NPD Group Capstone Final Report

Frank Kovacs, Ning Gao, Pragya Jain, Wonil Lee *

May 17, 2021

Abstract

The main scope of the project was to develop two anomaly detecting algorithms to identify and report data collection errors for the 3 key variables provided by the NPD group: `receipt_count`, `sum_total_paid`, and `item_total`. The First Differences Filter calculated the desired quantile of the slopes in the selected window size to check if future slopes lied between the calculated quantile. The Robust Linear Regression Filter (RLM) predicted a series of future points by fitting a local linear regression to points within a defined window size, and utilized Median Absolute Deviation to compute standard deviations, which served as the anomaly boundaries. The 4 moving sequential filters implemented in RLM ensured that both the slow drift and sudden patterns could be detected in time series data. The final output for the project was a series of visualization plots and a customized table of anomalies detected for different merchants, outputted using a Python package interface.

Keywords. time series anomaly detection, window size, median absolute deviation, robust filters

Contents

1 Introduction **2**

2 Data **3**
- 2.1 Data Sets . . . 3
- 2.2 Variable Selection . . . 3
- 2.3 Exploratory Data Analysis . . . 4

3 Methods **5**
- 3.1 First Differences Filter . . . 5
- 3.2 Robust Linear Regression Filter . . . 7
  - 3.2.1 Robust Linear Regression Model . . . 7
  - 3.2.2 Robust Estimate for Standard Deviation . . . 7
  - 3.2.3 Sequential Filters . . . 8
- 3.3 Trimmed Moving Weighted Average Filter . . . 8
  - 3.3.1 Robust Estimate for Weighted Average . . . 9
  - 3.3.2 Robust Estimate for Standard Deviation . . . 9

4 Results **10**
- 4.1 First Differences Filter - Anomaly Detection Sample Results . . . 10
- 4.2 Robust Linear Model Filter - Anomaly Detection Sample Results . . . 11

*All authors are from the Department of Statistics & Data Science, Carnegie Mellon University, Pittsburgh, PA, 15213 USA. Emails: fkovacs@alumni.cmu.edu, ningg@andrew.cmu.edu, pragyaj@andrew.cmu.edu, and wonillee@andrew.cmu.edu. Technical advisor: Valerie Ventura. Email: vventura@andrew.cmu.edu.

5 Discussion 14
5.1 Filter Comparisons 14
5.2 Limitations and Improvements 14
5.2.1 First Difference Filter 14
5.2.2 RLM Filter 15
5.3 Next Steps and Conclusion 16
5.3.1 Additional Response Variables 16
5.3.2 Other Checks on Response Columns 16
5.3.3 Removal of Identified Outliers 17
5.3.4 Boundaries Smoothing 17
5.3.5 Conclusion 17

6 Reference 18

7 Technical Appendices 19
7.1 EDA and Variable Selection 19
7.1.1 Data Preparation and Wrangling 19
7.1.2 Time Series Visualization 19
7.1.3 Variable Selection 20
7.2 Sanity check of the First Difference Filter 20
7.2.1 First Difference Data Set 20
7.2.2 Histogram and Quantile Calculations 20
7.3 Trimmed Moving Weighted Average Filter 22
7.3.1 Input Parameters Sanity Check 22
7.3.2 Deletion of Outliers in a Given Window 22
7.3.3 Half Normal PDF Weights Assignment 23
7.3.4 Weighted Average Columns 23
7.3.5 Standard Deviation Columns 23
7.3.6 Defining Bounds for Anomaly Detection 24
7.3.7 Detection of Outliers based on 4 filters 24
7.3.8 Anomaly Detection Method 25
7.3.9 Anomaly Detection Visualization 26
7.4 Python Interface 27
7.4.1 Data Preparation 27
7.4.2 Base Filter Class 29
7.4.3 First Difference Class 29
7.4.4 Robust Linear Regression Class 30
7.4.5 Moving Weighted Average 32

1 Introduction

Our client, the NPD Group, Inc, is a market research company. Being a leader in the industry, NPD deals with more than 8 billion B2B transactions per year. One aspect of their mission includes creating reliable raw data assets for their end clients. These data sets are useful for their clients to strategize on increasing their brand's market share, identifying significant trends in sales, and understanding customer behavior. As a result, ensuring the quality of data resources becomes critical for the NPD data science team. This report aims to assist the team in detecting existing data collection anomalies. The detected anomalies will then be taken a closer look by their data analysts to ensure provision of error-free data to their end clients such as big retailers.

The project objective is to develop a python package (Appendix 7.4) with implementations of two data-driven filtering methods to identify different types of anomalies in data and facilitate error-free data delivery. The report presents an automated anomaly flagging process which can be run weekly to flag new anomalies in the incoming data series by NPD. The report proposes solutions for the following 3 research questions:

1. Detection of unexpected shifts in incoming data series for 3 key variables: `receipt_count`, `sum_total_paid`, and `item_total`. Proposed methods should detect anomalies in addition to errors already flagged by the client.

2. Ability to detect both rapid changes occurred within one week and slow drifts spanned over larger time periods such as one month.

3. Detection of both dips and peaks with an emphasis on dips per NPD request.

2 Data

2.1 Data Sets

The data used for the project were provided by NPD. There were 6 candidate response variables that we were interested in using, which were `receipt_count`, `sum_total_paid`, `sum_items_distinct`, `sum_item_spend`, `item_total` and `panelists` (Table 1).

There were overall three `csv` data sets given: `source data`, `retail data`, and `issue data`. The source data set consisted of 516 rows and 8 columns, it provided weekly values of the key variables listed above by 4 different data source types which were iPhone, Android, Sift, and Receipt pal on device. The retail data set, consisted of 983,953 rows and 11 columns, it provided weekly values of the key variables per each merchant as well as information about the merchant name/ID and data acquire type/ID. Finally, the issue data set consisted of 31 rows and 5 columns, it provided information about when (the Acquired date) and where (merchant name and source type) the data collecting error had occurred.

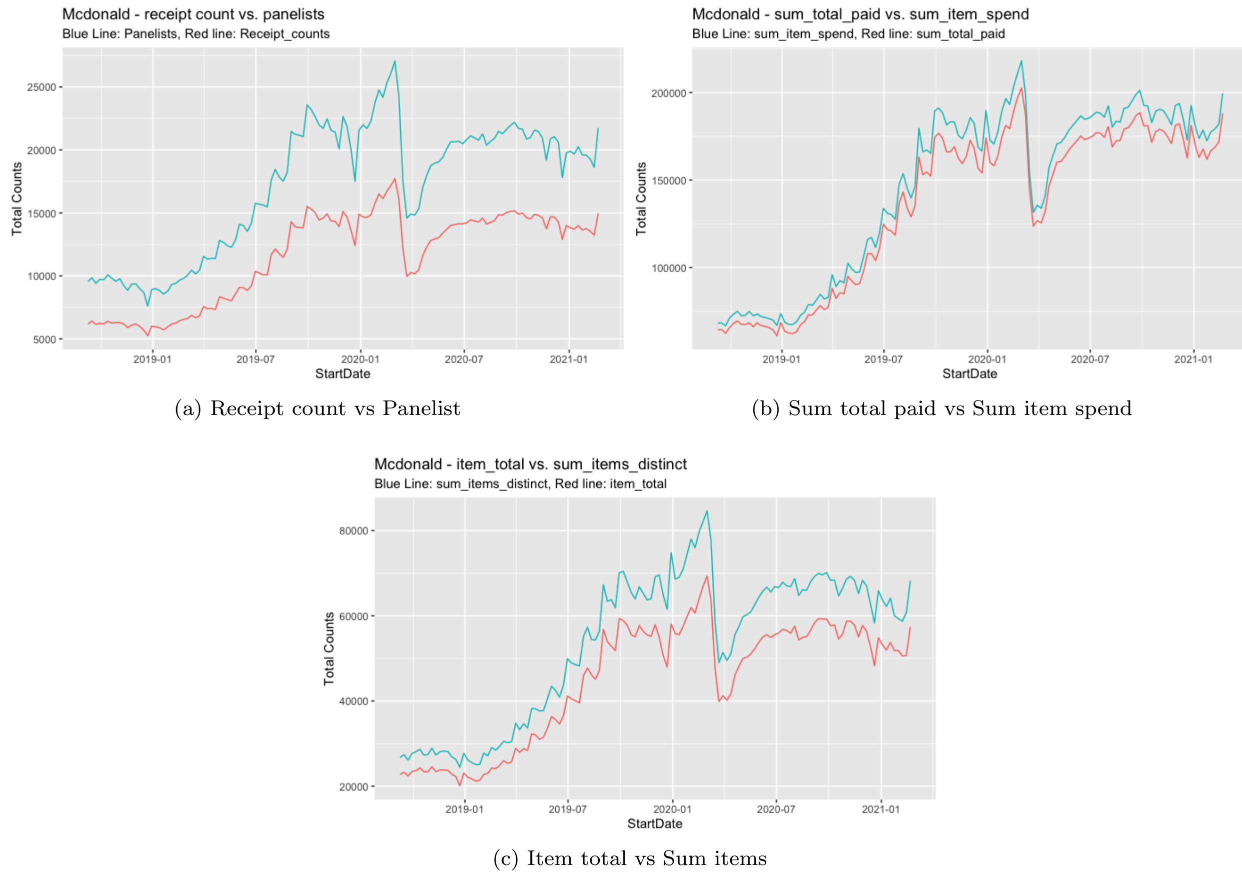
Variable Name	Values	Description
receipt_count	Integer	Weekly value of total number of receipt
Sum_total_paid	Integer	Weekly value of total amount paid in USD
sum_items_distinct	Integer	Weekly value of total number of distinct items
sum_item_spend	Integer	Weekly value of total number of items spent
panelists	Integer	Weekly value of total participants
item total	Integer	Weekly value of total items purchased
MerchantName	Text Labels	Name of the merchant where the transaction occurred
MerchantID	Integer	Unique identifier for each merchant
AcquireTypeDesc	Text Labels	Source where the data were acquired *(iPhone, Android, Sift, Receipt pal on Device)*
AcquireTypeID	Integer	Unique identifier for each data source

Table 1: Key Response Variables Description

2.2 Variable Selection

After taking a closer look at the data description provided from NPD, our team explored whether there were correlations between some of the key response variables, with detailed explanation provided in Appendix 7.1. To elaborate, suspected that `receipt_count`, which was the total number of transaction in given week might be closely associated with `panelist` which was the total number of participant in a given week. Similarly, `sum_total_paid`, which was the dollar amount paid in a given week might be closely associated with `sum_item_spend` which was the total number of items purchased. And the same logic went to `item_total` and `sum_items_distinct.`

In order to test our hypothesis, we first plotted the time series of the variables in pairs to check if the general trends of the two variables matched. As shown in Figure 1 below, the three pairs of response variables all showed highly similar trends.



(a) Receipt count vs Panelist

(b) Sum total paid vs Sum item spend

(c) Item total vs Sum items

Figure 1: Correlation Visualizations of Paired Variables: McDonald Case

After checking that the trends for each pair of variables were similar, our team calculated the average correlations between each pair for all merchants (Appendix 7.1.3). The calculated correlations in Table 2 strongly supported our hypothesis since all three correlations had a score over 0.99. With both visualization and mathematical backups, our team decided to use only the three variable, `receipt_count`, `sum_total_paid`, and `item_total`, as our key response variables for the rest of the project.

Pair of Columns	Correlation
Receipt_count & Panelists	**0.9937866**
Sum_total_paid & Sum_item_spend	**0.9990581**
Item_total & Sum_items_distinct	**0.9991476**

Table 2: Correlations of Paired Variables

2.3 Exploratory Data Analysis

Since our client could only provide 31 issues logged by them in the past 2 years, our team decided to use it as a guide to understand the types of errors their algorithm could detect and also discover hidden errors that might worth detecting. Exploratory Data Analysis (Appendix 7.1.1 and 7.1.2) was conducted with the goal of assessing possible error types existing in the data set and to compare possible errors with errors flagged by the client. Un-flagged errors were then studied and discussed with the client in order to discover client's preferences in detecting errors.

