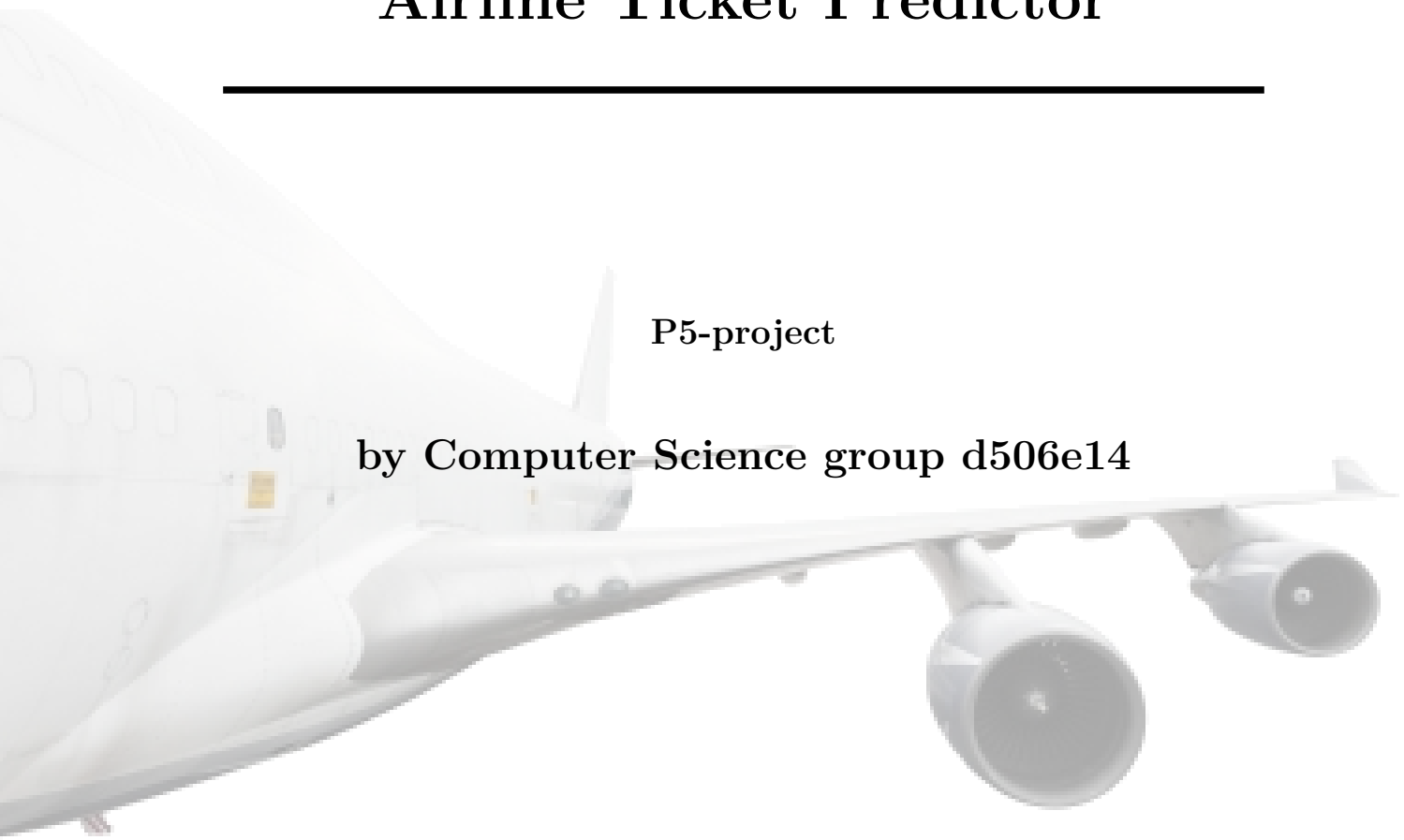
Airline Ticket Predictor



P5-project

by Computer Science group d506e14

Aalborg University
The Faculty of Engineering and Science
Institute of Computer Science
P5-project (5th semester)
Supervisor: Bin Yang
December 17, 2014

