Faculty of Natural and Mathematical Sciences
Department of Informatics

The Strand
Strand Campus
London WC2R 2LS
Telephone 020 7848 2145
Fax 020 7848 2851

7CCSMDPJ

Individual Project Submission

2016/17

Name:	Chandana Karunaratne
Student Number:	1621633
Degree Programme:	MSc Data Science
Project Title:	Exploring Socio-Demographic Traits of TfL Passengers
Supervisor:	Professor Elizabeth Sklar
Word Count:	12,243

RELEASE OF PRODUCT

Following the submission of your project, the Department would like to make it publicly available via the library electronic resources. You will retain copyright of the project.

☑ I **agree** to the release of my project

☐ I **do not** agree to the release of my project

Signature: **Date:** August 25, 2017

Abstract

This report aims to analyze the travel patterns and socio-demographic characteristics of Transport for London (TfL) passengers. It uses a dataset of passenger journeys taken on tube and rail services during the period, 20th October – 14th December 2013. Key research questions addressed in this report relate to identifying the approximate place of residence of passengers, grouping these passengers based on shared socio-demographic characteristics, and determining the major travel patterns of these groups. A variety of approaches were taken to address the research questions, including an analysis of the travel behaviour of selected passengers to determine an approximation of their most likely place of residence, followed by cluster analysis on passenger data and associated socio-demographic information, concluding with an examination of the travel patterns of each of the groups revealed in the clustering. Results from the analysis indicate that the passengers in the dataset can be grouped into eight distinct clusters based on shared characteristics relating to ethnicity, age group, marital status, employment sector, and education levels. Furthermore, each of these clusters exhibited specific travel behaviour with respect to the TfL stations most frequently used and the days of the week and the time of day during which journeys took place.

Table of Contents

List of Figures

List of Tables

Acknowledgements

Firstly, I would like to thank my project supervisor, Professor Elizabeth Sklar, who gave me encouragement, motivation, and valuable advice on a regular basis. Furthermore, I would not have been able to undertake this MSc programme without the love and support of my parents, to whom I will be eternally grateful. I would also like to express my gratitude to Eric Schneider and Roger Moore, who provided me with much-needed technical assistance when accessing the database used in this project. Lastly, but perhaps most importantly, I would like to thank my wife for sticking by my side throughout this journey.

1. Introduction

Transport for London is the government body providing integrated public transportation services that cover Greater London and its surrounding areas. It offers a variety of transport means, including underground transit, suburban rail, trams, and riverboat services, among others. During the 2015-2016 fiscal year, over 1.65 billion passenger journeys were taken on the London Underground, Docklands Light Railway, and London Overground services combined [24].

The purpose of this dissertation is to analyze a dataset consisting of a sample of these passenger journeys. Doing so can help provide an indication of TfL passengers' travel patterns, including the most frequently used stations, the days on which passengers travel most often, and the times during which passengers take these journeys. Furthermore, it is also possible to examine the most likely socio-demographic characteristics of passengers by merging TfL passenger data with socio-demographic data based on geographic location. A further step in this analysis is to group these passengers together based on shared characteristics, using a data mining technique called *clustering*. Once passengers are grouped together, it is possible to extract useful information on the travel patterns of these distinct groups, thereby providing an indication of how passengers who share similar characteristics may behave with regard to their travel patterns.

The dataset of passenger journeys consists of approximately 940 million different records, with each record containing data on 88 distinct attributes. All records consist of data that was collected from an Oyster smart card, which is an electronic contactless ticket that can hold data related to travel credit as well information on season passes [25]. It is important to note that all data was anonymized and no personal information was available. Key information provided in the dataset includes the unique identifier of the respective Oyster card used, the date and time of each journey, the station at which the journey commenced, and the station at which the journey ended.

In order to study the likely socio-demographic characteristics of TfL passengers, the London Output Area Classification (LOAC) dataset was used [14]. This is an open-source geospatial classification of areas throughout Greater London [14]. It uses Census data from 2011 to categorize approximately 25,000 geographic areas in the city based on their respective socio-demographic characteristics. There are over 60 socio-demographic attributes in the LOAC dataset relating to five key areas, including demographic, employment, household composition, housing, and socio-economic related characteristics.

The key research questions that are to be addressed in this report include the following:

i) What information can be uncovered about where passengers live based on their travel patterns?

ii) How can we group TfL passengers based on their socio-demographic characteristics?

iii) What information can be uncovered about the travel patterns of different groups of TfL passengers based on their socio-demographic characteristics?

The remaining sections of this report are structured in such a way as to provide a discussion of how the methodology was designed to address the above research questions and an interpretation of the results obtained from subsequent analysis. Specifically, to provide a basic understanding of the use of clustering and its applications in the realm of transport-related data, a brief survey of literature is provided in sections 2 and 3. This is followed by an overview of the approach used to analyze the data, in section 4, with an interpretation of the results provided in section 5. Finally, section 6 offers concluding remarks covering an overview of the key findings, ways in which the analysis can be used in real-world applications, and opportunities for future work.

2. Background

One of the key resources used in this project is the London Output Area Classification report [14] and its accompanying dataset. As mentioned above, it uses Census data to categorize geographic areas in Greater London based on their respective socio-demographic characteristics. The authors use a method called clustering to categorize the geographic areas into eight main groups, called *Supergroups*, based on similarities among their respective socio-demographic attributes. The characteristics of the LOAC dataset will be discussed in detail in section 4.4, when it is used in conjunction with the TfL dataset to derive socio-demographic traits of TfL passengers.

2.1. Overview of Clustering

Clustering is a method of grouping data together, in such a way that data points within a cluster are more similar to each other compared to data points in other clusters. In fact, it is regarded as a method of determining the "natural groupings" into which data can be categorized [11].

According to Tan et al. [23], there are three specific purposes of clustering - i) summarization, ii) compression, and iii) efficiently finding nearest neighbours. Summarization is used when there is a need to conduct algorithms that use high space and time complexities, such as regression or Principal Component Analysis (PCA), on large-scale datasets. Due to the fact that such algorithms are resource-heavy, it may be prohibitively expensive to run these algorithms on the full dataset being used. As such, summarization is used to run these algorithms on smaller datasets consisting of cluster prototypes, which are data points that represent each cluster in the full dataset. Doing so reduces the resources needed to conduct more advanced analysis like regression or PCA and potentially provides results that may be comparable to those obtained using the entire dataset. Tan et al. [23] also highlight the importance of clustering with regard to data compression, which is used in situations where there is a need to reduce the size of a dataset. Also known as vector quantization, compression uses cluster prototypes to help analyze large-scale data used in image, sound, and video files [23]. Tan et al. also point out that clustering can be used to efficiently find the nearest neighbour of a data point. Since calculating the distance between two points for all points in a large dataset is an inefficient method to find the nearest neighbor, using cluster prototypes can improve efficiency and obtain similar results [23].

2.2. Clustering Techniques

There are three key clustering techniques that are used frequently, with each having its own strengths and weaknesses.

2.2.1 K-means Clustering

The most common clustering technique is k-means clustering, which uses an algorithm that was first developed in 1955 [11]. It employs an iterative approach to determine the closest cluster center, or centroid, for each data point and continues to update the cluster centroids until there are no significant changes in centroid assignment. In order to determine the closest centroid, the algorithm uses a distance measure, the most common of which is Euclidean distance. Equation 1 below shows the formula used to calculate Euclidean distance, which is provided by the square root of the sum of squared distances between two data points [8]. The k-means algorithm uses cluster prototypes to represent each cluster, either as the centroid of the cluster when the data is continuous, or as the medoid of the cluster when the data is categorical. Another key characteristic of the k-means algorithm is that it uses partitional clustering, in which the data is divided into separate clusters that do not overlap.

$$d(q,p) = \sqrt{\sum_{i=1}^{n}(q_i - p_i)^2} \tag{1}$$

where d refers to the distance between two points, q and p, in the Cartesian coordinate system and i refers to the respective attribute, with a total of n attributes

Some of the key strengths of this algorithm is that it is simple compared to other clustering techniques and is highly efficient in finding clusters [23]. It can also be used for a variety of different types of data. As pointed out by Han et al. [9], this algorithm is particularly useful when conducting clustering on large-scale datasets. As stated by Cai et al. [4], the classical form of k-means clustering relies on centroid-based clustering. It is beneficial for the analysis of large-scale sets of data due to its low computation cost and the fact that it works well with parallelization processes used in distributed computing.

However, one of the key drawbacks of the k-means algorithm is that it does not perform effectively for data that features clusters of varying degrees of density, as discussed in [23]. Furthermore, the algorithm performs poorly on data that contains a large number of outliers. Han et al. [9] points out that a core weakness of the k-means algorithm is that it is susceptible to outliers because of their ability to impact the mean values that are calculated as part of the algorithm.

2.2.2 Agglomerative Hierarchical Clustering

Another common clustering technique is agglomerative hierarchical clustering. This algorithm initially treats each data point as a distinct cluster, and at each level of the hierarchy, each pair of clusters that are deemed most similar (or closest to each other) are merged together to form a new cluster [23] [6]. This technique can be represented using a dendrogram, which presents the sequence in which the clusters are merged in a tree-like hierarchical structure, as shown in Figure 1 below. The similarity, or cluster proximity, between clusters can be computed in a variety of ways, the most common of which include the shortest distance between any two clusters (known as the single link measure), the longest distance between any two clusters (known as the complete link measure), and the average distance between cluster members (known as average link or group average measures) [23] [6]. Visual representations of these three measures are shown in Figure 2 below.

Key benefits of using agglomerative hierarchical clustering techniques include their inherent application towards clustering applications which call specifically for hierarchical classification[23].

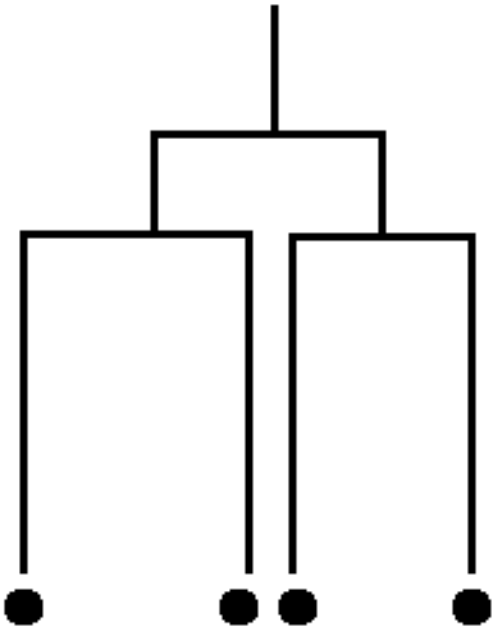


Figure 1: *Dendrogram of Hierarchical Clustering*

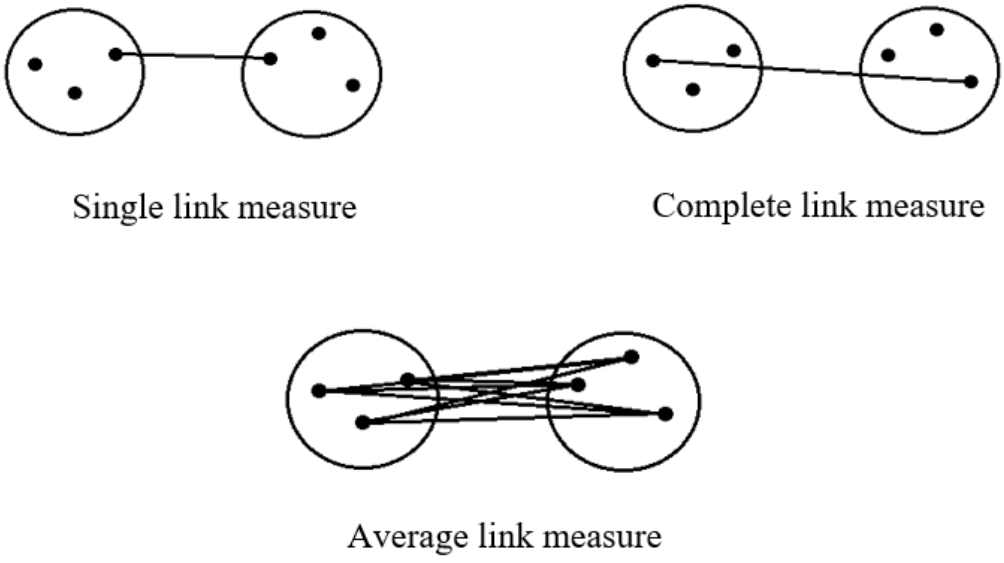


Figure 2: *Three Key Measures of Cluster Proximity*

An example of such applications includes grouping different biological organisms into a taxonomic hierarchy. In such applications, using the agglomerative hierarchical clustering algorithm may be the most appropriate clustering technique. Another benefit is that agglomerative hierarchical clustering can take into account clusters of different sizes by adding weights to each data point based on the size of the cluster to which it belongs. This is particularly useful for datasets which feature clusters that vary in size and density.

However, one key drawback of agglomerative hierarchical clustering is that it may perform poorly on large datasets due to the relatively high time and space complexities required by the algorithm. [23] Another potential weakness of the algorithm is that merges between clusters are irrevocable, which may prevent local minima or maxima from translating into global minima or maxima; this is particularly problematic for high-dimensional data [23]. Furthermore, Dudas et al. [6] point out that agglomerative hierarchical clustering may perform poorly on datasets with many outliers.

2.2.3 DBSCAN Clustering

DBSCAN clustering is a technique that uses density-based clustering, which groups together data points located in highly dense clusters, omitting data points that are located in low-density areas. Like the k-means algorithm, DBSCAN uses partitional clustering, which separates data points into clusters that do not overlap. However, unlike the k-means algorithm, DBSCAN automatically finds the most appropriate number of clusters. Furthermore, since DBSCAN omits data points in low-density areas, which it considers noise, it produces clusters that are more densely packed.

The DBSCAN algorithm [23], [6] classifies each data point as either a core point, which refers to a data point located towards the center of a highly dense cluster; a border point, which refers to a data point located towards the periphery of a highly dense cluster; or a noise point, which refers to a data point located outside the periphery of a highly dense cluster. A user-specified radius and a user-specified minimum threshold for the number of neighbouring points are used to determine which of these three categories each data point falls into. Specifically, if a data point has a certain number of neighbouring points within a user-specified radius, and if this number of neighbouring points exceeds the user-specified minimum threshold, the data point is classified as a core point. Alternatively, if a data point falls within the user-specified radius of a core point but does not have the minimum number of neighbouring points to consider it a core point, it is classified as a border point. Lastly, if a data point cannot be categorized as either a core point or a border point, it is classified as a noise point.

One key benefit of the DBSCAN clustering technique is that it is designed to discard noise in the dataset, and thereby deals effectively with outliers, as pointed out by Tan et al. [23] and Dudas et al. [6]. Furthermore, it has been shown to work effectively on clusters of various sizes [23]. Another benefit of the DBSCAN algorithm that Dudas et al. [6] highlight is that there is no need to determine the number of clusters to be used prior to running the cluster analysis since this is automatically computed during analysis.

However, a key drawback of the DBSCAN algorithm is that it does not perform effectively on clusters of various densities [23]. Hyde et al. [10] also point out that the DBSCAN algorithm has difficulty locating clusters of widely different densities since the algorithm uses a predefined value for the minimum density. Additionally, this algorithm performs poorly on high-dimensional datasets particularly because computing distances between data points and their nearest neighbours, as is required in the DBSCAN algorithm, is computationally expensive for high-dimensional

datasets [23] [10]. As Dudas et al. [6] point out, the DBSCAN algorithm has a relatively high time complexity of $O(n^2)$, which is particularly problematic for large datasets.

2.2.4 Evaluation Methods used in Clustering

In order to evaluate the performance of a particular clustering technique on a given dataset, it is important to consider various evaluation methods. Among the key measures used are the Inertia score and the Calinski-Harabasz score.

The Inertia score measures the distance between each data point and its respective cluster centroid and subsequently sums these values. As such, the less distance there is between data points and their respective cluster centroid, the higher the density of data points in that cluster. Thus, a relatively low Inertia score indicates that a cluster is relatively densely packed.

The Calinski-Harabasz score uses a measure that combines the distances between data points within a cluster and the distances between distinct clusters [17]. It does so by calculating the ratio between the average sum of squares between clusters and the average sum of squares of data points within a cluster [15]. As such, clusters that are relatively similar, or in close proximity, to one another will receive a relatively low score, while clusters that are densely packed, with short intra-cluster distances, will receive a relatively high score. Thus, a relatively high Calinski-Harabasz score indicates that a clustering fits the data well.

3. Related work

3.1. Applications of Clustering Using Transport Data

The following section provides specific examples of how the clustering techniques outlined above can be applied in real world scenarios.

3.1.1 K-Means Clustering

Martins-Yedenu [16] applies k-means clustering in the realm of transportation using two datasets. The first dataset covers TfL journeys conducted in Greater London over the course of two months; the second dataset relates to the amount of rainfall for locations throughout London. By using both datasets together, the author finds changes in travel patterns among TfL users based on the amount of rainfall at each TfL station. He uses k-means clustering to group TfL stations into clusters based on passengers' use of these stations during incidences of light, moderate, and heavy rainfall or none whatsoever. His clustering analysis determines three clusters, with the first referring to stations which experience no substantial change in usage between no rainfall and light rainfall and some change in usage between light and moderate rainfall; the second cluster refers to stations which experience a higher usage between no rainfall and light rainfall and a significant increase in usage between light and moderate rainfall; the third cluster refers to stations with high usage in all weather conditions. He concludes that the usage of TfL stations is impacted substantially by the severity of rainfall near a station, and in particular, that there is a significant increase in usage of a station when there is moderate rainfall in the area.

In another paper examining public transport systems, the authors of [3] use clustering techniques to analyze smart card data obtained from the bus transport system in Quebec, Canada. Using

a data set of approximately 9.4 million records over a one-year period, the authors apply k-means clustering to categorize the data set into three distinct clusters, with each relating to the approximate time of day during which the customer travels. The first cluster is distinguished by a tendency to travel during both morning and evening hours; the second cluster is distinguished by a tendency to travel predominantly during the morning hours; and the third cluster is distinguished by a tendency to travel predominantly during the evening hours.

Anand et al. [2] use a combination of factor analysis and k-means clustering to cluster data obtained from the Household Interview Survey relating to the use of transportation in Hyderabad, India. They use parallel coordinate plots to visualize the high-dimensional data for two scenarios - one in which k-means clustering is carried out on the original dataset, and another in which cluster analysis is carried out after carrying out factor analysis using principal component analysis (PCA). Doing so provides a visual representation of some of the benefits of conducting factor analysis prior to cluster analysis, namely, that it provides greater clarity when visualizing the data and more clearly shows strong correlations between different components. The authors highlight the benefits of carrying out factor analysis prior to conducting cluster analysis, particularly since they found that the clusters obtained solely from conducting cluster analysis on the original data did not accurately reflect the relationships between variables. Furthermore, the authors find that using factor analysis prior to cluster analysis facilitates the interpretation of clusters, improves the performance time of analysis, and reduces the error associated with clustering.

Ortega-Tong [21] uses clustering analysis on smart card data from London's public transport network to classify its users. The author uses a dataset of 500,000 records of Oyster card users randomly sampled from a database of approximately 130 million records (covering use of Transport for London's London Buses, Underground, Overground, National Rail, Tramlink, and DLR services) over two one-week periods in October 2011 and October 2012. The author carries out cluster analysis to categorize the users in the dataset into eight distinct groups relating to their frequency of travel throughout the week, with clusters 1-4 travelling relatively frequently and clusters 5-8 travelling relatively infrequently. Hierarchical clustering methods were not used due to the high computational costs associated with large datasets. To validate the results of the cluster analysis, the author uses two indices - namely, the Davies-Bouldin index and the Calinski-Harabasz pseudo-F-statistic.

Morency et al. [19] use smart card data collected over a period of ten months from a public bus transport system in Gatineau, Quebec, Canada. The dataset consists of approximately 6.2 million records and contains information about the type of card used for the journey, and the date and time of the journey. In order to conduct the k-means clustering analysis, the authors only use data pertaining to two specific users - an elderly passenger and an adult passenger. Data for each user is available for the entire duration of 277 days. The authors used the k-means algorithm to cluster the passenger data based on the time at which the journey commenced and the day on which the journey took place. The results indicate that the elderly passenger's travel behaviour could be grouped into 29 clusters while the adult passenger's travel behaviour could be grouped into 15 clusters. The authors contend that the results from the cluster analysis show that the adult passenger exhibited less varied travel times compared to the elderly passenger and thus has more regular travel patterns compared to the elderly passenger.

3.1.2 Agglomerative Hierarchical Clustering

Another popular clustering technique is agglomerative hierarchical clustering. Agard et al. [1] use this technique to analyze smart card data obtained from a public transport system in Quebec, Canada. The dataset was obtained from the public bus transport authority in Quebec and consists of records relating to individual bus journeys, with information provided on the specific card used, the location at which a passenger boards a bus, the time at which this occurs, the date of the journey, and the bus route for that respective trip. The data was collected over the course of 10 weeks and consists of approximately 2 million records. The authors use k-means clustering to obtain 20 distinct clusters, which they use to feed into an agglomerative hierarchical clustering algorithm to group passengers based on their travel patterns, particularly with regard to the time of day travelled and the day of week travelled. The authors produce four clusters consisting of passengers who i) travel regularly during peak hours (cluster 1); ii) travel regularly predominantly in the morning (cluster 2); iii) do not seem to exhibit any distinct travel patterns (cluster 3); and iv) do not use the public transport system much (cluster 4). Based on this clustering, the authors are able to determine that cluster 1 relates predominantly to adult passengers, while cluster 2 relates predominantly to student passengers and cluster 4 relates predominantly to elderly passengers. Passengers in cluster 3 do not fit any specific demographic.

Dudas et al. [6] also consider agglomerative hierarchical clustering when analyzing data on vehicle stops. They use a dataset of vehicle stops obtained from Germany over the course of one day and run clustering to determine the key hubs at which the vehicles make prolonged stops. The authors use the complete link measure, mentioned above in section 2.2, on the agglomerative hierarchical clustering to find 5, 50, and 100 clusters in the dataset. They conclude that this clustering method performs poorly in the presence of outliers and does not effectively remove noise from the dataset when creating clusters, opting instead for DBSCAN clustering, as discussed below.

3.1.3 DBSCAN

Dudas et al. [6] use DBSCAN as a clustering technique to analyze a separate dataset from the one discussed above, containing location data of vehicles in Stockholm, Sweden. The data used relates to the geographical coordinates of buses and trucks manufactured by a specific Swedish automotive company over the course of one year and includes information on the vehicle's position (in longitude and latitude), its velocity, and fuel level, among other indicators. The authors used this data to calculate the duration of each stop that each vehicle makes. They subsequently used the DBSCAN algorithm to cluster this data in order to determine the key hubs of vehicle stops across Stockholm. The DBSCAN algorithm determined that there were three clusters in the stop data, with the first relating primarily to stops that took place at approximately 6:30 am, another relating to stops that took place around 7:30 am, and the third relating to stops that took place in the city center but at no specific time of day.

In another study, Galba et al. [8] compare the two clustering algorithms, k-means and DBSCAN, to classify their dataset into groups. To carry out their clustering analysis, the authors use a transport dataset of approximately 4 million records from the city of Osijek, Croatia, consisting of data on vehicle positions (based on latitude and longitude), their speed, and their angle of direction over the course of four years. The authors report that DBSCAN, compared to k-means clustering, has the advantage of being able to establish which data points are outliers.

4. Approach

4.1. Justification for Using K-means Clustering for this Project

As indicated in the sections above, each of the three clustering techniques discussed has its own set of strengths and weaknesses and can be used in a variety of transport-related clustering applications. The agglomerative hierarchical clustering technique is particularly useful for data which requires hierarchical classification and performs well on data containing clusters of different sizes and densities. The DBSCAN clustering technique, on the other hand, is particularly suitable for data in which there is a high incidence of noise and can efficiently remove outliers and is useful for data containing clusters of various sizes.

However, the k-means clustering technique appears to be the most suitable for the dataset used in my project for a variety of reasons. Unlike the agglomerative hierarchical clustering algorithm and the DBSCAN algorithm, the k-means algorithm is particularly suitable for large-scale datasets because of its high efficiency (and low space and time complexities) [9], [4]. Considering the dataset used in my project consists of over 940 million records, the k-means algorithm may be most suitable among the three algorithms considered. Furthermore, as pointed out by Tan et al. [23], Hyde et al. [10], and Dudas et al. [6], the hierarchical agglomerative clustering technique and DBSCAN clustering technique perform poorly on high-dimensional data. Since my dataset consists of over 50 fields relating to various socio-economic attributes, these two clustering techniques may not be sufficiently efficient when clustering this data, whereas k-means may be more successful due to its relatively high efficiency.

Thus, the preceding survey of literature on key clustering techniques and their respective strengths, weaknesses, and applications in the realm of transport-relate data assisted in the selection of an appropriate clustering method to be used for this project.

4.2. Preliminary Analysis

The next stage in the approach used in this project was the Preliminary Analysis, which was conducted in order to gain an overview of the TfL dataset and a preliminary understanding of what might be achieved through analysis of the data. As such, both the literature survey and the Preliminary Analysis stages were used to design the approach to be used when conducting a comprehensive analysis of the data, particularly in relation to conducting cluster analysis and socio-demographic analysis of the passengers in the TfL dataset.

The Preliminary Analysis can be described as follows:

1) Obtain a basic understanding of the dataset.

Initially, the database that was used was that designed by Patrick Martins-Yedenu for his MSc Dissertation in 2016 [16]. He prepared a cleaned, filtered dataset consisting of approximately 750 million records, obtained from the original dataset provided by TfL which consisted of approximately 940 million records from the period, 20th October - 14th December 2013. All records were anonymized and no personal information was made available. Each record contains information on the unique identifier of the respective Oyster card used, the age group to which the registered cardholder belonged, the type of Oyster card used for the journey, the date on which the journey took place, the time at which the journey commenced, the identifier of the station at

which the journey commenced, and the identifier of the station at which the journey ended. The dataset prepared by Martins-Yedenu was filtered in such a way that it used only records referring to TfL journeys taking place on weekdays, and used only 22 out of the total 88 fields available in the original dataset. The filtered dataset was housed in a secure SQL database on the Melba server at King's College London.

However, since the purpose of my project was to use the entire TfL dataset, a database consisting of all 940 million records and 88 fields was subsequently used, covering the same period as the dataset used by Martins-Yedenu. This database was prepared by Roger Moore for his BSc Dissertation in 2017 [18] and was designed to be used with various Big Data technologies, including Spark SQL and PySpark, which aim to optimize the efficiency of queries conducted on the dataset. The data was housed in a secure Spark database on the Melba server at King's College London.

As the scope of this project was focused exclusively on passenger journeys taken on the TfL network of railway and underground services, the above datasets were filtered so that only journeys relating to the use of the London Underground, Docklands Light Railway, and London Overground services were included. Furthermore, the TfL data was further filtered to only include seven fields relating to the most pertinent attributes of each passenger journey, as shown below in Table 1.

Variable	**Description**
prestigeid	a unique id for each Oyster card used in the dataset
pptpassengeragekey	the age group of the registered Oyster card user; age groups are categorized by TfL as follows: Child, Youth, 16-17, Adult, Sixty Plus, and Unknown
cardtypekey	the type of Oyster card used; the key types of Oyster cards used are as follows: Retail Oyster Card, Oyster Photocard, Elderly Freedom Pass, Disabled Freedom Pass, and Staff Pass; other types of Oyster cards are also used, but these constitute a small fraction of the entire dataset
daykey	the date on which the respective journey took place; this was subsequently re-coded as one of the seven days of the week
transactiontime	the time at which the journey commenced
stationoffirstentrykey	the tube station at which the journey commenced
stationofexitkey	the tube station at which the journey ended

2) To conclude the stage of preliminary analysis, histograms for the key variables were obtained, including the age category of the cardholder, the type of card used, the day of the week, the time during the day, the 10 most popular stations of entry, and the 10 most popular stations of exit (shown in section 5.1).

4.3. Passenger Profiles

Following the Preliminary Analysis outlined above, Passenger Profiles were created for two randomly selected passengers in the TfL dataset. These profiles provided detailed information about a passenger's most frequent journeys and the time of day during which these journeys took

place. Such information provided an indication of whether the station of entry most frequently used by a passenger could be approximated as the station closest to that passenger's place of residence. If the Passenger Profiles revealed travel patterns that seemed to indicate that this was indeed the case, then we could make the assumption that a passenger's most frequently used station of entry could be used to identify the station closest to that person's place of residence, and by extension, approximate their place of residence.

Doing so could provide us with a useful information about an individual's most likely socio-demographic background. As indicated in [5], one's place of residence is an important indicator of one's most likely socio-demographic characteristics. There is empirical evidence that shows that humans tend to cluster together in communities in which they share similar socio-demographic characteristics, such as ethnicity, educational achievement, and employment sector, among other factors [12]. Thus, based on this empirical evidence, we make the assumption that TfL passengers who reside in the same location are likely to share certain socio-demographic characteristics. The London Output Area Classification (LOAC) dataset [14], with which we merge the TfL dataset (as discussed in section 4.4) comprises socio-demographic data on various geographic locations around Greater London. These locations are referred to as *output areas*, and there are approximately 25,000 output areas in Greater London, all of which are included in the LOAC dataset. Each output area is assigned a specific label, called a *Supergroup*, which features socio-demographic characteristics in various key categories.

Figure 3 below provides a visual overview of the 25,000 London output areas mapped onto Greater London. Each output area differs in size and shape and was determined based on information from the 2011 Census [14]. The dark areas in the map below indicate groups of particularly small output areas. Since it is difficult to see the individual output areas in Figure 3, a close-up of the Strand area in central London has been provided in Figure 4 below.

4.4. Cluster Analysis

The next step in the approach was to obtain a dataframe of all the unique passengers in the TfL dataset so that we can conduct cluster analysis on data points representing each passenger. Doing so would enable us to allocate a Supergroup code to each passenger based on their most frequently used station of entry and thereby obtain each passenger's most likely socio-demographic characteristics.

Initially, the intention was to create a dataframe of all the unique passengers in the TfL dataset. However, when attempting to do so, the Spark database produced errors related to an insufficient amount of memory[1]. As such, it was deemed necessary to obtain a sample dataset from the original TfL dataset.

This sample dataset was obtained by getting a random sample of records for each of the 55 days in the entire dataset, based on a calculated weight for each day. This weight was based on the number of trips occurring on the respective day as a proportion of the entire dataset and calculated in such a way that the number of records in each day in the sample dataset was proportional to the number of records in each day in the entire dataset. For example, if the journeys on a particular day constituted 2.3 per cent of the entire dataset, that particular day carried a weight of 2.3 per cent in the sample dataset. Thus, the sample dataset can be considered representative of

[1] An example of the error messages received when using the Spark database can be found in Appendix 4.

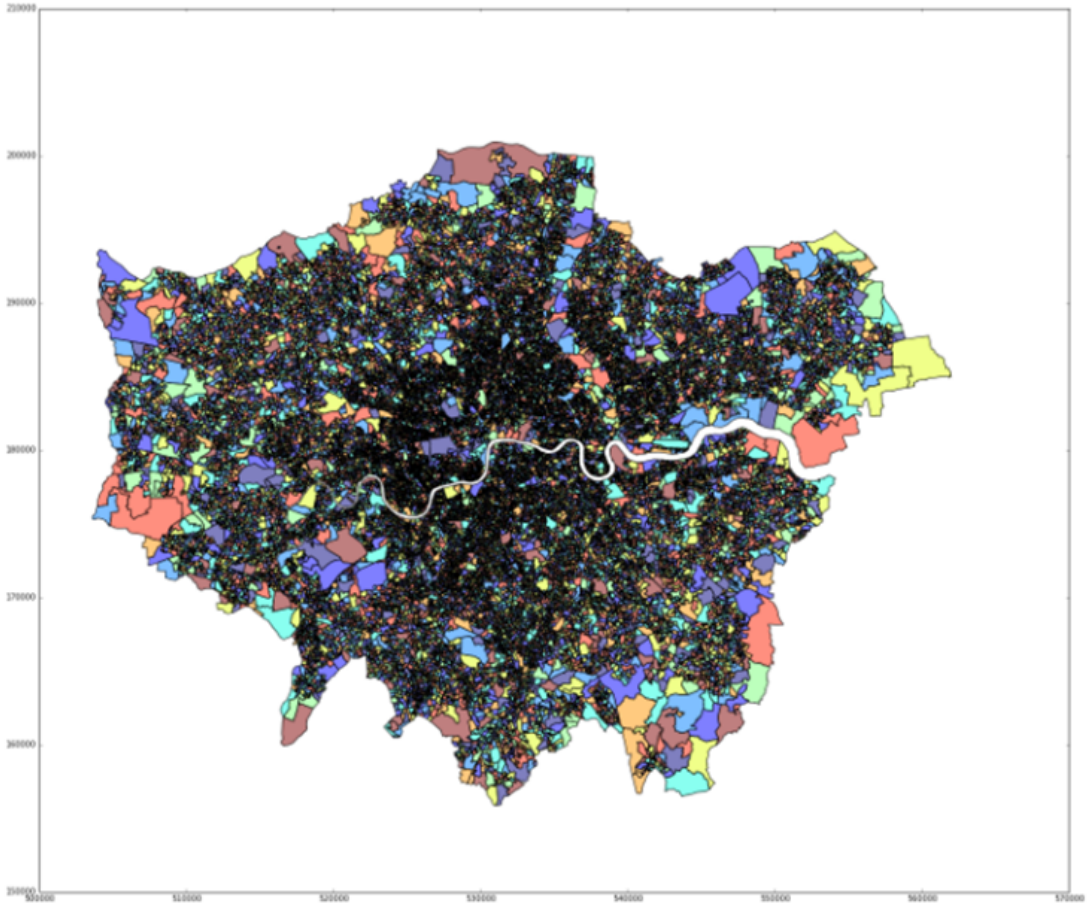


Figure 3: *Map of Output Areas in Greater London*

the entire dataset because it was randomly sampled such that each day in the entire dataset was proportionally represented.

The resulting size of the sample was 4,092,067 records, which constituted approximately 2 per cent of the entire dataset (after filtering out bus journeys, tram journeys, and any other journeys not taken on the TfL network of rail and subway lines, as mentioned in section 4.2 above. The passenger journeys that were used will herewith be referred to as journeys taken on the "tube").

In the meantime, there was a need to create a dataframe of all the unique TfL tube stations so that each tube station could be matched up with its respective Supergroup code. Using the Python geocoding libraries, *geopy*, *shapely*, and others, coupled with the shapefiles that accompanied the LOAC dataset, we were able to match up each unique TfL tube station with its respective Supergroup code based on the geographic coordinates of the station .

