M.Sc. Individual course
Master of Science in Engineering

DTU Compute
Department of Applied Mathematics and Computer Science

Measuring behavioral change within mobile app usage

Individual course

Abdulstar Kousa (s174360)

Supervisor: Sune Lehmann
Co-supervisor: Anna Sapienza

Kongens Lyngby 2022

DTU Compute
Department of Applied Mathematics and Computer Science
Technical University of Denmark

Matematiktorvet
Building 303B
2800 Kongens Lyngby, Denmark
Phone +45 4525 3031
compute@compute.dtu.dk
www.compute.dtu.dk

Contents

CHAPTER 1

Introduction

Time series analysis involves understanding various aspects about the inherent nature of the series. It's the first stage in developing a forecast, time series, model. Time series forecasting is highly significant in business since variables like mobile app usage, demand and sales, stock price, and so on are all essentially time series data.

This report focuses mainly on data manipulation, visualization, analysis, and forecasting of univariate time series, as well as how using changepoints in time series may improve forecasting.

Working with a users data that includes information about a sample of nearly 10K european users, we'll see how techniques such as outliers detection, missing values imputation, auto-correlation, rolling windows, stationarity tests, changepoints detection and power transformations, may be used to analyze and enhance the signal of user's daily screen time, number of checks, and median number of applications over time.

We will also see how models such as seasonal ARIMA and Facebook Prophet may be used to analyze and forecast user's number of checks, and how Prophet and Pruned Exact Linear Time (PELT) can detect the change points in a time series.

Finally we will conclude our work with an experiment on 300 different users to collect information and then compare the various methods and models learned throughout this report and check for reasonable correlations.

CHAPTER 2

Background

In this chapter we give a brief review of some basic concepts from time series. The treatment is by no means complete, and is meant mostly as gentle introduction and to set out the language used in this work.

2.1 What is a Time Series

Time series is a sequence of observations recorded at regular time intervals, for example, hourly weather measurements, daily counts of web site visits, or monthly sales totals. Time series can also be irregularly spaced and sporadic, for example, timestamped data in a computer system's event log.

2.2 Components of Time Series

Time-series data can be decomposed into: [Bro20]

- Level: The baseline value for the series if it were a straight line.
- Trend: Increasing or decreasing behavior of the series over time, often linear.
- Seasonality: Repeating patterns or cycles of behavior over time.
- Noise: variability in the observations that cannot be explained by the model.

All time series have a level and noise, the trend and seasonality are optional.[Bro20]

2.3 ACF and PACF

Autocorrelation Function (ACF) is simply the correlation of a series with its own lags. If a series is significantly autocorrelated, the previous values of the series may be useful in predicting the present value.[Bro20]

Autocorrelation Function (PACF) captures a series pure correlation and its lag, ignoring the correlation contributions from intermediate lags.[Bro20]

2.4 Stationarity in Time Series Data

A stationary series is one where the values of the series is independent of time. In other words, the mean, variance, and autocorrelation of the series are constant across time. A stationary time series is free of seasonal effects as well. [Bro20]

Test for Stationarity 'Unit Root Tests' can be used to test if a time series is non-stationary. There are multiple variations of these test, like: [Pra19]

- Augmented Dickey Fuller test (ADH Test): The most commonly used test to check for stationarity, where the null hypothesis is the time series possesses a unit root and is non-stationary. So, if the P-Value of ADH test is less than the significance level (e.g. 0.05), the null hypothesis is rejected and the series is stationary.
- Kwiatkowski Phillips Schmidt Shin test (KPSS test - (trend stationary)): KPSS is used to test for trend stationarity. The null hypothesis and the P-Value interpretation is just the opposite of ADH test.

Why to make a time series stationary? Some time series algorithms such as ARIMA make use of linear regression models which uses the series own lags as predictors. Moreover, linear regression models, work best when the predictors are not correlated and are independent of each other thus stationary.

How to make a series stationary? The most common approach is to difference the time series. That is, subtract the previous value from the current value. If the time series has defined seasonality, seasonal differencing may be considered. Seasonal differencing is similar to regular differencing, but, instead of subtracting consecutive terms, it subtract the value from previous season.

2.5 Power Transforms

In time series forecasting, data transforms aim to reduce noise and enhance signal. It can be very challenging to choose the best transform for a specific prediction task. There are several transforms available, and they all have various mathematical intuitions.[Bro20]

Box-Cox Transform The Box-Cox transform is a flexible data transformation technique that supports a number of related transforms, including square root and log transformations. Additionally, it may be set up to automatically assess a number of transforms and choose the best match.[Bro20] [**BoxCoxDoc**]

2.6 Rolling window and Downsampling

An additional crucial transformation for time series data is rolling window operations. Rolling windows divide the data into time windows, much like downsampling, then aggregate the data for each window using a function like mean(), median(), sum(), etc. Rolling windows overlap and ”roll” along at the same frequency as the data, so the transformed time series is at the same frequency as the original time series, unlike downsampling, where the time bins do not overlap and the output is at a lower frequency than the input.[Wal19]

2.7 Dealing with missing values

Generally replace missing values with the mean of the a time series, is not recommended especially if the series is not stationary. What could be done instead, depending on the nature of the series, is to try out multiple approaches like: [Pra19]

- Forward-fill, backward fill or Interpolation
- Rolling mean or Seasonal mean.

CHAPTER 3
Proposed Methods

This chapter presents the theory behind the main methods used in this work.

3.1 ARIMA and SARIMA

ARIMA, short for 'AutoRegressive Integrated Moving Average' is a class of models used to analyze and forecast time series data based on its own past values.[Bro20]

Shortly the main aspects of the model are: [Bro20]

- Autoregression (AR): A model that takes into account the dependency between an observation and a certain number of lagged observations.
- Integrated (I): Using differencing to make the time series stationary (i.e. subtracting an observation from observation at the previous time step) to make the time series stationary.
- Moving Average (MA): A model that uses the dependency between an observation and residual errors from a moving average model applied to lagged observations.

The parameters of the ARIMA model are defined as follows [Bro20]:

- The order of the AR term (p): The number of lag observations included in the model. We initially take the order of AR term to be equal to as many lags that crosses the significance limit in the PACF plot.
- The order of differencing term (d): The number of times that the raw observations are differenced. The right order of differencing is the minimum differencing required to get a near-stationary series.
- The order of the MA term (q): The size of the moving average window. We initially take the order of MA term to be equal to as many lags that crosses the significance limit in the ACF plot.

ARIMA models may be used to represent any "non-seasonal" time series that has patterns and isn't just random noise. However, the problem with plain ARIMA

model is it does not support seasonality. If a time series exhibits seasonal patterns, then there is a need to add seasonal terms creating SARIMA short for 'Seasonal ARIMA' which uses seasonal differencing. Similar to regular differencing, seasonal differencing subtracts the value from the prior season rather than consecutive periods. So, the model will be represented as SARIMA(p,d,q)(P,D,Q)[x], where, P, D and Q are seasonal AR, order of seasonal differencing and seasonal MA terms respectively and 'x' is the frequency of the time series. [Pra19]

3.2 Prophet

Facebook Prophet is an open-source algorithm for creating time-series models. It is especially good at modeling time series with a variety of seasonalities. Prophet generally manages outliers effectively and is robust to missing data and changes in the trend. Thus, doesn't face some of the drawbacks of other algorithms such as SARIMAX which have many stringent data requirements like stationarity and equally spaced values. [TL17]

Briefly the main aspects of the model are:[TL17]

- The Growth Function (and **change points**): The function simulates the data overall trend (with a basic linear and logistic functions). The new concept incorporated into Facebook Prophet is that the growth trend may be present throughout the data or may vary at certain "changepoints" - Changepoints are points in time where the direction of the data changes.
- The Seasonality Function: The function is simply a Fourier-Series as a function of time. It may be understood as the sum of several consecutive sines and cosines. A certain coefficient is multiplied by each sine and cosine term. This sum can approximate nearly any curve and in the case of Facebook Prophet, this total can roughly approximate the seasonality (cycle pattern) in the data.
- The Holiday Function: The function allows Facebook Prophet to adjust forecasting when a holiday or major event may change the forecast.

Facebook Prophet is made up of the sum of three time functions with an error term:

$$y(t) = g(t) + s(t) + h(t) + \epsilon_t$$

where g(t) stands for the growth, s(t) for seasonality, h(t) for holidays, ϵ_t for the error term. [Kri22]

CHAPTER 4

The Data Set

The datasets utilized in this work are introduced in this chapter.

4.1 The Users data

The users data includes information about a sample of nearly 10K european users and it is saved as a Dataframe with the following columns:

- user_id: unique identifier of the user.
- timestamp: a unix timestamp in seconds (there is a unique timestamp per unix hour).
- timezone: a timezone in seconds.
- screen_time: the amount of seconds a user used their phone in the hour bin.
- n_checks: the number of time the user pick up the phone in the hour bin.
- n_apps: the number of unique apps the user opened in the hour bin.

The data is sensitive. Thus, all the analysis was done on the ”sonydata” server.

4.2 User Data

A user data can be obtain by processing the users data (aggregation via PySpark [Spa]) and includes the following features:

- date — The date (yyyy-mm-dd format)
- sum(screen_time) — the daily totals of screen_time
- sum(n_checks) — the daily totals of n_checks
- median(n_apps) — the daily median of n_apps

Once again! The data is sensitive. Thus, all the analysis was done on the ”sonydata” server.

4.3 Data Preprocessing

To have a look into how we prepare the user data please refer to 6.2.

CHAPTER 5

User Data Analysis and Visualization

The data visualization and wrangling and components of time series analysis will be the major emphasis of this section. We'll look at how methods like rolling windows and resampling may be used to analyze a user's daily screen time, number of checks, and median number of applications over time when working with time series of user data. We'll covering the following subjects:

- Visualizing time series data
- Outlier Detection
- Filling missing data
- Seasonality
- Downsampling and Rolling Means
- AutoCorrelation Function (ACF)
- Stationary Tests
- Distributions plot
- Change Points Detection with Pruned Exact Linear Time (PELT)

5.1 Series Plots

First, let's create a line plot of the full time series of the user features.

Figure 5.1: Line plot of the full time series..

Already see some interesting patterns emerge:

- The number of checks is high during summer, presumably due to more free time during summer.
- There is an increase in screen time during the Christmas season (2015 December 25).
- It's also interesting to note that while the number of checks and screen time both decline from the middle of the 2017 first quarter to the end of the series, the number of applications stays around the same with more variance.

- Although it's hard to notice from the figure but the user series has some missing values so let's get into Data Preprocessing.

5.2 Outlier Detection

The plot below 5.2, shows the z-score upper and lower bounds for the user feature moving-averages. The shaded grey area shows the confidence interval around the moving average, the blue line shows the time series. The detected outliers are plotted with red dots. Few outliers was detected by the Rolling Z-Scores which is expected by looking at series structures. However, it's interesting that some of the outliers to the end of the series lies close to missing data points.

