



SOFTWARE ENGINEERING GROUP PROJECT

IMPERIAL COLLEGE LONDON

DEPARTMENT OF COMPUTING

Travelport Predictive Analysis

Authors:

Arthur-Mihai Niculae
Benjamin Gunton
Catalina Popescu
Harjuno Perwironegoro
Nimesh Subedi
Thomas Price

Supervisor:

Dr. Anandha Gopalan

Date: January 9, 2018

Executive Summary

Being able to plan for the future is vital for a growing travel agency, but knowing what to plan for needs more than just good guesswork. How can an agency find out about the trends in the future market?

The Travelport Predictive Analysis (TPA) tool is the web application that provides information on future booking trends and the projected market shares of bookings, for travel destinations all around the world. Taking into account the sales of your competitors, it provides the information you need to take advantage of the future.

Contents

1	Introduction	4
1.1	Motivation	4
1.2	Objectives	4
1.3	Travelport	5
1.4	Introducing TPA	5
2	Project Management	8
2.1	Development Methodology	8
2.1.1	Choosing a Sprint length	8
2.2	Tools	8
2.2.1	Trello boards	8
2.2.2	Slack	9
2.2.3	Git and Gitlab	9
2.3	Planning	10
2.4	Project Structure - Milestones	10
2.5	Group Organization	11
2.6	Task Allocation	11
2.7	User Feedback	11
3	Design and Implementation	12
3.1	TPA Framework	12
3.2	Web Frontend	13
3.3	Web Backend	14
3.4	Communication between Frontend and Backend	15
3.4.1	Representational State Transfer (REST)	15
3.5	Time Series Analysis	16
3.5.1	Seasonal ARIMA	16
3.5.2	Prophet	22
3.6	Cloud Storage and Azure	22
3.7	Communication between Backend and Azure	24
3.8	Continuous Integration (CI)	24
3.8.1	Frontend	24
3.8.2	Backend	25
4	Technical Challenges	26
4.1	Angular 2	26
4.2	Prophet	26
4.3	Microsoft Azure	26
4.4	Hadoop	27
5	Risks and Mitigations	28
5.1	Learning Curve	28
5.2	Cloud Services	28
5.3	External Data	28
5.4	Pivoting the Objectives	28

6	Evaluation	30
6.1	Efficiency Testing	30
6.2	ARIMA vs Prophet	31
7	Conclusion	33
7.1	Deliverables	33
7.2	What we learned	33
7.3	What we would be different if we had to do it again	33
8	Future Improvements	34
8.1	Booking Heatmap	34
8.2	Exogenous regressors [1]	34
8.3	Other Time Series Algorithms	34
8.4	Statistics	35
9	License	36
9.1	Component Licenses	36
9.2	TPA License	36
10	Bibliography	38
11	Appendix	40
11.1	User Guide	40
11.1.1	Client	40
11.1.2	Server	40

1 Introduction

1.1 Motivation

Businesses use software applications, known as e-commerce platforms, to manage and measure the success of the services they provide. As an e-commerce platform, being able to provide businesses with the most insight and opportunity for growth is a key factor in staying competitive.

Predictive software is currently being employed in the travel industry. Mobile applications, such as Hopper [2], predict dates for optimal ticket prices for travelling, information which is also advantageous for travel agencies.

As a travel commerce platform handling millions of bookings for thousands of travel agencies every year, Travelport has the means, and an incentive, to also provide their clients with predictive information, and keep a competitive edge.

1.2 Objectives

The aim of the project is to build a web-accessible predictive analytics tool that will assist travel agencies in predicting trends, with the success of the tool defined by its ability to provide informative, human-interpretable trends.

The initial trends that Travelport wished to locate were:

- Pricing - the future average prices of tickets for a given destination. Useful for deciding competitive, yet profitable prices.
- Market sales share - the proportion of sales of an agency out of the total, for a given destination. Useful for measuring growth against other agencies.
- Market sales growth - the future sales of an agency in a given destination. Useful for deciding where to invest in marketing.

Due to timing issues induced by anonymity requirements, it was not possible to use financial data and therefore discovering these trends was infeasible. Although the pricing trend had to be dropped, the information provided by the other two trends were instead approximated by using the number of bookings made instead of sales.

The new trends set by Travelport were:

- Market bookings share - the proportion of bookings through an agency out of the total, for a given destination.
- Market bookings growth - the future number of bookings through an agency in a given destination.

1.3 Travelport

This project was proposed by Travelport, a travel commerce platform providing distribution, technology, payment and other solutions for the travel and tourism industry.

Although the tool is designed for use by their travel agencies, Travelport will use the outcome as a proof of concept code for market trend prediction, which could be adapted and integrated with their existing services.

1.4 Introducing TPA

The Travelport Predictive Analysis tool, from now on referred to as TPA, is the developed web service providing booking predictions for travel agencies registered with Travelport. TPA demonstrates that it is feasible to dynamically create predictions for more relaxed¹ trends:

1. Market bookings share (across all destinations)
2. Market bookings growth (across all destinations)
3. Destination bookings growth (across all agencies)

TPA is available to all travel agencies that use Travelport, and provides information specific to each agency.

An agency may access TPA via a web browser², and request dynamic predictions for their bookings, performance against competitors, or bookings for a given airport code.

An example set of predictions are shown on the next pages in the form of various graphs:

¹Extremely few agencies and destinations have the scale of data required for prediction, so the current dataset of TPA is aggregated either across all agencies or across all destinations to produce meaningful trends that are fit for purpose.

²For browser requirements, see section 11.1.

