Distributed Data Analytics Framework for Cluster Analysis of Parking Violation

Nan Lin *, Evan Liu *, Fiorella Tenorio *, Xi Yang *, Diane Myung-kyung Woodbridge
{nlin7,eliu16,ltenoriocubas,xyang68,dwoodbridge}@usfca.edu
Data Science Program
University of San Francisco

***Abstract*—In this research, we employed distributed systems to explore the similarities in parking ticket records using unsupervised machine learning algorithms on a large dataset. Using 37 million ticket records (9 GB) collected by the New York City Department of Finance, we applied an algorithm to cluster existing tickets and dive deeper to find the distribution of precincts within different clusters. Amazon Web Services including S3, EC2 and EMR, and tools like MongoDB and Apache Spark were used in this endeavor. In this study, computational time and cost for different EMR settings were evaluated. We conclude that there are significant computational advantages to using distributed systems when implementing unsupervised learning on a large dataset as well as storing and managing data. We also observed that it is time efficient for a cluster with more workers instead of fewer workers with large memory space for the utilized data set. However we observed a trade-off between the execution time and the total cost for the cluster configuration.**

***Index Terms*—Distributed computing, Distributed databases, Distributed information systems, Machine learning, Transportation**

I. Introduction

In the fast growing cities, parking has been a major challenge for both residents and law enforcement agencies alike. A recent study showed that Americans, on average, spend 17 hours searching for parking spaces, resulting in a total of $20 billion in parking fines, and an individual cost to drivers in wasted time and fuel of $345 yearly [1]. Fees collected for parking violations are growing quickly, and have the potential to be a serious burden for the urban residents. For the last ten years, revenue from parking fines grew in New York City by 35 percent, raising $993 million, largely (55%) from parking violations in 2016 [2]. Many residents either cannot afford these parking fines or choose not to pay them, generating large amounts of uncollected court debt, not to mention wasted enforcement and judicial costs. In 2006, New York City had $600 million in uncollected parking violation fees [3].

There have been many studies analyzing parking behavior and proposing systemic improvements, with an emerging focus on leveraging the internet of things (IoT). Khana and Anand's recent work presented an IoT-based parking system that informs users about available parking spots using passive infrared and ultrasonic sensors installed in parking spaces [4]. Stenneth et al. utilized GPS and accelerometer sensors embedded in smartphones to detect vehicle locations and the locations of parking meters in an effort to analyze a driver's behavior and provide timely alerts to avoid parking violations [5]. Understanding a drivers; parking behavior using IoT sensors can help develop a dynamic pricing model based on supply and demand. A study by Holguin-Veras et al. conluded that New York and New Jerseys time of day pricing initiative had an impact on 7.4% of all trips, yielding behavioral changes in parking facility usage, time of travel, and payment type, among others [6]. Many companies are also developing and commercializing IoT infrastructure to improve real-time parking behavior monitoring, analysis and recommendations. Verizon recently announced 40% growth in IoT network connections for transportation systems in 2017 [7].

To store, manage and process real-time sensor data from multiple sources including drivers, parking enforcement officers, parking meters and sensors on both vehicles and roads, it is critical to develop scalable infrastructure that can store, manage and process data [8]. Relational databases, which are most widely used for storing and managing data, do not offer native mechanisms for data redundancy and availability in case of a database failure, nor scalability beyond a single server. Developing a machine learning model using large volumes of data on a single machine requires protracted execution times, and is often unable to provide feedback in real-time. While high-performance computing (HPC) or distributed computing showed improved execution time, HPC is more susceptible to failure and is more costly [9].

Understanding aforementioned issues, the purpose of this paper is to store parking ticket data generated by New York City (NYC) and to analyze data to find, if present, patterns of the vehicles being ticketed using a scalable data pipeline. We then utilized output clusters of ticketed vehicles for aligning with precincts, a geographical unit of the city. Also, the paper focuses on performance, time efficiency and expenses of the applied algorithms and compares the results with various deployment settings. We concluded that using distributed systems to implement unsupervised learning on a large dataset has big advantages regarding data management and time efficiency. We observed that it is time efficient for a cluster with more workers instead of fewer workers with large memory space for the utilized dataset.

* These authors contributed equally.