After obtaining the sample dataset, a dataframe of the unique passengers with their respective socio-demographic characteristics was created. This was carried out in three stages, as follows:

1. a dataframe was created featuring unique passengers and their most frequently used station

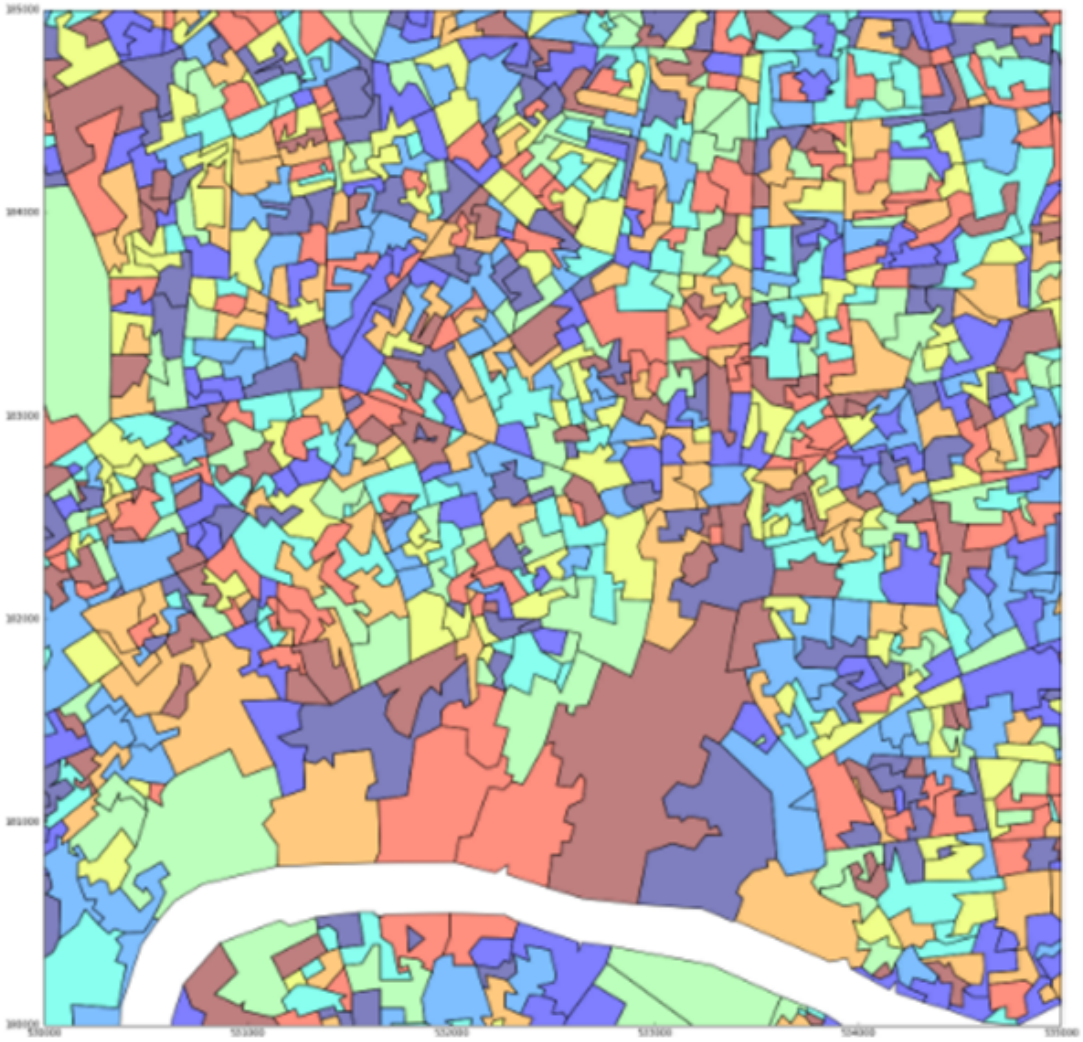


Figure 4: *Close-up of Output Areas in and around the Strand (Central London)*

of entry

2. a dataframe was created featuring unique TfL stations and their respective Supergroup code based on the station's geographic coordinates

3. a final dataframe was created which merged the above two dataframes, resulting in a dataframe of unique passengers and their respective Supergroup codes

This resulting dataframe was used to provide an overview of the distribution of Supergroups within the dataframe of unique passengers.

The above passenger dataframe was then merged with a dataframe containing socio-demographic characteristics of each Supergroup, producing a dataframe of unique passengers with their

respective Supergroup code and socio-demographic characteristics. This resulting dataframe of unique passengers and their respective socio-demographic characteristics was used to conduct cluster analysis and group passengers based on their shared characteristics.

As described above in section 4.1, the k-means clustering algorithm was chosen as the most appropriate method to conduct cluster analysis, and as such, the *scikit-learn* library in Python, which features k-means clustering, was used to carry this out. Each passenger and his associated socio-economic characteristics was considered a separate observation. As such, the cluster analysis was conducted on a dataset containing all unique passengers in the sample dataset and their respective socio-economic characteristics.

As there are 55 different features in the LOAC dataset, each referring to a specific socio-demographic attribute, there was a need to decide which features (and which combination of features) should be used in the cluster analysis.

As such, the key features to be used were grouped into the following categories:

- features related to age
- features related to marital status
- features related to ethnicity
- features related to employment sector
- features related to education levels

In order to evaluate the various clusterings and to determine the most appropriate number of clusters for each set of related features, the Inertia and Calinski-Harabasz scores were used. As described in section 2.2.4 above, these scores provide an indicator that can be used to gauge the best number of clusters for each set of features. The number of clusters evaluated are as follows: 2, 3, 4, 5, 6, 7, 8, 9, 10, 20, 30, 40, 50, and 100 clusters.

4.5. Cluster Travel Patterns

Once the clustering has been carried out, we can determine the cluster to which each passenger belongs. Subsequently, we can use this information to determine the travel patterns of each cluster as a whole. To do so, we merged the dataframe containing all unique passengers and their respective cluster label with the dataframe containing the sample dataset of 4 million tube journeys. This provided us with a new dataframe containing the 4 million records, their respective journey information (including the stations of entry and exit, the day, and the time of that journey) and the associated cluster label corresponding to the passenger that undertook that journey.

This resulting dataframe was then analyzed with regard to travel patterns. The key pieces of information obtained for each Supergroup are as follows: i) the five most frequently used stations of entry, ii) a distribution of the frequency of travel throughout the course of the day, and iii) a distribution of the frequency of travel throughout the course of the week.

5. Results and Analysis

5.1. Preliminary Analysis

The purpose of this section is to provide an overview of the dataset and a preliminary understanding of TfL passengers' travel patterns. The key variables that were used in the analysis for this project include the registered age of the Oyster cardholder, the type of Oyster card they were using for the respective journey, the date and time of the journey, the station at which the journey started, and the station at which the journey ended. Accordingly, histograms of the distribution of each variable are shown below.

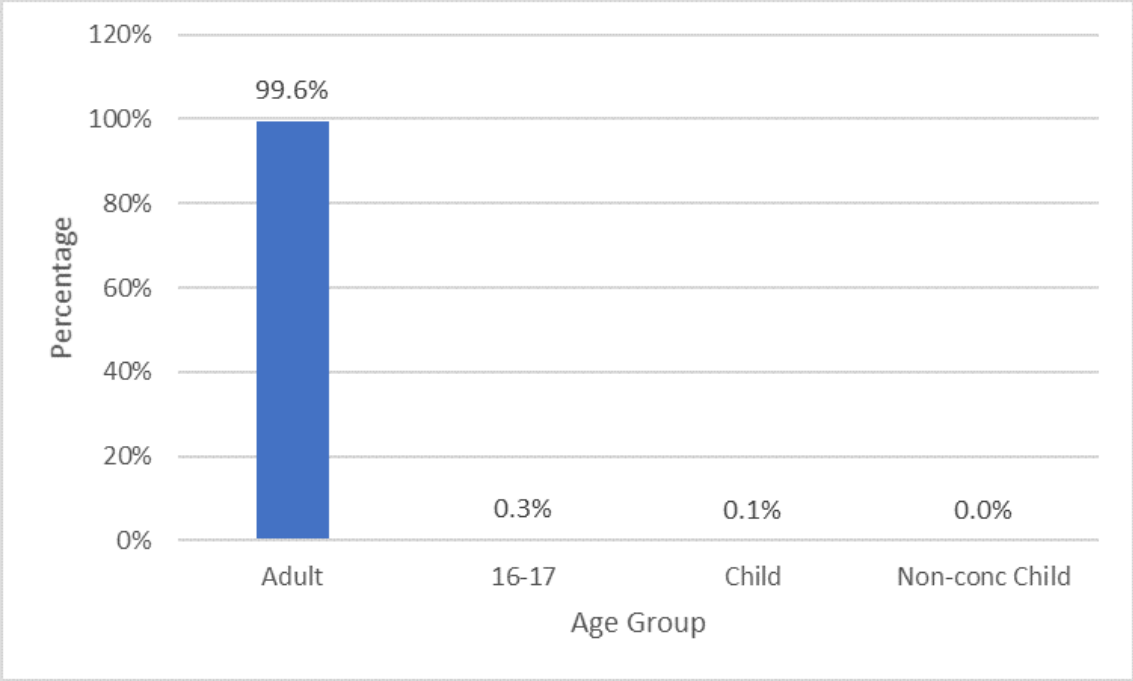


Figure 5: *Distribution of Age Group*

As shown in Figure 5 above, the vast majority of TfL passengers (99.6 per cent) fall into the adult age group, while only a small minority (0.3 per cent) belong to the 16-17 age group. These age categorizations were provided in the TfL dataset and are associated with the registered user of the respective Oyster card.

Figure 6 above shows the distribution of Oyster card type. As can be seen, the most popular card type used is the Retail Oyster card, used by approximately two-thirds of all TfL users. The Oyster Photocard is much less popular, used by only 14 per cent of TfL users.

Figure 7 below indicates that the frequency of travel throughout the workweek is approximately uniformly distributed across all weekdays, with Fridays being marginally more popular than other days.

Figure 8 provides a distribution of the ten most popular stations of entry. It is no surprise that

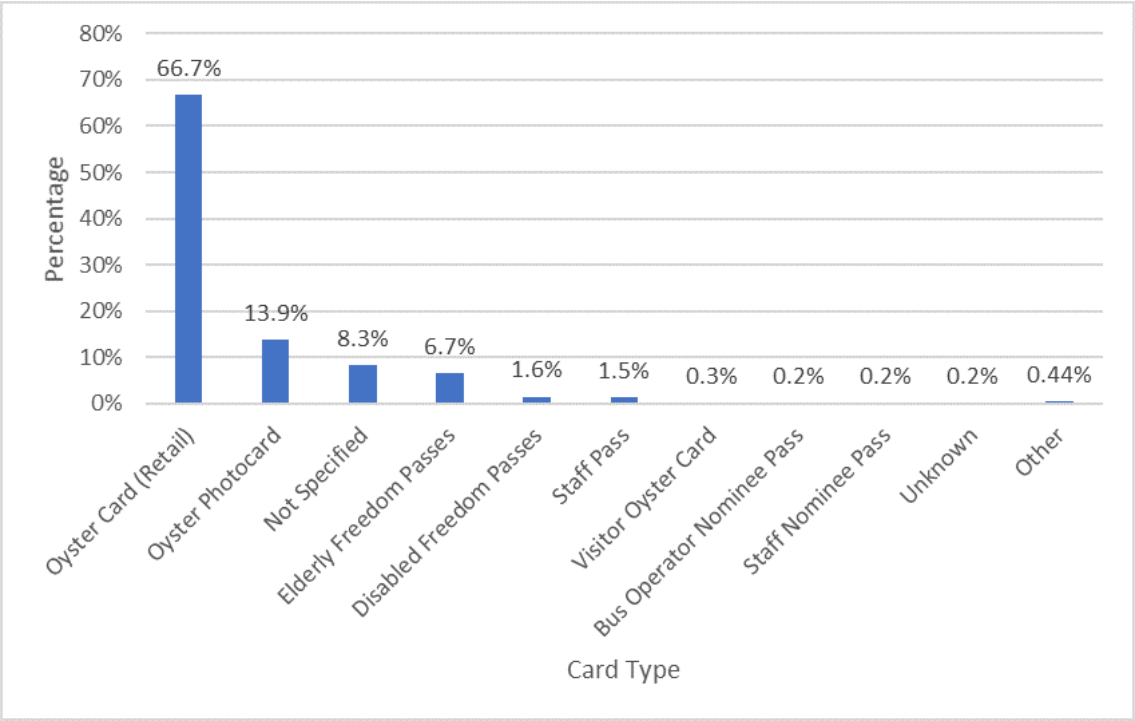


Figure 6: *Distribution of Card Type*

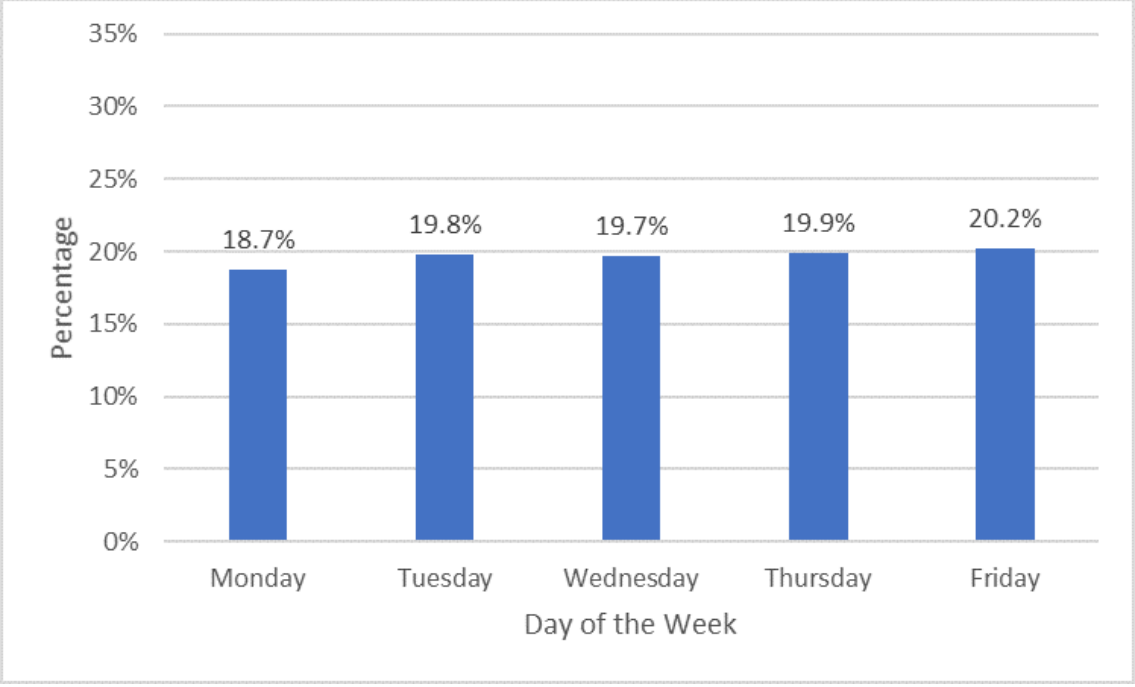


Figure 7: *Distribution of Day of the Week*

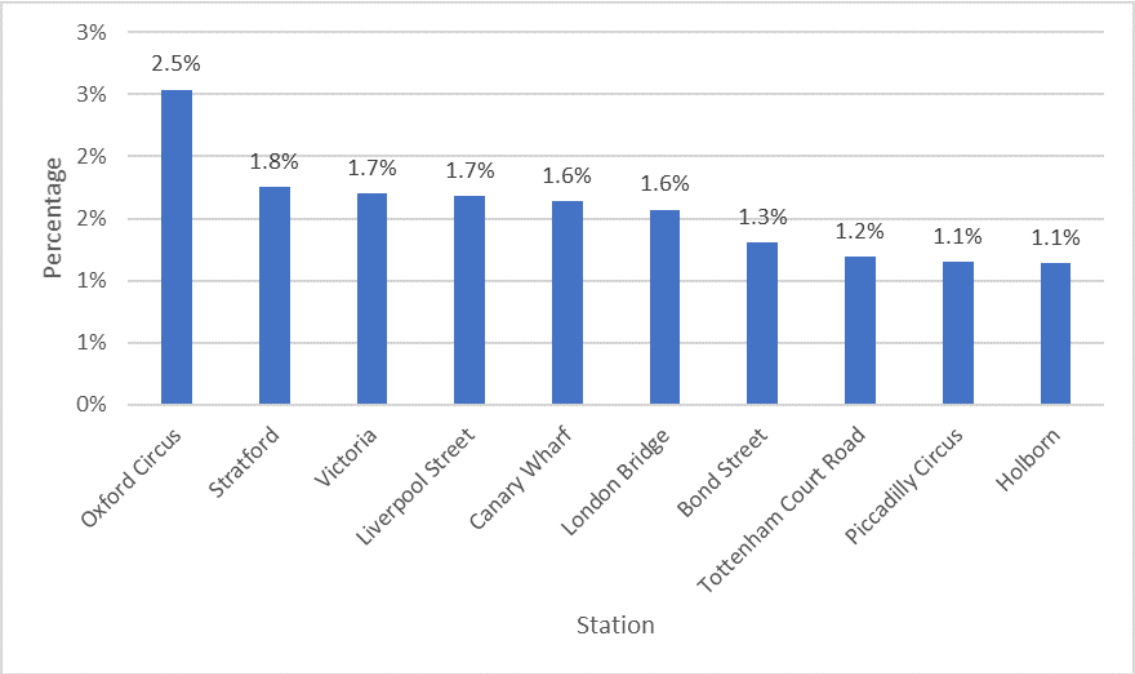


Figure 8: *Distribution of the 10 Most Popular Stations of Entry*

Oxford Circus is the most frequently used station of entry, considering that it is located among one of the most popular shopping districts in London and that it serves over 250,000 passengers on a daily basis [31].

Likewise, as shown in Figure 9, Oxford Circus is also the most popular station of exit. One may notice that the top ten stations of exit are different from the top ten stations of entry. One reason for this is that there are fewer Oyster card swipes that are registered when exiting tube stations compared to swipes registered when entering tube stations. In fact, based on the dataset provided by TfL there were significantly more Oyster card swipes registered when entering all tube stations compared to those that were registered when exiting the same tube stations.

Figure 10 below shows the frequency of travel throughout the day for all passengers in the dataset, with each bar in the histogram representing a two-hour interval in a 24-hour day. As one might expect, the most popular 2-hour intervals for travel are 8:00 am - 10:00 am and 4:00 pm - 6:00 pm.

As discussed above, the Preliminary Analysis section provides an overview of the dataset and aims to give a basic understanding of TfL passengers' travel patterns.

5.2. Passenger Profiles

This section provides examples of how the TfL data can be used to obtain information about a passenger's travel patterns. In particular, information provided in this section helps address research question (i), which refers to uncovering information about a passenger's place of residence based on their travel patterns.

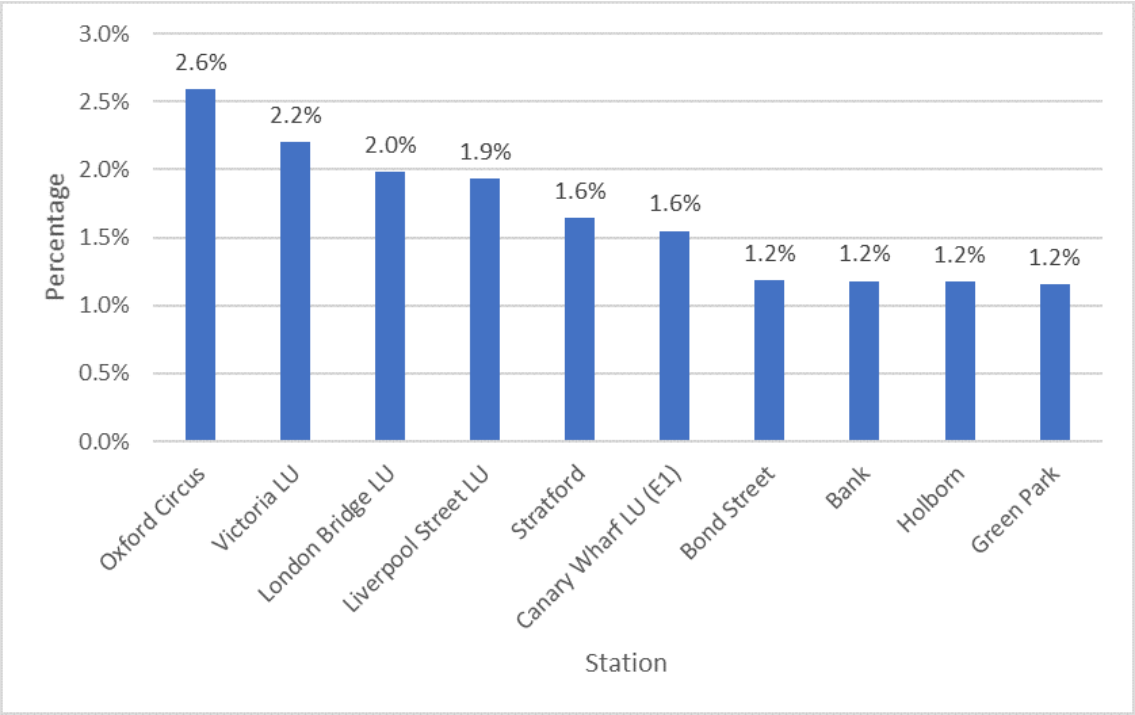


Figure 9: *Distribution of the 10 Most Popular Stations of Exit*

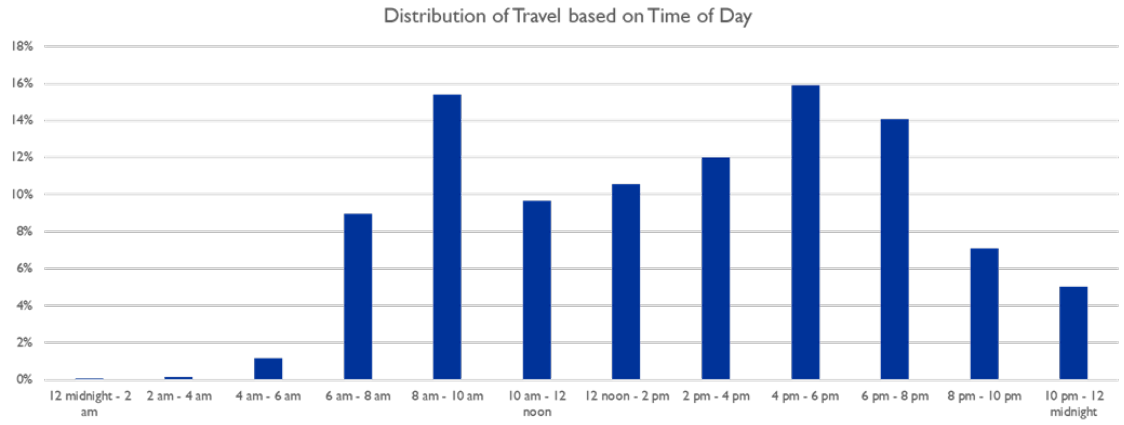


Figure 10: *Frequency of Travel throughout the Day*

In order to address this research question, two passengers were randomly selected from the dataset and their travel patterns were examined to provide an indication of how data in the TfL dataset can be used to approximate a passenger's place of residence.

Passenger 1:

Figure 11 below shows Passenger 1's five most frequently used stations of entry. A station of entry is defined as the station at which a tube journey commences. As indicated in the chart, Stratford

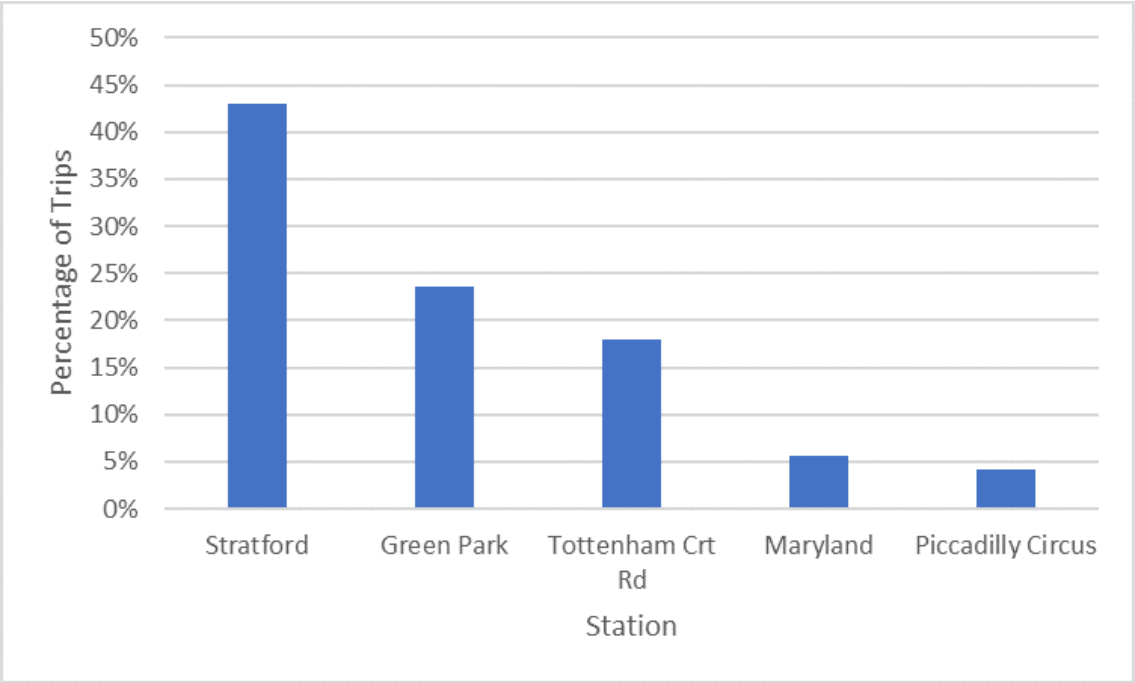


Figure 11: *Passenger 1's Most Frequently Used Stations of Entry*

station is the most frequently used station of entry, representing over 40 per cent of all trips undertaken by Passenger 1. This travel behaviour may indicate that Stratford station is the station closest to this passenger's home, since it is reasonable to assume that a) most journeys commence from one's home compared to any other location and b) a tube passenger would use the station closest to their home when traveling from home. However, it is also important to examine the temporal travel behaviour of this passenger when using this station, as this may further support the idea that the station that is most frequently used by a passenger is indeed the station closest to their home.

Figure 12 below shows Passenger 1's temporal travel patterns when commencing a tube journey from her most frequently used station of entry, Stratford station. As indicated in the chart, this passenger clearly uses Stratford station predominantly in the early morning, specifically between 4 am - 6 am. This is an indication that this passenger uses this station particularly when commuting to work. This further supports the idea that the station that is most frequently used by a passenger is an indication of the station that is closest to their home.

Passenger 2:

Figure 13 shows Passenger 2's five most frequently used stations of entry. As indicated in the chart, Sloane Square station is the most frequently used station of entry, representing over 90 per cent of all trips undertaken by Passenger 2. As mentioned above, this travel behaviour may indicate that Sloane Square station is the station closest to this passenger's home, since it is reasonable to assume that a) most journeys commence from one's home compared to any other location and b) a tube passenger would use the station closest to their home when traveling from home. However, as before, it is also important to examine the temporal travel behaviour of this passenger when

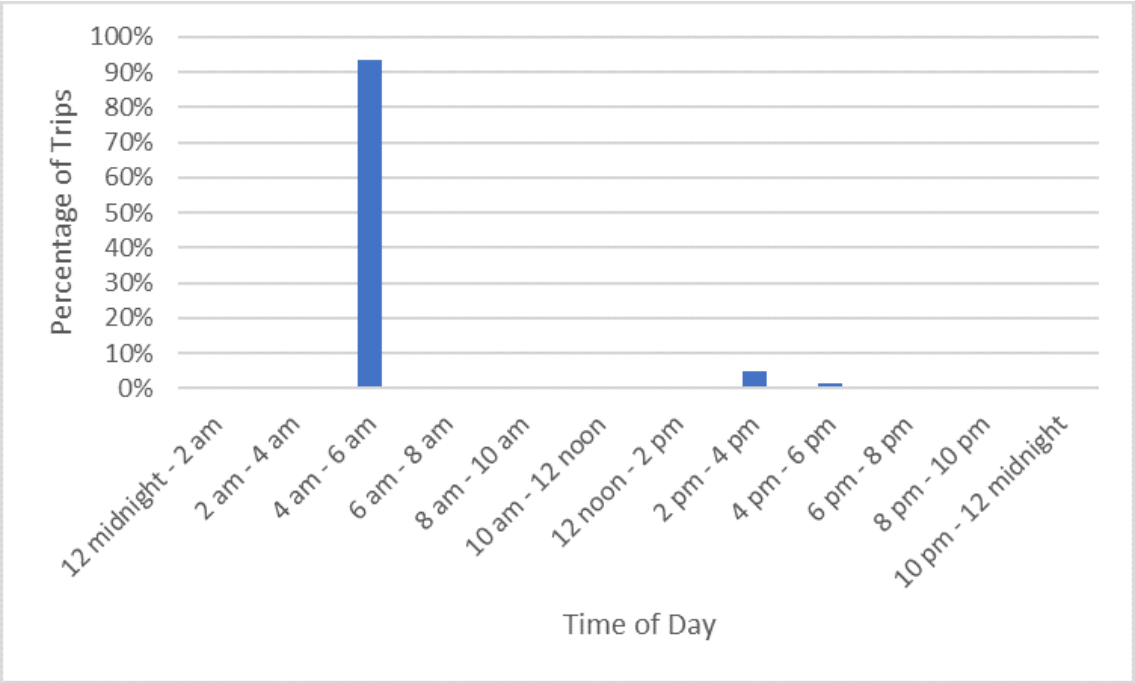


Figure 12: *Passenger 1's Temporal Travel Patterns when using Most Frequently Used Station of Entry*

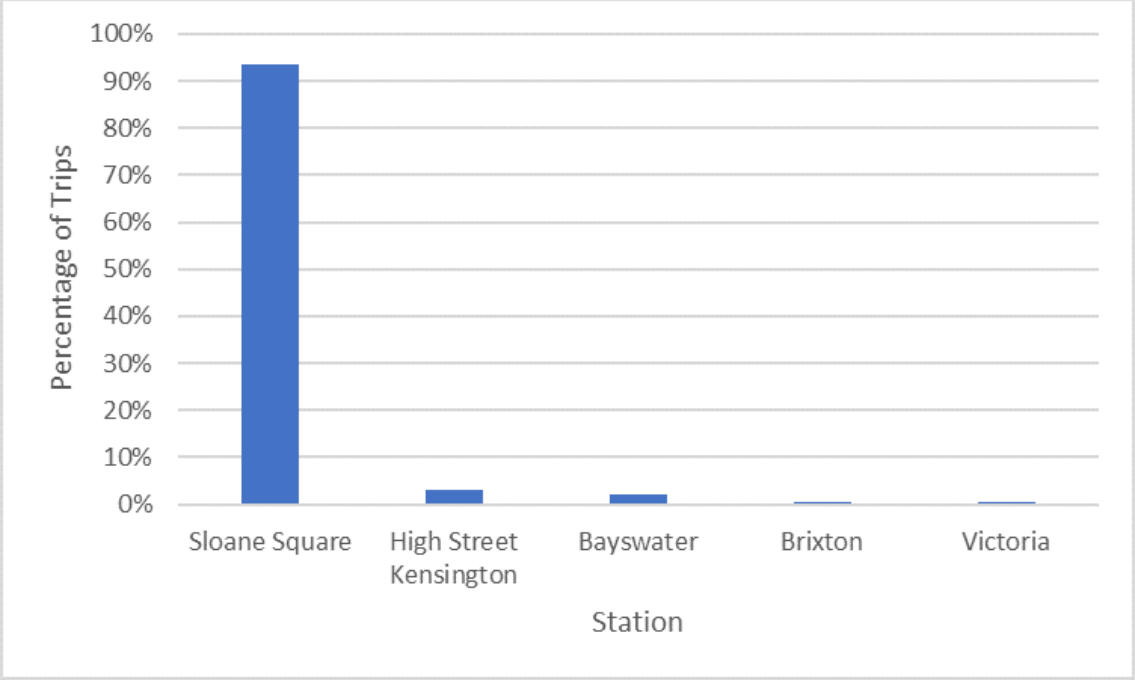


Figure 13: *Passenger 2's Most Frequently Used Stations of Entry*

using this station, as this may further support the idea that the station that is most frequently used

by a passenger is indeed the station closest to their home.

Figure 14 below shows Passenger 2's temporal travel patterns when commencing a tube journey from his most frequently used station of entry, Sloane Square station. As indicated in the chart, this passenger uses this station throughout the day, suggesting that it is a station that is readily accessible to this passenger, a possible indication that it is located in close proximity to his home. Furthermore, this passenger uses Sloane Square station most frequently between 8 am - 10 am. Similar to Passenger 1 above, this is an indication this this passenger uses this station most often when commuting to work, further supporting the idea that this station is the station that is closest to this passenger's home.

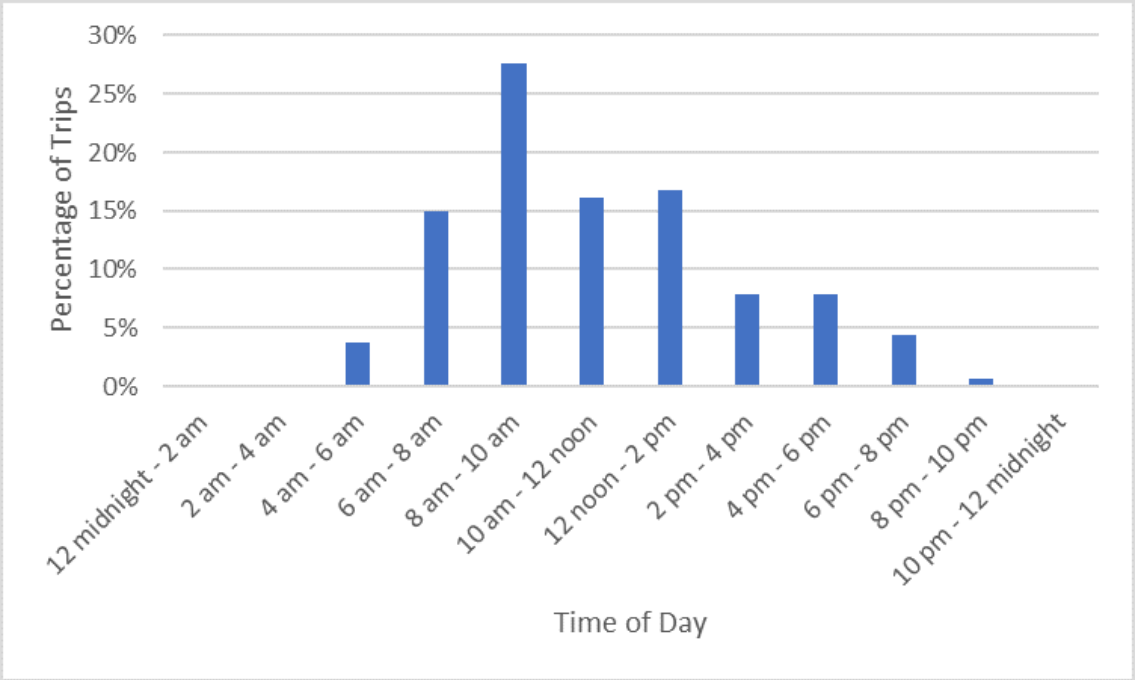


Figure 14: *Passenger 2's Temporal Travel Patterns when using Most Frequently Used Station of Entry*

Based on the information provided above for Passenger 1 and Passenger 2, I make the assumption that a passenger's most frequently used station of entry can be used to approximate the location of the passenger's place of residence. This assumption leads to the next section, which conducts cluster analysis on a dataset consisting of unique passengers and their associated socio-demographic characteristics based on the passenger's approximated place of residence.

5.3. Cluster Analysis

The purpose of this section is to carry out k-means clustering on a dataset of unique passengers and their socio-demographic characteristics. Doing so helps address research question (ii), which refers to grouping TfL passengers based on their socio-demographic characteristics.

As discussed in section 4.4, the socio-demographic features obtained from the LOAC dataset were grouped into related sets of features and subsequently clustered using the k-means algorithm

using the Python library, scikit-learn.

The Inertia and Calinski-Harabasz scores were used to evaluate the performance of each clustering. As mentioned in section 2.2.4, the Inertia score provides a measure of the distance between each data point and its corresponding cluster centroid, indicating how densely packed each cluster is, while the Calinski-Harabasz score provides a measure of how densely packed the clusters are as well as how distinct each cluster is compared to all other clusters.

