DSC 102 - Systems for Scalable Analytics

Final Project Discussion

Author: Colin Wang, Jerry Chan

Data Parallelism

Baseline Solution

Given the requirements of the assignment, we believe the figure we draw would maximize parallelism. Basically, feature engineering and label preparation can run at the same time, with the feature engineering part being run on an EC2 instance with Dask and the label preparation part being run on an EMR cluster with Spark. The modeling part has dependency on both feature engineering and label preparation. Therefore, we want to maximize efficiency preferably by completing both feature engineering and label preparation together. After that, we can continue using EMR to join the engineered features and labels, and use pyspark's machine learning library to split the data, select the model, train, validate, make predictions, and save model artifacts into the S3. This is also represented by the drawing we made.

A Better Solution

However, in case the question is asking for an ideal situation, then a better practice to maximize dataflow and parallelism is to use the EMR entirely. To be specific, instead of using an EC2 instance to do feature engineering, we can install the Dask environment on an EMR cluster with a bootstrapping shell. Then, we can submit the feature engineering step, the label preparation step, and the modeling step together to the EMR cluster. We can set the feature engineering step and label preparation step to run in concurrency, and instruct the cluster to dynamically distribute computing resources to make sure they complete roughly at the same time. Then the modeling part can directly happen after both steps are finished. This alternative procedure will maximize parallelism while making the whole thing more streamlined. This is not represented by the drawing we made, since it's not following the required steps in the assignments; nevertheless, we believe that this should be a better approach and definitely the case if one asks for a more ideal solution.

Configurations and Tradeoffs

EC2 configuration:

- Instance type - t2.xlarge: Our data isn't too large, so we choose T2 instances as they are cheaper, general-purpose instances that are sufficient for our task. We run our prototype on t2.micro and scale to t2.xlarge for its 4 vCPUs and 16 GB of memory. The t2.xlarge instance enables us to run the processes on more threads and work with a larger partition of data. With t2.xlarge, we are able to process data for a whole year under 10 minutes, bootstrapping included.
  Tradeoff: higher price -> higher performance.
- Operation System - Ubuntu 20.04: We choose a Linux operating system because it's cheaper than Windows (0.1856 vs 0.2266 per hour). Linux is also more compatible with most software we use.
  Tradeoff: None, Linux is just better in this case.

EMR configuration

Our EMR is mainly used to prepare labels, train models, and output results. Based on our 2009 annual data, it usually takes 15 minutes to initialize the cluster, and 5 minutes to finish the whole job with one master node and one slave node. Therefore, in terms of the amount of computational resources and cost efficiency, if your data is only under or about 6 years (i.e. from 2009 to 2014), then one master node and one slave node will cost as much as you do everything with only 1 year's data. This is due to the fact that EMR takes normalized hour as its pricing unit (both cases will make 1 * 8 [due to m5.xlarge] * num of clusters). However, if the data becomes larger, say 30 years of data, or a more complicated model is deployed, say some kind of neural network, or more feature engineering is done, say things involving lots of aggregation and calculation, then you can scale up the cluster to finish the jobs sooner. Here are three different strategies for three different goals.

1. If you care about cost efficiency, then make a cluster that will finish its job and terminate itself closer to an entire hour (i.e. 58 minutes, 2 hours and 57 minutes, etc). Also, using spot instances can lower the cost as well. The risks of this strategy are the loss of termination protection (due to spot instances) and bad luck (i.e. the cluster spends 1 hours 1 minute rather than 58 minutes).

2. If you care about time efficiency, then make a cluster that can auto-scale depending on workload.

Further, if you want to try if your code can work on EMR clusters, it's better to initialize a cluster with JupyterHub and JupyterEnterpriseGateway, so you can open up a JupyterLab, start a pyspark kernel, and perform some quick experiments on it. On the other hand, if you are pretty confident with your script, then avoid adding these two packages during initialization, as they slow down that initialization procedure.

Despite auto-scaling, you can also set up auto termination after all steps have completed. Some people may forget to terminate the cluster after jobs are completed, and they get charged a lot because the pricing of EMR is about rent time, instead of the actual runtime (i.e. time spent on running state).

Finally, in terms of our configuration, we used the cost efficiency approach. Therefore, we initialized our cluster with only necessary packages (pyspark, hadoop, etc), one master node and one slave node, and auto-termination upon job completion. We directly submitted our steps to the EMR cluster during initialization, with all python script files stored in the S3 instance where the EMR is attached to.

Handle a larger volume of data

Although the Dask pipeline can scale up very easily by increasing the number of processors and RAM, it’s very pricey to switch to a larger instance type. We can double the number of vCPUs and memory by upgrading to t2.2xlarge, but it’ll also double the cost. Also, there’s a 192 GB memory limit for EC2 instances. If the data gets too large, it’s more economical to switch to run the feature engineering on EMR. We can also modify the pipeline to calculate the aggregated values (mean, std, and categories) from sampled data, and process the data in chunks. This method saves resources by keeping most of the data in S3 and only read a partition of the data to EC2’s memory. However, this method might not be practical in real-time prediction as its speed is still limited by the number of CPU (how fast we can process the data) and memory (how much data we can process at once). Therefore, the best solution is to move the process to EMR.

Modeling

We have tried gradient boosting methods, random forest, and logistic regression to build our models. Since our labels are extremely unbalanced (with a 19:1 default ratio), traditional methods of training make the model predict non-default for all data points to maximize accuracy. Therefore, by looking at pyspark's documentation, we ended up using the logistic regression because it supports different class weights. We added a class weight column to make the training more balanced towards both labels. Even though we lost some accuracy (mostly by having some false positives), the model can successfully predict two thirds of the loans with a default label. The training and test metrics are roughly the same, therefore we didn't add regularization parameters into it. A potential improvement is to add more data into training. Currently our model is built upon all 2009's data with a 80:20 split ratio for training and testing set and a seed of 42.