Norwegian School of Economics

Bergen, Autumn, 2020

Good Days Ahead - Forecasting Ticket Sales for Go Fjords Using Weather Data

Christian Slåen Svendsen

Supervisor: Jonas Andersson

Master thesis, Economics and Business Administration,

Business Analytics

NORWEGIAN SCHOOL OF ECONOMICS

Acknowledgements

I would like to thank my thesis supervisor Professor Jonas Andersson of the Norwegian School of Economics, for giving me great advice in defining the angle of the study and for expanding my horizon of techniques within prediction and forecasting.

From Go Fjords I would like to thank CFO Silje Ytterdal Hopland for letting me study your company.

Finally, I would like to thank Arnt-Henning Moberg of TietoEVRY for giving me this opportunity and connecting me with Go Fjords, and for giving me invaluable technical guidance and feedback throughout the whole process.

This thesis has been a fantastic learning experience for me, introducing me to the many intricacies of successfully applying Business Analytics. I have had fun along the way, and I look forward to learning more about demand forecasting.

To everyone who have helped me, thank you.

Sincerely,

[illegible]

Christian Svendsen

December 20th, 2020

Abstract

The purpose of this study is to evaluate whether public weather data from MET Norway can be used to improve ticket sales forecasts for the travel company Go Fjords, and to demonstrate how such a forecasting model can be technically implemented to provide value over time. The problem statement is defined as follows:

Can weather forecast data make demand forecasts more accurate for Go Fjords, and how can business value be derived from such a forecast?

Throughout the study, a wide range of methods were used. This thesis outlines how to retrieve historical weather data from MET Norway's 'Frost API', how to scrape weather forecast data off yr.no, and how to assemble the data for use by forecasting models.

The following model types and frameworks were tested: A Generalized Additive Model (Facebook Prophet), a Dynamic Generalized Linear Model (PyBats), and a Random Forest Regression model created by Microsoft Azure's automated machine learning functionality. Performance metrics are discussed in depth, and Root Mean Squared Error was chosen as the basis for evaluation and comparison. A set of univariate 'benchmark' models were created to answer the problem statement: a naïve forecasting model and a seasonal ARIMA model.

The Facebook Prophet model was used to demonstrate deployment and was implemented to run daily in Microsoft Azure. The forecasts were pushed daily to Go Fjords' database, and made visible in their Microsoft Power BI dashboard, along with actionable advice on the optimal number of buses to rent, taking future weather into account.

The Prophet model performed worse than expected, and the PyBats model performed very well. Potential causes and ways to adjust the models are discussed. ARIMA and Random Forest Regression had similar RSME scores, strengthening the validity of their results.

To conclude: It is possible to create better demand forecasts for Go Fjords by using weather data, rather than by basing forecasts on sales data alone. By optimizing the models for RMSE, variance is minimized, consequently minimizing the frequency at which Go Fjords deploys the wrong number of buses, thus capturing more revenue and achieving cost savings.

Contents

List of figures

List of tables

1. Introduction

Go Fjords is a Norwegian tour operator owned by Det Stavangerske Dampskibselskap, a company with more than 150 years of history. Go Fjords provides tours all over Norway to natural outdoor tourist attractions like Preikestolen and Kjeragbolten, many of which take place in the fjords of the Norwegian west coast. The tours typically include transportation by means of ferry or bus, as well as tour guides, meals, and more. Certain tours also involve activities such as hiking, kayaking, biking, or dog sledding.

1.1 Topic question

As most of Go Fjords' tours are outdoor experiences, demand may be affected by weather conditions. For instance, if you are planning for the upcoming weekend, you might decide not to purchase tickets for an outdoor trip if weather forecast services predict it will rain all weekend.

This leads us to the topic question of this study:

Can weather forecast data make demand forecasts more accurate for Go Fjords, and how can business value be derived from such a forecast?

As the topic question can be viewed as a forecasting problem, the main topic question will be supplemented with a general formulation of a forecasting problem. *Forecasting* is defined as predicting the future values of a series using current information, where current information consists of current and past values of the series and other exogenous series (Yau, 2018). This thesis will construct and present forecasts of ticket sales based on past sales data, before attempting to improve the forecast accuracy by adding past observed weather data, as well as forecasted weather for the upcoming week, as predictor variables.

1.2 Usefulness and aim of the study

Demand for outdoor experiences depends greatly on the weather. This study aims to use weather data to forecast demand for up to seven days forward in time. The reason for this specific forecast horizon is that the forecast will be based on weather forecast data made available by the website yr.no, which only makes detailed predictions up to nine days forward in time. A seven-day horizon makes the forecast one week long, and removes some cases where the forecast would not have weather data for all of its predictions, in a few cases where weather data for eight and nine days forward was missing.

As of now, Go Fjords are not using any sophisticated method of forecasting future tickets sales, so any model with somewhat correct forecasts will be of value. Still, the mark should be set higher than ‘any forecast will do’. The benchmark for any model to be considered successful is that the addition of the weather data leads to a more accurate forecast than a forecasting model purely using historical sales data. Such a benchmark model could, for example, be a *naïve* model predicting that the ticket sales for all days of the upcoming week will equal the most recent day, or an *ARIMA* model forecasting solely based on trends in past ticket sales.

Having a good forecast for how many tickets one will have sold for each coming business day, allows the company to increase their capacity to meet surges in demand, and avoid wastefully high capacity when demand falls. Examples of how Go Fjords can benefit from the increased forecast accuracy is to decide how many buses to rent and tour guides to deploy, based on the weather adjusted demand forecasts. The different trips vary in nature, some involving more outdoor activity than others. “Preikestolen tur-retur” (round-trip to Pulpit Rock) is Go Fjords’ most popular tour. The trip involves a bus ride from Stavanger to the site, a good couple of hours of hiking up a mountain, and a bus trip back to Stavanger. Because of the hike, the trip is more popular when the weather is good. Having a good forecast and thus better basis for better decision making for the best-selling trip, as opposed to a less popular trip, yields more business impact. For these reasons, the Preikestolen trip was chosen as the case study of the thesis.

This thesis is intended as an exercise in data science, going further in demonstrating how insights can be derived, how to create and implement predictive models, use them for inference and present the forecast along with advice in a self-service Business Analytics solution, as opposed to a one-time data analysis report. In addition to answering the topic question regarding whether the use of weather variables as predictors can strengthen the ticket sales forecast, the thesis aims to provide value through improving the basis for decision making for Go Fjords, demonstrating the journey towards that goal, discussing considerations along the way.

2. Methodology

This section presents the methods used in acquiring data, preparing- and exploring the data, selecting appropriate forecasting algorithms, and measurements to evaluate their performance by. Although this may seem like a straightforward waterfall-style process and is sectioned as such in this chapter, it is commonplace to move back and forth through these stages. For example, transforming the data can lead to the exploration yielding more insights on how the data should further be prepared to be more useful. Data science is an iterative process, so having an agile approach can pay off, especially in the early stages. To reflect this reality, the 'Data' section that often precedes the 'Methodology' section in dissertations, is here included in the Methodology chapter. Data preparation- and exploration was integral to the method of the study and was thereby done iteratively throughout the whole process.

2.1 Data

Through their website *yr.no* and their public '*Frost*' Application Programme Interface (API), the Norwegian Meteorological Institute publishes weather data for free public use. Observed, historical weather data was gathered from the API, and forecasted weather for the coming week was gathered from the yr.no website by 'web scraping'. Below is a generic example table depicting the schema of the final data set that is used for model training and inference.

Table 1: Data schema

Date	**Quantity**	**Temperature**	**Precipitation**	**WindSpeedMps**
t = - 1	12	2.31	0.00	3.91
t = 0	18	3.14	0.82	12.90
t + 1	Y	4.74	0.03	1.34

'Date' is the variable that making the data a *time series* dataset. The Date variable is stored as a 'DateTime' datatype (for example having the format '2019-05-21') in the database and tables as it is being processed. In the example table above, *t* is the date of today.

'Quantity' is the total number of tickets sold within the start of the trip, and it is the target variable that the models will predict. In the example table above, future values of Quantity are thus unknown and simply represented by 'Y'. The Quantity variable is stored as a positive 'integer' value, meaning it must always be a whole, non-negative number.

'Temperature', 'Precipitation', and 'WindSpeedMps' are three weather-related variables created from data gathered from yr.no and the Frost API, telling us about the past observations of the weather or future forecasted weather. These variables are of the 'float' data type, meaning they can have decimals, unlike the Quantity variable.

The exploratory analysis section will further present each of the variables and their characteristics.

2.1.1 Acquiring internal data: Go Fjords

For this thesis, Go Fjords shared their historical sales data. This included time of sale, which trip was ordered, journey start date, and many other features. The data was made available through granted access to their Microsoft Azure SQL database, in which they store all their data. The relevant data were extracted by using SQL queries, then it was written to *Comma Separated Values* (CSV) format so it could easily be read by other applications for processing and exploration, forecasting, and presentation of results.

2.1.2 Acquiring external data: the weather

Yr.no is the biggest Norwegian provider of weather forecast information and is managed by the Norwegian Broadcasting Corporation (NRK) in collaboration with the Norwegian Meteorological Institute (MET Norway). As these are state-sponsored entities, they provide most of their forecast information for free public use through public APIs, in XML and other data formats.

Retrieving historical weather data from 'Frost API'

MET Norway provides actual weather data from the past in a machine-readable format through their open API, "Frost". For this project, the Frost API was used to get all historical data from a weather station in Stavanger, the city from which the trip to Preikestolen starts. The API-call was later integrated into the data pipeline to gather new historical data daily, to tune the models and make new forecasts.

To help third parties make use of their weather data, MET Norway provides script templates in both Python and R programming languages for downloading historical data through their API. These scripts require very little additional coding to work. As a third party, you must simply sign up with an email to receive a user authentication token which the service can recognize you by. Then the service is free to use. The need for an ID token is presumably so MET Norway can know if any particular user is violating their terms of service, for instance by spamming them with requests. You also need to specify which weather station you would like data from, which time interval you are interested in, and for which "element ID's" you want data, meaning what type of weather data you are interested in (temperature, precipitation, wind speed, cloud cover, humidity, and more). The API has extensive documentation to help users retrieve information from it.

Scraping weather forecast from yr.no

To make forecasts, meaning predictions of future values, it is necessary to make assumptions about the weather at that point in time. The Frost API sadly only makes available historical weather data, not any of its weather forecasts, from the past or present. This is likely because weather forecasts change rapidly and storing past forecasts would require massive amounts of storage space, with questionable business value. To get an educated guess regarding future weather for the predictive variables, data would have to be gathered directly from yr.no, where MET Norway continuously publish and update their forecast for the weather one week ahead in time.

To retrieve the weather forecast from yr.no, a Python script was created uses the '*requests*' package to get the raw XML data from the source of the web-page displaying the weather forecast. In this case, the webpage containing the relevant raw data was ‘https://www.yr.no/place/Norway/Rogaland/Stavanger/Stavanger/forecast.xml' (MET Norway, 2020).

To extract the weather data, the ‘*BeautifulSoup*’ package was used. BeautifulSoup is useful for *parsing* (reading and extracting information from) structured data, like XML and HTML. The XML was converted to a ‘soup’ object which could be queried to retrieve needed data in a simple manner. The script was configured to run daily along with the API-call, to get forecasted weather to use as input to the forecasting models. See the appendix for code used to retrieve and parse the weather forecast data in this project.

2.1.3 Cleaning and pre-processing

To get familiar with the data at the very start of the thesis work, a batch of historical data from Go Fjords was downloaded. The data was cleaned and pre-processed using R, to make it easier to explore and derive insights from. This *Extract-Transform-Load* (ETL) process was later refined to be re-useable and efficient, so it could run as part of the deployed model.

Some ‘cleaning and pre-processing’ had already taken place in the extraction process, for example using the BeautifulSoup Python package to remove unnecessary data from the retrieved weather forecast data. However, the historical weather data required further ‘wrangling’.

The data from the Frost API was returned in a ‘long’ format (few columns, many rows) that had a column ‘ElementID’, with text values identifying the element (temperature, precipitation, wind…) that the row gives information on, with the recorded value of the element stated in a separate ‘Value’ column. Thus, there are several rows per ‘Date’. The target schema needed for the forecasting models was unique rows for each Date, with one column per variable. To achieve this, a ‘for-loop’ was created in Python to keep one row per ‘Date’ and distribute the weather values in the ‘Value’ column onto three columns corresponding to the respective predictive variables ‘Temperature’, ‘Precipitation’, and ‘WindSpeedMps’. The retrieved values were rounded to two decimals, for the sake of consistency, while simultaneously keeping as much information as possible.

Outside of the processing that was done to derive the dataset for the forecasting models, certain variable transformations were done solely for the purpose of data exploration.

‘OrderDate’ and ‘JourneyStartDate’ are two features of every recorded ticket sale. An example of a variable transformation done in data exploration is subtracting ‘OrderDate’ from ‘JourneyStartDate’, to find a variable telling us how many days ahead of the start of the trip any ticket purchase was made.

After examining the properties of the ticket sales at the most granular level, single tickets, ticket sales were aggregated to derive the target variable, ‘Quantity’, referring to the total number of tickets sold with ‘JourneyStartDate’ equal to a given date. *Aggregation* is a form of *feature engineering*, which can be described as a process of ‘re-framing’ variables to make them more relevant to the problem at hand. Aggregating ‘Quantity’ on ‘JourneyStartDate’ resulted in a variable indicating not only when tickets are sold, but on which date the customers will be traveling on, specifically how many tickets in total are sold for each ‘JourneyStartDate’. The aggregation also drastically reduced the number of rows in the dataset, making it easier to handle, a positive side effect.

2.1.4 Variables

After having briefly looked at the data and performed the primary cleaning and pre-processing, it was time to explore the data in detail. This included looking at the data from different angles using a range of visualizations and measuring some relevant statistical metrics.

Below are displays of the variables values over time, along with the probability distributions for the *counts* of the variables, meaning the number of days where the variables were observed to have any given value, grouped in intervals typically described as ‘bins’. The figures were generated using the ‘generate profile’ functionality of Microsoft Azure. This section goes through the variables one by one, examining their properties and discussing their characteristics along with some preliminary assumptions about them.

Quantity variable

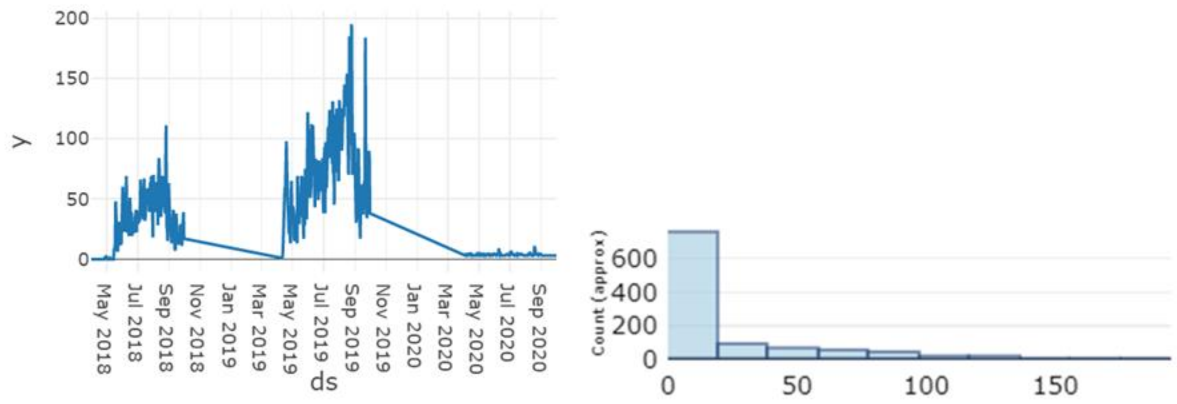


Figure 1: Quantity variable over time, and probability distribution

‘Quantity’, or the number of tickets sold with a given journey start-date, is the target variable to predict. By the look of the histogram above, the variable seems to have a probability distribution resembling the Poisson distribution, skewing strongly to the left. The Quantity count is distributed like this because from October to the start of April, Go Fjords do not operate any trips. When they do operate there are still days with few travellers, for instance on workdays outside of vacations, and on days with bad weather. Simultaneously, there is a ‘long tail’ of observed days with a high number of tickets sold, like on vacation days with good weather. It is quite common for count data to be Poisson distributed, or at least for their probabilities to resemble a Poisson distribution more closely than a Gaussian (normal) distribution.

A dispersion test was conducted to test the *goodness of fit* of the distribution of the counts of Quantity variable against the Poisson distribution, to determine to which degree the data resembles this distribution. The dispersion test judges goodness of fit to the Poisson distribution by evaluating whether the data has Poisson-like characteristics, such as its mean being equal to its variance, among other features (Cameron, 2019). The dispersion test could not say with any statistical significance that the variable was drawn from the Poisson distribution. Another potential test for evaluating goodness of fit is the ‘chi-squared test’. Such a test was not conducted, partly for time-constraint reason, and partly because it is not necessary to find a theoretical probability distribution that the data could have been drawn from with statistical significance. Real-life data seldom perfectly fits a theoretical framework. Dispersion test results can be found in the appendix.