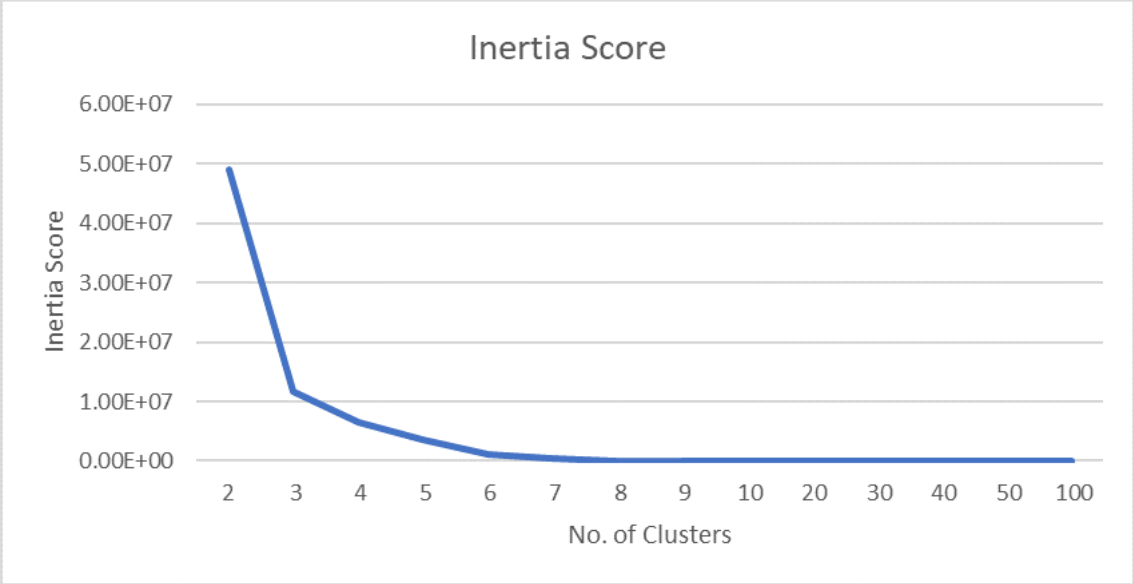


Figure 15: *Inertia Evaluation Scores for K-Means Clustering on Features Related to Employment Sector*

As discussed in section 2.2.4, the Inertia measure was used to determine the most appropriate number of clusters by identifying the value at which the score approached zero. On the other hand, the Calinski-Harabasz measure was used to determine the most appropriate number of clusters by identifying the value at which the score reaches its maximum value. To help determine the ideal number of clusters for each set of related features, visualizations of each score was used. A sample visualization of the Inertia scores and the Calinski-Harabasz scores for features related to employment sector are shown in Figures 15 and 16. One may note that the Inertia score appears to approach zero at approximately eight clusters; this is supported by the observation that the Calinski-Harabasz score reaches its optimal value at eight clusters. The same behaviour can be observed in the Inertia scores and Calinski-Harabasz scores for all other sets of features, as shown in Appendix 1. As such, it was determined that the ideal number of clusters for each set of related features is eight clusters.

After determining the most appropriate number of clusters for each set of features, the next and final step of the cluster analysis is to determine the most representative characteristics of each cluster. In order to do so, the characteristics of the cluster centroid of each cluster were obtained by computing the mean value of each feature for each cluster. A table of all centroid values are provided in Appendix 1. However, the key statistics for each cluster are discussed below.

Cluster 1:

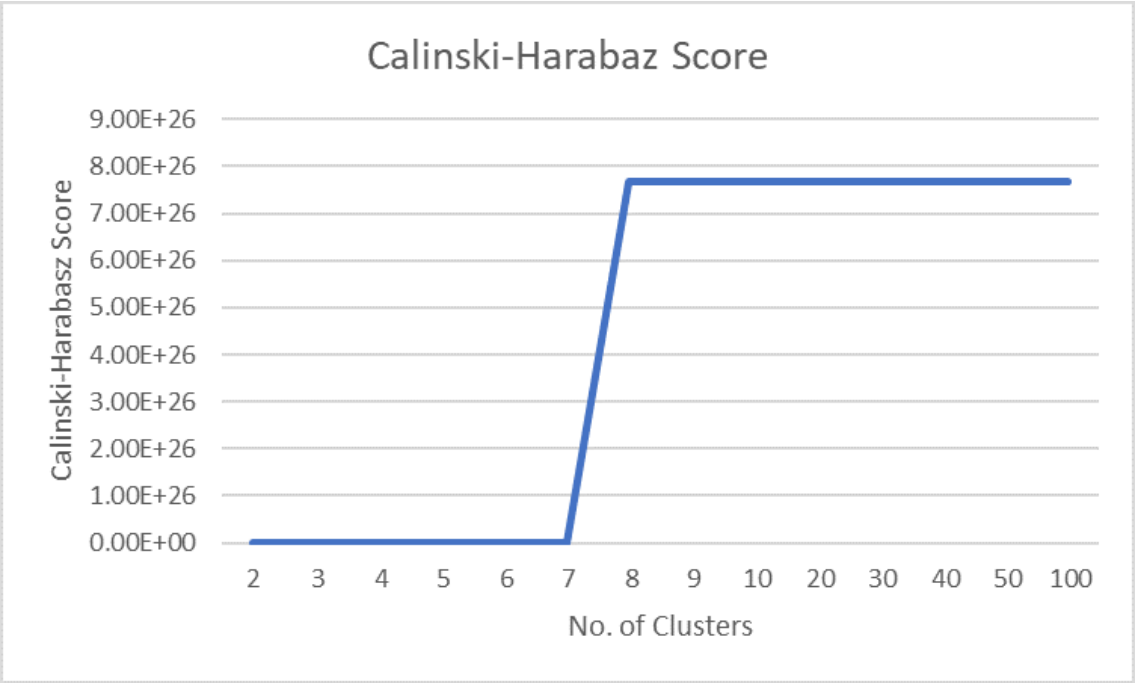


Figure 16: *Calinski-Harabasz Evaluation Scores for K-Means Clustering on Features Related to Employment Sector*

A key characteristic of Cluster 1 is that it includes passengers who are very likely to identify as White (9.3 per cent above the London average). They are also very likely to have an educational qualification that does not exceed Level 1, Level 2[2] , or an apprenticeship (9.3 per cent above the London average) and are unlikely to work in the IT or scientific sectors (7.4 per cent below the London average). This cluster also features passengers who are unlikely to be in the age group 25 to 44 (6.9 per cent below the London average) and are unlikely to be single (5.1 per cent below the London average). As such, Cluster 1 features passengers who may share one or more of the following characteristics - identifying as White, not single, not particularly young, not employed in a highly-skilled technical field, and not as highly educated as some of their fellow passengers.

Cluster 2:

One of the key characteristics of Cluster 2 is that it includes passengers who are very likely to identify as Black (13.4 per cent above the London average). They are also very unlikely to be married or in a civil partnership (11 per cent below the London average) and unlikely to have an educational qualification at Level 4 or above[3] (9.4 per cent below the London average). Furthermore, it is likely that passengers in this cluster work in the accommodation or food service sectors (4.2 per cent above the London average). There is no clear indication of their age group, suggesting that passengers in this cluster may come from a variety of different age groups. As such, Cluster 2 features passengers who may share one or more of the following characteristics - identifying as Black, not being married, employed in the accommodation or food service sectors, and not as highly educated as some of their fellow passengers.

[2]Level 1 refers to qualifications that include GCSEs and diplomas (City and Guilds, BTEC); Level 2 refers to qualifications that include GCSEs and O Levels.

[3]Level 4 or above refers to qualifications that include Higher National Certificates and degrees.

Cluster 3:

A key characteristic of Cluster 3 is that it includes passengers who are likely to identify as having Indian ethnicity (16.2 per cent above the London average) or Chinese ethnicity (5.6 per cent above the London average). They are also very likely to be married or in a civil partnership (11.5 per cent above the London average). Passengers in this cluster are rather unlikely to have an educational qualification at Level 4 or above (6.5 per cent below the London average). Furthermore, they are likely to be employed in the wholesale or retail trade sectors (4.6 per cent above the London average). Lastly, they are quite unlikely to be in the age group 25 to 44 years (4.1 per cent below the London average). As such, Cluster 3 features passengers who may share one or more of the following characteristics - identifying as having Indian ethnicity, being married, not particularly young, employed in the wholesale or retail trade sectors, and not as highly educated as some of their fellow passengers.

Cluster 4:

A notable attribute of Cluster 4 is that it includes passengers who are very likely to have an educational qualification at Level 4 or above (24.6 per cent above the London average). They are also very likely to be in the age group 25 to 44 (18.4 per cent above the London average) and similarly likely to be single (15.6 per cent above the London average). Passengers in Cluster 4 are also highly likely to be employed in the IT or scientific sectors (11.4 per cent above the London average) or the finance, insurance, or real estate sectors (10.4 per cent above the London average). They are also quite likely to identify as White (8.6 per cent above the London average). As such, Cluster 4 features passengers who may share one or more of the following characteristics - being highly educated, working in a technical field, being relatively young and single, and identifying as White.

Cluster 5:

Key identifying characteristics of Cluster 5 are that it includes passengers who are very likely to be single (12 per cent above the London average) and have an educational qualification at Level 4 or above (10.3 per cent above the London average). They are also likely to belong to the age group 25 to 44 years (7.9 per cent above the London average) and work in the IT or scientific sectors (7 per cent above the London average). They are also somewhat likely to identify as White (4.8 per cent above the London average). As such, Cluster 5 features passengers who may share one or more of the following characteristics - being single, relatively young, highly educated, employed in a highly skilled profession, and identifying as White.

Cluster 6:

A key characteristic of Cluster 6 is that it includes passengers who are very likely to identify as White (19.2 per cent above the London average). They are also very likely to have an educational qualification at Level 4 or above (16.9 per cent above the London average) and quite likely to be employed in the IT or scientific sectors (8.3 per cent above the London average). Furthermore, passengers in this cluster are rather likely to be married or in a civil partnership (5.9 per cent above the London average). There is no clear indication of their age group, suggesting that passengers in this cluster may come from a variety of different age groups. As such, Cluster 6 features passengers who may share one or more of the following characteristics - identifying as White, being highly educated, working in a highly skilled profession, and being married.

Cluster 7:

A notable identifying characteristic of Cluster 7 is that it includes passengers who are likely to identify as Black (9.6 per cent above the London average). They are also quite unlikely to have an educational qualification at Level 4 or above . Passengers in this cluster are somewhat likely to work in the accommodation or food service sectors (2.9 per cent above the London average) or the wholesale or retail trade sectors (2.3 per cent above the London average). There are no clear indications of their age group or marital status, suggesting that passengers in this cluster may come from a variety of age groups and may be either single, married, or divorced. As such, Cluster 7 features passengers who may share one or more of the following characteristics - identifying as Black, employed in the accommodation, food service, wholesale, or retail trade sectors, and not as highly educated as some of their fellow passengers.

Cluster 8:

One of the key characteristics of this cluster is that it includes passengers who are very likely to identify as White (21.2 per cent above the London average). They are also very likely to be married or in a civil partnership (17.8 per cent above the London average). Passengers in this cluster are quite likely to have an educational qualification that does not exceed Level 1, Level 2 , or an apprenticeship (8.8 per cent above the London average). Furthermore, they are rather likely to be in the age groups 45 to 64 years (7.8 per cent above the London average) or 65 to 89 years (7.4 per cent above the London average). There is no clear indication of their employment sector, suggesting that passengers in this cluster may be employed in a variety of different professions or may be retired. As such, Cluster 8 features passengers who may share one or more of the following characteristics - identifying as White, being married, and not as young nor as highly educated as some of their fellow passengers.

As indicated in the discussion above, cluster analysis provides us with a way to group the passenger data into clusters in which passengers may share similar characteristics. Based on the Inertia and Calinski-Harabasz scores, we can determine that the ideal clustering for each set of related features is eight clusters. Lastly, we can provide the representative characteristics of each of the eight clusters based on the cluster centroids and their corresponding socio-demographic characteristics.

In order to determine whether there is a relationship between the clusters provided by our k-means algorithm and the Supergroups provided by the LOAC dataset, we can compare the two sets of groupings. Figure 16 below shows the distribution of clusters and the distribution of Supergroups across the passenger dataset. Based on this comparison, it is clear that both groupings follow a similar distribution. In order to further support the idea that the eight clusters correspond closely to the eight Supergroups, we can compare the values of the cluster centroids, as discussed above and provided in full in table A.1 in Appendix 2, with the mean values of each Supergroup, provided in Table A.2 in Appendix 3. As one will notice, they share the same values for each socio-demographic feature. As such, we can assume that the clustering provided by our k-means analysis very closely resembles the categorization of Supergroups provided by the LOAC dataset.

5.4. Cluster Travel Patterns

Since there is a close relationship between the clusters provided by our k-means analysis and the Supergroups provided by the LOAC dataset, we can seek further information on each cluster,

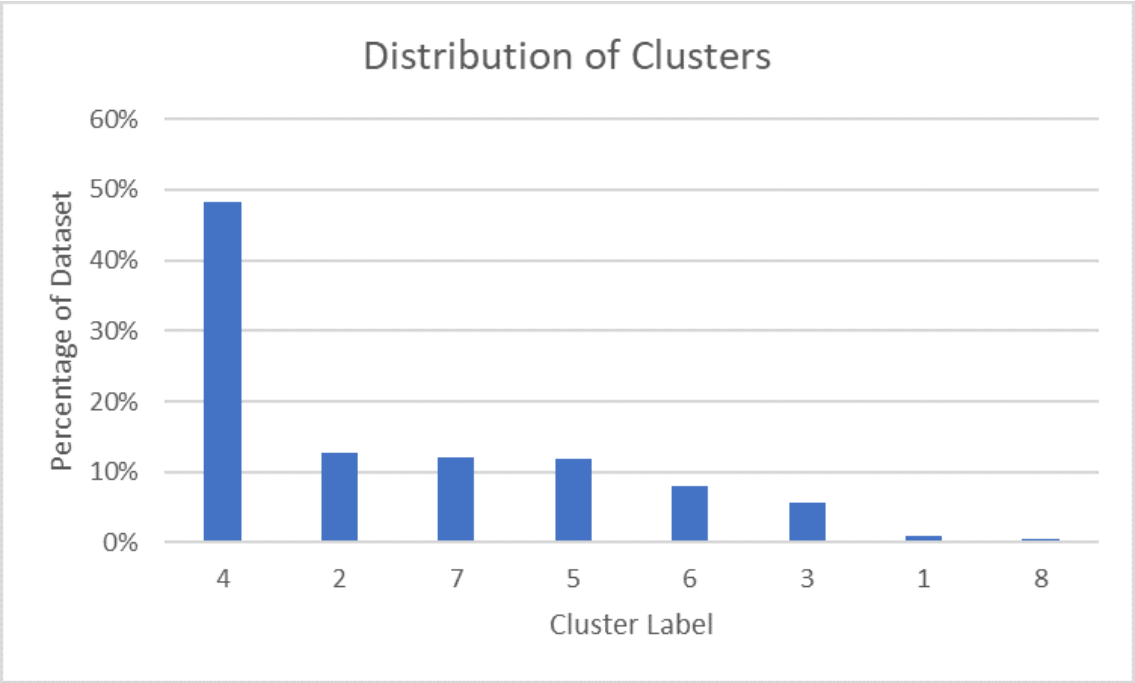


Figure 17: *Distribution of Passengers by Cluster*

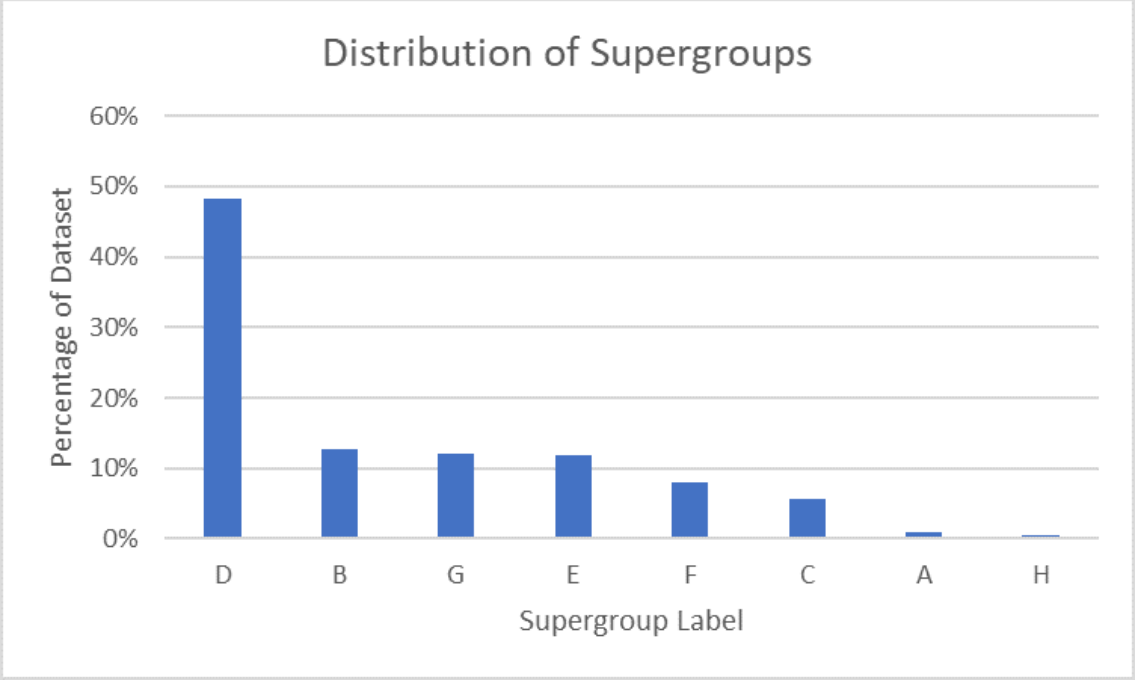


Figure 18: *Distribution of Passengers by Supergroup*

particularly with regard to the key travel patterns of passengers in each cluster. In order to do so,

we assign a Supergroup label to each passenger based on his approximated place of residence, as discussed in section 4.4. Once a passenger is assigned a Supergroup label, we can go back to the original sample dataset of 4 million records and apply the Supergroup label to each record based on the corresponding passenger id. By doing so, we can determine travel patterns for each Supergroup and corresponding cluster, such as the most frequently used stations of entry, and the most popular day of the week and time during the day to travel. Doing so helps address research question (iii), which refers to uncovering information about the travel patterns of different groups of TfL passengers based on their socio-demographic characteristics.

Cluster 1:

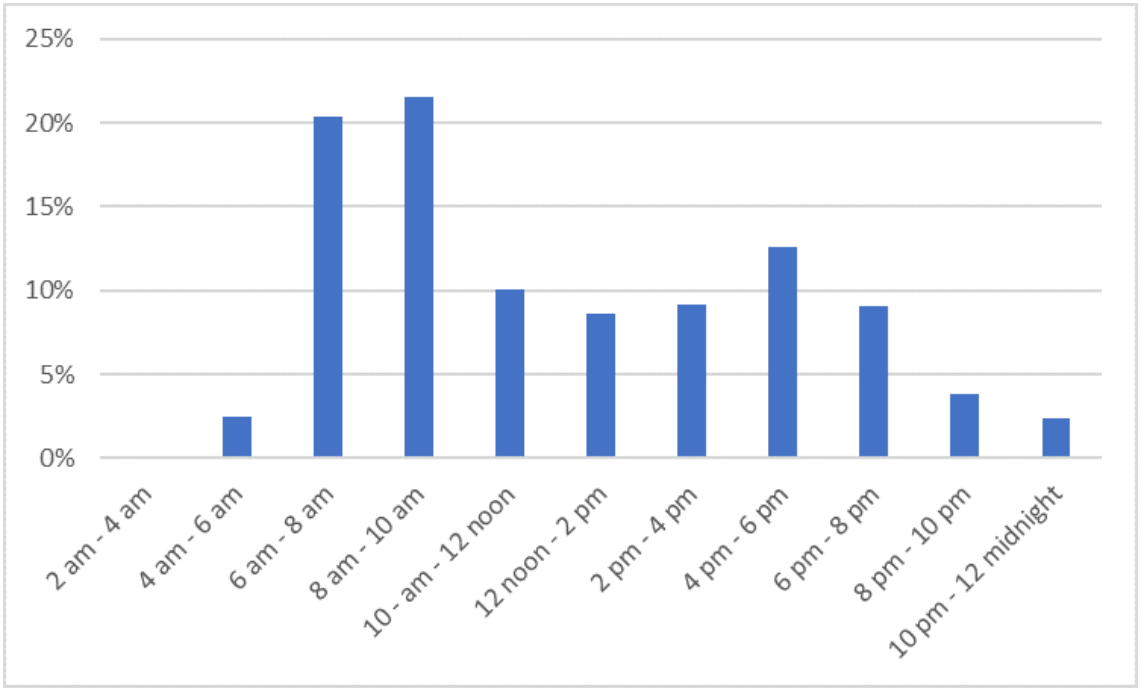


Figure 19: *Temporal Travel Patterns for Cluster 1*

Based on information in the TfL dataset, and after associating each passenger with their corresponding cluster, key travel patterns can be provided for passengers in Cluster 1. As indicated in Figure 19 above, their travel times are distinctly skewed towards the morning between 6 am - 8 am, with approximately 20 per cent of all journeys occurring during this interval and very little travel occurring after 8 pm (approx. 6 per cent of all journeys). As is the case with most other passengers, they travel most frequently on Fridays but have a relatively uniform distribution of travel throughout the work week, with decreased travel on weekends. Their five most frequently used stations of entry are Dagenham Heathway, Becontree, Elm Park, Dagenham East, and Ruislip Gardens.

Cluster 2:

Based on information in the TfL dataset, and after associating each passenger with their corresponding cluster, key travel patterns can be provided for passengers in Cluster 2. As indicated in

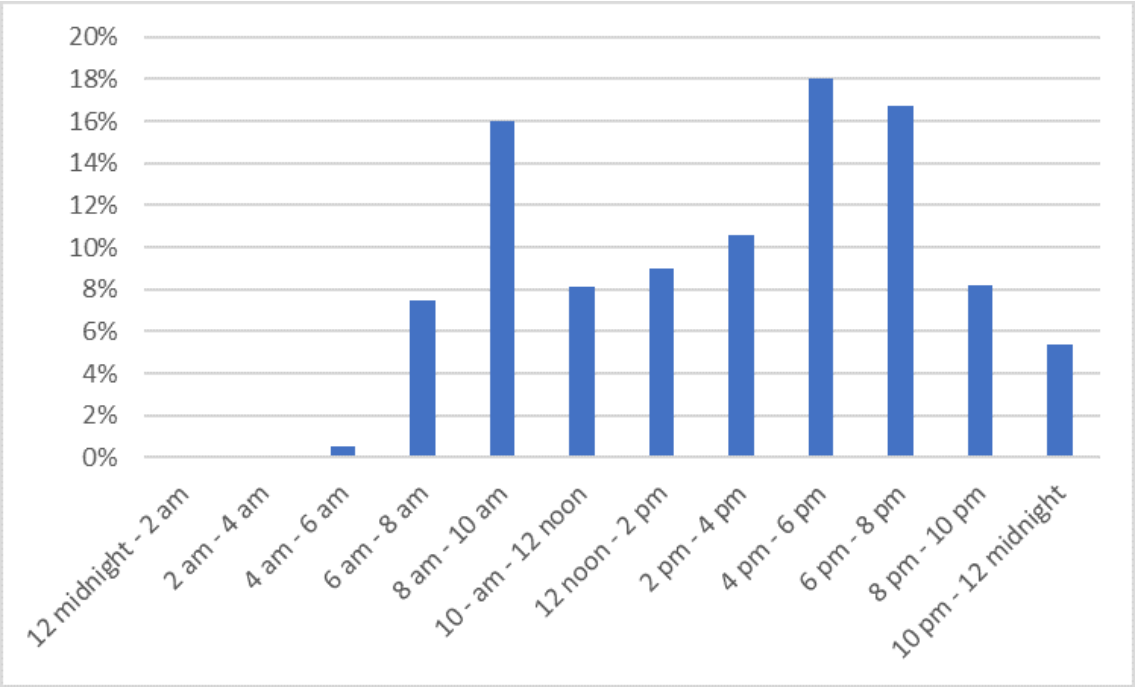


Figure 20: *Temporal Travel Patterns for Cluster 2*

Figure 20, their travel times are distinctly skewed towards the evening, with approximately 30 per cent of all journeys occurring after 6 pm. As is the case with most other passengers, they travel most frequently on Fridays but have a relatively uniform distribution of travel throughout the work week, with decreased travel on weekends. Their five most frequently used stations of entry are Bond Street, Warren Street, Mile End, Barking, and Whitechapel.

Cluster 3:

Based on information in the TfL dataset, and after associating each passenger with their corresponding cluster, key travel patterns can be provided for passengers in Cluster 3. As indicated in Figure 21, their travel patterns are somewhat skewed towards the morning, with approx. 40 per cent of all journeys occurring between 6 am - 10 am and very little travel occurring after 8 pm (approximately 8 per cent). As is the case with most other passengers, they travel most frequently on Fridays but have a relatively uniform distribution of travel throughout the work week, with decreased travel on weekends. Their five most frequently used stations of entry are Woodford, Gants Hill, Newbury Park, Gunnersbury, and Northolt.

Cluster 4:

Based on information in the TfL dataset, and after associating each passenger with their corresponding cluster, key travel patterns can be provided for passengers in Cluster 4. As indicated in Figure 22, their travel patterns are distinctly skewed towards the night, with relatively little travel occurring before noon (only 26 per cent) and, compared to other Supergroups, a relatively high proportion of travel occurring between 8 pm and midnight (approximately 16 per cent of all journeys). As is the case with most other passengers, they travel most frequently on Fridays

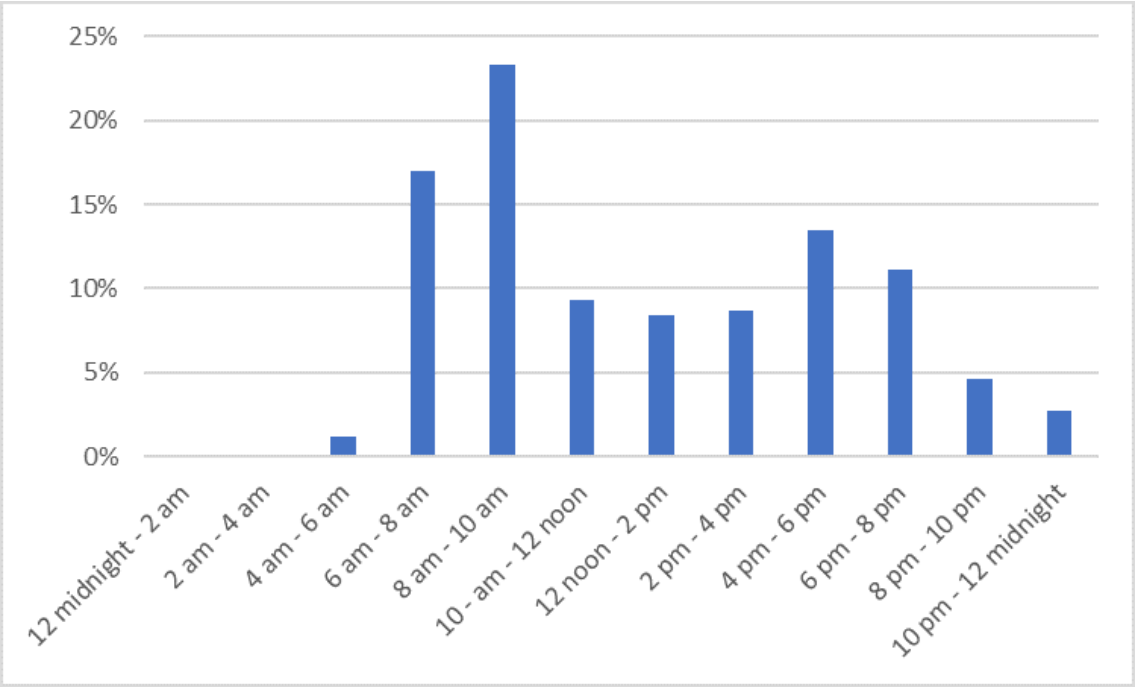


Figure 21: *Temporal Travel Patterns for Cluster 3*

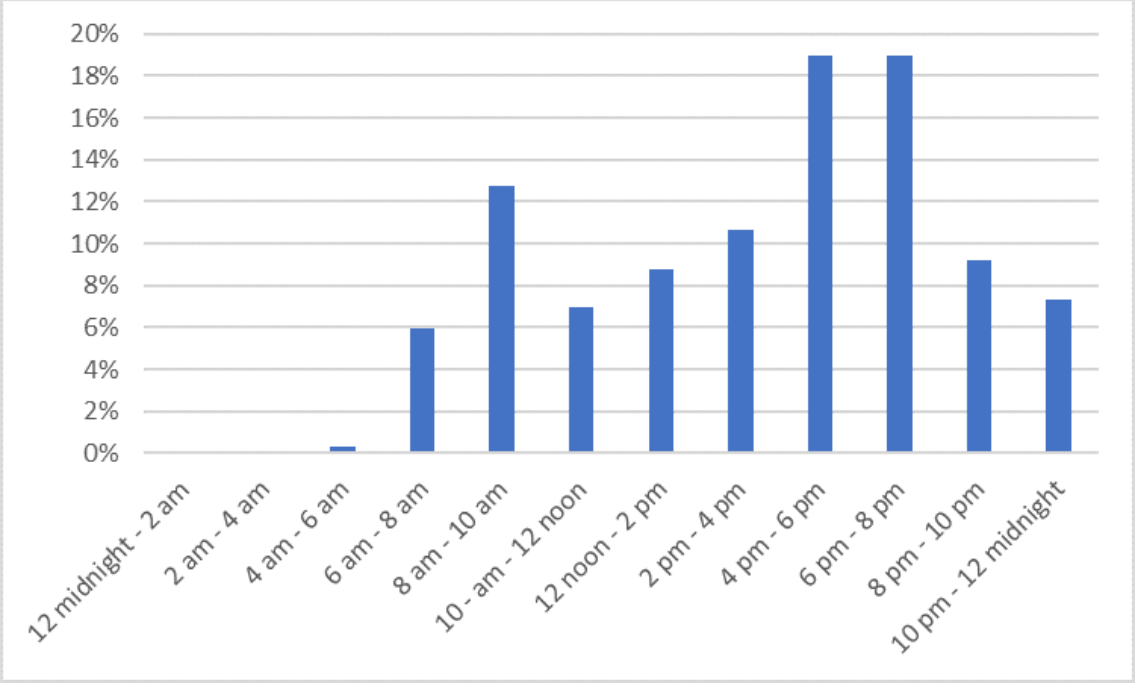


Figure 22: *Temporal Travel Patterns for Cluster 4*

but have a relatively uniform distribution of travel throughout the work week, with decreased travel on weekends. Their five most frequently used stations of entry are Oxford Circus, Stratford, Piccadilly Circus, Tottenham Court Road, and Holborn.

Cluster 5:

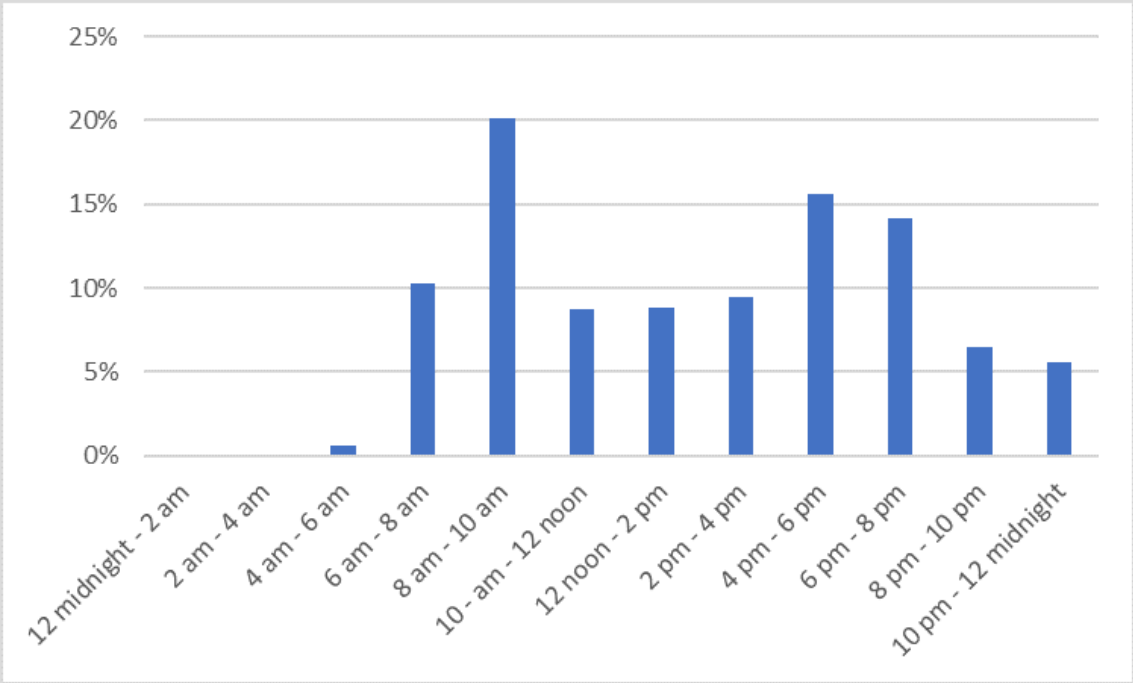


Figure 23: *Temporal Travel Patterns for Cluster 5*

Based on information in the TfL dataset, and after associating each passenger with their corresponding cluster, key travel patterns can be provided for passengers in Cluster 5. As indicated in Figure 23, their travel patterns are relatively regular, with little travel occurring after 8 pm (only 12 per cent). As is the case with most other passengers, they travel most frequently on Fridays but have a relatively uniform distribution of travel throughout the work week, with decreased travel on weekends. Their five most frequently used stations of entry are Ealing Broadway, North Greenwich, Camden Town, Canada Water, and Wood Green.

Cluster 6:

Based on information in the TfL dataset, and after associating each passenger with their corresponding cluster, key travel patterns can be provided for passengers in Cluster 6. As indicated in Figure 24 above, their travel patterns are relatively regular, with little travel occurring after 8 pm (only 12 per cent). As is the case with most other passengers, they travel most frequently on Fridays but have a relatively uniform distribution of travel throughout the work week, with decreased travel on weekends. Their five most frequently used stations of entry are Westminster, Notting Hill Gate, Tower Hill, Barbican, and Barons Court.