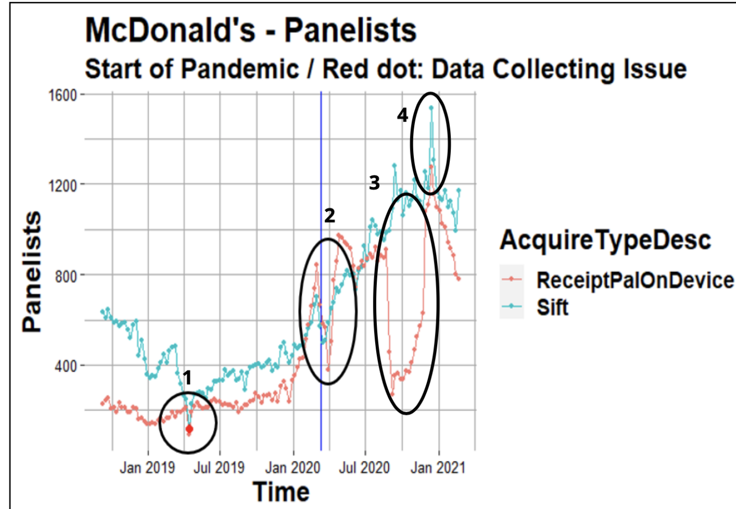


Figure 2: EDA Plot for Merchant: McDonald's

Figure 2 illustrates one compact example for error analysis on merchant 'McDonald's'. The red dot at Circle 1 indicated a flagged error, which was a rapid drop marked by NPD. When looking at Circle 1 and 3 at the same time, 3 had a larger rapid drop compared to 1, yet not detected. The observation at Circle 2 happened around the outbreak of COVID-19 pandemic on March 11th, 2020 [1], which was marked using a blue vertical line. This drop happened across more than a month, instead of over a short period of time like 1, but the dropping trend was not detected. Cirlce 4 contained a sudden peak. Even though the amplitude change of this jump was definitely greater than Circle 1, it was not detected, which brought the question of whether NPD had a preference of marking dips over peaks. This compact example was only one out of many EDA plots generated, please refer to our EDA section in Technical Appendix 7.1.2 for detailed error visualizations.

3 Methods

3.1 First Differences Filter

The first method our team implemented to detect anomalies in the time series was the First Differences Filtering method. The differences being calculated were the weekly changes from time `t` to `t+1`, or essentially the slope of change. Once the differences were calculated, the large absolute values, which represents the large sudden change from time `t` to `t+1`, could then be spotted. Ultimately, this allowed the user to detect anomalies based on the past trend of slopes. The logic of this method mainly consisted of two concepts: shifting windows and quantile calculation for the normal distribution (Appendix 7.2).

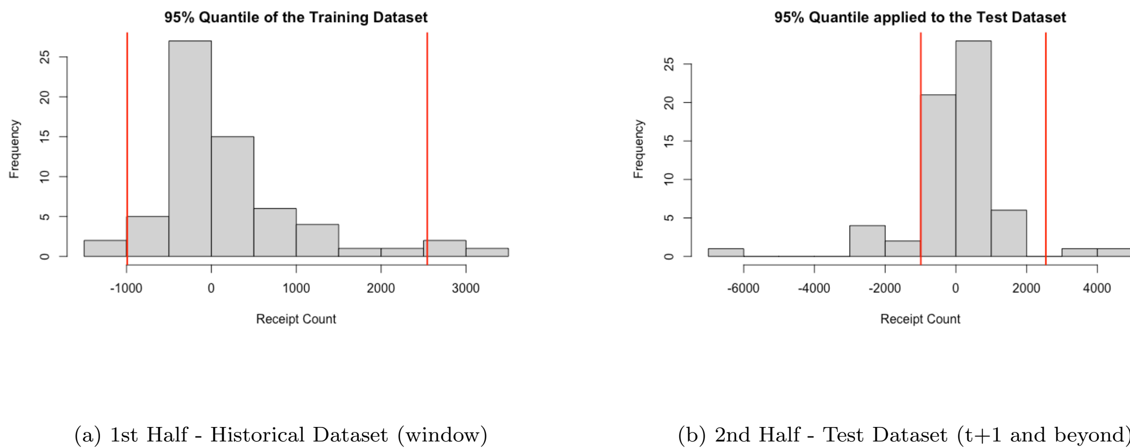
To begin with, the shifting window was a concept of selecting proper range of historical data points to be analyzed at time `t`, in order to detect anomaly at time `t+1`. The default window size selected for the study was 50 units, or 50 weeks. The window was not fixed but set as sliding to ensure that each selected window reflected the most recent historical trend of changes in slope. For the next point following the chosen window, the algorithm would perform anomaly detection using the most up-to-date window, i.e. filter of size 50.

Once the window size is selected by users, the next step is to calculate the appropriate percent quantile of differences within the defined window (Appendix 7.2.2). Due to the limited amount of data points, which in this case is for the 50 weeks, we assume the points to be normally distributed. Therefore, we used `norm`

[1]The World Health Organization declared a Public Health Emergency of International Concern regarding COVID-19 on 30 January 2020, and later declared a pandemic on 11 March 2020.

function from `scipy.stats` in python to calculate desired quantile of the first differences. Because our client is more interested in detecting sudden fall (decreasing slope) than in detecting sudden increase (increasing slope) as discussed in Section 1, the lower bound for the quantile is defaulted to be 2.5% while the upper bound is defaulted to be 99%. Once the quantile is calculated, the algorithm would identify the value at time `t+1` as an anomaly if it lies outside of the defined quantile ranges.

As a sanity check, the McDonald's Receipt Count data set collected using iPhone was used to illustrate how quantile worked. We first divided the data set into 1st half and 2nd half. The 1st half of the data set represents the data points within selected window and the 2nd half represents future data points to be collected at time `t+1` and forward. The 95% quantile from the 1st half of the data set (Figure 3(a)) was calculated and applied on the 2nd half of the data set (Figure 3(b)). In Figure 4 which is the first difference of 2nd half of the data with respect to time, any data point that lies outside of the two red quantile boundaries would be marked as an anomalous point, all detailed R code could be find in Appendix 7.2.2.



(a) 1st Half - Historical Dataset (window)　　(b) 2nd Half - Test Dataset (t+1 and beyond)

Figure 3: Sanity Check on McDonald's Data - 1/2

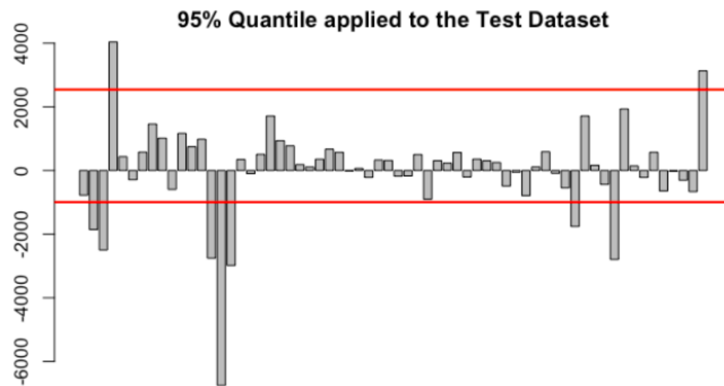


Figure 4: Sanity Check on McDonald's Data - 2/2

3.2 Robust Linear Regression Filter

3.2.1 Robust Linear Regression Model

The second method our team implemented for anomaly detection was the robust linear regression method [5], which fitted a robust linear model in a chosen time window and generated a prediction for the new point. For a given window say of size n, a robust linear model was fitted using window length from 1 till n as a predictor variable, and the response column values within that window, such as `receipt_count`, `item_total` etc. as the dependent variable. The fitted model was then used to generate one future prediction for the $n+1$ point in time. This predicted value, along with the standard deviation bounds defined in the following section, yielded ranges in which a future point would be accepted as non-anomalous.

3.2.2 Robust Estimate for Standard Deviation

In the robust linear regression filter, the standard deviation of the response column values (example: `receipt_count`, `item_total` etc.) in a window served in defining the boundaries when testing anomalous behaviour for a future observation. Since standard deviation could be influenced easily by outliers in a data set, a more stable, robust estimate was needed.

To achieve this goal, the first step was to take the first differences of points in a given window. For example, if at time t, the response column value was Y_t, then the difference D_t would be:

$$D_t = Y_t - Y_{t-1} \tag{1}$$

The second step was to calculate the variance of the first difference D_t. Since,

$$var[D_t] = var[Y_t] + var[Y_{t-1}] - 2cov[Y_t, Y_{t-1}] \tag{2}$$

Let τ^2 represent the variance for D_t, and σ^2 represent the variance of each Y_t, if Y_t and Y_{t-1} are independent, τ^2 became

$$\tau^2 = 2\sigma^2 \tag{3}$$

$$or, \sigma = \frac{\tau}{\sqrt{2}} \tag{4}$$

Thus, sample variance σ^2 was estimated from τ^2 i.e. the variance of first differences in a window. Further, to reduce the influence of outliers, median absolute deviation [2] (MAD) was used to estimate τ^2 for each window:

$$MAD = median(|D_t - median(D_t)|) \tag{5}$$

A robust estimate for τ can be obtained from median absolute deviation using a factor of 1.4826,

$$\tau = 1.4826 * MAD \tag{6}$$

Thus, the sample variance was estimated using:

$$\sigma^2 = \frac{(1.4826 * MAD)^2}{2} \tag{7}$$

or we can say,

$$\sigma = \frac{(1.4826 * MAD)}{\sqrt{2}} \tag{8}$$

Together, the robust linear regression model and robust standard deviation yielded ranges in which a future point would be accepted as non-anomalous. Because our client is more interested in detecting sudden falls than detecting sudden peaks this range was defined in terms of 5 standard deviations up and 4 standard deviations down the predicted mean. If an new observation had a value outside this range, it was then marked as an anomaly.

3.2.3 Sequential Filters

To facilitate the detection of slow trending anomalies, our algorithm applied 4 sequential filters (Figure 5). All filters had the same window size but covered different points in the past. For example, if the window size was `n`, for a point in the future `t+1`, 4 filters would perform anomaly detection of that point simultaneously. Filter 1 would cover points from `t-n` to `t`. Filter 2 would cover points from `t-n-1` to `t-1` and so on.

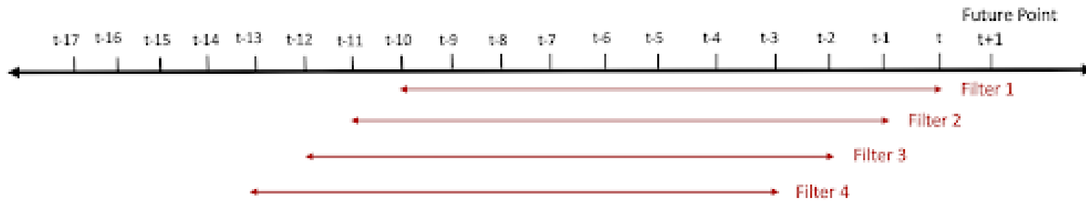
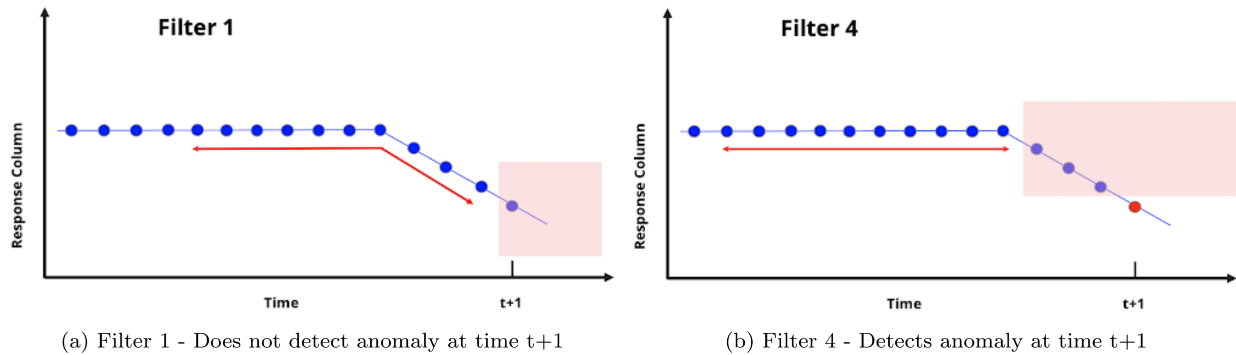


Figure 5: Illustration on Four Sequential Filters

In the following two illustrations in Figure 6, we can see why using sequential filters is beneficial for anomaly detection. The blue points in these two plots represent data points from the response column. Though we don't see a sharp dip in the data, a gradual downward trend in the data can be observed. The new point that we are checking for anomalous behaviour is present at time `t+1`. Filter 1 ends just before point `t+1` and is unable to detect value at point `t+1` as an anomaly, since it lies within its bounds. However, window for Filter 4 ends some points before `t+1`, and due to this gap, the new point ends up lying outside its bounds and we are able to detect an anomaly even in a slow moving trend.