The Quantity variable can only take the form of a discrete value, meaning only whole numbers. It must also be non-negative since negative sales do not have any intuitive meaning.

Temperature variable

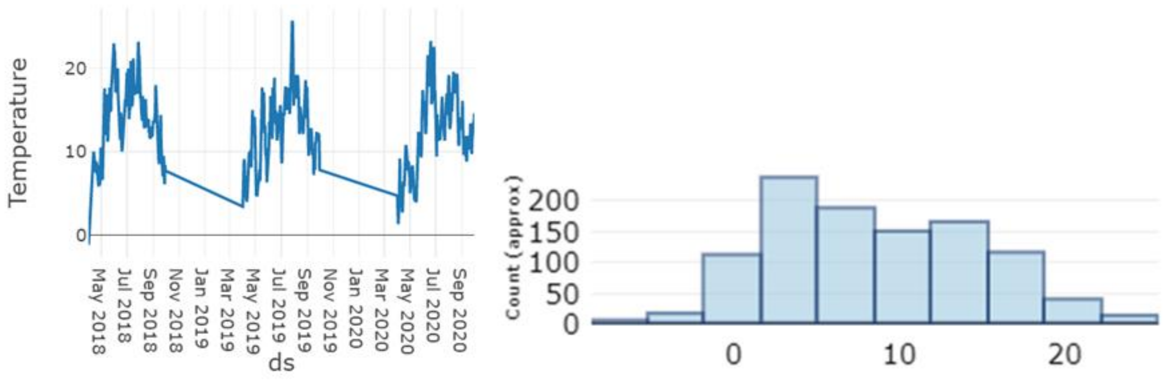


Figure 2: Temperature variable over time, and probability distribution

'Temperature' is the first of the three weather variables, which are all recorded every day at noon in Stavanger. The Temperature variable describes average temperature in degrees Celsius in a time interval of six hours, from 06am to noon. The variable has a probability distribution more closely resembling a normal distribution than a Poisson distribution.

A year in Norway typically sees large, predictable swings in temperature due to seasonal effects. One potential statistical issue to consider regarding the Temperature variable is its correlation to the time component. This can be regarded as a case of *multicollinearity*, which can be described as the occurrence of high intercorrelations among two or more independent variables in a multiple regression model. Multicollinearity is, everything else equal, undesirable, as it can cause less reliable statistical inferences (Hayes, 2020). In the extreme case where two variables are perfect covariates, meaning it is possible to deduce the value of one from the other with full certainty, then using both variables add nothing in terms of predictive strength, but certain popular performance metrics like, R-squared, will indicate that the model with more variables is better. Therefore, it is important to use critical judgement in deciding which variables to include, and tools like correlation plots and variable selection algorithms can be useful. However, this does not mean that one must remove a significant explanatory variable just because it correlates somewhat with another significant explanatory variable. If including both variables leads to better performance as opposed to excluding either of them, then keeping both is the best option.

Precipitation variable

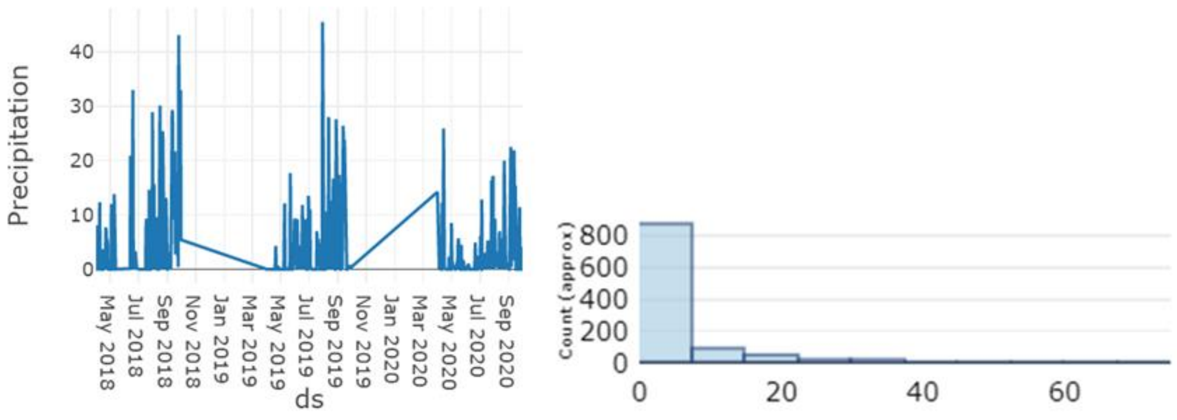


Figure 3: Precipitation variable over time, and probability distribution

'Precipitation' is a variable describing the total amount of rain- or snowfall, measured in millimetres, corresponding to litres per square metre. By examining the above histogram of counts of number of times when Precipitation-levels have fallen into different intervals, a probability distribution is derived resembling that of a Poisson distribution. Most days see little to no precipitation, while on a few days it is a lot. Like with the Quantity variable, a dispersion test was conducted, but the Precipitation variable could not either be classified as Poisson distributed with any notable certainty. Still, it is worth remembering that real-world data seldom conforms to an ideal statistical model, such as a theoretical probability distribution.

One way of transforming this variable to potentially increasing its predictive power, is to convert it into a binary variable (only taking the value of either 0 or 1), taking the value 0 if there was no precipitation that day, and 1 if there was any precipitation at all. The intuition behind this is that it is likely that many people prefer hiking when it is not raining. It might not matter very much how much it is raining, what matters might be whether it is raining at all or not. Keeping the Precipitation variable as a continuous value might just contribute noise to the model if it is true that differences between high values of Precipitation do not matter much. Still, the practice of coercing continuous data into integer format is not encouraged, as it may lead to information loss, often without yielding any benefit (Fedorov et.al., 2009).

WindSpeedMps variable

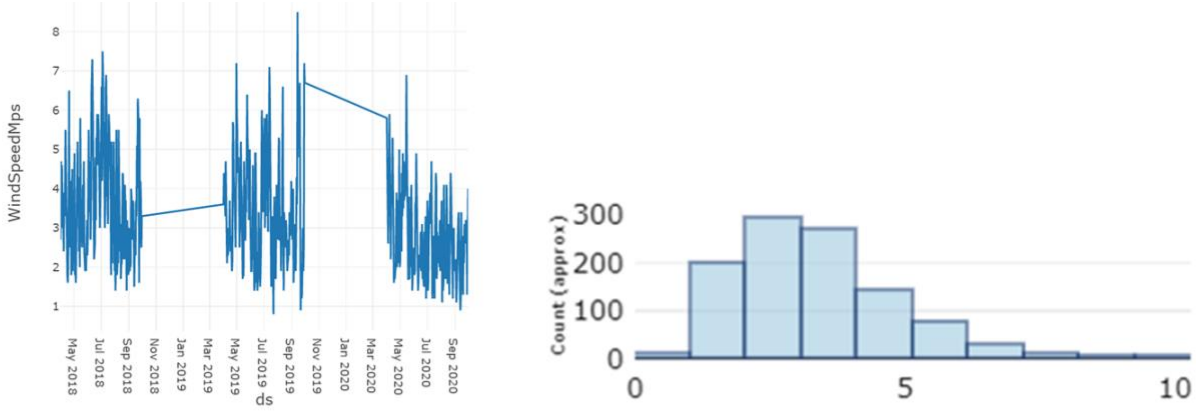


Figure 4: WindSpeedMps variable over time, and probability distribution

The final variable, 'WindSpeedMps', describes the average wind speed measured in meters per second, in an interval of six hours from 6 am to noon. Its distribution of count values loosely resembles a normal distribution, skewing slightly the left. Intuitively, one might think lower values of wind speed is most attractive for hiking.

2.1.5 Exploratory analysis

The initial exploration of the data was through Microsoft Power BI, a tool designed for visual inspection of data. Power BI can be described as a low-code tool, allowing the user to ask questions about the data using natural language, getting graphical representations in return. Power BI is a quick and easy way to get familiar with data so that further direction of the analysis can be established.

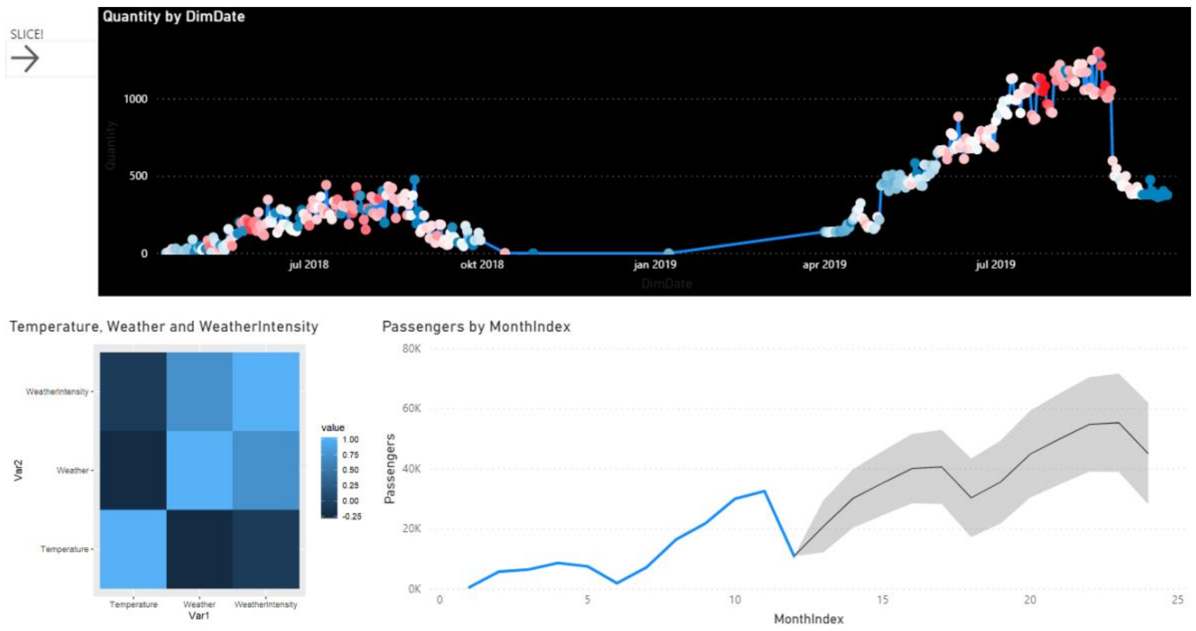


Figure 5: Initial Power BI data exploration

The above Power BI dashboard was created in the start of the exploration process before the start of the 2020 season and was used purely to get familiar with the data. It displays the Quantity by 'DimDate' (journey start-date), with colors indicating the Temperature variable. Notice that Quantity tends to be higher when the graph is red, where the Temperature is high, in line with the expected multicollinearity between Quantity and Temperature. The dashboard also presents a correlation matrix between a preliminary set of weather variables gathered from Bergen airport, and a forecast from the built-in forecasting functionality of Power BI, which proved to be less than perfect due to COVID-19 drastically shaping the possibilities for travel in the 2020 season. In hindsight, this preliminary forecast can serve as an example of the importance of having at least some built in learning mechanism to forecasting models, so they can adapt to drastic changes that will impact the target variable.

After having studied the data in Power BI, more in-depth insight was required. The relevant data was again exported to CSV made available for use in other applications like *R*, a statistical programming language well suited for data analysis. Through a large open-source community, R has many compatible packages making it efficient for gathering, cleaning, and pre-processing of data. Using the ‘*dplyr*’ package, the sales data was wrangled to yield new insights. This included aggregating by ‘journey destination’ and ‘journey start date’, allowing us to see how many tickets were sold for a given destination and a given journey start date.

To validate the belief that many customers order tickets on short notice, the distribution of how many days in advance of journey start date the tickets are sold needed to be evaluated. To create this measure, ‘OrderDate’ was subtracted from ‘JourneyStartDate’ for every ticket sale and binned the counts into intervals of one day. Below is a histogram of the measure.

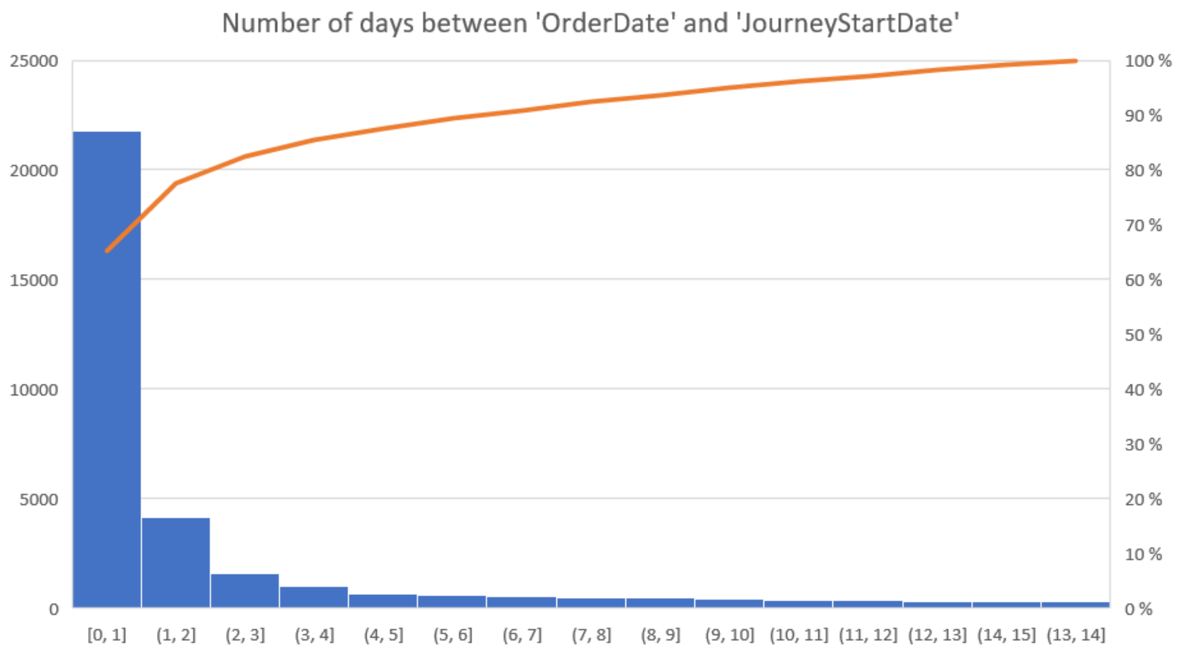


Figure 6: Distribution of time between ticket purchase date and journey start date

The histogram of this ‘days ahead’ metric seems to follow an exponentially decaying curve with a ‘long tail’, where most orders (ticket purchases) are done very close to the start of the trip, but tickets are also sold many days in advance. Ordering your ticket on the same day of the trip is the modal value (most occurrences) while ordering one day ahead is the median value.

Below is a more sophisticated view of the same measure, from the Power BI dashboard that Go Fjords have already implemented.

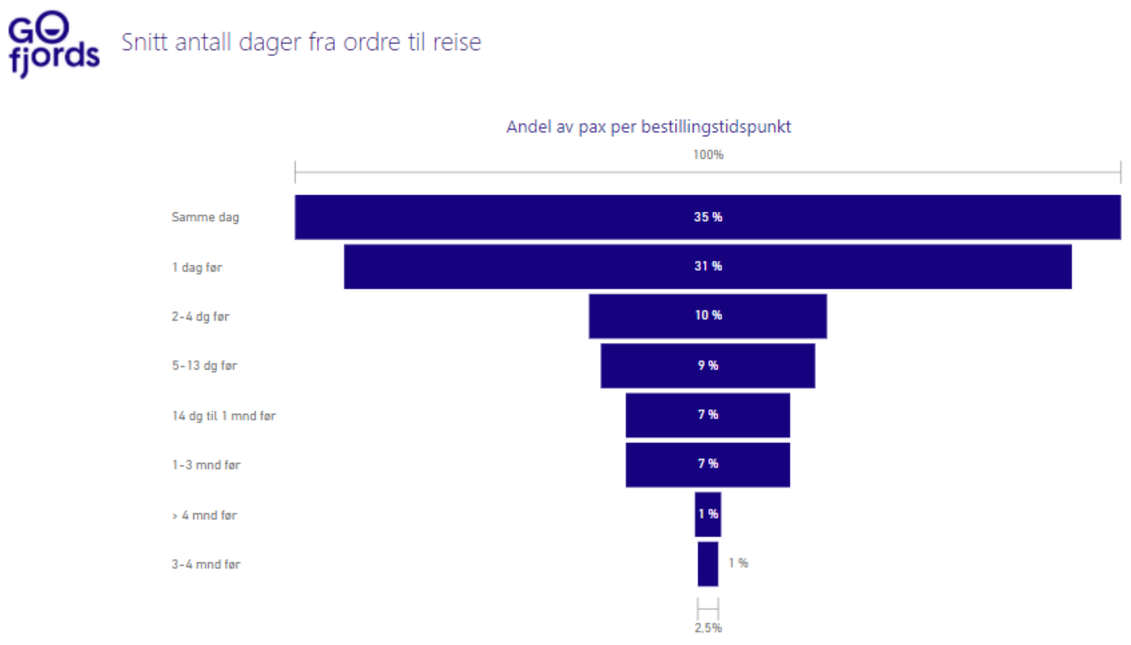


Figure 7: Percentage of tickets sold binned by order time

The displays confirm that most customers order tickets on short notice. It is not unreasonable to believe that many potential customers consider the weather forecast before ordering an outdoor trip, especially those that order close to the start of their trip, since weather forecasts with a short time horizon are more accurate that those with a longer time horizon. In the case that this is true, taking forecasted weather into account when predicting ticket sales would likely yield predictions of higher accuracy. This finding strengthened the belief in the potential predictive power of the weather variables.