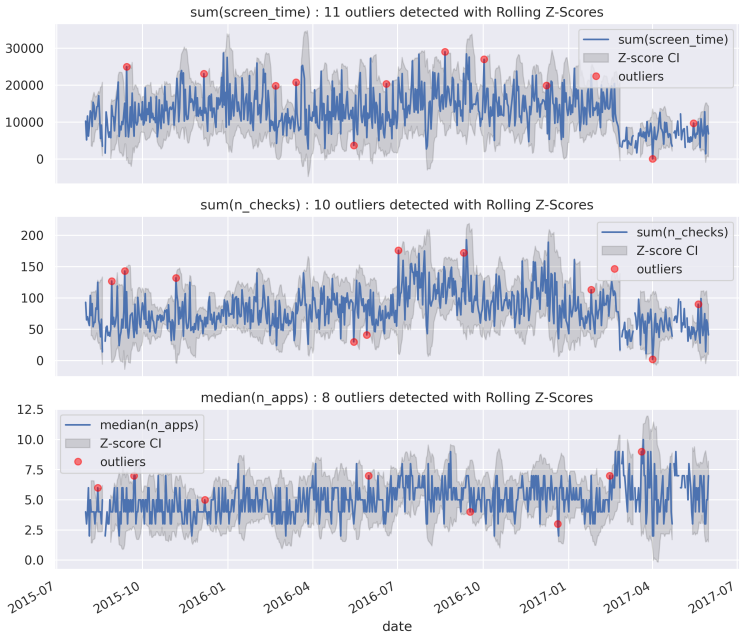

Figure 5.2: Outlier Detection via rolling Z-Scores for the user 3 series..

5.3 Filling missing data

Figure 5.3 shows a zoom-in into the user series. The filled missing data are shown in red. The series is plotted in green. It's seen from the plot that the chosen filling method here, 7 days rolling mean, seems to be working fine.

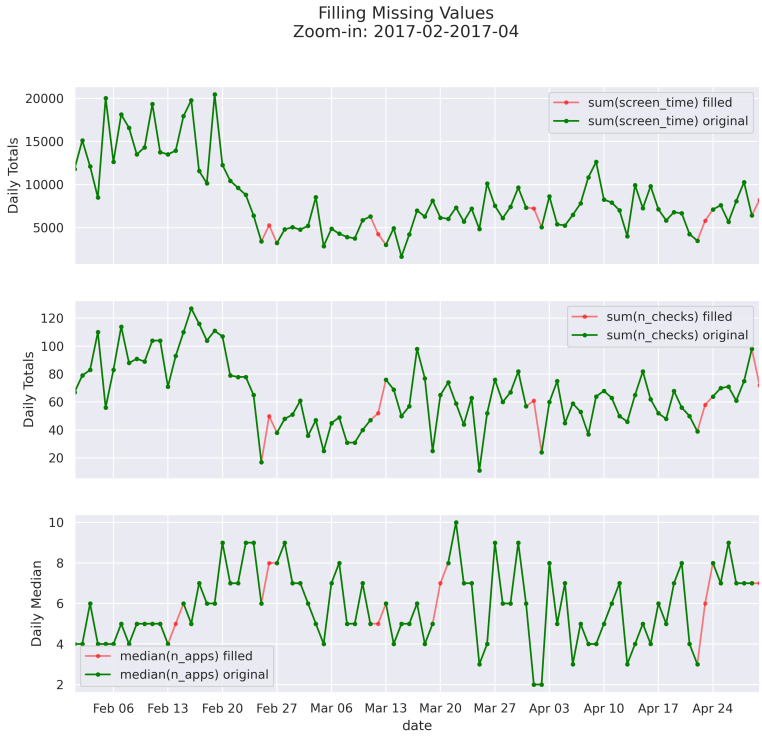

Figure 5.3: Filling missing data with 7 days rolling mean..

5.4 Seasonality

Next, let's explore the seasonality of the user data with box plots. first by grouping the user data by year quarters, to visualize seasonal seasonality.

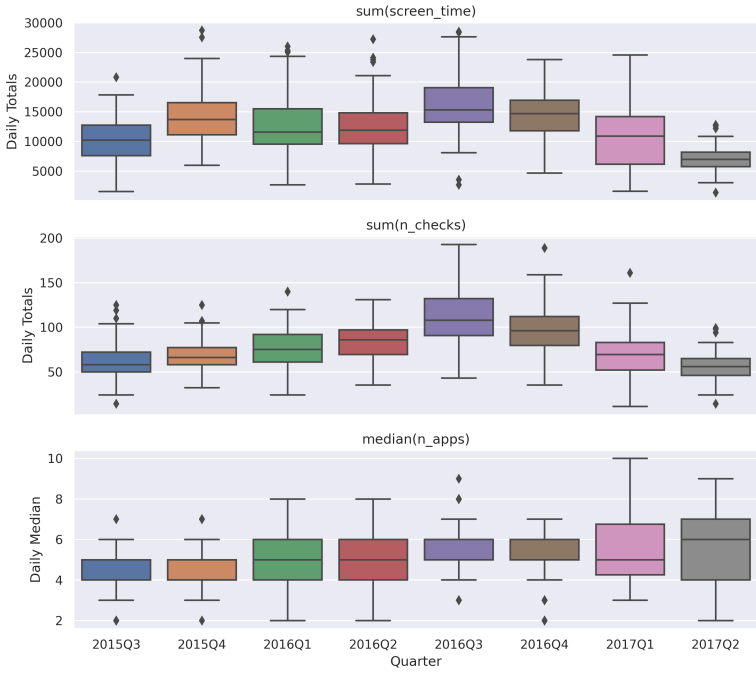

Figure 5.4: Grouping of user data by year quarters..

The box plots in figure 5.4 confirm the pattern found in earlier plots, towards the end of the series (2017 Q2) both the screen time and the number of checks decreased and the number of apps was high. Additionally, it's seen that 2016 Q3 shows the opposite.

Next, let's group the time series by month, to explore yearly seasonality.

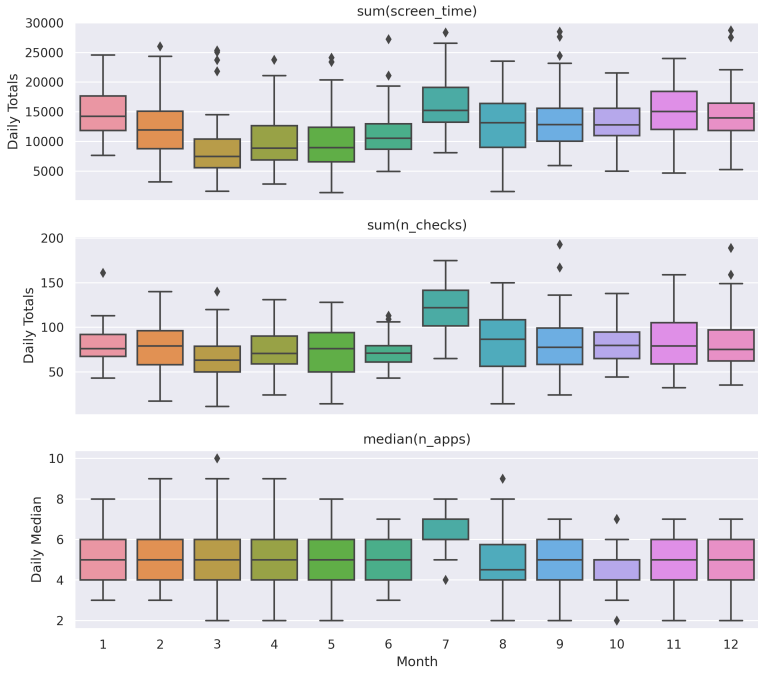

Figure 5.5: Grouping of user data by year month..

The box plots in figure 5.5 imply a potential correlation between the amount of screen time and the number of checks. The amount of time spent on screens and the number of checks are lowest in March, whereas all user' features rise in July, possibly because of summer vacation.

Finally, let's group the time series by day of the week, to explore weekly seasonality:

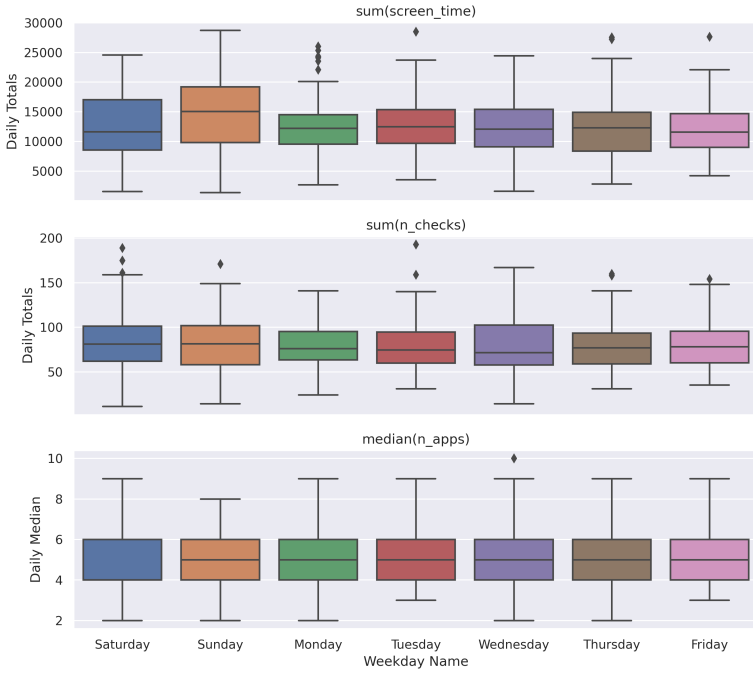

Figure 5.6: Grouping of user data by day of the week..

The box plots in figure 5.6 shows that both the screen time and the number of check are higher in weekends than on weekdays, possibly because of more free time! The high number of outliers on Mondays are presumably during holidays.

5.5 Downsampling and Rolling Means

Let's plot the 7-day and 30-day rolling mean together with the daily and weekly user time series.

Figure 5.7: 7-day and 30-day rolling mean together with the daily and weekly user time series..

From figure 5.7 it's seen that the weekly mean (orange) is smoother than the daily time series (blue) because higher frequency variability has been averaged out in the resampling. The differences between 7d rolling mean (green) and weekly mean resampled time series is also clear, it can be seen that data points in the 7d rolling mean time series have the same spacing as the daily series, but the curve is smoother because again higher frequency variability has been averaged out. In the 7d rolling mean time series, the peaks and troughs tend to align closely with the peaks and troughs of the daily time series. In contrast, the peaks and troughs in the weekly resampled time series are less closely aligned with the daily time series, since the resampled time series is at a coarser granularity. Looking at the 30d rolling mean (black), we can see that the long-term trend in the series is a bit flat.

5.6 AutoCorrelation Function (ACF)

Let's now plot the autocorrelation of a time series by lag (ACF).

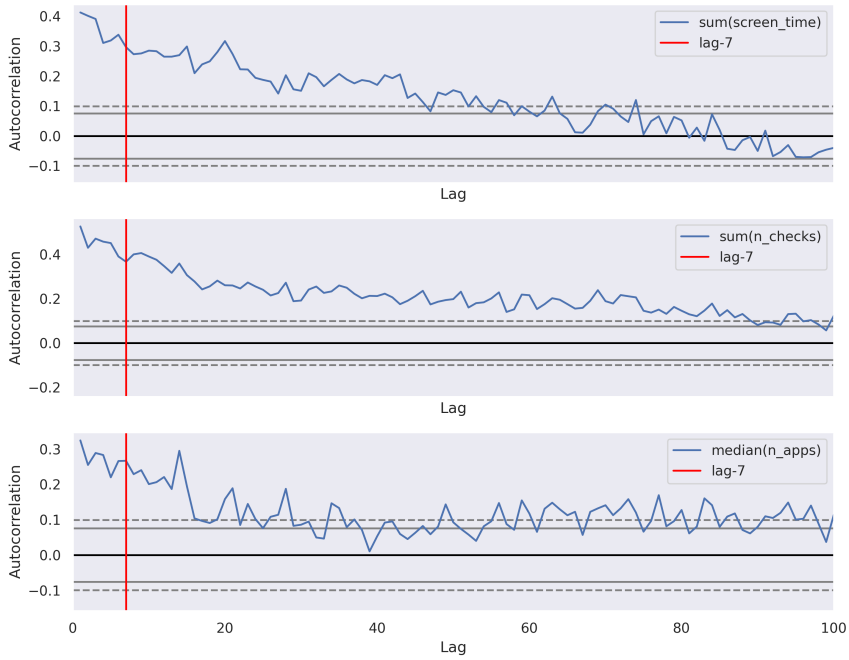

Figure 5.8: Autocorrelation of a time series by lag (ACF)..