II. Background

A. NoSQL and MongoDB

Over the past two decades, technology companies have tracked detailed user behavior through websites and IoT devices in real-time, generating a huge volume of data, whose structure (or schema) is constantly evolving [10]. To store data with such explosive volume growth, there was a clear need for a robust, affordable, and responsive data management/storage solution. In the late 2000s, many research and open source projects including Google BigTable [11] and Amazon Dynamo [12] demonstrated strong performance and scalability using newly developed non-relational databases, NoSQL (not only SQL). Many new database management systems also support distributed data sources. By dividing and storing data in different servers (shards), data availability is greatly improved upon by maintaining replicas in multiple servers. Additionally, many NoSQL databases support storing schemaless data, and are designed to store data that is closely related, as an aggregate, in the same server node.

MongoDB, one of the most popular NoSQL databases, stores data in schemaless JavaScript Object Notation (JSON) document format, allowing users to easily add and remove fields in a document. MongoDB is designed to scale and and split data across multiple servers, and manages loading data across a cluster, balancing data distribution in multiple servers and routing user requests to the correct server. This enables users to focus on programming rather than low level system architecture and data distribution. MongoDB additionally supports indexing, to improve query performance and aggregation pipelines by designing a complex pipeline through the sequential combination of multiple simple queries [13].

B. MapReduce and Apache Spark

Hadoop's MapReduce, introduced in 2004, implemented efficient distributed techniques in an attempt to speed up large scale data processing and analysis [14]. MapReduce splits data into smaller chunks across different nodes, and subsequently maps and processes a task, e.g., filtering and sorting, in parallel. The output of a mapped task becomes the input of a reduce operation, which performs a summary operation. This highly-effective model allows users to design programs with successive map and reduce operations, and is a popular and powerful programming paradigm.

Spark was designed in UC Berkeley's AMPLab in 2009 and open-sourced in 2010. Although Spark adopts MapReduce concepts, it runs up to 100 times faster than Hadoop MapReduce, utilizing in-memory computing and an advanced task-execution engine [15]. Spark also has built-in libraries that allow for efficient iterative computation. Included with version 0.8 in 2013, MLlib was the library within Spark to support machine learning features. The early version of MLlib APIs worked with Sparks native Resilient Distributed Datasets (RDD)—a fundamental data structure of Spark—and has since been placed into maintenance in favor of MLlib for DataFrames, a spreadsheet-like collection of data organized in columns and rows [16].

There is a paucity of academic research that benchmarks Spark MLlib performance, either evaluating the library itself under varying conditions. A number of researchers from DataBricks authored a 2016 paper providing an academic introduction to MLlib [17], and discussing its performance across versions. This research provides a similarly extensive look at performance, examining results generated when running Spark MLlib on a specific dataset [18], [19].

III. System Overview

A. System Workflow

In this research, we built a scalable pipeline to store and process parking violation data, designed to be robust when dealing with the high frequency and volume of such data. Technologies and platforms were selected to build an unsupervised clustering model with high availability and quick execution times. We therefore selected Amazon Web Services (AWS), a cloud platform service as the primary platform to host storage, data extraction, transform and load (ETL) processes and machine learning tasks. An overview of data pipeline is noted in Figure 1.

1) Data Storage: Data collected from NYC OpenData [20] was stored in Amazon Web Service (AWS) Simple Storage Service (S3), a cloud storage service that can handle large volumes of data. S3 provides replicated hosting in multiple data centers, allowing high availability, as well as interoperability with other AWS components, including Elastic Map Reduce (EMR) and Elastic Compute Cloud (EC2). In this study, we used four years of NYC parking tickets data, from 2015 to 2018 with an initial total volume of over 8 GB. After applying preprocessing steps in Section III-B1, we ultimately stored 3.2 GB data into S3.

2) Data Management: We launched a total of 13 Elastic Cloud Compute (EC2) instances on top of which we mounted MongoDB architecture. Of the 13 instances, we structured them so that we had 3 shards for splitting the entire data set into three subsets where each one contains 3 instances; other 3 instances were used as a configuration shard and its two replicas; and, the last instance was used as mongos, a routing service node which takes user requests to the correct instance that contains the requested data. In each of the 3 shards and 1 configuration shard, we use a total of 3 instances, where one of these instances can be used as the primary node, and the other two are the secondary nodes. Each primary node is responsible for reading and writing operations, and copies data to secondary nodes has the responsibility to maintain replicated data in case a master node fails due to networking, power outage and other system failures. The configuration shard has the responsibility to manage the metadata of other nodes and the overall database. The other 3 shards contain the distributed data.

