Group 31 - Final Report

CS 6240
Large Scale Parallel Data Processing
Fall 2019

Team Members

Derek Grubis
Harshit Gupta
Brian Rouse

Github repository: https://github.ccs.neu.edu/cs6240-f19/team31_grubis_gupta_rouse-Project

Project Overview

The goal of this project was to train an ensemble of classification models that can determine whether an airplane flight will be delayed by weather conditions. The project required two phases of work:

1. Generate a suitable testing and training dataset by combining four datasets that describe flight, airport, weather observation, and weather station data.
2. Train and test a classification ensemble using this new dataset.

Phase 1 was implemented using the 1-Bucket-Theta join algorithm, along with several helper jobs for preprocessing and postprocessing of data. The goal was to create a dataset that contains both flight and weather data, and to experiment with different partitioning and join strategies to find the most efficient approach. Phase 2 was implemented using the Apache Spark MLlib machine learning library. We performed training and testing for both a single model and an ensemble, with the goal of achieving highest possible accuracy within a reasonable running time.

Input Data

Input Datasets are divided into flight and weather datasets. The flight dataset involves the flight details like flight id, airports, flight time, delay time, cancellation reason etc. The weather dataset involves the recorded weather (temperature, pressure, wind, snow, rain etc.) at different weather stations across the US. Both datasets cover the year 2015, and can be scaled to include additional years using data from the National Oceanic and Atmospheric Administration (NOAA) and US Department of Transportation (DOT) websites.

Flight Data:

1. Airports.csv - Contains the airport codes, airport names, latitude and longitude details.

IATA_CODE	AIRPORT	CITY	STATE	COUNTRY	LATITUDE	LONGITUDE
ABE	Lehigh Valley International Airport	Allentown	PA	USA	40.65236	-75.4404

2. Flights.csv - Contains flight number, origin airport, destination airport, date of flight, departure time, arrival time, delay flag, cancellation flag, cancellation reason. (input sample too long to copy)

Weather Data:

1. Us_weather_stations.csv - Contains station id, station name, latitude and longitude

	USAF	WBAN	STATION NAME	CTRY	STATE	ICAO	LAT	LON	ELEV(M)
14005	691774	99999	LADD AAF	US	AK		64.833	-147.6	137.5

2. 1_2015.txt to 12_2015.txt - Contains the weather data for the corresponding stations for each day of the year 2015. Station id, date, temperature, dew point pressure, max wind speed, max temp, min temp, precipitation, snow depth.

```
STN---,WBAN , YEARMODA,   TEMP,  ,   DEWP,  ,  SLP  ,  ,  STP  ,  , VISIB,  ,  WDSP,  ,
MXSPD,  GUST,   MAX  ,  MIN  ,PRCP  ,SNDP , FRSHTT,
423630,99999, 20150108,   52.4, 7,   38.6, 6, 1031.6, 6, 9999.9, 0, 999.9, 0,  29.7, 7,
35.0, 999.9,   59.0*,  46.9*, 0.00I,999.9, 000000,
```

Tasks 1-2: Theta-Joins 1-Bucket (MapReduce)

Overview

A 1-Bucket-Theta join was used to create a dataset that contains both flight and weather data. To classify flight delays based on weather, each flight record must contain climate data pertaining to both the flight's origin airport and its destination airport. Our approach required four steps:

1. A preprocessing job to equi-join weather observations with weather stations, adding a latitude/longitude location to each observation. This job includes a random sampler to optionally reduce the size of the weather dataset.
2. Another preprocessing job to equi-join flights with airports, adding a latitude/longitude location to each flight's origin and destination airport. This job also includes a random sampler to optionally reduce the size of the flight dataset.
3. A 1-Bucket-Theta join to combine each flight's destination and origin airports with weather observations that occur on the flight date, within a specified distance radius of the airport. The radius size parameter controls the extent to which the join is output-dominated.
4. A postprocessing job to consolidate the data by flight, by picking the closest weather observation to each origin and destination for each flight. Another possible approach would have been to average all weather observations within the radius for each origin/destination.