The ACF plots above 5.8 show that the autocorrelation are not constant over time which indicate that the series are not stationary. The plots also show a significant lag for about 2 months for the screen time, 3 months for the number of checks and 1 month for the number of apps. That means, the previous values of the series (lags) may be helpful in predicting the current value (forecasting). The red vertical line is the lag 7, just to inspect the 7 lag seasonal pattern.

5.7 Stationary Tests

Let's run some stationarity 'Unit Root Tests' to test if the user time series is non-stationary.

- **Augmented Dickey Fuller test (ADH Test)**
  - screen time: non-stationary
  - number of checks: stationary
  - number of apps: stationary
- **Kwiatkowski Phillips Schmidt Shin test (KPSS tes)**
  - screen time: non-stationary
  - number of checks: non-stationary
  - number of apps: non-stationary

The results above suggest more support to exploring data transformation and some seasonal difference with one or two levels in order to make the series stationary prior to modeling SARIMA in the Forecasting chapter 6 later.

5.8 Histogram

The figure below 5.9 shows histogram plots of the user data. The histograms group values into bins, and the frequency (count) of observations in each bin can provide insight into the underlying distribution of the series.

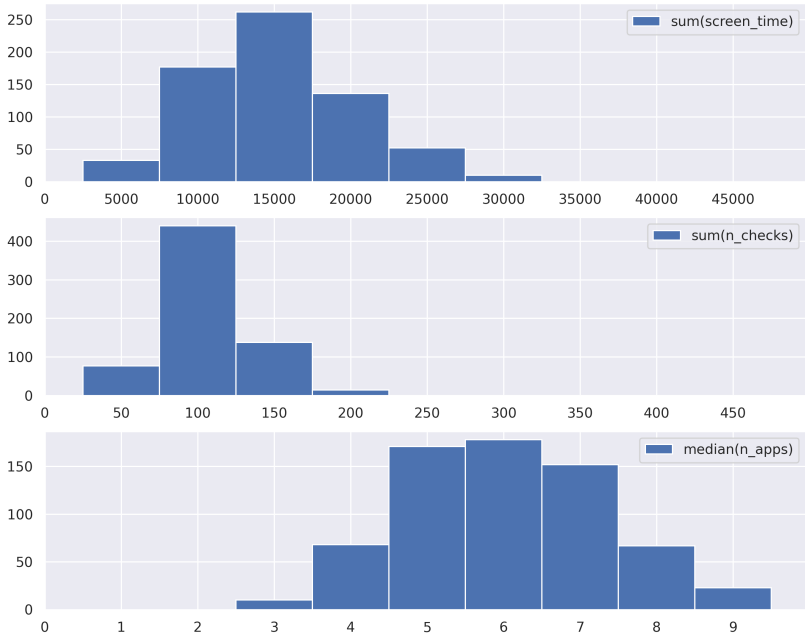

Figure 5.9: Histogram plots for the user series..

The histograms plot 5.9 shows that the distributions are not Gaussian, but is pretty close. A distribution is seen to have a long right tail and this may suggest an exponential distribution. This lends more support to exploring some power transforms of the data prior to modeling in the Forecasting chapter 6 later.

5.9 Pruned Exact Linear Time (PELT)

In time series, we try to uncover the underlying pattern, such as a regression line, modeling so that we may estimate the future. However, if there are change points, the regression line will not be a straight line. As a result, one intuitive technique is to construct segmented regression lines with kinked points representing change points. This technique is known as Pruned Exact Linear Time (PELT). [Kil12] [22a]

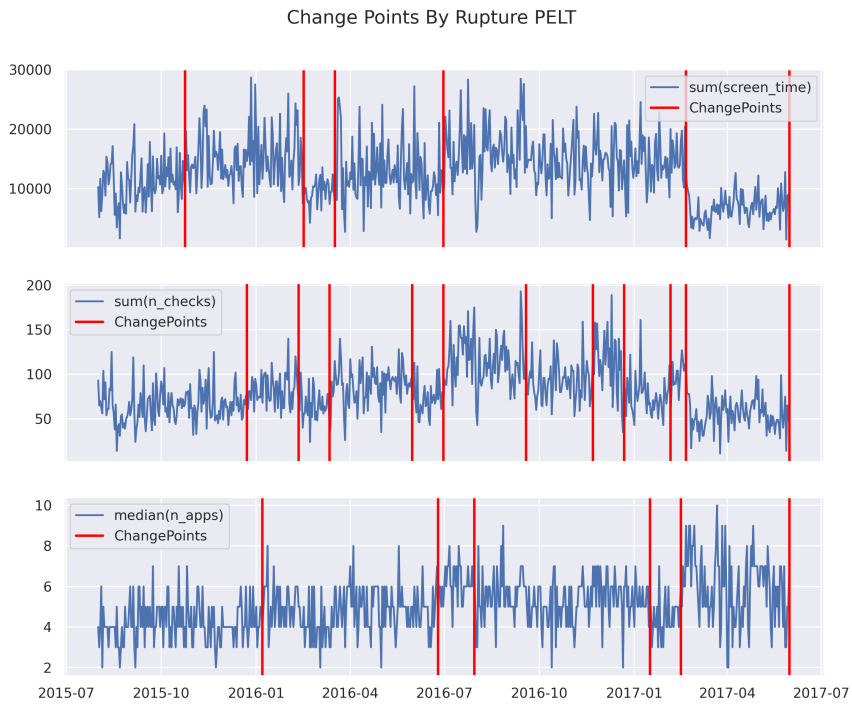

Figure 5.10: Changepoint detected with Rupture PELT with penalty = 3.

Figure 5.10 above shows the cangepoints (red lines) detected by Rupture PELT with penalty = 3 [Kil12] [22a]. It's seen that the cangepoints somewhat split the series into segments with minimum change in variance. This suggests that trying Prophet [TL17] which utilizes changepoints for forecasting may be a good idea.

CHAPTER 6

Univariate Time Series Forecasting

In this section, to keep the analysis concise and demonstrate the method used, we will continue with the same user and concentrate on one user feature, namely the number of checks (sum(n_checks)). We will analyze the methods used for forecasting and investigate, compare the results. We will also investigate Prophet changepoints detection.

6.1 The Problem

The main goal is to forecast the the final 30 values of the number of checks given the remaining data.

6.2 Data Preprocessing

6.2.1 Data Preparation

To prepare for upcoming forecasting tasks, the user data was selected and processed using a set of criteria and procedures which are highlighted below:

To guarantee that we only have entire months in the series, we first slice the user series such that it begins on the first day of a month and ends on the final day of a month.

Next, we confirm that the user has records going back more than 13 months.

Then we make sure that the last month in the data does not have more than 5 missing values. If the last month in the data contains more than five missing numbers, we drop it and repeat the previous step a maximum of five more times.

After that we confirm certain criteria, such as the user series having more than 13 months of data again, less than 10% of the data in the series are missing, the last month in the data has at least 25 records and no month has less than 15 recordings.

Next, we begin processing the user data. First, using Rolling Z-Scores (see appendix A.1 and section 5.2), we look for and eliminate outliers. Then we confirm certain criteria, such as less than 10% of the data in the series are missing, the last month in the data has at least 25 records and no month has less than 15 recordings.

If any of the above-mentioned tests fail, we consider another user; otherwise, we continue filling in missing values (see section 5.3). We utilize either the rolling mean or the seasonal mean to fill in missing data (we decide which one to use by RMSE which measures the difference between a data with no missing values and the corresponding filled data after dropping 10%).

6.2.2 Training and testing series

After Preprocessing the user data. The data is divided into the final 30 values for testing, with the remaining data being used for training. The test data will be used to evaluate the performance of the forecasts trained model.

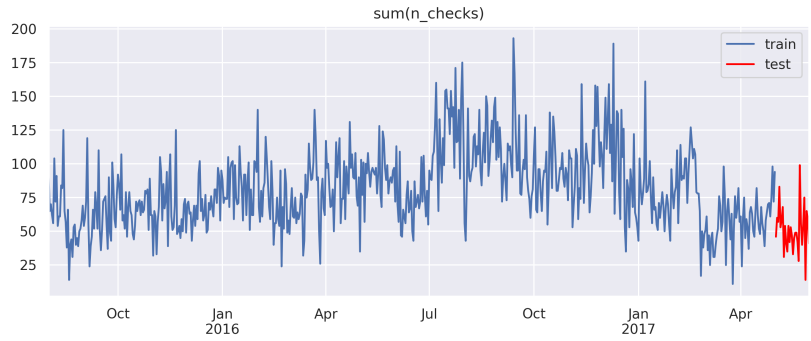

Figure 6.1: Training and testing series.

The figure 6.1 above shows the training and testing data by splitting the time series into 2 contiguous parts the test contains the final 30 values while the remaining data being used for training.

6.2.3 Data transforms

We transform the train using an automated Box-Cox transform [**BoxCoxDoc**] to minimize noise and improve the signals in the series. Running the automated Box-Cox function yields a square root transform with the statistically tuned value lambda = 0.46, which is quite near to lambda = 0.5. [Bro20]

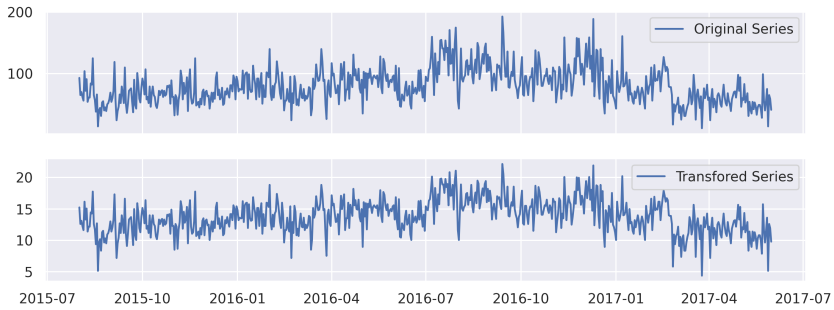

Figure 6.2: Line plot of the original and Box-Cox transformed sum(n_checks) series..

Figure 6.2 above shows that the trend was reduced but not eliminated. The line plot still shows an increasing variance from cycle to cycle.

6.3 Performance Measure

There are many popular error scores for time series forecasting. The performance measure used to evaluate forecasts models here is RMSE (root mean squared error) as its score gives worse performance to those models that make large wrong forecasts.

6.4 Persistence Model

We will be using the Persistence Model as a baseline to see how well the other models will perform on the problem. For additional information on the model and how it works, please refer to A.2.