Cluster 7:

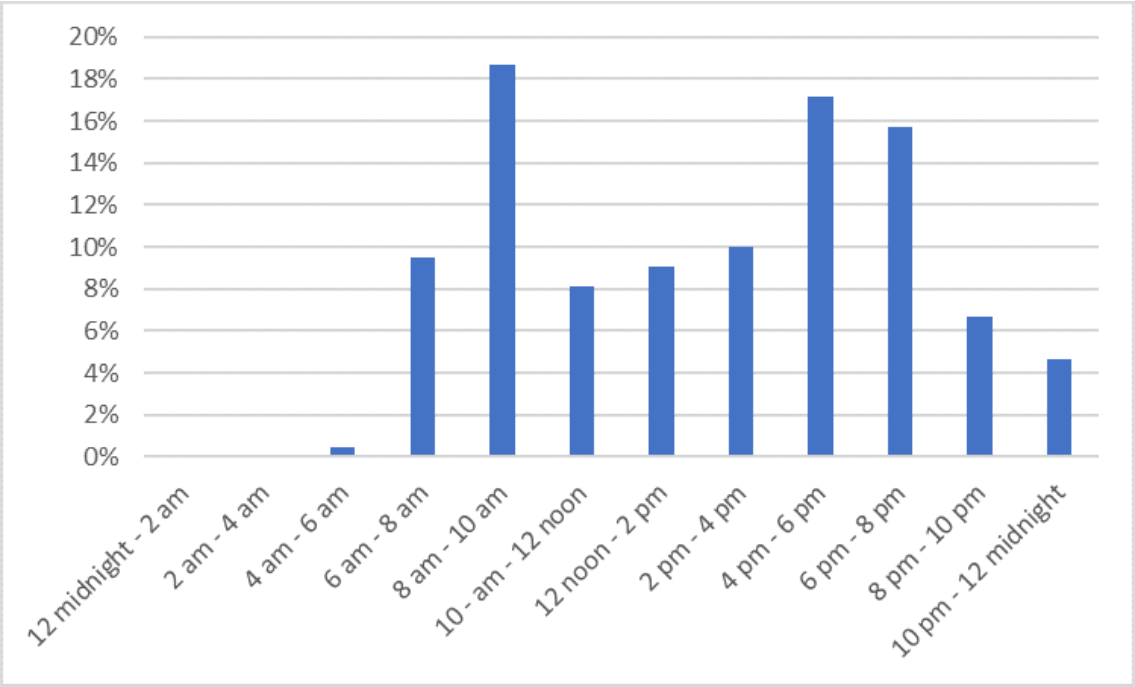


Figure 24: *Temporal Travel Patterns for Cluster 6*

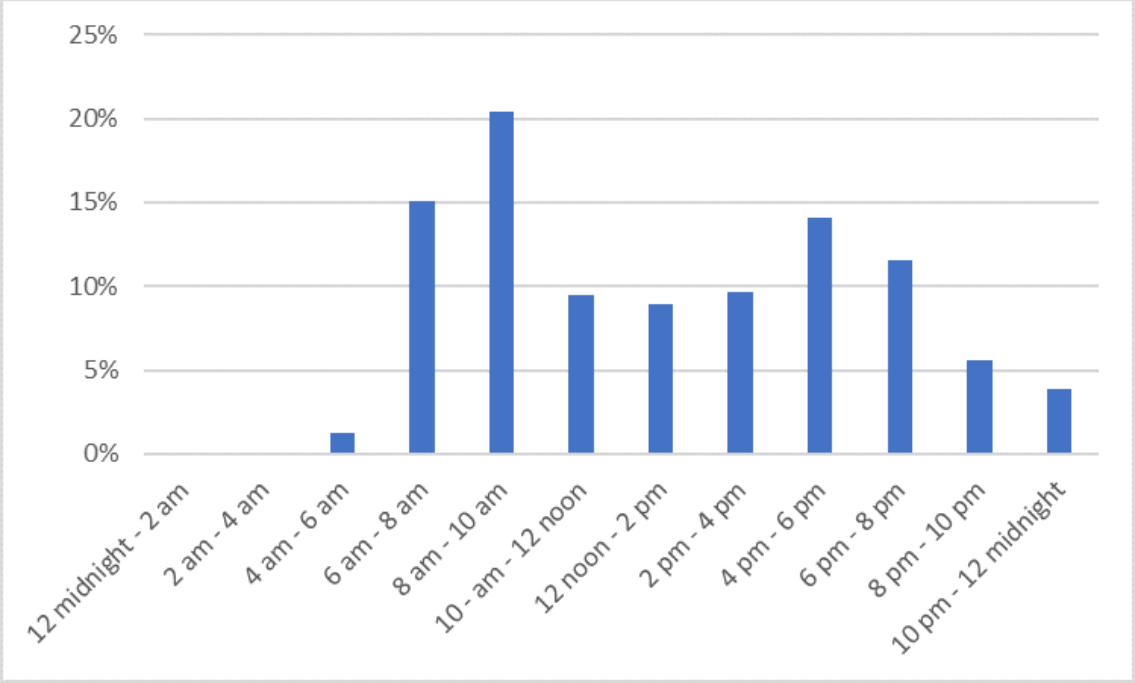


Figure 25: *Temporal Travel Patterns for Cluster 7*

Based on information in the TfL dataset, and after associating each passenger with their corresponding cluster, key travel patterns can be provided for passengers in Cluster 7. As indicated in Figure 25, their travel patterns are somewhat skewed towards the morning, with approx. 35 per cent of all journeys occurring between 6 am - 10 am and very little travel occurring after 8 pm (approximately 10 per cent). As is the case with most other passengers, they travel most frequently on Fridays but have a relatively uniform distribution of travel throughout the work week, with decreased travel on weekends. Their five most frequently used stations of entry are Walthamstow Central, Seven Sisters, East Ham, Wembley Park, and Leyton.

Cluster 8:

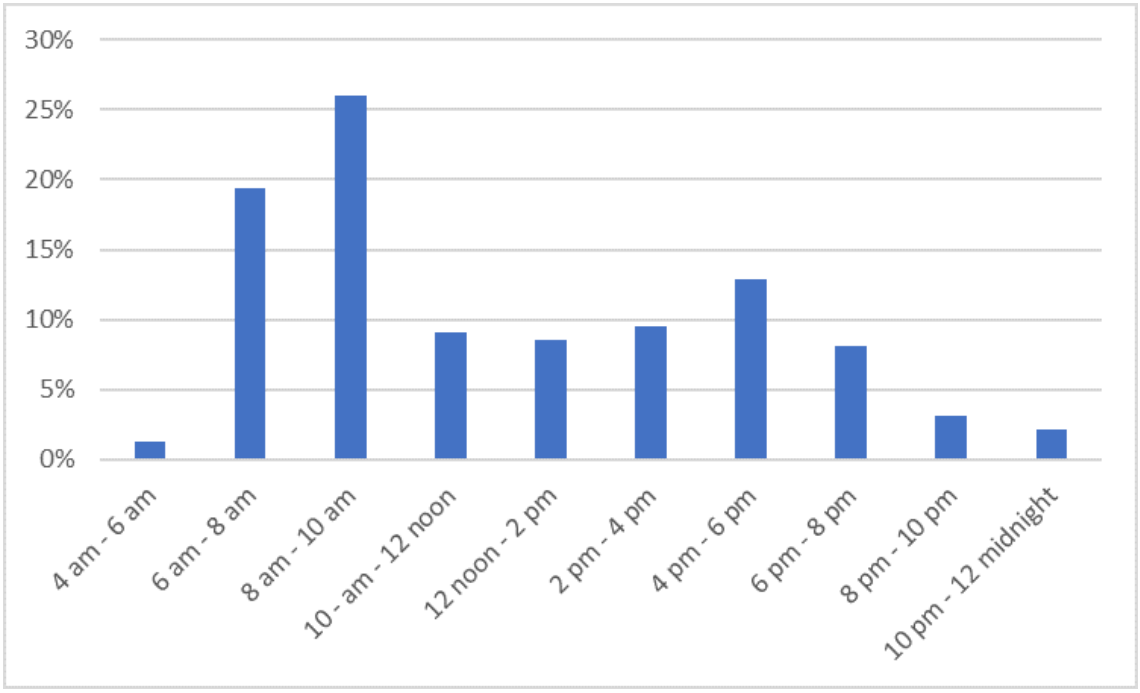


Figure 26: *Temporal Travel Patterns for Cluster 8*

Based on information in the TfL dataset, and after associating each passenger with their corresponding cluster, key travel patterns can be provided for passengers in Cluster 7. As indicated in Figure 26, their travel patterns are heavily skewed towards the morning, with approx. 45 per cent of all journeys occurring between 6 am - 10 am and very little travel occurring after 8 pm (approximately 5 per cent). Unlike many of their fellow passengers, they travel most frequently on Thursdays; however, they too have a relatively uniform distribution of travel throughout the work week, with decreased travel on weekends. Their five most frequently used stations of entry are Pinner, Hornchurch, West Ruislip, Upminster Bridge, and Ickenham.

5.5. Summary of Findings

As discussed in sections 5.3 and 5.4 above, we can determine the key socio-demographic characteristics and most significant travel patterns for passengers in each cluster. A summary of these findings is provided below in Table 3.

Cluster	Key Socio-Demographic Characteristics	Key Travel Characteristics
1	Most likely identify as White; not single; not aged 25-44 years; educational attainment unlikely to exceed Level 1, Level 2, or apprenticeship; unlikely to be employed in the IT or scientific sectors.	Top 5 stations: Dagenham Heathway, Becontree, Elm Park, Dagenham East, and Ruislip Gardens; high frequency of travel between 6 am - 8 am; very low frequency of travel after 8 pm; relatively uniform distribution of travel throughout work week and low frequency of travel on weekends.
2	Most likely identify as Black; not married; no clear indication of age group; educational attainment unlikely to be at Level 4 or above; likely to be employed in accommodation or food service sectors.	Top 5 stations: Bond Street, Warren Street, Mile End, Barking, and Whitechapel; high frequency of travel after 6 pm; relatively uniform distribution of travel throughout work week and low frequency of travel on weekends.
3	Most likely identify as having Indian or Chinese ethnicity; married; not aged 25-44 years; educational attainment unlikely to be at Level 4 or above; likely to be employed in wholesale or retail trade sectors.	Top 5 stations: Woodford, Gants Hill, Newbury Park, Gunnersbury, and Northolt; high frequency of travel between 6 am - 10 am; very low frequency of travel after 8 pm; relatively uniform distribution of travel throughout work week and low frequency of travel on weekends.
4	Most likely identify as White; single; aged 25-44 years; educational attainment likely to be at Level 4 or above; likely to be employed in finance, insurance, real estate, IT, or scientific sectors.	Top 5 stations: Oxford Circus, Stratford, Piccadilly Circus, Tottenham Court Road, and Holborn; relatively low frequency of travel in the morning; high frequency of travel after 8 pm compared to other clusters; relatively uniform distribution of travel throughout work week and low frequency of travel on weekends.
5	Most likely identify as White; single; aged 25-44 years; educational attainment likely to be at Level 4 or above; likely to be employed in IT or scientific sectors.	Top 5 stations: Ealing Broadway, North Greenwich, Camden Town, Canada Water, and Wood Green; peak travel occurring between 8 am - 10 am and between 4 pm - 6 pm; low frequency of travel after 8 pm; relatively uniform distribution of travel throughout work week and low frequency of travel on weekends.

6	Most likely identify as White; married; no clear indication of age group; educational attainment likely to be at Level 4 or above; likely to be employed in IT or scientific sectors.	Top 5 stations: Westminster, Notting Hill Gate, Tower Hill, Barbican, and Barons Court; peak travel occurring between 8 am - 10 am and between 4 pm - 6 pm; low frequency of travel after 8 pm; relatively uniform distribution of travel throughout work week and low frequency of travel on weekends.
7	Most likely to identify as Black; no clear indication of age or marital status; educational attainment unlikely to be at Level 4 or above; likely to be employed in accommodation or food service sectors.	Top 5 stations: Walthamstow Central, Seven Sisters, East Ham, Wembley Park, and Leyton; high frequency of travel between 6 am - 10 am; low frequency of travel after 8 pm; relatively uniform distribution of travel throughout work week and low frequency of travel on weekends.
8	Most likely identify as White; married; aged either between 45-64 years or 65-89 years; educational attainment unlikely to exceed Level 1, Level 2, or apprenticeship; no clear indication of employment sector.	Top 5 stations: Pinner, Hornchurch, West Ruislip, Upminster Bridge, and Ickenham; high frequency of travel between 6 am - 10 am; very low frequency of travel after 8 pm; relatively uniform distribution of travel throughout work week and low frequency of travel on weekends.

Table 1

As discussed above, the Further Socio-Demographic Analysis section provides us with potentially very useful information about the travel patterns of individuals in each cluster. We can identify which days of the week, which times during the day, and which stations are most likely to be used by an individual, or group of individuals, based on certain socio-demographic characteristics. This information can be used for a variety of purposes, including providing better customer services and delivering targeted advertising, as discussed in the section 6 below.

6. Conclusion

This report provides an overview of the findings from analysis conducted on a dataset of TfL passengers and their associated socio-demographic characteristics. Key research questions were addressed by these findings - namely: i) what information can be uncovered about where passengers live based on their travel patterns; ii) how can we group TfL passengers based on their socio-demographic characteristics; and iii) what information can be uncovered about the travel patterns of different groups of TfL passengers based on their socio-demographic characteristics?

A variety of approaches were taken to address the above research questions. An analysis of the travel patterns of two sample passengers, including their most frequently used stations of entry and the times during which they travelled, were used to formulate an indication of the respective passenger's approximate place of residence, addressing research question (i) as discussed in section

4.3. In order to tackle research question (ii), cluster analysis was conducted on a dataset consisting of unique passengers and their associated socio-demographic characteristics, as discussed in section 4.4. Research question (iii) was addressed by using the groups obtained from the cluster analysis and analyzing their individual travel patterns, as discussed in section 4.5.

The analysis conducted in this report provides key findings on the travel behaviour of TfL passengers and their most likely socio-demographic characteristics. Passengers in the dataset were grouped into eight distinct clusters and each cluster was examined with regard to various attributes, including ethnicity, age group, marital status, employment sector, and education levels, in addition to travel behaviour such as frequently used stations, travel frequency throughout the week, and travel frequency throughout the day, as discussed in sections 5.3 and 5.4.

Such information can be used for a range of purposes. One application is providing better customer service at TfL stations. If there is more information available on the types of passengers who use the London Underground service, particularly related to areas which TfL may not have access to, such as marital status, age group, ethnicity, employment sector, and education levels, then TfL could customize their services and better train their staff to more effectively cater to these customers.

Another application of the analysis conducted in this project is targeted advertising. TfL passengers' socio-demographic information could be very useful for advertising agencies to target products and services at specific demographic groups of passengers at selected stations and at specific times of the day. Using such information could help cut advertising expenditure and boost brand awareness among target audiences.

6.1. Lessons Learned and Future Work

Due to the size of the original dataset used, which consisted of over 940 million records, issues were encountered related to insufficient memory[4]. Despite using big data technologies such as Spark SQL and PySpark, certain queries, including the creation of a dataset of unique passengers and subsequent clustering analysis, was not possible on the entire dataset. As such, a representative random sample was obtained.

However, future analysis would ideally involve using the entire dataset. Doing so could help provide a more accurate set of results related to the travel patterns and socio-demographic characteristics of TfL passengers. As cloud computing platforms providing distributed processing power become increasingly available, there is no doubt that conducting such analysis will be possible.

References

[1] Agard, Bruno, Catherine Morency, and Martin Trepanier. "Mining Public Transport User Behaviour from Smart Card Data." Paper presented at the 12th IFAC Symposium on Information Control Problems in Manufacturing, Saint-Etienne, France, 17-19 May 2006, pp. 1-6. http://citeseer.ist.psu.edu/viewdoc/summary?doi=10.1.1.156.3799&rank=1.

[4] An example of the error messages encountered are provided in Appendix 3.

[2] Anand, Sesham, P. Padmanabham, and A. Govardhan. "Application of Factor Analysis to K-Means Clustering Algorithm on Transportation Data." *International Journal of Computer Applications*, 95(15), pp. 40-46, 2014.

[3] Bruno, Agard, Vahid Partovi-Nia, and Martin Trepanier. "Assessing Public Transport Travel Behaviour from Smart Card Data with Advanced Data Mining Techniques." Paper presented at the 13th World Conference on Transport Research Society, Rio de Janeiro, Brazil, 15-18 July 2013.

[4] Cai, Xiao, Feiping Nie, and Heng Huang. "Multi-View K-Means Clustering on Big Data." Paper presented at the Twenty-Third International Joint Conference on Artificial Intelligence, Beijing, China, August 3-9 2013.

[5] Case, Anne C. and Lawrence F. Katz. "The Company You Keep: The Effects of Family and Neighborhood on Disadvantaged Youths." National Bureau of Economic Research. NBER Working Paper No. 3705, 1991. http://www.nber.org/papers/w3705.

[6] Dudas, Catarina, Christopher Engstrom, Johan Karlsson, Frida Nellros, Luyuan Qi-Gautier, Sergei Silvestrov, and Jesper Ulke. "Investigate Graph and Network Algorithms in Transport Vehicle GPS Data to Detect and Quantify Hubs and Flow." Swedish Study Group on Mathematics in Industry. 2015. http://eu-maths-in.se/wp-content/uploads/SSGMI_2015_report_Scania.pdf.

[7] El Mahsri, Mohamed K., Etienne Come, Johanna Baro, and Latifa Oukhellou. "Understanding Passenger Patterns in Public Transit through Smart Card and Socioeconomic Data - A Case Study in Rennes, France." Paper presented at UrbComp '14, New York, USA, 24 August 2014, pp. 1-9. http://www.comeetie.fr/pdfrepos/urbcomp2014.pdf.

[8] Galba, Tomislav, Zoran Balkic, and Goran Martinovic. "Public Transportation Big Data Clustering." *International Journal of Electrical and Computer Engineering Systems*, 4 (1), pp. 21-26, 2013. http://hrcak.srce.hr/133181?lang=en.

[9] Han, Jiawei, Micheline Kamber, and Anthony Tung. "Spatial Clustering Methods in Data Mining: A Survey." In: Miller, H. and Han, J. (eds), *Geographic Data Mining and Knowledge Discovery, Research Monographs in GIS*, London: Taylor and Francis, pp. 188-217, 2001. http://www.comp.nus.edu.sg/~atung/publication/gkdbk01.pdf.

[10] Hyde, Richard and Plamen Angelov. "Data Density Based Clustering." Paper presented at the 14th UK Workshop on Computational Intelligence, Bradford, UK, 8-10 September 2014, pp. 1-7. http://ieeexplore.ieee.org/document/6930157/.

[11] Jain, Anil K. "Data Clustering: 50 Years Beyond K-means." *Pattern Recognition Letters* 31 (2010), pp. 651-666, 2009. https://msu.edu/~ashton/classes/866/papers/2010_jain_kmeans_50yrs__clustering_review.pdf.

[12] Kaplan, George A. "People and Places: Contrasting Perspectives on the Association between Social Class and Health." *International Journal of Health Services* 26 (3), pp. 507-519, 1996. http://journals.sagepub.com/doi/abs/10.2190/4CUU-7B3G-G4XR-0K0B.

[13] London Datastore. London Output Area Classification. https://data.london.gov.uk/loac/.

[14] Longley, Paul and Alex Singleton. "London Output Area Classification (LOAC) - Final Report." Census Information Scheme, 2011. https://data.london.gov.uk/dataset/london-area-classification.

[15] Liu, Yanchi, Zhongmou Li, Hui Xiong, Xuedong Gao, and Junjie Wu. "Understanding of Internal Clustering Validation Measures." Paper presented at the 2010 IEEE International Conference on Data Mining, Sydney, Australia, 13-17 December 2010, pp. 911-916. http://datamining.rutgers.edu/publication/internalmeasures.pdf.

[16] Martins-Yedenu, Patrick. "The Impact of Weather Events on TfL." MSc Individual Project Dissertation. Faculty of Natural and Mathematical Sciences, Department of Informatics, King's College London. 2016.

[17] Maulik, Ujjwal and Sanghamitra Bandyopadhyay. "Performance Evaluation of Some Clustering Algorithms and Validity Indices." *IEEE Transactions on Pattern Analysis and Machine Intelligence* 24 (12), pp. 1650-1654, 2002. http://ieeexplore.ieee.org/abstract/document/1114856/.

[18] Moore, Roger. "Big Data TfL Database - Appendix A: User Guide". BSc Individual Project Dissertation. Faculty of Natural and Mathematical Sciences, Department of Informatics, King's College London. 2017.

[19] Morency, Catherine, Martin Trepanier, and Bruno Agard. "Analysing the Variability of Transit Users Behaviour with Smart Card Data." Paper presented at the 2006 IEEE Intelligent Transportation Systems Conference, Toronto, Canada, 17-20 September 2006, pp. 44-49. http://ieeexplore.ieee.org/document/1706716/.

[20] Office for National Statistics. "2011 OAC Variables - Glossary." 2015. http://webarchive.nationalarchives.gov.uk/20160114210848/http://www.ons.gov.uk/ons/guide-method/geography/products/area-classifications/ns-area-classifications/ns-2011-area-classifications/methodology-and-variables/index.html.

[21] Ortega-Tong, Meisy. "Classification of London's Public Transport Users Using Smart Card Data." Thesis, Master of Science in Transportation, published by Massachusetts Institute of Technology, 2013.

[22] Pappa, Theodora. "Smart City Transport Spatiotemporal Framework." MSc Individual Project Dissertation. Faculty of Natural and Mathematical Sciences, Department of Informatics, King's College London. 2015.

[23] Tan, Pang-Ning, Michael Steinbach, and Vipin Kumar. "Cluster Analysis: Basic Concepts and Algorithms." In *Introduction to Data Mining*, pp.487-568. Harlow: Pearson Education Limited. 2014.

[24] Transport for London. "Annual Report and Statement of Accounts 2015/16." July 2016. http://content.tfl.gov.uk/tfl-annual-report-2015-16.pdf.

[25] Transport for London. "What is Oyster?" https://tfl.gov.uk/fares-and-payments/oyster/what-is-oyster.

[26] Sklar, Elizabeth. "sd-clean.py" Python code received in Data Mining 7CCSMDM1 Module, King's College London. 2017.

[27] Stackoverflow. "Group by and find top n value counts pandas." Posted by jezrael. 2016. https://stackoverflow.com/questions/35364601/group-by-and-find-top-n-value-counts-pandas.

[28] Stackoverflow. "Pandas create new column based on values from other columns." Posted by Tom Kimber. 2014. https://stackoverflow.com/questions/26886653/pandas-create-new-column-based-on-values-from-other-columns?noredirect=1lq=1.

[29] Stackoverflow. "Python: select most frequent using group by." Posted by Karl D. 2014. https://stackoverflow.com/questions/23692419/python-select-most-frequent-using-group-by.

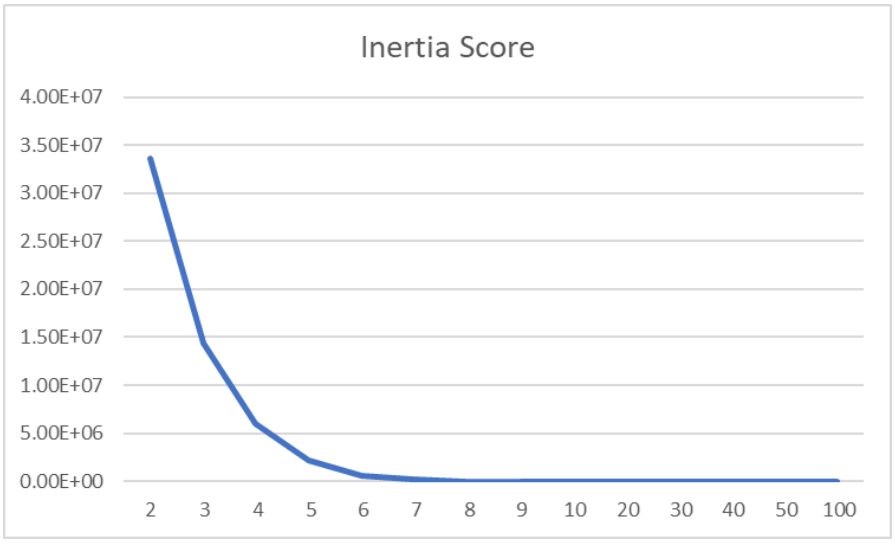
[30] Stackoverflow. "Insert a row to pandas dataframe." Posted by Piotr Migdal. 2014. https://stackoverflow.com/questions/24284342/insert-a-row-to-pandas-dataframe.

[31] Webb, Barry and Gloria Laycock. "Reducing Crime on the London Underground - An Evaluation of Three Pilot Projects." Home Office Crime Prevention Unit. Crime Prevention Unit: Paper No. 30. 1992. http://www.popcenter.org/library/scp/pdf/189-Webb_and_Laycock.pdf.

Appendices

Appendix 1: Inertia and Calinski-Harabasz Scores

Figure A.1: Inertia Scores and Calinski-Harabasz Scores for Set of Features related to Age Group



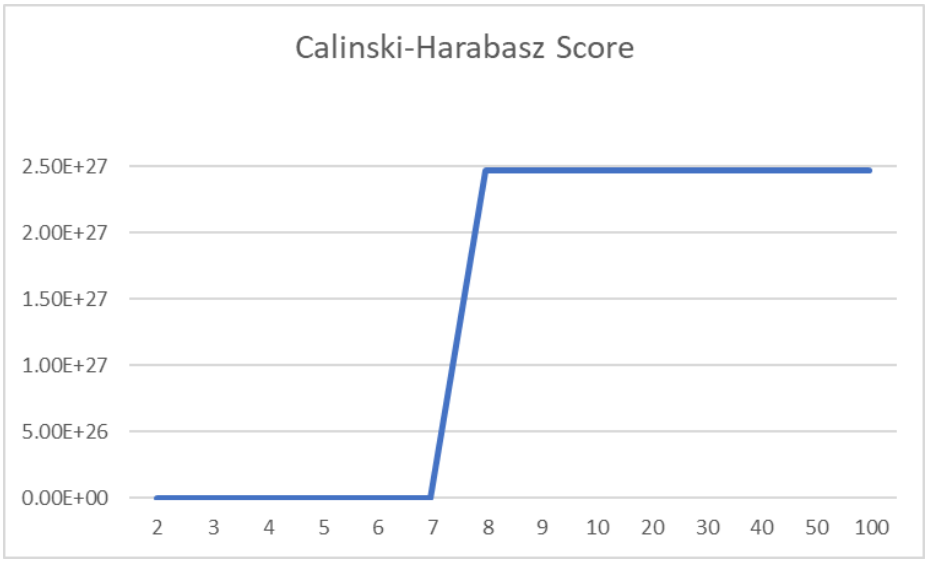
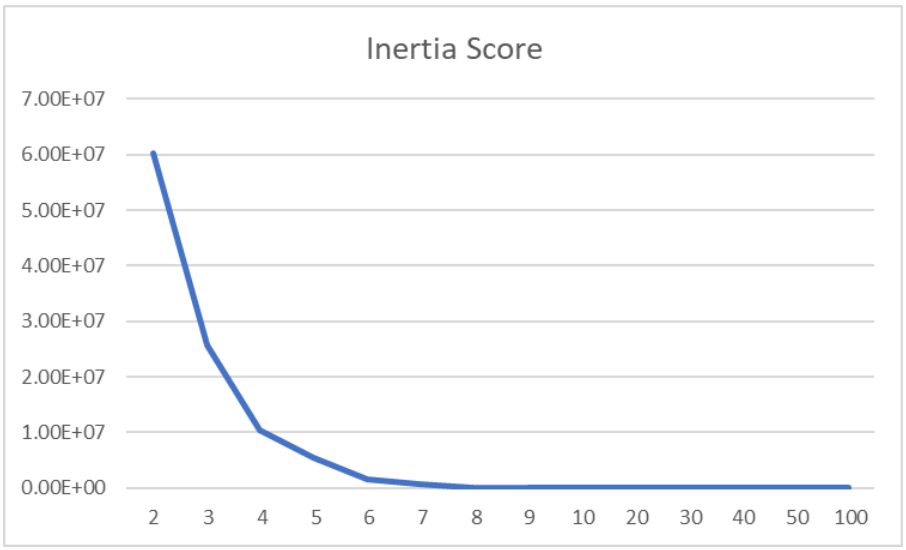


Figure A.2: Inertia Scores and Calinski-Harabasz Scores for Set of Features related to Marital Status



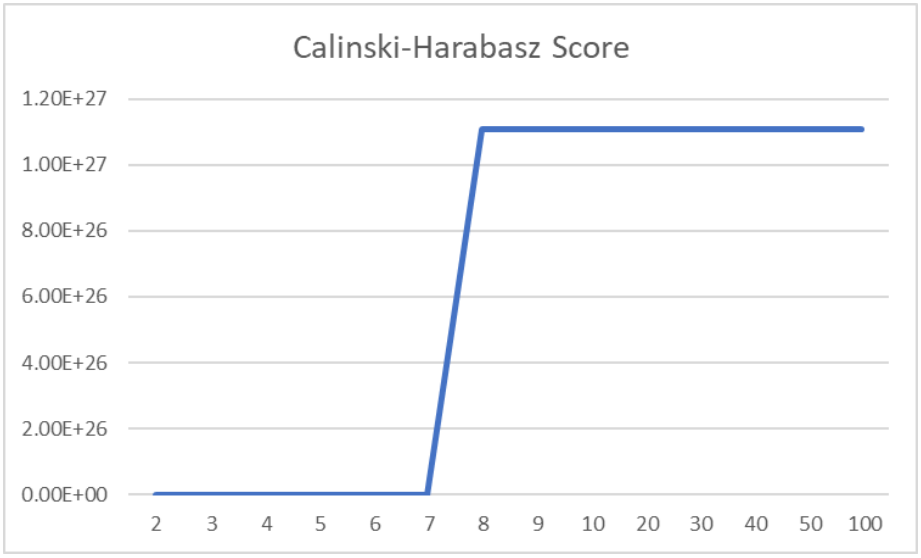
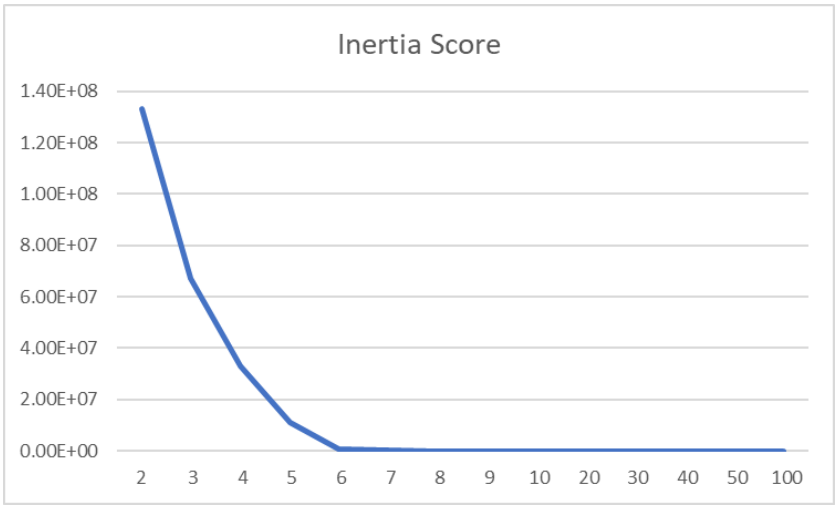


Figure A.3: Inertia Scores and Calinski-Harabasz Scores for Set of Features related to Ethnicity



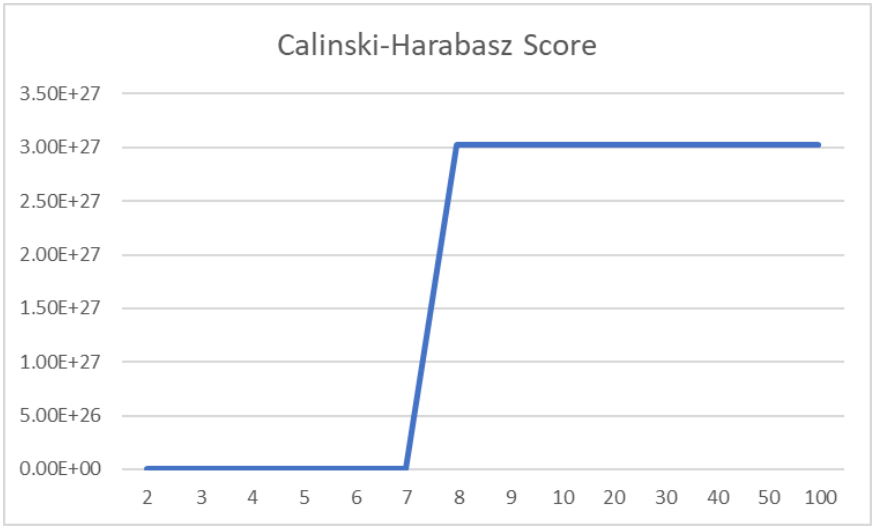
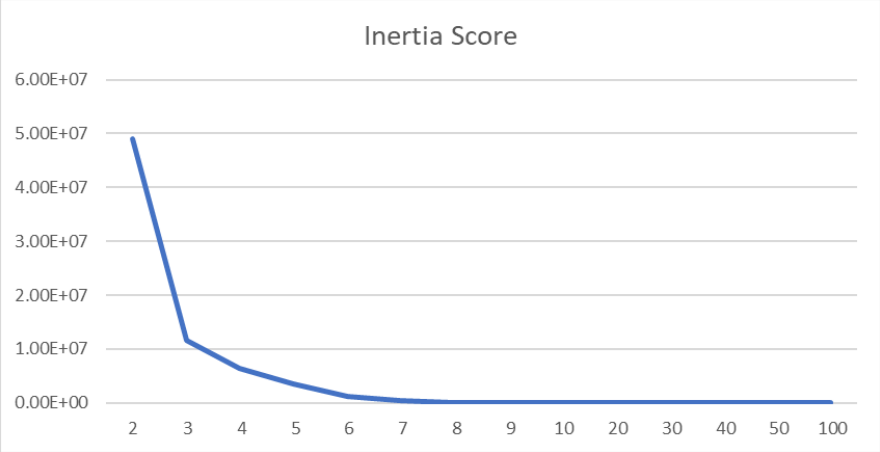


Figure A.4: Inertia Scores and Calinski-Harabasz Scores for Set of Features related to Employment Sector



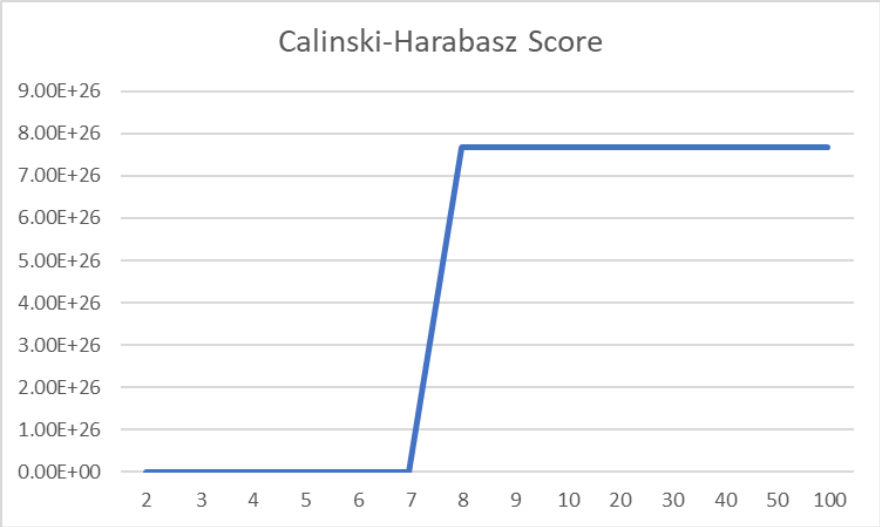
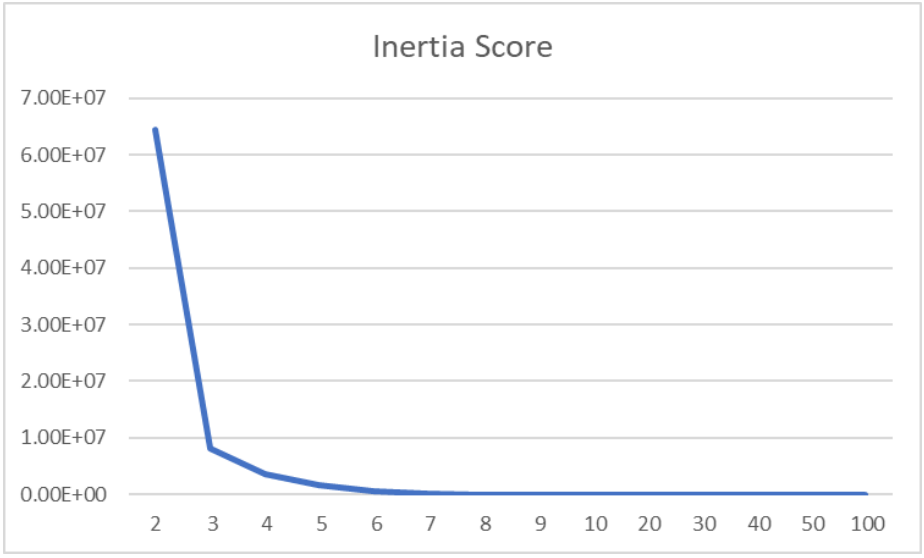


Figure A.5: Inertia Scores and Calinski-Harabasz Scores for Set of Features related to Education



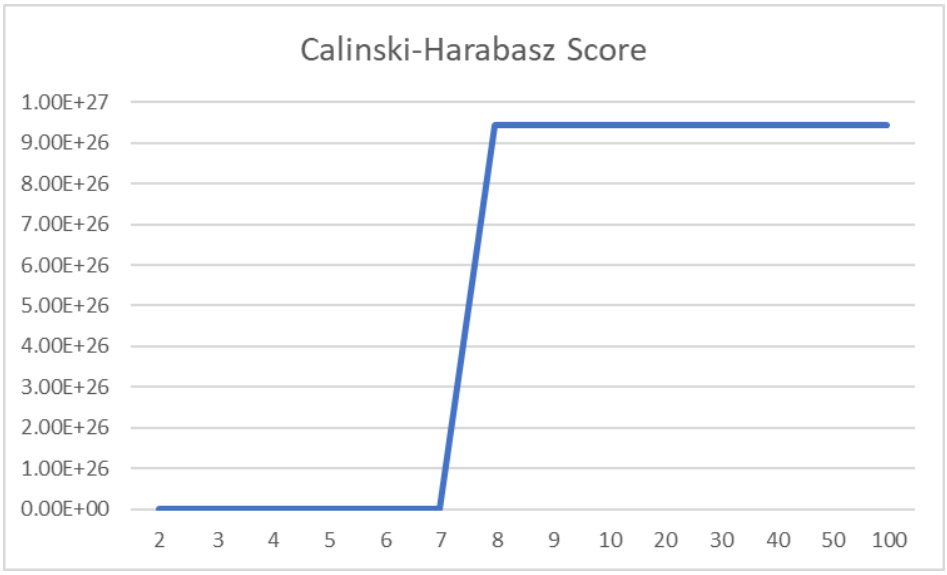


Table A.1: Inertia and Calinski-Harabasz (C-H) Evaluation Scores for K-Means Clustering for each Set of Related Features