1

[1]Frontpage photo ©FreeDigitalPhotos.net

AALBORG UNIVERSITY
STUDENT REPORT

Aalborg University
The Faculty of Engineering and Science
Institute of Computer Science
Selma Lagerlöfs Vej 300
Telephone +45 99 40 99 40
Fax +45 99 40 97 98
http://www.cs.aau.dk

Title:
Airline Ticket Predictor

Subject:
Intelligent or Massively Parallel Systems

Project period:
2nd September - 19th December 2014

Project group:
d506e14

Group members:
Christian Slot
Joachim Klokkervoll
Lynge Poulsgaard
Mike Pedersen
Philip Sørensen
Samuel Nygaard Pedersen

Supervisor:
Bin Yang

Page numbers: 76

Finished: December 17, 2014

Source code:
http://ppsa.dk/uni/P5.zip

MD5 hash:
6b820e59491bdc3e629bc1125d7ca128

Synopsis:

This report documents the development of an airline ticket predictor, using classification to determine whether the current ticket price is the optimal. The project examines different kinds of prediction methods and approaches to finding the cheapest ticket price for a given flight, including regression and classification using Multilayer Perceptrons, RandomTrees, J48 (C4.5 algorithm) and Ripper implemented through the open source machine learning software, Weka. The project reaches convincing results for some flights using different prediction algorithms to determine whether to buy or to wait, however, the general tendency shows that flight prices cannot be precisely predicted using the methods explored in this project.

Aalborg, December 17, 2014

Christian Slot
Student ID: 20127075
E-mail: cslot12@student.aau.dk

Joachim Klokkervoll
Student ID: 20124277
E-mail: jklokk12@student.aau.dk

Lynge Poulsgaard
Student ID: 20124272
E-mail: lkpo12@student.aau.dk

Mike Pedersen
Student ID: 20124283
E-mail: mipede12@student.aau.dk

Philip Sørensen
Student ID: 20124293
E-mail: ppsa12@student.aau.dk

Samuel Nygaard Pedersen
Student ID: 20124278
E-mail: snpe12@student.aau.dk

Preface

This report is written by group d506e14 at Aalborg University, the members of the group is: Christian Slot, Joachim Klokkervoll, Lynge Poulsgaard, Mike Pedersen, Philip Sørensen, and Samuel Pedersen on the 5th semester of computer science.

The project subject was "Intelligent or Massively Parallel Systems" and this report tries to answer the problem of when to buy an airfare ticket to obtain the cheapest price, as the price can fluctuate. And hereby focus mostly on the "Intelligent System"-part of the project description rather than "Massively Parallel System", although there will be some parts about distributing in the project.

The report is composed in LyX, which is an overlay editor using LaTeX to compile to PDF. The programming language used in the source code is Java and therefore all listings and figures containing programming code will be displayed with Java syntax and with common Java syntax highlighting. The report is written to be understandable for people with at least a level of education equal of 5th semester of computer science or similar with basic knowledge on the topic.

In the preceding title page a link to a zip archive is attached, this archive contains all the work from the project, which is: the source code (organized as a Java project folder) and the .lyx files and all of the resources used within the report (pictures, .csv files, bibtex etc.). To ensure integrity and deadline date of the zip archive, there is also included an MD5 hash from the archive in the title page. A backup of our database and a digital copy of the report (in PDF format) will also be available in the zip archive.

The data (airfare ticket prices) was gathered for flights departing from the 1st of October 2014 to the 31st of January 2015, and data was obtained once every day starting from the 1st of October to the 18th of December, as such the amount of useful data is limited to about two months.

This report firstly introduces the overall ideas and system design, which explains the different aspects of the problem and systems parts as well as their function without going into detail on how these are implemented. Later details on the actual implementation are explained. Hereafter the system is evaluated through tests and experiments leading to a conclusion. Finally there is a chapter explaining the agile working progress to documenting how the teamwork where conducted.

The report is written using numbered citation, e.g. "The King of Spain is Juan Carlos [5]". Here [5] is referring to the fifth entry in the bibliography, on all online references a timestamp in marked at the end to tell which date it was accessed.