6.4.1 Model Forecasting

Let's forecast for the next 30 days and compare it with the test data:

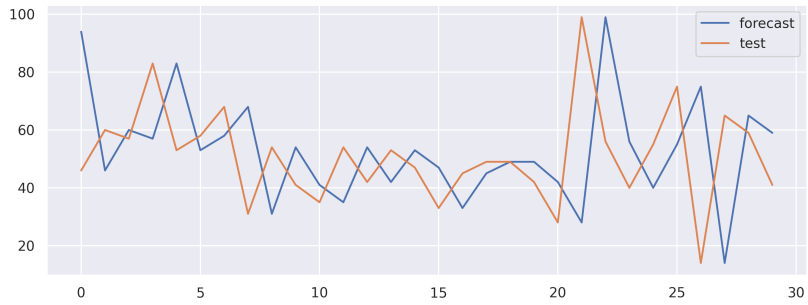

Figure 6.3: Persistence Model forecasts together with the actual test data..

The plot of the persistence model forecasts, Figure 6.3, highlights the method's shortcomings by demonstrating how the model is one step behind reality..

6.4.2 Model Evaluation

After converting the predictions by reversing the transform A.3, we evaluate the performance of the Persistence model on the test data with Root Mean Squared Error:

$$RMSE_{test} = 27.10 \text{ (around 27 checks)}.$$

6.5 SARIMA

As seen from the previous chapter, the series exhibits some seasonal patterns, thus we need to add seasonal terms to the ARIMA model resulting in SARIMA. For additional information on the model and how it works, please refer to 3.

6.5.1 Model Configuration

The model will be represented as SARIMA(p,d,q)(P,D,Q)[m], where: p, d and q are trend AR, order of trend differencing and trend MA terms respectively. P, D and Q are Seasonal AR, order of seasonal differencing and Seasonal MA terms respectively and 'm' the number of time steps for a single seasonal period.

We can specify one or more of these parameters, by utilizing our knowledge about ther problem. If not, we can try grid searching these parameters.

The pmdarima package [pmd] provides auto_arima() which uses a stepwise approach to search multiple combinations of the model parameters and chooses the best model that has the least AIC [Wik22].

Let's build the SARIMA model using pmdarima's auto_arima() [pmd]. To do that, we need to set 'seasonal=True'. As we assume the user to have a weekly seasonal cycle, we enforce D=1 for a given 'm = 7'. We let auto_arima() to work its magic to make the data stationary using Augmented Dickey Fuller test be setting 'test = adf' and 'd = None'. Then we let auto_arima() search for the best: $p \in \{1, 2, 3\}, q \in \{1, 2, 3\}, P \in \{0, 1, 2\}$ and $Q \in 1, 2$.

After fitting the model to the train data, the best model is find to be:

$$\text{SARIMA}(0,0,3)(1,1,2)[7].$$

Let's look at the model residual diagnostics plots to ensure there are no patterns.

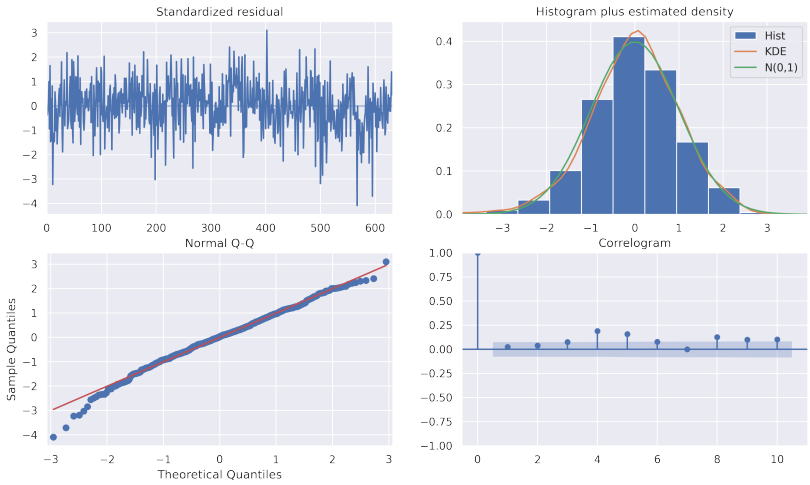

Figure 6.4: SARIMA model training residual diagnostics..

Looking at figure 6.4 we see:

- Top left: The residual errors seem seem fine with near zero mean and uniform variance.
- Top Right: The density plot suggest normal distribution with mean zero.
- Bottom left: Almost all the observations fall perfectly in line with the red line suggesting a normal distribution .
- Bottom Right: The ACF plot shows the residual errors are not autocorrelated. Meaning there is no pattern in the residual errors that is not explained in by model.

Overall, it seems to be a good fit. Let's forecast.

6.5.2 Model Forecasting

Let's forecast for the next 30 days and compare it with the test data:

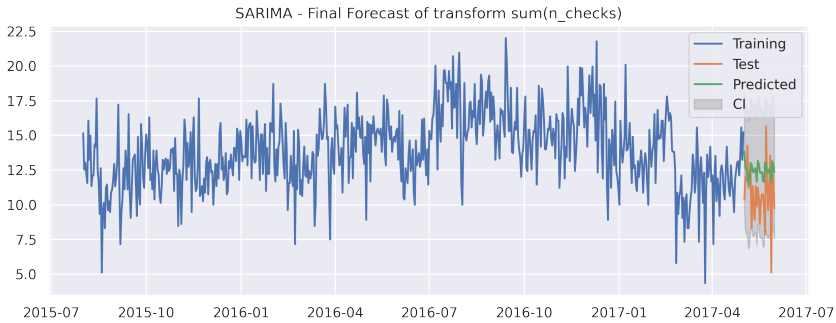

Figure 6.5: SARIMA model forecasts and 95% confidence interval together with the actual test and train data.

From the plot, 6.5, we see the observed test values lie within the 95% confidence interval except at the end of the series. It is a good sign indicating that the model is capturing the signal. However, most of the forecasts are above the observed test values which means there is a place for improvement.

6.5.3 Model Evaluation

After converting the predictions by reversing the transform A.3, we evaluate the performance of the SARIMA model on the test data with Root Mean Squared Error:

$$RMSE_{test} = 21.12 \text{ (around 21 checks).}$$

We see some improvements compared to the Persistence Model ($RMSE_{test} = 27.10$).

6.6 Facebook Prophet

The Facebook Prophet model is robust to trends changing, and it succeeds at modeling time series with various seasonalities. As was seen in the preceding chapter, the series displays certain seasonal trends and fluctuations in the trend, thus using Prophet might be tempting. In this section, we'll look into Prophet forecasting and the trend changepoint, which is a highly fascinating feature of Prophet (and time series analysis). For additional information on the model and how it works, please refer to 3.

6.6.1 Model Configuration

Prophet Cross-validation may be used to tune the model's hyperparameters, including changepoint_proir_scale seasonality_mode. It chooses cutoff points in the past and fits the model for each of them using just the data up to the relevant cutoff point. We specify the forecast horizon (horizon), and the spacing between cutoff dates (period). The first training period is by default set at three times the horizon. In this case, parameters are evaluated using an average RMSE over all cutoffs. We sit **horizon = 30** and **period = 30**, which means that the initial training period is almost 3 months then increasing by 30 days [22b].

There are many parameters to tune but we will focus on the following:

- *changepoint_range* = 1: The default setting for the range of data points considered when identifying changepoints is the first 80% of data in the time series to ensure that the model doesn't overfit. However, to incorporate 100% of the training data We fix this by setting changepoint_range = 1 when instantiating the model. [22b]
- *changepoint_proir_scale* $\in \{0.01, 0.05, 0.1, 0.5, 0.75\}$: According to Prophet documentation "This is probably the most impactful parameter. It determines the flexibility of the trend, and in particular how much the trend changes at the trend changepoints. The default of 0.05, but this could be tuned". [22b]
- *seasonality_mode* $\in \{multiplicative, additive\}$: Default is 'additive', but many time series will have multiplicative seasonality. [22b]

After cross-validating over the hyperparameters the model best parameters are find to be:

$$changepoint\_proir\_scale = 0.05 \text{ and } seasonality\_mode = additive$$

which are also the default parameters of the Prophet, interesting!

Let's move to forecasting.

6.6.2 Changepoints and Model Forecasting

By fitting the model with the best parameters to the train data, Prophet creates the final model. It has also (behind the scenes) created some potential changepoints. by default, Prophet adds 25 changepoints into *changepoint_range* of data, in our case the whole training data.

Let's forecast for the next 30 days and compare it with the test data:

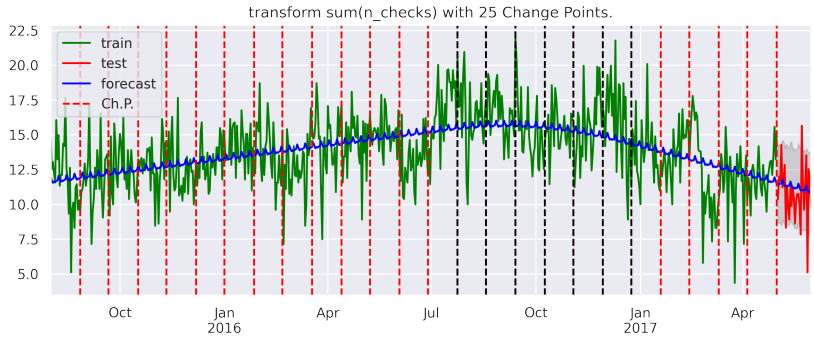

Figure 6.6: Prophet model forecasts and 95% confidence interval together with the actual test and train data in addition to Prophet changepoints (dashed red/black lines).

From the plot, 6.6, we see that most of the observed test values lie within the 95% confidence interval except for few places. Taking a look at the possible changepoints (drawn in dashed red/black lines), we can see they fit some of the highs and lows. The black dashed lines are significant change points according to the fitted model under the level 0.01 which Prophet can utilize in forecasting.

Prophet will also let us take a look at the magnitudes of these possible changepoints:

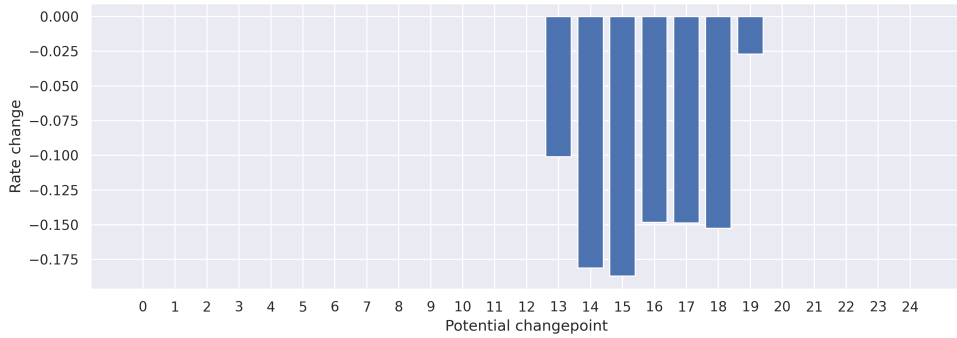

Figure 6.7: Prophel Model changepoint Magnitudes..

We can see from the above figure 6.7, that there are a lot of these changes points (found between 0 - 12 and 20 - 24) that are super minimal in magnitude and are most likely to be ignored by prophet during forecasting.
Now, let's take a look at the seasonality and trend components of our datamodelforecast.