	Age		Marital Status		Ethnicity		Employment Contract		Employment Sector		Education	
No. of clusters	**Inertia**	**C-H Score**	**Inertia**	**C-H Score**	**Inertia**	**C-H Score**	**Inertia**	**C-H Score**	**Inertia**	**C-H Score**	**Inertia**	**C-H Score**
2	3.36E+07	1.34E+07	6.03E+07	7.27E+06	1.33E+08	7.90E+06	2.61E+07	1.55E+07	4.89E+07	9.49E+06	6.45E+07	1.68E+07
3	1.44E+07	1.70E+07	2.58E+07	9.93E+06	6.71E+07	8.90E+06	1.85E+06	1.23E+08	1.17E+07	2.33E+07	8.06E+06	7.46E+07
4	5.96E+06	2.84E+07	1.05E+07	1.73E+07	3.32E+07	1.27E+07	514901	2.97E+08	6.45E+06	2.87E+07	3.57E+06	1.13E+08
5	2.17E+06	5.94E+07	5.50E+06	2.53E+07	1.12E+07	2.94E+07	62999.9	1.82E+09	3.50E+06	4.02E+07	1.60E+06	1.91E+08
6	513212	2.03E+08	1.64E+06	6.87E+07	579747	4.62E+08	10337.2	8.89E+09	1.23E+06	9.22E+07	581551	4.19E+08
7	186988	4.64E+08	712839	1.33E+08	103356	2.16E+09	2129.4	3.60E+10	442266	2.15E+08	54816.5	3.71E+09
8	1.07E-14	2.47E+27	1.63E-14	1.11E+27	2.97E-14	3.02E+27	2.06E-14	6.81E+26	3.60E-14	7.68E+26	1.12E-13	9.45E+26
9	1.07E-14	2.47E+27	1.05E-15	1.11E+27	2.97E-14	3.02E+27	3.07E-15	6.81E+26	3.60E-14	7.68E+26	1.22E-15	9.45E+26
10	1.07E-14	2.47E+27	1.05E-15	1.11E+27	3.83E-15	3.02E+27	3.07E-15	6.81E+26	3.60E-14	7.68E+26	1.22E-15	9.45E+26
20	7.69E-16	2.47E+27	6.93E-16	1.11E+27	2.46E-15	3.02E+27	3.07E-15	6.81E+26	1.45E-15	7.68E+26	1.19E-15	9.45E+26
30	5.87E-16	2.47E+27	3.60E-16	1.11E+27	2.46E-15	3.02E+27	9.07E-16	6.81E+26	1.45E-15	7.68E+26	8.19E-16	9.45E+26
40	5.87E-16	2.47E+27	3.60E-16	1.11E+27	2.46E-15	3.02E+27	9.07E-16	6.81E+26	1.45E-15	7.68E+26	8.19E-16	9.45E+26
50	5.87E-16	2.47E+27	3.60E-16	1.11E+27	2.46E-15	3.02E+27	9.07E-16	6.81E+26	1.45E-15	7.68E+26	8.19E-16	9.45E+26
100	2.71E-19	2.47E+27	3.50E-16	1.11E+27	2.08E-15	3.02E+27	7.98E-16	6.81E+26	1.45E-15	7.68E+26	5.22E-20	9.45E+26

Appendix 2: Cluster Centroid Values

Table A.2: Centroid Values for all features in each Cluster

	1	2	3	4	5	6	7	8
% persons aged 0 to 4	0.6	1.2	-0.1	-2	-0.5	0.3	1.5	-1.9
% persons aged 5 to 14	2.4	2.4	1.4	-6.7	-2.6	-1.3	1.9	0.9
% persons aged 25 to 44	-6.9	-0.9	-4.1	18.4	7.9	1.1	0.3	-13.3
% persons aged 45 to 64	1.9	-2.6	1.1	-5.8	-2	1.9	-2.5	7.8
% persons aged 65 to 89	1.6	-2	0.4	-4	-2.2	1.8	-2.8	7.4
% persons aged 90 and over	0.2	-0.1	-0.1	-0.2	-0.1	0.3	-0.2	0.3
% persons who are divorced or separated	2.6	4	-2.7	-2.4	0	-1.4	1.8	-3.2
% persons who are married or in a civil partnership	0.8	-11	11.5	-10.5	-10.8	5.9	-2.6	17.8
% persons who are single	-5.1	7.1	-9	15.6	12	-4.6	1.2	-16.6
% persons who have Arab or any other ethnicity	-1.7	2.1	0.8	1.6	-0.3	-1.1	1.1	-2.2
% persons who have Bangladeshi ethnicity	-1.6	6	0.4	-1.3	-1	-2.1	1.2	-2.1
% persons who have Black ethnicity	0.9	13.4	-2.7	-8.1	2	-9.7	9.6	-10
% persons who have Chinese or Other Asian ethnicity	-2.2	-0.6	5.6	2.1	-1.7	-1.8	1.8	-2.5
% persons who have Indian ethnicity	-3.2	-4	16.2	-1.4	-3.7	-2.3	0.4	-0.7
% persons who have mixed ethnicity	-0.3	1.6	-1.2	-0.1	1.4	-0.8	1.1	-2.3
% persons who have Pakistani ethnicity	-1.2	-1.4	5.8	-1.4	-1.6	-1.5	2.5	-1.4
% persons who have White ethnicity	9.3	-17.2	-24.8	8.6	4.8	19.2	-17.6	21.2
% working age adults who work in accommodation or food service	-1.3	4.2	0.5	-0.9	0.1	-2.8	2.9	-3.4
% working age adults who work in admin or support service	0.2	1.8	0.3	-1.6	-0.4	-1.4	1.9	-1.6
% working age adults who work in agriculture, forestry, or fishing	0	0	0	0	0	0	0	0
% working age adults who work in education	0.3	-0.1	-0.8	-2.8	0.7	0.9	-0.4	1.6
% working age adults who work in energy, water, or air conditioning supply	0.5	-0.1	-0.1	-0.2	-0.2	-0.2	0.1	0.2
% working age adults who work in finance, insurance, or real estate	-2.8	-2.8	-2.8	10.4	-0.2	3.8	-4	1.6
% working age adults who work in human health or social work	2	1.6	-0.6	-3.3	-0.6	-1.3	1.3	-0.3
% working age adults who work in mining, quarrying, or construction	2.9	-1.7	1.5	-3.6	-2.2	-1.7	1.6	2.6

% working age adults who work in professional, scientific, or technical industries	-7.4	-3.5	-5	11.4	7	8.3	-6.3	-1.9
% working age adults who work in public admin or defense	1.2	-0.4	-0.1	-1.5	-0.2	-0.2	-0.3	1.5
% working age adults who work in manufacturing industry	1	-0.9	1.3	-1	-0.9	-0.4	0	1
% working age adults who work in transport or storage industries	2.2	0.7	2.8	-3.1	-1.9	-2.3	1.4	-0.3
% working age adults who work in wholesale or retail trade	2.2	0.8	4.6	-4.6	-2.8	-3.5	2.3	0.2
% persons whose highest qualification is Level 3	0.2	0	0	-0.8	-0.3	-0.4	-0.2	1.4
% persons whose highest qualification is Level 1, Level 2, or Apprenticeship	9.3	1	1.6	-12.9	-5.6	-6.1	1.6	8.8
% persons whose highest qualification is Level 4 or above	-15.6	-9.4	-6.5	24.6	10.3	16.9	-9.3	-5.3

Appendix 3: Supergroup Characteristics

Table A.3: Mean Values of Supergroups based on Key Socio-Demographic Features

	A	B	C	D	E	F	G	H
% persons aged 0 to 4	0.6	1.2	-0.1	-2	-0.5	0.3	1.5	-1.9
% persons aged 5 to 14	2.4	2.4	1.4	-6.7	-2.6	-1.3	1.9	0.9
% persons aged 25 to 44	-6.9	-0.9	-4.1	18.4	7.9	1.1	0.3	-13.3
% persons aged 45 to 64	1.9	-2.6	1.1	-5.8	-2	1.9	-2.5	7.8
% persons aged 65 to 89	1.6	-2	0.4	-4	-2.2	1.8	-2.8	7.4
% persons aged 90 and over	0.2	-0.1	-0.1	-0.2	-0.1	0.3	-0.2	0.3
% persons who are divorced or separated	2.6	4	-2.7	-2.4	0	-1.4	1.8	-3.2
% persons who are married or in a civil partnership	0.8	-11	11.5	-10.5	-10.8	5.9	-2.6	17.8
% persons who are single	-5.1	7.1	-9	15.6	12	-4.6	1.2	-16.6
% persons who have Arab or any other ethnicity	-1.7	2.1	0.8	1.6	-0.3	-1.1	1.1	-2.2
% persons who have Bangladeshi ethnicity	-1.6	6	0.4	-1.3	-1	-2.1	1.2	-2.1
% persons who have Black ethnicity	0.9	13.4	-2.7	-8.1	2	-9.7	9.6	-10
% persons who have Chinese or Other Asian ethnicity	-2.2	-0.6	5.6	2.1	-1.7	-1.8	1.8	-2.5
% persons who have Indian ethnicity	-3.2	-4	16.2	-1.4	-3.7	-2.3	0.4	-0.7
% persons who have mixed ethnicity	-0.3	1.6	-1.2	-0.1	1.4	-0.8	1.1	-2.3
% persons who have Pakistani ethnicity	-1.2	-1.4	5.8	-1.4	-1.6	-1.5	2.5	-1.4
% persons who have White ethnicity	9.3	-17.2	-24.8	8.6	4.8	19.2	-17.6	21.2
% working age adults who work in accommodation or food service	-1.3	4.2	0.5	-0.9	0.1	-2.8	2.9	-3.4
% working age adults who work in admin or support service	0.2	1.8	0.3	-1.6	-0.4	-1.4	1.9	-1.6
% working age adults who work in agriculture, forestry, or fishing	0	0	0	0	0	0	0	0
% working age adults who work in education	0.3	-0.1	-0.8	-2.8	0.7	0.9	-0.4	1.6
% working age adults who work in energy, water, or air conditioning supply	0.5	-0.1	-0.1	-0.2	-0.2	-0.2	0.1	0.2
% working age adults who work in finance, insurance, or real estate	-2.8	-2.8	-2.8	10.4	-0.2	3.8	-4	1.6
% working age adults who work in human health or social work	2	1.6	-0.6	-3.3	-0.6	-1.3	1.3	-0.3
% working age adults who work in mining, quarrying, or construction	2.9	-1.7	1.5	-3.6	-2.2	-1.7	1.6	2.6

% working age adults who work in professional, scientific, or technical industries	-7.4	-3.5	-5	11.4	7	8.3	-6.3	-1.9
% working age adults who work in public admin or defense	1.2	-0.4	-0.1	-1.5	-0.2	-0.2	-0.3	1.5
% working age adults who work in manufacturing industry	1	-0.9	1.3	-1	-0.9	-0.4	0	1
% working age adults who work in transport or storage industries	2.2	0.7	2.8	-3.1	-1.9	-2.3	1.4	-0.3
% working age adults who work in wholesale or retail trade	2.2	0.8	4.6	-4.6	-2.8	-3.5	2.3	0.2
% persons whose highest qualification is Level 3	0.2	0	0	-0.8	-0.3	-0.4	-0.2	1.4
% persons whose highest qualification is Level 1, Level 2, or Apprenticeship	9.3	1	1.6	-12.9	-5.6	-6.1	1.6	8.8
% persons whose highest qualification is Level 4 or above	-15.6	-9.4	-6.5	24.6	10.3	16.9	-9.3	-5.3

Appendix 4: Spark Error Messages

Example of error messages received when attempting to create dataframe of unique passengers in Spark database:

```
scala> val sqlFrame = spark.sql("SELECT PRESTIGEID, STATIONOFFIRSTENTRYKEY
FROM (SELECT PRESTIGEID, STATIONOFFIRSTENTRYKEY, ROW_NUMBER() OVER
(PARTITION BY PRESTIGEID ORDER BY freq DESC) AS rn FROM (SELECT
PRESTIGEID, STATIONOFFIRSTENTRYKEY, COUNT('x') AS freq FROM tfldataset
GROUP BY 1, 2) freq) ranked_station_count WHERE rn = 1")
sqlFrame: org.apache.spark.sql.DataFrame = [PRESTIGEID: string,
STATIONOFFIRSTENTRYKEY: string]

scala> sqlFrame.show(20)
[Stage 28:>                                                 (0 + 24) / 200]17/07/27 18:00:53 ERROR
DiskBlockObjectWriter: Uncaught exception while reverting partial writes to file
/tmp/blockmgr-17538d4a-76fb-43aa-b086-c5efbd96fc23/00/temp_shuffle_2234d2c9-d38d-494e-
922b-13a3e4ebea73
java.io.FileNotFoundException: /tmp/blockmgr-17538d4a-76fb-43aa-b086-
c5efbd96fc23/00/temp_shuffle_2234d2c9-d38d-494e-922b-13a3e4ebea73 (Too many open files)
        at java.io.FileOutputStream.open(Native Method)
        at java.io.FileOutputStream.<init>(FileOutputStream.java:221)
        at
org.apache.spark.storage.DiskBlockObjectWriter.revertPartialWritesAndClose(DiskBlockObject
Writer.scala:210)
        at
org.apache.spark.shuffle.sort.BypassMergeSortShuffleWriter.stop(BypassMergeSortShuffleWrit
er.java:238)
        at org.apache.spark.scheduler.ShuffleMapTask.runTask(ShuffleMapTask.scala:102)
        at org.apache.spark.scheduler.ShuffleMapTask.runTask(ShuffleMapTask.scala:53)
        at org.apache.spark.scheduler.Task.run(Task.scala:99)
        at org.apache.spark.executor.Executor$TaskRunner.run(Executor.scala:282)
        at java.util.concurrent.ThreadPoolExecutor.runWorker(ThreadPoolExecutor.java:1145)
        at java.util.concurrent.ThreadPoolExecutor$Worker.run(ThreadPoolExecutor.java:615)
        at java.lang.Thread.run(Thread.java:745)
17/07/27 18:00:53 ERROR BypassMergeSortShuffleWriter: Error while deleting file
/tmp/blockmgr-17538d4a-76fb-43aa-b086-c5efbd96fc23/00/temp_shuffle_2234d2c9-d38d-494e-
922b-13a3e4ebea73
17/07/27 18:00:53 ERROR Executor: Exception in task 4.0 in stage 28.0 (TID 16828)
java.io.FileNotFoundException: /tmp/blockmgr-17538d4a-76fb-43aa-b086-
c5efbd96fc23/17/temp_shuffle_de7b22e3-156e-447d-9318-412db829857a (Too many open
files)
        at java.io.FileOutputStream.open(Native Method)
        at java.io.FileOutputStream.<init>(FileOutputStream.java:221)
```

```
	at
org.apache.spark.storage.DiskBlockObjectWriter.initialize(DiskBlockObjectWriter.scala:102)
	at
org.apache.spark.storage.DiskBlockObjectWriter.open(DiskBlockObjectWriter.scala:115)
	at
org.apache.spark.storage.DiskBlockObjectWriter.write(DiskBlockObjectWriter.scala:229)
	at
org.apache.spark.shuffle.sort.BypassMergeSortShuffleWriter.write(BypassMergeSortShuffleWri
ter.java:152)
	at org.apache.spark.scheduler.ShuffleMapTask.runTask(ShuffleMapTask.scala:96)
	at org.apache.spark.scheduler.ShuffleMapTask.runTask(ShuffleMapTask.scala:53)
	at org.apache.spark.scheduler.Task.run(Task.scala:99)
	at org.apache.spark.executor.Executor$TaskRunner.run(Executor.scala:282)
	at java.util.concurrent.ThreadPoolExecutor.runWorker(ThreadPoolExecutor.java:1145)
	at java.util.concurrent.ThreadPoolExecutor$Worker.run(ThreadPoolExecutor.java:615)
	at java.lang.Thread.run(Thread.java:745)
17/07/27 18:00:53 ERROR Executor: Exception in task 10.0 in stage 28.0 (TID 16834)
java.io.FileNotFoundException: /tmp/blockmgr-17538d4a-76fb-43aa-b086-
c5efbd96fc23/22/temp_shuffle_e42ed550-23b9-4909-9ba8-ae7a455ab862 (Too many open files)
	at java.io.FileOutputStream.open(Native Method)
	at java.io.FileOutputStream.<init>(FileOutputStream.java:221)
	at
org.apache.spark.storage.DiskBlockObjectWriter.initialize(DiskBlockObjectWriter.scala:102)
	at
org.apache.spark.storage.DiskBlockObjectWriter.open(DiskBlockObjectWriter.scala:115)
	at
org.apache.spark.storage.DiskBlockObjectWriter.write(DiskBlockObjectWriter.scala:229)
	at
org.apache.spark.shuffle.sort.BypassMergeSortShuffleWriter.write(BypassMergeSortShuffleWri
ter.java:152)
	at org.apache.spark.scheduler.ShuffleMapTask.runTask(ShuffleMapTask.scala:96)
	at org.apache.spark.scheduler.ShuffleMapTask.runTask(ShuffleMapTask.scala:53)
	at org.apache.spark.scheduler.Task.run(Task.scala:99)
	at org.apache.spark.executor.Executor$TaskRunner.run(Executor.scala:282)
	at java.util.concurrent.ThreadPoolExecutor.runWorker(ThreadPoolExecutor.java:1145)
	at java.util.concurrent.ThreadPoolExecutor$Worker.run(ThreadPoolExecutor.java:615)
	at java.lang.Thread.run(Thread.java:745)
17/07/27 18:00:53 WARN TaskSetManager: Lost task 4.0 in stage 28.0 (TID 16828, localhost,
executor driver): java.io.FileNotFoundException: /tmp/blockmgr-17538d4a-76fb-43aa-b086-
c5efbd96fc23/17/temp_shuffle_de7b22e3-156e-447d-9318-412db829857a (Too many open
files)
	at java.io.FileOutputStream.open(Native Method)
	at java.io.FileOutputStream.<init>(FileOutputStream.java:221)
	at
org.apache.spark.storage.DiskBlockObjectWriter.initialize(DiskBlockObjectWriter.scala:102)
```

```
	at
org.apache.spark.storage.DiskBlockObjectWriter.open(DiskBlockObjectWriter.scala:115)
	at
org.apache.spark.storage.DiskBlockObjectWriter.write(DiskBlockObjectWriter.scala:229)
	at
org.apache.spark.shuffle.sort.BypassMergeSortShuffleWriter.write(BypassMergeSortShuffleWri
ter.java:152)
	at org.apache.spark.scheduler.ShuffleMapTask.runTask(ShuffleMapTask.scala:96)
	at org.apache.spark.scheduler.ShuffleMapTask.runTask(ShuffleMapTask.scala:53)
	at org.apache.spark.scheduler.Task.run(Task.scala:99)
	at org.apache.spark.executor.Executor$TaskRunner.run(Executor.scala:282)
	at java.util.concurrent.ThreadPoolExecutor.runWorker(ThreadPoolExecutor.java:1145)
	at java.util.concurrent.ThreadPoolExecutor$Worker.run(ThreadPoolExecutor.java:615)
	at java.lang.Thread.run(Thread.java:745)

17/07/27 18:00:53 ERROR TaskSetManager: Task 4 in stage 28.0 failed 1 times; aborting job
[Stage 28:>                                                 (0 + 25) / 200]17/07/27 18:00:53 WARN
TaskSetManager: Lost task 10.0 in stage 28.0 (TID 16834, localhost, executor driver):
java.io.FileNotFoundException: /tmp/blockmgr-17538d4a-76fb-43aa-b086-
c5efbd96fc23/22/temp_shuffle_e42ed550-23b9-4909-9ba8-ae7a455ab862 (Too many open files)
	at java.io.FileOutputStream.open(Native Method)
	at java.io.FileOutputStream.<init>(FileOutputStream.java:221)
	at
org.apache.spark.storage.DiskBlockObjectWriter.initialize(DiskBlockObjectWriter.scala:102)
	at
org.apache.spark.storage.DiskBlockObjectWriter.open(DiskBlockObjectWriter.scala:115)
	at
org.apache.spark.storage.DiskBlockObjectWriter.write(DiskBlockObjectWriter.scala:229)
	at
org.apache.spark.shuffle.sort.BypassMergeSortShuffleWriter.write(BypassMergeSortShuffleWri
ter.java:152)
	at org.apache.spark.scheduler.ShuffleMapTask.runTask(ShuffleMapTask.scala:96)
	at org.apache.spark.scheduler.ShuffleMapTask.runTask(ShuffleMapTask.scala:53)
	at org.apache.spark.scheduler.Task.run(Task.scala:99)
	at org.apache.spark.executor.Executor$TaskRunner.run(Executor.scala:282)
	at java.util.concurrent.ThreadPoolExecutor.runWorker(ThreadPoolExecutor.java:1145)
	at java.util.concurrent.ThreadPoolExecutor$Worker.run(ThreadPoolExecutor.java:615)
	at java.lang.Thread.run(Thread.java:745)

17/07/27 18:00:53 WARN TaskSetManager: Lost task 8.0 in stage 28.0 (TID 16832, localhost,
executor driver): TaskKilled (killed intentionally)
17/07/27 18:00:53 WARN TaskSetManager: Lost task 11.0 in stage 28.0 (TID 16835, localhost,
executor driver): TaskKilled (killed intentionally)
17/07/27 18:00:53 WARN TaskSetManager: Lost task 24.0 in stage 28.0 (TID 16848, localhost,
executor driver): TaskKilled (killed intentionally)
```

17/07/27 18:00:53 WARN TaskSetManager: Lost task 9.0 in stage 28.0 (TID 16833, localhost, executor driver): TaskKilled (killed intentionally)
17/07/27 18:00:53 WARN TaskSetManager: Lost task 1.0 in stage 28.0 (TID 16825, localhost, executor driver): TaskKilled (killed intentionally)
17/07/27 18:00:53 WARN TaskSetManager: Lost task 12.0 in stage 28.0 (TID 16836, localhost, executor driver): TaskKilled (killed intentionally)
17/07/27 18:00:54 WARN TaskSetManager: Lost task 3.0 in stage 28.0 (TID 16827, localhost, executor driver): TaskKilled (killed intentionally)
17/07/27 18:00:54 WARN TaskSetManager: Lost task 0.0 in stage 28.0 (TID 16824, localhost, executor driver): TaskKilled (killed intentionally)
17/07/27 18:00:54 WARN TaskSetManager: Lost task 2.0 in stage 28.0 (TID 16826, localhost, executor driver): TaskKilled (killed intentionally)
17/07/27 18:00:54 WARN TaskSetManager: Lost task 19.0 in stage 28.0 (TID 16843, localhost, executor driver): TaskKilled (killed intentionally)
17/07/27 18:00:54 WARN TaskSetManager: Lost task 22.0 in stage 28.0 (TID 16846, localhost, executor driver): TaskKilled (killed intentionally)
17/07/27 18:00:54 WARN TaskSetManager: Lost task 6.0 in stage 28.0 (TID 16830, localhost, executor driver): TaskKilled (killed intentionally)
17/07/27 18:00:54 WARN TaskSetManager: Lost task 18.0 in stage 28.0 (TID 16842, localhost, executor driver): TaskKilled (killed intentionally)
17/07/27 18:00:54 WARN TaskSetManager: Lost task 14.0 in stage 28.0 (TID 16838, localhost, executor driver): TaskKilled (killed intentionally)
17/07/27 18:00:54 WARN TaskSetManager: Lost task 21.0 in stage 28.0 (TID 16845, localhost, executor driver): TaskKilled (killed intentionally)
17/07/27 18:00:54 WARN TaskSetManager: Lost task 16.0 in stage 28.0 (TID 16840, localhost, executor driver): TaskKilled (killed intentionally)
17/07/27 18:00:54 WARN TaskSetManager: Lost task 15.0 in stage 28.0 (TID 16839, localhost, executor driver): TaskKilled (killed intentionally)
17/07/27 18:00:54 WARN TaskSetManager: Lost task 13.0 in stage 28.0 (TID 16837, localhost, executor driver): TaskKilled (killed intentionally)
17/07/27 18:00:54 WARN TaskSetManager: Lost task 7.0 in stage 28.0 (TID 16831, localhost, executor driver): TaskKilled (killed intentionally)
17/07/27 18:00:54 WARN TaskSetManager: Lost task 17.0 in stage 28.0 (TID 16841, localhost, executor driver): TaskKilled (killed intentionally)
17/07/27 18:00:54 WARN TaskSetManager: Lost task 23.0 in stage 28.0 (TID 16847, localhost, executor driver): TaskKilled (killed intentionally)
17/07/27 18:00:54 WARN TaskSetManager: Lost task 5.0 in stage 28.0 (TID 16829, localhost, executor driver): TaskKilled (killed intentionally)
org.apache.spark.SparkException: Job aborted due to stage failure: Task 4 in stage 28.0 failed 1 times, most recent failure: Lost task 4.0 in stage 28.0 (TID 16828, localhost, executor driver): java.io.FileNotFoundException: /tmp/blockmgr-17538d4a-76fb-43aa-b086-c5efbd96fc23/17/temp_shuffle_de7b22e3-156e-447d-9318-412db829857a (Too many open files)
    at java.io.FileOutputStream.open(Native Method)
    at java.io.FileOutputStream.<init>(FileOutputStream.java:221)

```
        at
org.apache.spark.storage.DiskBlockObjectWriter.initialize(DiskBlockObjectWriter.scala:102)
        at
org.apache.spark.storage.DiskBlockObjectWriter.open(DiskBlockObjectWriter.scala:115)
        at
org.apache.spark.storage.DiskBlockObjectWriter.write(DiskBlockObjectWriter.scala:229)
        at
org.apache.spark.shuffle.sort.BypassMergeSortShuffleWriter.write(BypassMergeSortShuffleWri
ter.java:152)
        at org.apache.spark.scheduler.ShuffleMapTask.runTask(ShuffleMapTask.scala:96)
        at org.apache.spark.scheduler.ShuffleMapTask.runTask(ShuffleMapTask.scala:53)
        at org.apache.spark.scheduler.Task.run(Task.scala:99)
        at org.apache.spark.executor.Executor$TaskRunner.run(Executor.scala:282)
        at java.util.concurrent.ThreadPoolExecutor.runWorker(ThreadPoolExecutor.java:1145)
        at java.util.concurrent.ThreadPoolExecutor$Worker.run(ThreadPoolExecutor.java:615)
        at java.lang.Thread.run(Thread.java:745)

Driver stacktrace:
  at
org.apache.spark.scheduler.DAGScheduler.org$apache$spark$scheduler$DAGScheduler$$failJ
obAndIndependentStages(DAGScheduler.scala:1435)
  at
org.apache.spark.scheduler.DAGScheduler$$anonfun$abortStage$1.apply(DAGScheduler.scala:
1423)
  at
org.apache.spark.scheduler.DAGScheduler$$anonfun$abortStage$1.apply(DAGScheduler.scala:
1422)
  at scala.collection.mutable.ResizableArray$class.foreach(ResizableArray.scala:59)
  at scala.collection.mutable.ArrayBuffer.foreach(ArrayBuffer.scala:48)
  at org.apache.spark.scheduler.DAGScheduler.abortStage(DAGScheduler.scala:1422)
  at
org.apache.spark.scheduler.DAGScheduler$$anonfun$handleTaskSetFailed$1.apply(DAGSched
uler.scala:802)
  at
org.apache.spark.scheduler.DAGScheduler$$anonfun$handleTaskSetFailed$1.apply(DAGSched
uler.scala:802)
  at scala.Option.foreach(Option.scala:257)
  at org.apache.spark.scheduler.DAGScheduler.handleTaskSetFailed(DAGScheduler.scala:802)
  at
org.apache.spark.scheduler.DAGSchedulerEventProcessLoop.doOnReceive(DAGScheduler.scal
a:1650)
  at
org.apache.spark.scheduler.DAGSchedulerEventProcessLoop.onReceive(DAGScheduler.scala:1
605)
```

```
  at
org.apache.spark.scheduler.DAGSchedulerEventProcessLoop.onReceive(DAGScheduler.scala:1
594)
  at org.apache.spark.util.EventLoop$$anon$1.run(EventLoop.scala:48)
  at org.apache.spark.scheduler.DAGScheduler.runJob(DAGScheduler.scala:628)
  at org.apache.spark.SparkContext.runJob(SparkContext.scala:1918)
  at org.apache.spark.SparkContext.runJob(SparkContext.scala:1931)
  at org.apache.spark.SparkContext.runJob(SparkContext.scala:1944)
  at org.apache.spark.sql.execution.SparkPlan.executeTake(SparkPlan.scala:333)
  at org.apache.spark.sql.execution.CollectLimitExec.executeCollect(limit.scala:38)
  at
org.apache.spark.sql.Dataset$$anonfun$org$apache$spark$sql$Dataset$$execute$1$1.apply(Dat
aset.scala:2371)
  at
org.apache.spark.sql.execution.SQLExecution$.withNewExecutionId(SQLExecution.scala:57)
  at org.apache.spark.sql.Dataset.withNewExecutionId(Dataset.scala:2765)
  at org.apache.spark.sql.Dataset.org$apache$spark$sql$Dataset$$execute$1(Dataset.scala:2370)
  at org.apache.spark.sql.Dataset.org$apache$spark$sql$Dataset$$collect(Dataset.scala:2377)
  at org.apache.spark.sql.Dataset$$anonfun$head$1.apply(Dataset.scala:2113)
  at org.apache.spark.sql.Dataset$$anonfun$head$1.apply(Dataset.scala:2112)
  at org.apache.spark.sql.Dataset.withTypedCallback(Dataset.scala:2795)
  at org.apache.spark.sql.Dataset.head(Dataset.scala:2112)
  at org.apache.spark.sql.Dataset.take(Dataset.scala:2327)
  at org.apache.spark.sql.Dataset.showString(Dataset.scala:248)
  at org.apache.spark.sql.Dataset.show(Dataset.scala:636)
  at org.apache.spark.sql.Dataset.show(Dataset.scala:595)
  ... 48 elided
Caused by: java.io.FileNotFoundException: /tmp/blockmgr-17538d4a-76fb-43aa-b086-
c5efbd96fc23/17/temp_shuffle_de7b22e3-156e-447d-9318-412db829857a (Too many open
files)
  at java.io.FileOutputStream.open(Native Method)
  at java.io.FileOutputStream.<init>(FileOutputStream.java:221)
  at
org.apache.spark.storage.DiskBlockObjectWriter.initialize(DiskBlockObjectWriter.scala:102)
  at org.apache.spark.storage.DiskBlockObjectWriter.open(DiskBlockObjectWriter.scala:115)
  at org.apache.spark.storage.DiskBlockObjectWriter.write(DiskBlockObjectWriter.scala:229)
  at
org.apache.spark.shuffle.sort.BypassMergeSortShuffleWriter.write(BypassMergeSortShuffleWri
ter.java:152)
  at org.apache.spark.scheduler.ShuffleMapTask.runTask(ShuffleMapTask.scala:96)
  at org.apache.spark.scheduler.ShuffleMapTask.runTask(ShuffleMapTask.scala:53)
  at org.apache.spark.scheduler.Task.run(Task.scala:99)
  at org.apache.spark.executor.Executor$TaskRunner.run(Executor.scala:282)
  at java.util.concurrent.ThreadPoolExecutor.runWorker(ThreadPoolExecutor.java:1145)
  at java.util.concurrent.ThreadPoolExecutor$Worker.run(ThreadPoolExecutor.java:615)
  at java.lang.Thread.run(Thread.java:745)
```

scala> 17/07/27 18:00:54 WARN TaskSetManager: Lost task 20.0 in stage 28.0 (TID 16844, localhost, executor driver): TaskKilled (killed intentionally)

