CS6240 Project

eBird Sighting Prediction



Abhinav Maurya (maurya.a@husky.neu.edu)

Vikas Janardhanan (janardhanan.v@husky.neu.edu)

Pre-Processing:

Training data had 1701975 records with 1657 features. First step we did was to analyze the dataset and figure out which all features would be relevant to make the prediction of whether a Red-winged BlackBird would be sighted in a birding session or not.

Checklist features:

1. Since YEAR,COUNTRY is same for all the data, those columns are ignored
2. Among fields that represent the location like LATITUDE, LONGITUDE, STATE\_PROVINCE, COUNTRY, we considered LATITUDE, LONGITUDE to represent the location and ignored others since others had more missing values.
3. We made use of MONTH since based on migration pattern of the bird, the probability of sighting of the bird on certain month might be higher.
4. DAY,TIME associated with the sighting was also chosen since bird might have higher chances of appearing at certain times of day.
5. We remove all the other bird observation labels except Agelaius\_phoeniceus" since we felt that bird observations are not correlated.

Core- covariate features:

Since based on domain experts, core-covariate experts, core-covariate names are most important to bird sighting prediction, we have used all the core-covariate features except a few listed below:

1. Since around 1.6 million records out of 1.7 million, for NLCD2001\_FS\_C12\_7500\_PLAND

and NLCD2006\_FS\_C12\_7500\_PLAND was zero, we felt it’s not useful and ignored it.

1. ELEV\_NED had around 1.2 million records which had missing value, so we ignored the same.

Extended covariate features:

We haven’t used any extended covariate fields like CAUS\_TEMP\_AVG01, CAUS\_TEMP\_MIN01, CAUS\_TEMP\_MAX01 etc for each of the months and instead used only the average temperature fields across whole year.

**Pseudo Code:**

**Preprocessing** / **Data cleaning task:**

This task keeps only the features that are required for building the model and ignores the rest.

1. Use sqlContext read to read csv as dataframe replacing all ‘?’ with null
2. Select only those columns into the dataframe that is defined inside DataSetColumns class.
3. For Agelaius\_phoeniceus attribute, replace X with 1 and all values greater or equal to 1 with 1.
4. Change the datatype for all the columns to double and rename “Agelaius\_phoeniceus “as label so that it’s in format suitable for input to RandomForest algorithm.

**Training Task**:

This task reads the cleaned data and then builds the model required for prediction.

1. Use the attributes in the dataframe to create a vector of selected features and retain it as a column in dataframe using VectorAssembler.
2. Build the RandomForest model using label and vectorized features column as input.
3. Save the created model into HDFS.

Train Random Forest Model

M

Persist Model for prediction

M

Load Input as dataframe

Create Feature Vectors

Random Forest Spark Implementation:

* Spark performs partition by row. Each worker receives a subset of data to operate on.
* To understand the spark’s implementation for random forest, let’s first look at implementation of decision tree in spark mllib.
* The main idea behind creation of decision tree is finding the optimal split on each feature. To get the optimal feature spark performs following things:

|  |  |
| --- | --- |
| 1. Each worker computes partial histogram on each feature based on subset of data it has received. 2. This information is sent to master, which aggregates all histograms it received and chooses the best split using the predefined histogram bins parameter. 3. This result in an approximate split rather than optimal split since each worker just produces approximate histogram based on the subset of data. | ../Desktop/Screen%20Shot%202017-04-23%20at%206.24.35%20PM.png |

* As we are using Random Forest, we specify number of trees to be built in the model. MLlib implementation of Random Forest trains each tree independently hence multiple trees can be trained in parallel (in addition to the parallelization for single trees). The trees are built locally on each worker once they have obtained the split information.
* Random Forests use a different subsample of the data to train each tree. Instead of replicating data explicitly, it saves memory by using a TreePoint structure which stores the number of replicas of each instance in each subsample.

Parameter Tuning:

1. maxDepth: This sets the maximum depth of the trees in random forest. We observed that there is depth of the tree selection has huge impact on the running time.

If we increase the depth of the tree beyond 20, we get Heap error on aws EMR.

1. numOfTrees: This parameter set the number of trees to be used in random forest ensemble learning model. As per the documentation of spark, for classification problem minimum number of trees to be used are “sqrt(number of features)” to cover all the features once in a tree.

So, the min number of trees for 63 features are 8 and We decided to keep number of trees above that (35) so the features can be covered more than once by different decision trees.

Below are the performances comparison with the above parameter values.

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| |  |  | | --- | --- | | Model Training  Time (in min) | Depth | | 19 | 10 | | 21 | 12 | | 35 | 15 | | 49 | 18 | | 63 | 20 |   Depth = 12  No. of Machines = 10 |  |

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| |  |  | | --- | --- | | Model Training  Time (in Min) | Num Of Trees | | 18 | 20 | | 23 | 25 | | 29 | 40 | | 33 | 50 | | 36 | 75 | | 40 | 100 | | 42 | 125 |   Depth = 12  No of Machines = 5 |  |

**Prediction task:**

* Prediction task loads the model created in training task and distributes to each worker node.
* It partitions the unlabeled data and distributes a subset to each worker node.
* Worker nodes read each record and using the model loaded before predicts the label for the records.
* After the prediction, each predicted label collected to single node (using coalesce(1) function) and output file is generated in csv format with SAMPLING\_EVENT\_ID of corresponding label.

Psuedo Code:

1. Loads the test data into dataframe with the required predictor attributes and does similar preprocessing as mentioned above.
2. Adds a sequence number to the dataframe.
3. Loads the already created model from HDFS.
4. Predict the outcome of each test instance using the model.
5. Output the prediction ordered by the sequence number.

**Results:**

Train Accuracy for different parameter settings:

|  |  |  |  |
| --- | --- | --- | --- |
| Trees= 125 Depth = 10 Accuracy: 77.9% | Trees= 35 Depth = 10 Accuracy: 77.6% | Trees= 35 Depth = 15 Accuracy: 81.9% | Trees 35 Depth = 18 Accuracy: 84.6% |

Running Time for Random Forest Training:

Trees = 35 and Depth = 15

|  |  |  |
| --- | --- | --- |
| Worker Nodes = 1 Running Time: 1 hr 32 mins | Worker Nodes = 5 Running Time: 45 mins | Worker Nodes = 10 Running Time: 37 mins |

Speedup with 5 machines: 2.04  
Speedup with 10 machines: 2.48

We observed that with number of nodes = 15 there was not much difference. From this we can infer that at this point communication cost was higher which results in the poor performance in increased number of worker nodes.

Running Time for Prediction:

Trees = 35 and Depth = 15

|  |  |  |
| --- | --- | --- |
| Worker Nodes = 1 Running Time: 23 mins | Worker Nodes = 5 Running Time: 16 mins | Worker Nodes = 10 Running Time: 13 mins |

Speedup with 5 machines: 1.43  
Speedup with 10 machines: 1.77