(a) Filter 1 - Does not detect anomaly at time t+1

(b) Filter 4 - Detects anomaly at time t+1

Figure 6: Use of Sequential Filters

Thus, we see that in some cases Filter 1 might miss anomalies embedded in a slow moving trend since its bounds will reflect recent trends. This could impact the ability of Filter 1 in detecting anomalies. However, Filter 4 would be helpful in such scenarios, since there was a gap between it's window's end point and the future point `t+1` being tested for anomalous behaviour. Thus, ensuring some distance between filtering window and anomalous point would serve in the interest of picking up slow trends in the time series.

3.3 Trimmed Moving Weighted Average Filter

The third method our team implemented for anomaly detection was the trimmed moving weighted average method, which calculated the robust estimates of weighted average and standard deviation in a chosen time window. This method was later determined as less efficient comparing to the other two due to it's lack of ability in closely mimicking the real moving trend. We decided to only discuss meaningful outputs of the previous two methods in Section 4, but still included this method in the Python implementation for future research purposes (Appendix 7.3).

3.3.1 Robust Estimate for Weighted Average

The first step to ensure the weighted average in any given window was robust was to remove a reasonable amount of outliers from that window. Our algorithm can remove up to 20% of data in a set window. To define such outliers, the median of the data in the window was computed and used as a reference to calculate distances of all points from it. Depending on the maximum number of outliers user wanted to remove, the points with largest absolute distances from the median would be dropped. For example, if a user wanted to drop 2 out of 10 points (20%) in a window of size 10, the algorithm would drop the two points that were farthest away from the median, i.e. the point with the largest distance and the second largest distance. The remaining 8 points, in this case, were then used to calculate the robust weighted average (Appendix 7.3.2).

The second step was to assign weight to each remaining points in a given window and compute the weighted average for all rolling windows. Given that the time series data were not stationary, the algorithm assigned higher weights to more recent points in order to capture more relevant trend. For example, if the window spanned from `t-5` to `t`, with `t` being the current time, the weight assigned to point at `t-5` would be the lowest, and assigned to `t` would be the highest. Specifically, We utilized the reversed half normal distribution [4] to compute such weights for points in a given window (Appendix 7.3.3). The half normal pdf was:

$$w_i = \sqrt{\frac{2}{\pi\sigma^2}} exp(-\frac{i^2}{2\sigma^2}) \tag{9}$$

In equation (1), the variance σ^2 was represented in terms of window size. In particular, the standard deviation was taken to be a third of the window size. Since, 3 sigma covers 99.7 percent of data points in a half normal distribution, window size was assumed to be 3 sigma wide, and therefore sigma was taken to be a third of the window size. Let `n` represent the size of a window, equation (1) then becomes:

$$w_i = \sqrt{\frac{2}{\pi(\frac{n}{3})^2}} exp(-\frac{i^2}{2(\frac{n}{3})^2}) \tag{10}$$

Finally, to calculate the weighted average WA, the weight w_i for each point in the window would be mapped to the response column value y_i:

$$WA = \frac{\sum_{i=1}^{n} w_i y_i}{\sum_{i=1}^{n} w_i} \tag{11}$$

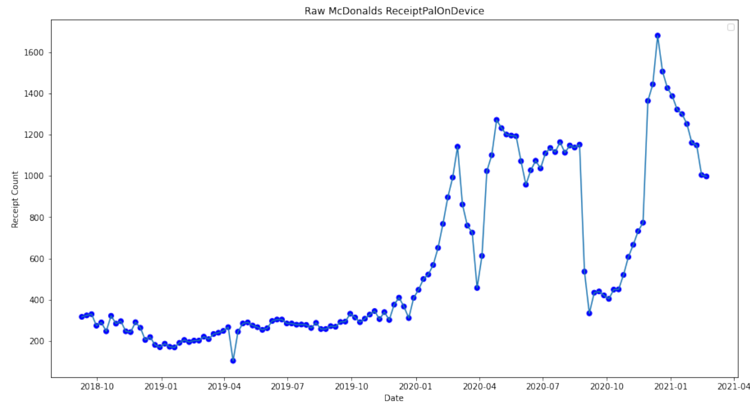
3.3.2 Robust Estimate for Standard Deviation

Robust estimate of standard deviation for the trimmed moving weighted average method follows the same steps as Section 3.2.2.

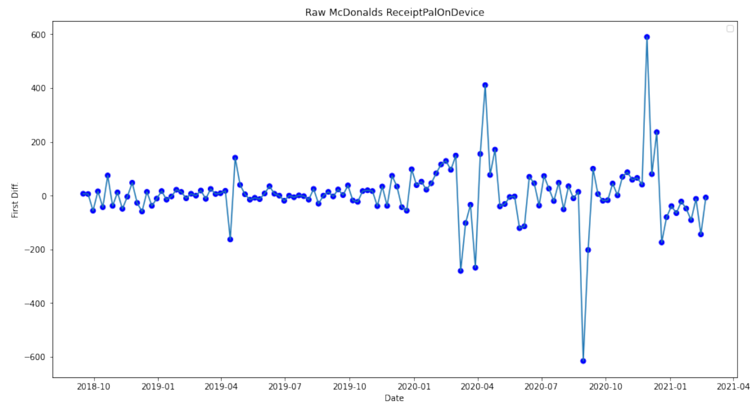
4 Results

4.1 First Differences Filter - Anomaly Detection Sample Results

The first difference filter proved to be quite flexible and capable of detecting seemingly overlooked anomalies. The nature of the first difference transformation (Eq.1) allows for the removal of long-term trends. This transformation resulted in previously identified "sharp jump" outliers and outliers within an overall positive or negative trend. Anomalies of the latter classification may have previously been missed due to their location within an overall trend. Figure **??** summarizes the results of the transformation.



(a) Raw McDonald's Data



(b) First Difference Transformation

Figure 7: McDonald's Data pre- and post- transformation

We see that the positive trend seen at the start of 2020 is smoother considerably in the difference filter while the large jumps in March 2020 and August 2020 are still maintained. In examining Figure **??**B we ultimately considered three sets of outliers we wished to classify. The anomalies manually identified are seen in Figure 8.

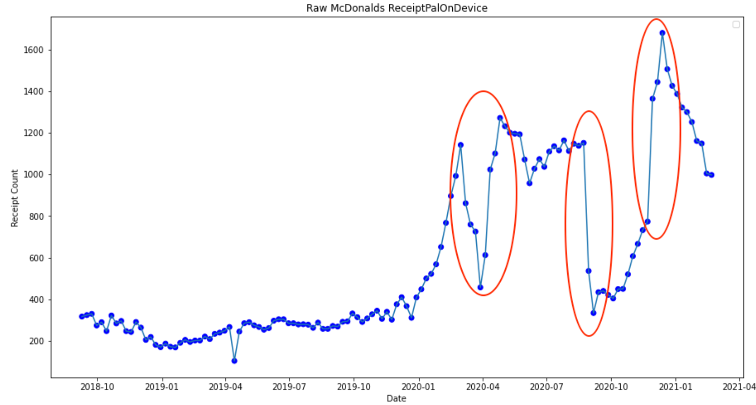


Figure 8: Illustration on Four Sequential Filters

With the outliers manually identified we analyzed this series with a First Difference filter using the following parameters:

- Window = 50
- $\alpha = 0.025$
- $\beta = 0.99$

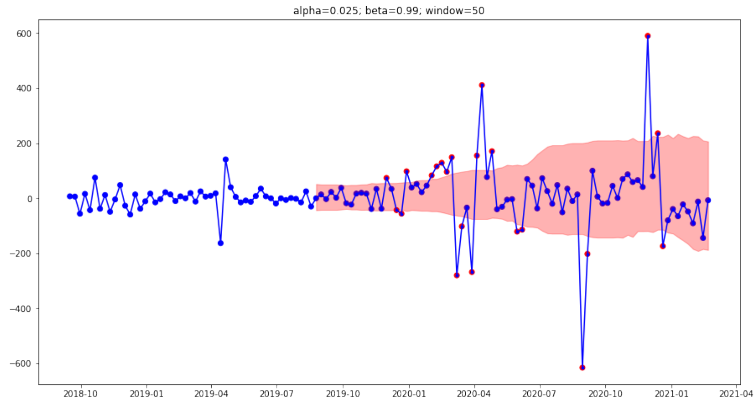


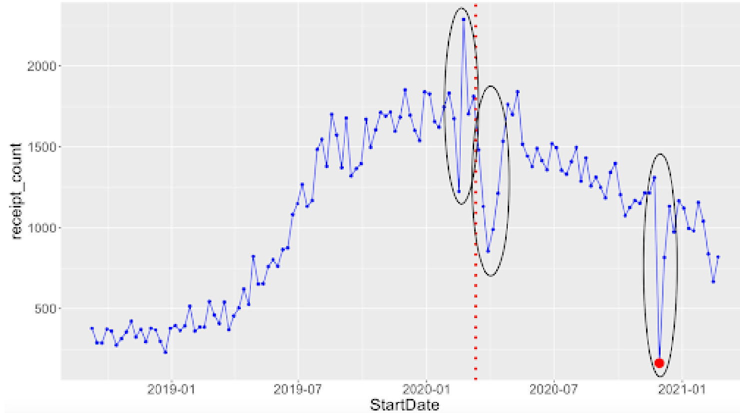
Figure 9: Fitted McDonald's Data

In Figure 9 we see the results of the First Difference filter. We note that the sharp differences identified were correctly classified as anomalies.

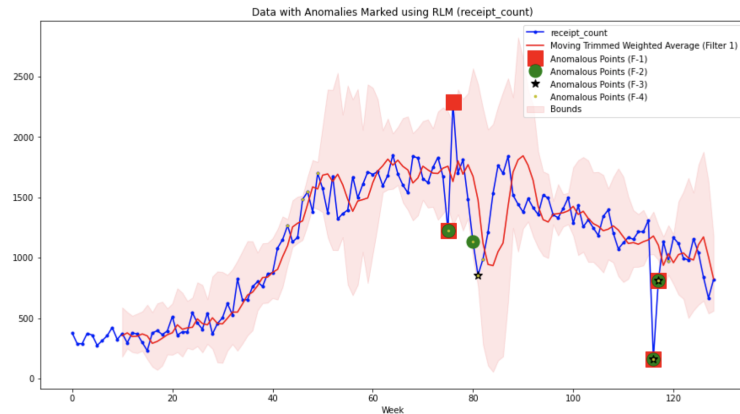
The strength of the first difference filter comes from its flexibility in using non-parametric method; with the implementation of mixed-normal distribution, user can fit nearly any data set with different distribution with relatively a good fit.