When "we" is mentioned it refers to the members of the project group and the overall opinion of the group. There is also a lot of different expressions in the report which might be ambiguous, therefore we explain what these expressions mean in this report here.

We would like to give special thanks to our supervisor Bin Yang for his great supervision, knowledge on the topic, input to the project, comprehensive revision of our worksheets.

Word	Explanation
Instance	A data entry regarding the predictor, this may be for training where the class attribute is given, or for prediction, where the class attribute is predicted.
IATA	International Air Transport Association.
CPH	The IATA code for Copenhagen airport, used in the report to refer to Copenhagen airport.
LHR	IATA code for London Heathrow airport.
AMS	IATA code for Amsterdam Schiphol airport.
FRA	IATA code for Frankfurt am Main airport.
CDG	IATA code for Paris Charles de Gaulle airport.
DKK	Danish Kroner, the currency used in the report.

Table 0.1: **Word list with explanations regarding this project report**

Contents

List of Figures

List of Tables

Listings

1 Introduction

Maximizing the return of your hard-earned money is something everybody can relate to – you want to get the most for your money. This concept concerns all consumer goods, but especially in airfare where there is a high amount of price fluctuation, which makes it more difficult to get the most out of your money. Based on the volatile nature of airfares there is an economic reward in buying the tickets just at the right time, much like the stock market. The hard part is, however, figuring out when that is, and this is the very problem this project seeks to solve. The goal will be achieved by building a web scraper, scanning an internet site for flight prices over time. These prices will then be analyzed with a prediction algorithm in order for the system to provide its users with an educated guess on how the prices will be in the future. This will show users when it is the best time to buy their airfare ticket, securing them a cheap price.

1.1 Problem statement

Based on the previous section, we have reached the following problem statement that the system should comply with:

How can flight price data be collected and used to predict future flight price trends, so that a customer can save money by using the predicted optimal buying time?

To outline the problem statement into more tangible tasks the system should have the following functionality:

- The system has to collect flight information off the websites of airline travel companies or from a search engine.
- The system has to predict if a customer should wait or buy, based on earlier observations. This can be either an immediate purchase ("buy now"), when the system decides that the price is optimal now ("buy when I tell you"), or some time in the future (e.g. "buy in three days").
- The system can be distributed on several machines in order to scale to thousands of flight routes.
- The system has to save the customer money in the average case.
- The prediction should be able to assert with 90% certainty that the customer does not pay more than they would without use of the predictor.

A 90% certainty may be optimistic given that we have little knowledge of the topic, but an end user would want a high confidence of not losing money using the system and a certainty this high would therefore be preferable.

According to IATA *37.5 million* flights landed safely in 2012 [25]. Due to the limited amount of time in this project period instead of collecting flight prices for all possible 37.5 mill. flights, we decided to only analyze flights from Copenhagen Airport (CPH) to the four busiest airports in Europe (LHR, AMS, FRA and CDG).

1.2 Terminology

The price changes of a single flight can be seen as a time series with points in time $T = \{t_1, t_2, t_3, \ldots, t_n\}$ and a function $f : T \mapsto \mathbb{R}^+$ which denotes the price at a given point in time. We say that a price is *optimal* if it is the lowest price for a flight i.e. that t is optimal if $f(t) \leq f(t')$ for all $t' \in T$. A price is *future optimal* if it is the lowest price of all future prices i.e. that t is future optimal if $f(t) \leq f(t')$ for all $t' \in T$ where $t' > t$. See Figure 1.1 for an example of how this applies to a single flight.