Pseudo-code

S = |weather data|
T = |flight data|
r = number of workers

computeAB(S, T, r)
    C = S/T
    if C < 1/r
        A = 1 and B = r
    else
        A = ⌊(C * r)^(½)⌋
        B = ⌊(1/C * r)^(½)⌋

map(tuple x)
    if (x is a weather observation record) {

```
            // Select a random integer from range [0,..., A-1]
            row = random(0, A-1)
        // Emit tuple for all regions in the selected row
        for regionId = (row * B) to (row * B + B - 1)
            emit ((regionId, “weather”), x)
        }
        else { // x is a flight record
            col = random(0, B-1)
            for regionId = col to ((A-1) * B + col) step B
                emit ((regionId, “flight”), x)
        }

getPartition((regionId, flag))
        return regionId

keyComparator((regionId, flag))
        // Sorts on regionId first
        // If regionsIds are equal, weather comes before flights

groupingComparator((regionId, flag))
        // Considers regionId only

reduce(regionID, [x1, x2,...])
        [Initialize weatherList]
        For all x in input list:
            if x.isWeather
                weatherList.add(x)
            else
                // Compare the flight’s origin and destination to each weather observation
                joinResult = joinByDateAndDistance(flight, weatherList)
                For each tuple t in joinResult
                    emit (NULL, t)
```

Algorithm and Program Analysis

S = size of pre-processed weather file - 1,068,437 records, 103.2 MB
T = size of pre-processed flights file - 5,323,699 records, 441.2 MB

We focused on two analysis goals: improving performance and determining which metrics affect 1-Bucket-Theta’s effectiveness for this problem, as compared to other algorithms like a Replicated Theta-Join or M-Bucket.

1. Improving Performance

Random sampling:

AWS runtime was too slow while running the full 2015 datasets on 10 m4.xlarge machines. Our solution was to randomly sample flight and weather data to decrease input size. Random sampling can also achieve better performance by adjusting the S/T ratio via random samples. In seeking to achieve lower bounds for max reducer input and output, 1-Bucket does not always

leverage every worker machine made available to it. For example, with the full 2015 dataset and 10 machines, 1-Bucket Computes A = 1 and B = 7. Three machines go unused in the reduce phase. But if you sample the flight data by p ≈ 0.5, you get A = 2 and B = 5, utilizing all 10 workers while still benefiting from 1-Bucket's formula for approximating the lower bounds. We observed how getting a good S/T/r ratio can affect results during our speedup tests.

Memory usage:
Memory usage was an obstacle to achieving good performance, as a memory shortage was causing the cluster to time out in the reducer phase (for 10 m4.xlarge machines). This timeout problem was resolved by implementing secondary sort in the 1-Bucket algorithm to alleviate memory costs. After adding secondary sort, we also experimented with increasing the JVM heap size from 2458mb to 7374mb and physical memory from 3072mb to 9126mb. This may have addressed the timeout issue as well, but did not produce any additional discernable gains in performance.

2. Analyze 1-Bucket Effectiveness
Radius size:
When we increase the distance radius in the theta function, we find more weather observations per airport, potentially increasing the accuracy of the per-airport weather data (when using the averaging approach in post-processing. See "Overview" step 4). This also increases the output relative to the join input, justifying 1-Bucket-Theta, which is known to perform well for output-dominated joins. Thus, we would expect our join algorithm to be an optimal choice when Radius is high, and to be less efficient than a targeted matrix cover join, like M-Bucket, when Radius is low.

Increasing radius increases the ratio of hits to possible hits, e.g., bringing the join closer to a cross product, for which 1-Bucket is proven to be effective. At lower radius values, a different matrix covering algorithm might identify the cells that do not need to be covered:

p(Weather)	p(Flight)	Radius	r	Hits	1-Bucket Runtime	Input: Output Ratio
0.5	0.125	0.0	10	0	1 hour, 34 minutes	1 : 0
0.5	0.125	3.0	10	710,184	1 hour, 8 minutes	1 : 0.59
0.5	0.125	100.0	10	26,743,520	1 hour, 5 minutes	1 : 22.29

Number of workers:
There are circumstances where 1-Bucket-Theta will not necessarily be an improvement over a Replicated Theta-Join. For example, using a small number of worker machines for the full 2015 datasets results in A = 1. As long as the weather observations data can fit in memory, a Replicated Theta-Join could potentially improve runtime by broadcasting the observations and avoiding a shuffle. However, it would not include the output-skew neutralization achieved by randomization. 1-Bucket-Theta becomes more appropriate as we add more years to the dataset (observations will become too large to broadcast) or as we add more workers (A >= 2 is harder

for Replicated Theta-Join to emulate). For example, A = 2 occurs for r = 20 for the full 2015 datasets.