4.2 Robust Linear Model Filter - Anomaly Detection Sample Results

In the following section are some resultant visualizations using the Robust Linear Model Filter:



(a) Exploratory Data Analysis Plot for Merchant: Dollar General, Acquire Type: Receipt Pal on Device, Response Column: Receipt Count

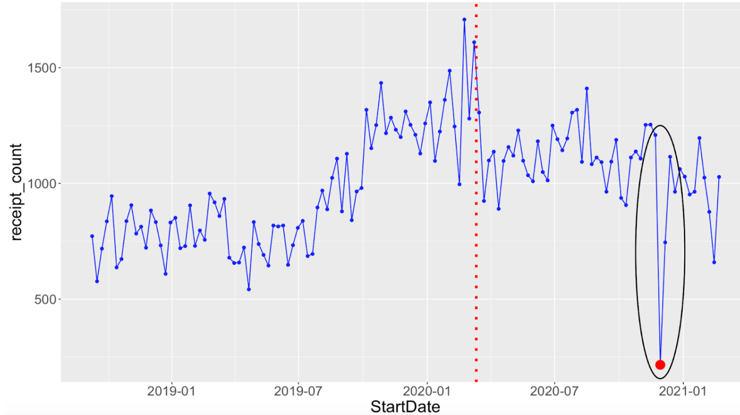


(b) Resultant Visualization with Anomalies detected using Robust Linear Model Filter

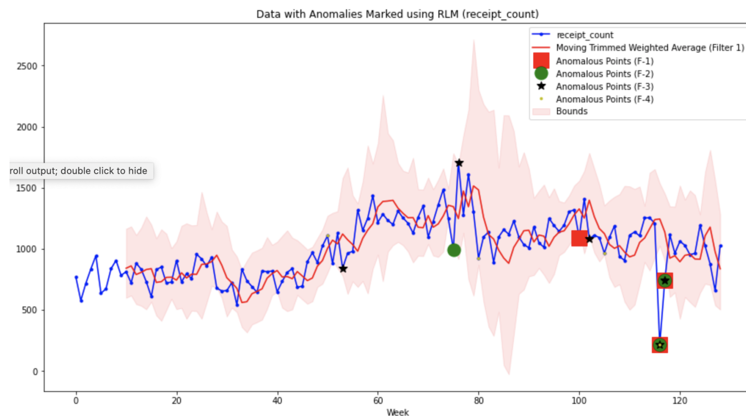
Figure 10: Anomaly Detection using Robust Linear Model Filter - Example 1

In the example above, we first have an exploratory plot for the `receipt_count` data series for merchant `Dollar General` and acquire type `Receipt Pal On Device`. The red dotted line marks the start of the pandemic COVID-19. The big red dot just before January 2021 represents the existing error flagged by client's team, which was shared with us in the original data (Section 2). The circles mark some anomalies we would want to catch. In addition to the existing error flagged, it will be helpful to detect the dip observed after the pandemic and large peaks.

Figure 9(b) is the resultant visualization after applying RLM filter to this data series with highlighted anomalies. The blue line represents the original data points and the red line is the RLM fitted line. The light-red shaded region represents the standard deviation bounds around the fitted line for the most recent sequential filter: Filter 1 (Section 3.2.3). In the legend we can see that, different shapes and colors have been used to mark anomalies being detected by 4 different filters. In particular, red square marks anomalies for Filter 1, green circle marks anomalies for Filter 2, black star marks anomalies for Filter 3 and yellow dot marks anomalies for Filter 4. We see that in this example, RLM successfully detected the original error by marking it as an anomaly by all 4 filters. The slower, more gradual dip however, around week 80, was not detected by Filter 1, but detected by Filter 3 and Filter 4. Thus, sequential filters were helpful in detecting slow drift right after pandemic, which we would not have detected with a single moving filter.



(a) Exploratory Data Analysis Plot for Merchant: Dollar General, Acquire Type: Sift, Response Column: Receipt Count



(b) Resultant Visualization with Anomalies detected using Robust Linear Model Filter

Figure 11: Anomaly Detection using Robust Linear Model Filter - Example 2

Similar to the previous example, we first have an exploratory plot for the receipt count data series for merchant `Dollar General` and acquire type `Sift` shown in Figure 10(a). Again, the big red dot just before January 2021 represents the existing error flagged by client's team and the circles mark some anomalies we would want to catch.

In Figure 10(b) we see the resultant visualization after applying RLM filter to this data series, and can see the anomalies highlighted. Again, We see that all 4 filters detected the original error. There are other errors which have been identified by 1 to 2 moving filters (out of 4). It is important to note that the number of filters detecting an error is not associated with the severity of that error point. In the case of slow drifts, errors will open be spotted only by 1 or 2 filters.

5 Discussion

5.1 Filter Comparisons

Features	RLM	1st Difference
Choice of window size	✓	
Detection of slow drifts	✓	
Robust to outliers	✓	✓
Data set is detrended	✓	✓
Non-parametric approach (non-normal fit)		✓

Figure 12: Filter Comparisons

The two filters developed have both unique and shared advantages, which are listed under Figure 12. For RLM, one of the advantages is that the choice of the window size is quite flexible. User's can easily input different window size parameters and check if the filter performs nicely. RLM is also able to detect slow drifts due the fact that it has 4 sequential filters (Section 3.2.3). Say if there's a gradual increase or decrease in receipt-count, which could be alarming, RLM will be able to detect such trends.

Both filters are robust in a sense that the impact of outliers was reduced. The robust linear model used in RLM filter helped predict the real trend without being affected by potential outliers. The first difference stabilizes the mean and variance, thus also reduces the effects of outliers (Section 3.1). Both filters are de-trended. RLM grants local predictions that reflect only previous trend within a certain time limit, and first difference removes seasonality and transforms a non-stationary time series to stationary.

Lastly, since first difference is non-parametric, users do not need to know a lot about the data itself before generating meaningful results. All users have to do is to input a window size that is greater than 50 for the filter to do a good job.

5.2 Limitations and Improvements

5.2.1 First Difference Filter

Figure 13 shows two limitations of First Difference Filter and improvements that could be made.

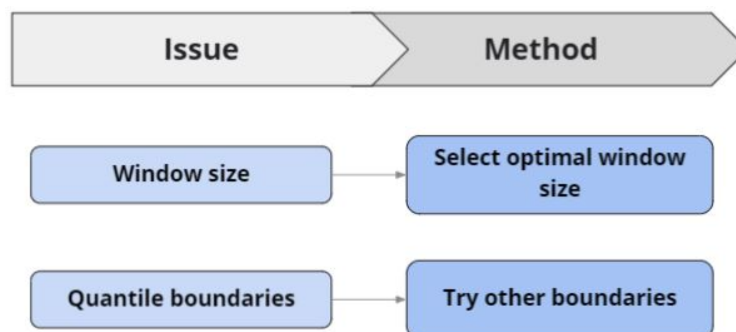


Figure 13: First Difference Filter Limitations and Improvements

Firstly, the error boundaries for First Difference Filter are based on quantile calculations (Section 3.1), thus are greatly impacted by the size of windows selected. For such filter to work nicely, the window sizes

cannot go under a certain limit (currently that limit is set as 50). A consequence of some of the data series having significantly less data points than others is that, if the number of observations is too low, the anomalies would not be meaningful. As a result, there is no point in running the First Difference Filter. Thus it would be ideal if we could define an optimal window size which includes just enough points for the filter to generate reliable results, and prevents eliminating too many data series due to lack of data.

Secondly, the current quantile boundaries are set to be 2.5% and 99%, which could be easily changed by users to adjust the number of errors spotted.

5.2.2 RLM Filter

Figure 14 shows three limitations of RLM Filter and improvements that could be made.

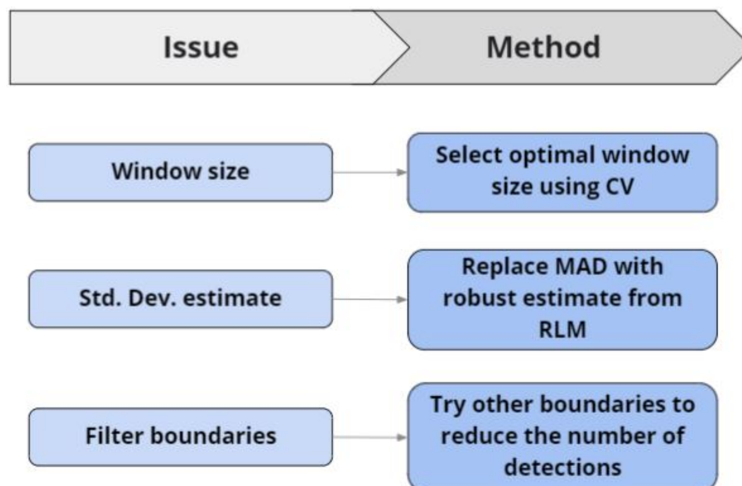


Figure 14: RLM Filter Limitations and Improvements

To begin with, currently, users can manually choose a window size for the filter. What could happen is that, users may pick a window size too small or too large, and the anomalies picked by the filters would be not so accurate. Thus, it is ideal to design a cross validation (CV) function which has the ability to determine the best window size by minimizing the CV absolute error loss. If we set an acceptable rolling window size range to be between 10 and 30, then users could generate a CV score for each window size, then pick the optimal window size to be the one that has the smallest CV score. Noticing that while the filters are expected to be ran on a weekly basis, the cross validation function does not need too much tuning in a short period of time – a monthly tuning would be sufficient since running cross validation usually takes more time than model fitting. To reduce computing complexity and increase model accuracy, it is also suggested that instead of using all historical data of a specific `Merchant` and `Acquire Type` for CV, using the most recent 50 observations would be enough.

Secondly, we are currently using Median Absolute Deviation of the first difference to calculate standard deviations of each rolling window. However, if there was a trend in the series, the expectation of the first differences $E[d_t]$ would not necessarily be 0, which was unlike the expectation of residuals after fitting a regression model. This could result in a slightly inflated sample variance (Section 3.3.2). The alternative way is to replace the current approach with computing residuals of RLMs [3]. This new approach could potentially be more relevant and time efficient.

Additionally, for now we are using 4 standard deviations up and 3 standard deviations down as our default error boundaries for RLM Filter (Section 3.1 and 3.3.2). In order to manipulate the total number of errors detected, users could try lower and higher boundaries instead.

5.3 Next Steps and Conclusion

In this section, we will discuss some future steps in order to better utilize the two filters delivered to NPD.

5.3.1 Additional Response Variables

Due to the high correlations between sets of variables mentioned in Section 7.1.3, as well as data complexity mentioned in Section 2.1, we are currently only using `receipt_count` as our response column.

For future steps, we could easily apply the existing filters to another 2 variables, which are `sum_total_paid` and `item_total`. For both filter, the only thing that the user will need to change is to replace the response column name with one out of these 2 variables.

On top of the existing variables, we could compute two new variables, which are the `average_sum_total_paid` per receipt and `average_item_total` per receipt. The motivation behind is that when observations are aggregated, some information about individual receipt is lost. All of the previous variables we studied so far are summations. To detect the variation of user behaviors in the amount spent per receipt, we will need to break-down the summation accordingly. Tracking the two average mentioned will help detect if an average customer is spending more or less than usual. A caveat to keep in mind is that, for these two newly created variables, the standard deviation needed to be scaled accordingly to generate meaningful boundaries.

5.3.2 Other Checks on Response Columns

A further next step is to compare response column values with another response column. For example, since it is possible for a panelist to upload more than one receipt, values in `receipt_count` response column should be equal or higher than `panelists`. Similarly, values in response column `item_total` should be equal or higher than column `sum_items_distinct` since `sum_items_distinct` captures the same information but only for distinct items. Further, it may be helpful to study the ratio of `sum_total_paid` by `sum_item_spend`. Column `sum_total_paid` refers to the final receipt amount, while column `sum_item_spend` is the sum of itemized prices.

Ratio of 'sum_total_paid' by 'sum_item_spend'	Proportion
Ratio<1	16%
Ratio=1	12%
Ratio=1 to 1.2	65%
Ratio=1.2 to 2	6%
Ratio>2	1%

Table 3: Ratio of `sum_total_paid` by `sum_item_spend`

We see that the ratio varies a lot. It can be less than 1 in cases where a final discount was applied before calculating the final receipt total. The ratio can be higher than 1 where taxes were added after adding the itemized amounts. We see for majority of cases ratio ranges from 1 to 1.2. This could reflect taxes of 0 to 20 percent, but they can be even higher in some locations. However, if the ratio is too high, like we see in around 1 percent cases, further checks can be incorporated to analyse the cause and check for any data recording errors.

These checks can be performed at an overall level or after splitting the data merchant and acquire type wise. If a merchant usually follows a known ratio range, any fluctuations can be marked as an anomaly for future investigations.

5.3.3 Removal of Identified Outliers

Although it requires one additional step in the anomaly detection process, we recommend to remove obvious outliers by setting cut-offs in a given rolling window before calculating the trend (mean) and anomaly boundaries (either standard deviation or quantile) for both filters. The main goal is to reduce the slightly inflated boundaries that present in both filters.