Since the Quantity target variable is observed over time, rendering the data a time series, it was interesting to investigate how Quantity varies over time. Below are plots of three time series characteristics of the Quantity variable, with the plot names written vertically on the y-axis of the respective plots.

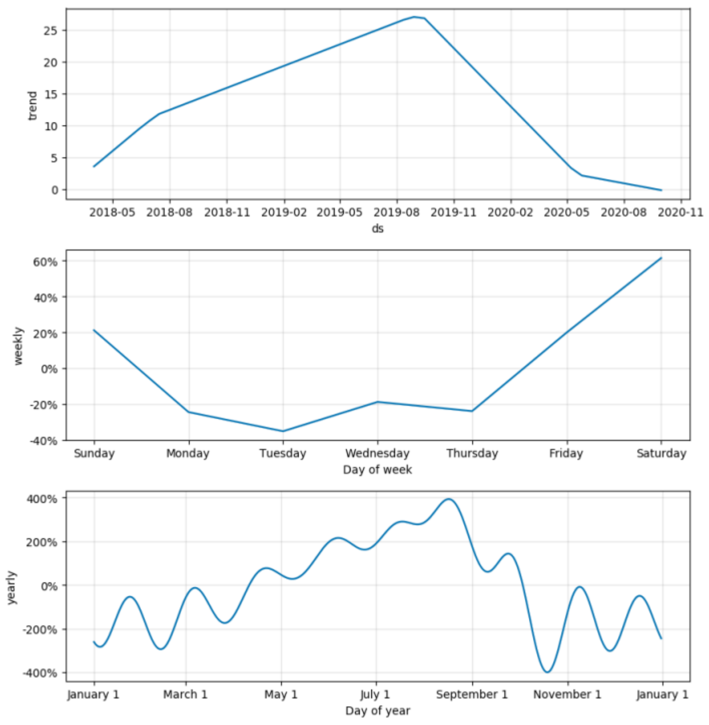
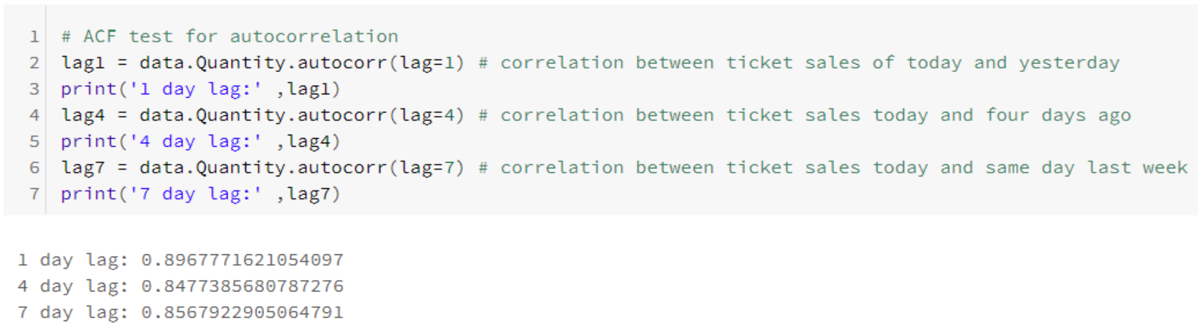


Figure 8: Trend, weekday effects, and seasonality

‘Trend’ describes a rolling average of tickets sales over time. Go Fjords was experiencing growth in their first two years of business, until COVID-19 largely prohibited both domestic and international travel in 2020.

‘Weekly’ describes weekday effects on sales, measures by the percentage difference between the weekday in question and the average of the other weekdays. There is a clear preference among Go Fjords customers for going on tours during the weekend. This might also be because Go Fjords are operating tours more frequently in the weekend, but the reason for having more tours on certain days is likely that these days are when customers want to travel. The customer is king.

‘Yearly’ reflects what can be called the *seasonality* affecting Go Fjords. Seasonality is variation in business or economic activity that takes place on a recurring basis. Seasonality may be caused by various factors, such as weather, vacation, and holidays. (Allbusiness.com, 2020).



```
# ACF test for autocorrelation
lag1 = data.Quantity.autocorr(lag=1) # correlation between ticket sales of today and yesterday
print('1 day lag:' ,lag1)
lag4 = data.Quantity.autocorr(lag=4) # correlation between ticket sales today and four days ago
print('4 day lag:' ,lag4)
lag7 = data.Quantity.autocorr(lag=7) # correlation between ticket sales today and same day last week
print('7 day lag:' ,lag7)
```

```
1 day lag: 0.8967771621054097
4 day lag: 0.8477385680787276
7 day lag: 0.8567922905064791
```

Figure 9: Autocorrelation measures

Above is a screenshot from the calculation of the Autocorrelation Function (ACF) between Quantity and its lags (past values) of three different intervals. The autocorrelation function defines how data points in a time series are related, on average, to the preceding data points (Box, Jenkins and Reinsel, 1994). Note how the autocorrelation is greater between Quantity and its 7-day lag than between Quantity and its 4-day lag, owing to the weekday effects where the day of the week influences peoples propensity to travel. In general, the high values of the ACF tests indicate that on any given day of business, number of travellers is likely to be quite similar to the number of travellers yesterday, and on the same day last week.

2.2 Model selection

Having explored the data, both sales- and weather relate, there was now a basis for finding appropriate models for the forecasting problem.

To forecast ticket sales, a wide range of models were considered. Models ranging in complexity from simple univariate regression and time series forecasting to deep neural networks, could, in theory, be used to predict the Quantity variable.

The following section presents the chosen ‘candidate’ models and ‘benchmark’ models to compare them to, explains the most important theory behind them, and discusses their benefits and drawbacks in general and for this scenario.

2.2.1 Candidate models

The candidate models are a set of regression algorithms, implemented by a specific set of frameworks. The models were considered at different stages of the process. Finding an appropriate model for a prediction problem is often not straightforward and knowing where to start looking can be challenging. The approach in this thesis was to start simple, and gradually consider model of higher complexity until performance seemed to plateau.

The natural place to start when looking to predict a continuous value, is *regression*. Regression is a statistical method used to determine the strength and character of the relationship between one dependent variable and a series of other variables (Investopedia, 2020).

Ordinary Least Squares (OLS) is the estimation method most used for regression, a true classic among statistical methods. Simple in its use with intuitive results, simple regression should therefore be the go-to model in a lot of applications. Specifically, when OLS is ‘BLUE’ (the Best Linear Unbiased Estimator), it should be sufficient for the task, according to the Gauss-Markov theorem.

In the case of this forecasting problem however, the relationship between ticket sales and some of the predictive variables is likely non-linear in nature. For example, the relationship between Quantity and Precipitation is likely not linear in form. Linear regression would in that case not be able to capture the intricacies in the variable relationships, leading to sub-par performance.

Generalized Linear Models (GLM):

Compared to simple linear regression models, *Generalized* Linear Models offer certain benefits that make them more practical for real-world problems.

“Generalized linear models (GLM) are conventionally taught as the primary method for analysis of count data, key components of their specification being a statement of how the mean response relates to a set of predictors and how the variance is assumed to vary as the mean varies.” (McCullagh and Nelder, 1989).

Seeing as Quantity is an instance of count data, a GLM is seemingly a good choice in this case. Furthermore, the fact that a GLM can specify how the target variable relates to the predictors, allows the model to be instructed to account for the probability distribution of the target variable, which here is assumed to resemble the Poisson distribution.

A GLM can generally be represented as follows:

$$g(E[y|\mathbf{X}]) = B_0 + B_1(x_1) + B_2(x_2)$$

g: generalized link function connecting target with predictors

E[y|**x**]: Expected target variable (y) given a set of predictors (**x** here represents the set [x_1, x_2]

y: target variable

B_i: Coefficients

x_i: predictor variables

The main component separating a GLM from simple linear regression is the link function *g*, which enables the model to account for non-normally distributed data. The way the link function models a probability distribution, is by applying a mathematical transformation to the outputs of the weighted sum of the predictive variables, in this case, applying the natural logarithmic function (log-lin).

An additional way a GLM can be enhanced, is by using using *Bayes' theorem* to make the model 'learn' as it is exposed to more data, making the forecasts more responsive to changes in underlying conditions affecting the response variable.

PyBats is a Python package made for 'Bayesian time series forecasting and decision analysis' (Cron and Lavine, 2020). At its core, PyBats is an enhanced GLM. PyBats makes the GLM more intelligent by introducing a *Dynamic* component, where the otherwise constant coefficients B_i are re-estimated through *Bayes law* as the model moves forward chronologically. Bayes law is a statistical result that allows for updating a belief about the likelihood of an event (such as different levels of the variables), through new observed evidence. This allows the algorithm to learn from new observations without having to be entirely retrained. For example, if the contents of a Go Fjords trip changes, or a new tourist demographic enters the market that has different preferences than the rest of the population, a DGLM can learn these new preferences or changes by adjusting its coefficient.

The name PyBats is an acronym, stemming from some of the techniques used by the algorithm. *Py* from being a Python package, *B* from using a 'Box-Cox' transformation to normalize the predictor variables' distributions, *a* from using ARMA (Autoregressive Moving Averages), *t* for accounting for trend, and *s* for accounting for seasonality.

Generalized Additive Model (GAM) – Facebook's 'Prophet':

Generalized Additive Models (GAM) are the slightly more flexible cousins of the GLMs. A GAM represents another way to enhance the concept of a GLM, this time to account for non-linear relationships between individual predictors and the target variable.

A GAM can be represented as follows:

$$g(E[y|\mathbf{X}]) = B_0 + f_1(x_1) + f_2(x_2)$$

g: generalized link function connecting target with predictors

$E[y|\mathbf{x}]$: Expected target variable (y) given a set of predictors (**x** here represents the set $[x_1, x_2]$

y: target variable

B_i: Coefficients

f_i: smoothing functions

x_i: predictor variables

The key difference between GAMs and GLMs lies in the smoothing functions f_i on the right-hand side of the equation, allowing for individual estimation of the functional form of the relationship between each respective predictor variable and the target variable. Having functions as opposed to simple values as coefficients allow GAMs to represent more complex relationships between target- and predictor variables.

The smoothing functions make GAMs more suited than GLMs to model situations where non-linearity applies, which is often the case in the real world. A relevant example could be the relation between rain and ticket sales. It is reasonable to believe the difference between 0 mm and 0.1 mm of rain makes a much larger impact willingness to purchase a ticket than the difference between 0.3 mm and 0.4 mm. Thus, the relation between rain and ticket sales is likely non-linear, and better accounted for by a GAM than a GLM.

The ability to account for non-linearity also makes GAMs highly flexible, allowing them to yield good predictions without any presumption about the form of the relation between a predictor- and the target variable. This benefit does not come for free though. The smoothing functions f_i need to be estimated from the data. In general, this requires many data points and is computationally intensive (Hastie and Tibshirani, 1990).

A contemporary, popular framework that uses Generalized Additive Models is the 'Prophet' package developed by Facebook (Taylor and Letham, 2017). Prophet fits non-linear trends together with yearly, weekly, and daily effects, as well as custom holiday effects. Despite merits like the ease of use and relative flexibility, Prophet has one drawback in this context. It does not allow for Poisson-distributed data terms, which you typically have in counts of discrete values, such as in the case of the count of ticket sales. Nevertheless, Prophet was tested for this task, as was deemed well-suited for demand forecasting.

The Prophet framework will be further discussed in the 'Deployment' chapter of this thesis, as the case study for technical implementation.

Wild-card: Microsoft Azure Automated Machine Learning

Being one of the leading providers of Machine Learning as a service in the cloud, Microsoft has highly sophisticated tools for machine learning in its Azure platform. Among these tools is Automated Machine Learning (AML), which streamlines the process of finding appropriate predictive models, requiring little to no domain expertise. Since Azure was already being used to host the data and models, there was no good reason not to try the AML feature, to see if the hand-picked DGLM and GAM were performing in the same ballpark as the best model from a state-of-the-art Automated Machine Learning solution.

Azure was simply given the data, along with the instruction of forecasting with a time-horizon of seven days. After churning the numbers and training and testing a long list of algorithms, Azure returned a long list of different model types, where the top contenders scored very evenly in terms of most performance metrics. Among the top contenders, only one model maintained *explainability*, that is, being able to rank the variables by predictive power, and allowing for intuitively answering *why* any given prediction was made. This model was a *Random Forest Regression* model.

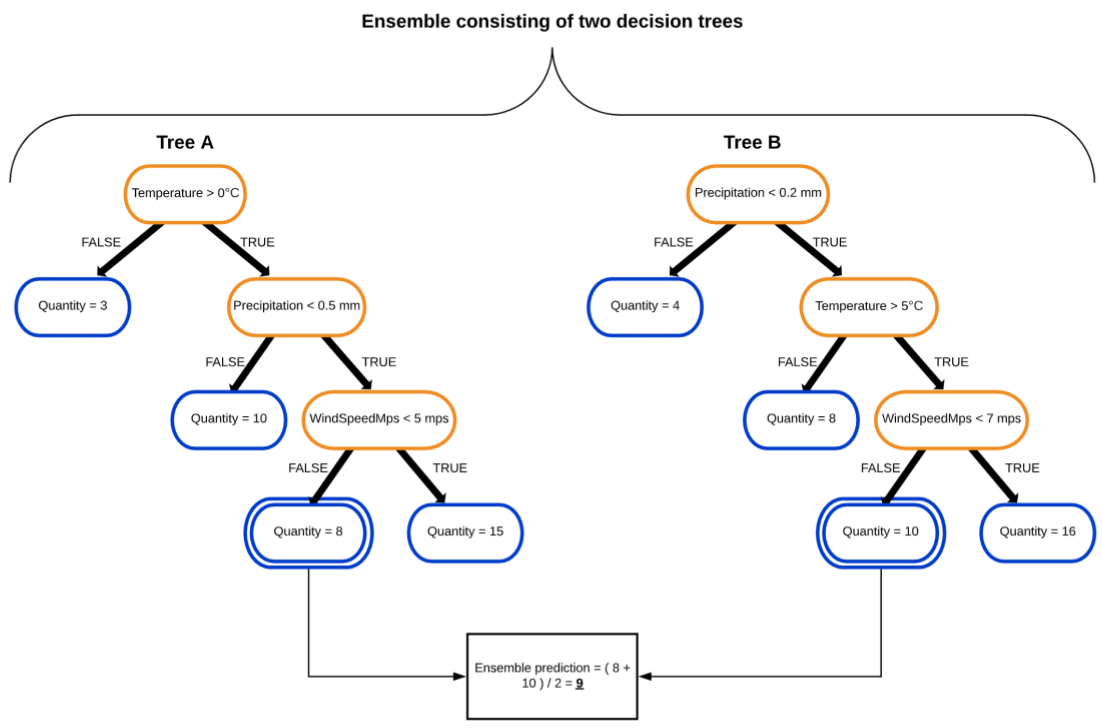


Figure 10: Random Forest Regression

In the above example, a simple ensemble consisting of two trees makes a prediction for the Quantity variable. The above ensemble would make this prediction for example with following set of values for the weather variables: [Temperature, Precipitation, WindSpeedMps] = [8, 0, 9].

Random Forest models are a type decision tree model. A Random Forest can be described as an ensemble model, since it takes the average prediction from several independent tree models, as its final prediction. To ensure that the trees in the ensemble are distinct enough, Random Forests use certain techniques like randomizing which variables appear in which order in the trees, limiting how far the trees can ‘grow’, meaning how many variables and nodes they can include, and randomizing which part of the data is used for training. A key benefit of the Random Forest model is that its ensemble nature reduces the variance of the predictions, leading to more stable predictions.

The Random Forest model that Azure presented used an additional technique called *MaxAbsScaler*, which is short for Max Absolute Scaler. MaxAbsScaler can be described as a form of feature engineering where the predictive variables are *normalized* so that the lowest observed value of each variable is set to 0, and the highest is set to 1, effectively scaling all the data into a fixed interval. The intention is to make the model more stable and to make sure that variable importance is not simply linked to the size of the numbers of the variables. For example, if it were decided that Precipitation should be measured in a unit smaller than millimetres, leading to higher numbers for every observation, this could impact the relative variable importance of certain algorithms. MaxAbsScaler prevents this. Scaling of variables is in general considered good practice in machine learning.

2.2.2 Benchmark models

In forecasting and machine learning, it is standard practice to compare model performance against the performance of certain simple and widely known models, to set a minimum target-to-beat, and to set the model performance into perspective relative to something familiar.

Naïve model

In time series forecasting, one model commonly used as a benchmark is the *naïve* model, which simply selects the most recent observation as its prediction.

