Documentation (serial numbers reference R code chunks in Rmd file).

**1.1 Reading the csv file and saving it as Parquet format**

We converted our dataset, "transactions.csv", to the Parquet format using *arrow::open_dataset() and* saved it to the “parquet_folder” which we directory created using *dir.create()*. The parquet file is **transaction_parquet**.

**1.2 Opening the parquet file and creating a reference in Spark**

We then opened **transaction_parquet** file using *arrow::open_dataset()* function, and collected it into an R dataframe named **transaction_df** using the *collect()* from the loaded dplyr package. Subsequently, we connected to a local Spark cluster using *spark_connect()*, after loading the sparklyr package. **Transaction_df** is then copied into Spark memory, creating a Spark dataframe called **transaction_ref**.

**2.1 Data Checking**

2.1.1) Computed summary statistics: count, mean, standard deviation, minimum and maximum values for every column in **transaction_ref** using *sdf_describe()*.

2.1.2) Calculate the number of NA values in each column of the **transaction_ref** using *summarise_all()*, with its argument as a sum of null values .

2.1.3) Identify unique products, represented by the ProductNames variable, of **transaction_ref** using *select()* and *distinct()* functions.

2.1.4) Filtered and displayed rows where the Quantity column has negative values using the *filter()* function.

2.1.5) Filtered and displayed rows where the Quantity column has outliers, values greater than 10,000, using *filter()*.

### 2.2 Data Cleaning

2.2.1) Removed rows with missing or null values in the CustomerNo column using the *filter()* with *is.na()* and *is.null()*. Rows with quantities that exceeded 10000 or had negative values were also removed.

2.2.2) The dates were originally characters. We converted it to date format by splitting the day, month and year portion using *substring_index()*, added leading zeroes to the month and day values using *lpad()* and subsequently, concatenated the modified year, month and day with *concat()*. Lastly, we used *to_date()* to convert the column to be of date type. The result is stored in **transaction_clean**.

### 3.1 EDA - Geographical Analysis

Create a bar plot using *dbplot_bar()* from the dbplot package to visualise the number of customers in different countries, majority of which were from the United Kingdom.

### 3.2 EDA - Visualizing Transaction Volume by Month

A new spark dataframe, **temp_df1** derives monthly transaction counts from **transaction_clean**, summarizing unique transactions per month using *summarize()*, and collected results into the R environment using *collect()*. A line plot is created using *ggplot()* and *geom_line()* from the ggplot2 package to visualize **temp_df1**.

### 3.3 EDA - Average Spend per Transaction

A new spark dataframe, **temp_df2** calculates average spend per transaction for each customer in **transaction_clean** using *summarize()*. We described the summary statistics on **temp_df2** using *sdf_describe()* and subsequently filtered for values less than 25000 using *filter()*. Using *collect()* we bring back results to R for

visualisation. A histogram is created using *ggplot()* and *geom_histogram()* to visualise the distribution and assess high variability.

### 3.4 EDA - Number of Unique Products Bought per Customer

A new spark dataframe, **temp_df3** computes the number of unique products bought per customer in **transaction_clean**. We obtained the summary statistics on **temp_df3** using *sdf_describe()*, filtered for values below 500, and collected the results for visualisation in R. Using *ggplot()* and *geom_histogram()*, a histogram is created to visualise the distribution of unique products bought per customer

### 3.5 EDA - Average Basket Size

A new spark dataframe, **temp_df4** calculates the average basket size per customer in **transaction_clean**, We obtained the summary statistics on **temp_df4** using *sdf_describe()*, filtered for values below 5000, and collected the results for visualisation in R. A histogram is created using *ggplot()* and *geom_histogram()* to visualise the distribution of average basket size per customer.

### 4.1 Feature Engineering for ref_customer

A Spark dataframe, **ref_customer,** is created by first filtering out customers in the United Kingdom from **transaction_clean**. We then grouped it by the unique customer and using *summarise()* to feature engineer RFM variables, duration, unique_products_bought, average_basketsize, and average_spend_per_trxcn. Additionally, a dataframe **temp_df5** is generated to identify the month each customer spent the most, by summarising the monthly spending and filtering for the customers' highest spending month. The resulting dataframe is then left-joined with **ref_customer**, and a new column, **festive_spender**, is created to categorize

customers based on whether they spent the most from October to December. Lastly, a target variable loyal_customer is engineered by comparing customer durations to the median duration. The result is stored in the dataframe **ref_customer (**Figure 11).