Figure 2 describes the MongoDB configuration. MongoDB manages data balancing and routing from instances with different hardware specifications.

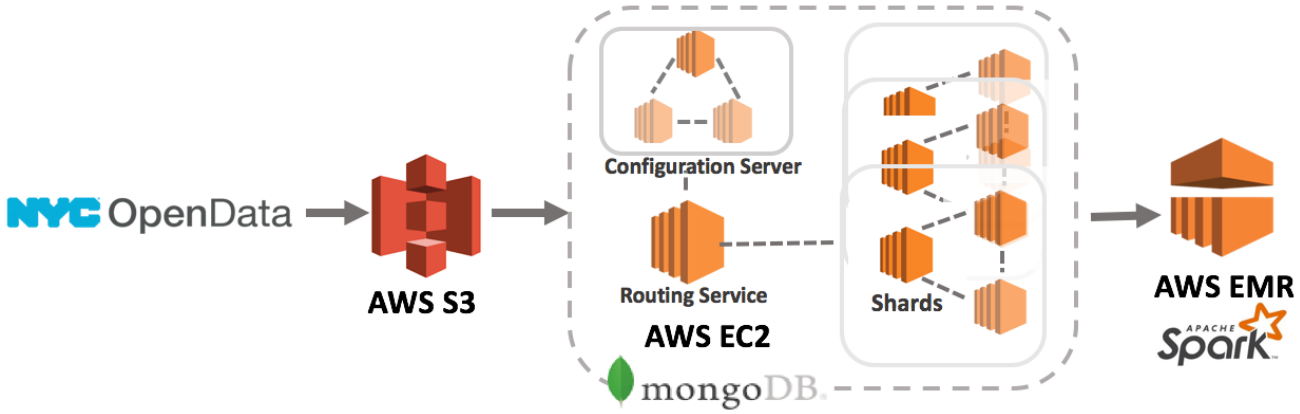

Fig. 1: System workflow

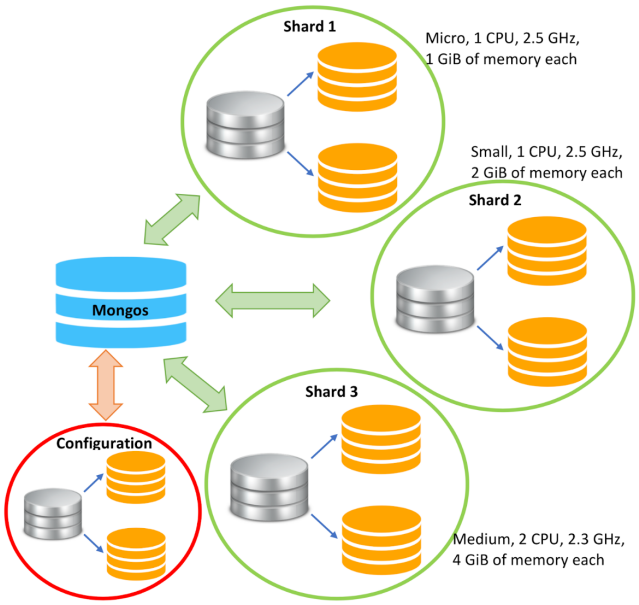

Fig. 2: MongoDB setting

3) Data Analysis: AWS EMR uses Hadoop's YARN (Yet Another Resource Negotiator) as the cluster manager, and automatically provisions hardware resources (EC2 instances) and installs the required software for running Apache Spark. Apache Spark comes with Spark SQL and Spark MLlib, which enable distributed data preprocessing, feature engineering and modeling. In order to compare the efficiency of different clusters, we launched four types to clusters with different configurations. The detailed setting are described in Table I.