Such a model might seem so simple that it is useless, but in data that is very hard to predict and that can resemble a random walk, such as the price of a single stock, a naïve model is sometimes among the best approaches. Recall that in the Quantity variable, the ACF metric with a one-day lag is approximately 0.88, meaning that the correlation between Quantity of today and Quantity of yesterday is very high. That means guessing the Quantity value of yesterday might not be very *naïve* after all. At least it can serve as a reasonable benchmark to beat when evaluating whether a more sophisticated approach has any merits.

ARIMA model

ARIMA stands for *Auto-Regressive Integrated Moving Average* and is a commonly used method in time series forecasting. Since they are adaptable to many different problems, while being easy to use, an ARIMA model will be included as a benchmark model to compare the heavier candidate models against, to see if their added complexity is worth it. There is no reason to "shoot sparrows with cannons", as the Norwegian saying goes.

ARIMA models estimate the target variable, Y, from a constant and/or a weighted sum of one or more recent values of Y and/or a weighted sum of one or more recent values of the errors (Nau, 2020). This means that they adjust their predictions based on the error of previous predictions, thus 'learning' in a sense. ARIMA models can be made simpler or more complex to suit the data, for example, to account for seasonality or to include predictive variables.

A good way to describe how an ARIMA model works is to explain each letter of the acronym:

Auto-regressive stems from the regression coefficients being estimated based on past values of Y, and alternatively also, past values of the predictive variables.

Integrated stems from the fact that ARIMA models require *stationarity*, meaning that the time series that is being predicted needs to have no trend, it needs to have a constant mean over time. To achieve this, ARIMA models construct a time series that depicts the *difference* in the target value Y value compared to the previous observation and creates predictions for this time series instead. In this case, that translates to the ARIMA model predicting *'how many more tickets will be sold for tomorrow, compared to yesterday?'*. Using this method, the time series will usually become stationary and suitable for ARIMA forecasting. In some rare cases, the differencing procedure needs to be done twice to produce a stationary time series, depending on the nature of the data.

Moving Average stems from ARIMA models adjusting their forecast based on the *error* from previous predictions, ensuring that the model has a self-correcting, learning component.

Our ARIMA model was created using the '*auto*.arima' function of the '*Forecast*' package in R. Auto.arima is a good place to start when considering ARIMA style models, because as the 'auto' part of its name suggests, the package does a lot of the work related to finding an appropriate model for the user. Since the variables used in this problem have quite pronounced seasonality, auto.arima added a seasonal component in its suggested model, technically rendering it a 'SARIMA' model, with S for seasonal.

2.2.3 Benefits and drawbacks of models

Table 2: Benefits and drawbacks of models

Model type	**Benefits**	**Drawbacks**
Naïve forecasting	- Simple to implement and interpret - Stable predictions - Good when autocorrelation is high	- Low flexibility - So simple that it does not allow for predictive variables or seasonality
ARIMA (benchmark)	- Easy to use, common ARIMA packages like *Forecast* in R adjust the model to account for inherent features of the data. - Can account for seasonality	- Can include predictive variables but is particularly suited to forecasting based only on past values of the target variable.
Generalized Linear Model (GLM)	- Simple and interpretable. - Can account for data that is not normally distributed. - Easy to incorporate predictive variables.	- Not suited to capture non-linear relationships between variables.
Generalized Additive Model (GAM)	- Can account for data that is not normally distributed. - Can handle non-linearity between variables, without prior knowledge of their relationship. - Easy to incorporate predictive variables.	- Does not implicitly account for autocorrelation. - Relies on assumptions about the data generating process. If these underlying assumptions are no

		longer true, the model loses its intuitiveness and explainability. (Christoph Molnar, 2020)
Random Forest Regression	- Ensemble model, meaning it takes average prediction of many tree models as its final prediction. Thus, more robust to overfitting. - Maintains the explainability and ability to assign variable importance, despite being complex.	- Not memory efficient, all the models that the ensemble consists of need to be stored. - Not necessarily the best at extrapolation and forecasting since the model by default does not consider trend.

2.3 Measuring performance

In finding an appropriate forecasting method fit for the data and goal, it is important to find the right metric to optimize for, and to test on representative data. This section presents and discusses potential performance metrics, explains which of them that were chosen to optimize for and reasoning behind this, as well as the chosen method for validating model performance.

2.3.1 Performance metrics

There are many ways one can measure the performance of a machine learning model. Since the data set consists of time series data and all the models can be described as relying on regression, the scope is narrowed somewhat. Hyndman & Athanasopoulos (2018) divide ways of evaluating model errors into the following categories, with some examples of prevalent metrics to minimize:

Table 3: Performance metrics

	Scale-dependent errors	**Percentage errors**	**Scaled errors**
Description	‘Errors are on the same scale as the data’	‘…unit-free, and so are frequently used to compare forecast performance between data sets.’	‘an alternative to using percentage errors… based on the training MAE’
Examples	MAE, RMSE	MAPE, sMAPE	MASE

(Hyndman & Athanasopoulos, 2018).

The latter two categories are appropriate for *panel data*, when comparing performance between different datasets where the scale of the variables can be different, so that the errors will still be comparable. It is not necessary to worry about scale of the variables in this case since the models are being optimized for the Preikestolen trip only, rendering the dataset an instance of time series data, not panel data. If Go Fjords, however, were to decide that they want one unified model that is optimized for several locations, the percentage- and scaled errors would be worth considering. This is because the scale of the variables could change. A high number of sales and a high temperature in one location and for one trip is not necessarily an equally high number when looking at another location and trip.

Thus, only the scale-dependent errors remain, where two of the most prominent metrics are the ‘Mean Average Error’ (MAE), and ‘Root Mean Squared Error’ (RMSE).

The way RMSE is calculated can be generalized as such:

$$RMSE = \sqrt{\frac{1}{N}\sum_{i=1}^{N}\left(Y_i - \hat{Y}_i\right)^2}$$

While the way MAE is calculated can be generalized as such:

$$MAE = \frac{1}{N}\sum_{i=1}^{N}|Y_i - \hat{Y}_i|$$

Legend for RMSE and MAE equations

N: number of samples

Y: observed value of target variable

Ŷ: estimated value of target variable

Since RMSE has a squared component, unlike MAE, it penalizes large errors between observation and prediction. This means that optimizing a model for minimizing RMSE, using RMSE as the *loss function*, will lead to a model leaning toward high bias, while on the other hand minimizing MAE leads to a model leaning toward a higher variance, comparatively. This decision regarding metrics to optimize for is a case of the *bias-variance tradeoff*, a common problem in machine learning, where lowered bias or variance often comes at the expense of an increase in the other metric. All else equal, both high variance and high bias are undesirable qualities, at least when their levels are higher than that of the actual data the model tries to predict. High variance is bad because it means predictions will be unstable, and large errors can occur. High bias is bad because it means the model is inflexible and will yield unwavering, often wrong predictions.

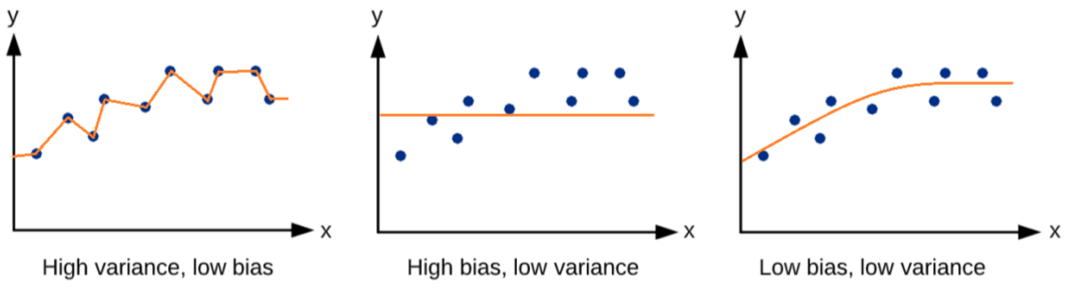


Figure 11: Bias-variance tradeoff

The charts above illustrate three models scoring differently in the bias-variance tradeoff on the same data. In the chart to the right is a model that strikes a good balance in the tradeoff.

It is worth noting that RMSE, as it penalizes large deviations between predicted- and observed values with its ‘squared’ component, discourages *overfitting*. Overfitting can be described as ‘an analysis which corresponds too closely or exactly to a particular set of data and may therefore fail to fit additional data or predict future observations reliably’ (OxfordDictionaries.com, 2020). The left-most chart in the illustration exemplifies an *overfit* model. When a model is overfit to the training data, everything might seem good on the surface because the chosen performance metrics indicate that the model is effective. Meanwhile, the model may have been *overly fit* to the training data to the extent that it adapted to patterns in the training data that might simply be caused by randomness or a small sample size. This can then lead to a high variance in predictions for unseen data since the new data may not display the same random patterns that the model overfit to in the training data.

The other end of the bias-variance tradeoff would be a completely biased model, like the one in the middle of the illustration, which gives a constant, unwavering prediction regardless of input parameters.

The primary purpose of creating a ticket sales forecast in this project is to be able to select a better number of buses to rent. The context has an implication for which metric is most appropriate to optimize for. The decision around how many buses to rent is one of *discrete optimization*. If the number of tickets sold is likely to be in the interval is between 0 and ‘n’, assuming ‘n’ is the passenger capacity of a bus, then the optimal number of buses to rent is one. This implies that it is important that the forecasts are *somewhat* accurate, and that large errors are avoided if possible, as they are more likely to result in advice that lead to renting a non-optimal number of buses. For example, if the model usually has an error within +/- 2 tickets, it will usually recommend the right number of buses for rental. If the model has frequent spikes of large errors (high variance), it will often recommend the wrong number of buses.

Due to RMSE having the benefit of discouraging overfitting when compared to MAE, it was decided to use RMSE as the performance metric by which to compare all the models.

2.3.2 Time series cross-validation

The method of time series cross validation can be described as a series of test sets, each consisting of a single observation, where the corresponding training set consists only of observations that occurred prior to the observation that forms the test set. Thus, no future observations can be used in constructing the forecast (Hyndman and Athanasopoulos, 2018).

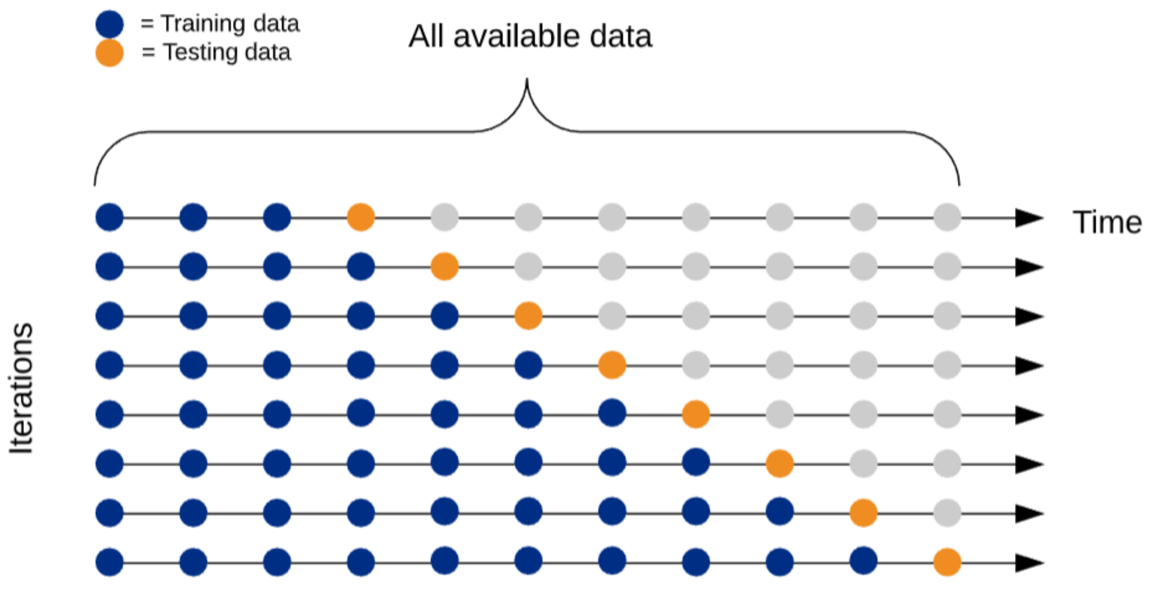


Figure 12: Time series cross-validation

Above is an illustration of the time series cross-validation method.

Since no future observations are used to train the model, testing it in this manner is akin to testing it in practice, and one can feel more certain that good performance is not simply caused by overfitting. The model is trained on past data, tested on future data, and moves forward chronologically to include more of the data in the training set, and even newer data in the test set. When all available data has been used, the average of the chosen metric from all the iterations of train-test splits is returned as the final, cross-validated performance metric, less prone to overfitting. Time series cross-validation is also sometimes referred to as ‘walk forward validation’, as it moves one time-step ahead for every new train-test split (Brownlee, 2016).

All models tested in this paper, both the candidate models and the benchmark models, were tested using time series cross-validation, and compared by RMSE.

3. Deployment

In the first round of evaluating the candidate models, the Generalized Additive Model regression option of Facebook Prophet was deemed to be the most promising technique, seemingly hitting the sweet spot of low complexity and high accuracy. At this point in time, The GLM candidate that was being considered was a simpler one than the PyBats DGLM, and the Random Forest Regression model had not been considered yet. Thus, it was decided that the demonstration of deployment should be done using the Prophet GAM. The Prophet forecasting model was deployed and yielded decent predictions. Later, as it became apparent that the Prophet model was not entirely fit for purpose, other models with tolerance for non-normally distributed variable values were explored. Nevertheless, this chapter demonstrates the implementation of the Prophet model, as an example of deploying an end-to-end Business Analytics solution on a cloud platform.

3.1 Choice of tools and implementation

The tools used in all parts of this project, from data exploration to conveying insights, were chosen primarily to fit the needs of Go Fjords, and to be compatible with the solutions that TietoEVRY are already using. This included the choice of Microsoft Azure as the cloud platform on which to run the data pipeline and forecasting models, as well as Microsoft Power BI as the tool through which to convey the results and insights.

For a large part of the time this thesis was worked on, the Facebook Prophet model was believed to be the most suitable model. This was based on broad research which found that other analysts are having success in demand forecasting using Prophet in recent years. Being a Generalized Additive Model, Prophet is in theory highly flexible and suited for a wide range of use cases, so it was assumed that it would perform well in the Go Fjords case. It yielded better accuracy than the simple naïve benchmark model, but not by a lot. This was attributed largely to the most recent year, the 2020 season, being an outlier, and thus hard to predict. And so, it was decided quite early that the Prophet model was good, so there would be adequate time for the demonstration of implementation of the model in the cloud, which is a part of the goal of this thesis.

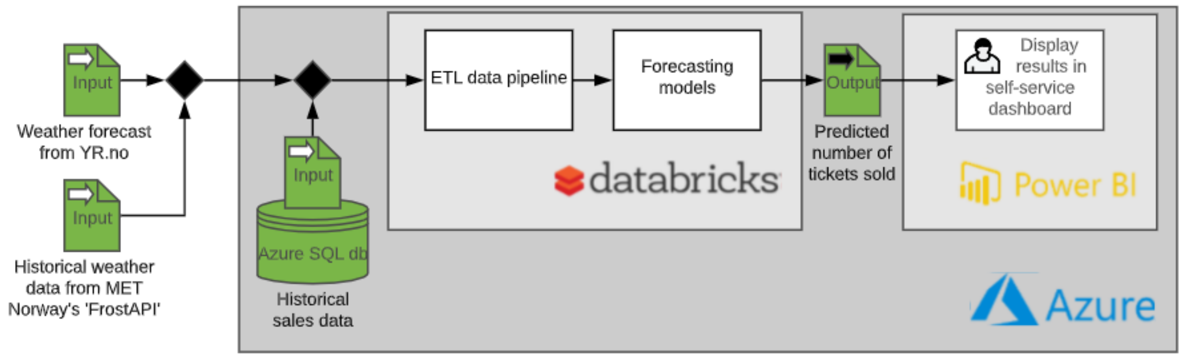

Input
Weather forecast from YR.no
Input
Historical weather data from MET Norway's 'FrostAPI'
Input
Azure SQL db
Historical sales data
ETL data pipeline
Forecasting models
databricks
Output
Predicted number of tickets sold
Display results in self-service dashboard
Power BI
Azure

Figure 13: Data model overview

Above is an illustration of the full deployed data model. The scripts fetching and preparing data, as well as the Prophet model, was implemented in a Databricks notebook running as a daily 'job' at a set time every day. The Databricks notebook was hosted in Azure, the Microsoft solution for cloud computing- and storage. The data pipeline roughly consists of the following sequence:

1. **Gather weather data from external sources**

   a. **'Web scrape' weather forecast for one week ahead from yr.no.**

      The way this script scraped the webpage, was by entering the source page of the yr.no page for the given location (Stavanger), which contains the data in XML format. The script then uses primarily the 'requests' and 'beautifulsoup' packages to retrieve the desired information. The script created for retrieving the weather forecast from yr.no can be found here in the appendix.

   b. **Get historical observed weather data by API-call from the 'Frost API' of MET Norway.**

      Done by adapting a template script provided in the documentation pages of the Frost API, available in both R and Python. Other than specifying which data one wants, the only input need is a user identification token, so that the API can recognize whether certain actors are using the service for unwanted purposes.

