INTRODUCTION:

Since the problem in hand is a binary classification problem and there were several categorical features to consider to accurately predict the sighting of the Red-winged Blackbird, the intuition was to choose a decision-tree based classifier to build a model from the given labeled dataset and use it for predicting the bird occurrence in the unlabeled dataset.

Keeping prediction accuracy in mind, we chose to explore ensemble based decision tree algorithms to approach this problem. From the spark documentation, we explored the following algorithms for ensemble of trees:

https://spark.apache.org/docs/2.1.0/mllib-ensembles.html

Type of model used and parameters explored for it.

Random Forest

They are an ensemble of decision trees widely used for solving machine learning problems. It combines multiple decision trees to produce powerful models that can be used for predicting untrained data sets.

Random Forests train each tree independently, using a random sample of the data. This randomness helps to make the model more robust than a single decision tree, and less likely to overfit on the training data.

Parameters Set (And/Or) Tuned for Training:

numTrees = Number of trees to be generated by the algorithm. Increasing this number improves accuracy. As more subtrees are created, there is more reduction in variance. We have currently set it to 1000.
featureSubsetStrategy = The number of features to consider for splits at each tree node. We have to set it to “auto” . This makes sure that depending on the number of trees, the square root if it is selected as the number of features for classification. A very low number is expected to provide poor prediction results. When we decrease this number in the training set, our results are less accurate.
maxDepth: Depth of the trees. This also determines when to terminate in a prediction. Varying it did not improve accuracy much, so we stuck to the default value.

Model used but not implemented for final prediction :

Decision Trees :

A classification tree is very similar to a regression tree, except that it is classification used to predict a qualitative response rather than a quantitative one. For a classification tree, we predict that each observation belongs to the most commonly occurring class of training observations in the region to which it belongs. In interpreting the results of a classification tree, we are often interested not only in the class prediction corresponding to a particular terminal node region, but also in the class proportions among the training observations that fall into that region.

Type of model not used but researched

Gradient Boosted Trees:

GBTs train one tree at a time, where each new tree helps to correct errors made by previously trained trees. With each tree added, the model becomes even more expressive. It is our understanding that GBT train each tree one at a time and hence we decided to discard this type of model. Also, from our understanding GBT have high variance and hence we chose not to try this method.

STEPS INVOLVED IN SOLVING THE PROBLEM.

Data Cleanup:

Identified columns that we were planning to use for feature selection based on the document provided (ebird.pdf). Majority of the features that were selected were core-covariates. All columns that contained values "?" or "X" were replaced by "0" during the preprocessing stage. Only double values were considered during feature selection as majority of core covariates were double value and only a few were string values. Since we were creating LabeledPoint objects from MLlib and these objects only hold double values, we ignored certain string values to avoid the overhead of converting all string values to double values for feature selection.

Data Preparation and Feature selection:

Prepared a new dataset with only the selected columns (40) from the document ebird.pdf.
The new dataset is converted to LabeledPoints which contain a label and a set of features (vector). This data set is split 70% into training dataset and 30% for validation and testing dataset.

Model Training:

The training set data is passed to the Random Forest classifier that trains the dataset with the parameters discussed in the introduction section. The output of the classifier is a model which is preserved to be used later for predicting the unlabeled dataset.

Prediction :

The model created from training is persisted and applied to an untrained data set which will predict the possible outcome of the data given. The untrained data also has to be preprocessed to remove unused columns and selecting only a subset of columns from the original dataset.

Results :

The results from the actual prediction is stored to HDFS in a Comma Separated Values format.

Observation 1 :	Observation 2:	Observation 3:
Number of Trees - 3	Number of Trees - 100	Number of Trees - 100
Depth - 4	Depth - 4	Depth - Default (Strategy)
Accuracy - 76.6%	Accuracy - 78.8%	Accuracy - 79.56%

When we set the parameters set from the default strategy defined by Stategy.DefaultStrategy class, we achieve more accuracy.

Running Time and Speedup :

Program running time in 5 machines AWS : ~ 32 minutes
Program running time in 10 machines AWS : ~ 16 minutes

The program shows good speedup as the running time is decreased by half when the number of machines is increased.

PSEUDOCODE:

Function handleTrainingData(){

```
// Load labeled data from input
Data = load(unlabeled_Data)
// Remove header from the data
Data.remove(header)
```

```
// Perform cleanup of data by inserting “0” wherever there is “?” or “X”
Data.columns.replace(“?”).with(“0”)
Data.columns.replace(“X”).with(“0”)

// Identify columns/features from ebird.pdf required for this problem
// Also consider mostly Double features during the selection. If there are
important string features, map them to a double value for prediction.
DataWithRightColumns = Data.select(column0, column1, column2 ….. )

// Convert all columns in the data created to double values
DataWithRightColumns.Convert([column1,column2,column3...]. ToDouble)

// Convert the data to LabeledPoints so that they can be used by MLlib Apis
// Column 27 contains the information if the red winged bird is spotted or not.
// Column 27 is the label and other columns are the features to be given as input to
// the classifier.
Convert.ToLabelPoints( Column(27) , Other Columns  )

// Split data into 70 percent training data and 30 percent test data
 Array(trainingData, testData) = LabledPoints.randomSplit(Array(0.7, 0.3))
```

// Set parameters for the Random Forest classifier

```
  val seed = 5043
  val numTrees = 3
  val featureSubsetStrategy = "auto"
  val numClasses = 2
  val categoricalFeaturesInfo = Map[Int, Int]()
val impurity = "gini"
val maxDepth = 4
val maxBins = 32

// Train the classifier using RandomForest Classification Method.
val model = RandomForest.trainClassifier(trainingData,    numClasses,
categoricalFeaturesInfo,numTrees, featureSubsetStrategy, impurity, maxDepth,
maxBins)
// Predict output based on model created on the test data
model.predict(testData)
```

```
        // Print to log for reference
        calculate(trainingError)
        calculate(accuracy)
}
```

Function handleTestingData(Model)

```
    {
        UnTrainedData = GetDataFromInput()
        // Perform the same preprocessing as done for labeledData.
        // Get the data in the form of Labeledpoints so that MLlib apis can use the data.
        // Get the model created from labeledData.

        // Predict untrained data
        Prediction = Model.predict(UntrainedData);
        foreach line in UntrainedData
            emit(line.EventId, line.prediction
    }
```

Partitions and Parallel operations :

Spark by default has a parallelism setting that defines the default number of partitions for RDD. For distributed shuffle operations depending on the parent RDD, more number of partitions are created. Some operations like parallelize that do not have a parent RDD, spark uses its default parallelism.

Clusters will not be fully utilized unless you set the level of parallelism for each operation high enough. Spark automatically sets the number of "map" tasks to run on each file according to its size (though you can control it through optional parameters to SparkContext.textFile, etc).

Spark's shuffle operations (sortByKey, groupByKey, reduceByKey, join, etc) build a hash table within each task to perform grouping, which can often be large. By increasing the level of parallelism, so that each task's input set is smaller, we can overcome memory bottlenecks.

Since our dataset uses a random forest training classifier, data is trained in parallel depending on the number of trees generated and number of split attributes passed to the classifier. In essence partition has to happen and shuffling and sorting should take place to accumulate the final results in a final output file. Spark gives us options to define parallelism in program as well. This is needed if we consider the following scenario.

Every block in HDFS generates a partition of RDD. Files that are very big and contain multiple blocks, tend to have multiple partitions. Since the number of partitions will be high for this specific file, the parallelism for increases for the program. One of the ways to control parallelism in a spark program is through the use of sc.textfile(path, num_of_partition).