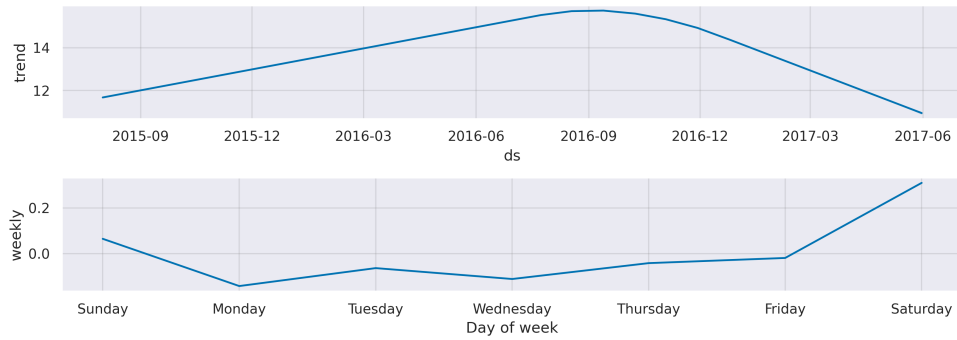

Figure 6.8: Prophet seasonality and trend components..

From the trend and seasonality in figure 6.8, we can see that the trend is a playing a large part in the underlying time series and seasonality comes into play more toward the weekend (Saturday and Sunday) this is also what we so during the Exploratory analysis chapter 5.

6.6.3 Model Evaluation

After converting the predictions by reversing the transform A.3, we evaluate the performance of the Prophet model on the test data with Root Mean Squared Error:

$$RMSE_{test} = 16.52 \text{ (around 17 checks)}.$$

We see a clear improvements compared to both the Persistence Model ($RMSE_{test} = 27.10$) and the SARIMA Model ($RMSE_{test} = 21.12$).

In the upcoming chapter, we will see if what we found here can apply for other users, we want to see if Prophet will outperforms SARIMA and the Persistence Model on the Users Data.

CHAPTER 7

Design of the Experiment

In this chapter, we will explain how an experiment was planned and carried out to collect relevant information from 300 selected users and see if what we found in chapter 6 can apply for other users.

First, the preparation of the user data was done as described in section 6.2. Then, for each of the three user features, we repeat the forecasting tasks completed in Chapter 6, gathering data for each user feature and forecasting model. This is done until 300 approved users have been analyzed. Experiment code can by found at appendix B.

It's worth noting that the experiment took around 32 hours to complete.

Next, let's look into the Experimental Results.

CHAPTER 8

Experimental Results

In this section, although we ran the experiment over all user features, we will concentrate on one user feature, namely the number of checks (sum(n_checks)), to keep the analysis concise and demonstrate the results.

The main goal of the analysis here is to compare the different methods, models and algorithms used in earlier chapters. We will analyze the methods used for forecasting and investigate, compare the results obtained, we will also look into investigate changepoints.

A quick look into the summary statistics of the collected information reveals some interesting findings:

- Prophet outperforms SARIMA in 56% percent of selected users and Persistence model in 75% percent of selected users. While SARIMA outperforms the Persistence model in 74%of chosen users.
- SARIMA's most frequent optimal order is

  $$SARIMA(0,0,3)(0,1,1)[7]$$

  (reported 42 times). This means that the majority of SARIMA models use a single seasonal difference, a first order seasonal moving average, and a third order trend moving average.
- Prpohet most frequent optimal parameters (reported 58 times) is :

  $$\{seasonality\_mode = additive,\ \ changepoint\_prior\_scale = 0.01\}$$

  Meaning that most frequent Prophet model think that the trend does not need to change much at what Prophet thinks are changepoints.
- When compared to KPSS, the ADF test is more likely to believe that the number of checks series is stationary.

Let us have a look into the 300 user RMSEs box-plots.

Figure 8.1: Persistence, Prophet and SARIMA models forecast RMSEs..

The figure 8.1 shows that prophet has more outliers than the other two models. This is most likely due to the fact that when we built the Prophet model, the changepoint range was set to be 100% of the train data, resulting in overfitting for a few users. However, the median of prophet and SARIMA is still lower than that of Persistence, with prophet having the lowest median.

Next, we examine the changepoints box-plots:

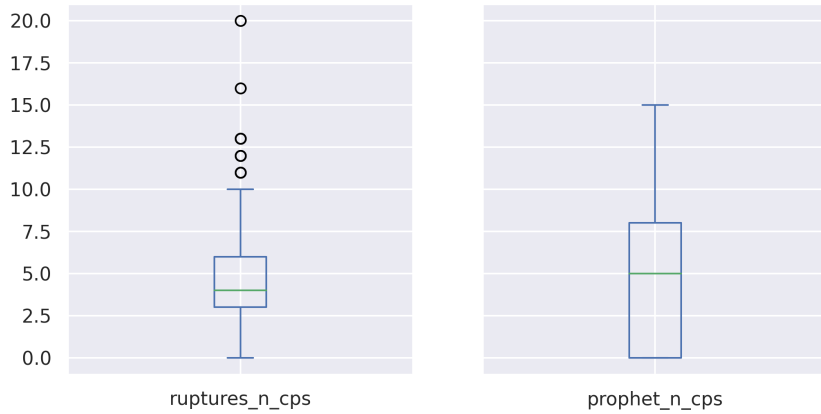

Figure 8.2: Changepoints detected by Ruptures PELT and Prophet..

The left box-plot in Figure 8.2 displays the number of changepoints discovered on the train data using Ruptures PELT with penalty = 3. Ruptures PELT box-plot shows several outliers, which is likely due to the fact that we set the penalty to 3, allowing Ruptures PELT to identify more of what it thinks to be a changepoint. The number of significant changepoints discovered by Prophet under the 0.01 threshold is shown in the right box-plot (Prophet default). It was discovered that 50% of the selected users had 5 or more major changepoints that Prophet may use to forecast. [Kil12] [22a]

Now, let us take a look at the users' number of outliers and missing data, as well as the chosen, transformation lambdas.

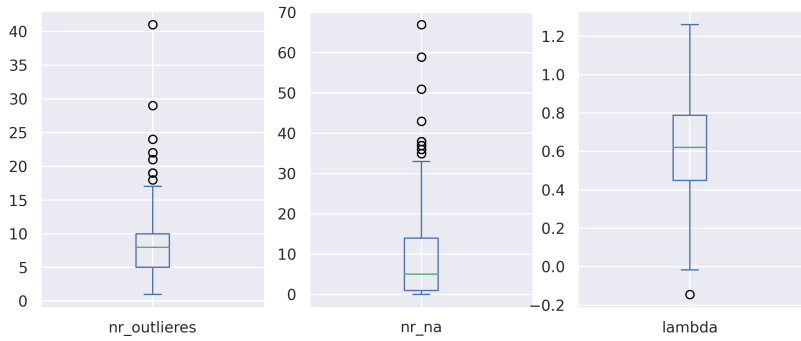

Figure 8.3: Number of outliers, number of missing points and transformation lambdas..

The left box-plot from Figure 8.2 shows the number of detected outliers using Rolling Z-Scores (see 6.2 A.1), we can observe that few users have more than 15 outliers. The number of missing data is shown in the center box-plot, it can be seen that the user with the highest number of missing point still has less than 70 missing records. The lambda utilized by the automated Box-Cox function [22c] is shown in the right box-plot. it's seen that more than half of the chosen users were transformed by a square root transform or no transformation. [Bro20]

Let us have a look at the selected users' RMSE distribution

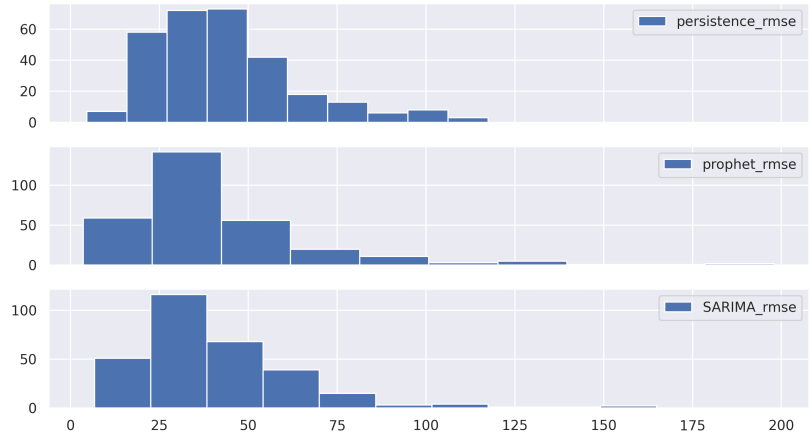

Figure 8.4: Persistence, Prophet and SARIMA models forecast RMSEs histograms.

Figure 8.4 shows that all of the Models' RMSEs are right-tailed, indicating that the RMSEs have an exponential distribution. It can also be noted that Prophet's first bin is larger than the other two model first bins, indicating that Prophet has more lower MSEs when forecasting.
Let us have a look at the correlation between the saved selected users information. It's important to say that correlation does not necessarily imply causation but it might be something analysis.

Let us explore the correlation in data collected by the experiment. It's important to note that correlation here, and in general, does not always suggest causality, but it might be something to look into:

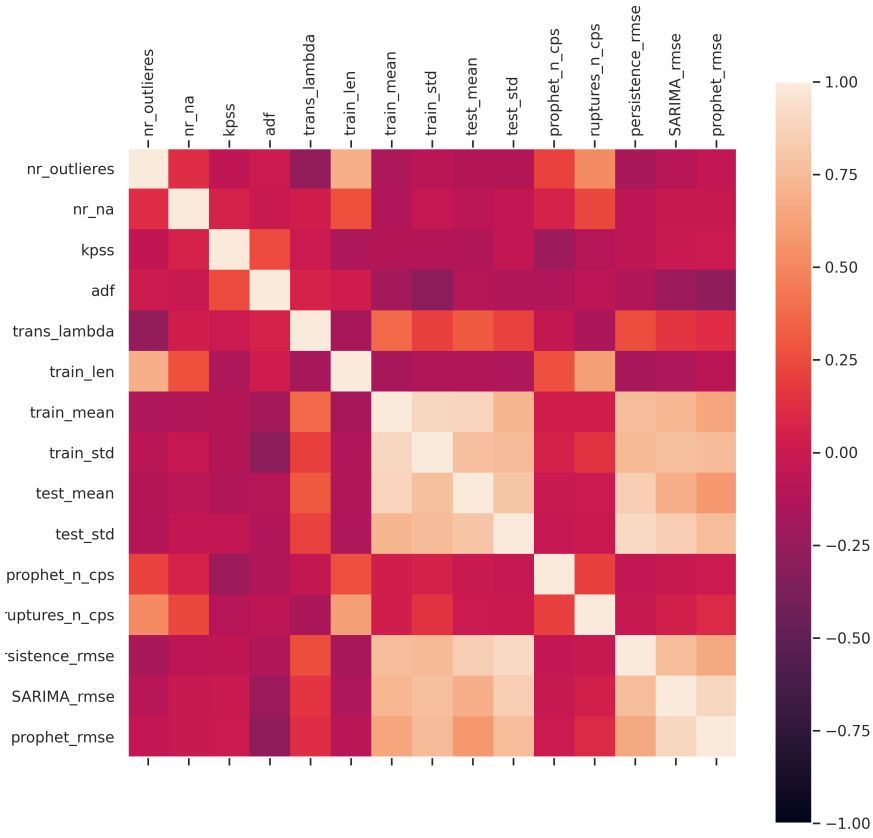

Figure 8.5: Correlation matrix of the data collected by the experiment..