Appendix 5: Source Code

5.1: Python Code

```
#################################################################
## TfL Station Matching ##
#################################################################


# The code below is used to match each station key with its corresponding name

import pandas as pd
import numpy as np

# Read in the necessary input files

station_entry_df = pd.read_csv("StationsOfEntry_CK1.csv")     # file containing all station of
entry keys
station_exit_df = pd.read_csv("StationsOfExit_CK1.csv")       # file containing all station of exit
keys
stationdic_df = pd.read_csv("StationsDictionary_CK1.csv")     # file containing station keys and
their corresponding names

# Match each station of entry key with its corresponding name using the dictionary of stations

count = 0
dic_count = 0
station_name = 'not found'

while count < len(station_entry_df):
    dic_count = 0
    station_name = 'not found'
    while (dic_count < len(stationdic_df) and station_name == 'not found'):
        if station_entry_df['stationoffirstentrykey'][count] ==
stationdic_df['STATIONKEY'][dic_count]:
            station_entry_df['Station of Entry'][count] = stationdic_df['STATIONNAME'][dic_count]
            station_name = 'found'
        dic_count = dic_count + 1
    count = count + 1

# Show station_entry_df dataframe

station_entry_df

# Save the station_entry_df dataframe to an output file
```

```
station_entry_df.to_csv("StationsOfEntryNEW_CK1.csv", sep = '\t')

# Match each station of exit key with its corresponding name using the dictionary of stations

count = 0
dic_count = 0
station_name = 'not found'

while count < len(station_exit_df):
    dic_count = 0
    station_name = 'not found'
    while (dic_count < len(stationdic_df) and station_name == 'not found'):
        if station_exit_df['stationofexitkey'][count] == stationdic_df['STATIONKEY'][dic_count]:
            station_exit_df['Station of Exit'][count] = stationdic_df['STATIONNAME'][dic_count]
            station_name = 'found'
        dic_count = dic_count + 1
    count = count + 1

# Show station_exit_df dataframe

station_exit_df

# Save station_exit_df dataframe as an output file

station_exit_df.to_csv("StationsOfExitNEW_CK1.csv", sep = '\t')


#################################################################
## Geospatial Matching ##
#################################################################


# The code below is used to match each TfL station with its respective Supergroup label based on the station's geographic
# coordinates.

%matplotlib inline
import pandas as pd
import numpy as np
import fiona
import shapely.geometry
from shapely.geometry import Point, shape
import geopandas as gpd
import pylab
from geopy.geocoders import Nominatim
from geopy.exc import GeocoderTimedOut
```

```
import pyproj


# Read in the input files

stationdic_df = pd.read_csv("StationsDictionary_CK4.csv")     # Read in a file containing the keys and the corresponding names of all the stations and stops in the TfL dataset (including TfL tube stations, bus stops, and tram stations)
stationlist_df = pd.read_csv("TfL_Stations.csv")      # Read in a file containing the names of only unique TfL tube stations

# Create a new column in stationdic_df called 'Keep' and set the default value as 'False'. This is to be used to determine which
# stations in the TfL dataset refer to tube stations and will be used to filter out journeys taken on buses, trams, and other non-tube and non-rail services.

stationdic_df['Keep'] = 'False'

# Match each station in stationdic_df with the actual list of tube and rail stations in stationlist_df (so that I can later delete the stations in stationdic_df that are not tube and rail stations)

count = 0
true_count = 0

while (count < len(stationdic_df)):
    list_count = 0
    while (list_count < len(stationlist_df) and (stationdic_df['Keep'][count] != 'True')):
        if stationdic_df['STATIONNAME'][count] == stationlist_df['Station'][list_count]:
            stationdic_df['Keep'][count] = 'True'
            true_count = true_count + 1
        list_count = list_count + 1
    count = count + 1


# Delete all rows where 'Keep' = 'False' (essentially, I'm only keeping those stations which were matched up with the actual list of tube and rail stations)

stationdic_df = stationdic_df[stationdic_df.Keep != 'False']

stationdic_df.head()

# Re-start the numbering of the index for stationdic_df (without creating a new column for it)

stationdic_df = stationdic_df.reset_index(drop=True)

stationdic_df.head()
```

```
# Create a new column called 'Supergroup' in stationdic_df and set the default value to 0.

stationdic_df['Supergroup'] = 0

# Add the string 'Station, London' to the end of each station name so that it is more easily
searcheable using the geopandas library.

stationdic_df['STATIONNAME'] = stationdic_df['STATIONNAME'].astype(str) + ' Station,
England'

stationdic_df.head()

# Read in the LOAC shapefile using geopandas to enable the creation of a visualization of the
output areas in the LOAC dataset.

loac_map = gpd.read_file("Shapefiles/LOAC_London.shp")

# Create a visualization of the 25,000 output areas in Greater London.

loac_map.plot(figsize=(30,20))

# Save the above visualization to an output file

pylab.savefig("LOAC_Overview")

# Now, zoom in to create a visualization of the output areas surrounding the Strand area in
Central London.

ax = loac_map.plot(figsize=(30,20))
ax.set_xlim(530000,535000)
ax.set_ylim(180000,185000)


# Save the above visualization to an output file

pylab.savefig("LOAC_ZoomedIn")

# Need this for geocoding

geocoder = Nominatim()

# Now read in the same LOAC shapefile containing the Supergroup label for each of the 25,000
output areas in Greater London
# using the fiona library
```

```
loac = fiona.open('Shapefiles/LOAC_London.shp', 'r')

# Need this for interpreting different coordinate systems

bng = pyproj.Proj(init='epsg:27700')
wgs84 = pyproj.Proj(init='epsg:4326')

# This assigns a Supergroup code from the LOAC dataset to each tube and rail station in the TfL
database

counter = 0

while (counter < len(stationdic_df)):
   try:
      station_of_entry = geocoder.geocode(stationdic_df['STATIONNAME'][counter])
      x1, y1 = pyproj.transform(wgs84,bng,station_of_entry.longitude,station_of_entry.latitude)
      station_of_entry_pnt = Point(x1,y1)
      q1 = [w for w in loac if station_of_entry_pnt.within(shape(w["geometry"]))]
      if len(q1) != 0:
         stationdic_df['Supergroup'][counter] = q1[0]["properties"]['supgrp_cd']
   except GeocoderTimedOut as e:
      print ("Error: geocode failed on input %s with message %s"
%(stationdic_df['STATIONNAME'][counter], e.msg))
   counter = counter + 1


# Check to see whether the stations were assigned a Supergroup label

stationdic_df

# Save the new stationdic_df with the assigned Supergroup codes to a new output file

stationdic_df.to_csv("StationswithLOACcode_CK1.csv", sep = '\t')


#################################################################
## TfL Passenger Profiles ##
#################################################################


# The code below is used to create travel profiles for two passengers. These profiles contain the
frequency distribution
# of their stations of entry and the frequency distribution of their travel times when using their
most frequently used
# station of entry.
```

```
import pandas as pd
import numpy as np
import math
import matplotlib.pyplot as plt
import pylab


# Read in a file containing all station keys and their corresponding names

stationdic_df = pd.read_csv("StationsDictionary_CK2.csv")

# Show the head of the stationdic_df dataframe

stationdic_df.head()

# Read in the files containing journey data for each passenger

user1_df = pd.read_csv("Traveller Matrix_User1_CK6.csv")
user2_df = pd.read_csv("Traveller Matrix_User3_CK3.csv")

# Show the head of the dataframe containing journey data for the first passenger

user1_df.head()

# Show the frequency distribution of the stations of entry used by Passenger 1

user1_df['stationoffirstentrykey'].value_counts()

# Assign a timeblock to each value of 'transactiontime' so that each transaction time falls into a
2-hour time interval
# throughout a 24-hour day

# Source: https://stackoverflow.com/questions/26886653/pandas-create-new-column-based-on-
values-from-other-columns?noredirect=1&lq=1

def timeblock(row):
    if (row['transactiontime'] >= 0 and row['transactiontime'] < 120):
        return 1
    elif (row['transactiontime'] >= 120 and row['transactiontime'] < 240):
        return 2
    elif (row['transactiontime'] >= 240 and row['transactiontime'] < 360):
        return 3
    elif (row['transactiontime'] >= 360 and row['transactiontime'] < 480):
        return 4
    elif (row['transactiontime'] >= 480 and row['transactiontime'] < 600):
        return 5
```

```
    elif (row['transactiontime'] >= 600 and row['transactiontime'] < 720):
        return 6
    elif (row['transactiontime'] >= 720 and row['transactiontime'] < 840):
        return 7
    elif (row['transactiontime'] >= 840 and row['transactiontime'] < 960):
        return 8
    elif (row['transactiontime'] >= 960 and row['transactiontime'] < 1080):
        return 9
    elif (row['transactiontime'] >= 1080 and row['transactiontime'] < 1200):
        return 10
    elif (row['transactiontime'] >= 1200 and row['transactiontime'] < 1320):
        return 11
    elif (row['transactiontime'] >= 1320 and row['transactiontime'] < 1440):
        return 12
    else:
        return 99      # assign the value 99 for transaction times that fall outside the 24-hour window


user1_df['Timeblock'] = user1_df.apply(lambda row: timeblock (row), axis=1)


# Show the head of the user1_df dataframe to check whether timeblocks were assigned correctly

user1_df.head()

# We now obtain a dataset of all journeys for Passenger 1 which commenced at her most
frequently used station of entry

user1_df_filtered = user1_df.query('stationoffirstentrykey==1349')

# Show the head of the user1_df_filtered dataframe

user1_df_filtered.head()

# Show Passenger 1's frequency distribution of her travel times when using her most frequently
used station of entry.

user1_df_filtered['Timeblock'].value_counts()

# Show the head of the dataframe containing journey data for the second passenger

user2_df.head()

# Show the frequency distribution of the stations of entry used by Passenger 2

user2_df['stationoffirstentrykey'].value_counts()
```

```
# Assign a timeblock to each value of 'transactiontime' so that each transaction time falls into a 2-hour time interval
# throughout a 24-hour day

# Source: https://stackoverflow.com/questions/26886653/pandas-create-new-column-based-on-values-from-other-columns?noredirect=1&lq=1

def timeblock(row):
    if (row['transactiontime'] >= 0 and row['transactiontime'] < 120):
        return 1
    elif (row['transactiontime'] >= 120 and row['transactiontime'] < 240):
        return 2
    elif (row['transactiontime'] >= 240 and row['transactiontime'] < 360):
        return 3
    elif (row['transactiontime'] >= 360 and row['transactiontime'] < 480):
        return 4
    elif (row['transactiontime'] >= 480 and row['transactiontime'] < 600):
        return 5
    elif (row['transactiontime'] >= 600 and row['transactiontime'] < 720):
        return 6
    elif (row['transactiontime'] >= 720 and row['transactiontime'] < 840):
        return 7
    elif (row['transactiontime'] >= 840 and row['transactiontime'] < 960):
        return 8
    elif (row['transactiontime'] >= 960 and row['transactiontime'] < 1080):
        return 9
    elif (row['transactiontime'] >= 1080 and row['transactiontime'] < 1200):
        return 10
    elif (row['transactiontime'] >= 1200 and row['transactiontime'] < 1320):
        return 11
    elif (row['transactiontime'] >= 1320 and row['transactiontime'] < 1440):
        return 12
    else:
        return 99      # assign the value 99 for transaction times that fall outside the 24-hour window


user2_df['Timeblock'] = user2_df.apply(lambda row: timeblock (row), axis=1)


# Show the head of the user2_df dataframe to check whether timeblocks were assigned correctly

user2_df.head()

# We now obtain a dataset of all journeys for Passenger 2 which commenced at his most frequently used station of entry
```

```
user2_df_filtered = user2_df.query('stationoffirstentrykey==187')

# Show the head of the user2_df_filtered dataframe

user2_df_filtered.head()

# Show Passenger 2's frequency distribution of his travel times when using his most frequently used station of entry.

user2_df_filtered['Timeblock'].value_counts()


######################################################################
## Sample Dataset Analysis
######################################################################

# The code in this file is used to analyze the travel patterns of passengers using the following steps:

# Step 1: Read in the sample data on TfL passenger journeys
# Step 2: Clean and prepare the data so that the time and the date on which each journey takes place is recoded to facilitate analysis
# Step 3: Create a dataframe of unique passengers and their respective most frequently used station of entry
# Step 4: Assign each passenger his most likely socio-demographic characteristics based on his
# approximated place of residence (proxied by his most frequently used station of entry)
# Step 5: Conduct cluster analysis on this dataset of unique passengers and their corresponding socio-demographic characteristics
# Step 6: Obtain information on the travel patterns of each cluster, including the most frequently used stations,
# the distribution of travel throughout a day, and the distribution of travel throughout a week.

import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import matplotlib
import patsy
import statsmodels.formula.api as sm

#############
## Step 1:
#############

# Read in the sample data (they are in parts because creating csv files for the entire sample at once took too much memory)
```

```
# These files were obtained by getting a random sample of records for each day in the Spark database

sample1_df = pd.read_csv("Full_Sample_Part1.csv")
sample2_df = pd.read_csv("Full_Sample_Part2.csv")
sample3_df = pd.read_csv("Full_Sample_Part3.csv")
sample4_df = pd.read_csv("Full_Sample_Part4.csv")
sample5_df = pd.read_csv("Full_Sample_Part5.csv")
sample6_df = pd.read_csv("Full_Sample_Part6.csv")
sample7_df = pd.read_csv("Full_Sample_Part7.csv")
sample8_df = pd.read_csv("Full_Sample_Part8.csv")

# Concatenate all datafromes into one large dataframe which contains all randomly sampled records

frames = [sample1_df, sample2_df, sample3_df, sample4_df, sample5_df, sample6_df, sample7_df, sample8_df]

final_df = pd.concat(frames)

# Show the head of the final_df dataframe

final_df.head()

##########
## Step 2:
##########

# Create a new column called 'Timeblock' and assign null values for the timebeing.

final_df['Timeblock'] = np.nan

# Assign a timeblock to each value of TRANSACTIONTIME so that each falls into a 2-hour time interval throughout a 24-hour day

# Source: https://stackoverflow.com/questions/26886653/pandas-create-new-column-based-on-values-from-other-columns?noredirect=1&lq=1

def timeblock(row):
    if (row['TRANSACTIONTIME'] >= 0 and row['TRANSACTIONTIME'] < 120):
        return 1
    elif (row['TRANSACTIONTIME'] >= 120 and row['TRANSACTIONTIME'] < 240):
        return 2
    elif (row['TRANSACTIONTIME'] >= 240 and row['TRANSACTIONTIME'] < 360):
        return 3
    elif (row['TRANSACTIONTIME'] >= 360 and row['TRANSACTIONTIME'] < 480):
```

```
        return 4
    elif (row['TRANSACTIONTIME'] >= 480 and row['TRANSACTIONTIME'] < 600):
        return 5
    elif (row['TRANSACTIONTIME'] >= 600 and row['TRANSACTIONTIME'] < 720):
        return 6
    elif (row['TRANSACTIONTIME'] >= 720 and row['TRANSACTIONTIME'] < 840):
        return 7
    elif (row['TRANSACTIONTIME'] >= 840 and row['TRANSACTIONTIME'] < 960):
        return 8
    elif (row['TRANSACTIONTIME'] >= 960 and row['TRANSACTIONTIME'] < 1080):
        return 9
    elif (row['TRANSACTIONTIME'] >= 1080 and row['TRANSACTIONTIME'] < 1200):
        return 10
    elif (row['TRANSACTIONTIME'] >= 1200 and row['TRANSACTIONTIME'] < 1320):
        return 11
    elif (row['TRANSACTIONTIME'] >= 1320 and row['TRANSACTIONTIME'] < 1440):
        return 12
    else:
        return 99      # assign the value 99 for transaction times that fall outside the 24-hour window


final_df['Timeblock'] = final_df.apply(lambda row: timeblock (row), axis=1)


# Show the head of the final_df dataframe after assigning a timeblock to each transaction time

final_df.head()

# Create an array of the unique days in the dataset

day_array = final_df['DAYKEY'].unique()

day_array

# Check if there are any null values in the 'DAYKEY' column

final_df['DAYKEY'].isnull().sum()

# Match each daykey with its corresponding day of the week

columns = ['DAYKEY', 'Day of the Week']

index = np.arange(56)

day_df = pd.DataFrame(columns = columns, index = index)
```

```
count = 0

while count < len(day_array):
    if (day_array[count] == 12347 or day_array[count] == 12354 or day_array[count] ==  12361
or day_array[count] ==  12368 or day_array[count] == 12375 or day_array[count] == 12382 or
day_array[count] == 12389 or day_array[count] == 12396):
        day_df['DAYKEY'][count] = day_array[count]
        day_df['Day of the Week'][count] = 'Monday'
    elif (day_array[count] == 12348 or day_array[count] == 12355 or day_array[count] == 12362
or day_array[count] == 12369 or day_array[count] == 12376 or day_array[count] == 12383 or
day_array[count] == 12390 or day_array[count] == 12397):
        day_df['DAYKEY'][count] = day_array[count]
        day_df['Day of the Week'][count] = 'Tuesday'
    elif (day_array[count] == 12349 or day_array[count] == 12356 or day_array[count] == 12363
or day_array[count] == 12370 or day_array[count] == 12377 or day_array[count] == 12384 or
day_array[count] == 12391 or day_array[count] == 12398):
        day_df['DAYKEY'][count] = day_array[count]
        day_df['Day of the Week'][count] = 'Wednesday'
    elif (day_array[count] == 12350 or day_array[count] == 12357 or day_array[count] == 12364
or day_array[count] == 12371 or day_array[count] == 12378 or day_array[count] == 12385 or
day_array[count] == 12392 or day_array[count] == 12399):
        day_df['DAYKEY'][count] = day_array[count]
        day_df['Day of the Week'][count] = 'Thursday'
    elif (day_array[count] == 12351 or day_array[count] == 12358 or day_array[count] == 12365
or day_array[count] == 12372 or day_array[count] == 12379 or day_array[count] == 12386 or
day_array[count] == 12393 or day_array[count] == 12400):
        day_df['DAYKEY'][count] = day_array[count]
        day_df['Day of the Week'][count] = 'Friday'
    elif (day_array[count] == 12345 or day_array[count] == 12352 or day_array[count] == 12359
or day_array[count] == 12366 or day_array[count] == 12373 or day_array[count] == 12380 or
day_array[count] == 12387 or day_array[count] == 12394 or day_array[count] == 12401):
        day_df['DAYKEY'][count] = day_array[count]
        day_df['Day of the Week'][count] = 'Saturday'
    else:
        day_df['DAYKEY'][count] = day_array[count]
        day_df['Day of the Week'][count] = 'Sunday'
    count = count + 1


# Show the day_df dataframe

day_df

# Merge the final_df dataframe with the day_df dataframe so that you assign days of the week
(e.g. Monday, Tuesday, etc.)
# to each value of DAYKEY in the final_df dataframe
```

```
finalmerged_df = final_df.merge(day_df, left_on='DAYKEY', right_on='DAYKEY', how='left')

# Show the head of the finalmerged_df dataframe

finalmerged_df.head()

##########
## Step 3:
##########

# Find the most frequently used station of entry for each prestigeid:
# The code below does a 'groupby' on PRESTIGEID for the STATIONOFFIRSTENTRYKEY column, gets a count for the values in
# each group in descending order, and then takes the first row for each group.

# SOURCE: https://stackoverflow.com/questions/23692419/python-select-most-frequent-using-group-by

station_mode = finalmerged_df.groupby('PRESTIGEID')['STATIONOFFIRSTENTRYKEY'].agg(lambda x: x.value_counts().index[0])

# Convert the station_mode series to a dataframe.

passenger_df = pd.DataFrame({'PRESTIGEID': station_mode.index, 'STATIONOFFIRSTENTRYKEY': station_mode.values})

# Show the head of the passenger_df dataframe

passenger_df.head()

###########
## Step 4:
###########

# Read in a file which contains the Supergroup code for each TfL tube and rail station (prepared in a separate Jupyter Notebook file)

loac_input_df = pd.read_csv("Stations with LOAC Code_FINAL.csv")

# Show the head of the loac_input_df dataframe

loac_input_df.head()
```

```
# Delete the string 'Station, London' from the end of each Station Name because it is no longer
needed.

import re

count = 0

while count < len(loac_input_df):
    loac_input_df['STATIONNAME'][count] = re.sub('\ Station, England$', '',
loac_input_df['STATIONNAME'][count])
    count = count + 1


# Merge the passengerfinal_df dataframe with the loac_input_df dataframe so that you get a
dataframe with unique passengers and
# their corresponding Supergroup code based on their most frequently used station of entry

passengerfinal_df = passenger_df.merge(loac_input_df,
left_on='STATIONOFFIRSTENTRYKEY', right_on='STATIONKEY', how='left')

# Show the head of the passengerfinal_df dataframe, which contains all unique passenger id's
(PRESTIGEID's) and their
# respective most frequently used station of entry and corresponding Supergroup code

passengerfinal_df.head()

# Check if there are any null values

passengerfinal_df.isnull().values.any()

# Read in the CSV file with the socio-economic traits for each Supergroup

traits_df = pd.read_csv("LOAC_SocEc_Traits_CK2.csv")

# Show the head of the traits_df dataframe

traits_df.head()

# Merge the passengerfinal_df with the traits_df so that we assign socio-economic characteristics
corresponding to each
# Supergroup code for each passenger

passengerfinal_df2 = passengerfinal_df.merge(traits_df, left_on='Supergroup',
right_on='Supergroup', how='left')

# Show the head of the passengerfinal_df2 dataframe
```

```
passengerfinal_df2.head()

# Check the frequency distribution of the 'Supergroup' column (in terms of proportion)

passengerfinal_df2.Supergroup.value_counts(normalize=True)


##########
## Step 5:
##########


import sklearn
import sklearn.cluster as cluster
from sklearn.cluster import KMeans
from sklearn import metrics

# This deletes non-essential columns from the passenger_df dataframe and prepares it for k-means clustering

passengerfinal_df_clustering = passengerfinal_df2.drop('STATIONNAME', axis=1)
passengerfinal_df_clustering = passengerfinal_df_clustering.drop('Keep', axis=1)

# Show the head of the passengerfinal_df_clustering dataframe

passengerfinal_df_clustering.head()

# Check if the new dataframe has any null values

passengerfinal_df_clustering.isnull().values.any()

# Check which columns have null values and how many each column has

passengerfinal_df_clustering.isnull().sum()

# Drop rows with null values, since the null values relate to those records in which a Supergroup code was not assigned
# (these rows are designated by the Supergroup code "X"), and as such, these records in which a Supergroup code was not
# assigned are no longer relevant for the next stage of our analysis

passengerfinal_df_clustering = passengerfinal_df_clustering.dropna()

# Based on sd-clean.py by Elizabeth Sklar (28-mar-2017)
```

```
# Input the passengerfinal_df_clustering dataframe into a list in preparation for the k-means clustering


import sys
import csv

passengerfinal_df_clustering.to_csv("passengerfinal_df_clustering.csv", sep = ',')

try:
# open data file in csv format
    f = open( 'passengerfinal_df_clustering.csv', 'rU' )
# read contents of data file into "rawdata" list
    indata = csv.reader( f )
# parse data in csv format
    rawdata = [rec for rec in indata]
# handle exceptions:
except IOError as iox:
    print '** I/O error trying to open the data file> ' + str( iox )
    sys.exit()
except Exception as x:
    print '** error> ' + str( x )
    sys.exit()

# Based on sd-clean.py by Elizabeth Sklar (28-mar-2017)

# The code below handles the header in the input file by printing it and then deleting it from the data file

header = rawdata[0]
del rawdata[0]
print 'header='
print header
print( 'number of features = %d' % len( header ) )
print 'features:'
for i, f in zip( range(len(header)), header ):
    print( 'index=%d  feature=[%s]' % ( i, f ))

# The following cells conduct cluster analysis on each set of related features

# Features related to Age:

features = [7, 8, 9, 10, 11, 12]

num_features = len(features)
```

```
label = 5

#-gather fields of interest from data set into X and y
X = []
y = []

for rec in rawdata:
   instance = []
   for f in features:
      instance.append( float( rec[f] ))
   X.append( instance )       # Get each record
   y.append( rec[label] )     # Get the corresponding Supergroup label of each record

X = np.array( X )
y = np.array( y )

num_instances = X.shape[0]

print 'number of instances = %d' % num_instances
print 'shape of input data = %d x %d' % ( X.shape[0], X.shape[1] )
print 'shape of target data = %d' % ( y.shape[0] )


# Create a dataframe with the results of the Inertia and Calinski-Harabasz scores


columns = ['No. of clusters', 'Inertia', 'Calinski-Harabaz Score']

index = np.arange(14)

df1 = pd.DataFrame(columns = columns, index = index)

line_count = 0

n_clusters = [2, 3, 4, 5, 6, 7, 8, 9, 10, 20, 30, 40, 50, 100]

cluster_count = 0

while cluster_count < 14:
   km = cluster.KMeans( n_clusters[cluster_count] ).fit( X )
   labels = km.labels_
   ch_score = metrics.calinski_harabaz_score(X, labels)
   df1['No. of clusters'][line_count] = n_clusters[cluster_count]
   df1['Inertia'][line_count] = km.inertia_
   df1['Calinski-Harabaz Score'][line_count] = ch_score
   cluster_count = cluster_count + 1
```

```
    line_count = line_count + 1


# Show the results of the Inertia and Calinski-Harabasz scores

df1

# Since the above results show that the ideal clustering is 8 clusters, fit the model such that it
partitions the data into
# 8 clusters.

km = cluster.KMeans(n_clusters=8).fit(X)

# Show cluster centers (centroids) indicating mean values of each feature for each cluster

km.cluster_centers_

# Show cluster labels

my_label = km.labels_

my_label

# Show distribution of cluster labels

np.unique(my_label, return_counts=True)

# Show distribution of Supergroup labels

np.unique(y, return_counts=True)

# Features related to Marital Status

features = [14, 15, 16]

num_features = len(features)

#-gather fields of interest from full data set into X and y
X = []

for rec in rawdata:
    instance = []
    for f in features:
        instance.append( float( rec[f] ))
    X.append( instance )
```

```
X = np.array( X )
num_instances = X.shape[0]

print 'number of instances = %d' % num_instances
print 'shape of input data = %d x %d' % ( X.shape[0], X.shape[1] )


# Create a dataframe with the results of the Inertia and Calinski-Harabasz scores


columns = ['No. of clusters', 'Inertia', 'Calinski-Harabaz Score']

index = np.arange(14)

df2 = pd.DataFrame(columns = columns, index = index)

line_count = 0

n_clusters = [2, 3, 4, 5, 6, 7, 8, 9, 10, 20, 30, 40, 50, 100]

cluster_count = 0

while cluster_count < 14:
    km = cluster.KMeans( n_clusters[cluster_count] ).fit( X )
    labels = km.labels_
    ch_score = metrics.calinski_harabaz_score(X, labels)
    df2['No. of clusters'][line_count] = n_clusters[cluster_count]
    df2['Inertia'][line_count] = km.inertia_
    df2['Calinski-Harabaz Score'][line_count] = ch_score
    cluster_count = cluster_count + 1
    line_count = line_count + 1


# Show the results of the Inertia and Calinski-Harabasz scores

df2

# Since the above results show that the ideal clustering is 8 clusters, fit the model such that it
partitions the data into
# 8 clusters.

km = cluster.KMeans(n_clusters=8).fit(X)

# Show cluster centers (centroids) indicating mean values of each feature for each cluster

km.cluster_centers_
```

```
# Show cluster labels

my_label = km.labels_

my_label

# Show distribution of cluster labels

np.unique(my_label, return_counts=True)

# Show distribution of Supergroup labels

np.unique(y, return_counts=True)

# Features related to Ethnicity

features = [17, 18, 19, 20, 21, 22, 23, 24]

num_features = len(features)

#-gather fields of interest from full data set into X and y
X = []

for rec in rawdata:
   instance = []
   for f in features:
      instance.append( float( rec[f] ))
   X.append( instance )

X = np.array( X )
num_instances = X.shape[0]

print 'number of instances = %d' % num_instances
print 'shape of input data = %d x %d' % ( X.shape[0], X.shape[1] )


# Create a dataframe with the results of the Inertia and Calinski-Harabasz scores


columns = ['No. of clusters', 'Inertia', 'Calinski-Harabaz Score']

index = np.arange(14)

df3 = pd.DataFrame(columns = columns, index = index)
```

```
line_count = 0

n_clusters = [2, 3, 4, 5, 6, 7, 8, 9, 10, 20, 30, 40, 50, 100]

cluster_count = 0

while cluster_count < 14:
    km = cluster.KMeans( n_clusters[cluster_count] ).fit( X )
    labels = km.labels_
    ch_score = metrics.calinski_harabaz_score(X, labels)
    df3['No. of clusters'][line_count] = n_clusters[cluster_count]
    df3['Inertia'][line_count] = km.inertia_
    df3['Calinski-Harabaz Score'][line_count] = ch_score
    cluster_count = cluster_count + 1
    line_count = line_count + 1


# Show the results of the Inertia and Calinski-Harabasz scores

df3

# Since the above results show that the ideal clustering is 8 clusters, fit the model such that it
partitions the data into
# 8 clusters.

km = cluster.KMeans(n_clusters=8).fit(X)

# Show cluster centers (centroids) indicating mean values of each feature for each cluster

km.cluster_centers_

# Show cluster labels

my_label = km.labels_

my_label

# Show distribution of cluster labels

np.unique(my_label, return_counts=True)

# Show distribution of Supergroup labels

np.unique(y, return_counts=True)

# Features related to Employment Sector
```

```
features = [28, 29, 30, 31, 32, 33, 34, 35, 36, 37, 38, 39, 40]

num_features = len(features)

#-gather fields of interest from full data set into X and y
X = []

for rec in rawdata:
   instance = []
   for f in features:
      instance.append( float( rec[f] ))
   X.append( instance )

X = np.array( X )
num_instances = X.shape[0]

print 'number of instances = %d' % num_instances
print 'shape of input data = %d x %d' % ( X.shape[0], X.shape[1] )


# Create a dataframe with the results of the Inertia and Calinski-Harabasz scores


columns = ['No. of clusters', 'Inertia', 'Calinski-Harabaz Score']

index = np.arange(14)

df5 = pd.DataFrame(columns = columns, index = index)

line_count = 0

n_clusters = [2, 3, 4, 5, 6, 7, 8, 9, 10, 20, 30, 40, 50, 100]

cluster_count = 0

while cluster_count < 14:
   km = cluster.KMeans( n_clusters[cluster_count] ).fit( X )
   labels = km.labels_
   ch_score = metrics.calinski_harabaz_score(X, labels)
   df5['No. of clusters'][line_count] = n_clusters[cluster_count]
   df5['Inertia'][line_count] = km.inertia_
   df5['Calinski-Harabaz Score'][line_count] = ch_score
   cluster_count = cluster_count + 1
   line_count = line_count + 1
```

```
# Show the results of the Inertia and Calinski-Harabasz scores

df5

# Since the above results show that the ideal clustering is 8 clusters, fit the model such that it partitions the data into
# 8 clusters.

km = cluster.KMeans(n_clusters=8).fit(X)

# Show cluster centers (centroids) indicating mean values of each feature for each cluster

km.cluster_centers_

# Show cluster labels

my_label = km.labels_

my_label

# Show distribution of cluster labels

np.unique(my_label, return_counts=True)


# Show distribution of Supergroup labels

np.unique(y, return_counts=True)

# Features related to Education Levels

features = [50, 51, 52]

num_features = len(features)

#-gather fields of interest from full data set into X and y
X = []

for rec in rawdata:
    instance = []
    for f in features:
        instance.append( float( rec[f] ))
    X.append( instance )

X = np.array( X )
```

```
num_instances = X.shape[0]

print 'number of instances = %d' % num_instances
print 'shape of input data = %d x %d' % ( X.shape[0], X.shape[1] )


# Create a dataframe with the results of the Inertia and Calinski-Harabasz scores


columns = ['No. of clusters', 'Inertia', 'Calinski-Harabaz Score']

index = np.arange(14)

df7 = pd.DataFrame(columns = columns, index = index)

line_count = 0

n_clusters = [2, 3, 4, 5, 6, 7, 8, 9, 10, 20, 30, 40, 50, 100]

cluster_count = 0

while cluster_count < 14:
    km = cluster.KMeans( n_clusters[cluster_count] ).fit( X )
    labels = km.labels_
    ch_score = metrics.calinski_harabaz_score(X, labels)
    df7['No. of clusters'][line_count] = n_clusters[cluster_count]
    df7['Inertia'][line_count] = km.inertia_
    df7['Calinski-Harabaz Score'][line_count] = ch_score
    cluster_count = cluster_count + 1
    line_count = line_count + 1


# Show the results of the Inertia and Calinski-Harabasz scores

df7

# Since the above results show that the ideal clustering is 8 clusters, fit the model such that it
partitions the data into
# 8 clusters.

km = cluster.KMeans(n_clusters=8).fit(X)

# Show cluster centers (centroids) indicating mean values of each feature for each cluster

km.cluster_centers_
```

```
# Show cluster labels

my_label = km.labels_

my_label

# Show distribution of cluster labels

np.unique(my_label, return_counts=True)

# Show distribution of Supergroup labels

np.unique(y, return_counts=True)

############
## Step 6:
############

# Create a new dataframe from the loac_input_df dataframe that has all the TfL tube and rail stations but discard
# unnecessary fields like 'Keep', 'Supergroup', and 'id'

station_match_df = loac_input_df.drop('Keep', axis=1)

station_match_df = station_match_df.drop('Supergroup', axis=1)

station_match_df = station_match_df.drop('id', axis=1)

# Show the head of the station_match_df dataframe

station_match_df.head()

# Add a new row that refers to unknown stations

# Source: https://stackoverflow.com/questions/24284342/insert-a-row-to-pandas-dataframe

station_match_df.loc[-1] = [-1, 'Unknown'] # add a new row for unknown station keys (marked as -1)
station_match_df.index = station_match_df.index + 1 # shift the index
station_match_df = station_match_df.sort() # sort the dataframe by index

# Merge the finalmerged_df dataframe (which contains all the passenger journey records from the sample dataset) with the
# Station Names dataframe so that each Station of Entry Key is matched up with its respective station name.
```

```
finalstation_df = finalmerged_df.merge(station_match_df,
left_on='STATIONOFFIRSTENTRYKEY', right_on='STATIONKEY', how='left')


finalstation_df.head()

# Rename the 'STATIONNAME' column as 'Station of Entry'

finalstation_df = finalstation_df.rename(columns={'STATIONNAME': 'Station of Entry'})

finalstation_df.head()

# Drop the 'STATIONKEY' column

finalstation_df = finalstation_df.drop('STATIONKEY', axis=1)

finalstation_df.head()

# Merge the revised finalmerged_df dataframe with the Station Names dataframe so that each
Station of Exit Key is matched up with its respective station name.

finalstation_df = finalstation_df.merge(station_match_df, left_on='STATIONOFEXITKEY',
right_on='STATIONKEY', how='left')


finalstation_df.head()

# Rename the 'STATIONNAME' column as 'Station of Exit'

finalstation_df = finalstation_df.rename(columns={'STATIONNAME': 'Station of Exit'})

# Drop the 'STATIONKEY' column

finalstation_df = finalstation_df.drop('STATIONKEY', axis=1)

finalstation_df.head()

passengerfinal_df.head()

# Delete all unnecessary columns from passengerfinal_df to create a simple dataframe with just
the prestigeid and its associated Supergroup code

passenger_simple_df = passengerfinal_df.drop('STATIONOFFIRSTENTRYKEY', axis=1)
passenger_simple_df = passenger_simple_df.drop('id', axis=1)
passenger_simple_df = passenger_simple_df.drop('STATIONKEY', axis=1)
passenger_simple_df = passenger_simple_df.drop('STATIONNAME', axis=1)
```

```
passenger_simple_df = passenger_simple_df.drop('Keep', axis=1)

passenger_simple_df.head()

# Merge the Passenger dataframe with the TfL journey dataframe so that each journey record in the original dataset
# has an associated Supergroup code.

tflfinal_df = finalstation_df.merge(passenger_simple_df, left_on='PRESTIGEID', right_on='PRESTIGEID', how='left')


tflfinal_df.head()

# The purpose of this section is to determine the travel behaviour (including most frequently used stations, most frequent
# travel times throughout the day, and most frequent days travelled throughout the week) of each Supergroup

# Get the frequency distribution of journeys travelled on each day of the week categorized by Supergroup

supergroup_day = tflfinal_df.groupby(['Supergroup', 'Day of the Week']).size()

# Save the above in an output file

supergroup_day.to_csv("Supergroup_day.csv", sep = '\t')

# Show the frequency distribution of journeys travelled on each day of the week categorized by Supergroup

tflfinal_df.groupby(['Supergroup', 'Day of the Week']).size()

# Get the frequency distribution of journeys travelled during each time interval (timeblock) in a 24-hour day categorized by
# Supergroup

supergroup_time = tflfinal_df.groupby(['Supergroup', 'Timeblock']).size()

# Save the above in an output file

supergroup_time.to_csv("Supergroup_time.csv", sep = '\t')

# Show the frequency distribution of journeys travelled during each time interval (timeblock) in a 24-hour day categorized by
# Supergroup
```

```
tflfinal_df.groupby(['Supergroup', 'Timeblock']).size()

# Get the frequency distribution of journeys commencing at each station categorized by Supergroup

# Source: https://stackoverflow.com/questions/35364601/group-by-and-find-top-n-value-counts-pandas

s_entry = tflfinal_df['Station of Entry'].groupby(tflfinal_df['Supergroup']).value_counts()

# Show the frequency distribution of journeys commencing at the five most frequently used stations of entry for each Supergroup

# Source: https://stackoverflow.com/questions/35364601/group-by-and-find-top-n-value-counts-pandas

print s_entry.groupby(level=0).nlargest(5)

# Get the frequency distribution of journeys ending at each station categorized by Supergroup

# Source: https://stackoverflow.com/questions/35364601/group-by-and-find-top-n-value-counts-pandas

s_exit = tflfinal_df['Station of Exit'].groupby(tflfinal_df['Supergroup']).value_counts()

# Show the frequency distribution of journeys ending at the five most frequently used stations of exit for each Supergroup

# Source: https://stackoverflow.com/questions/35364601/group-by-and-find-top-n-value-counts-pandas

print s_exit.groupby(level=0).nlargest(6)
```