2. **Get historical sales data by SQL query from Go Fjords' Azure SQL database.**

   Done simply by SQL 'select' statement, selecting all ticket sales with 'TripID' equal to the internal ID corresponding to the 'Preikestolen tur-retur' trip.

3. **Perform data assembly process in Databricks notebook, cleaning, pre-processing, and transforming the data to ensure it fits the schema required by the Prophet model.**

First assembling the weather data (historical- and forecasted), as they are retrieved from separate sources. Included turning the historical API data from ‘long’ to ‘short’ format (many rows, to one row per date, one column per variable). Code for retrieving the data from the Frost API can be downloaded on the documentation web page: https://frost.met.no/examples2.html (MET Norway, 2020). Code to transform the output to ‘wide’ format typically preferred as input for predictive models can be found here in the appendix.

4. **Use forecasting model, here Facebook Prophet, to forecast ticket sales one week forward in time, based on the weather.**

   The Prophet framework is available in several programming languages and was here implemented using Python. The script that was used as a starting point for the modelling can be found in the following article: https://databricks.com/blog/2020/01/27/time-series-forecasting-prophet-spark.html (Obeidat, Smith and Heintz, 2020). Notable inputs to the model were, for example, instructing it to forecast one week ahead in time, telling it to use the weather variables as predictors, and restricting its forecasts to be non-negative. The reason the forecasts needed to be ‘clipped’ to non-negative values will be discussed further in the ‘Results’ and ‘Discussion’ chapters.

5. **Export the output of the model to Go Fjords’ Azure SQL database.**

   The final forecast is converted from a Spark ‘dataframe’ object especially suited for distributed computing, back to a single ‘*Pandas’* dataframe, then to an SQL table, and exported to Go Fjords’ Azure SQL database.

6. **Display the forecast in Microsoft Power BI dashboard, putting it into context of how many buses to rent, to facilitate better decision making.**

   The Microsoft Power BI dashboard is already connected to Go Fjords’ Azure SQL database, rendering the forecast available in the dashboards. Any illustration including the forecasts can be refreshed, updating the graphics to include the newest data. An example of a dashboard illustrating how many buses should be rented for different times of the season, can be found in the ‘Discussion’ chapter.

To build and automate this value chain, a range of programming languages were used. This is a key benefit of Databricks notebooks, they allow for multiple programming languages to be used, namely R, SQL, Python, and Scala. All of these were used in the work on this thesis. An alternative to running one Databricks notebook could have been to use one programming language per notebook, but that would have required more saving, converting, exporting, and importing of data between more notebooks, increasing complexity and risk of errors. The programming languages were used for the following purposes, in roughly the following order:

The *R* programming language was used early on for data wrangling and -exploration. R is a general-purpose programming language, with powerful extensions for data analytics and statistics, and for data manipulation. Using the '*dplyr*' package, the data was processed at scale. R was then used to plot the data, and to display the correlation between the variables.

SQL is a database-oriented programming language, used to retrieve, write, or wrangle data from databases. It was used to extract historical sales data from Go Fjords' database, and to later store model results in the database. It was also used to make the data available for display in Power BI.

Python is another general-purpose programming language, with many similarities to R. It is arguably even more general in its capabilities, and is suitable for building applications, as well as for powerful data analysis. Whether to use Python or R naturally depends on the use-case and needs, but in many cases, including this one, Python is more lightweight, runs faster, and its data structures scale better memory-wise. After the exploration was done, the data pipeline was written in Python, including web scraping, API-calls, data pre-processing, and the training- and usage of the candidate forecasting models.

Scala was used for certain technical purposes, including the configuration of the Databricks *cluster* that the data pipeline runs in. A cluster is a group of computing resources that can be expanded if there is a need for more processing power. The distributed computing done by the cluster was handled using the *Apache Spark* framework. Using distributed computing makes training of the forecasting models quicker, as this is a computationally heavy activity (depending on the model, the GAM and Random Forest Regression, for example, took a while to train, while the DGLM was quick). It also speeds up inference, meaning using the models daily to improve predictions. The Scala programming language is in some ways similar to Python. The primary difference is that Scala is *statically typed*, while Python is *dynamically typed*. This makes Scala less prone to bugs, and easier to refactor (changing code while maintaining functionality).

A Minimum Viable Product (MVP) of the data pipeline with Facebook Prophet was ready for deployment early in the work on this thesis, ready to be deployed as soon as the 2020 travel restrictions would be removed.

3.2 Delayed start of the season

The intended start of Go Fjords' 2020 season was 1st of April. However, due to the COVID-19 pandemic, the Norwegian government announced a nationwide lockdown starting 12th of March. This forced Go Fjords to postpone the start of their season.

The first Preikestolen trip of the season was carried out in July when the first social lockdown rules had been loosened. Only certain trips in Go Fjords' portfolio were being operated, while others had to remain closed. The best-seller trip "Preikestolen tur-retur", was one of the few trips being operated, though only on certain weekdays, until the end of September. Thus, the Preikestolen trip became the natural case study for this thesis.

The Facebook Prophet model was able to run in production and record its predictions for the entirety of September 2020. This was a goal of this thesis, to learn how to deploy a forecasting solution in production, to learn how to apply Business Analytics in practice. However, the month of September 2020 proved to be a less than exiting month for Norwegian tourism. Ticket sales were so low that results were trivial when looking at September in isolation. Therefore, it was decided that even though results derived from live testing are ideal, it is necessary to back-test all the candidate models against historical sales data and evaluate or models based on that instead. Since the back-testing was done using time-series cross validation, the models still make predictions based only on data available before the point of forecast, making the results equally as valid as if testing was done live in production.

4. Results

In this chapter, key results will be presented by visualizing the forecasts of the deployed Prophet GAM model against the actual observed Quantity, and the accuracy of the forecasts will be compared against the accuracy of the other candidate- and benchmark models, by Root Mean Squared Error.

4.1.1 Performance of models

The deployed Prophet GAM model produced the following forecast (*yhat*, in orange), compared to the actual observed Quantity (*y*, in blue):

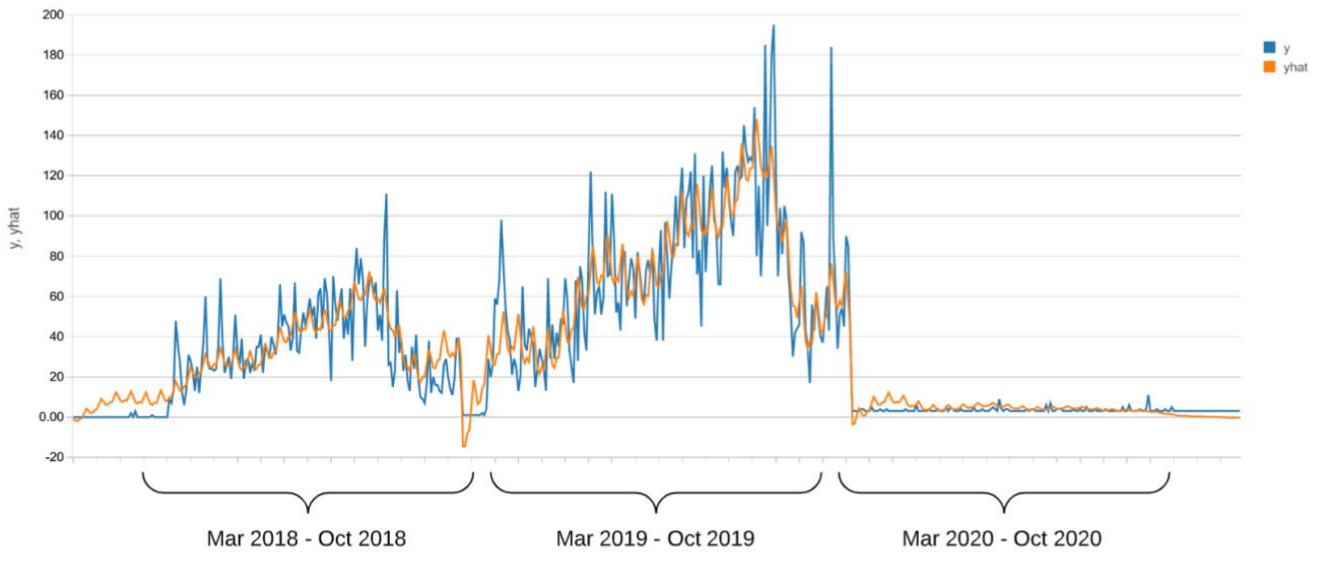


Figure 14: Facebook Prophet forecast

Observe how the forecast (the orange line) has regular, small spikes to account for weekday effects, where more tickets are being sold for weekend trips.

Notice also how the forecast line has a lower variance that the observed Quantity, meaning the forecast has the desired trait of low variance, in exchange for some bias.

A third thing to notice is how the model learns from the data, staying closer to the actual Quantity in the second season than in the first.

4.1.2 Performance compared to benchmark models

Knowing the DGLM beat the GAM is interesting, but to put its performance properly in perspective it is necessary to measure against standard benchmark models.

The performance of the benchmark naïve- and ARIMA models, and the candidate models (DGLM, GAME, and Random Forest Regression), can be summarized by Root Mean Squared Error (MSE) as such:

Table 4: Comparing models to benchmarks by RMSE with 7-day forecast

	Naïve model	**SARIMA model**	**Prophet GAM**	**PyBats DGLM**	**Random Forest Regression**
RMSE	17.16	12.68	16.53	2.21	11.65

As can be seen in the table above, the PyBats DGLM beat all the other models by a large margin, including the best explainable model from Azure Automated ML. This is an interesting result, even more so if one considers that it was quicker to train than for example the Prophet GAM model.

The DGLM model performs drastically better than the Prophet GAM, having an RMSE of 2.21 as opposed to the GAM, which has an RMSE of 9.65 for its shortest forecast horizon, and an RMSE of 16.53 in its seven-day forecast.

It is worth noting that the DGLM's RMSE of 2.21 is so low that it might be the result of overfitting, or some other issue. Predicting ticket sales one week ahead of time with +/- 2 tickets of accuracy on average seems like it might be too good to be true. Still, time series cross validation was used in obtaining the RMSE, which reduces the risk of overfitting.

4.1.3 Further optimization of DGLM

The DGLM model performed very well but could be optimized further to minimize RMSE. With its current configuration, it selects the median of its probability density function as its official prediction, which is ideal for minimizing Mean Average Error. By selecting the mean as the prediction RMSE would be minimized. This was not deemed necessary, however, as the model already scores very well. Here it was decided to follow the default method of the PyBats package, choosing the median of the estimated probability density function as the models point prediction.

Another way to potentially improve on the usefulness of the DGLM model is to use *regularization* to lessen overfitting. Regularization is the practice of adding a term that penalizes complexity. When applied, the model scores worse when it has more variables and more complex relationships among them. This leads to the optimal model being one that is less overfit to the training data, and thus hopefully more data-agnostic and effective on new, unseen data.

In the visualization below are three different perspectives of the full forecast made by the DGLM model, a blue line representing the forecast and black dots representing the later observed values. The largest graph spans seasons 2018-2020, the middle one 2019-2020, and the smallest one only displays the 2020 season.

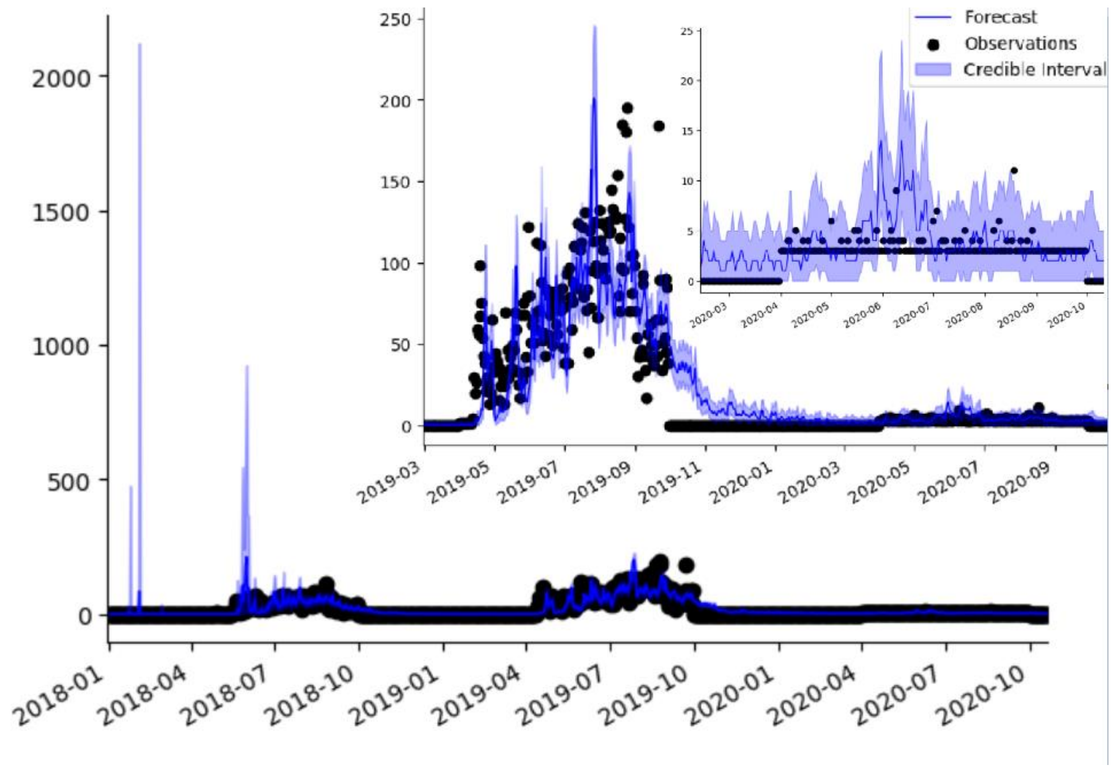


Figure 15: PyBats DGLM forecast

Notice how in the very beginning of 2018, the forecast had huge spikes. This is likely because of the Bayesian learning the algorithm deploys. At that time, the algorithm had not yet learned the properties of the data, and a slight uptrend in Quantity led to a parabolic response in the predictions. As time went by and the coefficients were adjusted, the predictions become more and more stable and accurate. In this way, the DGLM learns from the data and adapts itself to changes continuously.

One potential way the 'on paper' performance of the model could have been slightly improved, is to by trial and error tune the *hyperparameter* of *prior belief*. Hyperparameters are model settings that are typically set before the model is trained on data. There are some algorithmic ways to tune hyperparameters (like *grid-search* or *Bayesian optimization*), but an intuitive approach by a human is often the best approach. PyBats, like any model using Bayesian statistics, depends on the model learning to have 'beliefs' about the data and updating these beliefs as it is exposed to more data. Bayesian models, in a way, care about your opinion, because they require a prior belief from which they can start to make predictions, and then update that belief as time goes by. One reason the early forecasts spiked so high, may be that the prior belief was too high. If it was adjusted down, this might lead to better performance, especially in the first season. However, this would not matter much in the full three years of forecasting, as the model quickly updated itself and learned a more appropriate belief about the level and trend of the Quantity variable.

4.1.4 Why did the Facebook Prophet GAM underperform?

These results raise a question: Why did the Prophet GAM perform so much worse than expected, barely beating out the naïve model in terms of RMSE?

It may be argued that Generalized Additive Models like Prophet are more sophisticated and flexible than Generalized Linear Models since they allow for non-linear effects between predictors and the target variable. Thus, the early hypothesis in this thesis was that the Prophet framework would yield better forecasts than the GLM.

Here are two potential explanations as to why the DGLM outperformed the GAM in this case:

1) Facebook Prophet is as of the writing of this thesis not adapted to work with probability distributions other than the Gaussian distribution, commonly known as the normal distribution. The distribution of the target variable, Quantity, more closely resembles a Poisson distribution. This may cause the model to underperform, and even produce negative forecasts, which needed to be ‘clipped’ to non-negative values. The issue is known in the community of developers working on Facebook Prophet, and the framework will likely soon support non-Gaussian probability distributions. In the meantime, it could be worthwhile testing other GAM frameworks, as GAMs that support Poisson-distributed variables exist.

2) Our GLM is not a traditional variant, it is enhanced with a *dynamic* component, rendering it a DGLM. This dynamic component stems from the introduction of priors, which are adapted through Bayes’ law as the model ‘learns’ trends and relations between variables. Thus, the Prophet model is not necessarily the more complicated model. Either way, it is not a given that any of them should outperform the other.