From figure 8.5 we can see That:

- there is almost no correlation between Prophet RMSE and the training size. This indicate that Prophet by utilizing changepoints needed less data in the training stage for example compared to SARIMA.
- There is a positive correlation between the training mean + stander deviation with the RMSE of the three models. This means that the three models have harder time foresting a user with high number of checks!

- There is some positive correlation between KPSS and ADF tests, expected!
- There are also negative correlation between ADF test and training stander deviation, indicating that users with high variations tend to be non-stationary, expected!
- There is a negative correlation between ADF test with SARIMA and Prophet RMSE indicating that as ADF thinks the series is non-stationary Prophet and SARIMA have harder time forecasting.
- There is also some negative correlation between KPSS test and the number of changepoints utilized by Prophet, indicating that as series being non-stationary Prophet tends to utilize more change points.
- There is some negative correlation between the transformation lambda and the number of detected outliers, indicating that automatic Box-Cox function [22c] tends to use square root or log transformations as the number of detected outliers (high positive values) increases, expected!
- There is some positive correlation between the training size with the number of outliers and missing values, expected!
- There is a positive correlation between the number of changepoints detected by ruptures PLT with the training size and also with the number of detected outliers.

CHAPTER 9

Conclusion and Future Work

In this report, we have outlined a set of commonly used univariate times series methods for forecasting. We showed how data processing and time series analysis can help understanding various aspects about the inherent nature of the series so that we are better informed to create meaningful and accurate, automatic, time series forecasting models. We also investigated a set of methods to detect the change points in a time series.

We have introduced a novel methods like seasonal ARIMA and Facebook Prophet, to forecast for 30 days of user daily number of checks time series.

During our initial forecast tasks 6 we saw how Prophet was able to outperform the persistence and seasonal ARIMA models by utilizing change point for forecasting 30 days of user daily number of checks. This was further verified by running an experiment over 300 different users.

The experiment showed that Prophet outperforms SARIMA in 56% percent of selected users and Persistence model in 75% percent of selected user, by utilizing some power to alter the trend at detected change point. By Investigating the matrix correlations of the experiment collected data, We also saw signs that Prophet by utilizing changepoints needed less data in training stage compared to for example seasonal ARIMA.

There is a lot to look at for improvement, one is Walk-Forward Validation, in which the seasonal ARIMA and Prophet models are updated each time step new data is received. Another option is to maintain Prophet's changepoint range at 80% or lower to guarantee that the model does not overfit the training data and can understand the remaining 20% on its own. Another possibility is that if we know where trends have changed in the past, we can add these known changepoints to be utilize by Prophet "holiday effect" or combine rupture detection of changepoints and let Prophet use it. For future work it is tempting to look into multivariate timeseries (e.g. Vector Autoregression -VAR) and Deep Learning for time series (e.g. Long Short-Term

Memory - LSTM).

APPENDIX A

Appendix

A.1 Outlier Detection with Rolling Z-Scores

Time series gaps or big leaps can be found using rolling z-score thresholds.

A z-score, represents the number of standard deviations from the mean.

Using the moving average and moving standard deviation, the z-score upper and lower bound can be calculated, as following:

$$B_U(N) = \mu_{ma}(N) + 2 * \sigma_{ma}(N)$$
$$B_L(N) = \mu_{ma}(N) - 2 * \sigma_{ma}(N)$$

Here, the z-score upper and lower bounds are set to two standard deviations above and below the mean, assuming a normal distribution (the empirical rule).

A.2 Persistence Model

A baseline is required for every time series forecasting task. A performance baseline offers you a sense of how well all other models will perform on your problem. The Zero Rule algorithm is the most often used baseline approach for supervised machine learning. In the case of classification, this method predicts the majority class, whereas in the case of regression, it predicts the average outcome. This might be used to time series, however it does not take into account the serial correlation pattern found in time series datasets. The persistence algorithm is the analogous approach for use with time series datasets. The persistence method predicts the expected outcome at the next time step (t+1) to be the value at the present time step (t) [Bro20].

A.3 Inverse Box-Cox transform given lambda

Function to reveres transform the data [Bro20]:

$$x = \exp(\text{ transform }), \text{ if lambda } = 0$$
$$x = \exp\left(\frac{\log(\text{lambda} \times \text{ transform } + 1)}{\text{lambda}}\right), \text{ if lambda } \neq 0$$

Code1

APPENDIX B

Code

B.1 Experiment Helper Functions

```

# !pip install statsmodels
# !pip install pmdarima
# !pip install prophet
# !pip install --upgrade plotly
# !pip install ruptures

# basic
import os
import random
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import matplotlib.dates as mdates
import seaborn as sns
sns.set()
sns.set(rc={'figure.figsize':(11, 4)})

# Ipython
from IPython.display import display

# warnings
import warnings
warnings.simplefilter("ignore")


# pyspark
import pyspark
from pyspark.sql import SparkSession
from pyspark.sql import functions as F
from pyspark import SparkContext, SparkConf

# StatsModels
from statsmodels.tsa.seasonal import seasonal_decompose
from statsmodels.tsa.stattools import adfuller, kpss
from statsmodels.tsa.arima.model import ARIMA
from statsmodels.tsa.stattools import acf
from statsmodels.graphics.tsaplots import plot_acf, plot_pacf, plot_predict


```

```
# sklearn
from sklearn.preprocessing import PolynomialFeatures
from sklearn.linear_model import LinearRegression
from sklearn.model_selection import ParameterGrid
from sklearn.metrics import mean_squared_error
from sklearn.metrics import mean_absolute_error
from sklearn.model_selection import ParameterGrid

# prophet
from prophet import Prophet
from prophet.diagnostics import cross_validation
from prophet.diagnostics import performance_metrics
from prophet.plot import add_changepoints_to_plot

# pmdarima
import pmdarima as pm
from pmdarima.arima.utils import ndiffs
from pmdarima.arima.utils import ndiffs # estimate ARIMA differencing term

# ruptures
import ruptures as rpt

#  Scipy
from scipy.stats import boxcox


# ========================================================================
    define loading functions
# define load data function
def load_Data(path, spark):
    return spark.read.option('header', True) \
                .option('inferSchema', True) \
                .option('mode', 'DROPMALFORMED')\
                .parquet(path)

def load_data(user_id, spark, Data):
    # process timestamp feature
    spark.conf.set("spark.sql.session.timeZone", "UTC")
    data = Data\
            .withColumn('timestamp',F.col('timezone') + F.col('timestamp'))\
            .select(
                    F.col('user_id'),
                    F.from_unixtime('timestamp').cast("timestamp").alias('
                        timestamp'),
                    F.col('screen_time'),
                    F.col('n_checks'),
                    F.col('n_apps'),
            )

    # get user data
    U = data\
        .where(F.col('user_id') == user_id)\
        .drop(F.col('user_id'))\
        .withColumn("date", F.col("timestamp").cast("date"))

