Data Analytics Pipeline for Historical Taxi Data

Group Members: Parth Nagori, Anirudha Tambolkar
Course name : CSC 591 Data Intensive Computing
December 13th, 2018

Abstract
The industry today relies heavily on data analytics to make predictions. These predictions lead to successful business models that incentivise heavily from machine learning. Popular taxi services such as Uber and Lyft provide their users with a prediction of taxi fare before the customer is mapped to a driver. We try to provide a similar solution using the open dataset provided by the NYC Taxi and Limousine Commision (NYC-TLC). The intention is to process voluminous data in parallel that gets generated in NYC-TLC's public data repository, perform feature engineering on it and train and deploy a prediction model using it. The key idea is to understand and implement a data analytics pipeline that forms the basis of data processing in today's software engineering.

Keywords
Data Analytics Pipeline, Batch processing, Machine Learning, Amazon Web Services, Data analytics, Scalable Architectures

Motivation
The NYC TLC generates huge amounts of local trip data everyday which can be mined to learn important traits about the trips being taken in the city. For eg. Taxi Fare Prediction, Surge prediction, determine hot spots, i.e. areas high in demand based on number of requests from a certain geographical location.

The main motivation in solving this problem is to be able to design a system which can deal with huge amounts of data generated everyday and still be able to maintain the efficiency in terms of processing it. Thus we focussed on the volume aspect of big data. Machine Learning models in general tend to discard the future repercussions on efficiency when data starts to grow in size. We want to tackle that problem from day one.

Introduction

Our system will process the inflow of data in order of Gigabytes from various taxi trips in the most efficient manner possible. Data processing can be further specialized here as reading the data in parallel using a data pipeline, performing various data preprocessing tasks like data cleaning and featuring engineering in parallel, storing the preprocessed data in-memory for faster access and then training a machine learning model on top of it to perform fare prediction. The architecture diagram in Figure 2 provides a high level view of these steps and the tools used.
The solution that we have built uses various Amazon Web Services. We pull in records of data from S3 into EMR cluster where the data would be preprocessed. We would also create jobs for EMR to perform feature engineering in parallel. This task would be followed by storing the processed dataset on S3. Once this step is completed, we can play around with multiple machine learning models using Sagemaker and try to find the best approach for prediction. At the end, we would provide an API to the user for making predictions on demand.

The major tasks here can be broken down into: 1) Efficiently performing read operations on billions of records in parallel stored on S3 in form of a CSV file. 2) Processing the data in order to make it ready for consumption by the machine learning model. This would involve performing data cleaning, feature engineering etc. 3) Storing the processed dataset to a new bucket which will act as the training set for machine learning models. 4) Training machine learning models and finding out the best prediction system through cross-validation. 5) Creating an API that takes parameters such as start and ending location and time of travel and receive a prediction.

Verification of this data pipeline would depend on the accuracy of the machine learning models as well as the scalability of our data pipeline to large amounts of data. We focus on the latter which would be performed by replicating the data on to larger number of records and testing the breaking point of our pipeline.

The 4 major milestones for this project are:

1) Breaking huge csv datasets into chunks for parallel processing and further feature engineering on a EMR cluster.
2) Storing the cleaned dataset from EMR on S3.
3) Training multiple machine learning models on the prepared data and then performing hyperparameter optimization using Sagemaker.
4) Provide an API for the user to get predictions using the trained model given some inputs.

Architecture

Figure 1 shows the solution Lambda architecture that we had proposed initially which was inspired by [3]. Lambda architecture has both batch layer and speed layer in addition to the

serving layer. The data provided by NYC-TLC is not real time. It comes in batches of days and months. This makes the speed layer redundant.