5.2: Spark-SQL Code

```
# Open datafile in Spark database

scala> val dataframe = spark.read.format("csv").option("header",
"true").load("hdfs://localhost:54310/tfl-data/unzipped")


# Create temporary dataframe called "tfldataset"

scala> dataframe.createTempView("tfldataset")


# Get number of unique users in Spark Database:

scala> val sqlFrame = spark.sql("SELECT COUNT(DISTINCT PRESTIGEID) FROM
tfldataset")

scala> sqlFrame.show(200)



# Get frequency distribution of journeys based on age group (as recorded by TfL)

scala> val sqlFrame = spark.sql("SELECT PPTPASSENGERAGEKEY,
COUNT(PPTPASSENGERAGEKEY) Frequency,
COUNT(PPTPASSENGERAGEKEY)/(SELECT COUNT(PPTPASSENGERAGEKEY) FROM
tfldataset)*100 Percentage FROM tfldataset GROUP BY PPTPASSENGERAGEKEY ORDER
BY Frequency DESC")

scala> sqlFrame.show(200)


# Get frequency distribution of journeys based on date

scala> val sqlFrame = spark.sql("SELECT DAYKEY, COUNT(DAYKEY) Frequency,
COUNT(DAYKEY)/(SELECT COUNT(DAYKEY) FROM tfldataset)*100 Percentage FROM
tfldataset GROUP BY DAYKEY ORDER BY Frequency DESC")

scala> sqlFrame.show(200)


# Get frequency distribution of journeys based on card type
```

```
scala> val sqlFrame = spark.sql("SELECT CARDTYPEKEY, COUNT(CARDTYPEKEY)
Frequency, COUNT(CARDTYPEKEY)/(SELECT COUNT(CARDTYPEKEY) FROM
tfldataset)*100 Percentage FROM tfldataset GROUP BY CARDTYPEKEY ORDER BY
Frequency DESC")

scala> sqlFrame.show(200)


# Get distribution of travel times throughout the course of a day

# Note: TRANSACTIONTIME refers to number of minutes past midnight

scala> val sqlFrame = spark.sql("SELECT COUNT(*) FROM tfldataset WHERE
TRANSACTIONTIME >= 0 AND TRANSACTIONTIME < 120")

scala> sqlFrame.show(200)


scala> val sqlFrame = spark.sql("SELECT COUNT(*) FROM tfldataset WHERE
TRANSACTIONTIME >= 120 AND TRANSACTIONTIME < 240")

scala> sqlFrame.show(200)


scala> val sqlFrame = spark.sql("SELECT COUNT(*) FROM tfldataset WHERE
TRANSACTIONTIME >= 240 AND TRANSACTIONTIME < 360")

scala> sqlFrame.show(200)


scala> val sqlFrame = spark.sql("SELECT COUNT(*) FROM tfldataset WHERE
TRANSACTIONTIME >= 360 AND TRANSACTIONTIME < 480")

scala> sqlFrame.show(200)


scala> val sqlFrame = spark.sql("SELECT COUNT(*) FROM tfldataset WHERE
TRANSACTIONTIME >= 480 AND TRANSACTIONTIME < 600")

scala> sqlFrame.show(200)


scala> val sqlFrame = spark.sql("SELECT COUNT(*) FROM tfldataset WHERE
TRANSACTIONTIME >= 600 AND TRANSACTIONTIME < 720")

scala> sqlFrame.show(200)
```

```
scala> val sqlFrame = spark.sql("SELECT COUNT(*) FROM tfldataset WHERE
TRANSACTIONTIME >= 720 AND TRANSACTIONTIME < 840")

scala> sqlFrame.show(200)


scala> val sqlFrame = spark.sql("SELECT COUNT(*) FROM tfldataset WHERE
TRANSACTIONTIME >= 840 AND TRANSACTIONTIME < 960")

scala> sqlFrame.show(200)


scala> val sqlFrame = spark.sql("SELECT COUNT(*) FROM tfldataset WHERE
TRANSACTIONTIME >= 960 AND TRANSACTIONTIME < 1080")

scala> sqlFrame.show(200)


scala> val sqlFrame = spark.sql("SELECT COUNT(*) FROM tfldataset WHERE
TRANSACTIONTIME >= 1080 AND TRANSACTIONTIME < 1200")

scala> sqlFrame.show(200)


scala> val sqlFrame = spark.sql("SELECT COUNT(*) FROM tfldataset WHERE
TRANSACTIONTIME >= 1200 AND TRANSACTIONTIME < 1320")

scala> sqlFrame.show(200)


scala> val sqlFrame = spark.sql("SELECT COUNT(*) FROM tfldataset WHERE
TRANSACTIONTIME >= 1320 AND TRANSACTIONTIME < 1440")

scala> sqlFrame.show(200)



# Create a filtered dataframe from the entire TfL dataset in the Spark database. This filtered
dataframe only uses the fields we are
# interested in, including the unique Oyster card identification number, the passenger's age
(based on the information on the Oyster
# card, the date on which the journey took place, the type of Oyster card used, the station of
entry, the station exit, and the time
```

at which the journey took place. Furthermore, we filter out journeys where the station of entry is unknown.

```
scala> val sqlFrame = spark.sql("CREATE TABLE tfl_stage_1 SELECT PRESTIGEID,
PPTPASSENGERAGEKEY, DAYKEY, CARDTYPEKEY, STATIONOFFIRSTENTRYKEY,
STATIONOFEXITKEY, TRANSACTIONTIME FROM tfldataset WHERE
STATIONOFFIRSTENTRYKEY != '-1'")
```

Now we create a further filter where we only want records pertaining to journeys which commence at one of the known tube or rail stations
end in one of the known tube or rail stations (or end in an unknown station).

```
scala> val sqlFrame = spark.sql("CREATE TABLE tfl_stage_2 SELECT PRESTIGEID,
PPTPASSENGERAGEKEY, DAYKEY, CARDTYPEKEY, STATIONOFFIRSTENTRYKEY,
STATIONOFEXITKEY, TRANSACTIONTIME FROM tfl_stage_1 WHERE
(STATIONOFFIRSTENTRYKEY LIKE '1' OR STATIONOFFIRSTENTRYKEY LIKE '2' OR
STATIONOFFIRSTENTRYKEY LIKE '3' OR STATIONOFFIRSTENTRYKEY LIKE '4' OR
STATIONOFFIRSTENTRYKEY LIKE '5' OR STATIONOFFIRSTENTRYKEY LIKE '6' OR
STATIONOFFIRSTENTRYKEY LIKE '7' OR STATIONOFFIRSTENTRYKEY LIKE '8' OR
STATIONOFFIRSTENTRYKEY LIKE '9' OR STATIONOFFIRSTENTRYKEY LIKE '10' OR
STATIONOFFIRSTENTRYKEY LIKE '11' OR STATIONOFFIRSTENTRYKEY LIKE '12'
OR STATIONOFFIRSTENTRYKEY LIKE '13' OR STATIONOFFIRSTENTRYKEY LIKE
'14' OR STATIONOFFIRSTENTRYKEY LIKE '15' OR STATIONOFFIRSTENTRYKEY
LIKE '16' OR STATIONOFFIRSTENTRYKEY LIKE '17' OR
STATIONOFFIRSTENTRYKEY LIKE '18' OR STATIONOFFIRSTENTRYKEY LIKE '19'
OR STATIONOFFIRSTENTRYKEY LIKE '20' OR STATIONOFFIRSTENTRYKEY LIKE
'21' OR STATIONOFFIRSTENTRYKEY LIKE '22' OR STATIONOFFIRSTENTRYKEY
LIKE '23' OR STATIONOFFIRSTENTRYKEY LIKE '24' OR
STATIONOFFIRSTENTRYKEY LIKE '25' OR STATIONOFFIRSTENTRYKEY LIKE '26'
OR STATIONOFFIRSTENTRYKEY LIKE '27' OR STATIONOFFIRSTENTRYKEY LIKE
'28' OR STATIONOFFIRSTENTRYKEY LIKE '30' OR STATIONOFFIRSTENTRYKEY
LIKE '31' OR STATIONOFFIRSTENTRYKEY LIKE '32' OR
STATIONOFFIRSTENTRYKEY LIKE '33' OR STATIONOFFIRSTENTRYKEY LIKE '34'
OR STATIONOFFIRSTENTRYKEY LIKE '35' OR STATIONOFFIRSTENTRYKEY LIKE
'36' OR STATIONOFFIRSTENTRYKEY LIKE '37' OR STATIONOFFIRSTENTRYKEY
LIKE '38' OR STATIONOFFIRSTENTRYKEY LIKE '39' OR
STATIONOFFIRSTENTRYKEY LIKE '40' OR STATIONOFFIRSTENTRYKEY LIKE '41'
OR STATIONOFFIRSTENTRYKEY LIKE '42' OR STATIONOFFIRSTENTRYKEY LIKE
'43' OR STATIONOFFIRSTENTRYKEY LIKE '44' OR STATIONOFFIRSTENTRYKEY
LIKE '45' OR STATIONOFFIRSTENTRYKEY LIKE '46' OR
STATIONOFFIRSTENTRYKEY LIKE '47' OR STATIONOFFIRSTENTRYKEY LIKE '48'
OR STATIONOFFIRSTENTRYKEY LIKE '49' OR STATIONOFFIRSTENTRYKEY LIKE
'50' OR STATIONOFFIRSTENTRYKEY LIKE '51' OR STATIONOFFIRSTENTRYKEY
LIKE '52' OR STATIONOFFIRSTENTRYKEY LIKE '53' OR
```

STATIONOFFIRSTENTRYKEY LIKE '54' OR STATIONOFFIRSTENTRYKEY LIKE '55' OR STATIONOFFIRSTENTRYKEY LIKE '56' OR STATIONOFFIRSTENTRYKEY LIKE '57' OR STATIONOFFIRSTENTRYKEY LIKE '59' OR STATIONOFFIRSTENTRYKEY LIKE '60' OR STATIONOFFIRSTENTRYKEY LIKE '61' OR STATIONOFFIRSTENTRYKEY LIKE '62' OR STATIONOFFIRSTENTRYKEY LIKE '63' OR STATIONOFFIRSTENTRYKEY LIKE '64' OR STATIONOFFIRSTENTRYKEY LIKE '65' OR STATIONOFFIRSTENTRYKEY LIKE '66' OR STATIONOFFIRSTENTRYKEY LIKE '67' OR STATIONOFFIRSTENTRYKEY LIKE '68' OR STATIONOFFIRSTENTRYKEY LIKE '69' OR STATIONOFFIRSTENTRYKEY LIKE '70' OR STATIONOFFIRSTENTRYKEY LIKE '71' OR STATIONOFFIRSTENTRYKEY LIKE '72' OR STATIONOFFIRSTENTRYKEY LIKE '73' OR STATIONOFFIRSTENTRYKEY LIKE '74' OR STATIONOFFIRSTENTRYKEY LIKE '76' OR STATIONOFFIRSTENTRYKEY LIKE '77' OR STATIONOFFIRSTENTRYKEY LIKE '78' OR STATIONOFFIRSTENTRYKEY LIKE '79' OR STATIONOFFIRSTENTRYKEY LIKE '80' OR STATIONOFFIRSTENTRYKEY LIKE '81' OR STATIONOFFIRSTENTRYKEY LIKE '82' OR STATIONOFFIRSTENTRYKEY LIKE '83' OR STATIONOFFIRSTENTRYKEY LIKE '84' OR STATIONOFFIRSTENTRYKEY LIKE '85' OR STATIONOFFIRSTENTRYKEY LIKE '86' OR STATIONOFFIRSTENTRYKEY LIKE '87' OR STATIONOFFIRSTENTRYKEY LIKE '88' OR STATIONOFFIRSTENTRYKEY LIKE '89' OR STATIONOFFIRSTENTRYKEY LIKE '90' OR STATIONOFFIRSTENTRYKEY LIKE '92' OR STATIONOFFIRSTENTRYKEY LIKE '93' OR STATIONOFFIRSTENTRYKEY LIKE '94' OR STATIONOFFIRSTENTRYKEY LIKE '95' OR STATIONOFFIRSTENTRYKEY LIKE '96' OR STATIONOFFIRSTENTRYKEY LIKE '97' OR STATIONOFFIRSTENTRYKEY LIKE '98' OR STATIONOFFIRSTENTRYKEY LIKE '99' OR STATIONOFFIRSTENTRYKEY LIKE '100' OR STATIONOFFIRSTENTRYKEY LIKE '101' OR STATIONOFFIRSTENTRYKEY LIKE '102' OR STATIONOFFIRSTENTRYKEY LIKE '103' OR STATIONOFFIRSTENTRYKEY LIKE '104' OR STATIONOFFIRSTENTRYKEY LIKE '105' OR STATIONOFFIRSTENTRYKEY LIKE '106' OR STATIONOFFIRSTENTRYKEY LIKE '107' OR STATIONOFFIRSTENTRYKEY LIKE '109' OR STATIONOFFIRSTENTRYKEY LIKE '110' OR STATIONOFFIRSTENTRYKEY LIKE '111' OR STATIONOFFIRSTENTRYKEY LIKE '112' OR STATIONOFFIRSTENTRYKEY LIKE '113' OR STATIONOFFIRSTENTRYKEY LIKE '115' OR STATIONOFFIRSTENTRYKEY LIKE '116' OR STATIONOFFIRSTENTRYKEY LIKE '117' OR STATIONOFFIRSTENTRYKEY LIKE '118' OR STATIONOFFIRSTENTRYKEY LIKE '119' OR STATIONOFFIRSTENTRYKEY LIKE '120' OR STATIONOFFIRSTENTRYKEY LIKE '121' OR STATIONOFFIRSTENTRYKEY LIKE '122' OR STATIONOFFIRSTENTRYKEY LIKE '123' OR STATIONOFFIRSTENTRYKEY LIKE '124' OR STATIONOFFIRSTENTRYKEY LIKE '125' OR STATIONOFFIRSTENTRYKEY LIKE '126' OR STATIONOFFIRSTENTRYKEY LIKE '127' OR STATIONOFFIRSTENTRYKEY LIKE '128' OR STATIONOFFIRSTENTRYKEY LIKE '129' OR STATIONOFFIRSTENTRYKEY LIKE '130' OR STATIONOFFIRSTENTRYKEY LIKE '131' OR STATIONOFFIRSTENTRYKEY LIKE '132' OR STATIONOFFIRSTENTRYKEY LIKE '133' OR STATIONOFFIRSTENTRYKEY LIKE '134' OR STATIONOFFIRSTENTRYKEY LIKE '135' OR STATIONOFFIRSTENTRYKEY LIKE '136' OR STATIONOFFIRSTENTRYKEY LIKE '137' OR STATIONOFFIRSTENTRYKEY LIKE '138' OR STATIONOFFIRSTENTRYKEY LIKE

'139' OR STATIONOFFIRSTENTRYKEY LIKE '141' OR STATIONOFFIRSTENTRYKEY LIKE '142' OR STATIONOFFIRSTENTRYKEY LIKE '143' OR STATIONOFFIRSTENTRYKEY LIKE '144' OR STATIONOFFIRSTENTRYKEY LIKE '145' OR STATIONOFFIRSTENTRYKEY LIKE '146' OR STATIONOFFIRSTENTRYKEY LIKE '147' OR STATIONOFFIRSTENTRYKEY LIKE '148' OR STATIONOFFIRSTENTRYKEY LIKE '149' OR STATIONOFFIRSTENTRYKEY LIKE '150' OR STATIONOFFIRSTENTRYKEY LIKE '151' OR STATIONOFFIRSTENTRYKEY LIKE '152' OR STATIONOFFIRSTENTRYKEY LIKE '153' OR STATIONOFFIRSTENTRYKEY LIKE '154' OR STATIONOFFIRSTENTRYKEY LIKE '155' OR STATIONOFFIRSTENTRYKEY LIKE '156' OR STATIONOFFIRSTENTRYKEY LIKE '157' OR STATIONOFFIRSTENTRYKEY LIKE '158' OR STATIONOFFIRSTENTRYKEY LIKE '159' OR STATIONOFFIRSTENTRYKEY LIKE '160' OR STATIONOFFIRSTENTRYKEY LIKE '161' OR STATIONOFFIRSTENTRYKEY LIKE '162' OR STATIONOFFIRSTENTRYKEY LIKE '163' OR STATIONOFFIRSTENTRYKEY LIKE '164' OR STATIONOFFIRSTENTRYKEY LIKE '166' OR STATIONOFFIRSTENTRYKEY LIKE '167' OR STATIONOFFIRSTENTRYKEY LIKE '168' OR STATIONOFFIRSTENTRYKEY LIKE '169' OR STATIONOFFIRSTENTRYKEY LIKE '171' OR STATIONOFFIRSTENTRYKEY LIKE '172' OR STATIONOFFIRSTENTRYKEY LIKE '173' OR STATIONOFFIRSTENTRYKEY LIKE '175' OR STATIONOFFIRSTENTRYKEY LIKE '176' OR STATIONOFFIRSTENTRYKEY LIKE '177' OR STATIONOFFIRSTENTRYKEY LIKE '178' OR STATIONOFFIRSTENTRYKEY LIKE '179' OR STATIONOFFIRSTENTRYKEY LIKE '183' OR STATIONOFFIRSTENTRYKEY LIKE '187' OR STATIONOFFIRSTENTRYKEY LIKE '188' OR STATIONOFFIRSTENTRYKEY LIKE '189' OR STATIONOFFIRSTENTRYKEY LIKE '190' OR STATIONOFFIRSTENTRYKEY LIKE '191' OR STATIONOFFIRSTENTRYKEY LIKE '192' OR STATIONOFFIRSTENTRYKEY LIKE '193' OR STATIONOFFIRSTENTRYKEY LIKE '194' OR STATIONOFFIRSTENTRYKEY LIKE '195' OR STATIONOFFIRSTENTRYKEY LIKE '196' OR STATIONOFFIRSTENTRYKEY LIKE '197' OR STATIONOFFIRSTENTRYKEY LIKE '198' OR STATIONOFFIRSTENTRYKEY LIKE '199' OR STATIONOFFIRSTENTRYKEY LIKE '200' OR STATIONOFFIRSTENTRYKEY LIKE '201' OR STATIONOFFIRSTENTRYKEY LIKE '202' OR STATIONOFFIRSTENTRYKEY LIKE '203' OR STATIONOFFIRSTENTRYKEY LIKE '205' OR STATIONOFFIRSTENTRYKEY LIKE '206' OR STATIONOFFIRSTENTRYKEY LIKE '207' OR STATIONOFFIRSTENTRYKEY LIKE '208' OR STATIONOFFIRSTENTRYKEY LIKE '209' OR STATIONOFFIRSTENTRYKEY LIKE '210' OR STATIONOFFIRSTENTRYKEY LIKE '211' OR STATIONOFFIRSTENTRYKEY LIKE '213' OR STATIONOFFIRSTENTRYKEY LIKE '214' OR STATIONOFFIRSTENTRYKEY LIKE '215' OR STATIONOFFIRSTENTRYKEY LIKE '216' OR STATIONOFFIRSTENTRYKEY LIKE '217' OR STATIONOFFIRSTENTRYKEY LIKE '218' OR STATIONOFFIRSTENTRYKEY LIKE '219' OR STATIONOFFIRSTENTRYKEY LIKE '220' OR STATIONOFFIRSTENTRYKEY LIKE '221' OR STATIONOFFIRSTENTRYKEY LIKE '222' OR STATIONOFFIRSTENTRYKEY LIKE '223' OR STATIONOFFIRSTENTRYKEY LIKE '224' OR STATIONOFFIRSTENTRYKEY LIKE '225' OR STATIONOFFIRSTENTRYKEY LIKE '227' OR STATIONOFFIRSTENTRYKEY LIKE '228' OR STATIONOFFIRSTENTRYKEY LIKE '229' OR STATIONOFFIRSTENTRYKEY LIKE '231' OR STATIONOFFIRSTENTRYKEY LIKE '232' OR STATIONOFFIRSTENTRYKEY LIKE '233' OR

STATIONOFFIRSTENTRYKEY LIKE '234' OR STATIONOFFIRSTENTRYKEY LIKE '235' OR STATIONOFFIRSTENTRYKEY LIKE '236' OR STATIONOFFIRSTENTRYKEY LIKE '237' OR STATIONOFFIRSTENTRYKEY LIKE '238' OR STATIONOFFIRSTENTRYKEY LIKE '239' OR STATIONOFFIRSTENTRYKEY LIKE '240' OR STATIONOFFIRSTENTRYKEY LIKE '241' OR STATIONOFFIRSTENTRYKEY LIKE '242' OR STATIONOFFIRSTENTRYKEY LIKE '243' OR STATIONOFFIRSTENTRYKEY LIKE '244' OR STATIONOFFIRSTENTRYKEY LIKE '245' OR STATIONOFFIRSTENTRYKEY LIKE '246' OR STATIONOFFIRSTENTRYKEY LIKE '247' OR STATIONOFFIRSTENTRYKEY LIKE '248' OR STATIONOFFIRSTENTRYKEY LIKE '249' OR STATIONOFFIRSTENTRYKEY LIKE '251' OR STATIONOFFIRSTENTRYKEY LIKE '252' OR STATIONOFFIRSTENTRYKEY LIKE '254' OR STATIONOFFIRSTENTRYKEY LIKE '255' OR STATIONOFFIRSTENTRYKEY LIKE '256' OR STATIONOFFIRSTENTRYKEY LIKE '257' OR STATIONOFFIRSTENTRYKEY LIKE '261' OR STATIONOFFIRSTENTRYKEY LIKE '262' OR STATIONOFFIRSTENTRYKEY LIKE '263' OR STATIONOFFIRSTENTRYKEY LIKE '264' OR STATIONOFFIRSTENTRYKEY LIKE '267' OR STATIONOFFIRSTENTRYKEY LIKE '268' OR STATIONOFFIRSTENTRYKEY LIKE '269' OR STATIONOFFIRSTENTRYKEY LIKE '284' OR STATIONOFFIRSTENTRYKEY LIKE '299' OR STATIONOFFIRSTENTRYKEY LIKE '302' OR STATIONOFFIRSTENTRYKEY LIKE '303' OR STATIONOFFIRSTENTRYKEY LIKE '318' OR STATIONOFFIRSTENTRYKEY LIKE '336' OR STATIONOFFIRSTENTRYKEY LIKE '338' OR STATIONOFFIRSTENTRYKEY LIKE '373' OR STATIONOFFIRSTENTRYKEY LIKE '384' OR STATIONOFFIRSTENTRYKEY LIKE '393' OR STATIONOFFIRSTENTRYKEY LIKE '402' OR STATIONOFFIRSTENTRYKEY LIKE '431' OR STATIONOFFIRSTENTRYKEY LIKE '449' OR STATIONOFFIRSTENTRYKEY LIKE '454' OR STATIONOFFIRSTENTRYKEY LIKE '462' OR STATIONOFFIRSTENTRYKEY LIKE '463' OR STATIONOFFIRSTENTRYKEY LIKE '489' OR STATIONOFFIRSTENTRYKEY LIKE '493' OR STATIONOFFIRSTENTRYKEY LIKE '521' OR STATIONOFFIRSTENTRYKEY LIKE '526' OR STATIONOFFIRSTENTRYKEY LIKE '538' OR STATIONOFFIRSTENTRYKEY LIKE '587' OR STATIONOFFIRSTENTRYKEY LIKE '602' OR STATIONOFFIRSTENTRYKEY LIKE '606' OR STATIONOFFIRSTENTRYKEY LIKE '608' OR STATIONOFFIRSTENTRYKEY LIKE '610' OR STATIONOFFIRSTENTRYKEY LIKE '643' OR STATIONOFFIRSTENTRYKEY LIKE '644' OR STATIONOFFIRSTENTRYKEY LIKE '662' OR STATIONOFFIRSTENTRYKEY LIKE '670' OR STATIONOFFIRSTENTRYKEY LIKE '671' OR STATIONOFFIRSTENTRYKEY LIKE '691' OR STATIONOFFIRSTENTRYKEY LIKE '701' OR STATIONOFFIRSTENTRYKEY LIKE '708' OR STATIONOFFIRSTENTRYKEY LIKE '717' OR STATIONOFFIRSTENTRYKEY LIKE '744' OR STATIONOFFIRSTENTRYKEY LIKE '745' OR STATIONOFFIRSTENTRYKEY LIKE '752' OR STATIONOFFIRSTENTRYKEY LIKE '765' OR STATIONOFFIRSTENTRYKEY LIKE '789' OR STATIONOFFIRSTENTRYKEY LIKE '791' OR STATIONOFFIRSTENTRYKEY LIKE '822' OR STATIONOFFIRSTENTRYKEY LIKE '825' OR STATIONOFFIRSTENTRYKEY LIKE '861' OR STATIONOFFIRSTENTRYKEY LIKE '870' OR STATIONOFFIRSTENTRYKEY LIKE '871' OR STATIONOFFIRSTENTRYKEY LIKE '872' OR STATIONOFFIRSTENTRYKEY LIKE '876' OR STATIONOFFIRSTENTRYKEY LIKE '882' OR STATIONOFFIRSTENTRYKEY LIKE '903' OR STATIONOFFIRSTENTRYKEY LIKE '910' OR STATIONOFFIRSTENTRYKEY LIKE

'915' OR STATIONOFFIRSTENTRYKEY LIKE '918' OR STATIONOFFIRSTENTRYKEY LIKE '930' OR STATIONOFFIRSTENTRYKEY LIKE '932' OR STATIONOFFIRSTENTRYKEY LIKE '974' OR STATIONOFFIRSTENTRYKEY LIKE '981' OR STATIONOFFIRSTENTRYKEY LIKE '1123' OR STATIONOFFIRSTENTRYKEY LIKE '1124' OR STATIONOFFIRSTENTRYKEY LIKE '1293' OR STATIONOFFIRSTENTRYKEY LIKE '1294' OR STATIONOFFIRSTENTRYKEY LIKE '1295' OR STATIONOFFIRSTENTRYKEY LIKE '1335' OR STATIONOFFIRSTENTRYKEY LIKE '1337' OR STATIONOFFIRSTENTRYKEY LIKE '1338' OR STATIONOFFIRSTENTRYKEY LIKE '1339' OR STATIONOFFIRSTENTRYKEY LIKE '1341' OR STATIONOFFIRSTENTRYKEY LIKE '1342' OR STATIONOFFIRSTENTRYKEY LIKE '1343' OR STATIONOFFIRSTENTRYKEY LIKE '1345' OR STATIONOFFIRSTENTRYKEY LIKE '1348' OR STATIONOFFIRSTENTRYKEY LIKE '1349' OR STATIONOFFIRSTENTRYKEY LIKE '1350' OR STATIONOFFIRSTENTRYKEY LIKE '1351' OR STATIONOFFIRSTENTRYKEY LIKE '1352' OR STATIONOFFIRSTENTRYKEY LIKE '1355' OR STATIONOFFIRSTENTRYKEY LIKE '1356' OR STATIONOFFIRSTENTRYKEY LIKE '1357' OR STATIONOFFIRSTENTRYKEY LIKE '1365' OR STATIONOFFIRSTENTRYKEY LIKE '1366' OR STATIONOFFIRSTENTRYKEY LIKE '1367' OR STATIONOFFIRSTENTRYKEY LIKE '1368' OR STATIONOFFIRSTENTRYKEY LIKE '1371' OR STATIONOFFIRSTENTRYKEY LIKE '1378' OR STATIONOFFIRSTENTRYKEY LIKE '1409' OR STATIONOFFIRSTENTRYKEY LIKE '1486' OR STATIONOFFIRSTENTRYKEY LIKE '1510' OR STATIONOFFIRSTENTRYKEY LIKE '1521' OR STATIONOFFIRSTENTRYKEY LIKE '1572' OR STATIONOFFIRSTENTRYKEY LIKE '1577' OR STATIONOFFIRSTENTRYKEY LIKE '1615' OR STATIONOFFIRSTENTRYKEY LIKE '1616' OR STATIONOFFIRSTENTRYKEY LIKE '1617' OR STATIONOFFIRSTENTRYKEY LIKE '1618' OR STATIONOFFIRSTENTRYKEY LIKE '1619' OR STATIONOFFIRSTENTRYKEY LIKE '1620' OR STATIONOFFIRSTENTRYKEY LIKE '1679' OR STATIONOFFIRSTENTRYKEY LIKE '1680' OR STATIONOFFIRSTENTRYKEY LIKE '1681' OR STATIONOFFIRSTENTRYKEY LIKE '1682' OR STATIONOFFIRSTENTRYKEY LIKE '1683' OR STATIONOFFIRSTENTRYKEY LIKE '1685' OR STATIONOFFIRSTENTRYKEY LIKE '1686' OR STATIONOFFIRSTENTRYKEY LIKE '1688' OR STATIONOFFIRSTENTRYKEY LIKE '1731' OR STATIONOFFIRSTENTRYKEY LIKE '1946' OR STATIONOFFIRSTENTRYKEY LIKE '1955' OR STATIONOFFIRSTENTRYKEY LIKE '1956' OR STATIONOFFIRSTENTRYKEY LIKE '1958' OR STATIONOFFIRSTENTRYKEY LIKE '1959' OR STATIONOFFIRSTENTRYKEY LIKE '1960' OR STATIONOFFIRSTENTRYKEY LIKE '1961' OR STATIONOFFIRSTENTRYKEY LIKE '1962' OR STATIONOFFIRSTENTRYKEY LIKE '1963' OR STATIONOFFIRSTENTRYKEY LIKE '1964' OR STATIONOFFIRSTENTRYKEY LIKE '1965' OR STATIONOFFIRSTENTRYKEY LIKE '1966' OR STATIONOFFIRSTENTRYKEY LIKE '1968' OR STATIONOFFIRSTENTRYKEY LIKE '1969' OR STATIONOFFIRSTENTRYKEY LIKE '1971' OR STATIONOFFIRSTENTRYKEY LIKE '1972' OR STATIONOFFIRSTENTRYKEY LIKE '1973' OR STATIONOFFIRSTENTRYKEY LIKE '1977' OR STATIONOFFIRSTENTRYKEY LIKE '1980' OR STATIONOFFIRSTENTRYKEY LIKE '1984' OR STATIONOFFIRSTENTRYKEY LIKE '1988' OR STATIONOFFIRSTENTRYKEY LIKE '1989' OR STATIONOFFIRSTENTRYKEY LIKE '1995' OR STATIONOFFIRSTENTRYKEY LIKE

'2000' OR STATIONOFFIRSTENTRYKEY LIKE '2070' OR STATIONOFFIRSTENTRYKEY LIKE '2117' OR STATIONOFFIRSTENTRYKEY LIKE '2118')")

```
scala> val sqlFrame = spark.sql("CREATE TABLE tfl_final SELECT PRESTIGEID,
PPTPASSENGERAGEKEY, DAYKEY, CARDTYPEKEY, STATIONOFFIRSTENTRYKEY,
STATIONOFEXITKEY, TRANSACTIONTIME FROM tfl_stage_2 WHERE
(STATIONOFEXITKEY LIKE '-1' OR STATIONOFEXITKEY LIKE '1' OR
STATIONOFEXITKEY LIKE '2' OR STATIONOFEXITKEY LIKE '3' OR
STATIONOFEXITKEY LIKE '4' OR STATIONOFEXITKEY LIKE '5' OR
STATIONOFEXITKEY LIKE '6' OR STATIONOFEXITKEY LIKE '7' OR
STATIONOFEXITKEY LIKE '8' OR STATIONOFEXITKEY LIKE '9' OR
STATIONOFEXITKEY LIKE '10' OR STATIONOFEXITKEY LIKE '11' OR
STATIONOFEXITKEY LIKE '12' OR STATIONOFEXITKEY LIKE '13' OR
STATIONOFEXITKEY LIKE '14' OR STATIONOFEXITKEY LIKE '15' OR
STATIONOFEXITKEY LIKE '16' OR STATIONOFEXITKEY LIKE '17' OR
STATIONOFEXITKEY LIKE '18' OR STATIONOFEXITKEY LIKE '19' OR
STATIONOFEXITKEY LIKE '20' OR STATIONOFEXITKEY LIKE '21' OR
STATIONOFEXITKEY LIKE '22' OR STATIONOFEXITKEY LIKE '23' OR
STATIONOFEXITKEY LIKE '24' OR STATIONOFEXITKEY LIKE '25' OR
STATIONOFEXITKEY LIKE '26' OR STATIONOFEXITKEY LIKE '27' OR
STATIONOFEXITKEY LIKE '28' OR STATIONOFEXITKEY LIKE '30' OR
STATIONOFEXITKEY LIKE '31' OR STATIONOFEXITKEY LIKE '32' OR
STATIONOFEXITKEY LIKE '33' OR STATIONOFEXITKEY LIKE '34' OR
STATIONOFEXITKEY LIKE '35' OR STATIONOFEXITKEY LIKE '36' OR
STATIONOFEXITKEY LIKE '37' OR STATIONOFEXITKEY LIKE '38' OR
STATIONOFEXITKEY LIKE '39' OR STATIONOFEXITKEY LIKE '40' OR
STATIONOFEXITKEY LIKE '41' OR STATIONOFEXITKEY LIKE '42' OR
STATIONOFEXITKEY LIKE '43' OR STATIONOFEXITKEY LIKE '44' OR
STATIONOFEXITKEY LIKE '45' OR STATIONOFEXITKEY LIKE '46' OR
STATIONOFEXITKEY LIKE '47' OR STATIONOFEXITKEY LIKE '48' OR
STATIONOFEXITKEY LIKE '49' OR STATIONOFEXITKEY LIKE '50' OR
STATIONOFEXITKEY LIKE '51' OR STATIONOFEXITKEY LIKE '52' OR
STATIONOFEXITKEY LIKE '53' OR STATIONOFEXITKEY LIKE '54' OR
STATIONOFEXITKEY LIKE '55' OR STATIONOFEXITKEY LIKE '56' OR
STATIONOFEXITKEY LIKE '57' OR STATIONOFEXITKEY LIKE '59' OR
STATIONOFEXITKEY LIKE '60' OR STATIONOFEXITKEY LIKE '61' OR
STATIONOFEXITKEY LIKE '62' OR STATIONOFEXITKEY LIKE '63' OR
STATIONOFEXITKEY LIKE '64' OR STATIONOFEXITKEY LIKE '65' OR
STATIONOFEXITKEY LIKE '66' OR STATIONOFEXITKEY LIKE '67' OR
STATIONOFEXITKEY LIKE '68' OR STATIONOFEXITKEY LIKE '69' OR
STATIONOFEXITKEY LIKE '70' OR STATIONOFEXITKEY LIKE '71' OR
STATIONOFEXITKEY LIKE '72' OR STATIONOFEXITKEY LIKE '73' OR
STATIONOFEXITKEY LIKE '74' OR STATIONOFEXITKEY LIKE '76' OR
STATIONOFEXITKEY LIKE '77' OR STATIONOFEXITKEY LIKE '78' OR
STATIONOFEXITKEY LIKE '79' OR STATIONOFEXITKEY LIKE '80' OR
```

STATIONOFEXITKEY LIKE '81' OR STATIONOFEXITKEY LIKE '82' OR
STATIONOFEXITKEY LIKE '83' OR STATIONOFEXITKEY LIKE '84' OR
STATIONOFEXITKEY LIKE '85' OR STATIONOFEXITKEY LIKE '86' OR
STATIONOFEXITKEY LIKE '87' OR STATIONOFEXITKEY LIKE '88' OR
STATIONOFEXITKEY LIKE '89' OR STATIONOFEXITKEY LIKE '90' OR
STATIONOFEXITKEY LIKE '92' OR STATIONOFEXITKEY LIKE '93' OR
STATIONOFEXITKEY LIKE '94' OR STATIONOFEXITKEY LIKE '95' OR
STATIONOFEXITKEY LIKE '96' OR STATIONOFEXITKEY LIKE '97' OR
STATIONOFEXITKEY LIKE '98' OR STATIONOFEXITKEY LIKE '99' OR
STATIONOFEXITKEY LIKE '100' OR STATIONOFEXITKEY LIKE '101' OR
STATIONOFEXITKEY LIKE '102' OR STATIONOFEXITKEY LIKE '103' OR
STATIONOFEXITKEY LIKE '104' OR STATIONOFEXITKEY LIKE '105' OR
STATIONOFEXITKEY LIKE '106' OR STATIONOFEXITKEY LIKE '107' OR
STATIONOFEXITKEY LIKE '109' OR STATIONOFEXITKEY LIKE '110' OR
STATIONOFEXITKEY LIKE '111' OR STATIONOFEXITKEY LIKE '112' OR
STATIONOFEXITKEY LIKE '113' OR STATIONOFEXITKEY LIKE '115' OR
STATIONOFEXITKEY LIKE '116' OR STATIONOFEXITKEY LIKE '117' OR
STATIONOFEXITKEY LIKE '118' OR STATIONOFEXITKEY LIKE '119' OR
STATIONOFEXITKEY LIKE '120' OR STATIONOFEXITKEY LIKE '121' OR
STATIONOFEXITKEY LIKE '122' OR STATIONOFEXITKEY LIKE '123' OR
STATIONOFEXITKEY LIKE '124' OR STATIONOFEXITKEY LIKE '125' OR
STATIONOFEXITKEY LIKE '126' OR STATIONOFEXITKEY LIKE '127' OR
STATIONOFEXITKEY LIKE '128' OR STATIONOFEXITKEY LIKE '129' OR
STATIONOFEXITKEY LIKE '130' OR STATIONOFEXITKEY LIKE '131' OR
STATIONOFEXITKEY LIKE '132' OR STATIONOFEXITKEY LIKE '133' OR
STATIONOFEXITKEY LIKE '134' OR STATIONOFEXITKEY LIKE '135' OR
STATIONOFEXITKEY LIKE '136' OR STATIONOFEXITKEY LIKE '137' OR
STATIONOFEXITKEY LIKE '138' OR STATIONOFEXITKEY LIKE '139' OR
STATIONOFEXITKEY LIKE '141' OR STATIONOFEXITKEY LIKE '142' OR
STATIONOFEXITKEY LIKE '143' OR STATIONOFEXITKEY LIKE '144' OR
STATIONOFEXITKEY LIKE '145' OR STATIONOFEXITKEY LIKE '146' OR
STATIONOFEXITKEY LIKE '147' OR STATIONOFEXITKEY LIKE '148' OR
STATIONOFEXITKEY LIKE '149' OR STATIONOFEXITKEY LIKE '150' OR
STATIONOFEXITKEY LIKE '151' OR STATIONOFEXITKEY LIKE '152' OR
STATIONOFEXITKEY LIKE '153' OR STATIONOFEXITKEY LIKE '154' OR
STATIONOFEXITKEY LIKE '155' OR STATIONOFEXITKEY LIKE '156' OR
STATIONOFEXITKEY LIKE '157' OR STATIONOFEXITKEY LIKE '158' OR
STATIONOFEXITKEY LIKE '159' OR STATIONOFEXITKEY LIKE '160' OR
STATIONOFEXITKEY LIKE '161' OR STATIONOFEXITKEY LIKE '162' OR
STATIONOFEXITKEY LIKE '163' OR STATIONOFEXITKEY LIKE '164' OR
STATIONOFEXITKEY LIKE '166' OR STATIONOFEXITKEY LIKE '167' OR
STATIONOFEXITKEY LIKE '168' OR STATIONOFEXITKEY LIKE '169' OR
STATIONOFEXITKEY LIKE '171' OR STATIONOFEXITKEY LIKE '172' OR
STATIONOFEXITKEY LIKE '173' OR STATIONOFEXITKEY LIKE '175' OR
STATIONOFEXITKEY LIKE '176' OR STATIONOFEXITKEY LIKE '177' OR
STATIONOFEXITKEY LIKE '178' OR STATIONOFEXITKEY LIKE '179' OR