TABLE I: Cluster Types and Specifications

	YARN 1	YARN 2	YARN 3	YARN 4
Type of EC2 instance(s)	M4.2xLarge	M4.xLarge	M4.xLarge	M4.xLarge
Number of instances	3	3	5	6
Number of worker nodes	2	2	4	5
Number of cores per worker	16	8	8	8
RAM per worker (GB)	32	16	16	16
Disk Storage per worker (GB)	32	16	16	16
Cost per hour ($)	0.36	0.18	0.3	0.36

B. Algorithms

We implement an unsupervised machine learning algorithm, *k*-means clustering, in an effort to identify similarities among ticketing data [21]. In our final model, each parking citation is assigned to a cluster. We subsequently map precincts to their corresponding clustered observations. Once mapped, we then calculate the distribution of tickets per precinct within a cluster, allowing us to identify the precincts in which a vehicle is more prone to be ticketed.

1) Data Preprocessing: We use two major classes of features from the source data: vehicle-related features (e.g., vehicle color, type, etc.) and time of day. We eliminated variables from the original data as follows:

- Variables with uniquely-identifying information information about both the car and the driver, to ensure anonymity.
- Certain features related to violation details such as officer information and violation description, as our focus is on vehicle characteristics and time of day.
- Location-related data is removed during the clustering process, but added back post-clustering to map cars to precincts.

After removing unnecessary variables, the data used as input to the clustering algorithm was 3.2 GB.

2) Feature Engineering: Many of the features used as input to the clustering algorithm were categorical, such as vehicle color, make, type, registration state and license plate type. Since the *k*-means clustering alorithm only operates on numeric data, we converted categorical data to a numerical representation by applying string indexing and one hot encoding algorithms. String indexer algorithms convert categorical string values into integer indexes. One hot encoding expands a column to as many columns as distinct strings in the category and creates a sparse vectors where only one column contains a 1 and others are 0 [22].

Other numeric features of the data include violation time, where we extracted hour (from 1 to 24), day of the week (from 1 to 7) and month (from 1 to 12) from the violation date.

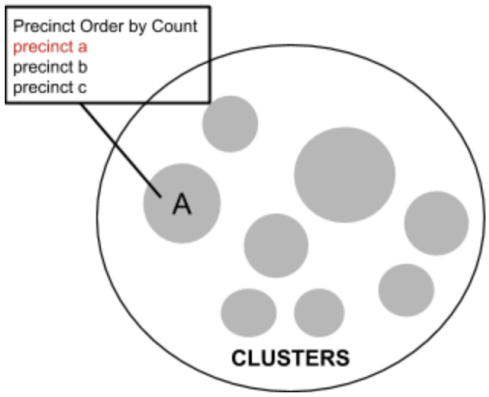

Fig. 3: The k-means algorithm clusters parking tickets based on the vehicle characteristics and time.

3) Machine Learning Model: k-means clustering aims to partition n observations into k cluster, where each observation belongs to the cluster with the nearest mean distance. k-means performs the process by repeatedly assigning data points to initial centroids, and iteratively updating centroids and assignment of obersvations to those centroids until a stopping condition is satisfied.

For an m dimensional feature vector, let us assume that the observations are labeled $X_{(1)}; X_{(2)}; :::; X_{(n)}$. The clustering algorithm requires a hyper-parameter as an input, namely, k, the number of clusters to be identified. The algorithm executes as follows:

Algorithm 1 k-Means Algorithm

Initialize cluster centroids $u_1; u_2; :::; u_k \in R^m$ randomly
while u_j *is not converged* **do**
  foreach i **do**
    $u^{(i)} := arg\ min_j ||x^{(i)} - \ _j||^2$
  end
  foreach j **do**
    $\ _j := \frac{\sum_{i=1}^{m} fc^{(i)}=jgx^{(i)}}{\sum_{i=1}^{m} fc^{(i)}=jg}$
  end
end

To select the hyper-parameter k, we chose an optimal value of k based on a scaled within set sum of squared errors (WSSSE) metric. WSSSE is defined as the sum of squared errors within each cluster. We then employed the optimal k across different AWS EMR configurations to compare execution performance, time efficiency and overall cost.