4.1.5 Usefulness of predictive variables

Some of the packages used to create forecasts yielded auxiliary information regarding the predictive power of the three weather variables. Among these were the aforementioned ‘Explanations’ section of the Random Forest Regression model from Azure AML. It offers a range of visualizations for variable importance, the simplest and most intuitive of which is this one:

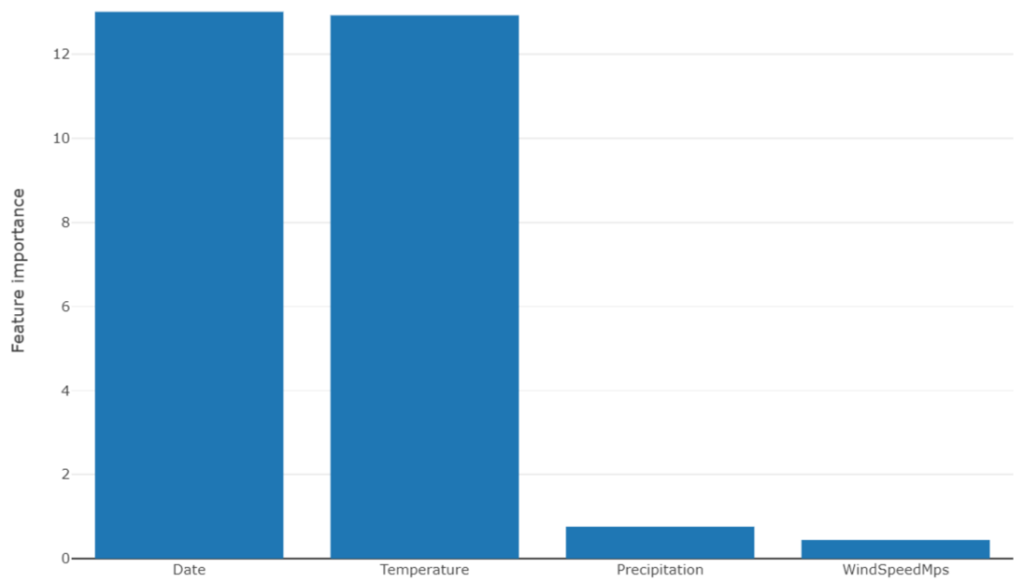
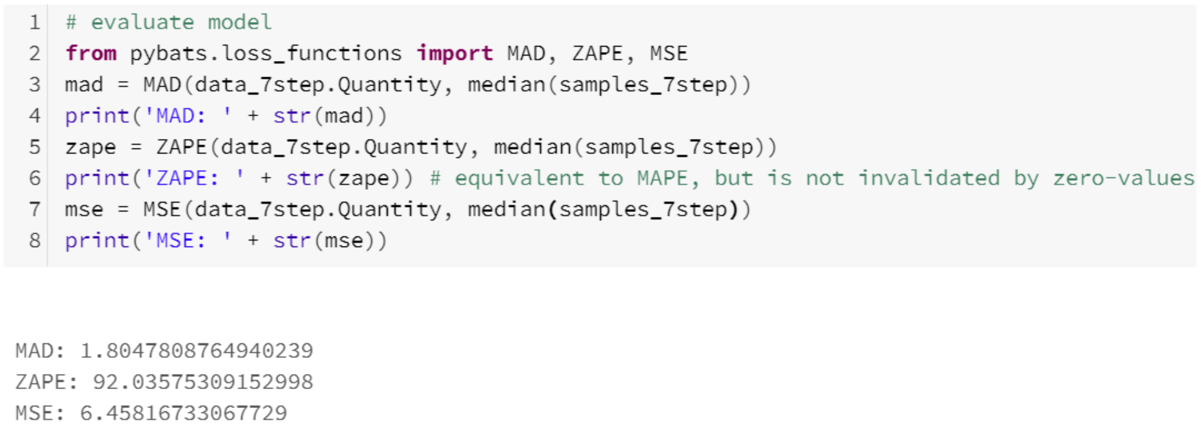


Figure 16: Variable importance in Azure AML's Random Forest model

As one can see, Date and Temperature are by far the two most important factors for predicting ticket sales. This is in line with initial expectations. It is interesting to note that Precipitation is almost twice as important as WindSpeedMps, though neither of these variables seem to be very important in this analysis. One thing to consider that lessens the belief in the variable importance metrics from Azure, is that the strong correlation between Date (time of the year) and Temperature might lead to a lot of the explanatory power in the time dimension (accounting for seasonality), might be attributed falsely to the Temperature variable.

A bit of further tinkering was done with the PyBats DGLM since it was the best performing model. Retraining the model with Temperature as the only weather-related explanatory variable, not Precipitation and WindSpeedMps, yielded the following performance:



```python
# evaluate model
from pybats.loss_functions import MAD, ZAPE, MSE
mad = MAD(data_7step.Quantity, median(samples_7step))
print('MAD: ' + str(mad))
zape = ZAPE(data_7step.Quantity, median(samples_7step))
print('ZAPE: ' + str(zape)) # equivalent to MAPE, but is not invalidated by zero-values
mse = MSE(data_7step.Quantity, median(samples_7step))
print('MSE: ' + str(mse))
```

```
MAD: 1.8047808764940239
ZAPE: 92.03575309152998
MSE: 6.45816733067729
```

Figure 17: PyBats' DGLM with only Temperature as predictive variable

Recall how the full model that includes all weather variables resulted in a RMSE of 2.21, which is quite a bit better than the model without ‘Precipitation’ and ‘WindSpeedMps’, which yields RMSE of 2.54. RMSE, unlike for example the popular R-squared metric, does not suffer from false increases in performance due to added predictive variables. Thus, since RMSE is considerably lower in the model including all variables, the full set of weather variables should be included in the forecasting model, even though the Azure AML function suggests ‘Date’ and ‘Temperature’ is doing most of the work.

5. Discussion

This section aims at discussing the results and reflecting on how business value can be derived from having increased forecasting accuracy, aided by an example. Then, certain assumptions and limitations of the study will be presented and discussed, with some potential points of improvement for anyone who might want to pursue similar challenges.

5.1 Business value of demand forecasts

Through accurate demand forecasts, it is possible for Go Fjords to prepare for accommodating anticipated increases in passengers, or cut unnecessary expenditures when demand is anticipated to decrease. This can lead to double upsides:

1) The firm can seize more revenue when the demand is high and avoid the loss of revenue and unhappy customers that can result from not having enough capacity.

2) Achieve cost-savings when demand is low by not having idle capacity.

These goals are in line with the *Lean* methodology, which aims to increase business value by reducing wasteful capacity.

In the case of Go Fjords, the most immediate action that can be taken upon having a more accurate forecast is to in the short term adjust the number of buses, boats, or possibly tour guides to deploy in the near future. For the purposes of this thesis, the number of buses to rent will be used as the example to show how the forecast can be used as input to aid in the goal of optimizing capacity.

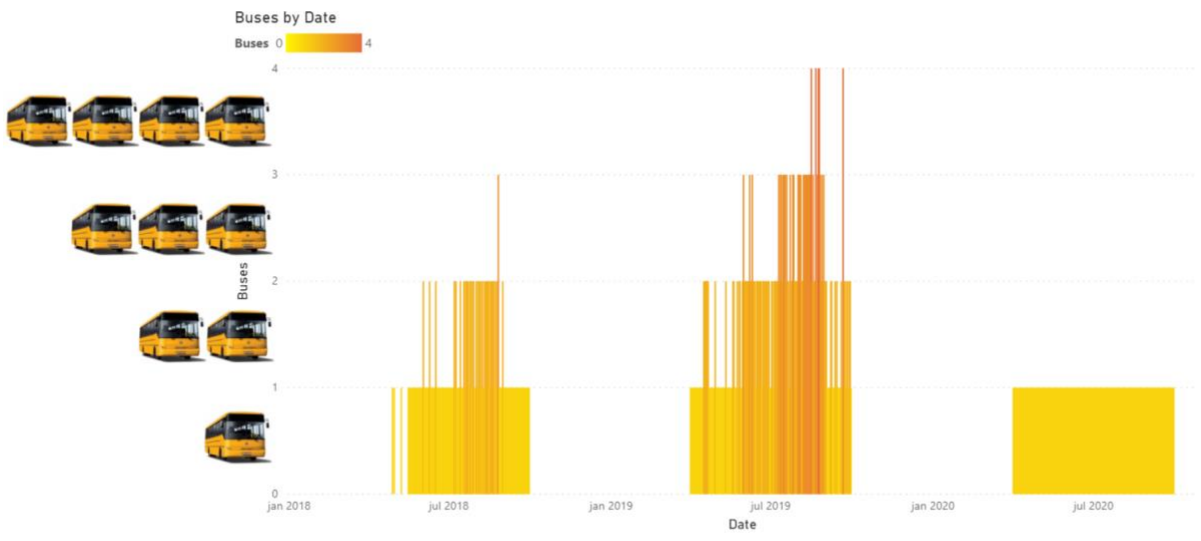


Figure 18: Optimal number of buses dashboard

From Power BI dashboards, Go Fjords can get decision support for adjusting their capacity, like illustrated in the dashboard above. The chart was made by binning the predicted number of tickets sold into intervals of 50, the approximate capacity of a typical tour bus. As the dashboard shows, the year of 2020 never required more than one tour bus to be operated for the Preikestolen trip. In the previous seasons, however, it would be profitable to rent a second, third, and even fourth bus in some periods.

Imagine an example where the company is using forecasts based solely on past sales. The forecast for the coming week is 110 travellers each day. However, the weather report says the weather will be unpleasant in the coming week, with temperatures dropping and lots of rain. The company has in advance ordered three buses to accommodate the 110 passengers, but only around 70 tickets are sold for each day. One bus is left idle the entire week.

If a forecasting algorithm considering the weather forecast was used instead, it could have predicted something closer to the true value of 70 travellers per day, say, 88 passengers a day. In this case, the idle bus would not have been rented, resulting in thousands of Norwegian kroner in cost-savings.

5.2 Assumptions, limitations, and possible improvements

5.2.1 Limited amount of data

At the start of this project- and thesis, Go Fjords had only been in business for two seasons. This meant that they had a limited amount of data to share. Two seasons worth of data is the threshold for certain popular forecasting frameworks, such as the ‘Forecast’ package available in the R programming language, to be able to infer seasonal effects.

Having only three seasons worth of data, the third of which was undoubtedly non-representative as it was impacted by the nationwide COVID-19 lockdown, yielded an uncertain picture. Had more years of data been available, one could feel more certain that the available data gave a representative view of the seasonal patterns Go Fjords are subject to. In addition to more certainty that patterns were representative, the sheer increase in data for model training would have enabled the usage of more sophisticated algorithms, for example, Long Short-Term Memory (LSTM) neural networks. LSTMs are powerful in part because of their ability to remember important events far back in time, while simultaneously placing most emphasis on recent events. Still, such a complex model would likely not perform significantly better than the simpler ones, like the DGLM, since there are quite few variables and little data. Due to the relative simplicity of the problem, the principle of “Occam’s Razor” urges us to choose the simpler solution if they seem equally effective.

Ending the study in an outlier-season can be said to have some benefits, to look on the bright side. It allowed for both the opportunity to look for models that perform well in conditions of stable growth (2018 - 2019), as well as those who perform well in unpredictable conditions like in 2020. The Dynamic GLM might have gotten its edge from its dynamic component, being able to retune itself continuously through Bayes’ law to adapt to the unpredictability of the last of the three seasons.

5.2.2 Limitations of Facebook Prophet

Facebook Prophet is a relatively new and powerful framework, but it turns out it has its limitations, at least at the time of this paper (Motoharu, 2020).

1) *Prophet does not allow non-Gaussian noise distribution (at the moment)*

2) *Prophet does not take autocorrelation on residual into account*

3) *Prophet does not assume stochastic trend*

The first of these limitations is likely the ‘Achilles heel’ of Prophet for the purposes of this thesis. Daily ticket sale is a count value, with a probability density function that more closely resembles the Poisson distribution than the Gaussian distribution.

5.2.3 Forecasted weather is not always *actual* weather

The models are trained on actual *observed* weather data, while forecasts are made based on *forecasted* weather. This means that the models take for granted that the weather forecast gathered from yr.no will be correct without failure, which is not the case. Weather forecasts are, despite people joking about them being unreliable, impressively accurate most of the time, especially on short notice. The weather is a chaotic system and is hard to forecast with high accuracy, especially far ahead in time.

Nevertheless, potential customers do not know the weather better than yr.no does in advance and are likely to base their purchase decision on the forecasts. Thus, it could be an idea to rather train the models on *past predicted weather* from yr.no, as the weather forecast for the travel date made on the date of purchase might have a more causal effect on purchase decision, than actual weather on the travel date does. However, this proves difficult in practice, as even yr.no themselves do not store (or at least make publicly available) their past predictions. This is likely because this would entail storing large volumes of data (predictions for many places, many forecast horizons, being updated often), and the information also having questionable societal- and business value compared to the big data storage cost. To achieve this, Go Fjords would have to store historical records of the data retrieved from the ‘web scraper’, and use this for model training. This is feasible, as the size of the data would not have to be larger than the observed historical weather data that is now being stored.

For these reasons, the discrepancy between the observed weather data used for training, and the speculative forecasted weather used for the demand forecasting, will simply be accepted. No one knows tomorrows weather for certain, not even MET Norway.

5.2.4 More probability distributions could have been tested

From visual inspection of the histograms of the Quantity- and Precipitation variables, the data seems like it would fit a Poisson distribution *more* than it would fit a Gaussian distribution. Meanwhile, that does not rule out the possibility of other probability distributions fitting the data even better than the Poisson distribution did.

Towards the end of writing this thesis, it became apparent that the presumed Poisson-distributed variables perhaps more closely resemble a '*Tweedie Distribution'* (cran.r-project.org, 2020).

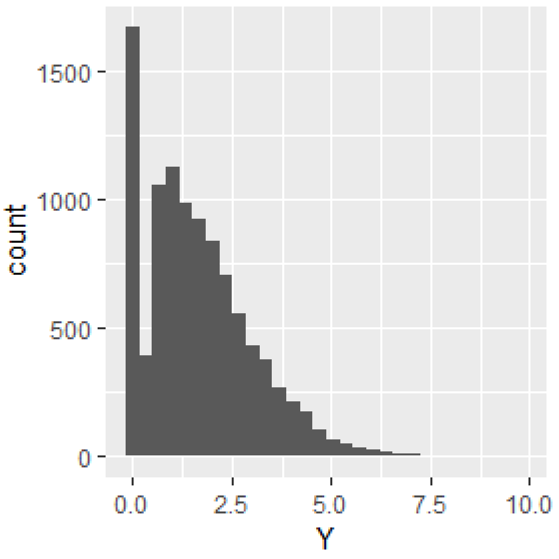


Figure 19: Tweedie distribution

Tweedie distributed data is similar to Poisson distributed data in the sense that the data skews to the left of the distribution. The key difference lies in that the value of 0, or at least the lowest value, stands out as the modal value, by a large margin. This is a characteristic seen in the Quantity variable, where many days have no travelers because tours are not being operated on that day, for example in the winter months. Using the Tweedie distribution instead of the Poisson distribution in all the models might lead to better performance across the board, without having to 'remove' data from parts of the year where the training data is not relevant (like months where there is no business).

5.2.5 Some potential variables left unexplored

If you were to be asked 'how many tickets will Go Fjords sell for the trip to Preikestolen on Friday morning', and you were allowed to ask one question before answering, it might be wise to ask: 'how many tickets for the trip have been sold already?'. '*Number of tickets sold at the time of prediction*' could be a useful predictor variable in practice, but it has not been used in this thesis.

Another variable that might be smart to include is '*cloud cover'*. This is another variable that is made available by yr.no and their API. Forecasted cloud cover percentage, how cloudy or sunny the day is expected to be, might impact the willingness of the customers to purchase tickets for outdoor experiences.

The reason these variables were not included in the models is primarily due to time constraints, the need for settling at some point with a set of variables, so the focus could shift toward testing and implementing models and documenting the process. These variables stand as directions in which to explore to potentially improve performance, for Go Fjords and anyone else looking to use the weather to forecast demand.

6. Conclusion

The goal of this thesis was to evaluate whether using weather data as predictive variables could increase the accuracy of a demand forecast for Go Fjords' trip to Preikestolen, as compared to forecasting solely on the basis of past sales data. A bonus goal was to demonstrate how such a demand forecasting solution can be technically implemented to provide a better basis for decision making, thus having the potential to provide value over time. The specific formulation of the topic question was:

Can weather forecast data make demand forecasts more accurate for Go Fjords, and how can business value be derived from such as forecast?

In order to answer the first part of the question, a range of models were tested for forecasting "Quantity" of tickets sold within the start of the trip. All models were tested using time series cross-validation, meaning essentially tested once for every data point, retraining itself after each forecast based on data that could at that point be considered 'observed', then moving to the next day and forecasting from there.

Two univariate benchmark models: a naïve model, and an ARIMA model that considers past values of "Quantity", as well as seasonal effects on the demand.

Three "candidate" models: A Generalized Additive Model (GAM), a Generalized Linear Model (GLM), and a Random Forest Regression model.

GLMs are regression models that can be used on data that is not normally distributed, by linking the Quantity variable to the predictive variables through a function that can account for the probability distribution of the Quantity variable. The GLM was implemented using the PyBats package which uses Bayesian statistics to update a "belief" the model holds about the data. This allowed the model to learn characteristics about the data, leading it to improve over time. This Bayesian learning feature renders the model a *Dynamic* GLM, or *DGLM*.