STATIONOFEXITKEY LIKE '183' OR STATIONOFEXITKEY LIKE '187' OR
STATIONOFEXITKEY LIKE '188' OR STATIONOFEXITKEY LIKE '189' OR
STATIONOFEXITKEY LIKE '190' OR STATIONOFEXITKEY LIKE '191' OR
STATIONOFEXITKEY LIKE '192' OR STATIONOFEXITKEY LIKE '193' OR
STATIONOFEXITKEY LIKE '194' OR STATIONOFEXITKEY LIKE '195' OR
STATIONOFEXITKEY LIKE '196' OR STATIONOFEXITKEY LIKE '197' OR
STATIONOFEXITKEY LIKE '198' OR STATIONOFEXITKEY LIKE '199' OR
STATIONOFEXITKEY LIKE '200' OR STATIONOFEXITKEY LIKE '201' OR
STATIONOFEXITKEY LIKE '202' OR STATIONOFEXITKEY LIKE '203' OR
STATIONOFEXITKEY LIKE '205' OR STATIONOFEXITKEY LIKE '206' OR
STATIONOFEXITKEY LIKE '207' OR STATIONOFEXITKEY LIKE '208' OR
STATIONOFEXITKEY LIKE '209' OR STATIONOFEXITKEY LIKE '210' OR
STATIONOFEXITKEY LIKE '211' OR STATIONOFEXITKEY LIKE '213' OR
STATIONOFEXITKEY LIKE '214' OR STATIONOFEXITKEY LIKE '215' OR
STATIONOFEXITKEY LIKE '216' OR STATIONOFEXITKEY LIKE '217' OR
STATIONOFEXITKEY LIKE '218' OR STATIONOFEXITKEY LIKE '219' OR
STATIONOFEXITKEY LIKE '220' OR STATIONOFEXITKEY LIKE '221' OR
STATIONOFEXITKEY LIKE '222' OR STATIONOFEXITKEY LIKE '223' OR
STATIONOFEXITKEY LIKE '224' OR STATIONOFEXITKEY LIKE '225' OR
STATIONOFEXITKEY LIKE '227' OR STATIONOFEXITKEY LIKE '228' OR
STATIONOFEXITKEY LIKE '229' OR STATIONOFEXITKEY LIKE '231' OR
STATIONOFEXITKEY LIKE '232' OR STATIONOFEXITKEY LIKE '233' OR
STATIONOFEXITKEY LIKE '234' OR STATIONOFEXITKEY LIKE '235' OR
STATIONOFEXITKEY LIKE '236' OR STATIONOFEXITKEY LIKE '237' OR
STATIONOFEXITKEY LIKE '238' OR STATIONOFEXITKEY LIKE '239' OR
STATIONOFEXITKEY LIKE '240' OR STATIONOFEXITKEY LIKE '241' OR
STATIONOFEXITKEY LIKE '242' OR STATIONOFEXITKEY LIKE '243' OR
STATIONOFEXITKEY LIKE '244' OR STATIONOFEXITKEY LIKE '245' OR
STATIONOFEXITKEY LIKE '246' OR STATIONOFEXITKEY LIKE '247' OR
STATIONOFEXITKEY LIKE '248' OR STATIONOFEXITKEY LIKE '249' OR
STATIONOFEXITKEY LIKE '251' OR STATIONOFEXITKEY LIKE '252' OR
STATIONOFEXITKEY LIKE '254' OR STATIONOFEXITKEY LIKE '255' OR
STATIONOFEXITKEY LIKE '256' OR STATIONOFEXITKEY LIKE '257' OR
STATIONOFEXITKEY LIKE '261' OR STATIONOFEXITKEY LIKE '262' OR
STATIONOFEXITKEY LIKE '263' OR STATIONOFEXITKEY LIKE '264' OR
STATIONOFEXITKEY LIKE '267' OR STATIONOFEXITKEY LIKE '268' OR
STATIONOFEXITKEY LIKE '269' OR STATIONOFEXITKEY LIKE '284' OR
STATIONOFEXITKEY LIKE '299' OR STATIONOFEXITKEY LIKE '302' OR
STATIONOFEXITKEY LIKE '303' OR STATIONOFEXITKEY LIKE '318' OR
STATIONOFEXITKEY LIKE '336' OR STATIONOFEXITKEY LIKE '338' OR
STATIONOFEXITKEY LIKE '373' OR STATIONOFEXITKEY LIKE '384' OR
STATIONOFEXITKEY LIKE '393' OR STATIONOFEXITKEY LIKE '402' OR
STATIONOFEXITKEY LIKE '431' OR STATIONOFEXITKEY LIKE '449' OR
STATIONOFEXITKEY LIKE '454' OR STATIONOFEXITKEY LIKE '462' OR
STATIONOFEXITKEY LIKE '463' OR STATIONOFEXITKEY LIKE '489' OR
STATIONOFEXITKEY LIKE '493' OR STATIONOFEXITKEY LIKE '521' OR

STATIONOFEXITKEY LIKE '526' OR STATIONOFEXITKEY LIKE '538' OR
STATIONOFEXITKEY LIKE '587' OR STATIONOFEXITKEY LIKE '602' OR
STATIONOFEXITKEY LIKE '606' OR STATIONOFEXITKEY LIKE '608' OR
STATIONOFEXITKEY LIKE '610' OR STATIONOFEXITKEY LIKE '643' OR
STATIONOFEXITKEY LIKE '644' OR STATIONOFEXITKEY LIKE '662' OR
STATIONOFEXITKEY LIKE '670' OR STATIONOFEXITKEY LIKE '671' OR
STATIONOFEXITKEY LIKE '691' OR STATIONOFEXITKEY LIKE '701' OR
STATIONOFEXITKEY LIKE '708' OR STATIONOFEXITKEY LIKE '717' OR
STATIONOFEXITKEY LIKE '744' OR STATIONOFEXITKEY LIKE '745' OR
STATIONOFEXITKEY LIKE '752' OR STATIONOFEXITKEY LIKE '765' OR
STATIONOFEXITKEY LIKE '789' OR STATIONOFEXITKEY LIKE '791' OR
STATIONOFEXITKEY LIKE '822' OR STATIONOFEXITKEY LIKE '825' OR
STATIONOFEXITKEY LIKE '861' OR STATIONOFEXITKEY LIKE '870' OR
STATIONOFEXITKEY LIKE '871' OR STATIONOFEXITKEY LIKE '872' OR
STATIONOFEXITKEY LIKE '876' OR STATIONOFEXITKEY LIKE '882' OR
STATIONOFEXITKEY LIKE '903' OR STATIONOFEXITKEY LIKE '910' OR
STATIONOFEXITKEY LIKE '915' OR STATIONOFEXITKEY LIKE '918' OR
STATIONOFEXITKEY LIKE '930' OR STATIONOFEXITKEY LIKE '932' OR
STATIONOFEXITKEY LIKE '974' OR STATIONOFEXITKEY LIKE '981' OR
STATIONOFEXITKEY LIKE '1123' OR STATIONOFEXITKEY LIKE '1124' OR
STATIONOFEXITKEY LIKE '1293' OR STATIONOFEXITKEY LIKE '1294' OR
STATIONOFEXITKEY LIKE '1295' OR STATIONOFEXITKEY LIKE '1335' OR
STATIONOFEXITKEY LIKE '1337' OR STATIONOFEXITKEY LIKE '1338' OR
STATIONOFEXITKEY LIKE '1339' OR STATIONOFEXITKEY LIKE '1341' OR
STATIONOFEXITKEY LIKE '1342' OR STATIONOFEXITKEY LIKE '1343' OR
STATIONOFEXITKEY LIKE '1345' OR STATIONOFEXITKEY LIKE '1348' OR
STATIONOFEXITKEY LIKE '1349' OR STATIONOFEXITKEY LIKE '1350' OR
STATIONOFEXITKEY LIKE '1351' OR STATIONOFEXITKEY LIKE '1352' OR
STATIONOFEXITKEY LIKE '1355' OR STATIONOFEXITKEY LIKE '1356' OR
STATIONOFEXITKEY LIKE '1357' OR STATIONOFEXITKEY LIKE '1365' OR
STATIONOFEXITKEY LIKE '1366' OR STATIONOFEXITKEY LIKE '1367' OR
STATIONOFEXITKEY LIKE '1368' OR STATIONOFEXITKEY LIKE '1371' OR
STATIONOFEXITKEY LIKE '1378' OR STATIONOFEXITKEY LIKE '1409' OR
STATIONOFEXITKEY LIKE '1486' OR STATIONOFEXITKEY LIKE '1510' OR
STATIONOFEXITKEY LIKE '1521' OR STATIONOFEXITKEY LIKE '1572' OR
STATIONOFEXITKEY LIKE '1577' OR STATIONOFEXITKEY LIKE '1615' OR
STATIONOFEXITKEY LIKE '1616' OR STATIONOFEXITKEY LIKE '1617' OR
STATIONOFEXITKEY LIKE '1618' OR STATIONOFEXITKEY LIKE '1619' OR
STATIONOFEXITKEY LIKE '1620' OR STATIONOFEXITKEY LIKE '1679' OR
STATIONOFEXITKEY LIKE '1680' OR STATIONOFEXITKEY LIKE '1681' OR
STATIONOFEXITKEY LIKE '1682' OR STATIONOFEXITKEY LIKE '1683' OR
STATIONOFEXITKEY LIKE '1685' OR STATIONOFEXITKEY LIKE '1686' OR
STATIONOFEXITKEY LIKE '1688' OR STATIONOFEXITKEY LIKE '1731' OR
STATIONOFEXITKEY LIKE '1946' OR STATIONOFEXITKEY LIKE '1955' OR
STATIONOFEXITKEY LIKE '1956' OR STATIONOFEXITKEY LIKE '1958' OR
STATIONOFEXITKEY LIKE '1959' OR STATIONOFEXITKEY LIKE '1960' OR

STATIONOFEXITKEY LIKE '1961' OR STATIONOFEXITKEY LIKE '1962' OR
STATIONOFEXITKEY LIKE '1963' OR STATIONOFEXITKEY LIKE '1964' OR
STATIONOFEXITKEY LIKE '1965' OR STATIONOFEXITKEY LIKE '1966' OR
STATIONOFEXITKEY LIKE '1968' OR STATIONOFEXITKEY LIKE '1969' OR
STATIONOFEXITKEY LIKE '1971' OR STATIONOFEXITKEY LIKE '1972' OR
STATIONOFEXITKEY LIKE '1973' OR STATIONOFEXITKEY LIKE '1977' OR
STATIONOFEXITKEY LIKE '1980' OR STATIONOFEXITKEY LIKE '1984' OR
STATIONOFEXITKEY LIKE '1988' OR STATIONOFEXITKEY LIKE '1989' OR
STATIONOFEXITKEY LIKE '1995' OR STATIONOFEXITKEY LIKE '2000' OR
STATIONOFEXITKEY LIKE '2070' OR STATIONOFEXITKEY LIKE '2117' OR
STATIONOFEXITKEY LIKE '2118')")

Now that we have a final, filtered dataset which contains only the fields we want and journeys that take place only on tube or rail services, we can get the frequency distribution of key variables, as follows:

Get the frequency of journeys for each day

scala> val sqlFrame = spark.sql("SELECT DAYKEY, COUNT(DAYKEY) AS Frequency, COUNT(DAYKEY)/(SELECT COUNT(DAYKEY) FROM tfl_final)*100 AS Percentage FROM tfl_final GROUP BY DAYKEY ORDER BY Frequency DESC")

scala> sqlFrame.show(1000)

CREATING A SAMPLE DATASET

Now, based on the frequency distribution of journeys taken on each day, we can get a sample of records from the entire TfL dataset. To do so, we get a random sample of records from each day based on
the frequency of journeys occurring on each day, using the following queries:

scala> val sqlFrame = spark.sql("SELECT PRESTIGEID, PPTPASSENGERAGEKEY, DAYKEY, CARDTYPEKEY, STATIONOFFIRSTENTRYKEY, STATIONOFEXITKEY, TRANSACTIONTIME FROM tfl_final WHERE DAYKEY LIKE 12393 ORDER BY RAND() LIMIT 101443")

scala> val sqlFrame = spark.sql("SELECT PRESTIGEID, PPTPASSENGERAGEKEY, DAYKEY, CARDTYPEKEY, STATIONOFFIRSTENTRYKEY, STATIONOFEXITKEY, TRANSACTIONTIME FROM tfl_final WHERE DAYKEY LIKE 12393 ORDER BY RAND() LIMIT 101443")

```
scala> val sqlFrame = spark.sql("SELECT PRESTIGEID, PPTPASSENGERAGEKEY,
DAYKEY, CARDTYPEKEY, STATIONOFFIRSTENTRYKEY, STATIONOFEXITKEY,
TRANSACTIONTIME FROM tfl_final WHERE DAYKEY LIKE 12386 ORDER BY RAND()
LIMIT 101069")

scala> val sqlFrame = spark.sql("SELECT PRESTIGEID, PPTPASSENGERAGEKEY,
DAYKEY, CARDTYPEKEY, STATIONOFFIRSTENTRYKEY, STATIONOFEXITKEY,
TRANSACTIONTIME FROM tfl_final WHERE DAYKEY LIKE 12400 ORDER BY RAND()
LIMIT 99259")

scala> val sqlFrame = spark.sql("SELECT PRESTIGEID, PPTPASSENGERAGEKEY,
DAYKEY, CARDTYPEKEY, STATIONOFFIRSTENTRYKEY, STATIONOFEXITKEY,
TRANSACTIONTIME FROM tfl_final WHERE DAYKEY LIKE 12399 ORDER BY RAND()
LIMIT 97878")

scala> val sqlFrame = spark.sql("SELECT PRESTIGEID, PPTPASSENGERAGEKEY,
DAYKEY, CARDTYPEKEY, STATIONOFFIRSTENTRYKEY, STATIONOFEXITKEY,
TRANSACTIONTIME FROM tfl_final WHERE DAYKEY LIKE 12391 ORDER BY RAND()
LIMIT 97638")

scala> val sqlFrame = spark.sql("SELECT PRESTIGEID, PPTPASSENGERAGEKEY,
DAYKEY, CARDTYPEKEY, STATIONOFFIRSTENTRYKEY, STATIONOFEXITKEY,
TRANSACTIONTIME FROM tfl_final WHERE DAYKEY LIKE 12398 ORDER BY RAND()
LIMIT 97433")

scala> val sqlFrame = spark.sql("SELECT PRESTIGEID, PPTPASSENGERAGEKEY,
DAYKEY, CARDTYPEKEY, STATIONOFFIRSTENTRYKEY, STATIONOFEXITKEY,
TRANSACTIONTIME FROM tfl_final WHERE DAYKEY LIKE 12392 ORDER BY RAND()
LIMIT 96895")

scala> val sqlFrame = spark.sql("SELECT PRESTIGEID, PPTPASSENGERAGEKEY,
DAYKEY, CARDTYPEKEY, STATIONOFFIRSTENTRYKEY, STATIONOFEXITKEY,
TRANSACTIONTIME FROM tfl_final WHERE DAYKEY LIKE 12385 ORDER BY RAND()
LIMIT 96820")

scala> val sqlFrame = spark.sql("SELECT PRESTIGEID, PPTPASSENGERAGEKEY,
DAYKEY, CARDTYPEKEY, STATIONOFFIRSTENTRYKEY, STATIONOFEXITKEY,
TRANSACTIONTIME FROM tfl_final WHERE DAYKEY LIKE 12379 ORDER BY RAND()
LIMIT 95575")

scala> val sqlFrame = spark.sql("SELECT PRESTIGEID, PPTPASSENGERAGEKEY,
DAYKEY, CARDTYPEKEY, STATIONOFFIRSTENTRYKEY, STATIONOFEXITKEY,
TRANSACTIONTIME FROM tfl_final WHERE DAYKEY LIKE 12372 ORDER BY RAND()
LIMIT 95231")
```

```
scala> val sqlFrame = spark.sql("SELECT PRESTIGEID, PPTPASSENGERAGEKEY,
DAYKEY, CARDTYPEKEY, STATIONOFFIRSTENTRYKEY, STATIONOFEXITKEY,
TRANSACTIONTIME FROM tfl_final WHERE DAYKEY LIKE 12383 ORDER BY RAND()
LIMIT 94683")

scala> val sqlFrame = spark.sql("SELECT PRESTIGEID, PPTPASSENGERAGEKEY,
DAYKEY, CARDTYPEKEY, STATIONOFFIRSTENTRYKEY, STATIONOFEXITKEY,
TRANSACTIONTIME FROM tfl_final WHERE DAYKEY LIKE 12378 ORDER BY RAND()
LIMIT 94200")

scala> val sqlFrame = spark.sql("SELECT PRESTIGEID, PPTPASSENGERAGEKEY,
DAYKEY, CARDTYPEKEY, STATIONOFFIRSTENTRYKEY, STATIONOFEXITKEY,
TRANSACTIONTIME FROM tfl_final WHERE DAYKEY LIKE 12397 ORDER BY RAND()
LIMIT 93657")

scala> val sqlFrame = spark.sql("SELECT PRESTIGEID, PPTPASSENGERAGEKEY,
DAYKEY, CARDTYPEKEY, STATIONOFFIRSTENTRYKEY, STATIONOFEXITKEY,
TRANSACTIONTIME FROM tfl_final WHERE DAYKEY LIKE 12376 ORDER BY RAND()
LIMIT 92467")

scala> val sqlFrame = spark.sql("SELECT PRESTIGEID, PPTPASSENGERAGEKEY,
DAYKEY, CARDTYPEKEY, STATIONOFFIRSTENTRYKEY, STATIONOFEXITKEY,
TRANSACTIONTIME FROM tfl_final WHERE DAYKEY LIKE 12384 ORDER BY RAND()
LIMIT 92320")

scala> val sqlFrame = spark.sql("SELECT PRESTIGEID, PPTPASSENGERAGEKEY,
DAYKEY, CARDTYPEKEY, STATIONOFFIRSTENTRYKEY, STATIONOFEXITKEY,
TRANSACTIONTIME FROM tfl_final WHERE DAYKEY LIKE 12371 ORDER BY RAND()
LIMIT 92029")

scala> val sqlFrame = spark.sql("SELECT PRESTIGEID, PPTPASSENGERAGEKEY,
DAYKEY, CARDTYPEKEY, STATIONOFFIRSTENTRYKEY, STATIONOFEXITKEY,
TRANSACTIONTIME FROM tfl_final WHERE DAYKEY LIKE 12364 ORDER BY RAND()
LIMIT 91877")

scala> val sqlFrame = spark.sql("SELECT PRESTIGEID, PPTPASSENGERAGEKEY,
DAYKEY, CARDTYPEKEY, STATIONOFFIRSTENTRYKEY, STATIONOFEXITKEY,
TRANSACTIONTIME FROM tfl_final WHERE DAYKEY LIKE 12390 ORDER BY RAND()
LIMIT 91800")

scala> val sqlFrame = spark.sql("SELECT PRESTIGEID, PPTPASSENGERAGEKEY,
DAYKEY, CARDTYPEKEY, STATIONOFFIRSTENTRYKEY, STATIONOFEXITKEY,
TRANSACTIONTIME FROM tfl_final WHERE DAYKEY LIKE 12370 ORDER BY RAND()
LIMIT 90649")
```

```
scala> val sqlFrame = spark.sql("SELECT PRESTIGEID, PPTPASSENGERAGEKEY,
DAYKEY, CARDTYPEKEY, STATIONOFFIRSTENTRYKEY, STATIONOFEXITKEY,
TRANSACTIONTIME FROM tfl_final WHERE DAYKEY LIKE 12365 ORDER BY RAND()
LIMIT 90261")

scala> val sqlFrame = spark.sql("SELECT PRESTIGEID, PPTPASSENGERAGEKEY,
DAYKEY, CARDTYPEKEY, STATIONOFFIRSTENTRYKEY, STATIONOFEXITKEY,
TRANSACTIONTIME FROM tfl_final WHERE DAYKEY LIKE 12356 ORDER BY RAND()
LIMIT 90035")

scala> val sqlFrame = spark.sql("SELECT PRESTIGEID, PPTPASSENGERAGEKEY,
DAYKEY, CARDTYPEKEY, STATIONOFFIRSTENTRYKEY, STATIONOFEXITKEY,
TRANSACTIONTIME FROM tfl_final WHERE DAYKEY LIKE 12377 ORDER BY RAND()
LIMIT 89734")

scala> val sqlFrame = spark.sql("SELECT PRESTIGEID, PPTPASSENGERAGEKEY,
DAYKEY, CARDTYPEKEY, STATIONOFFIRSTENTRYKEY, STATIONOFEXITKEY,
TRANSACTIONTIME FROM tfl_final WHERE DAYKEY LIKE 12355 ORDER BY RAND()
LIMIT 89598")

scala> val sqlFrame = spark.sql("SELECT PRESTIGEID, PPTPASSENGERAGEKEY,
DAYKEY, CARDTYPEKEY, STATIONOFFIRSTENTRYKEY, STATIONOFEXITKEY,
TRANSACTIONTIME FROM tfl_final WHERE DAYKEY LIKE 12351 ORDER BY RAND()
LIMIT 89589")

scala> val sqlFrame = spark.sql("SELECT PRESTIGEID, PPTPASSENGERAGEKEY,
DAYKEY, CARDTYPEKEY, STATIONOFFIRSTENTRYKEY, STATIONOFEXITKEY,
TRANSACTIONTIME FROM tfl_final WHERE DAYKEY LIKE 12357 ORDER BY RAND()
LIMIT 89305")

scala> val sqlFrame = spark.sql("SELECT PRESTIGEID, PPTPASSENGERAGEKEY,
DAYKEY, CARDTYPEKEY, STATIONOFFIRSTENTRYKEY, STATIONOFEXITKEY,
TRANSACTIONTIME FROM tfl_final WHERE DAYKEY LIKE 12363 ORDER BY RAND()
LIMIT 88787")

scala> val sqlFrame = spark.sql("SELECT PRESTIGEID, PPTPASSENGERAGEKEY,
DAYKEY, CARDTYPEKEY, STATIONOFFIRSTENTRYKEY, STATIONOFEXITKEY,
TRANSACTIONTIME FROM tfl_final WHERE DAYKEY LIKE 12369 ORDER BY RAND()
LIMIT 88752")

scala> val sqlFrame = spark.sql("SELECT PRESTIGEID, PPTPASSENGERAGEKEY,
DAYKEY, CARDTYPEKEY, STATIONOFFIRSTENTRYKEY, STATIONOFEXITKEY,
TRANSACTIONTIME FROM tfl_final WHERE DAYKEY LIKE 12358 ORDER BY RAND()
LIMIT 88577")
```

```
scala> val sqlFrame = spark.sql("SELECT PRESTIGEID, PPTPASSENGERAGEKEY,
DAYKEY, CARDTYPEKEY, STATIONOFFIRSTENTRYKEY, STATIONOFEXITKEY,
TRANSACTIONTIME FROM tfl_final WHERE DAYKEY LIKE 12350 ORDER BY RAND()
LIMIT 88562")

scala> val sqlFrame = spark.sql("SELECT PRESTIGEID, PPTPASSENGERAGEKEY,
DAYKEY, CARDTYPEKEY, STATIONOFFIRSTENTRYKEY, STATIONOFEXITKEY,
TRANSACTIONTIME FROM tfl_final WHERE DAYKEY LIKE 12362 ORDER BY RAND()
LIMIT 86968")

scala> val sqlFrame = spark.sql("SELECT PRESTIGEID, PPTPASSENGERAGEKEY,
DAYKEY, CARDTYPEKEY, STATIONOFFIRSTENTRYKEY, STATIONOFEXITKEY,
TRANSACTIONTIME FROM tfl_final WHERE DAYKEY LIKE 12348 ORDER BY RAND()
LIMIT 86673")

scala> val sqlFrame = spark.sql("SELECT PRESTIGEID, PPTPASSENGERAGEKEY,
DAYKEY, CARDTYPEKEY, STATIONOFFIRSTENTRYKEY, STATIONOFEXITKEY,
TRANSACTIONTIME FROM tfl_final WHERE DAYKEY LIKE 12349 ORDER BY RAND()
LIMIT 86623")

scala> val sqlFrame = spark.sql("SELECT PRESTIGEID, PPTPASSENGERAGEKEY,
DAYKEY, CARDTYPEKEY, STATIONOFFIRSTENTRYKEY, STATIONOFEXITKEY,
TRANSACTIONTIME FROM tfl_final WHERE DAYKEY LIKE 12396 ORDER BY RAND()
LIMIT 85164")

scala> val sqlFrame = spark.sql("SELECT PRESTIGEID, PPTPASSENGERAGEKEY,
DAYKEY, CARDTYPEKEY, STATIONOFFIRSTENTRYKEY, STATIONOFEXITKEY,
TRANSACTIONTIME FROM tfl_final WHERE DAYKEY LIKE 12389 ORDER BY RAND()
LIMIT 84522")

scala> val sqlFrame = spark.sql("SELECT PRESTIGEID, PPTPASSENGERAGEKEY,
DAYKEY, CARDTYPEKEY, STATIONOFFIRSTENTRYKEY, STATIONOFEXITKEY,
TRANSACTIONTIME FROM tfl_final WHERE DAYKEY LIKE 12382 ORDER BY RAND()
LIMIT 81450")

scala> val sqlFrame = spark.sql("SELECT PRESTIGEID, PPTPASSENGERAGEKEY,
DAYKEY, CARDTYPEKEY, STATIONOFFIRSTENTRYKEY, STATIONOFEXITKEY,
TRANSACTIONTIME FROM tfl_final WHERE DAYKEY LIKE 12368 ORDER BY RAND()
LIMIT 80320")

scala> val sqlFrame = spark.sql("SELECT PRESTIGEID, PPTPASSENGERAGEKEY,
DAYKEY, CARDTYPEKEY, STATIONOFFIRSTENTRYKEY, STATIONOFEXITKEY,
TRANSACTIONTIME FROM tfl_final WHERE DAYKEY LIKE 12375 ORDER BY RAND()
LIMIT 79487")
```

```
scala> val sqlFrame = spark.sql("SELECT PRESTIGEID, PPTPASSENGERAGEKEY,
DAYKEY, CARDTYPEKEY, STATIONOFFIRSTENTRYKEY, STATIONOFEXITKEY,
TRANSACTIONTIME FROM tfl_final WHERE DAYKEY LIKE 12347 ORDER BY RAND()
LIMIT 78539")

scala> val sqlFrame = spark.sql("SELECT PRESTIGEID, PPTPASSENGERAGEKEY,
DAYKEY, CARDTYPEKEY, STATIONOFFIRSTENTRYKEY, STATIONOFEXITKEY,
TRANSACTIONTIME FROM tfl_final WHERE DAYKEY LIKE 12361 ORDER BY RAND()
LIMIT 77558")

scala> val sqlFrame = spark.sql("SELECT PRESTIGEID, PPTPASSENGERAGEKEY,
DAYKEY, CARDTYPEKEY, STATIONOFFIRSTENTRYKEY, STATIONOFEXITKEY,
TRANSACTIONTIME FROM tfl_final WHERE DAYKEY LIKE 12354 ORDER BY RAND()
LIMIT 68538")

scala> val sqlFrame = spark.sql("SELECT PRESTIGEID, PPTPASSENGERAGEKEY,
DAYKEY, CARDTYPEKEY, STATIONOFFIRSTENTRYKEY, STATIONOFEXITKEY,
TRANSACTIONTIME FROM tfl_final WHERE DAYKEY LIKE 12387 ORDER BY RAND()
LIMIT 50084")

scala> val sqlFrame = spark.sql("SELECT PRESTIGEID, PPTPASSENGERAGEKEY,
DAYKEY, CARDTYPEKEY, STATIONOFFIRSTENTRYKEY, STATIONOFEXITKEY,
TRANSACTIONTIME FROM tfl_final WHERE DAYKEY LIKE 12401 ORDER BY RAND()
LIMIT 50006")

scala> val sqlFrame = spark.sql("SELECT PRESTIGEID, PPTPASSENGERAGEKEY,
DAYKEY, CARDTYPEKEY, STATIONOFFIRSTENTRYKEY, STATIONOFEXITKEY,
TRANSACTIONTIME FROM tfl_final WHERE DAYKEY LIKE 12394 ORDER BY RAND()
LIMIT 47452")

scala> val sqlFrame = spark.sql("SELECT PRESTIGEID, PPTPASSENGERAGEKEY,
DAYKEY, CARDTYPEKEY, STATIONOFFIRSTENTRYKEY, STATIONOFEXITKEY,
TRANSACTIONTIME FROM tfl_final WHERE DAYKEY LIKE 12373 ORDER BY RAND()
LIMIT 40869")

scala> val sqlFrame = spark.sql("SELECT PRESTIGEID, PPTPASSENGERAGEKEY,
DAYKEY, CARDTYPEKEY, STATIONOFFIRSTENTRYKEY, STATIONOFEXITKEY,
TRANSACTIONTIME FROM tfl_final WHERE DAYKEY LIKE 12366 ORDER BY RAND()
LIMIT 39785")

scala> val sqlFrame = spark.sql("SELECT PRESTIGEID, PPTPASSENGERAGEKEY,
DAYKEY, CARDTYPEKEY, STATIONOFFIRSTENTRYKEY, STATIONOFEXITKEY,
TRANSACTIONTIME FROM tfl_final WHERE DAYKEY LIKE 12359 ORDER BY RAND()
LIMIT 39436")
```

```
scala> val sqlFrame = spark.sql("SELECT PRESTIGEID, PPTPASSENGERAGEKEY,
DAYKEY, CARDTYPEKEY, STATIONOFFIRSTENTRYKEY, STATIONOFEXITKEY,
TRANSACTIONTIME FROM tfl_final WHERE DAYKEY LIKE 12352 ORDER BY RAND()
LIMIT 38013")

scala> val sqlFrame = spark.sql("SELECT PRESTIGEID, PPTPASSENGERAGEKEY,
DAYKEY, CARDTYPEKEY, STATIONOFFIRSTENTRYKEY, STATIONOFEXITKEY,
TRANSACTIONTIME FROM tfl_final WHERE DAYKEY LIKE 12380 ORDER BY RAND()
LIMIT 36506")

scala> val sqlFrame = spark.sql("SELECT PRESTIGEID, PPTPASSENGERAGEKEY,
DAYKEY, CARDTYPEKEY, STATIONOFFIRSTENTRYKEY, STATIONOFEXITKEY,
TRANSACTIONTIME FROM tfl_final WHERE DAYKEY LIKE 12395 ORDER BY RAND()
LIMIT 23930")

scala> val sqlFrame = spark.sql("SELECT PRESTIGEID, PPTPASSENGERAGEKEY,
DAYKEY, CARDTYPEKEY, STATIONOFFIRSTENTRYKEY, STATIONOFEXITKEY,
TRANSACTIONTIME FROM tfl_final WHERE DAYKEY LIKE 12388 ORDER BY RAND()
LIMIT 22239")

scala> val sqlFrame = spark.sql("SELECT PRESTIGEID, PPTPASSENGERAGEKEY,
DAYKEY, CARDTYPEKEY, STATIONOFFIRSTENTRYKEY, STATIONOFEXITKEY,
TRANSACTIONTIME FROM tfl_final WHERE DAYKEY LIKE 12353 ORDER BY RAND()
LIMIT 19777")

scala> val sqlFrame = spark.sql("SELECT PRESTIGEID, PPTPASSENGERAGEKEY,
DAYKEY, CARDTYPEKEY, STATIONOFFIRSTENTRYKEY, STATIONOFEXITKEY,
TRANSACTIONTIME FROM tfl_final WHERE DAYKEY LIKE 12367 ORDER BY RAND()
LIMIT 18227")

scala> val sqlFrame = spark.sql("SELECT PRESTIGEID, PPTPASSENGERAGEKEY,
DAYKEY, CARDTYPEKEY, STATIONOFFIRSTENTRYKEY, STATIONOFEXITKEY,
TRANSACTIONTIME FROM tfl_final WHERE DAYKEY LIKE 12381 ORDER BY RAND()
LIMIT 17468")

scala> val sqlFrame = spark.sql("SELECT PRESTIGEID, PPTPASSENGERAGEKEY,
DAYKEY, CARDTYPEKEY, STATIONOFFIRSTENTRYKEY, STATIONOFEXITKEY,
TRANSACTIONTIME FROM tfl_final WHERE DAYKEY LIKE 12374 ORDER BY RAND()
LIMIT 16828")

scala> val sqlFrame = spark.sql("SELECT PRESTIGEID, PPTPASSENGERAGEKEY,
DAYKEY, CARDTYPEKEY, STATIONOFFIRSTENTRYKEY, STATIONOFEXITKEY,
TRANSACTIONTIME FROM tfl_final WHERE DAYKEY LIKE 12360 ORDER BY RAND()
LIMIT 16150")
```

```
scala> val sqlFrame = spark.sql("SELECT PRESTIGEID, PPTPASSENGERAGEKEY,
DAYKEY, CARDTYPEKEY, STATIONOFFIRSTENTRYKEY, STATIONOFEXITKEY,
TRANSACTIONTIME FROM tfl_final WHERE DAYKEY LIKE 12346 ORDER BY RAND()
LIMIT 13334")
```

5.3: SQL Code

The following queries were carried out on the SQL database designed by Patrick Martins-Yedenu.

Get total number of records

```
mysql> SELECT COUNT(*) FROM stg_txntable;
```

Get total number of distinct passengers

```
mysql> SELECT COUNT(DISTINCT prestigeid) FROM stg_txntable;
```

Get a description of the entire dataset

```
mysql> DESCRIBE stg_txntable;
```

Get frequency of journeys based on passenger age (as recorded by TfL)

```
mysql> SELECT pptpassengeragekey, COUNT(pptpassengeragekey) AS Frequency,
COUNT(pptpassengeragekey)/(SELECT COUNT(pptpassengeragekey) FROM
stg_txntable)*100 AS Percentage FROM stg_txntable GROUP BY pptpassengeragekey ORDER
BY Frequency DESC;
```

Get frequency of journeys based on Oyster card type

```
mysql> SELECT cardtypekey, COUNT(cardtypekey) AS Frequency,
COUNT(cardtypekey)/(SELECT COUNT(cardtypekey) FROM stg_txntable)*100 AS
Percentage FROM stg_txntable GROUP BY cardtypekey ORDER BY Frequency DESC;
```

Get frequency of journeys based on station of entry

```
mysql> SELECT stationoffirstentrykey, COUNT(stationoffirstentrykey) AS Frequency,
COUNT(stationoffirstentrykey)/(SELECT COUNT(stationoffirstentrykey) FROM
stg_txntable)*100 AS Percentage FROM stg_txntable GROUP BY stationoffirstentrykey
ORDER BY Frequency DESC;
```

Get frequency of journeys based on station of exit

```
mysql> SELECT stationofexitkey, COUNT(stationofexitkey) AS Frequency,
COUNT(stationofexitkey)/(SELECT COUNT(stationofexitkey) FROM stg_txntable)*100 AS
Percentage FROM stg_txntable GROUP BY stationofexitkey ORDER BY Frequency DESC;
```

Get frequency of journeys based on the day of the week

```
mysql> SELECT daytype, COUNT(daytype) AS Frequency, COUNT(daytype)/(SELECT
COUNT(daytype) FROM stg_txntable)*100 AS Percentage FROM stg_txntable GROUP BY
daytype ORDER BY Frequency DESC;
```