IV. Experiment Output

A. Data

The data used for the experiments is the New York parking tickets data set from NYC OpenData, which spans the fiscal years between 2015 and 2018 [20]. After an initial preprocessing, the size of data used as input of the clustering algorithm was 3.2 GB which contains 36.5 million observations. Features of the input data include:

Plate Type: Passenger, commercial, taxi, bus, etc.
Registration State: The state where the vehicle was registered
Vehicle Color : The color of the vehicle
Vehicle Body Type: SUV, sedan, truck, trailer, etc.
Issue Quarter: The quarter when the ticket was issued
Issue Month: The month when the ticket was issued
Issue Day of the Week: The day of the week when the ticket was issued
Violation Time: The hour of the day at which the ticket was issued

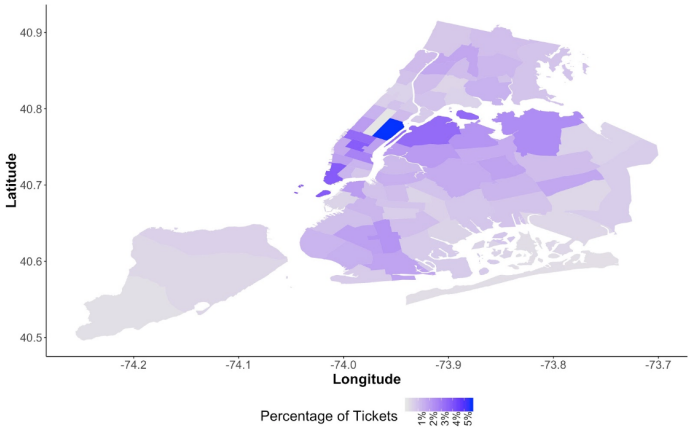

Fig. 4: Heat map of the number of tickets per precinct in 2018

As part of our initial exploratory data analysis of the data, we were created a heatmap for all parking violations in 2018. The frequency of parking tickets within each precinct are shown in Figure 4. The darkest region shows the Upper East Side of Manhattan with the most parking tickets while the lighter colored regions including Brooklyn and Queens indicates the precincts that were not heavily ticketed.

B. Results

The first step in our analysis requires determining the correct number of clusters, k. To ascertain optimal k, we needed to obtain a within set sum of squared error (WSSSE) value for different k values for determining the optimal k with the maximal improvements in WSSSE and minimal execution time. For each cluster k, the WSSSE is computed as follows:

$$WSSSE_k = \sum_{k=1}^{K} \frac{1}{2n_k} D_k$$

where

$$D_k = \sum_{x_i 2 C_k} \sum_{x_j 2 C_k} ||x_i - x_j||^2 = 2n_k \sum_{x_i 2 C_k} ||x_i - \ _k||^2$$