```

```
94      # aggregate over date feature
95      U_aggs = U.groupBy("date")\
96              .agg(F.sum('screen_time'),F.sum('n_checks'),F.percentile_approx('
                    n_apps', 0.5).alias("median(n_apps)"))\
97              .sort('date')
98
99      # to pandas dataframme
100     sample = U_aggs.toPandas()
101
102     # process date feature
103     sample['date'] = pd.to_datetime(sample['date'])
104
105     # set index as date
106     sample = sample.set_index('date')
107
108     # set frequencies
109     sample = sample.asfreq('D')
110
111     return sample
112
113 # ========================================================================
        define checking methods
114 # def slice dataframe function consider only whole months
115 def slice_sample(sample):
116     start = sample.index[sample.index.is_month_start == True][0]
117     end = sample.index[sample.index.is_month_end == True][-1]
118     return sample.loc[start:end, :]
119
120 # def check sample length > 370 + 40
121 def check_sample_len(sample, threshold= 370 + 40):
122     if len(sample) < threshold:
123         return False
124     else: return True
125
126 # def check missing values precentage
127 def check_sample_na(sample, threshold=0.10):
128     for missing_values_precentage in (sample.isna().sum() / len(sample)):
129         if missing_values_precentage >= threshold:
130             return False
131     return True
132
133 # def check missing values for ending months
134 def check_month_ending(sample, threshold=5):
135     groubing = sample[sample.columns[0]].isna().groupby(sample.index.strftime('
            %Y-%m')).sum()
136     if groubing.iloc[-1] > threshold:
137         return False
138     else:
139         return True
140
141 # def check missing values within months
142 def check_month_na(sample, threshold=15):
143     groubing = sample[sample.columns[0]].isna().groupby(sample.index.strftime('
            %Y-%m')).sum()
144     mask = groubing >= threshold  # each month have at least 15 day
```

```
    if sum(mask) > 0: return False
    else: return True

# define outlier detection function with Rolling Z-Scores
def check_outliers(sample, window=7, return_bonds=[]):
    sample = sample.copy()
    dic = {}
    for feature in sample.columns:
        # build DataFrame
        series_df = pd.DataFrame(sample[feature])
        series_df['mean'] = sample[feature].rolling(window=window).mean()
        series_df['std']  = sample[feature].rolling(window=window).std()
        series_df['LB']   = series_df['mean'] - 2 * series_df['std']
        series_df['UB']   = series_df['mean'] + 2 * series_df['std']
        series_df['outliers'] = (series_df['LB'] > series_df[feature]) | (
            series_df[feature] > series_df['UB'])
        # bons selected features
        if feature in return_bonds:
            dic[f'{feature}_LB'] = series_df['LB']
            dic[f'{feature}_UB'] = series_df['UB']
        # remove outliers
        dic[f'{feature}'] = sample[feature].loc[series_df['outliers'] == True]
        sample[feature].loc[series_df['outliers'] == True] = np.nan

    # return
    return sample, dic

# def get number of missing values
def get_n_na(sample):
    lst = np.array([sum(sample[feature].isna()) for feature in sample.columns
        [:3]])
    return lst



# ========================================================================
    define filling methods
# fill missing values with rolling-window:7 and rest_method:'interpolate'.
def fill_rolling_window(sample, window=7, rest_method = 'interpolate'):
    # copy
    copy = sample.copy()

    # Fill with rolling window
    if type(window) is int:
        # update with rolling window
        update = sample[sample.columns[:3]].rolling(window = window,
            min_periods=1, center=True).mean().round()
        copy.update(update, overwrite=False)
        del(update)

    elif (type(window) is list) or (type(window) is np.ndarray):
        for w in window:
            # update with rolling window
            update = sample[sample.columns[:3]].rolling(window = w, min_periods
                =1, center=True).mean().round()
```

```
195             copy.update(update, overwrite=False)
196             del(update)
197 
198     # fill with method
199     if rest_method in ['interpolate', 'ffill', 'bfill']:
200         copy = getattr(copy, rest_method)().round()
201     else:
202         raise NotImplementedError
203 
204     #return
205     return copy
206 
207 # fill missing values with seasonal_mean.
208 def fill_seasonal_mean(sample, n=7, lr=1):
209     # copy
210     copy = sample.copy()
211 
212     # update seasonal_mean
213     for feature in sample.columns[:3]:
214         ts  = sample[feature].values
215         out = np.copy(ts)
216         for i, val in enumerate(ts):
217             if np.isnan(val):
218                 ts_seas = ts[i-1::-n]  # previous seasons only
219                 if np.isnan(np.nanmean(ts_seas)):
220                     ts_seas = np.concatenate([ts[i-1::-n], ts[i::n]])  #
                        previous and forward
221                 out[i] = np.nanmean(ts_seas) * lr
222         copy[feature] = out
223 
224     #return
225     return copy.round()
226 
227 def chose_filling_method(sample, drop_rate=0.10):
228     # create data
229     rate = 0.10 # rate of drop
230     sample_no_na = sample.dropna()
231     ix = np.random.choice(sample_no_na.index, size=int(sample_no_na.shape[0]*
            rate), replace=False)
232     sample_with_na = sample_no_na.copy()
233     for row in ix:
234         for col in sample_with_na.columns:
235             sample_with_na.loc[row,col] = np.nan
236 
237     # compare
238     sample_filled_RW = fill_rolling_window(sample_with_na, window = 7,
            rest_method = 'interpolate')
239     sample_filled_SM = fill_seasonal_mean(sample_with_na, n = 7, lr= 1)
240 
241     M01, M02 = [], []
242     for feature in sample_no_na.columns:
243         values_raw = sample_no_na[feature].values
244         values_RW = sample_filled_RW[feature].values
245         values_SM = sample_filled_SM[feature].values
246         M01.append(np.round(mean_squared_error(values_raw,values_RW),2))
```

```
247         M02.append(np.round(mean_squared_error(values_raw,values_SM),2))
248
249     # save results
250     test_result = ~np.greater(M01,M02)
251
252     # fill missing values
253     if sum(test_result) > 1:
254         sample_filled = fill_rolling_window(sample, window = 7, rest_method = '
                interpolate')
255         return sample_filled, 'rolling window'
256     else:
257         sample_filled = fill_seasonal_mean (sample, n = 7, lr= 1)
258         return sample_filled, 'seasonal mean'
259
260
261 # ========================================================================
        define stationary tests
262 # adfuller
263 def adfuller_test(sample, verbose=True): # H0 the series is Non-stationary
264     if verbose: print('adfuller-test:')
265     if type(sample) is pd.core.frame.DataFrame:
266         for feature in sample.columns[:3]:
267             result = adfuller(sample[feature])
268             if verbose: print(feature + ': ' + ('Non-Stationary' if result[1] >
                    0.05 else 'Stationary'))
269     else:
270         result = adfuller(sample)
271         if verbose: print(sample.name + ': ' + ('Non-Stationary' if result[1] >
                0.05 else 'Stationary'))
272         return False if result[1] > 0.05 else True
273
274
275 # kpps
276 def kpss_test(sample, verbose=True): # H0 the series is stationary
277     if verbose: print('kpps-test:')
278     if type(sample) is pd.core.frame.DataFrame:
279         for feature in sample.columns:
280             result = kpss(sample[feature])
281             if verbose: print(feature + ': ' + ('Stationary' if result[1] >
                    0.05 else 'Non-Stationary'))
282     else:
283         result = kpss(sample)
284         if verbose: print(sample.name + ': ' + ('Stationary' if result[1] >
                0.05 else 'Non-Stationary'))
285         return True if result[1] > 0.05 else False
286
287
288 # ========================================================================
        define metrics
289 # function to compute Accuracy metrics
290 def forecast_accuracy(forecast, actual, criteria = 'all'):
291     if criteria == 'all':
292         mape = np.mean(np.abs(forecast - actual)/np.abs(actual))  # MAPE
293         me = np.mean(forecast - actual)             # ME
294         mae = np.mean(np.abs(forecast - actual))    # MAE
```

```
        mpe = np.mean((forecast - actual)/actual)   # MPE
        rmse = np.mean((forecast - actual)**2)**.5  # RMSE
        corr = np.corrcoef(forecast, actual)[0,1]   # corr
        mins = np.amin(np.hstack([forecast[:, None], actual[:,None]]), axis=1)
        maxs = np.amax(np.hstack([forecast[:,None], actual[:,None]]), axis=1)
        minmax = 1 - np.mean(mins/maxs)             # minmax
        acf1 = acf(forecast-test)[1]                # AC1
        return({'mape':mape, 'me':me, 'mae': mae,
                'mpe': mpe, 'rmse':rmse, 'acf1':acf1,
                'corr':corr, 'minmax':minmax})
    elif criteria == 'mape':
        mape = np.mean(np.abs(forecast - actual)/np.abs(actual))  # MAPE
        return mape
    elif criteria == 'rmse':
        rmse = np.mean((forecast - actual)**2)**.5  # RMSE
        return rmse
    else:
        raise NotImplementedError('use criteria = all')
        return


# ========================================================================
    define power transformer
# def Box-Cox series transformer
def boxcox_transformer(series):
    series_trans , lam = boxcox(series)
    series_trans = pd.Series(series_trans, index=series.index, name=f'{series.
        name}')
    return series_trans, lam

# invert Box-Cox transform
def boxcox_inverse(value, lam):
    if lam == 0: return np.exp(value)
    else: return np.exp(np.log(lam * value + 1) / lam)
```

Listing B.1: Experiment Helper Functions.

B.2 Experiment Pipe

```

# ---
#  # <span style="color:BlueViolet">Init workspace </span>
# ---

# ### Load Libraries

# !pip install statsmodels
# !pip install pmdarima
# !pip install prophet
# !pip install --upgrade plotly
# !pip install ruptures