Experiments

All experiments were performed on the theta-join between flight records and weather observation records. We did not experiment with the pre- and post-processing jobs, as these were comparatively trivial equi-join and aggregation by key programs. The minimum Radius threshold is somewhere between 2.0 and 3.0 miles. If Radius is set below the threshold, some flight origins and/or destinations will not find a sufficiently close weather observation to join with. This was determined through testing data samples on a local machine.

Speedup:

Workers on AWS	Worker Type	Runtime	Radius	Sampling Rate	A	B
5	m4.xlarge	1 hour, 38 minutes	3.0	p(flights)=0.125 p(weather)=0.5	2	2
10	m4.xlarge	1 hour, 8 minutes	3.0	p(flights)=0.125 p(weather)=0.5	2	3

Speedup is achieved, but not as substantial as expected (only 30% faster runtime while doubling workers - 50% reduction would be optimal). We attribute this to inefficiencies inherent in 1-Bucket while working with certain S/T/r ratios at a smaller scale. The 5-worker job leveraged 4 out of 5 available machines, while the 10-worker job only leveraged 6 out of 10. See “Random sampling.”

Scalability:

Workers on AWS	Worker Type	Runtime	Radius	Sampling Rate	A	B
10	m4.xlarge	1 hour, 8 minutes	3.0	p(flights)=0.125 p(weather)=0.5	2	3
10	m4.xlarge	2 hour, 3 minutes	3.0	p(flights)=0.25 p(weather)=0.5	2	4

Scalability was achieved to a degree. Data increased by a factor of 1.55 had a runtime increase of 1.8.

Result Sample:

Black = flight data, Red = origin weather data, Blue = destination weather data

DATE,ORIGIN,ORIGIN_LAT,ORIGIN_LONG,DEST,DEST_LAT,DEST_LON,AIRLINE,FLIGHT_NUMBER,TAIL_NUMBER,DISTANCE,WEATHER_CANCELLATION,WEATHER_DELAY,
DATE,LATITUDE,LONGITUDE,TEMP,DEWP,SLP,STP,VISIB,WDSP,MXSPD,GUST,MAX,MIN,PRCP,SNDP,FRSHTT,
,DATE,LATITUDE,LONGITUDE,TEMP,DEWP,SLP,STP,VISIB,WDSP,MXSPD,GUST,MAX,MIN,PRCP,SNDP,FRSHTT,
2015-01-01,ABI,32.41132,-99.6819,DFW,32.89595,-97.0372,MQ,2908,N697MQ,158,1,0,0,

2015-01-01,32.411,-99.682,23.3,21.1,1027.8,961.6,5.2,4.2,18.1,999.9,28.9,15.1,0.07,0.0,0,1,1,0,0,0,
,2015-01-01,32.898,-97.019,33.3,27.3,1028.4,1005.6,6.6,5.8,11.1,999.9,36.0,30.9,0.00,0.0,0,1,1,0,0,0,

Logs:
https://github.ccs.neu.edu/cs6240-f19/team31_grubis_gupta_rouse-Project/tree/master/thetaJoins/logs

Output files:
https://github.ccs.neu.edu/cs6240-f19/team31_grubis_gupta_rouse-Project/tree/master/thetaJoins/output
*Output for Radius = 100 not included due to size. Total hits value was aggregated in the logs.

Task 3: Classification and Prediction in Spark MLlib

Overview

With the joined data of flights with weather, we used that joined data as our dataset for training both a single model and an ensemble model. Specifically, we built a classifier that predicts whether a certain flight will get delayed or not based on the weather. Our motivation could fuel an app where a person can type in their flight info hours before their flight and our model(s) can output a probability that their flight will be delayed. Valuable information to know about your flight beforehand.

Due to the vast amount of data we'll have at our disposal, especially with an output-dominated Theta-Join, we trained a bagged Random Forest model instead of a boosted model to take advantage of the nice quality bagging has over boosting of being able to train models independently in parallel. We also trained a single decision tree to compare with the Random Forest.

With our trained models we explored getting the most accurate model possible while dealing with memory problems and run time constraints. We also compared random forest models fixing the number of trees in the assemble to look at how well more trees parallelizes. This was an interesting investigation for our problem since our motivation lies around users being able to access predictions about their flights quickly and efficiently.