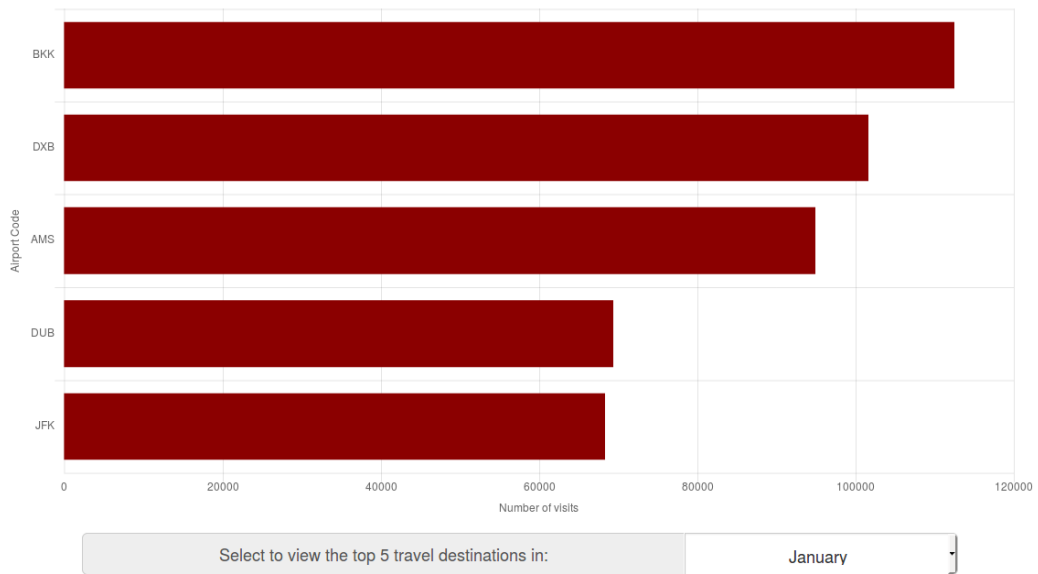


Figure 1: A chart showing the top 5 most popular destinations in a given month

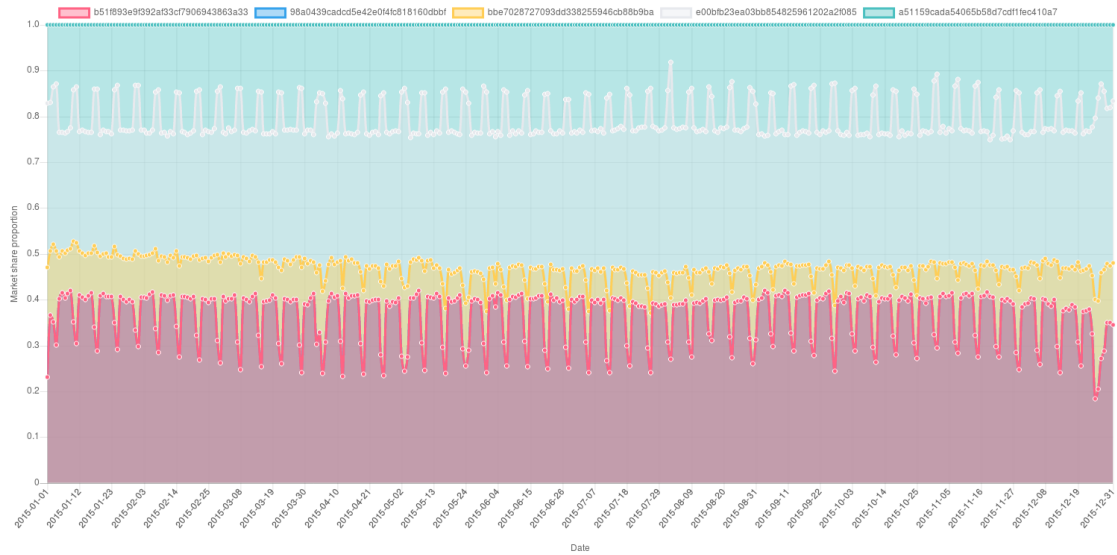


Figure 2: A chart showing the market share for a given agent

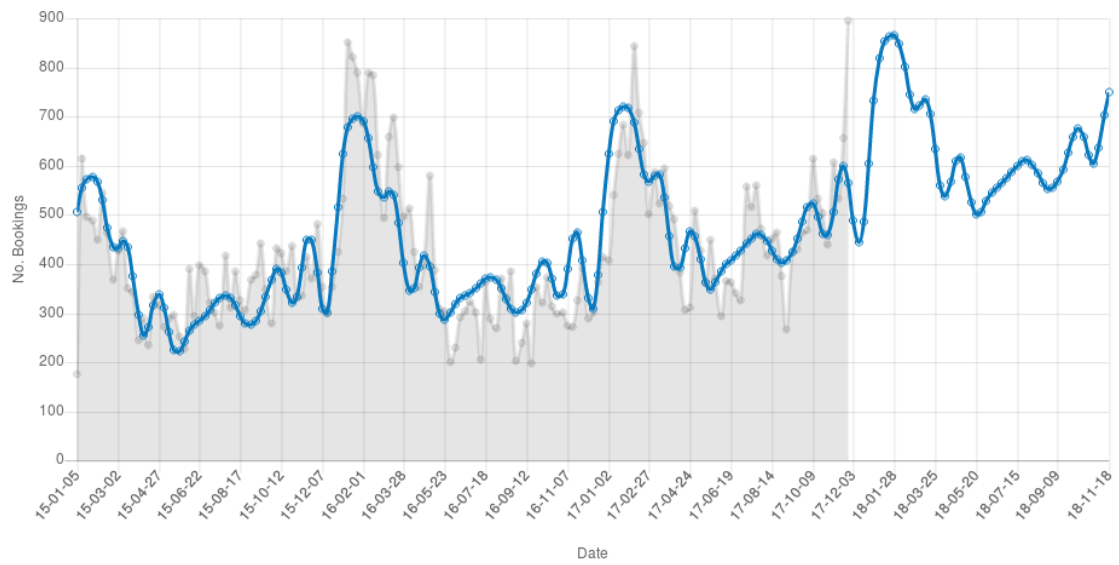


Figure 3: Booking predictions made using Prophet for a travel agent

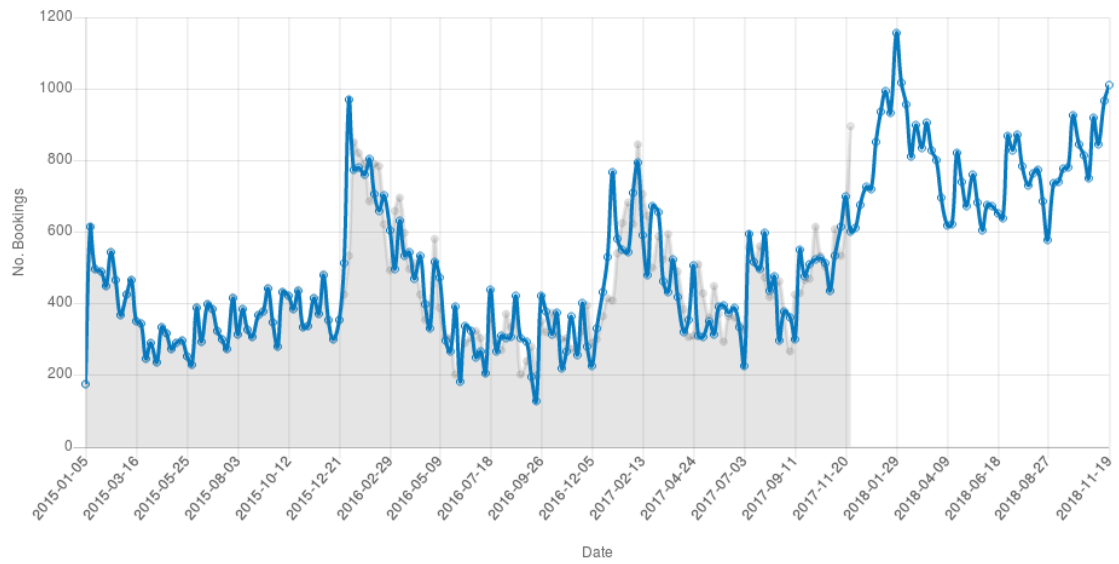


Figure 4: Booking predictions made using ARIMA for the same travel agent

2 Project Management

2.1 Development Methodology

After researching various possible software development methodologies, we came to the conclusion that **Scrum** was the best option for our team, especially considering that the majority of our team had a positive experience with it. We also decided on using TDD and pair programming where possible, but recognised that in the interest of time and conflicting schedules it may not have always been possible.

2.1.1 Choosing a Sprint length

A Sprint is a unit of time in which a usable feature is to be developed. As the project was to be balanced with our studies, and we required time to research unfamiliar content for the project, one-week Sprints would not have been enough time to complete a feature and thus receive constructive feedback. We decided that two-week Sprints were ideal for our project.

2.2 Tools

Throughout our planning and development process we have used a number of tools to assist rapid development.

2.2.1 Trello boards

To support our Scrum approach towards the project, we decided to use Trello as our Agile Project Management System. The Trello boards helped us to use agile planning constructs via virtual backlog, Sprints, stories and tasks (to do, in progress, and done). This approach was particularly useful for being able to scroll through our current Sprint, discuss the current stories, and gain quick feedback from the team when making decisions and implementing features.

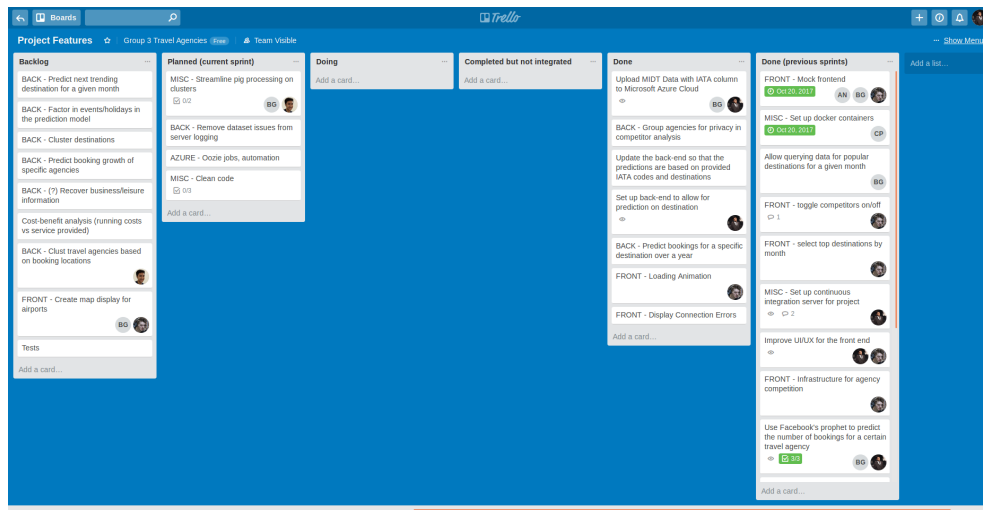


Figure 5: Project Features board in Trello

2.2.2 Slack

Since face-to-face communication was not always an option, we relied on electronic communication methods to stay in contact.

Slack was mainly used as a medium to ask specific technical questions, consult with the group when making smaller design decisions, and update others on key changes. Having multiple discussion boards for different topics was particularly useful for keeping notifications focused. Additionally, the calendar integration feature allowed us to use Slack to organise meetings.

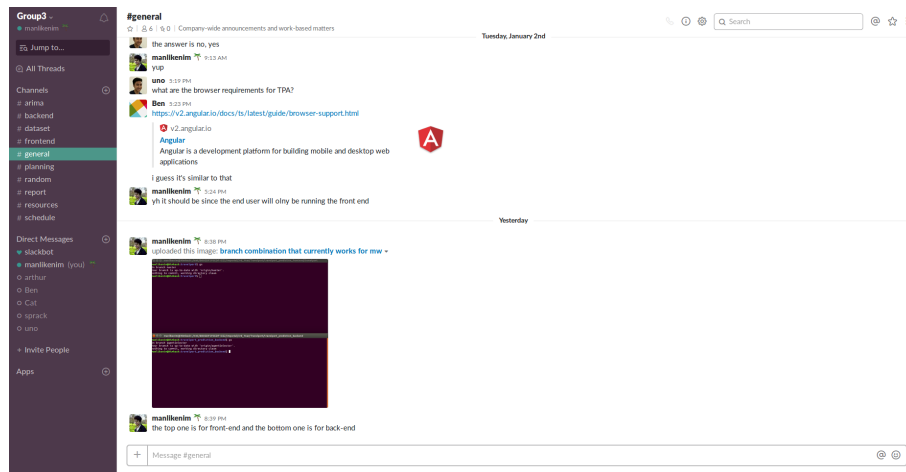


Figure 6: General channel on our Slack domain

2.2.3 Git and Gitlab

When working on a team project, being able to revisit earlier versions of the project is crucial. For our version control, we chose to use Git hosted on the Imperial Computing department's Gitlab.

Git is a distributed version control system. This means that a user has a local repository which contains all of the changes made to a project. Additionally, there is a remote repository on the server which is used to coordinate changes between different users.

Git is particularly powerful as a version control tool, as it supports branches. When using a branch, we can work on particular features separately without interfering with other branches. Then, when a feature is finished, it can be merged completely into the main branch. Using branches made it easy to concurrently work on different features of our project.

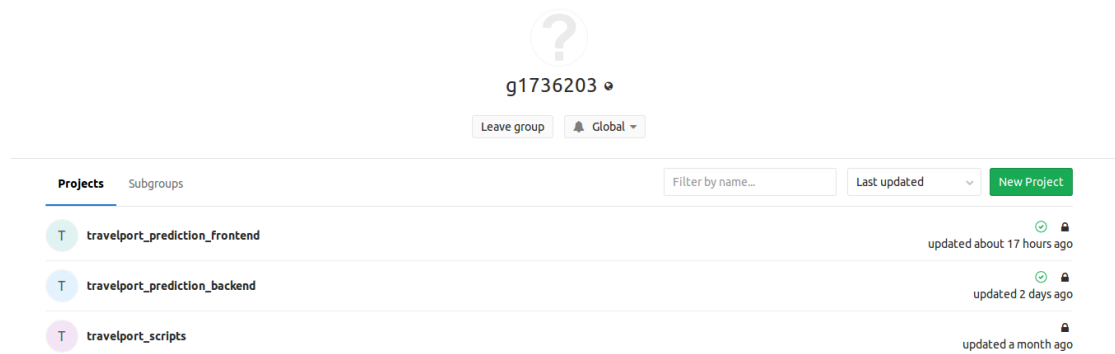


Figure 7: Gitlab group consisting of project-related repositories

2.3 Planning

To fill in our backlog, we talked to our supervisor, Anandha, and our contact from Travelport, Guido, to gain an understanding of the user's point of view. We then created stories based on the user requirements.