5.3.4 Boundaries Smoothing

A further next step is to smooth the boundaries for both methods. For the RLM Filter, at time `t`, we are finding the prediction at `t+1` by applying RLM to the `k` previous observations, and predicting the one ahead (Section 3.2.1). In the future, we could find the prediction for standard deviation at time `t+1` by smoothing the `k` previous robust standard deviation estimates and predicting the one ahead. This would help to remove some of the wiggles in the boundaries shown in Section 4.2.

For the First Differences Filter, we could select the last `k` points, and then smooth the estimated quantile using by applying RLM on these points. The advantage of applying smoothing technique on the quantile is that it permits using a smaller `k` (i.g.`k = 10` instead of `k = 50`) when choosing reasonable window sizes.

5.3.5 Conclusion

We believe that the current python generative anomaly detection implementations could produce relevant and insightful results for all data-series provided by our client. Section 5 mainly serves as a guide in cases that our client wants to investigate our current implementation and add more functionalities and improvements. We also hope that this report could serve as a reference for future researchers who are interested in implementing robust anomaly detection filters using both parametric and non-parametric approaches on time series data.

6 Reference

References

[1] Samaneh Aminikhanghahi and Diane J. Cook. "A Survey of Methods for Time Series Change Point Detection". In: *Knowledge Information System* 51.2 (2017). DOI: doi:https://doi.org/10.1007/s10115-016-0987-z.

[2] Christophe Leys and Olivier Klein. "Detecting outliers: Do not use standard deviation around the mean, use absolute deviation around the median". In: *Experimental Social Psychology* (2013).

[3] Peter J. Rousseeuw and Christophe Croux. "Alternatives to the Median Absolute Deviation". In: *American Statistical Association* 88.424 (1993). DOI: https://doi.org/10.2307/2291267.

[4] Michael Wallner. "A half-normal distribution scheme for generating functions". In: *European Journal of Combinatorics* 97 (2020). DOI: https://doi.org/10.1016/j.ejc.2020.103138.

[5] Chun Yu and Weixin Yao. "Robust linear regression: A review and comparison". In: *Communications in Statistics - Simulation and Computation* 46.8 (2017). DOI: https://doi.org/10.1080/03610918.2016.1202271.

7 Technical Appendices

7.1 EDA and Variable Selection

7.1.1 Data Preparation and Wrangling

As a first step of EDA, our team decided to merge the three data sets provided by the client. The resulting data set, or the merged data set consisted of 12 columns and 4907 rows and information about the sources and merchant as well as information about the anomalies detected for the 6 response variables,`receipt_count`, `sum_total_paid`, `sum_items_distinct`, `sum_item_spend`, `panelist` and `item_total`, were provided.

```
26 ```{r}
27 issue_data <- read.csv("cmu_issue_log_examples.psv", sep = "|", header = T, stringsAsFactors = FALSE)
28 source_data <- read.csv("cmu_data_collection_source.psv", sep = "|", header = T, stringsAsFactors = FALSE)
29 retailer_data <- read.csv("cmu_data_collection_retailer.psv", sep = "|", header = T, stringsAsFactors = FALSE)
30 retailer_data$StartDate <- as.Date(retailer_data$StartDate)
31 source_data$StartDate <- as.Date(source_data$StartDate)
32 issue_data$StartDate <- as.Date(issue_data$StartDate,format = "%m/%d/%Y")
33 issue_data$problem_detected <- rep(1, nrow(issue_data))
34 merged_data <- subset(retailer_data, MerchantName %in% unique(issue_data$MerchantName) &
35                         MerchantName != "Jerseymikes") %>%
36   left_join(x = ., y = issue_data,
37             by = c("MerchantID", "MerchantName","AcquireTypeID", "AcquireTypeDesc",
38                    "StartDate"))
39 merged_data$problem_detected[which(is.na(merged_data$problem_detected))] <- 0
40 jerseymikes_retailer <- subset(retailer_data, MerchantName == "Jerseymikes")
41 jerseymikes_retailer$problem_detected <- rep(0, nrow(jerseymikes_retailer))
42 jm_add_df <- subset(issue_data, MerchantName == "Jerseymikes")
43 jm_add_df$problem_detected <- NULL
44 jm_add_df$receipt_count <- rep(0, nrow(jm_add_df))
45 jm_add_df$sum_total_paid <- rep(0, nrow(jm_add_df))
46 jm_add_df$item_total <- rep(0, nrow(jm_add_df))
47 jm_add_df$sum_items_distinct <- rep(0, nrow(jm_add_df))
48 jm_add_df$sum_item_spend <- rep(0, nrow(jm_add_df))
49 jm_add_df$panelists <- rep(0, nrow(jm_add_df))
50 jm_add_df$problem_detected <- rep(1, nrow(jm_add_df))
51 jerseymikes_retailer <- rbind(jerseymikes_retailer, jm_add_df)
52 merged_data <- rbind(merged_data, jerseymikes_retailer)
53 dim(merged_data)
54 ```
```

7.1.2 Time Series Visualization

Once the data sets have been cleansed and merged to be used for the analysis, the next step was to visualize the time series of response variables. With the R codes shown below, our team has visualized the plots for the response variables with the error detected (reflected with the red dots) and the vertical line for when the COVID pandemic has occurred.

```
218 ```{r}
219 JimmyJohns_data <- subset(merged_data, MerchantName == "Jimmyjohns.com")
220
221 ggplot(data = JimmyJohns_data,
222        aes(x = StartDate, y = receipt_count, col = AcquireTypeDesc)) +
223   geom_line() +
224   geom_point(data=JimmyJohns_data[which(JimmyJohns_data$problem_detected==1), ],aes(x=StartDate, y=receipt_count), colour="red",
    size=2) +
225   geom_vline(xintercept = as.Date("2020-03-11"), col = "blue") +
226   labs(title = 'JimmyJohns - Receipt Count', subtitle = "Blue line: Start of Pandemic / Red dot: Data Collecting Issue") + NPD_theme
227
228 ggplot(data = JimmyJohns_data,
229        aes(x = StartDate, y = sum_total_paid, col = AcquireTypeDesc)) +
230   geom_line() +
231   geom_point(data=JimmyJohns_data[which(JimmyJohns_data$problem_detected==1), ],aes(x=StartDate, y=sum_total_paid), colour="red",
    size=2) +
232   geom_vline(xintercept = as.Date("2020-03-11"), col = "blue") +
233   labs(title = 'JimmyJohns - sum total paid', subtitle = "Blue Line: Start of COVID Pandemic, Red Dot: Data Collecting Issue")
234
235 ggplot(data = JimmyJohns_data,
236        aes(x = StartDate, y = item_total, col = AcquireTypeDesc)) +
237   geom_line() +
238   geom_point(data=JimmyJohns_data[which(JimmyJohns_data$problem_detected==1), ],aes(x=StartDate, y=item_total), colour="red", size=2)
    +
239   geom_vline(xintercept = as.Date("2020-03-11"), col = "blue") +
240   labs(title = 'JimmyJohns - item total', subtitle = "Blue Line: Start of COVID Pandemic, Red Dot: Data Collecting Issue")
241 ```
```

7.1.3 Variable Selection

Given limited amount of time for the project, performing anomaly detection for all 6 response variable was not feasible. Therefore, we have decided to see whether there were similarity in trends among the variables and checked both the trends of time series and correlations between selected pairs. The following R codes visualize the time series of the response variables and calculates the average correlations among the variables.

7.2 Sanity check of the First Difference Filter

7.2.1 First Difference Data Set

In order to perform the sanity check of the first difference filter, our team first created new data sets of the first differences. Four data sets were created for each merchant which was based on the data acquire sources as shown in the following codes.

```
272 ```{R}
273 sample_data <- subset(merged_data, MerchantName == "McDonald's")
274 sample_data1 <- subset(sample_data, AcquireTypeID == 3)
275 sample_data2 <- subset(sample_data, AcquireTypeID == 4)
276 sample_data3 <- subset(sample_data, AcquireTypeID == 7)
277 sample_data4 <- subset(sample_data, AcquireTypeID == 8)
278 mcdonald_diff <- list()
279 mcdonald_diff$receipt_count <- diff(sample_data1$receipt_count)
280 mcdonald_diff$sum_total_paid <- diff(sample_data1$sum_total_paid)
281 mcdonald_diff$item_total <- diff(sample_data1$item_total)
282 mcdonald_diff$sum_items_distinct <- diff(sample_data1$sum_items_distinct)
283 mcdonald_diff$sum_item_spend <- diff(sample_data1$sum_item_spend)
284 mcdonald_diff$panelists <- diff(sample_data1$panelists)
285 mcdonald_iphone <- as.data.frame(mcdonald_diff)
286 mcdonald_diff <- list()
287 mcdonald_diff$receipt_count <- diff(sample_data2$receipt_count)
288 mcdonald_diff$sum_total_paid <- diff(sample_data2$sum_total_paid)
289 mcdonald_diff$item_total <- diff(sample_data2$item_total)
290 mcdonald_diff$sum_items_distinct <- diff(sample_data2$sum_items_distinct)
291 mcdonald_diff$sum_item_spend <- diff(sample_data2$sum_item_spend)
292 mcdonald_diff$panelists <- diff(sample_data2$panelists)
293 mcdonald_android <- as.data.frame(mcdonald_diff)
294 mcdonald_diff <- list()
295 mcdonald_diff$receipt_count <- diff(sample_data3$receipt_count)
296 mcdonald_diff$sum_total_paid <- diff(sample_data3$sum_total_paid)
297 mcdonald_diff$item_total <- diff(sample_data3$item_total)
298 mcdonald_diff$sum_items_distinct <- diff(sample_data3$sum_items_distinct)
299 mcdonald_diff$sum_item_spend <- diff(sample_data3$sum_item_spend)
300 mcdonald_diff$panelists <- diff(sample_data3$panelists)
301 mcdonald_sift <- as.data.frame(mcdonald_diff)
302 mcdonald_diff <- list()
303 mcdonald_diff$receipt_count <- diff(sample_data4$receipt_count)
304 mcdonald_diff$sum_total_paid <- diff(sample_data4$sum_total_paid)
305 mcdonald_diff$item_total <- diff(sample_data4$item_total)
306 mcdonald_diff$sum_items_distinct <- diff(sample_data4$sum_items_distinct)
307 mcdonald_diff$sum_item_spend <- diff(sample_data4$sum_item_spend)
308 mcdonald_diff$panelists <- diff(sample_data4$panelists)
309 mcdonald_mobile <- as.data.frame(mcdonald_diff)
310 mcdonald_iphone$StartDate <- sample_data1$StartDate[-1]
311 mcdonald_android$StartDate <- sample_data2$StartDate[-1]
312 mcdonald_sift$StartDate <- sample_data3$StartDate[-1]
313 mcdonald_mobile$StartDate <- sample_data4$StartDate[-1]
314 ```
```