GAMs can be considered an extended form of GLMs, that like GLMs are flexible with respect to the probability distribution of the target variable, also uses a link function on each individual explanatory variable. These link functions allow GAMs to account for non-linear effects of the predictive variables, for example, that the difference between no rain and a little rain matters more than the difference between a heavy rain and very heavy rain. The

GAM was implemented using the “Prophet” framework, developed by Facebook. Prophet allows for highly scalable solutions for forecasting of many time series, with complicated seasonality. In the early phase of this thesis, Prophet was deemed to be the most promising candidate model. As such, a Prophet model was implemented in Microsoft Azure, which is the cloud provider Go Fjords uses and that TietoEVRY specializes in.

Random Forest Regression models produce predictions through creating multiple decision trees with some randomness to them to ensure they are not too similar and then averaging their predictions to create the final forecast. The Random Forest Regression model was first considered in the later stages of this thesis. After having implemented the GAM in Microsoft Azure, it became apparent that the Azure platform has sophisticated built-in functionality for machine learning. Azure’s Automated Machine Learning functionality (AML) was used to evaluate a large set of models, returning the Random Forest Regression model as the most accurate model that still allowed for explainability and scoring of variable predictive power.

All the models were compared by the metric Root Mean Squared Error (RMSE). RMSE was chosen as the basis for evaluation because, compared to other popular performance metrics like Mean Average Error (MAE), RMSE penalizes large errors disproportionately from having a ‘squared’ component. This characteristic makes RMSE a good metric to optimize for in cases where the forecasts are to be used as an input in discrete optimization with large intervals, such as in this thesis, with the number of buses to rent. Picking the model that minimizes out-of-sample RMSE ensures that large errors are avoided, and the wrong number of buses is rented as seldom as possible.

The model that performed best was the PyBats Dynamic GLM, by a large margin. The Prophet GAM barely beat the naïve benchmark model, which leads us to believe that the GAM was not tuned optimally. Additionally, it was discovered after implementation that although GAMs in theory allow for non-normally distributed data, the Prophet framework is not able to account for data being non-normally distributed, as of 2020. This resulted in forecasts that were sometimes even negative and had to be “clipped” to zero, and overall lacklustre performance. The Random Forest Regression model and the ARIMA model performed well, achieved an RMSE of 11.65 and 12.68, respectively. As both these models were implemented using thoroughly tested frameworks that automatically finds an appropriate model, and both frameworks yielded models that achieved scores in approximately the same ballpark, this strengthens the belief in the validity of these forecasts.

The PyBats Dynamic GLM achieved a much higher accuracy, with an RMSE of only 2.21. The exact reason for this is not clear, but factors that may have caused the big difference in performance is that the GLM has a dynamic component to it, where it learns about the data by updating the coefficients for its predictive variables over time. This renders the PyBats model highly adaptive to changes in the characteristics of the target variable, leading its performance to decline less than that of the other models, in the outlier year of 2020. However, the large discrepancy in performance between the DGLM and the other models, questions the validity of the DGLM forecast. More research is necessary.

The technical implementation was demonstrated using the Prophet model as a case study. The utilized stack of technologies was discussed, such as programming languages (Python, R, SQL, and Scala), and platforms (Microsoft Azure, Databricks), explaining their respective advantages and disadvantages. The fact that different programming languages have their unique strengths was the driving force behind using Databricks notebooks on top of the Microsoft Azure cloud platform. Databricks notebooks allowed code written in multiple programming languages to easily be deployed, allowing for them to be used for their respective strengths.

The process of retrieving data and performing the necessary cleaning- and preprocessing, was explained in the 'Deployment' section, and Python code examples on how this can be done, can be found in the appendix.

The usefulness of having a good forecast was demonstrated by constructing a Microsoft Power BI dashboard that displays the optimal number of buses to rent throughout the season, taking the weather into account. The Power BI dashboard updates itself automatically with the daily forecasts produced by the Prophet model, demonstrating how a forecasting solution can be deployed to create forecasts over time, thus having the potential to create continuous value through improved basis for decision making.

Sources

Allbusiness.com (2020). Retrieved from https://www.allbusiness.com/barrons_dictionary/dictionary-seasonality-4946957-1.html

Beers, B. (2020). Retrieved from https://www.investopedia.com/terms/r/regression.asp#:~:text=Regression%20is%20a%20statistical%20method,(known%20as%20independent%20variables)

Box, G. E. P., Jenkins, G. M. and Reinsel, G. C. (1994). Time Series Analysis: Forecasting and Control (3rd ed.). Upper Saddle River, NJ: Prentice–Hal

Brownlee, J. (2016). Retrieved from https://machinelearningmastery.com/backtest-machine-learning-models-time-series-forecasting/

Cameron, A. (2019). Retrieved from https://stackoverflow.com/questions/59809960/how-do-i-know-if-my-data-fit-a-poisson-distribution-using-r#:~:text=1%20Answer&text=You%20could%20try%20a%20dispersion,n%2D1%20degrees%20of%20freedom

cran.r-project.org. (2020). Retrieved from https://cran.r-project.org/web/packages/GlmSimulatoR/vignettes/tweedie_distribution.html

Cron, A. and Lavine, I. (2020). Retrieved from https://github.com/lavinei/pybats

Fedorov, V., Mannino, F. and Zhang, R. (2009). Consequences of dichotomization. Pharmaceutical statistics, 8(1), 50–61. https://doi.org/10.1002/pst.331

Hastie, T. J., Tibshirani, R. J. (1990). Generalized Additive Models. Chapman and Hall

Hayes, A. (2020). Retrieved from https://www.investopedia.com/terms/m/multicollinearity.asp

Hyndman, R. J. and Athanasopoulos, G. (2018). Forecasting: principles and practice. OTexts.

McCullagh, P. and Nelder, J.A. (1989). Generalized Linear Models. Chapman and Hall

MET Norway. (2020). Retrieved from Yr.no

MET Norway. (2020). Retrieved from https://www.yr.no/place/Norway/Rogaland/Stavanger/Stavanger/forecast.xml

MET Norway. (2020). Frost API. Retrieved from https://frost.met.no/index.html

Molnar, C. (2020). Retrieved from https://christophm.github.io/interpretable-ml-book/extend-lm.html?fbclid=IwAR2rXDmLhqGcePbvkP8vBNlWGqidsglg88fiLGuZniNsh2z2zbC0f9byOLc

Motoharu, D. (2020). Retrieved from https://medium.com/swlh/facebook-prophet-426421f7e331

Nau, R. (2020). Retrieved from https://people.duke.edu/~rnau/411arim.htm

Obeidat, B., Smith, B. and Heintz, B. (2020). Fine-Grained Time Series Forecasting at Scale with Prophet and Apache Spark. Retrieved from https://pages.databricks.com/rs/094-YMS-629/images/Fine-Grained-Time-Series-Forecasting.html?_ga=2.220205147.2112692442.1591844546-225663068.1585060489

Oxford Dictionaries (2020). Retrieved from https://www.lexico.com/definition/overfitting

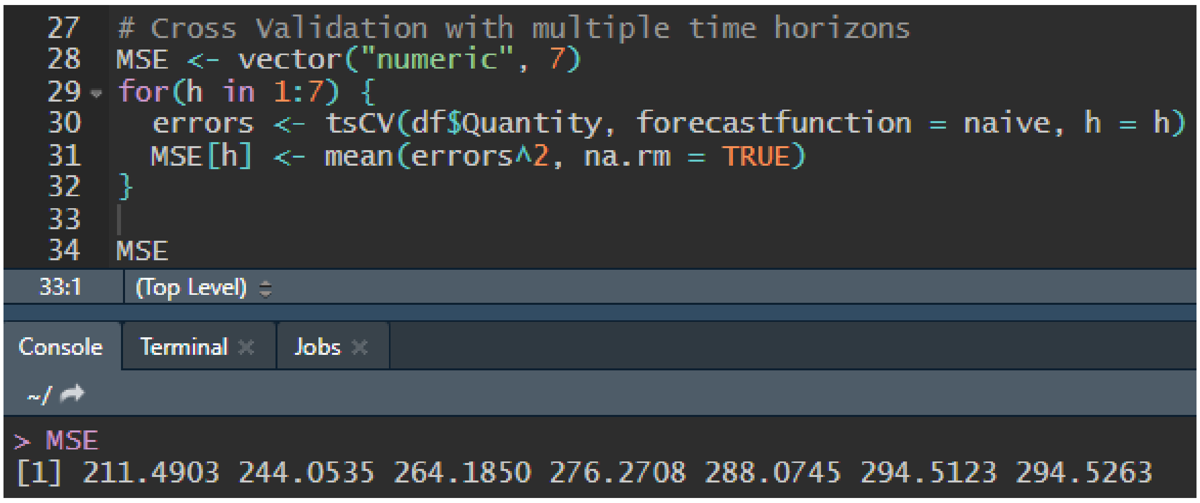
Shrivastava, S. (2020). Cross Validation in Time Series. Retrieved from https://medium.com/@soumyachess1496/cross-validation-in-time-series-566ae4981ce4

Taylor, S. J., Letham, B. (2017). Forecasting at scale. PeerJ Preprints 5:e3190v2 Retrieved from https://doi.org/10.7287/peerj.preprints.3190v2

Yau, J. (2018). Retrieved from https://www.youtube.com/watch?v=i40Road82No

Appendix

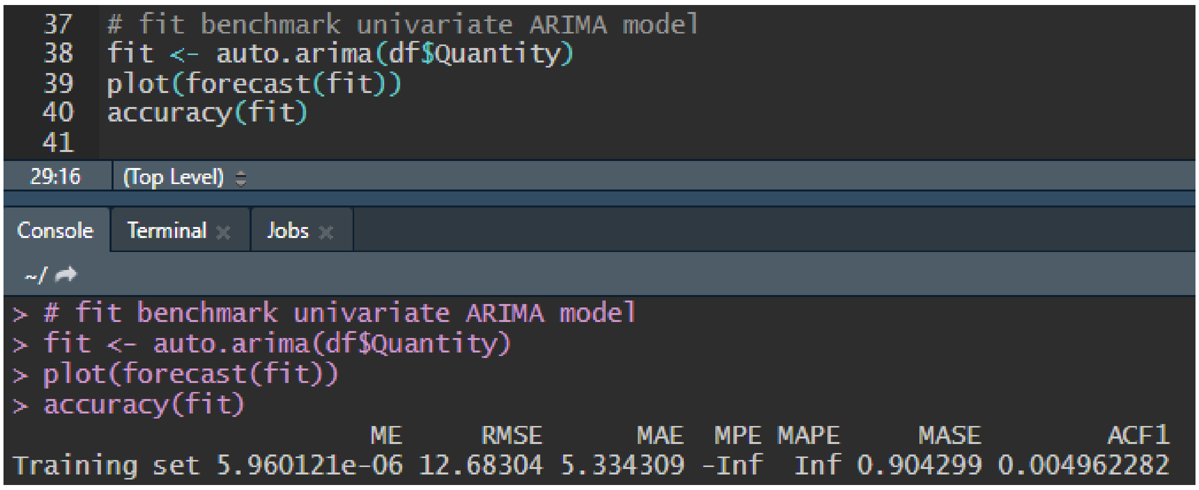
Appendix 1: Performance of models



```
# Cross Validation with multiple time horizons
MSE <- vector("numeric", 7)
for(h in 1:7) {
  errors <- tsCV(df$Quantity, forecastfunction = naive, h = h)
  MSE[h] <- mean(errors^2, na.rm = TRUE)
}

MSE
```

```
> MSE
[1] 211.4903 244.0535 264.1850 276.2708 288.0745 294.5123 294.5263
```

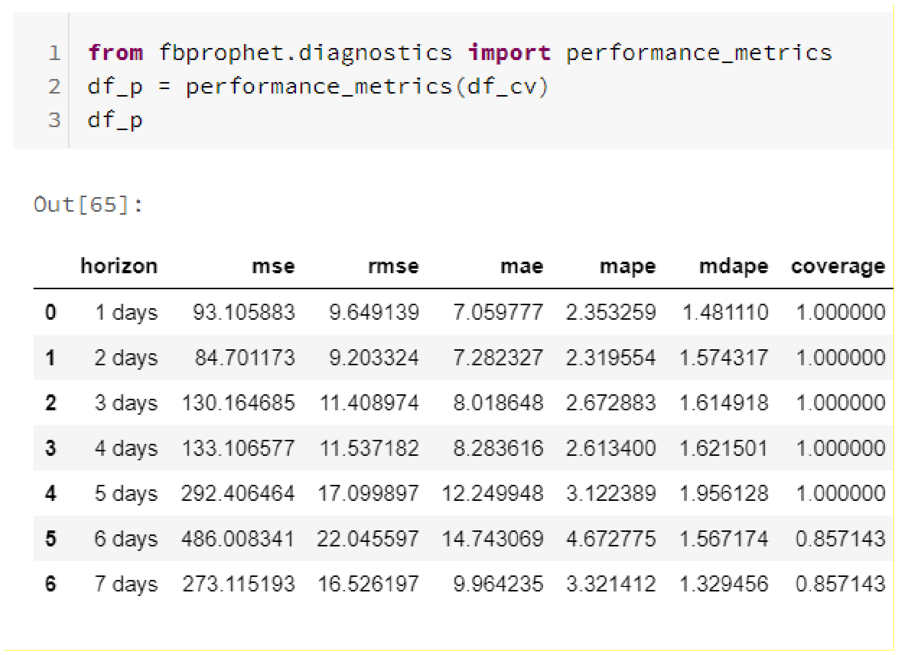
Appendix 1 a: Benchmark naïve model w/ forecast horizons 1 to 7 performance metrics



```
# fit benchmark univariate ARIMA model
fit <- auto.arima(df$Quantity)
plot(forecast(fit))
accuracy(fit)
```

```
> # fit benchmark univariate ARIMA model
> fit <- auto.arima(df$Quantity)
> plot(forecast(fit))
> accuracy(fit)
                       ME     RMSE      MAE  MPE MAPE     MASE        ACF1
Training set 5.960121e-06 12.68304 5.334309 -Inf  Inf 0.904299 0.004962282
```

Appendix 1 b: Benchmark univariate ARIMA model performance metrics

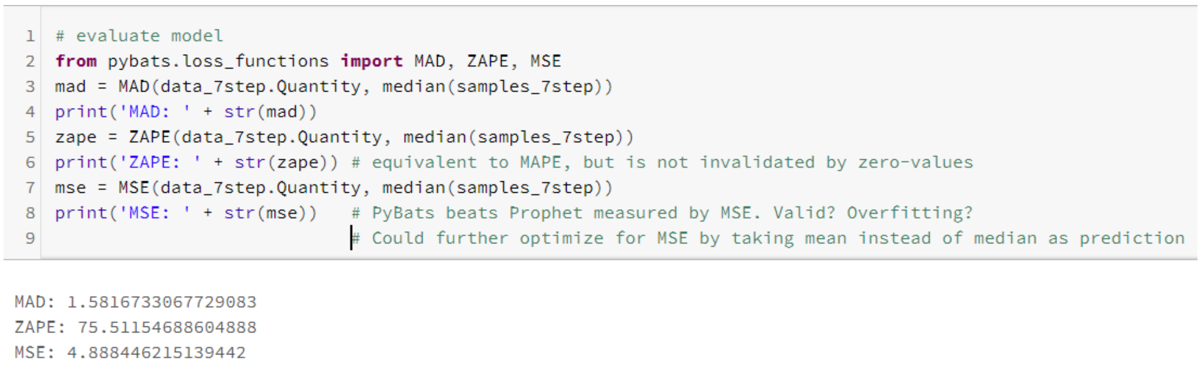


```python
from fbprophet.diagnostics import performance_metrics
df_p = performance_metrics(df_cv)
df_p
```

Out[65]:

	horizon	mse	rmse	mae	mape	mdape	coverage
0	1 days	93.105883	9.649139	7.059777	2.353259	1.481110	1.000000
1	2 days	84.701173	9.203324	7.282327	2.319554	1.574317	1.000000
2	3 days	130.164685	11.408974	8.018648	2.672883	1.614918	1.000000
3	4 days	133.106577	11.537182	8.283616	2.613400	1.621501	1.000000
4	5 days	292.406464	17.099897	12.249948	3.122389	1.956128	1.000000
5	6 days	486.008341	22.045597	14.743069	4.672775	1.567174	0.857143
6	7 days	273.115193	16.526197	9.964235	3.321412	1.329456	0.857143

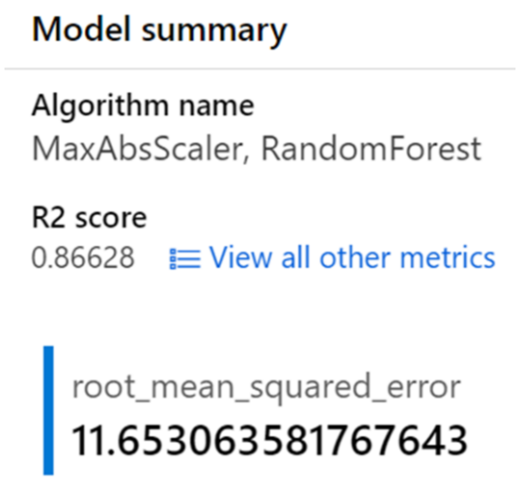
Appendix 1 c: Prophet GAM performance metrics



```python
# evaluate model
from pybats.loss_functions import MAD, ZAPE, MSE
mad = MAD(data_7step.Quantity, median(samples_7step))
print('MAD: ' + str(mad))
zape = ZAPE(data_7step.Quantity, median(samples_7step))
print('ZAPE: ' + str(zape)) # equivalent to MAPE, but is not invalidated by zero-values
mse = MSE(data_7step.Quantity, median(samples_7step))
print('MSE: ' + str(mse))   # PyBats beats Prophet measured by MSE. Valid? Overfitting?
                            # Could further optimize for MSE by taking mean instead of median as prediction
```

```
MAD: 1.5816733067729083
ZAPE: 75.51154688604888
MSE: 4.888446215139442
```