2.4 Project Structure - Milestones

At the beginning of each milestone, the group decided on two main aims to be completed before the next checkpoint meeting. These decisions were often made with the assistance of the industrial partners. The aims for each checkpoint were as follows:

Milestone 1

- Obtain the MIDT dataset from Travelport
- Create a mock-up frontend for the application

Milestone 2

- Add a feature to analyze popular destinations for a given month
- Improve the application frontend

Milestone 3

- Predict bookings for a given destination over the year using Prophet [3] and ARIMA
- Cluster travel agencies into appropriate groups

Milestone 4

- Predict travel agency booking growth for a location based on market predictions
- Group competitors for privacy

2.5 Group Organization

To keep the team organized during meetings, Harjuno was appointed Scrum Master, and the role of the user was taken by Guido and Eytan, our partners from Travelport. As Scrum required, we didn't assign specific development roles to team members, meaning that each team member was free to take on any task they liked when they were free. In this manner, all of us were familiar with most parts of the project and the discussions and meetings were much more meaningful as all members could contribute to all topics arising in meeting.

During the **Sprint**, members owned specific stories, meaning that they had more responsibility on the story than the other members participating in designing it. Stories could be broken down into tasks, and tasks could possibly be broken down into smaller tasks if appropriate. The advantage of this action was that members could focus on individual tasks and there was a clearer indication on what each member was doing. Smaller tasks allowed multiple developers to work concurrently.

Before a Sprint started, our team had a planning meeting to assign tasks to our stories. During the Sprint, we had meetings every two days to report what we had done and what we were going to do next. We reported anything that was blocking our progress in order to allow discussions on possible solutions. At the end of each Sprint we had a retrospective meeting along with our supervisor and some of the employees from our industrial partner, Travelport, where we discussed what could be improved in our workflow or what was particularly successful.

2.6 Task Allocation

- **Arthur** mainly worked on writing and optimizing Pig scripts for queries on the dataset, and Python scripts to manipulate the resulting tables and give them the correct format as needed by frontend.
- **Ben** worked on setting up and managing the Azure services for storing and querying the data, and also worked on the backend of the project.
- **Catalina** mainly focused on implementing the predictions using autoregressive integrated moving averages and some dynamic regression, and set up the Docker containers along with the backend server.
- **Harjuno** worked on implementing competitor analysis, and streamlined working with Pig and Azure.
- **Nimesh** mainly worked on implementing the predictions using Facebook Prophet, along with adding some features on frontend, as well as backend, and setting up Continuous Integration servers.
- **Tom** was in charge of the front end UI of the application.

2.7 User Feedback

Because our project was proposed by an industrial partner of Imperial College London, our main user feedback came from Eytan and Guido, two Travelport employees that we have worked with throughout the development of the project, as well as Anthony, an employee of Beyond Analysis which is the company that handles Travelport's data.

Although they would not be the primary users of our tool, they had plenty of experience with clients, and could also draw comparisons from Travelport's current web tools for travel agencies.

3 Design and Implementation

3.1 TPA Framework

Travelport Predictive Analysis is an application which allows users to view graphs describing the data collected by Travelport and perform some prediction on trends which might occur. The features available to a user are:

- View the top most traveled to destinations per month.
- Display a stacked area graph of the share of total bookings of different booking agencies³.
- View a graph of total bookings per travel agency against time, and predict how this might change using ARIMA and Prophet models.
- View a graph of total bookings of flights to a given destination against time, and predict how this might change using ARIMA and Prophet models.

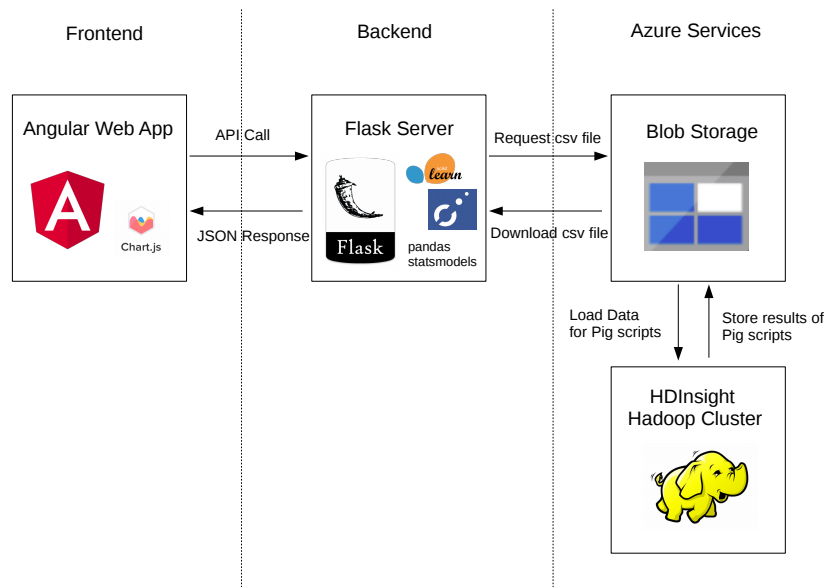


Figure 8: Communication between various components of TPA

³An agency is being distinguished by the IATA code. In reality one agency may book through multiple IATA codes, however the data supplied for the project did not have a mapping between IATA code and the company it belongs to.

For any of the features listed on the previous page, the typical timeline for loading the feature is as follows:

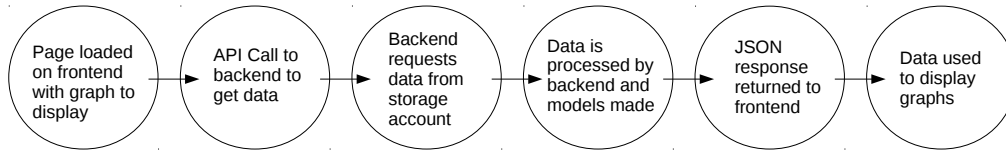


Figure 9: Typical timeline for loading a feature

Each part of the framework will be described in more detail.

3.2 Web Frontend

The frontend for the application was written in TypeScript using the Angular 2 framework [4]. As most of the work would be put into analyzing the data for trends, we required a malleable framework that was capable of displaying many data sets in a variety of ways. Another framework that was considered was ReactJS [5] as both have excellent command line interfaces [6] for generating large amounts of the bare bones functionality that was required to start our application. As a group we decided that it was in our best interest to use Angular due to familiarity from previous projects.

Another large reason why Angular 2 was a good fit for our application was its compatibility with Chart.js [7]. Chart.js is an open source HTML5/JavaScript chart library that was modified to be compatible with Angular 2 taking the form of ng2-charts [8]. When ng2-charts is imported as a module it is possible to use the underlying Chart.js features within Angular. This was advantageous to the project as it meant we could display the data dynamically in a variety of ways early on which kept our options open for what to display. This also meant the turnaround time on charting the data was quick which was important for getting feedback on prediction models during the checkpoint meetings.

However ng2-charts doesn't support two way data binding of HTML elements and so is considered more of a wrapper for Chart.js than a fully integrated Angular module. In Angular, two way data binding is when a HTML element is bound to both a variable and any events that trigger a change in that variable so that the HTML element can be updated at the same time. This led to some problems regarding the updating of chart datasets when querying a different value that required a work-around. If the application were deployed it would be advisable to switch to a different method of displaying the data or to contribute to ng2-charts by fixing the bug.

The application was arranged using the following Angular components:

- **app.component.ts** - the main component of the application. It has a navigation bar at the head of the page for selecting which of each component is displayed in the router outlet underneath.

- **about-page.component.ts** - the default page loaded when visiting the website. It explains some basic information about the project.
- **top-destinations.component.ts** - displays the top 5 travel destinations for a selected month. See Figure 1
- **booking-prediction.component.ts** - displays the booking predictions for a selected travel agent. See Figure 4
- **agency-competition.component.ts** - displays a stacked graph representing the market share of a selected agent. See Figure 2
- **destination-prediction.component.ts** - displays the booking predictions for a selected destination.
- **http-api-service.service.ts** - an injectable service that provided a layer of abstraction to each component when querying the Back-end API. It handles the creation of URIs and the subscription to JSON observables from HTTP requests

Angular uses Jasmine/Karma as its testing framework [9]. Due to the simplicity of the application, each component could only be unit tested for some basic functionality such as object creation and dependency injection. End to end tests included checking correct page navigation, routing and connection to the backend.

3.3 Web Backend

The application's backend uses Python Flask. We planned to use a Python backend before beginning the project, since we were already aware that various Python modules existed for data analysis and machine learning. Flask is an easy to use Python framework that allowed us to quickly set up the application's Back-end and get basic functionality running.

For making predictions about the data, there are two methods used by the backend. These are ARIMA, a statistical model implemented by members of the team, and Prophet, a Facebook library which creates a prediction according to its own method. Both of these methods take the existing data, and attempt to fit a model to it which allows prediction of data points past the date of the input. Both methods were selected because they each have a method for fitting a seasonality to the data which allows for a more accurate prediction. This was vital since the project required us to spot trends values over the course of a year, which meant that our models needed to take into account the repeating nature of our data. By using two distinct methods, we were able to compare the strengths and weaknesses of each. More detail on each implementation is given below.

The tasks performed by the backend are best described by the functions in the various Python files. These are:

- **app.py** - sets up the server to receive and respond to requests, containing one function per API call.
- **arima_predictions.py** - contains functions for making predictions using the ARIMA model. There are functions to fit a model for the number of bookings for a given destination, and the number of bookings for a given agency.

- **prophet_prediction.py** - contains functions for making predictions using the Prophet library. Similarly to `arima_predictions.py` there are functions for modeling the number of bookings for a given destination or agency.
- **util.py** - contains functions for communicating with the azure storage space (see below) and the retrieving data used to create models.
- **competitors.py** - functions to support the generation of the stacked area graph to analyze various agencies' share of the total bookings.

3.4 Communication between Frontend and Backend

The Flask backend serves as a middleman between the application frontend and the data we have stored. The frontend uses the API to request statistics or predictions, which are then returned using JSON file. Typically the responses consist of arrays of data points, which can be plotted by the frontend's graphics libraries. Since we had separate repositories for frontend and backend, as well as separate groups of people working on them at any given time, we needed to document the API calls so that there was no confusion on either ends.

Show Prophet prediction on destination

Returns json data of Prophet prediction of a given destination for a given number of weeks.