7.2.2 Histogram and Quantile Calculations

For the sanity check, McDonald's data sets have been selected to be the sample data sets. Data sets of each source have been divided into first and second half where the 95% qunatile has been calculated on the first-half of the data set and used in second-half of the data set for anomaly detection.

```
345 ```{r}
346 library(dplyr)
347 library(zoo)
348 #receipt count
349 mcdonald_iphone_1st <- mcdonald_iphone[1:64,]
350 mcdonald_iphone_2nd <- mcdonald_iphone[65:128,]
351
352 qts_rc <- quantile(mcdonald_iphone_1st$receipt_count, probs = c(0.025,0.975))
353
354 hist(mcdonald_iphone_2nd$receipt_count, nclass = 10)
355 abline(v=qts_rc[1], col = 'red')
356 abline(v=qts_rc[2], col = 'red')
357
358 barplot(mcdonald_iphone_2nd$receipt_count)
359 abline(h=qts_rc[1], col = 'red')
360 abline(h=qts_rc[2], col = 'red')
361
362 #sum total paid
363 qts_stp <- quantile(mcdonald_iphone_1st$sum_total_paid, probs = c(0.025,0.975))
364 barplot(mcdonald_iphone_2nd$sum_total_paid)
365 abline(h=qts_stp[1], col = 'red')
366 abline(h=qts_stp[2], col = 'red')
367
368 #items total
369 qts_it <- quantile(mcdonald_iphone_1st$item_total, probs = c(0.025,0.975))
370 barplot(mcdonald_iphone_2nd$item_total)
371 abline(h=qts_it[1], col = 'red')
372 abline(h=qts_it[2], col = 'red')
373
374 mcdonald_iphone_2nd$rc_anom <- (mcdonald_iphone_2nd %>% mutate(i = receipt_count > qts_rc[2] | item_total < qts_rc[1]))$i
375 mcdonald_iphone_2nd$stp_anom <- (mcdonald_iphone_2nd %>% mutate(i = sum_total_paid > qts_stp[2] | item_total < qts_stp[1]))$i
376 mcdonald_iphone_2nd$it_anom <- (mcdonald_iphone_2nd %>% mutate(i = item_total > qts_it[2] | item_total < qts_it[1]))$i
377 ```
```

```
380 # Mcdonald's - Android
381 ```{r}
382 #receipt count
383 mcdonald_android_1st <- mcdonald_android[1:64,]
384 mcdonald_android_2nd <- mcdonald_android[65:128,]
385
386 qts_rc <- quantile(mcdonald_android_1st$receipt_count, probs = c(0.025,0.975))
387 barplot(mcdonald_android_2nd$receipt_count)
388 abline(h=qts_rc[1], col = 'red')
389 abline(h=qts_rc[2], col = 'red')
390
391 #sum total paid
392 qts_stp <- quantile(mcdonald_android_1st$sum_total_paid, probs = c(0.025,0.975))
393 barplot(mcdonald_android_2nd$sum_total_paid)
394 abline(h=qts_stp[1], col = 'red')
395 abline(h=qts_stp[2], col = 'red')
396
397 #items total
398 qts_it <- quantile(mcdonald_android_1st$item_total, probs = c(0.025,0.975))
399 barplot(mcdonald_android_2nd$item_total)
400 abline(h=qts_it[1], col = 'red')
401 abline(h=qts_it[2], col = 'red')
402
403 mcdonald_android_2nd$rc_anom <- (mcdonald_android_2nd %>% mutate(i = receipt_count > qts_rc[2] | item_total < qts_rc[1]))$i
404 mcdonald_android_2nd$stp_anom <- (mcdonald_android_2nd %>% mutate(i = sum_total_paid > qts_stp[2] | item_total < qts_stp[1]))$i
405 mcdonald_android_2nd$it_anom <- (mcdonald_android_2nd %>% mutate(i = item_total > qts_it[2] | item_total < qts_it[1]))$i
406 ```
```

```
409 # Mcdonald's - Sift
410 ```{r}
411 #receipt count
412 mcdonald_sift_1st <- mcdonald_sift[1:64,]
413 mcdonald_sift_2nd <- mcdonald_sift[65:128,]
414
415 qts_rc <- quantile(mcdonald_sift_1st$receipt_count, probs = c(0.025,0.975))
416 barplot(mcdonald_sift_2nd$receipt_count)
417 abline(h=qts_rc[1], col = 'red')
418 abline(h=qts_rc[2], col = 'red')
419
420 #sum total paid
421 qts_stp <- quantile(mcdonald_sift_1st$sum_total_paid, probs = c(0.025,0.975))
422 barplot(mcdonald_sift_2nd$sum_total_paid)
423 abline(h=qts_stp[1], col = 'red')
424 abline(h=qts_stp[2], col = 'red')
425
426 #items total
427 qts_it <- quantile(mcdonald_sift_1st$item_total, probs = c(0.025,0.975))
428 barplot(mcdonald_sift_2nd$item_total)
429 abline(h=qts_it[1], col = 'red')
430 abline(h=qts_it[2], col = 'red')
431
432 mcdonald_sift_2nd$rc_anom <- (mcdonald_sift_2nd %>% mutate(i = receipt_count > qts_rc[2] | item_total < qts_rc[1]))$i
433 mcdonald_sift_2nd$stp_anom <- (mcdonald_sift_2nd %>% mutate(i = sum_total_paid > qts_stp[2] | item_total < qts_stp[1]))$i
434 mcdonald_sift_2nd$it_anom <- (mcdonald_sift_2nd %>% mutate(i = item_total > qts_it[2] | item_total < qts_it[1]))$i
435 ```
```

````
438 # Mcdonald's - recipepaidpal
439 ```{R}
440 #receipt count
441 mcdonald_mobile_1st <- mcdonald_mobile[1:64,]
442 mcdonald_mobile_2nd <- mcdonald_mobile[65:128,]
443
444 qts_rc <- quantile(mcdonald_mobile_1st$receipt_count, probs = c(0.025,0.975))
445 barplot(mcdonald_mobile_2nd$receipt_count)
446 abline(h=qts_rc[1], col = 'red')
447 abline(h=qts_rc[2], col = 'red')
448
449 #sum total paid
450 qts_stp <- quantile(mcdonald_mobile_1st$sum_total_paid, probs = c(0.025,0.975))
451 barplot(mcdonald_mobile_2nd$sum_total_paid)
452 abline(h=qts_stp[1], col = 'red')
453 abline(h=qts_stp[2], col = 'red')
454
455 #items total
456 qts_it <- quantile(mcdonald_mobile_1st$item_total, probs = c(0.025,0.975))
457 barplot(mcdonald_mobile_2nd$item_total)
458 abline(h=qts_it[1], col = 'red')
459 abline(h=qts_it[2], col = 'red')
460
461 mcdonald_mobile_2nd$rc_anom <- (mcdonald_mobile_2nd %>% mutate(i = receipt_count > qts_rc[2] | item_total < qts_rc[1]))$i
462 mcdonald_mobile_2nd$stp_anom <- (mcdonald_mobile_2nd %>% mutate(i = sum_total_paid > qts_stp[2] | item_total < qts_stp[1]))$i
463 mcdonald_mobile_2nd$it_anom <- (mcdonald_mobile_2nd %>% mutate(i = item_total > qts_it[2] | item_total < qts_it[1]))$i
464 ```
````

7.3 Trimmed Moving Weighted Average Filter

7.3.1 Input Parameters Sanity Check

This section of code was implemented to ensure that the users only input parameters that were acceptable. If not, an error message would pop up to guide users to re-choose a parameter in the correct form.

```
df_clean = df.copy(deep = True)

res_arr = df_clean[response_column]
window = self.window
outlier = self.max_outliers
x = list(range(window-outlier-1))

if df_clean.shape[1] < 2:
    raise IndexError\
        ('Expected at least 2 dimentional dataframe, %d dimensions given.' % (df_clean.shape[1]))
if (scaling==True) and (df_clean.shape[1] < 3):
    raise IndexError\
        ('Expected at least 3 dimentional dataframe, %d dimensions given.' % (df_clean.shape[1]))
if (scaling==True) and (scaling_column==None):
    raise IndexError\
        ('Expected at least 3 dimentional dataframe, %d dimensions given.' % (df_clean.shape[1]))
if (outlier > np.around(window*0.2)) or (outlier < 0):
    raise IndexError\
        ('Please input an acceptable outlier range. You can input a minimum of 0 outlier,
        and a maximum of %d outliers.' % (np.around(window*0.2)))
```

7.3.2 Deletion of Outliers in a Given Window

In order to remove outliers in rolling windows, total_list was used to store list of all possible rolling windows with size n. For each rolling window, the median was firstly computed. The absolute distance between each point and the median was then computed and ranked. Based on user's max_outlier (x) preference, the top x points which had the biggest distance would be dropped from the list.

```python
as_strided = np.lib.stride_tricks.as_strided
total_lst = as_strided(res_arr, (len(res_arr) - (window - 1), window), (res_arr.values.strides * 2)).tolist()
median_lst = list(res_arr.rolling(window).median())
del median_lst[:window-1]

absolute = []
for i in range(len(total_lst)):
    dists = []
    for j in range(window):
        dist = abs(total_lst[i][j]-median_lst[i])
        dists.append(dist)
    absolute.append(dists)

remove_idxs = []
for abss in absolute:
    sorted_abss = sorted(((v, i) for i, v in enumerate(abss)), reverse=True)

    remove_idx = []
    for i, (value, index) in enumerate(sorted_abss):
        if i <= outlier:
            remove_idx.append(index)
    remove_idxs.append(remove_idx)

for i in range(len(total_lst)):
    for j in range(window):
        if j in remove_idxs[i]:
            total_lst[i][j] = 0
    total_lst[i] = list(filter(lambda num:num != 0, total_lst[i]))
```

7.3.3 Half Normal PDF Weights Assignment

Now having the modified total_list after dropping outliers, the next step was to assign weight to each points remaining in the rolling windows. The weights were calculated and mapped to each point using the reversed half-normal distribution. Then the weighted averages were calculated accordingly.

```python
# https://arxiv.org/pdf/1610.00541.pdf
pdf_lst = [math.sqrt(2/math.pi)/((window-outlier)/3)*math.exp((-i**2)/(2*((window-outlier)/3)**2)) for i in x]
# reverse list to assign more value to near-current points
pdf_reverse = list(reversed(pdf_lst))

w_avg = []
for sub_lst in total_lst:
    value = sum([a*b for a,b in zip(sub_lst,pdf_reverse)])
    w_avg.append(value/sum(pdf_reverse))
```

7.3.4 Weighted Average Columns

Once all the weighted averages were computed, the values were then added to the actual data frame 4 times. In particular, a shifting effect was applied to each w_avg column to ensure that the weighted averages were matched up with the actual period of time in the data set, with w_avg_1 started at (initial time + window size), w_avg_1 started at (initial time + window size + 1) and so on.

```python
w_avg = pd.DataFrame({'w_avg_1': np.around(w_avg, 2)})

df_w_avg = pd.concat([df_clean,w_avg], axis=1)

df_w_avg['w_avg_1'] = df_w_avg.w_avg_1.shift(int(window))
df_w_avg['w_avg_2'] = df_w_avg.w_avg_1.shift(1)
df_w_avg['w_avg_3'] = df_w_avg.w_avg_2.shift(1)
df_w_avg['w_avg_4'] = df_w_avg.w_avg_3.shift(1)
```

7.3.5 Standard Deviation Columns

need to add more code for standard deviation computation. Similar to weighted average, once all the standard deviations were computed, the values were then added to the actual data frame 4 times. In

particular, a shifting effect was applied to each w_std column to ensure that the standard deviations were matched up with the actual period of time in the data set, with w_std_1 started at (initial time + window size), w_std_1 started at (initial time + window size + 1) and so on.

```
w_std = pd.DataFrame({'w_std_1': np.around(w_std, 2)})

df_w_avg_std = pd.concat([df_w_avg,w_std], axis=1)
df_w_avg_std['w_std_1'] = df_w_avg_std.w_std_1.shift(int(window))
df_w_avg_std['w_std_2'] = df_w_avg_std.w_std_1.shift(1)
df_w_avg_std['w_std_3'] = df_w_avg_std.w_std_2.shift(1)
df_w_avg_std['w_std_4'] = df_w_avg_std.w_std_3.shift(1)
```