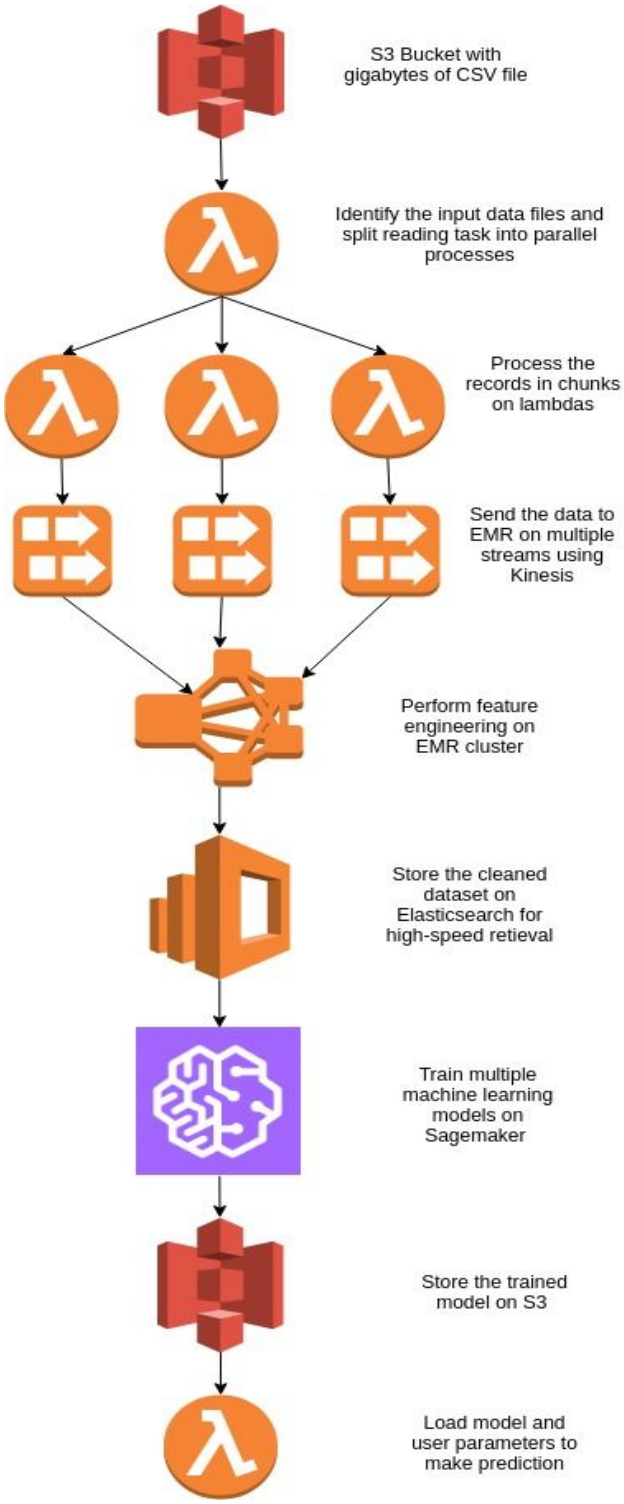

Figure 1: Data pipeline based on the lambda architecture

We moved away from the lambda architecture and focused on batch processing mode. This decision was taken based on the following concerns: 1) Over engineered solution - Since we are using AWS, it comes with a lot of added benefits. The previous architecture might fare well for real time data pipelines. Another scenario where it would be a good fit would be if it is a non-AWS infrastructure. In our case, we used AWS to deploy all the components. This laid the foundation for an inherently connected infrastructure that reduced the complexity. 2) AWS is inherently well integrated. Both EMR and Sagemaker can easily connect with S3. The makes a pipeline with Lambda functions and Kinesis redundant. 3) Complexity of the previous solution makes it difficult to manage. The new solution is fairly simple and efficient. 4) Cost effectiveness is another factor that we had to consider in our solution. With so many components, it seemed that Lambda architecture was not cost effective. The new solution is cost effective and minimalistic.