- **URL**
`/prediction/prophet/destination/:destination/:prediction_period`
- **Method:**
GET
- **URL Params**
Required:
destination = [string]
prediction_period = [int]
- **Call**
`get_prophet_destination_prediction(destination, prediction_period)`

Figure 10: Documenting an API to support a feature

3.4.1 Representational State Transfer (REST)

Flask routing has been used to provide RESTful web services so that the frontend could request to access the previously mentioned JSON files using a uniform and predefined set of stateless operations. Following this model, the backend would provide a route for getting the top 5 most popular travel destinations as follows:

```
@app.route('/popular/<int:month>')
def get_popular(month):
    ...
    return jsonify(month_data)
```

On the other hand, the frontend would request for getting the top 5 most popular travel destinations as follows:


```

public getTopDestinationsInMonth(monthNo: number): Observable<JSON>{
  return this.http.get(AppSettings.API_ENDPOINT +
    '/popular/' + monthNo).map(res => res.json());
}

```

By using REST, we improve reliability, and maintainability, by re-using components that can be managed and updated without affecting the system as a whole, even while it is running.

3.5 Time Series Analysis

The dataset that has been used throughout the project is time dependent i.e. information has been provided about travel bookings on a specific date time. Therefore, time series analysis was considered when designing the prediction model because the basic assumption of a linear regression model, that the observations are independent, does not hold in this case. Time Series Models are univariate, the variable being the time period.

There are a number of **pattern components** that we aim to decompose a time series:

- *Trend*: long term smooth pattern.
- *Seasonal*: pattern appears after a regular interval.
- *Residual*: remainder of the dataset after removing the trend and seasonal patterns. This pattern is correlated to *White Noise*. White Noise is a random signal having equal intensity at different frequencies and it represents the error terms of the time series: ε_t , where $E(\varepsilon_t) = 0$ and $\sigma^2(\varepsilon_t)$ is constant.

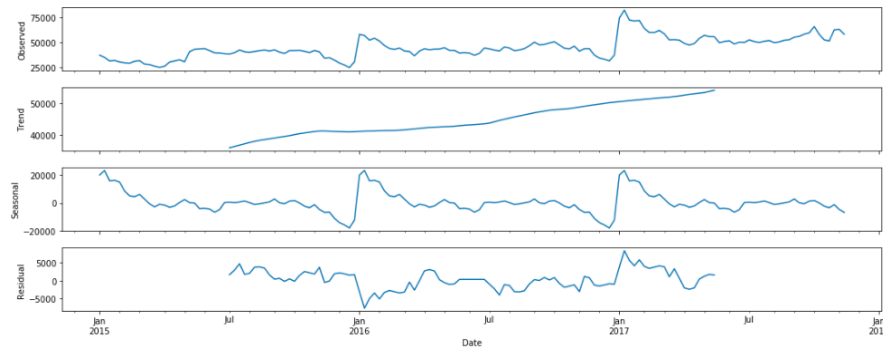


Figure 11: Time Series Decomposition

3.5.1 Seasonal ARIMA

What is ARIMA?

ARIMA is a univariate prediction model for Time Series analysis and forecasting. ARIMA is an acronym that stands for AutoRegressive Integrated Moving Average. It is a generalization of the simpler AutoRegressive Moving Average and adds the notion of integration.

I **AR(p)** : *Autoregression*. A model that depends on its own past values, where p is number of past values taken.

$$X_t = \beta_0 + \varepsilon_t + \sum_{i=1}^p \beta_i X_{t-i}$$

II **I(d)** : *Integrated*. The use of differencing of raw observations (e.g. subtracting an observation from an observation at the previous time step) in order to make the time series stationary.

III **MA(q)** : *Moving Average*. A model that uses the dependency between an observation and a residual error that follow a white noise process from a moving average model applied to lagged observations.

$$X_t = \mu + \varepsilon_t + \sum_{i=1}^q \phi_i \varepsilon_{t-i}$$

The **ARMA(p,q)** model refers to the model with p autoregressive terms and q moving-average terms.

$$X_t = \beta_0 + \varepsilon_t + \sum_{i=1}^p \beta_i X_{t-i} + \sum_{i=1}^q \phi_i \varepsilon_{t-i}$$

Visualize the Time Series

The data was first visualised to understand what type of model should be used. Is there an overall trend in your data that you should be aware of? Does the data show any seasonal trends? This is important when deciding which type of model to use. If there isn't a seasonal trend in your data, then you can just use a regular ARIMA model instead of the Seasonal one. From the Figures 10 and 11, it can be observed that there is a clear seasonality pattern in the

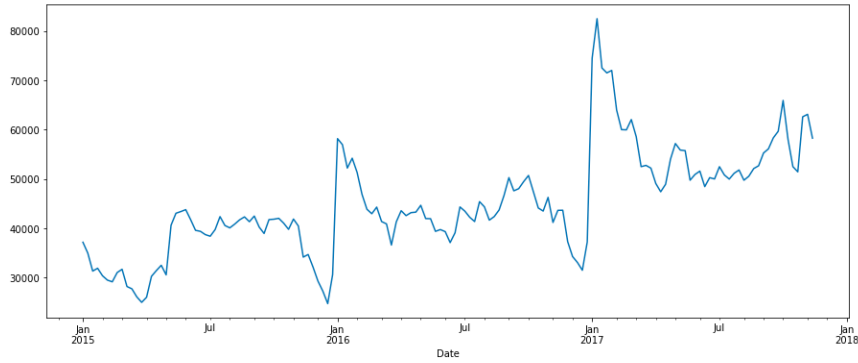


Figure 12: Number of bookings made through a travel agency over a 3 years time period dataset which leads to using Seasonal ARIMA model.

Stationarise the Time Series [10]

When running a linear regression, the assumption is that all of the observations are all independent of each other. In a time series, however, observations are time dependent. By making

the data stationary, regression techniques can be applied to this time dependent variable. A data is stationary if the mean, the variance and the covariance of the i^{th} term and the $(i + m)^{th}$ term of the series are not functions of time. In order to do a *Stationarity Test* on the data we perform the **Dickey-Fuller** method.

```
#Determining rolling statistics
rolmean = pd.rolling_mean(timeseries, window=52)
rolstd = pd.rolling_std(timeseries, window=52)

#Plot rolling statistics:
orig = plt.plot(timeseries, color='blue',label='Original')
mean = plt.plot(rolmean, color='red', label='Rolling Mean')
std = plt.plot(rolstd, color='black', label = 'Rolling Std')
plt.legend(loc='best')
plt.title('Rolling Mean & Standard Deviation')
plt.show(block=False)

#Perform Dickey-Fuller test:
print('Results of Dickey-Fuller Test:')
dfctest = adfuller(timeseries, autolag='AIC')
dfcoutput = pd.Series(dfctest[0:4], index=['Test Statistic','p-value','#Lags Used','Number of C
for key,value in dfctest[4].items():
    dfcoutput['Critical Value (%s)'%key] = value
print(dfcoutput)
```

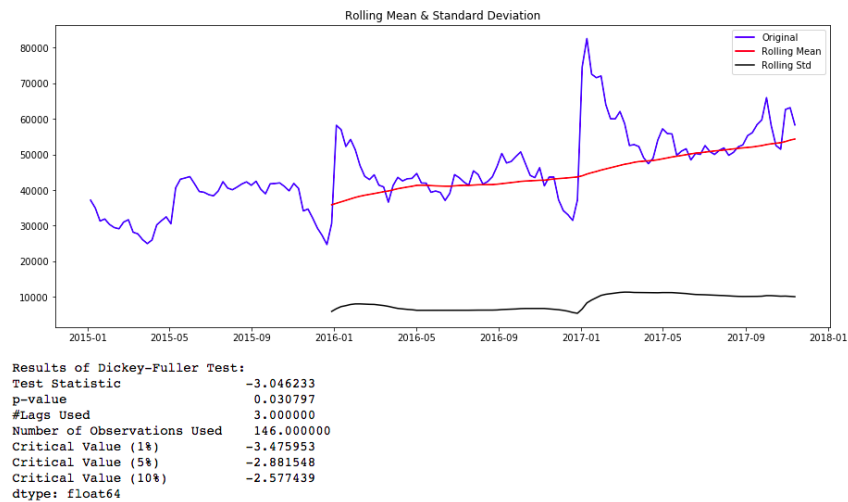


Figure 13: Rolling mean & Standard deviation & Dickey-Fuller test

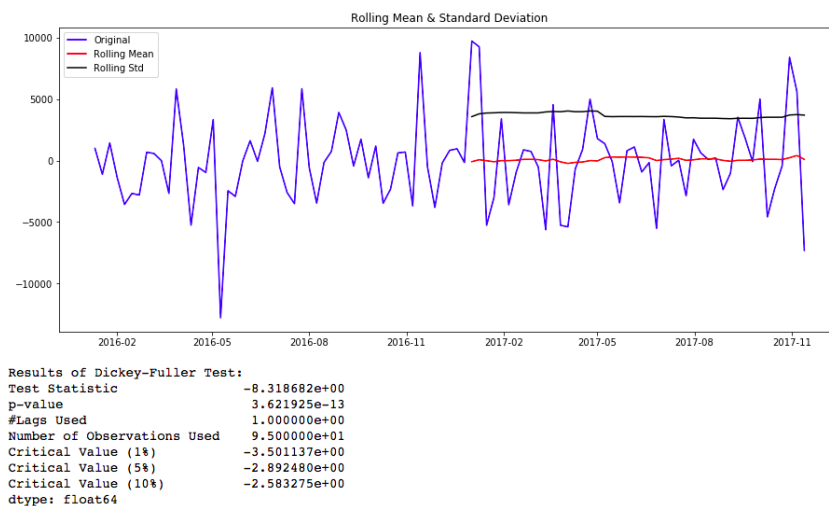
Figure 12 is showing a non-stationary time series and therefore we need to perform the first differencing and the seasonal first differencing in order to make it stationary.