7.3.6 Defining Bounds for Anomaly Detection

Now having both weighted average values and standard deviation values ready, the 3 sigma boundaries were calculated and appended to the same data frame for anomaly filtering purpose.

```
# filter 1
df_w_avg_std['pos_std_w_1'] = df_w_avg_std.w_avg_1 + df_w_avg_std.w_std_1
df_w_avg_std['neg_std_w_1'] = df_w_avg_std.w_avg_1 - df_w_avg_std.w_std_1
df_w_avg_std['pos_std_2_w_1'] = df_w_avg_std.w_avg_1 + 2*df_w_avg_std.w_std_1
df_w_avg_std['neg_std_2_w_1'] = df_w_avg_std.w_avg_1 - 2*df_w_avg_std.w_std_1
df_w_avg_std['pos_std_3_w_1'] = df_w_avg_std.w_avg_1 + 3*df_w_avg_std.w_std_1
df_w_avg_std['neg_std_3_w_1'] = df_w_avg_std.w_avg_1 - 3*df_w_avg_std.w_std_1
df_w_avg_std['pos_std_4_w_1'] = df_w_avg_std.w_avg_1 + 4*df_w_avg_std.w_std_1
df_w_avg_std['neg_std_4_w_1'] = df_w_avg_std.w_avg_1 - 4*df_w_avg_std.w_std_1
# filter 2
df_w_avg_std['pos_std_w_2'] = df_w_avg_std.w_avg_2 + df_w_avg_std.w_std_2
df_w_avg_std['neg_std_w_2'] = df_w_avg_std.w_avg_2 - df_w_avg_std.w_std_2
df_w_avg_std['pos_std_2_w_2'] = df_w_avg_std.w_avg_2 + 2*df_w_avg_std.w_std_2
df_w_avg_std['neg_std_2_w_2'] = df_w_avg_std.w_avg_2 - 2*df_w_avg_std.w_std_2
df_w_avg_std['pos_std_3_w_2'] = df_w_avg_std.w_avg_2 + 3*df_w_avg_std.w_std_2
df_w_avg_std['neg_std_3_w_2'] = df_w_avg_std.w_avg_2 - 3*df_w_avg_std.w_std_2
df_w_avg_std['pos_std_4_w_2'] = df_w_avg_std.w_avg_2 + 4*df_w_avg_std.w_std_2
df_w_avg_std['neg_std_4_w_2'] = df_w_avg_std.w_avg_2 - 4*df_w_avg_std.w_std_2
# filter 3
df_w_avg_std['pos_std_w_3'] = df_w_avg_std.w_avg_3 + df_w_avg_std.w_std_3
df_w_avg_std['neg_std_w_3'] = df_w_avg_std.w_avg_3 - df_w_avg_std.w_std_3
df_w_avg_std['pos_std_2_w_3'] = df_w_avg_std.w_avg_3 + 2*df_w_avg_std.w_std_3
df_w_avg_std['neg_std_2_w_3'] = df_w_avg_std.w_avg_3 - 2*df_w_avg_std.w_std_3
df_w_avg_std['pos_std_3_w_3'] = df_w_avg_std.w_avg_3 + 3*df_w_avg_std.w_std_3
df_w_avg_std['neg_std_3_w_3'] = df_w_avg_std.w_avg_3 - 3*df_w_avg_std.w_std_3
df_w_avg_std['pos_std_4_w_3'] = df_w_avg_std.w_avg_3 + 4*df_w_avg_std.w_std_3
df_w_avg_std['neg_std_4_w_3'] = df_w_avg_std.w_avg_3 - 4*df_w_avg_std.w_std_3
# filter 4
df_w_avg_std['pos_std_w_4'] = df_w_avg_std.w_avg_4 + df_w_avg_std.w_std_4
df_w_avg_std['neg_std_w_4'] = df_w_avg_std.w_avg_4 - df_w_avg_std.w_std_4
df_w_avg_std['pos_std_2_w_4'] = df_w_avg_std.w_avg_4 + 2*df_w_avg_std.w_std_4
df_w_avg_std['neg_std_2_w_4'] = df_w_avg_std.w_avg_4 - 2*df_w_avg_std.w_std_4
df_w_avg_std['pos_std_3_w_4'] = df_w_avg_std.w_avg_4 + 3*df_w_avg_std.w_std_4
df_w_avg_std['neg_std_3_w_4'] = df_w_avg_std.w_avg_4 - 3*df_w_avg_std.w_std_4
df_w_avg_std['pos_std_4_w_4'] = df_w_avg_std.w_avg_4 + 4*df_w_avg_std.w_std_4
df_w_avg_std['neg_std_4_w_4'] = df_w_avg_std.w_avg_4 - 4*df_w_avg_std.w_std_4
```

7.3.7 Detection of Outliers based on 4 filters

In details, a new point was categorized as an anomaly when it landed outside of 3 standard deviations in any of the 4 filters defined. The outliers were then recorded in binary forms, with 1 as an indicator of anomaly.

```
for i in range(len(df_w_avg_std)):
    if (df_w_avg_std.loc[i, response_column] > df_w_avg_std.loc[i, 'pos_std_4_w_1']) or
    (df_w_avg_std.loc[i, response_column] < df_w_avg_std.loc[i, 'neg_std_4_w_1']):
        df_w_avg_std.loc[i, 'outlier_1'] = 1
    else:
        df_w_avg_std.loc[i, 'outlier_1'] = 0

    if (df_w_avg_std.loc[i, response_column] > df_w_avg_std.loc[i, 'pos_std_4_w_2']) or
    (df_w_avg_std.loc[i, response_column] < df_w_avg_std.loc[i, 'neg_std_4_w_2']):
        df_w_avg_std.loc[i, 'outlier_2'] = 1
    else:
        df_w_avg_std.loc[i, 'outlier_2'] = 0

    if (df_w_avg_std.loc[i, response_column] > df_w_avg_std.loc[i, 'pos_std_4_w_3']) or
    (df_w_avg_std.loc[i, response_column] < df_w_avg_std.loc[i, 'neg_std_4_w_3']):
        df_w_avg_std.loc[i, 'outlier_3'] = 1
    else:
        df_w_avg_std.loc[i, 'outlier_3'] = 0

    if (df_w_avg_std.loc[i, response_column] > df_w_avg_std.loc[i, 'pos_std_4_w_4']) or
    (df_w_avg_std.loc[i, response_column] < df_w_avg_std.loc[i, 'neg_std_4_w_4']):
        df_w_avg_std.loc[i, 'outlier_4'] = 1
    else:
        df_w_avg_std.loc[i, 'outlier_4'] = 0


self.df_w_avg_std = df_w_avg_std

return df_w_avg_std
```

7.3.8 Anomaly Detection Method

This function served as an initializer for the trimmed moving weighted average filter to run and output anomalies in a data frame as the result. The data frame would then be called by the plot function to output anomaly detection visualizations for any given time series data.

```
df = pd.DataFrame(df)

if self.method == 'weighted_average':
    self.difference(df, response_column, anom_detect=True)
    df_out = self.moving_weighted_average(df, response_column, time_column, scaling, scaling_column)
    self.results = df_out

if self.method == 'median':
    self.moving_median(df, response_column, time_column, scaling, scaling_column)
    df_out = self.deviation_stats_median(response_column, time_column, scaling, scaling_column)
    self.results = df_out

# create subsets of outliers based on filters
if anom_detect:
    anom_points_1 = df_out.loc[df_out['outlier_1'] > 0]
    anom_points_2 = df_out.loc[df_out['outlier_2'] > 0]
    anom_points_3 = df_out.loc[df_out['outlier_3'] > 0]
    anom_points_4 = df_out.loc[df_out['outlier_4'] > 0]

    ## TODO: MAKE IT BETTER
    self.anom_points_1 = anom_points_1
    self.anom_points_2 = anom_points_2
    self.anom_points_3 = anom_points_3
    self.anom_points_4 = anom_points_4

    return anom_points_1, anom_points_2, anom_points_3, anom_points_4
```

7.3.9 Anomaly Detection Visualization

Once all outliers were identified, matplotlib was utilized to visualize anomalies detected using all filters. Four different shapes and colors were used on anomalies to distinguish which filter detected them.

```
df = self.results

anom_points_1 = self.anom_points_1
anom_points_2 = self.anom_points_2
anom_points_3 = self.anom_points_3
anom_points_4 = self.anom_points_4

fig, ax1 = plt.subplots(1, 1,figsize=(15, 8))
ax1.plot(list(df.index),df[response_column],'b.',label=str(response_column))
ax1.plot(list(df.index),df.w_avg_1,'r',label='Moving Trimmed Weighted Average (Filter 1)')
ax1.fill_between(df.index,df.pos_std_w_1,df.neg_std_w_1,color='red',alpha=0.3,label='1Sigma')
ax1.fill_between(df.index,df.pos_std_2_w_1,df.neg_std_2_w_1,color='red',alpha=0.2,label='2Sigma')
ax1.fill_between(df.index,df.pos_std_3_w_1,df.neg_std_3_w_1,color='red',alpha=0.1,label='3Sigma')

ax1.plot(list(anom_points_1.index),anom_points_1[response_column],'rs', markersize=18, label='Anomalous Points (F-1)')
ax1.plot(list(anom_points_2.index),anom_points_2[response_column],'go', markersize=15, label='Anomalous Points (F-2)')
ax1.plot(list(anom_points_3.index),anom_points_3[response_column],'k*', markersize=10, label='Anomalous Points (F-3)')
ax1.plot(list(anom_points_4.index),anom_points_4[response_column],'y.', markersize=5, label='Anomalous Points (F-4)')

ax1.set_xlabel('Week')
ax1.set_ylabel(data_label)
ax1.set_title('Data with Anomalies starred'+' ('+str(response_column)+')')
ax1.set_xlim(left=left,right=right)
ax1.set_ylim(bottom=bottom,top=top)
ax1.legend()
```

7.4 Python Interface

As per NPD's request we packaged our anomaly detection functionality in a python package **PyAnomalyDetect**. The package was written with the principles of single responsibility in that class, method and function performs one specific action and tracability. To reflect these principles we divided the anomaly detection process into two distinct steps; data preparation and analysis.

7.4.1 Data Preparation

What user does: 1. Subset by merchant and acquire type

What package does: 2. everything else

To ensure that our package was broadly applicable to the data analyzed by NPD. The data provided was defined primarily by the merchants, data acquisition methodology and a collection of six variables. To provide optimal functionality we designed a data structure, `AnomalyDf`, to capture the relevant information needed to prepare the data for analysis. The data structure can be defined by the following parameters within the __init__ function:

```python
def __init__(self, data: pd.DataFrame,
             response_column: str,
             time_column: str, scale_column: str = ""):
```

- data: A `pandas.DataFrame` with $p \geq 2$ columns and $n \geq 3$ rows. When provided it is assumed that the data will be reflective of a unique merchant-acquisition combination. We will refer to the data parameter as "the raw data" henceforth and will distinguish this from "processed data" produced in the constructor function.
- response_column: A python string indicating the name of the column to be considered as the response variable. As this parameter will be used to subset the provided data the constraint `response_column` $\in$ columns(`data`).
- time_column: A python string indicating the name of the column of `data` containing the time variable. As this parameter will be used to subset the provided data the constraint `time_column` $\in$ columns(`data`).
- scale_column: A python string indicating the name of the column used to scale the measures of spread. Defaults to `None`.

Provided the above parameters the `AnomalyDf` constructor will create an instance of the `AnomalyDf` class. In imagining and implementing the class we sought to balance processing work done by the user while standardizing the interface. Our initial exploration and calculations saw repeated tracking of the columns to be examined and therefore implemented the relevant columns as parameters. The `get_data(column)` method is used almost exclusively to access the data to reduce the repetition of the column names. The interface furthermore removes the burden from the user to format the data in a specific way.

- `df`: A `pandas.DataFrame` with n rows and $p \in [2,\ 3]$ columns. The columns present are the columns corresponding to the time column, response variable and the scaling column, if applicable. Note that this variable is public to allow for easy examination of the data but should be accessed with the `get_data(variable)` function.
- `sd`: A `numpy.array` with dimension `(n,)`. This field is initialized to an array of zeroes (produced by `self.sd = numpy.zeros(self.df.shape[0])`. We designated the calculation of the spread measures to the `update_sd` method both to adhere to the principle of single responsibility and to ensure that the user can change the standard deviation calculation without needing to recreate the instance.
- `diff_taken`: A boolean indicating if the first difference of the processed data (`df`) was computed. If `True` then the first differences of the response column may be found in the `diff` field.