```

```
# basic
import os
import random
import numpy as np
import pandas as pd
import matplotlib
import matplotlib.pyplot as plt
import matplotlib.dates as mdates
import seaborn as sns
sns.set()
sns.set(rc={'figure.figsize':(11, 4)})

# Ipython
get_ipython().run_line_magic('matplotlib', 'inline')
from IPython.display import display

# warnings
import warnings

# pyspark
import pyspark
from pyspark.sql import SparkSession
from pyspark.sql import functions as F
from pyspark import SparkContext, SparkConf

# StatsModels
from statsmodels.tsa.seasonal import seasonal_decompose
from statsmodels.tsa.stattools import adfuller, kpss
from statsmodels.tsa.arima.model import ARIMA
from statsmodels.tsa.stattools import acf
from statsmodels.graphics.tsaplots import plot_acf, plot_pacf, plot_predict

# sklearn
from sklearn.preprocessing import PolynomialFeatures
from sklearn.linear_model import LinearRegression
from sklearn.model_selection import ParameterGrid
from sklearn.metrics import mean_squared_error
from sklearn.metrics import mean_absolute_error
from sklearn.model_selection import ParameterGrid

# prophet
from prophet import Prophet
from prophet.diagnostics import cross_validation
from prophet.diagnostics import performance_metrics
from prophet.plot import add_changepoints_to_plot

# pmdarima
import pmdarima as pm
from pmdarima.arima.utils import ndiffs
from pmdarima.arima.utils import ndiffs # estimate ARIMA differencing term

# ruptures
import ruptures as rpt

#  Scipy
```

```
69 from scipy.stats import boxcox
70
71 # local
72 from helper_functions import *
73
74
75 # ### Init Spark Session
76
77 # set memory and cores:
78 memory = 10
79 cores = 5
80
81 # assert resources
82 assert memory < 20 , 'memory should not exceed 20'
83 assert cores < 10, 'number of cores should not exceed 10'
84
85 # Sets memory limit on driver and to use all CPU cores
86 conf = SparkConf()        .set('spark.ui.port', '4050')        .set('spark.
       driver.memory', str(memory)+'g')         .setMaster(f'local[{str(cores)}]')
87 if 'sc' in locals(): sc.stop()
88 sc = pyspark.SparkContext(conf=conf)
89 spark = SparkSession.builder.getOrCreate()
90
91 # free memory
92 del(memory, cores)
93
94 # ### Load Data
95
96 # load data
97 Data = load_Data("Sorry!", spark)
98
99 # ---
100 # ### Flag DataFrames
101 # ---
102
103 # ========= Set DataFrame for saving
104
105 # general data frame
106 columns_general = [
107     'user_id',
108     'len_sample_before',
109     'len_sample_after',
110     'filling_method',
111     ]
112
113 # feature dataframe for all feaures
114 columns_feature = [
115     'user_id',
116     'nr_outlieres',
117     'nr_na',
118     'kpss',
119     'adf',
120     'ruptures_cps_train',
121     'lambda',
122     'split_date',
```

```
    'train_len',
    'train_mean',
    'train_std',
    'test_len',
    'test_mean',
    'test_std',
    'persistence_rmse',
    'persistence_mape',
    'prophet_params',
    'prophet_cps',
    'prophet_rmse',
    'prophet_mape',
    'SARIMA_order',
    'SARIMA_rmse',
    'SARIMA_mape',
    ]

DF_G  = pd.DataFrame(index=range(310) ,columns=columns_general)   # This df is about general user's data info
DF_F0 = pd.DataFrame(index=range(310) ,columns=columns_feature)   # This df is about user feature 01
DF_F1 = pd.DataFrame(index=range(310) ,columns=columns_feature)   # This df is about user feature 02
DF_F2 = pd.DataFrame(index=range(310) ,columns=columns_feature)   # This df is about user feature 02
DF_F  = [DF_F0, DF_F1, DF_F2]

# ---
# # <span style="color:BlueViolet">Pip </span>
# ---

# checked users list
checked_users = []
_user_nr_  = 0

while _user_nr_ < 300 and len(checked_users)<5000:
    try:
        # ============================================================ Load Users for analysis
        # ============================ Set local flags
        # set continue flag
        _continue_ = True
        _user_id_  = int(np.random.randint(1, 10358, 1)[0]) # 40
        print(f'Trying user: {_user_nr_} with id: {_user_id_}:')

        # check user id
        _continue_ = _user_id_ not in checked_users
        if not _continue_ :
            print(f'user: {_user_nr_} with id: {_user_id_} failed chek 01!')
            continue

        # append user
        checked_users.append(_user_id_)

        # Update # // 'user_id' = _USER_
```

```
        DF_G ['user_id'][_user_nr_] = _user_id_
        for i in range(3): DF_F[i]['user_id'][_user_nr_] = _user_id_


        # ============================ Load User
        sample = load_data(_user_id_, spark, Data)

        # ============================ Checks and Preprocessing
        # check
        _continue_ = check_sample_len(sample) # check sample length > 370 + 40
        if not _continue_ :
            print(f'user: {_user_nr_} with id: {_user_id_} failed chek 02!')
            continue

        # Update # // 'len_sample_before',
        DF_G['len_sample_before'][_user_nr_] = len(sample)

        # slice
        sample = slice_sample(sample) # slice stating and ending months if not
            full
        for i in range(5): # check the last 2 months Remove in case of matching
            if check_month_ending(sample, threshold=5):
                break
            else:
                sample = sample.drop(sample.last('1M').index)

        # check
        _continue_ = (check_sample_len(sample) & # check sample length > 370 +
            40
                      check_sample_na(sample) &  # check missing values
                          precentage < 0.10
                      check_month_ending(sample) &  # check ending month > 25
                      check_month_na(sample)) # check missing values within
                          months < 15
        if not _continue_ :
            print(f'user: {_user_nr_} with id: {_user_id_} failed chek 03!')
            continue

        # Update # // 'len_sample_after',
        DF_G['len_sample_after'][_user_nr_] = len(sample)

        # Update # // 'nr_na' * 3
        n_na_lst = get_n_na(sample)
        for i in range(3): DF_F[i]['nr_na'][_user_nr_] = n_na_lst[i]

        # ============================ Outlier Detection
        # remove outliers in-sample
        sample , outliers_dict = check_outliers(sample)

        # Update # // 'nr_outlieres' *3
        for i in range(3): DF_F[i]['nr_outlieres'][_user_nr_] = len(
            outliers_dict[sample.columns[i]])

        # check
```

```
222         _continue_ = (check_sample_na(sample) &   # check missing values
                precentage < 0.10
223                       check_month_na(sample,20) & # check missing values within
                              months < 20
224                       check_month_ending(sample)) # check ending month > 25 and
                              length
225         if not _continue_ :
226             print(f'user: {_user_nr_} with id: {_user_id_} failed chek 04!')
227             continue
228
229         # ============================== Fill Missing Values
230         sample_filled, filling_method = chose_filling_method(sample, drop_rate
                =0.10)
231
232         # Update # // 'filling_method',
233         DF_G['filling_method'][_user_nr_] = filling_method
234
235         # free memory
236         del(sample)
237
238         # ============================== Stationary Test
239         # Update # // 'kpss', * 3
240         for i in range(3):
241             DF_F[i]['kpss'][_user_nr_] = kpss_test(sample_filled[sample_filled.
                    columns[i]], verbose=False)
242
243         # Update # // 'adf',  * 3
244         for i in range(3):
245             DF_F[i]['adf'][_user_nr_] = adfuller_test(sample_filled[
                    sample_filled.columns[i]], verbose=False)
246
247         # ============================================================== Loop
                through features
248         print(f' Analysing user: {_user_nr_} with id: {_user_id_}:')
249         for feature_index in range(3):
250             # ============== Load Feature Data
251             # select feature
252             feature = sample_filled.columns[feature_index]
253
254             # select series
255             series = sample_filled[feature]
256
257             # ============== Split Train Test
258             # split
259             split_date = series.index[-30]
260             train = series.loc[(series.index  < split_date)]
261             test  = series.loc[(series.index >= split_date)]
262
263             # Update
264             # // 'train_len', 'train_mean', 'train_std', 'test_len', 'test_mean
                    ', 'test_std',
265             DF_F[feature_index]['split_date'][_user_nr_] = str(split_date)
266             DF_F[feature_index]['train_len'][_user_nr_] = len(train)
267             DF_F[feature_index]['train_mean'][_user_nr_] = np.mean(train)
268             DF_F[feature_index]['train_std'][_user_nr_] = np.std(train)
```

```
269             DF_F[feature_index]['test_len'] [_user_nr_] = len(test)
270             DF_F[feature_index]['test_mean'][_user_nr_] = np.mean(test)
271             DF_F[feature_index]['test_std'][_user_nr_] = np.std(test)
272
273
274             # ========================== Change Points Methods
275
276             # ==============  Ruptures Module
277             # Detect the change points with The Ruptures Module
278             algo = rpt.Pelt(model="rbf").fit(train.values)
279             change_location = algo.predict(pen=3)
280
281             # Update 'ruptures_cps_train',
282             cps = list(train.index[np.array(change_location)[:-1]])
283             DF_F[feature_index]['ruptures_cps_train'][_user_nr_] = str(cps)
284
285             # free memory
286             del(algo, change_location)
287
288             # ========================== Transform
289             train, lam = boxcox_transformer(train)
290
291             # Update 'lambda',
292             DF_F[feature_index]['lambda'][_user_nr_] = lam
293
294             # ============== Persistence Predctions Model
295             # Persistence model
296             persistence = test.shift(1).values
297             persistence = boxcox(persistence, lmbda=lam)
298             persistence[0] = train[-1]
299
300             # Update 'persistence_rmse',
301             prd = boxcox_inverse(persistence, lam) # prd = np.where(np.isnan(
                    prd), 0, prd)
302             obs = test.values
303             rmse = forecast_accuracy(prd, obs, criteria='rmse')
304             DF_F[feature_index]['persistence_rmse'][_user_nr_] = rmse
305
306             # Update 'persistence_mape',
307             mape = forecast_accuracy(prd, obs, criteria='mape')
308             DF_F[feature_index]['persistence_mape'][_user_nr_] = mape
309
310             # free memory
311             del(persistence, prd, obs, rmse, mape)
312
313
314             # ========================== Prophet Model
315
316             # ============== Prophet Format
317             # load train datafrane
318             X = pd.DataFrame({ 'ds': train.index, 'y':  train.values})
319
320             # ============== Tuning with CV
321             # set parameters grid
322             grid = {
```

```
            'changepoint_range': [1],
            'changepoint_prior_scale': [0.01, 0.05, 0.1 , 0.5 , 0.75],
            'seasonality_mode'       : ['multiplicative','additive'],
            }

        grid = ParameterGrid(grid)

        # Tune with cross validation to evaluate all parameters
        rmse = []
        for params in grid:
            m = Prophet(**params).fit(X)  # it model with given params
            df_cv = cross_validation(m, horizon='30 days', period='30 days'
                , parallel="processes")
            df_p  = performance_metrics(df_cv, rolling_window=1)
            rmse.append(df_p['rmse'].values[0])

        # ============== Final Model
        # save best parameters
        best_parameters = grid[np.argmin(rmse)]
        # best_parameters = {'seasonality_mode': 'additive',
        #                    'changepoint_range': 1,
        #                    'changepoint_prior_scale': 0.01}

        # Update 'prophet_params',
        DF_F[feature_index]['prophet_params'][_user_nr_] = str(
            best_parameters)

        # fit
        model = Prophet(**best_parameters)
        model.fit(X)

        # forecast
        future = model.make_future_dataframe(periods=len(test), freq='D')
        forecast = model.predict(future)

        # Update 'prophet_cps',
        _, ax = plt.subplots();
        a = add_changepoints_to_plot(ax,model,forecast, trend=False,
            threshold=0.01);
        plt.close()
        cps= []
        for i in range (len(a)):
            cps.append(matplotlib.artist.get(a[i], property='xdata')[0])
        DF_F[feature_index]['prophet_cps'][_user_nr_] = str(cps)

        # Update 'prophet_rmse',
        prd = boxcox_inverse(forecast.iloc[-len(test):]['yhat'].values, lam
            ) # prd = np.where(np.isnan(prd), 0, prd)
        obs = test.values
        rmse = forecast_accuracy(prd, obs, criteria='rmse')
        DF_F[feature_index]['prophet_rmse'][_user_nr_] = rmse

        # Update 'prophet_mape',
        mape = forecast_accuracy(prd, obs, criteria='mape')
        DF_F[feature_index]['prophet_mape'][_user_nr_] = mape
```

```
374
375             # free memory
376             del(X, grid, future, forecast, best_parameters, model)
377             del(prd, obs, rmse, mape)
378
379             # ========================== ARIMA Model
380
381             # ============== Auto ARIMA
382             # Seasonal - fit stepwise auto-ARIMA
383             smodel = pm.auto_arima(train,
384                                    start_p=1, start_q=1,
385                                    test='adf', # use adftest to find optimal #
                                           adf # kpss
386                                    max_p=3, max_q=3, # maximum p and q
387                                    m=7, # frequency of series
388                                    start_P=0,
389                                    seasonal=True, # with Seasonality
390                                    d=None, # let model determine 'd'
391                                    D=1, # enforce D=1 for a given frequency m=7
392                                    trace=True,
393                                    error_action='ignore',
394                                    suppress_warnings=True,
395                                    stepwise=True)
396
397             # forecast
398             fitted = smodel.predict(n_periods=len(test), return_conf_int=False)
399
400             # Update
401             # //    'SARIMA_order',
402             DF_F[feature_index]['SARIMA_order'][_user_nr_] = str(smodel)
403
404             # Update 'SARIMA_rmse',
405             prd = boxcox_inverse(fitted, lam) # prd = np.where(np.isnan(prd),
                    0, prd)
406             obs = test.values
407             rmse = forecast_accuracy(prd, obs, criteria='rmse')
408             DF_F[feature_index]['SARIMA_rmse'][_user_nr_] = rmse
409
410             # Update //    'SARIMA_mape',
411             mape = forecast_accuracy(prd, obs, criteria='mape')
412             DF_F[feature_index]['SARIMA_mape'][_user_nr_] = mape
413
414             # free memory
415             del(feature, series, train, test,
416                 lam, smodel, fitted)
417
418         # ============================ Update Flags
419         _user_nr_ += 1
420
421         # ============================ Save DataFrames
422         DF_G.to_csv('DF_G.csv')
423         for i in range(3): DF_F[i].to_csv(f'DF_F{i}.csv')
424
425         # ============================ Print
426         print('_'*80)
```

```
427         print('_'*80)
428         print('\n'*5)
429         print(f'user: {_user_nr_} with id: {_user_id_} Done!')
430         print('\n'*5)
431         print('_'*80)
432         print('_'*80)
433 
434     except Exception as e:
435         print(e)
436         print('passed checks learning error')
437         continue
```

Listing B.2: Experiment Pip.

Bibliography

[22a] *Pelt*. July 8, 2022. URL: https://centre-borelli.github.io/ruptures-docs/code-reference/detection/pelt-reference/#ruptures.detection.pelt.Pelt.

[22b] *Quick Start*. July 8, 2022. URL: https://facebook.github.io/prophet/docs/quick_start.html.

[22c] *scipy.stats.boxcox — SciPy v1.8.1 Manual*. July 8, 2022. URL: https://docs.scipy.org/doc/scipy/reference/generated/scipy.stats.boxcox.html.

[Bro20] Jason Brownlee. *Introduction to Time Series Forecasting With Python*. Machine Learning Mastery. Jason Brownlee (eBooks), 2020. ISBN: self-published.

[Kil12] David Killick. "Killick, D. (2012) Seeing-Ourselves-in-the-World: Developing Global Citizenship Through International Mobility and Campus Community." In: *Journal of Studies in International Education* 16 (September 2012), pages 372–389. DOI: 10.1177/1028315311431893.

[Kri22] Mitchell Krieger. *Time Series Analysis with Facebook Prophet: How it works and How to use it*. February 2, 2022. URL: https://towardsdatascience.com/time-series-analysis-with-facebook-prophet-how-it-works-and-how-to-use-it-f15ecf2c0e3a.

[pmd] pmdarima. *pmdarima: ARIMA estimators for Python — pmdarima 1.8.5 documentation*. URL: http://alkaline-ml.com/pmdarima/.

[Pra19] Selva Prabhakaran. *Time Series Analysis in Python – A Comprehensive Guide with Examples*. 2019. URL: https://www.machinelearningplus.com/time-series/time-series-analysis-python/.

[Spa] Apache Spark. *PySpark Documentation — PySpark 3.3.0 documentation*. URL: https://spark.apache.org/docs/latest/api/python/#:%5C%7E:text=PySpark%5C%20is%5C%20an%5C%20interface%5C%20for,data%5C%20in%5C%20a%5C%20distributed%5C%20environment. (visited on July 27, 2022).

[TL17] Sean Taylor and Benjamin Letham. *Forecasting at scale*. September 27, 2017. URL: https://peerj.com/preprints/3190/.

[Wal19] Jennifer Walker. *Tutorial: Time Series Analysis with Pandas*. 2019. URL: www.dataquest.io/blog/tutorial-time-series-analysis-with-pandas.

[Wik22] Wikipedia contributors. *Akaike information criterion*. In: June 3, 2022. URL: https://en.wikipedia.org/wiki/Akaike_information_criterion (visited on July 27, 2022).