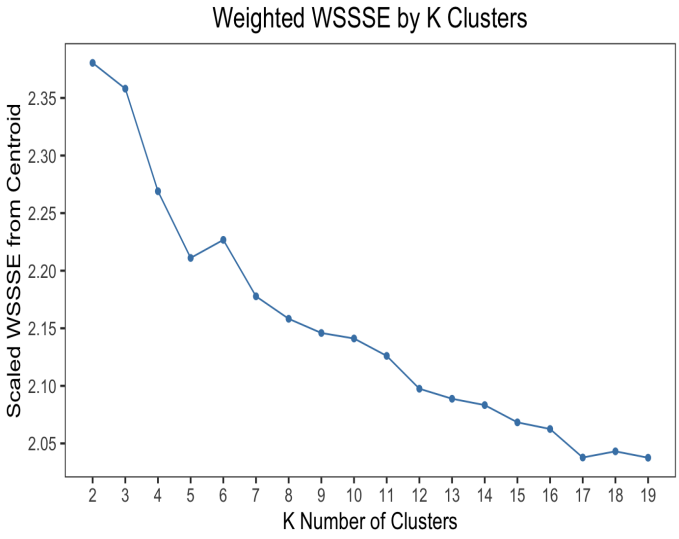

Fig. 5: Weighted WSSSE and k

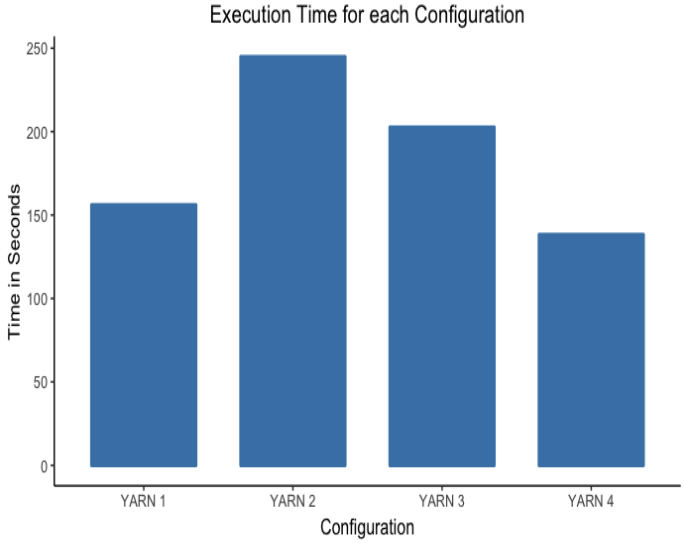

Fig. 6: Execution Times for Different YARN Configurations

TABLE II: Cost for building a model and average distances from centroids, when $k = 12$.

	YARN 1	YARN 2	YARN 3	YARN 4
Number of instances	3	3	5	6
Number of cores per worker	16	8	8	8
RAM per worker (GB)	32	16	16	16
Time per model (sec)	156.1064	244.8653	202.6197	138.2841
Cost per model (USD)	0.94	0.73	1.01	0.83
Distance from Centroid (standardized WSSSE)	2.1277	2.1210	2.1089	2.1136

As WSSSE values are difficult to interpret, owing to the size of the data, we opted to standardize the WSSSE by dividing by n, the number of observations, and taking the square root.

$$Standardized \quad WSSSE_k = \frac{\sqrt{\sum_{k=1}^{k} \frac{1}{2n_k} D_k}}{n}$$

We plotted scaled WSSSEs where k ranges from 2 to 20. We ultimately selected $k = 12$ as an ideal numnber of clusters, where we begin to observe diminished returns if k were to increase to larger values (see Figure 5).

We ran the k-means algorithm on four different EMR clusters— described in Table I—and compared speed, performance and overall cost as shown in Table II. As a general result, a large number of workers in a cluster performs faster. Figure 6 shows that the execution time for **YARN 2**, with only 3 workers and the lowest memory settings is the slowest among them all; the other three produced similar results. **YARN 4** is the fastest among all tested configurations, owing to it having the most workers. Mean execution time per worker was plotted in Figure 7. When comparing **YARN 1** and **YARN 2**, the configuration with more memory has a shorter execution time, *ceteris paribus*. Similarly, configurations with more workers had shorter execution times. YARN 4, with the largest number of workers, performs faster than all comparative YARN configurations.

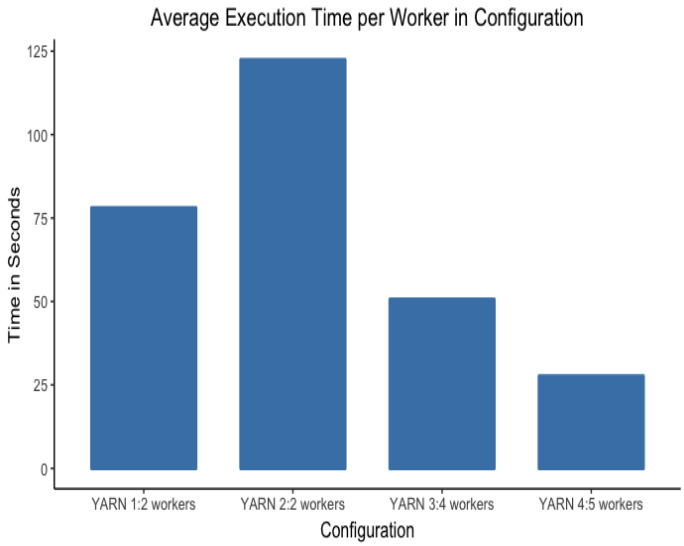

Fig. 7: Mean Execution Time per Worker for Different YARN Configurations

We also observed that the **YARN 2** configuration, with the slowest execution time, actually has the lowest total cost, as shown in Table II. Therefore, if operational costs are an issue, the configuration with shortest execution time is not the most cost-effective solution.

We also imported individual clusters and visualized them on a heatmap, as shown in Figures 8, allowing us to visualize the frequency of ticketed precincts. Of note is that Times Square is more frequently than other precincts, likely owing to the high level of traffic.

V. Conclusion

In this research, we applied k-means clustering to 37 million ticketed observations and compared the computational time