Test Stationarity after First Seasonal Difference

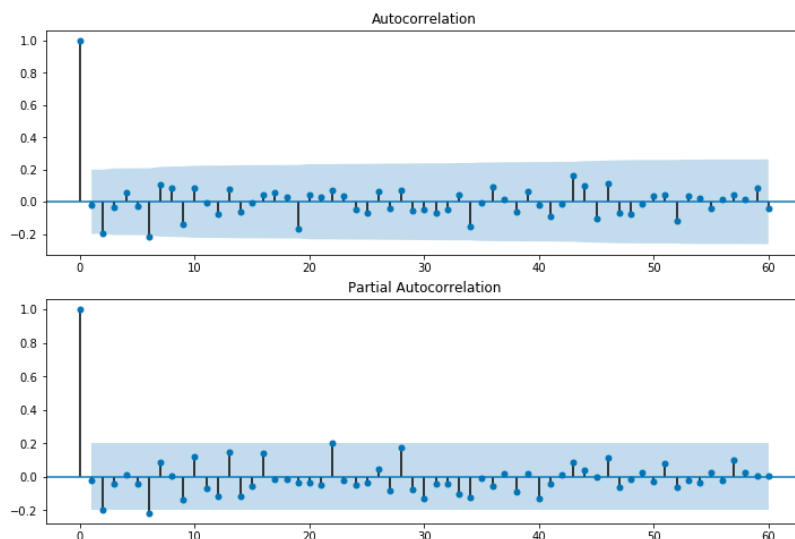
We use the time period of 52 weeks which is equivalent to one year.

```
ts_seasonal = ts - ts.shift(52)
ts_seasonal_first_diff = ts_seasonal - ts_seasonal.shift()
```

And then plot the stationarity test.



Find the optimal parameters using ACF and PACF



Observing the figure above and making use of the rules for identifying ARIMA models [11] we can observe that the autocorrelation function (ACF) of the differenced series displays a sharp cutoff and/or the lag-1 autocorrelation is negative—i.e., if the series appears slightly “overdifferenced”—then consider adding an MA term to the model. The lag beyond which the ACF cuts off is the indicated number of MA terms. Therefore we shall consider $q=2$ for the ARIMA(p,d,q) model.

It is possible for an AR term and an MA term to cancel each other’s effects so we need to avoid using multiple AR and MA terms in the same model. Respecting that we need to have $p=0$.

Another rule states that if the series has a strong and consistent seasonal pattern, then you must use an order of seasonal differencing (otherwise the model assumes that the seasonal pattern will fade away over time). However, you should never use more than one order of seasonal differencing or more than 2 orders of total differencing (seasonal + nonseasonal). Therefore we will have $d=1$ and $D=1$.

Build Model

First we remove the last 15 weeks from the dataset.

```
prediction_period = 15
train = ts[:-prediction_period]
test = ts[-prediction_period:]
```

Now we can feed the SARIMAX model with the parameters that we have deducted from the rules $((0,1,2) \times (0,1,0,52))$.

```
mod = sm.tsa.statespace.SARIMAX(train, trend='n', order=(0,1,2), seasonal_order=(0,1,0,52))
results = mod.fit(dispatch=0)
```

```

Statespace Model Results
=====
Dep. Variable:          Nr_Of_Bookings      No. Observations:          135
Model:                SARIMAX(0, 1, 2)x(0, 1, 0, 52)  Log Likelihood            -785.032
Date:                  Sun, 07 Jan 2018              AIC                      1576.065
Time:                  23:08:39                      BIC                      1584.781
Sample:                01-05-2015                    HQIC                     1579.607
                    - 07-31-2017
Covariance Type:                opg
=====
              coef      std err          z      P>|z|      [0.025      0.975]
-----
ma.L1          -0.0401         0.094      -0.426      0.670      -0.225         0.145
ma.L2          -0.1217         0.131      -0.930      0.353      -0.378         0.135
sigma2         1.288e+07    1.53e+06       8.435      0.000    9.89e+06    1.59e+07
=====
Ljung-Box (Q):                27.16    Jarque-Bera (JB):                17.13
Prob(Q):                      0.94    Prob(JB):                      0.00
Heteroskedasticity (H):        0.65    Skew:                          0.16
Prob(H) (two-sided):           0.27    Kurtosis:                      5.21
=====
```

Make predictions

Calculating the Root Mean Square Error of the predictions compared to the real data we get

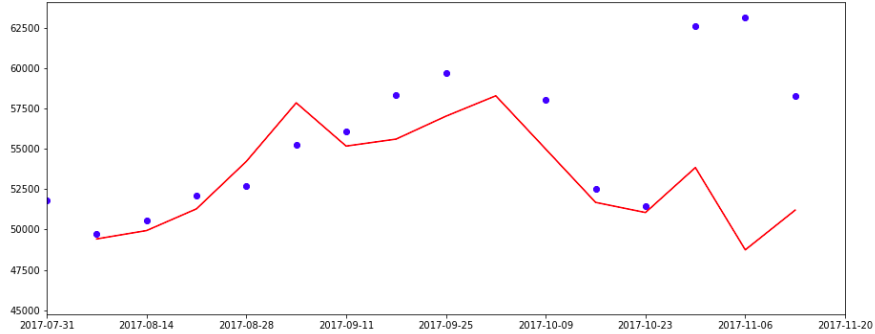


Figure 14: Prediction over 15 weeks

5342.0857368 which is a relatively small value compared to the weekly number of bookings for this travel agency. We can also observe from Figure 13 that the peaks and waves are predicted with high accuracy.

Using the observations from Figure 14 we can say that the SARIMAX model is respecting the seasonality and the trend of the time series over a long period of time.



Figure 15: Prediction over one year

3.5.2 Prophet

Prophet is a forecasting procedure developed by Facebook. We have used it as one of the prediction engines to fit time series data with a weekly seasonality and derive corresponding predictions in the nearest future. The reasons we had chosen to use Prophet were not only because it is automated, easy to use and is compatible with our pre-existing Python code base, but also due to its robustness to missing data, shifts in the trend and large outliers. A general procedure of how Prophet is used is given as follows:

- **Input** - Prophet requires a dataframe with two columns: `ds` and `y` as its only input. The `ds` column must contain a date or datetime, while the `y` column must be numeric and represents the entity we wish to predict. In our implementation, we use the first available day of the week in `ds` column, while we use weekly sum of the bookings for a particular agent or a destination depending on what is being predicted. The reason behind doing this has been provided in the next step. For example

```
dataframe = getAgencyBookingData('agent', travel_agency,
                                  'ds', 'y')
```

- **Model** - A model is an instantiation of a Prophet object and is an essential component throughout the procedure. So, the next step is to create one and fit the model with the dataframe that we had previously obtained. This is the step where you can update the seasonality in data. In our implementation, we have turned off weekly seasonality since only the data for week days was given to us which resulted in continuous spikes in the graph. Consequently, we only used one data point which represents a week's data as input in the first step.

```
model = Prophet(weekly_seasonality=False)
model.fit(dataframe)
```

- **Future** - Future as the name suggests is used to extend the provided dataframe into the future. It includes all the dates from the history so that we will see the model fit as well. Since we are using 1 data point per week, we only look 52 weeks into the future representing a year if we want to predict future trends. On the other hand, we also use this feature to test the efficiency of Prophet by asking it to predict the data which we already have.

```
future = model.make_future_dataframe(periods = period_val,
                                     freq = 'W')
```

- **Forecast** - Forecast uses a predict method on each row future to assign a predicted value, \hat{y} . Essentially, the forecast object is a new dataframe that includes a column \hat{y} along with other columns representing the uncertainty intervals. For our implementation, we have only focused on the \hat{y} column.

```
forecast = model.predict(future)
```

3.6 Cloud Storage and Azure

In order to query the vast amounts of data, it was necessary to upload it to a cloud storage account, which can be accessed by Hadoop clusters for processing. Hadoop is one of the most popular software frameworks which uses the MapReduce technique in order to process very large

datasets on distributed clusters. Microsoft Azure services are used throughout the project for storing and processing the data. The data is uploaded to an azure storage account [12], which can be linked to any HDInsight (Hadoop) cluster [13] created. This is the process for querying the data. A Hadoop cluster is provisioned with the Hadoop File System set to the storage account containing the data. Various queries are then run on the data using PigLatin [14] so that the query processing can be distributed over the cluster as necessary, speeding up computation. The results of these queries are then stored in the storage account so that they may be requested by the application backend.

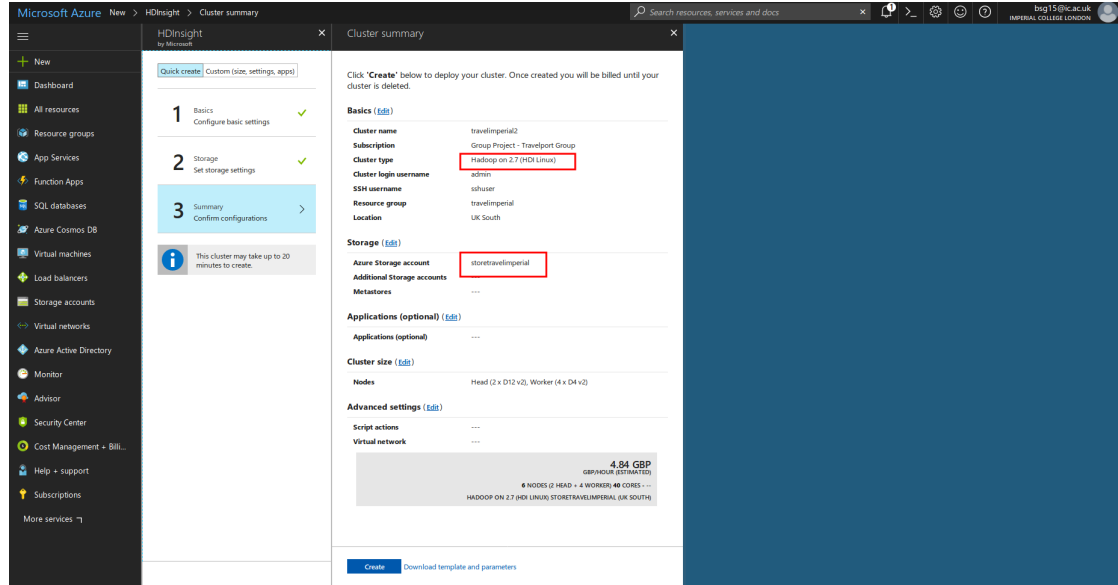


Figure 16: Provisioning a Hadoop HDInsight cluster with the storage space attached

Currently, there are PigLatin scripts stored on the storage account for a range of requests which are necessary for the presentation of the data on the frontend. For example, one script is used to gather the top five airport destinations per month (excluding UK destinations). Each script is run once and the resulting CSV file is downloaded to the Back-end whenever they are needed. This is made possible by the fact that there are only a finite number of queries supported by the frontend. The benefit of saving the results in this way is that the user does not need to wait for the potentially lengthy query to take place. For some queries, the backend will request a file from the storage space that is more general than the query needed for the front-end, and the CSV file will then be reduced further before the results are passed on to the front-end.