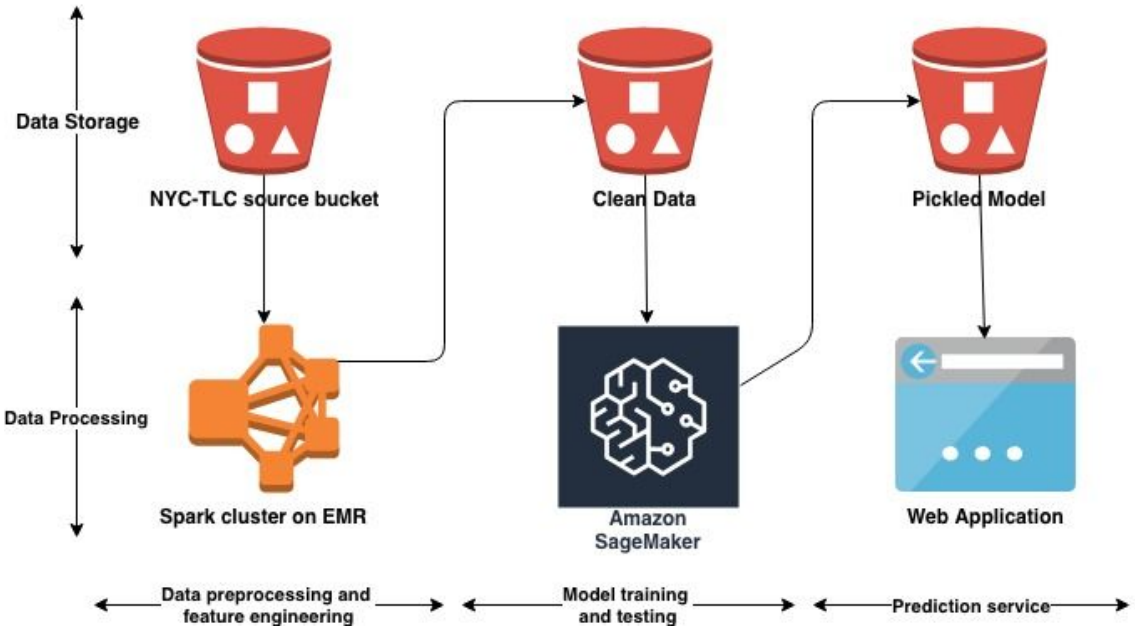

Figure 2: Batch processing data pipeline

Figure 2 shows the new architecture that processes the data in batch mode. Our pipeline ingests the data from a public repository of NYC-TLC hosted on S3. This repository houses three kinds of data. The first one is data related to 'for hire vehicles' or FHVs. The second type of data the yellow trip data' which comes from the yellow taxi cabs. The final one and the one we have based our study on is green taxi cab data.

Implementation

The resources used to accomplish this project are: 1) Public dataset hosted NYC-TLC on S3, 2) AWS EMR, 3) Apache Spark 4) AWS Sagemaker, 5) Pandas, 6) Python and 7) Flask.

In the final solution, we dropped the speed layer from the lambda architecture and only kept the batch and serving layer. This meant that we no longer required components like Kinesis, Lambda and ElasticSearch. We now implemented a solution where we ingest data directly into EMR running spark cluster on it from S3. The cluster preprocesses the data. Preprocessing includes removing NaN values, removing columns that were not useful, dropping rows with empty or illegal values and feature engineering. We engineered new features to make our predictions model robust - Zip Codes for the corresponding pickup and dropoff coordinates and Trip Frequency features for a particular geo-location to better learn about surge in bookings in an area. Generating these features is time taking in nature as it involves querying the past records for every trip record and perform some computations on top of it.

One of the key design choices of this project was choosing between a Lambda architecture and batch processing mode. As explained in the previous sections, Lambda architecture was not suitable for our use case. Another key design decision was the choosing the granularity of parallelisation. One choice was loading the files as RDD and splitting each of the files into chunks which would then be worked on in parallel. Another way to do this was to split the list of files into partitions and parallelise their processing in batches. The way we had to process the data, we had some component of the code that was serial and other that was parallel. The serial part of the code included identifying the columns in the data. The column naming is not consistent throughout the datafiles and we had to extract the columns based on certain logic that could not be parallelised. This was the bottleneck that we had to tackle.

