees stopped around a depth of 10. For the final training run, 50 tree with a maximum depth of 10 were used.

```
function tuneParam(trainData,testData,param,minParam,maxParam):
      bestParam = null
      bestAccr = 0.5
      for param in minParam to maxParam:
            model = train(trainData,param)
            accr = test(testData,model)
            if (accr > bestAccr):
                  bestAccr = accr
                  bestParam = param
      return bestParam
```

Final training

Once the desired parameters were found, a random forest was trained on 70% of the data using MLlib. The resulting model was then saved, along with the chosen features and map of replacements. The remaining 30% of the data was then used to evaluate the accuracy of the model in the same manner as during tuning.

```
function train(data,features,replace):
      trainData, testData = data.split(0.7,0.3)
      model = train(trainData,features,replace)
      model.save()
      features.save()
      replace.save()
      accr = test(testData,model)
      return accr
```

Prediction

The model created in by the training program is first loaded into the prediction program along with the selected features and means for replacing missing features. The unlabeled data is then loaded into an RDD, and an RDD of of the sample ID's is labeled before parsing the input data in the same manner as in the train job. The loaded model, features and means are broadcast to the works and the final predictions are created by mapping each unlabeled point through the model. The saved sample ID's are zipped back together with the predictions and written out with a header line.

```
function predict(data):
      model = load(model)
      features = load(features)
      replace=load(replace)
      broadcast(model)
      broadcast(features)
      broadcast(replace)
      sampleIDs = data.map({values:values(SAMPLE_ID_INDEX)})
      predictions = data.map({values:parse(values,features,replace)})
            .map({values:model.predict(values)})
      header = "SAMPLING_EVENT_ID,SAW_AGELAIUS_PHOENICEUS"
      header.union(sampleIDs.zip(predictions)).save()
```

Results

It took roughly 18 minutes to train the model on the labeled data set using 16 machines and 60 partitions. The accuracy of the trained model was 75.14%. It took roughly 6 minutes to produce predictions from the unlabeled data using the same parameters.