If the data being processed were to be updated regularly, for example if new data were to be added daily, then it would be possible to automate a cluster to run the scripts at regular intervals to keep the query results up to date. For the purposes of this project however, the data is unchanging and there is no such automation in place. Instead a Hadoop HDInsight cluster has been created whenever more computation was needed. It should be noted that while the storage account which holds the data, pig scripts and result files exists permanently, the HDInsight clusters which perform the computation are created as and when they are required. The creation and deletion of clusters could also be automated if necessary, but this was not

needed for the unchanging dataset used for this project.

3.7 Communication between Backend and Azure

When the backend requires use of the data, it communicates with the Azure storage container and downloads a CSV file. This data can then be processed and used to return the requested data to the front-end. In order to download a file from the storage account, the function `getfile` in `util.py`. This function uses a `BlockBlobService` [15] to download a file from the storage space.

The function:

```
block_blob_service.get_blob_to_path('travelimperial',  
                                     filename, output)
```

retrieves a file matching filename from the storage space, named travelimperial.

When creating the `BlockBlobService` object, login details to the storage space must be provided. These are imported from constants in the file `loginDetail.js`. For the development of the project the login details remained the same, however if the project were to be shipped to various different agencies, we would need to ensure that they each had their own login details which would allow restricted access to their individual data. The code for importing the login details is shown below:

```
from loginDetail import *  
block_blob_service = BlockBlobService(account_name=STORE_ACCOUNT,  
                                       account_key=STORE_ACCOUNT_KEY)
```

After downloading a file the CSV is converted to a python dictionary so that the data can be easily passed onto to functions which perform prediction, or process the data in some way before returning results to our frontend.

Initially the process for communicating with Azure also allowed the backend to send a request to start a Pig job running on a HDInsight cluster, so that it would retrieve up to date results to the query. This method was replaced with the current one, when it became clear that the run time for queries on the backend would be too long to keep the application responsive.

3.8 Continuous Integration (CI)

In order to maintain a good Software Development practice, we decided to make use of two distinct CI servers for our frontend and backend repositories respectively. In regards to choosing one CI provider amongst many available online, we went with Gitlab CI [16] since it is not only integrated to GitLab which we have been using to maintain both of our repositories, but also provides faster results due to parallel builds besides being free to use.

3.8.1 Frontend

For our frontend CI implementation, we have separated the CI load into two stages, namely test and build. We incorporate the latest node docker image [17] which is used to install required node packages and build the frontend implementation, concluding our build stage. For the later stage, we make use of PhantomJS [18], a headless WebKit scriptable with a JavaScript API, to perform various unit tests.

























 [NS] Split the CI configuration file into build and test stages. Nimesh Subedi committed a month ago	 f7d9e3e2 	Browse Files
 Merge branch 'experimentalCompetitors' Nimesh Subedi committed a month ago	 2e3425f7 	Browse Files
17 Nov, 2017 11 commits		
 Merge remote-tracking branch 'origin/master' Thomas Price committed a month ago	 2f37daa4 	Browse Files
 agency competition component fully integrated and documented Thomas Price committed a month ago	01ea9466 	Browse Files
 Update .gitlab-ci.yml Thomas Price committed a month ago	 9bc115cb 	Browse Files
 Update .gitlab-ci.yml Thomas Price committed a month ago	 11635fcc 	Browse Files
 fixed so tests actually pass Thomas Price committed a month ago	 b7e156ce 	Browse Files

Figure 17: Use of Continuous Integration

 **passed** Pipeline #45564 triggered about 17 hours ago by  Thomas Price

improved the UI for destination predictions

2 jobs from **master** in 5 minutes 1 second (queued for 2 seconds)

 ed487c15 

Pipeline Jobs 2

Build **Test**





 build   test 

Figure 18: Build and Tests stages

3.8.2 Backend

For our backend CI implementation, we follow similar approach to divide the load into two stages. For our build stage, we had initially incorporated a python docker image [19] to install required modules and build the backend. It would then move onto running the tests. However, the CI server started failing in the build stage eventually due to the addition of Prophet procedure in our implementation. After researching the causes of build failure, we found that Prophet needs 4GB of memory for installation which wasn't supported by the CI server provided to us. In an attempt to fix the issue, we tried looking for a new CI server with higher memory, only to find out that it would not be possible. After bringing this up in our checkpoint as well as Software Engineering meeting, we were advised to isolate the code that doesn't make use of Prophet and run tests on that. However, it was not possible to do that, at least not in a straight forward way, and since we had spent more time than we had allocated to fix it, we decided that we should come back to it at a later stage.

4 Technical Challenges

4.1 Angular 2

There was very little technical challenge that came from using Angular due to familiarity from last years Webapps group project. The largest hurdle was working around the bug in ng2-charts [20] that prevented the chart from displaying updated data due to the lack of two way data binding. The work around was to find each chart’s JavaScript object and update the values of the axes and labels manually as well as updating the Angular model of the chart, or to simply destroy the chart and create a new one in its place.

4.2 Prophet

As with any new technology, the first challenge is always to learn how to use it. Prophet not being excluded from this took some time in getting to grips with. Though Prophet seemed really convenient to work with at the beginning, we later realised it was not be able to support our later demands. Though Prophet is really useful for constructing prediction graphs and such, it does not provide the user with any of the underlying statistics other than the uncertainty interval related to a predicted data point restricting us with what we could do with the data. Furthermore, since Prophet relies on time series data for at least a year, we were unable to work with any datasets which only had a limited amount of data points, constraining us from utilizing all the data that had been kindly provided to us.

On top of that, Prophet has caused a lot of time being spent on fixing the backend CI server due to it consuming a high amount of memory during installation which is unsupported by our current CI servers. This lead to any code that had used Prophet from not being able to be tested on the server.

4.3 Microsoft Azure

Microsoft Azure took some effort to get going, as there was plenty of terminology to learn, tools to install, and official tutorials to follow, before being able to connect it with the rest of the project. Once it was set up, working with the clusters and storage system proved to be very straightforward.

The main advantage of Azure was the ability to independently scale the performance of each component, i.e. storage container capacity, cluster RAM and cores. This was useful when we needed to test and improve our Pig scripts. Also, being able to easily create and delete an Azure VM for the initial transfer of the zipped dataset took much less time than having to directly send the dataset to the storage. It also had additional tools for automating daily cluster creation, job execution, and deletion, which was very important functionality for allowing us to update our dataset for future work.

The biggest disadvantage of Azure was that there were costs associated with each Azure component that we used. Whilst storage cost was negligible, having clusters running during the entire development period would have been very expensive. To avoid spending thousands of pounds, we had to start and stop clusters as required. This resulted in having to plan Pig script development in advance in order to avoid the fifteen minute cluster creation time.

The tradeoff between cost and job time is mostly a concern for script debugging and improvement. As each script would eventually be scheduled to run once per day, and updating our data depends on script time, the cost to keep the processing time the same scales with the features that TPA provides. To reduce this cost in the future, script debugging could have perhaps be first done with test data on a local machine cluster.

4.4 Hadoop

When we wrote our first Pig scripts we realised that the queries were particularly slow, due to the fact that not much optimization is done behind the scenes by Pig. Operations in Pig were translated to Map-Reduce steps, and then executed on the cluster using our dataset. We quickly learned to filter out irrelevant data by using projections and selections earlier on in the scripts. Learning how to minimise the amount of data in each Map-Reduce phase was vital in achieving adequate performance. By optimising our scripts, we found a huge speed increase, e.g. roughly 22s for a query on a 4GB file as opposed to a few minutes.

In addition to this, many of the queries relied on identical intermediate results, so we simply removed the duplicated operations by storing those intermediate results into our filesystem.

5 Risks and Mitigations

5.1 Learning Curve

At the beginning of the project, most of us didn't have any experience working with Big Data or Machine Learning. The fact that this project also overlapped with our courses made it even harder to learn due to the limited amount of time we had. In order to speed up the process, we used online courses on how to process Big Data with Hadoop in Azure HDInsight [21], how to work with clusters, and did some research about time series algorithms in order to understand the data and to do some meaningful forecasting for Travelport. It was also tricky and non-trivial to learn how to optimize the parameters required by the algorithms, and set a good error measurements for the predictions.

5.2 Cloud Services

The main concerns about working with an extremely large dataset and having to store it on the Cloud were related to security and unauthorized access to the data. In order to combat these issues we chose to use a large organization like Microsoft to store and protect our data in a safe way.

5.3 External Data

Dealing with data that belonged to somebody else made the project challenging, as it imposed a lot of responsibility on our side when having to deal with such important information. We faced a lot of challenges during the project that were solved with good collaboration between us and the company. One of the main issues that was encountered when building our time series algorithms was the assumption of regular reporting times (large, densely populated data sets), which unfortunately didn't happen. For example there were a lot of data points missing for December, which we think may have been caused by the fact that travel agencies are closed during that period of the year.

5.4 Pivoting the Objectives

The project initially started with the objective of developing predictions in pricing, market sales share and market sales growth aspects of the travel industry. This was later confirmed at our initial meeting with Guido Verweij, head of global strategy at Travelport.

During the meeting, he also advised us that the data that we needed for the project was still being processed due to timing issues induced by anonymity requirements. Since we could not have started the development without the data schema, we decided to use this time to research technologies that would assist us in this project and learn how to use the ones that seemed most useful. However, it was later clarified that since we were not employed by Travelport and hence had not signed any Non-Disclosure Agreement (NDA), it would not be possible to provide us with the data regarding sales and other financial aspects of the company. This led to us discarding our preparations for predictions based on sales and focusing on the remaining two objectives.