In case of chunking the file, this approach failed to perform to our expectations. The reason for that was that by performing some part serially and some part parallely, we essentially had done a lot of disk based I/O. This is because once we load the file, the serial part is always executed on the master node and the parallel part is then sent to various worker nodes. Since the serial part required some action function on RDDs, we were essentially materialising the RDDs. This meant that in order to share the files among worker nodes, we had to make multiple action calls. Since RDDs are read only, this implied that new RDDs would need to be created after each action call and effectively, a lot of disk based operations would be incurring. The disk based IO made this approach extremely slow. Though this approach is very much correct, since the input file size of taxi data is maximum of 2GB, we chose to instead load the input files into memory as pandas dataframe. We still implemented this via RDDs. However, instead of moving data towards the

process, we moved the process towards data. This made a lot of difference in the performance as discussed in the next section.

We query S3 bucket with a specific prefix that contains green trip data. The master node loads the list into memory and partitions it. Each partition of the list is then sent to the worker nodes which then load each file into the memory, process the file and store it back to S3 bucket. This is much faster since the file is loaded into memory directly from S3 without intervention of disk.

The processed data is then stored to S3 by each of the worker node which completes the preprocessing of the data. After that, multiple models on Sagemaker are trained using Sagemaker's Jupyter notebook interface. We performed Grid Search over multiple models like Random Forest Regressor, Linear Regression, XGBoost, etc. for hyperparameter optimization and a 5 fold cross validation over all the models to find out which model gives out the least MSE and MAE scores. The model is trained on the algorithm that performs the best then it is stored back to S3 in a pickled format.

The pickled model can be downloaded and used. We developed a Python Flask web app that would allow users to query the model given parameters such as source address, destination address and passenger count. The flask app uses Google Maps API and geopy library to convert addresses to latitude and longitude.

Performance

We measured and compared the performance of our Spark implementation against a serial processing engine. The data preprocessing took upwards of 46 hours when done serially. On the other hand, parallel processing on 2 m4.xlarge worker instance with a total of 16 cores between them took nearly 3 hours. We see a linear scale-up in this case which is expected.

Throughout the implementation of this project, we tried to keep in mind the aspect of scalability and fault tolerance. AWS is inherently fault tolerant and scalable. To be specific data stored on S3 is has durability of 11 nines. EMR cluster is using Spark which use RDDs. RDDs contain lineage of operations performed on them which can be easily replicated if lost.

Related Work

Data pipelines are not a new concept and they have been in place since the early 2010s. We have studied some cases where Amazon provides an architecture for tackling such problems using their popular services. Our initial solution based on Lambda architecture was inspired from [3]. We

later moved on to a different architecture as described in Figure 2. The method described in [3] is a more general purpose solution and our implementation can be considered its subset.

Our solution is also inspired by [2], that explains in detail how a model data pipeline can look like. The key difference between our implementation and [2] is that the former uses a batch processing mode and latter uses stream processing mode. Due to this key difference, the tools used for the solution change as well. [2] uses Kinesis to stream messages and Apache Flink to process them, whereas in our implementation, we use Apache Spark for processing.

Conclusion

We were able to build a data analytics pipeline to perform data preprocessing and feature engineering of NTC-TLC trip data repository. We performed batch processing of over 20 million records in parallel and trained machine learning models on top of it. In addition to that, as part of our serving layer, we built a web application to query the models based on some parameters.

Future Scope

We have identified a couple of area where we can make improvements. One is the automation of entire pipeline. Currently there are some steps that we need to perform manually. Another thing that we can do is to run the whole processing in streaming mode. To elaborate more on this, we would be treating data as a stream of records that can be consumed through Kinesis and work on the problem as a real time analytics pipeline. It is arguable that the performance and results might not be as good as compared to the batch processing mode. In this case, Lambda architecture suggested in Figure 1 makes a lot of sense.

References

[1] Public NYC TLC data repository http://www.nyc.gov/html/tlc/html/about/trip_record_data.shtml

[2] Building data pipeline using Amazon Web Services https://aws.amazon.com/blogs/big-data/build-a-real-time-stream-processing-pipeline-with-apache-flink-on-aws/

[3] Lambda Architecture for Batch and Stream Processing https://d1.awsstatic.com/whitepapers/lambda-architecure-on-for-batch-aws.pdf