Pseudo-code

The Spark code for our Decision Tree Model (same for Random Forest):

```
val FeatureIndexer = new VectorAssembler()
                     .setInputCols(myFeatures)
                     .setOutputCol("features")
val LabelIndexer = new StringIndexer()
                   .setInputCol("WEATHER_DELAY")
                   .setOutputCol("label")
val Array(trainingData, testingData) = df.randomSplit(Array(0.75, 0.25))
```

```
val dtClassifier = new DecisionTreeClassifier()
                .setFeaturesCol("features")
                .setLabelCol("label")
                .setImpurity("gini")
                .setMaxDepth(5)
val pipeline = new Pipeline().setStages(Array(FeatureIndexer, LabelIndexer, dtClassifier))
val dtModel = pipeline.fit(trainingData)
val predictionData = dtModel.transform(testingData)
val metricEvaluator = new BinaryClassificationEvaluator()
                .setLabelCol("label")
                .setMetricName("areaUnderROC")
val auc = metricEvaluator.evaluate(predictionData)
println("AUC of Decision Tree Model is " + auc)
val gridSearch = new ParamGridBuilder()
                 .addGrid(dtClassifier.maxDepth, Array(5, 8, 12))
                 .addGrid(dtClassifier.maxBins, Array(20, 25, 32))
                 .addGrid(dtClassifier.impurity, Array("entropy", "gini"))
                 .build()
val cv = new CrossValidator()
           .setEstimator(pipeline)
           .setEvaluator(metricEvaluator)
           .setEstimatorParamMaps(gridSearch)
           .setNumFolds(10)
           .setParallelism(3)
val CVmodel = cv.fit(trainingData)
val cvPredictionData = CVmodel.transform(testingData)
val cvAUC = metricEvaluator.evaluate(cvPredictionData)
println("AUC of Optimized Decision Tree Model is " + cvAUC)
```

Algorithm and Program Analysis

For getting the most accurate model possible we focused on the best way to do this efficiently since tuning hyper-parameters can be a very expensive task. For both the decision tree and random forest models our program can be “split in half”. We first trained a model on the training data specifying the hyper-parameters for the model manually and then set up a grid search, cross-validation method to search through a space of parameters and testing the performance of each one at each fold in the data.

Tuning Manually

To achieve the actual model building we used the pipeline feature Spark has available. This allows a model to be trained given the label, features and classifier to be used. This will also be needed for cross-validation so creating the model as a pipeline saves resources as multiple copies of the training/testing data don’t need to be created again. We collected the features we wanted to feed into our model through a VectorAssembler(). This creates a new column that collects all the features into a vector. We used the WEATHER_DELAY flag as the label for our model where 1 means the flight had a delay, 0 otherwise. While this column doesn’t need to be encoded we pass it through the StringIndexer() for cross-validation later. Next, we split the data

75,25 into training and testing sets and set up the pipeline. We fit the pipeline and used AUC as the metric to evaluate our model. According to the Spark MLlib docs, this is one of two supported metrics but AUC better captures a model's performance than strictly accuracy. AUC is the area under the ROC curve. The ROC curve is the true positive rate against the false positive rate. An AUC closer to 1.0 is preferable. The results of different runs for both models is in the Experiments section. Not much training can happen in parallel for the Decision Tree but the Random Forest can achieve parallelism through training the ensemble of trees separately then bringing them all together for the final aggregated model.

Tuning Using Cross-Validation

While the above part can extract a decent model, finding the optimal model through tuning different parameters manually is not realistic. Training a single model is inexpensive but having a more statistically-motivated method for deriving the optimal method is preferred. One such popular method is using cross-validation to search through a grid space of different parameters to find the subset that returns the best performing model. This is where the pipeline we built in the first half comes in handy. Spark now allows the implementation to train different pipelines together in parallel. Essentially one pipeline can consist of one subset of the parameter space. Each can be trained separately then in a reduce phase the global subset is chosen as the optimal winner.

Spark allows the user to set this parallelism parameter; the max number of parameters that can be evaluated in parallel. There is a certain heuristic to choosing this value. Setting it too low can under-utilize resources in a cluster but setting it too high can lead to memory issues in overloading a cluster. This feature that also reduces the amount of data transferred at each fold. Under the hood, the training can be thought of as a tree. The first parameter in the first row is cached as the child rows are then trained and evaluated. Then that first dataframe is unpersisted and the next dataframe is cached. This process is repeated until the whole tree is traversed and the optimal path is then returned.