After that, we first decided to work on predicting number of bookings made by a given travel agent extracted from a column which was misunderstood as a collection of tokenised travel agencies. We had managed to get the predictions working on both ARIMA and Prophet and had also begun looking into market shares prediction using the same column. However, it was later explained to us that the column that we had been using actually represented the PNR vendor codes and could not be used in any way to identify the corresponding agent. This caused a lot of uncertainty in what we could do with the development so far and where we would go next since we were at the final checkpoint and our entire implementation was based on the wrong column.

Fortunately, this issue was solved with heavy support from Beyond Analysis as they provided us with a different data set consisting of tokenised International Air Transport Association (IATA) codes used by travel agents while making bookings. Although an agent could use multiple IATA codes to book flights for customers and we did not have enough data to link an agent with their IATA codes, we created a framework that could be used to find bookings corresponding to IATA codes which could later be replaced with the agents. With this in mind, we also created another feature that allows prediction on the number of bookings to a given destination as well as the market share predictions.

6 Evaluation

6.1 Efficiency Testing

While there are tests to validate the integrity of the code, we have also used our framework to test the efficiency of its predictions. The predictions are made on given data for a given period of time. In order to test how good a chosen algorithm is for predicting using some data, we simply remove a certain number of data points from the data and predict the same number of data points. Then it's possible to plot the predicted values from the algorithm against the real data, which can be used to verify the effectiveness of the algorithm visually.

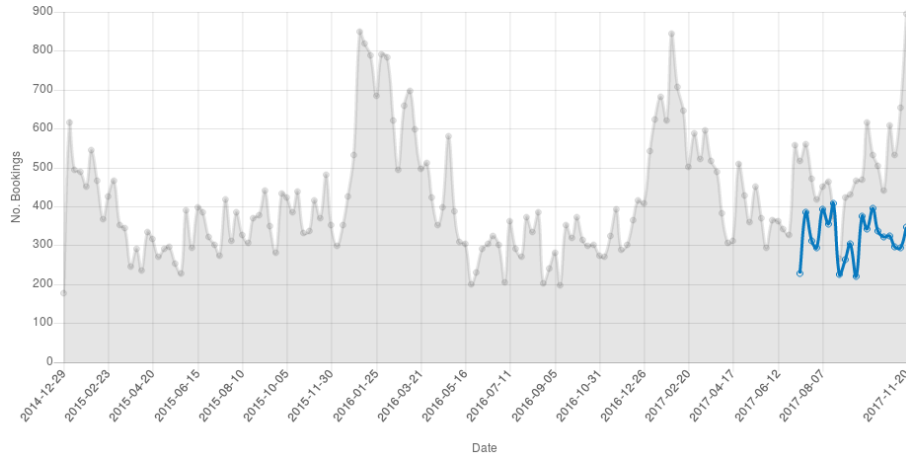


Figure 19: Efficiency Testing in ARIMA

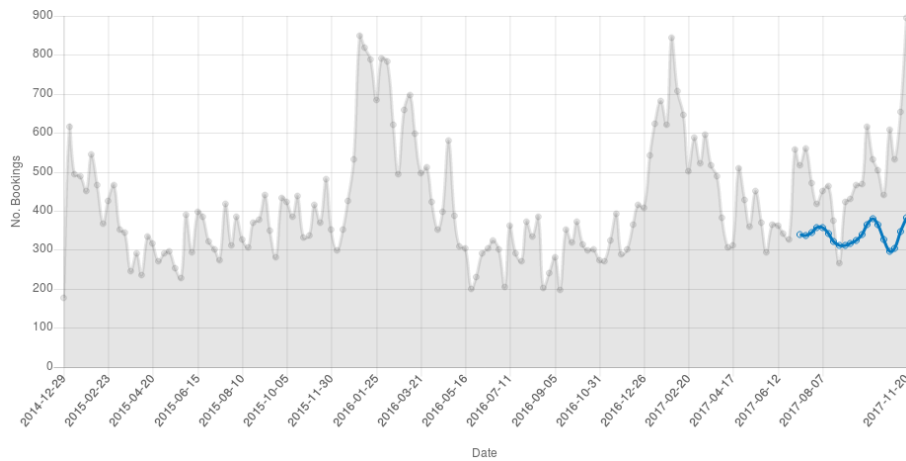


Figure 20: Efficiency testing in Prophet

6.2 ARIMA vs Prophet

While ARIMA and Prophet have their own advantages as well as disadvantages as previously mentioned in sections 3 and 4, one does not always perform better than the other in all situations. Using the efficiency tests mentioned above, we saw that during cases where higher volumes of data was available, ARIMA was able to predict the trends very closely while Prophet was only able to produce a general trend.

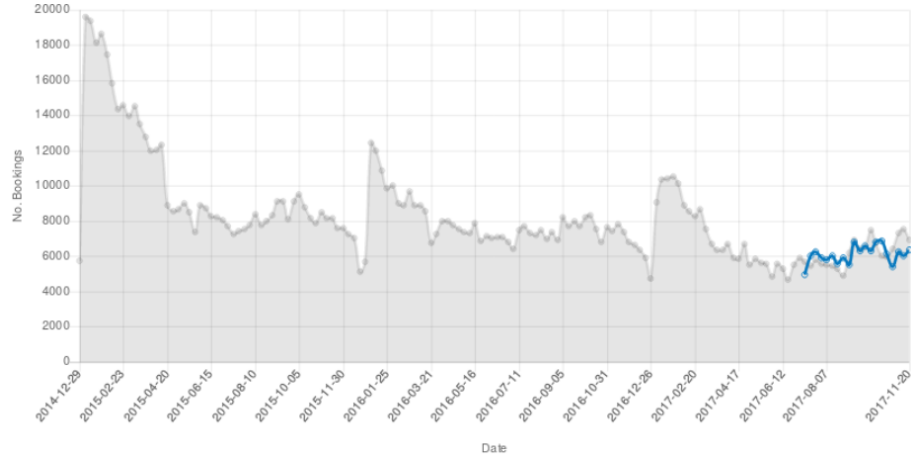


Figure 21: Efficiency Testing in ARIMA with high volumes of data

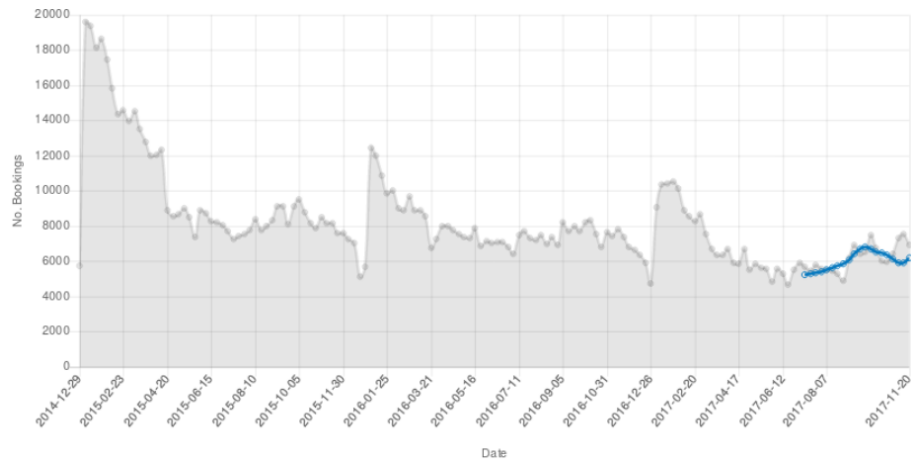


Figure 22: Efficiency testing in Prophet with high volumes of data

On the other hand, Prophet seems to have the upper hand when there is a low amount of data.

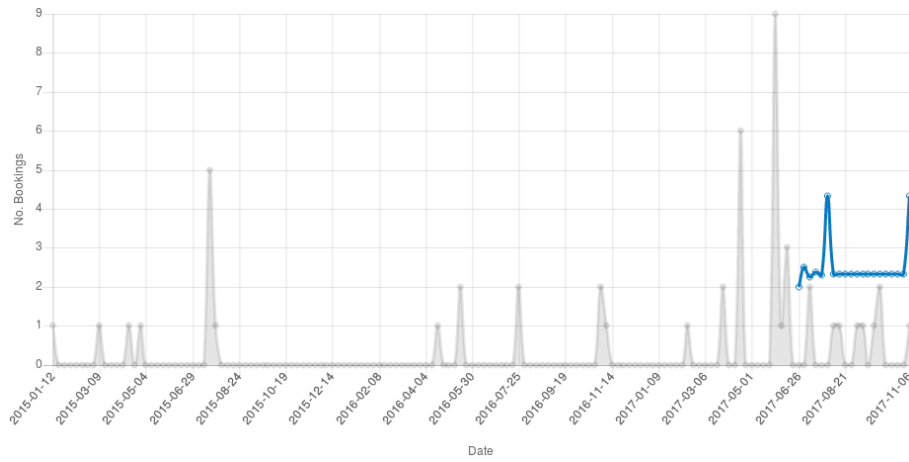


Figure 23: Efficiency Testing in ARIMA with low volumes of data

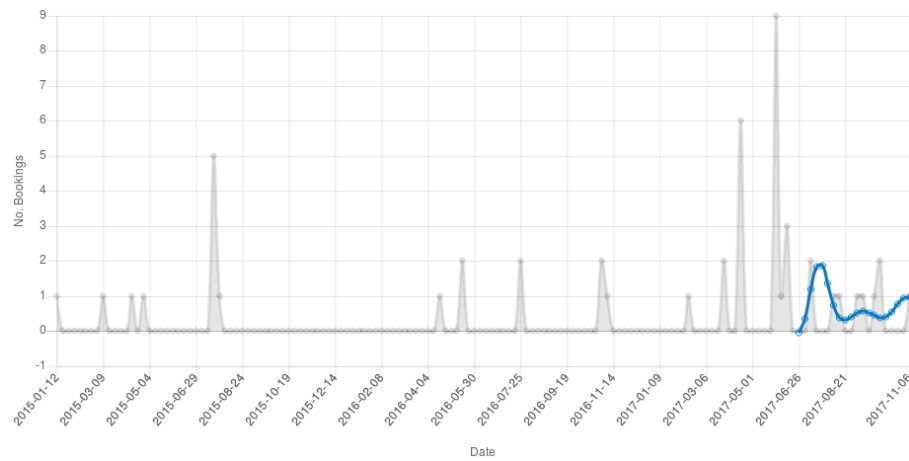


Figure 24: Efficiency testing in Prophet with low volumes of data

Through various testings, we found that Prophet needs relatively fewer adjustments and fewer amount of data to produce a general trend. On the contrary, ARIMA not only needs a lot of adjustments but also a larger dataset to produce a trend, but it is able to produce a close fitting trend when its requirements are met.