Appendix 1 d: PyBats DGLM performance metrics



Model summary

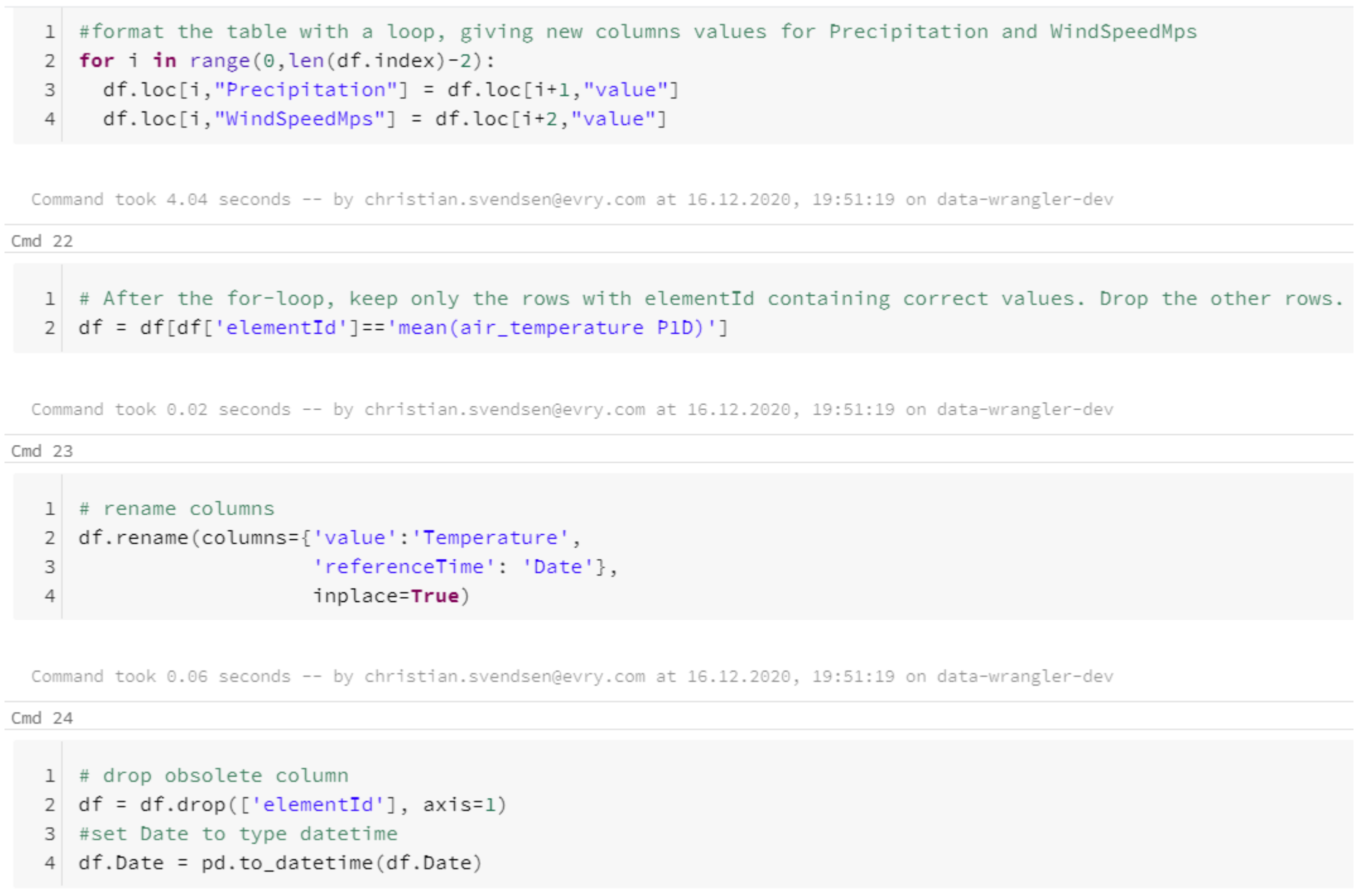
Algorithm name
MaxAbsScaler, RandomForest

R2 score
0.86628 View all other metrics

root_mean_squared_error
11.653063581767643

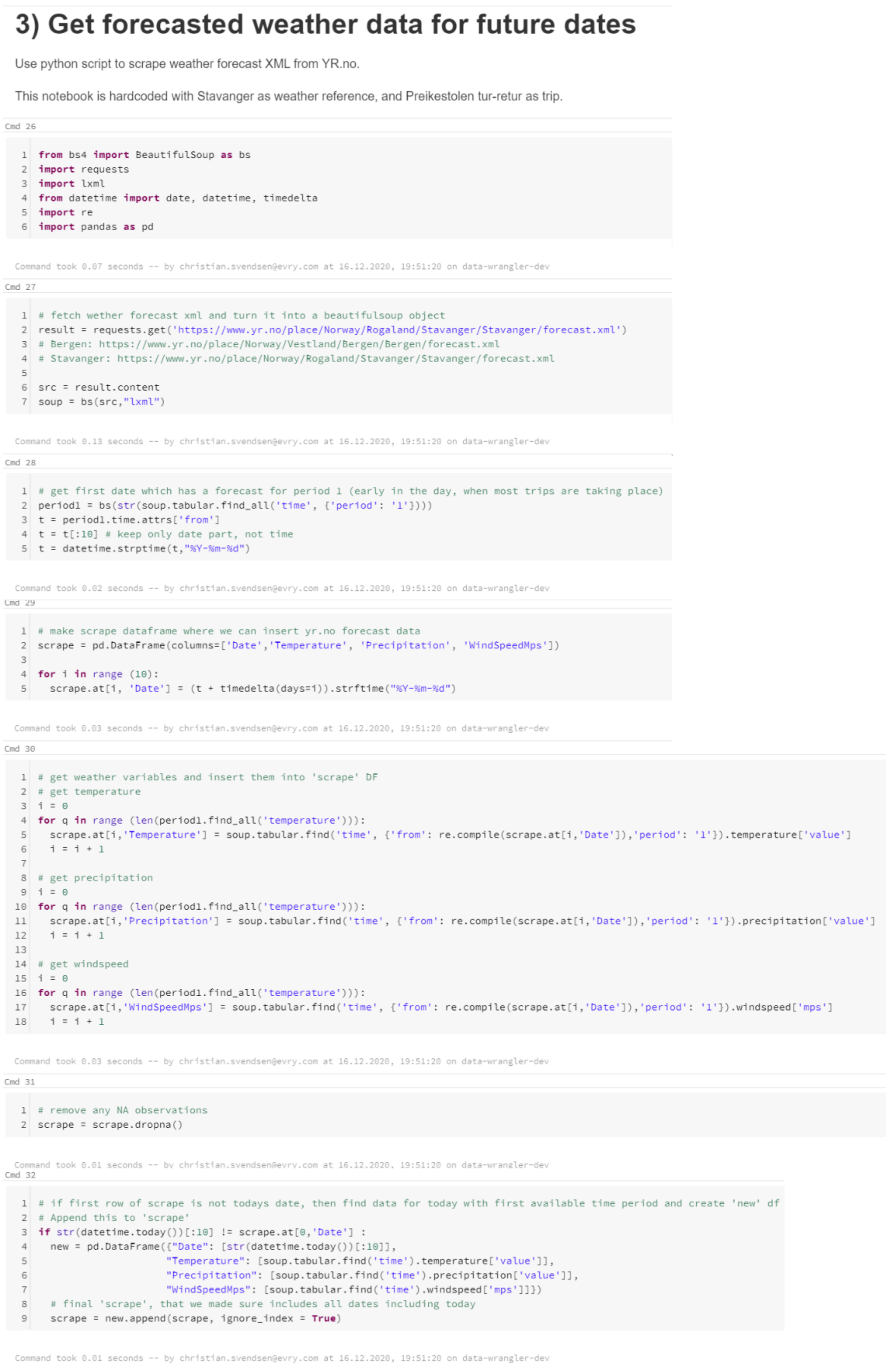
Appendix 1 e: Azure Random Forest Regression performance metrics

Appendix 2: Code examples



```
#format the table with a loop, giving new columns values for Precipitation and WindSpeedMps
for i in range(0,len(df.index)-2):
  df.loc[i,"Precipitation"] = df.loc[i+1,"value"]
  df.loc[i,"WindSpeedMps"] = df.loc[i+2,"value"]

Command took 4.04 seconds -- by christian.svendsen@evry.com at 16.12.2020, 19:51:19 on data-wrangler-dev
Cmd 22

# After the for-loop, keep only the rows with elementId containing correct values. Drop the other rows.
df = df[df['elementId']=='mean(air_temperature P1D)']

Command took 0.02 seconds -- by christian.svendsen@evry.com at 16.12.2020, 19:51:19 on data-wrangler-dev
Cmd 23

# rename columns
df.rename(columns={'value':'Temperature',
                   'referenceTime': 'Date'},
                   inplace=True)

Command took 0.06 seconds -- by christian.svendsen@evry.com at 16.12.2020, 19:51:19 on data-wrangler-dev
Cmd 24

# drop obsolete column
df = df.drop(['elementId'], axis=1)
#set Date to type datetime
df.Date = pd.to_datetime(df.Date)
```

Appendix 2 a: Turn historical data into 'wide' format



3) Get forecasted weather data for future dates

Use python script to scrape weather forecast XML from YR.no.

This notebook is hardcoded with Stavanger as weather reference, and Preikestolen tur-retur as trip.

Cmd 26

```
from bs4 import BeautifulSoup as bs
import requests
import lxml
from datetime import date, datetime, timedelta
import re
import pandas as pd
```

```
Command took 0.07 seconds -- by christian.svendsen@evry.com at 16.12.2020, 19:51:20 on data-wrangler-dev
```

Cmd 27

```
# fetch wether forecast xml and turn it into a beautifulsoup object
result = requests.get('https://www.yr.no/place/Norway/Rogaland/Stavanger/Stavanger/forecast.xml')
# Bergen: https://www.yr.no/place/Norway/Vestland/Bergen/Bergen/forecast.xml
# Stavanger: https://www.yr.no/place/Norway/Rogaland/Stavanger/Stavanger/forecast.xml

src = result.content
soup = bs(src,"lxml")
```

```
Command took 0.13 seconds -- by christian.svendsen@evry.com at 16.12.2020, 19:51:20 on data-wrangler-dev
```

Cmd 28

```
# get first date which has a forecast for period 1 (early in the day, when most trips are taking place)
period1 = bs(str(soup.tabular.find_all('time', {'period': '1'})))
t = period1.time.attrs['from']
t = t[:10] # keep only date part, not time
t = datetime.strptime(t,"%Y-%m-%d")
```

```
Command took 0.02 seconds -- by christian.svendsen@evry.com at 16.12.2020, 19:51:20 on data-wrangler-dev
```

Cmd 29

```
# make scrape dataframe where we can insert yr.no forecast data
scrape = pd.DataFrame(columns=['Date','Temperature', 'Precipitation', 'WindSpeedMps'])

for i in range (10):
  scrape.at[i, 'Date'] = (t + timedelta(days=i)).strftime("%Y-%m-%d")
```

```
Command took 0.03 seconds -- by christian.svendsen@evry.com at 16.12.2020, 19:51:20 on data-wrangler-dev
```

Cmd 30

```
# get weather variables and insert them into 'scrape' DF
# get temperature
i = 0
for q in range (len(period1.find_all('temperature'))):
  scrape.at[i,'Temperature'] = soup.tabular.find('time', {'from': re.compile(scrape.at[i,'Date']),'period': '1'}).temperature['value']
  i = i + 1

# get precipitation
i = 0
for q in range (len(period1.find_all('temperature'))):
  scrape.at[i,'Precipitation'] = soup.tabular.find('time', {'from': re.compile(scrape.at[i,'Date']),'period': '1'}).precipitation['value']
  i = i + 1

# get windspeed
i = 0
for q in range (len(period1.find_all('temperature'))):
  scrape.at[i,'WindSpeedMps'] = soup.tabular.find('time', {'from': re.compile(scrape.at[i,'Date']),'period': '1'}).windspeed['mps']
  i = i + 1
```

```
Command took 0.03 seconds -- by christian.svendsen@evry.com at 16.12.2020, 19:51:20 on data-wrangler-dev
```

Cmd 31

```
# remove any NA observations
scrape = scrape.dropna()
```

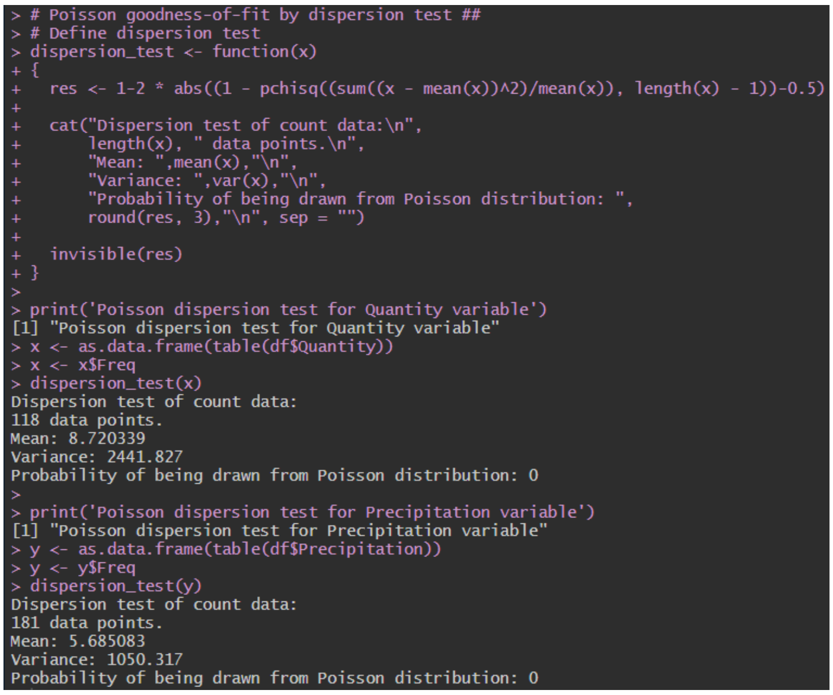
```
Command took 0.01 seconds -- by christian.svendsen@evry.com at 16.12.2020, 19:51:20 on data-wrangler-dev
```

Cmd 32

```
# if first row of scrape is not todays date, then find data for today with first available time period and create 'new' df
# Append this to 'scrape'
if str(datetime.today())[:10] != scrape.at[0,'Date'] :
  new = pd.DataFrame({"Date": [str(datetime.today())[:10]],
                      "Temperature": [soup.tabular.find('time').temperature['value']],
                      "Precipitation": [soup.tabular.find('time').precipitation['value']],
                      "WindSpeedMps": [soup.tabular.find('time').windspeed['mps']]})
  # final 'scrape', that we made sure includes all dates including today
  scrape = new.append(scrape, ignore_index = True)
```

```
Command took 0.01 seconds -- by christian.svendsen@evry.com at 16.12.2020, 19:51:20 on data-wrangler-dev
```

Appendix 2 b: Web scraping weather forecast from yr.no



```
> # Poisson goodness-of-fit by dispersion test ##
> # Define dispersion test
> dispersion_test <- function(x)
+ {
+   res <- 1-2 * abs((1 - pchisq((sum((x - mean(x))^2)/mean(x)), length(x) - 1))-0.5)
+
+   cat("Dispersion test of count data:\n",
+       length(x), " data points.\n",
+       "Mean: ",mean(x),"\n",
+       "Variance: ",var(x),"\n",
+       "Probability of being drawn from Poisson distribution: ",
+       round(res, 3),"\n", sep = "")
+
+   invisible(res)
+ }
>
> print('Poisson dispersion test for Quantity variable')
[1] "Poisson dispersion test for Quantity variable"
> x <- as.data.frame(table(df$Quantity))
> x <- x$Freq
> dispersion_test(x)
Dispersion test of count data:
118 data points.
Mean: 8.720339
Variance: 2441.827
Probability of being drawn from Poisson distribution: 0
>
> print('Poisson dispersion test for Precipitation variable')
[1] "Poisson dispersion test for Precipitation variable"
> y <- as.data.frame(table(df$Precipitation))
> y <- y$Freq
> dispersion_test(y)
Dispersion test of count data:
181 data points.
Mean: 5.685083
Variance: 1050.317
Probability of being drawn from Poisson distribution: 0
```

Appendix 2 c: Poisson dispersion tests for Quantity and Precipitation