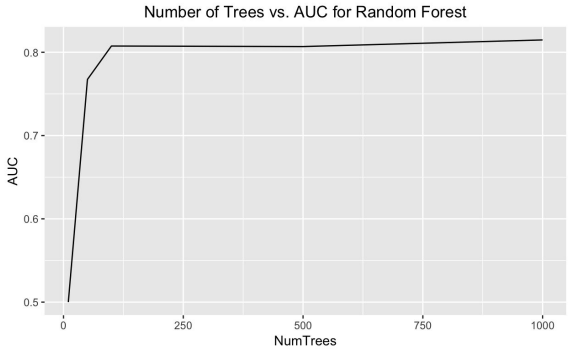
Experiments

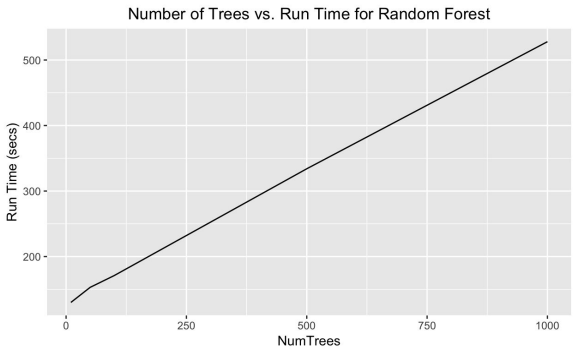
We ran two sets of experiments. First was comparing the Decision Tree with the Random Forest for different sizes of clusters. This is using the cross-validation method for both models to find how long in parallel it takes each to find the best model. We can then compare how the running time affects the accuracy and look at the trade-off between the two. The next set of experiments is looking at just our ensemble method (random forest) and for each run on AWS fixing the number of trees in the ensemble. This uses just the "first-half" of the program to see how increasing the number of trees affects both AUC and the running time.

We first ran into memory issues running the random forest on five clusters. We set the parallelism parameter to 5 to start and had a large space of parameters to search through. This overloaded the cluster after running for over 2 hours so we dropped our search space down to a 2x2x2 space and lowered the max threshold of parameters in parallel to 3. This allowed the program to run without running out of memory and returned an optimized model (improvement over the manually tuned one). Setting this parallelized parameter correctly is a way to achieve good parallelism for this very expensive task.

Speedup

Algorithm	Workers on AWS	Worker Type	Runtime	AUC
Decision Tree	5	m4.large	30 minutes	0.5
	10	m4.large	16 minutes	0.5
Random Forest	5	m4.large	2 hours, 4 minutes	0.8395
	10	m4.large	1 hour, 4 minutes	0.8413





Both Decision Tree and Random Forest have great speedup as they essentially cut half the time from doubling machines. Our decision tree model performs poorly but the random forest has a good AUC score. We then took a look at just the random forest by increasing the number of trees in the ensemble leaving all other parameters constant. The AUC score jumps up after only increasing the trees in the 50-100 range where it then levels off as you increase the number of trees. The running time grows linearly at a very constant rate. Therefore, the optimal number of trees would be at that elbow on the plot on the left (around 100 trees).

Scalability
For this task, no scalability was feasible to look at since we want as much training data as possible.

Result Sample
AUC of Random Forest Model is 0.8000077587699423
AUC of Optimized Random Forest Model is 0.8395479290394945

Logs:
https://github.ccs.neu.edu/cs6240-f19/team31_grubis_gupta_rouse-Project/tree/master/sparkClassification/SparkClassification/logs
Output files:
https://github.ccs.neu.edu/cs6240-f19/team31_grubis_gupta_rouse-Project/tree/master/sparkClassification/SparkClassification/output

Conclusion

For the theta-join component, we were able to accurately join weather and flight datasets by distance and date while demonstrating speedup and scalability. We also determined how certain parameters (S, T, r, and radius) can be used to tune the performance of our 1-Bucket theta-join. One improvement to our join strategy would be to change the post-processing job to average weather results per origin/destination, rather than pick the closest weather observation. As we increase the radius size, this would potentially increase the range and accuracy of the weather observations for the final dataset. This change is also naturally supported by 1-Bucket, which performs well for output-dominated joins.

For the classification component we were able to train both a random forest and decision tree model using the output of the theta join. We achieved good parallelism and scaleup by tuning how the parallelism works for the cross-validation component of our model tuning. The random forest model gave us good results and speedup setting up future work of building more on the model to add a regression aspect to predict how long a delay will be for. We can also turn this into a multi-class classification problem to predict if a flight will be delayed cancelled or leave on-time. Effectively handling big data parallel processing allows a host of interesting applications to take form as we have demonstrated.