7 Conclusion

Working on TPA was an exciting big data opportunity and naturally presented many challenges, but working through each was highly rewarding.

We hope that this project will be useful to Travelport after we hand it over, and that they will be able to build on what we have created in their own work.

7.1 Deliverables

Our submission contains three main components:

- Prediction for an IATA which plots the future number of bookings through an IATA for overall destinations.
- Prediction to a destination which plots the future number of bookings to a given destination through overall IATAs.
- Market share for an IATA which displays a chart showing the proportion of bookings through an IATA out of the total, for a given destination.

7.2 What we learned

It is important to have designed and plan the architecture of an application before beginning to create it. Due to time constraints from the application pivoting early on it was not possible to take a couple of days to plan out our application for the first checkpoint. This led to a large amount of code duplication in both the frontend and predictions section which had a high refactoring cost in later checkpoints. The maintainability of our code base was necessary for developing new features for each checkpoint.

7.3 What we would be different if we had to do it again

The project as a whole proved quite challenging even besides the non-technical real world obstacles, and with hindsight, there was plenty of room for improvement. If we were to do it again, we would surely have approached the implementation differently to avoid them.

Starting at the very beginning, we would have chosen to use AWS cloud service instead of Microsoft Azure since it would have been easier to transfer the required files directly from the data holders rather than following the long-winded process of collecting the data from them and manually uploading them one-by-one.

Another aspect of our implementation that we would have changed was the framework we used for the frontend. Whilst Angular2 provides widely used libraries, such as ng2-charts, to allow dynamic presentation of data in the form of various charts and such, we found that it was unable to facilitate our demands during later stages of development, leading us to spend a lot of time in finding workarounds.

8 Future Improvements

8.1 Booking Heatmap

A potentially useful feature would be to display the trends on a map consisting of cities around the world, showing which would be the busiest around a certain period of the year. For example you could look in England, see which cities would be the most popular to fly to, and also the number of flights around Christmas time.

Whilst you could see this kind of information using simple queries, it could be helpful for agencies to visualise the correlation between cities. For example, a heat map would allow agencies to connect geographical locations and easily find groups of interlinked cities for special occasions (take for example a very big event like a football world cup).

8.2 Exogenous regressors [1]

To improve our model for bookings forecasting, we can add external events as variables. These events should influence the prediction with a given power of impact. In order to add these type of variables we need to modify the ARIMA model to accept exogenous regressors.

An ARIMA model can be considered as a special type of regression model—in which the dependent variable has been stationarized and the independent variables are all lags of the dependent variable and/or lags of the errors—so it is straightforward in principle to extend an ARIMA model to incorporate information provided by leading indicators and other exogenous variables: you simply add one or more regressors to the forecasting equation.

8.3 Other Time Series Algorithms

We noticed that ARIMA and Prophet are not perfect and perform well in certain situations, with certain data and parameters. It would be a good idea to try and use other time series algorithms. We will now explain a few alternatives and the basic ideas behind them, in order to be able to compare them with what we have used so far. The first example would be Loess Regression, which splits the data into a seasonal component, a trend and a remainder. The main idea is to try and fit local polynomial models and then merge the results together. First, you would have to define a window width α (generally $\approx 5\%$) and use α nearest neighbours that are within the window to do local regression. Then you would choose a weight function which gives higher weight to the nearer points to the center. The most used one is:

$$W(u) = \begin{cases} (1 - |u|^3)^3 & |u| \leq 1 \\ 0 & |u| > 1 \end{cases}$$

Finally you do local regression using Taylor polynomial and the weights defined at previous step. At the end you should have no trend in the stationary component you get from Loess and check for second-order stationarity conditions:

- I Constant mean (Use Box-plots to check that)
- II Constant variance
- III Auto covariance does not depend on time

Another potential useful algorithm is Moving Average Smoothing which finds a trend by averaging over 1 cycle length. The idea is very simple, for every point you do an average of the last 10 points in the time series, in order to find the correct position. It is fine grained, looks only at the previous cycle. The downsides are that you need to know the precise cycle length and trend must be smooth, otherwise the prediction will be meaningless.

A more clever approach can be found in the STL package, which combines the algorithms mentioned above. The idea is simple:

- I Use Loess to find a general trend T
- II Use Moving Average Smoothing to find a more fined grained trend C
- III Subtract both to find the seasonal component $S = X - T - C$ (where X is the initial data)
- IV Smooth the total trend $T + C$ with Loess to get total trend V
- V Remainder is $R = X - S - V$

8.4 Statistics

A travel agent will definitely find our project useful as it will forecast the expected general market trend in the future. However, a travel service provider might want to see other statistical data besides the graph which would clarify how accurate the trend is likely to be and the ratio of impact various factors have on the trend, so that they could make appropriate business decisions.

9 License

9.1 Component Licenses

The components that we have used throughout the project and their corresponding licenses are as follows:

- Docker Images - Apache License
- Gitlab CE - MIT License
- PhantomJS - BSD 3-clause License
- Prophet - BSD License
- Pandas - BSD License
- Flask - BSD 3-clause
- Flask-CORS - MIT License
- Requests - Apache 2.0
- Werkzeug - BSD License
- Azure Storage Blob - Apache 2.0
- Matplotlib - based on PSF (BSD compatible)
- Statsmodels - BSD 3-clause License
- Sci-kit learn - BSD License
- Angular - MIT License
- ng2-charts - MIT License

9.2 TPA License

We have used GNU GPLv3 license for our project since all our components use either some form of BSD license, MIT license or Apache license. This ensures that our licensing terms are compatible with all the components that we have used throughout this project.

Acknowledgements

We would like to thank Dr. Anandha Gopalan for his expert advice and encouragement throughout this project. He has always been helpful in suggesting solutions whenever we got stuck with an unforeseen issue. This project would have been impossible without his aid in keeping the clusters running as well.

We would also like to convey our deepest appreciation towards Guido Verweij and Eytan Goldfinger from Travelport as well as Anthony Beresford, Andrew Edge and Alice Cheshire from Beyond Analysis for their continuous support and guidance throughout the project. We would not have been able to make much progress without your help.

10 Bibliography

References

- [1] ARIMA models with regressors. <https://people.duke.edu/~rnau/arimreg.htm>.
- [2] Hopper. Hopper (Mobile App). <http://www.hopper.com/>, 2017. [Online; accessed 04-January-2018].
- [3] Facebook. Prophet. <https://facebook.github.io/prophet/>, 2017. [Online; accessed 04-January-2018].
- [4] Google. Angular. <https://angular.io/>, 2010-2017. [Online; accessed 04-January-2018].
- [5] Facebook. ReactJS. <https://reactjs.org/>, 2017. [Online; accessed 04-January-2018].
- [6] Google. Angular CLI. <https://cli.angular.io/>, 2010-2017. [Online; accessed 04-January-2018].
- [7] Simon Brunel. ChartJS. <http://www.chartjs.org/>, 2017. [Online; accessed 04-January-2018].
- [8] valor software. ng2-charts. <https://github.com/valor-software/ng2-charts>, 2017. [Online; accessed 04-January-2018].
- [9] MIT. Karma. <http://karma-runner.github.io/0.13/index.html>, 2017. [Online; accessed 04-January-2018].
- [10] Sean Abu. Seasonal ARIMA with Python. <http://www.seanabu.com/2016/03/22/time-series-seasonal-ARIMA-model-in-python/>, 2016.
- [11] Summary of rules for identifying ARIMA models. <https://people.duke.edu/~rnau/arimrule.htm>.
- [12] Microsoft. Azure Blob Storage. <https://azure.microsoft.com/en-gb/services/storage/blobs/>, 2017. [Online; accessed 04-January-2018].
- [13] Microsoft. Azure HSI Insight Cluster. <https://azure.microsoft.com/en-us/services/hdinsight/>, 2017. [Online; accessed 04-January-2018].
- [14] Apache. PigLatin. <https://pig.apache.org/>, 2017. [Online; accessed 04-January-2018].
- [15] Microsoft. Github BlockBlobService Module. <https://azure.github.io/azure-storage-python/ref/azure.storage.blob.blockblobservice.html>, 2015. [Online; accessed 04-January-2018].
- [16] Gitlab. Gitlab CI/CD. <https://about.gitlab.com/features/gitlab-ci-cd/>, 2017. [Online; accessed 04-January-2018].
- [17] The Node.js Docker Team. Node.js Docker Image. https://hub.docker.com/_/node/, 2017. [Online; accessed 04-January-2018].
- [18] Ariya Hidayat. PhantomJS. <http://phantomjs.org/>, 2017. [Online; accessed 04-January-2018].

- [19] the Docker Community. Python Docker Image. https://hub.docker.com/_/python/, 2017. [Online; accessed 04-January-2018].
- [20] Stack overflow community. ng2-charts bug. <https://stackoverflow.com/questions/42629819/ng2-charts-update-labels-and-data>, 2017. [Online; accessed 04-January-2018].
- [21] edX. Processing Big Data with Hadoop in Azure HDInsight. <https://www.edx.org/course/processing-big-data-hadoop-azure-microsoft-dat202-1x-4>, 2018. [Online; accessed 04-January-2018].

11 Appendix

11.1 User Guide

11.1.1 Client

You must have access to an IATA code in order to access to predictions made using your Travelport data. As of writing, there is no login functionality, and you must receive the anonymised hash of the IATA code from Travelport.

The web browser requirements for full functionality of TPA are that of Angular 2, provided in the following URL:

<https://v2.angular.io/docs/ts/latest/guide/browser-support.html>

Enter your hashed IATA code when requested to view your predictions. You may navigate between predictions using the tabs at the top of the page. Predictions may take around five to ten seconds to load, and are updated daily.

11.1.2 Server

The server itself runs using Docker, so there are no server dependencies. See the README for detailed instructions. Refer to Docker's manual for how to build and run the Docker image:

<https://docs.docker.com/get-started/>

Access to the Azure Storage will not be possible as the security details are not, and will not, be provided.