The following methods are needed to modify, access and ultimately utilize the data in a standardized manner. The seemingly trivial methods (such as `nrows()` and `ncols`) were designed to reduce the user's interacions with the underlying data and discourage modification of the data.

```python
def nrows(self):
```

This method takes no arguments and returns the number of rows in the processed data (`self.df`).

```python
def ncols(self):
```

This method takes no arguments and returns the number of columns in the processed data (`self.df`). Note that this method will always return either two or three.

```python
def get_copy(self):
```

This method returns a deep copy of the processed data. Users are strongly encouraged to use this function when accessing or modifying the data to prevent aliasing issues that can be abundant in pandas.

```python
def get_data(self, column: str):
```

This method takes a string `column` argument and returns an $n \times 1$ `pandas.DataFrame` with data of the indicated `column` type. `column` must be one of `"response"`, `"time"` or `"scale"` indicating the response, timing and scaling columns respectively.

```python
def update_sd(self, sd_method: str, window: int):
```

This method is a non-constructive method that modifies the `sd` and `sd_initialized` functions to represent the measure of spread and a boolean indicating if it has been modified, respectively. The parameter `sd_method` is a string that can be one of `"Qn"`, `"IQR"` or `"mad"` indicating Rossousseeuw-Croux, interquartile range or median absolute deviation metric respectively.

```python
def take_difference(self):
```

This method computes the first difference and stores it in a field that can be accessed with the `get_difference(self)` method. Note that the difference will be of dimension $n \times 1$ with the first row (at index 0) is `NaN`. This relic was kept to maintain consistency and reduce confusion for later operations.

```python
def get_difference(self):
```

This method provides an easy way to access the previously calculated difference. The method returns an $n \times 1$ `pandas.DataFrame` with indices set as $[0, n-1]$ and a column named `first_diff`. The method will raise an `AttributeError` if differences have not been calculated.

```python
import numpy as np
import pandas as pd
from PyAnomDetect import AnomalyDf

MERCHANT_WANTED = "Mcdonald's"
METHOD_WANTED = "iPhone"
DATE_COLUMN = "StartDate"

## Load the data and user processing
raw_data = pd.read_csv(PATH_TO_DATA)
raw_data = raw_data[raw_data.MerchantName == MERCHANT_WANTED]
raw_data = raw_data[raw_data.AcquireTypeDesc == METHOD_WANTED]
raw_data[DATE_COLUMN] = pd.to_datetime(raw_data.DATE_COLUMN)

## create the class
data_obj = AnomalyDf(df, "panelists", "StartDate")

## get number of rows and columns
data_obj.nrows()
data_obj.ncols()

## calculate the standard deviations
data_obj.update_sd("Qn", 10)

## create a deep copy of the data
copied_data = data_obj.get_copy()

## get the response column
resp_column = data_obj.get_data("response")

## get the time column
time_column = data_obj.get_data("time")
```

7.4.2 Base Filter Class

Most of the analysis will be conducted within implementations of the `Filter` interface. The interface was created with the GRASP design framework as a guiding principle. In our `pyAnomDetect` package `Filter` is an interface written as a meta class so that the filters will be written as implementations of this abstract class. Our decision to use a meta class rather than an interface stems from our desire to re-use code. The initialization processes and preconditions are nearly identical for all methods. The primary methods and preconditions of the meta class can be seen below. Note that certain static methods and functionality was omitted from the sample to emphasize the core aspects of the interface. Note that the function definitions can be altered for subclasses in minor terms (such as changing parameter options).

```
1  from abc import ABCMeta, abstractmethod
2  import numpy as np
3  import pandas as pd
4  import AnomalyDf
5
6  class Filter(metaclass=ABCMeta):
7      def __init__(self, data_obj: AnomalyDf, window: int, max_outliers: int):
8          if window <= 2 or window >= data_obj.nrows():
9              raise ValueError("""Window must be greater than 2 and
10             less than the size of the data""")
11         if max_outliers == 0 or max_outliers >= data_obj.nrows():
12             raise ValueError("""Max_outliers must be a positive integer
13             less than the rows of the data""")
14         # rest of initialization code omitted...
15
16     @abstractmethod
17     def fit(self, scaling=False):
18         pass
19
20     @abstractmethod
21     def detect(self, new_data: pd.DataFrame, alpha: float = 0.05):
22         if alpha <= 0 or alpha >= 1:
23             raise ValueError("Alpha must be a float in (0, 1)")
24         pass
25
26     @abstractmethod
27     def deviation_stats(self):
28         pass
29
30     def plot(self, data_label):
31         pass
```

The implementation of a filter (as we have described them in this paper) are the filters we will use.

7.4.3 First Difference Class

```
1  def __init__(self, data_obj: pyAnomDetect.AnomalyDf, window: int, max\_outliers: int):
2      pass
```

- `data_obj`: An instance of the `AnomalyDf` class. This class prepared and standardizes the data for analysis. The object can be passed as a parameter with no initialization (save for the creation of the `AnomalyDf` object).
- `window`: Window size indicating how many prior points are to be considered in our anomoly detection.

```
1  def fit(self, slide = True,  alpha = 0.025, beta = 0.99):
```

The `fit` method is used to calculate the fitted values for each point, determine the upper and lower bounds and classify each point as an outlier or not. Error bounds are calculated from an inverse normal PDF of the first differences with the median used as an estimate of μ and 1.4826 $\times MAD$ for the first differences.

- `slide`: Determine if a sliding window should be used when calculating error bounds. Defaults to 'True'.
- `alpha`: Floating point value to determine the cutoff point below which a point will be considered an outlier. At α points less than the bottom α percent of all observations. Defaults to 0.025. Passed to an internal call of `deviation_stats`
- `beta`: Floating point value to determine the cutoff point above which a point will be considered an outlier. At β points greater than the top β percent of all observations. Defaults to 0.99. Passed to an internal call of `deviation_stats`.

```
def deviation_stats(self, slide = True,  alpha = 0.025, beta = 0.99):
```

The `deviation_stats` method adds the `sd` column to the `quantile_df` object. This method will be called (if not already called) within the `fit` method. Note that the parameters are repeated from the `fit` method.

```
def get_plot_variables(self)
```

The `get_plot_variables` method is an accessor method to extract the variables used to create a plot. Returns the time, time point, fitted value, original value, lower and upper bounds. For the first difference bounds the dimensions of the dataframe returned is $n \times 6$.

```
def plot(self):
```

The `plots` method is allows for the plotting of transformed data with the lower and upper bounds plotted.

```
## import statements
from pyAnomDetect.Filters.FirstDifference import FirstDifference as FD
from pyAnomDetect.AnomalyDf import AnomalyDf

raw_data = pd.read_csv("data/merged_data.csv")
raw_data["StartDate"] = pd.to_datetime(raw_data.StartDate)

data_obj = AnomalyDf(df, "receipt_count", "StartDate")
data_obj.take_difference()

## plot it
data_obj.plot()

fd_series = FD(data_obj, 50)
fd_series.fit()
fd_series.plot()
```

7.4.4 Robust Linear Regression Class

The Robust Linear Regression class is an implementation of the `Filter` meta class and performs the calculations described in our analysis.

```
def __init__(self, data_obj: pyAnomDetect.AnomalyDf, window: int):
    pass
```

- `data_obj`: An instance of the `AnomalyDf` class. This class prepared and standardizes the data for analysis. The object can be passed as a parameter with no initialization (save for the creation of the `AnomalyDf` object).
- `window`: Window size indicating how many prior points are to be considered in our anomoly detection.

```
def fit(self, c_down = 4, c_up = 5, use_differences):
```

The `fit` method is used to calculate the fitted values for each point, determine the upper and lower bounds and classify each point as an outlier or not. This method calls the method `deviation_stats` method if it has not been called at this time. Note that in the definition of the outliers the `c_up` and `c_down` parameters have the following roles:

$$[lower, upper] = [\widehat{x}_i - cDown \times SD, \widehat{x}_i + cUp \times SD]$$

- `c_down`: Constant to multiply the spread by when formulating the lower bound. Defaults to 4.
- `c_up`: Constant to multiply the spread by when formulating the upper bound. Defaults to 5.
- `use_differences`: use_differences: Indicate if the median absolute deviation (MAD) of the first differences should be used in the calculation of the error bounds. If false the stadnard deviation of the raw values are used to calculate the measure of spread. Defaults to True.

```
def deviation\_stats(self, c_down = 4, c_up = 5, use_differences):
```

The `deviation_stats` method adds the `sd` column to the `quantile_df` object. This method will be called (if not already called) within the `fit` method. Note that the parameters are repeated from the `fit` method.

```
def get_plot_variables(self, model_number: int = 1)
```

- `model_number`: A variable determining the number of shifts to use when calculating the upper and lower bounds

The `get_plot_variables` method is an accessor method to extract the variables used to create a plot. Returns the time, time point, fitted value, original value, lower and upper bounds. For the first difference bounds the dimensions of the dataframe returned is $n \times 6$. For the `FirstDifference` filter the `model_number` parameter defines how many shifts should be used when determining if a point is an anomaly.

```
def plot(self, model_number: int = 0):
```

The `plot` method is allows for the plotting of transformed data with the lower and upper bounds plotted.

- `model_number`: A variable determining the number of shifts to use when calculating the upper and lower bounds. If 0 all possible models are plotted.

```
## import statements
from pyAnomDetect.Filters.RobustLinearRegression import RobustLinearRegression as RLM
from pyAnomDetect.AnomalyDf import AnomalyDf

raw_data = pd.read_csv("data/merged_data.csv")
raw_data["StartDate"] = pd.to_datetime(raw_data.StartDate)

data_obj = AnomalyDf(df, "receipt_count", "StartDate")
data_obj.take_difference()

## plot it
data_obj.plot()

rlm_series = RLM(data_obj, 50)
rlm_series.fit()
rlm_series.plot()
```

7.4.5 Moving Weighted Average

The `MovingWeightedAverage` class is an implementation of the `Filter` class and allows for the implementation of methods previously described.

```
def __init__(self, data_obj: pyAnomDetect.AnomalyDf, window: int, max_outliers: int):
    pass
```

- `data_obj`: An instance of the `AnomalyDf` class. This class prepared and standardizes the data for analysis. The object can be passed as a parameter with no initialization (save for the creation of the `AnomalyDf` object).
- `window`: Window size indicating how many prior points are to be considered in our anomoly detection.
- `max_outliers`: Maximum number of outliers to search for. Defaults to 0. Note that a value of 0 indicates that no outliers will be removed. If set to a number greater than the number of rows in a data set this will be reset to the number of rows minus 1. Large max_outliers values can drastically slow down computation time.

```
def fit(self, c_down = 3, c_up = 4, use_differences):
```

The `fit` method is used to calculate the fitted values for each point, determine the upper and lower bounds and classify each point as an outlier or not. This method calls the method `deviation_stats` method if it has not been called at this time. Note that in the definition of the outliers the `c_up` and `c_down` parameters have the following roles:

$$[lower, upper] = [\widehat{x}_i - cDown \times SD, \widehat{x}_i + cUp \times SD]$$

- `c_down`: Constant to multiply the spread by when formulating the lower bound. Defaults to 3.
- `c_up`: Constant to multiply the spread by when formulating the upper bound. Defaults to 4.
- `use_differences`: use_differences: Indicate if the median absolute deviation (MAD) of the first differences should be used in the calculation of the error bounds. If false the stadnard deviation of the raw values are used to calculate the measure of spread. Defaults to True.

```
def deviation\_stats(self, c_down = 3, c_up = 4, use_differences):
```

The `deviation_stats` method adds the `sd` column to the `quantile_df` object. This method will be called (if not already called) within the `fit` method. Note that the parameters are repeated from the `fit` method.

```
def get_plot_variables(self, model_number: int = 1)
```

- `model_number`: A variable determining the number of shifts to use when calculating the upper and lower bounds

The `get_plot_variables` method is an accessor method to extract the variables used to create a plot. Returns the time, time point, fitted value, original value, lower and upper bounds. For the first difference bounds the dimensions of the dataframe returned is $n \times 6$. For the `FirstDifference` filter the `model_number` parameter defines how many shifts should be used when determining if a point is an anomaly.

```
def plot(self, model_number: int = 0):
```

The `plot` method is allows for the plotting of transformed data with the lower and upper bounds plotted.

- `model_number`: A variable determining the number of shifts to use when calculating the upper and lower bounds. If 0 all possible models are plotted.

```
## import statements
from pyAnomDetect.Filters.MovingWeightedAverage import MovingWeightedAverage as MWA
from pyAnomDetect.AnomalyDf import AnomalyDf

raw_data = pd.read_csv("data/merged_data.csv")
raw_data["StartDate"] = pd.to_datetime(raw_data.StartDate)

data_obj = AnomalyDf(df, "receipt_count", "StartDate")
data_obj.take_difference()

## plot it
data_obj.plot()

mwa_series = MWA(data_obj, 50)
mwa_series.fit()
mwa_series.plot()
```