**5.1) Logistic Regression Model in Spark**

5.1.1) Using *select()* and *correlate()*, we created **corr_matrix** which is the correlation matrix for numeric variables in Model_2. We plotted it using *rplot()* and *shave()*.

5.1.2) We partition the **ref_customer** dataset into training (80%) and testing (20%) sets using the *sdf_random_split()* function and stored in the **ref_customer_split** object. Dataframes, **ref_customer_split_train** and **ref_customer_split_test**, are created to represent the training and testing sets, respectively.

5.1.3) We calculate the mean and standard deviation for the training set variables using *summarize()*. These statistics are stored in **ref_customer_stats** and applied to standardise numeric variables in the training and testing sets. This involves creating standardised variables such as R_standardized, F_standardized, M_standardized, unique_products_standardized, and average_basket_size_standardized.

5.1.4) We ran a logistic regression model using *ml_generalized_linear_regression()* from the training set, obtaining coefficient estimates stored in **tidy_glm_fit**. A confidence interval (CI) of **tidy_glm_fit** is plotted using *ggplot()*.

5.1.5) Two logistic regression models are fitted using the training set. **Model_1** uses only standardised RFM variables. **Model_2** includes the standardised RFM variables, festive_spender, average_basket_size_standardized and unique_products_standardized. With *ml_evaluate(),* we assessed the performance using the test set and selected **Model_2** as it had a higher AROC value.

### 5.2. ML Pipelines for Logistic Regression

5.2.1) We then built a ML pipeline model, **logistic_pipeline**, using *ml_pipeline()*. In the first stage, we used *ft_string_indexer()* to convert the festive_spender variable to an index column and encoded it via *ft_one_hot_encoder()* in the second stage. In the third stage, we selected the numeric variables present in **Model_2** using ft_vector_assembler() and standardised them using *ft_standard_scaler()* in the fourth stage. In the 5th stage, we assembled all variables into a single vector using ft_vector_assembler(). Lastly, we ran our logistic regression for loyal_customer using *ml_logistic_regression()*.

5.2.2) Using ml_cross_validator(), we created a cross-validation ML pipeline, **cv**, for **logistic_pipeline** with alpha and lambda values ranging from 0 to 1 and 0 to 0.1 **cv** used 10-fold cross-validation with a set seed value of 1337. We fit **cv on the training data**, using *ml_fit(),* and stored the result to **cv_model**. Using *ml_validation_metrics(), arrange(), desc()*, we found the values of lambda and alpha that gave us the highest AROC value. We saved the best model using *ml_save()* to the path “spark_model”.

### 5.3 Documentation for K-means clustering Spark model

5.3.1) We create object **k_values** which is a vector of numbers 2,3,4,5,6. Using *numeric()* and *length()*, we produce a placeholder vector called **silhouette_scores**. We then ran a loop function for the different values within **k_values** for the K-means clustering model using the *ml_kmeans()* function and their respective silhouette scores via *ml_compute_silhoutte_measure()*. The k-means clustering model, which was labelled as **k_means_model**, used the **ref_customer_for_kmeans** dataframe has the predictors R_standardized, F_standardized, and M_standardized with the

maximum iterations being 1000, set seed value and the initial cluster centroids being randomly assigned.**ref_customer_for_kmeans** builds on **ref_customer** where we added on the standardised RFM variables. The computation of the silhouette scores which, **silhouette_scores,** utilised **k_means_model**, the **ref_customer** as the database. Using the *data.frame()*, we created a dataframe called **silhouette_data** that shows the silhouette scores for the varying cluster numbers. Using the *ggplot(), geom_line() and theme_minimal()* functions, we plotted **silhouette_data** with the silhouette scores against the respective number of clusters.

**5.4 Documentation for K-means clustering Machine Learning pipeline**

5.4.1) We then built a three-stage ML pipeline model, **kmeans_pipeline**, using *ml_pipeline()*. In the first stage, we used *ft_vector_assembler()* to select the RFM variables to create a feature vector called “features”. In the second stage, we used *ft_standard_scaler()* to standardise “features” into “features_stdz”. In the last stage. we used *ml_kmeans()* function to generate our 4 clusters under the column “cluster” with the same specifications as **k_means_model**. We fitted **kmeans_pipeline** and **ref_customer** to a pipeline model, **kmeans_pipeline_model** via *ml_fit()*

5.4.3) Using *ml_transform()* on **kmeans_pipeline_model** and **ref_customer**, we calculated our model predictions, **predictions**. We sorted **prediction** by the *group_by()*, *summarise()* and *arrange()* functions to show the average RFM values, represented by **mean_recency**, **mean_frequency** and **mean_monetary**, for each cluster.