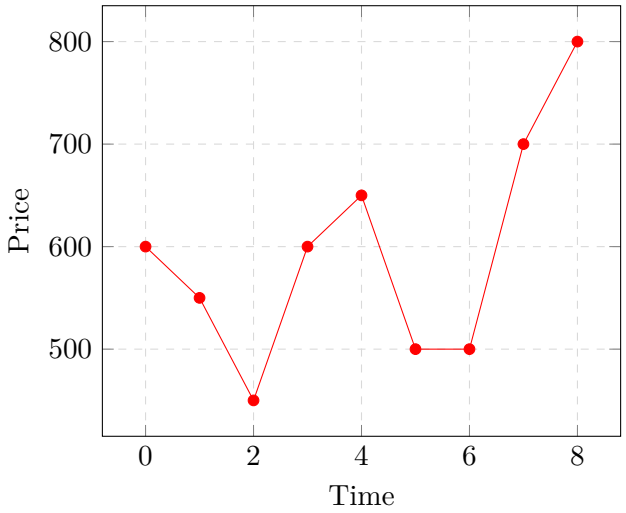


Figure 1.1: **Example of a price curve for a single flight. The optimal price is at $t = 2$, while the future optimal prices are $t = \{2, 5, 6, 7, 8\}$.**

These different concepts introduce a number of useful properties:

- An optimal price is also future optimal.
- The set of all future optimal prices are monotonically non-decreasing.
- If a point is on a monotonically non-decreasing function, it is future optimal (see Figure 1.2).

– If a point is on a monotonically decreasing function, it is not future optimal unless it is the final point (see Figure 1.2).

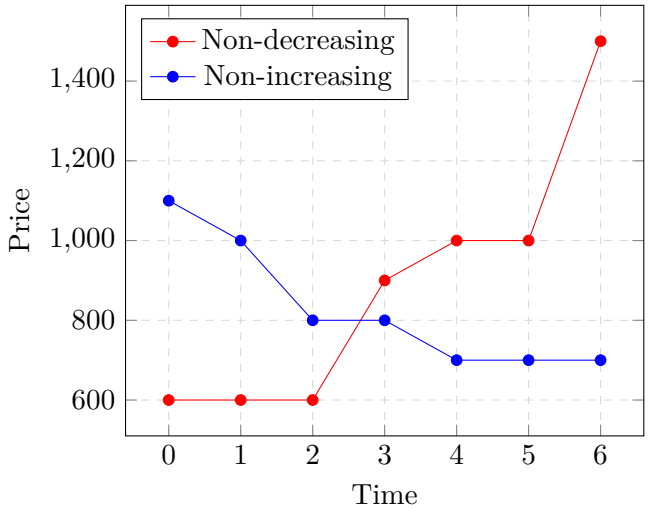


Figure 1.2: **Monotonically non-decreasing/increasing price curves. All points on the non-decreasing curve are future optimal, while only the final point on the non-increasing curve is future optimal.**

In addition, we define the concept of a *future optimal price* for a point in time t which is the minimum of $f(t')$ for all $t' \in T$ where $t' > t$. The *future optimal price difference* is the difference between the future optimal price of t and the price $f(t)$. I.e the future optimal price difference is negative if we can save money by waiting and positive if we cannot. For example, the future optimal price for $t = 3$ in Figure 1.1 is $t = 5$. The future optimal price difference is thus $f(5) - f(3) = 500 - 600 = -100$ indicating that we can save at most 100 by waiting. Likewise the future optimal price for $t = 6$ is $t = 7$ with a price difference of $f(6) - f(7) = 700 - 500 = 200$ indicating that we will lose at least 200 by waiting.

If a person wants to purchase a flight ticket and the current price is not future optimal, it would be beneficial to wait until a price that is future optimal occurs. This, however, depends on data about the future that is impossible to know at a given time. It is therefore necessary to estimate whether the current price is future optimal based on other factors such as the similarity to other flights.

2 Flight price analysis

The graphs presented in this section are based on the data collected using the web scraper, which will be introduced later in Chapter 3.

In order to devise a method for classifying whether a point in time is price optimal for a specific flight, it is necessary to consider how the price may change over time and what may influence the price. In Figure 2.1, we can see that the price curves are generally increasing when approaching the departure time but still has some minor fluctuations along the way. Some price trends progress quite extremely, e.g. the price for the flight taking off the 12th of Nov. 18:25 (the green line in Figure 2.1) increases 11-fold in the interval from 7 to 3 days before departure.

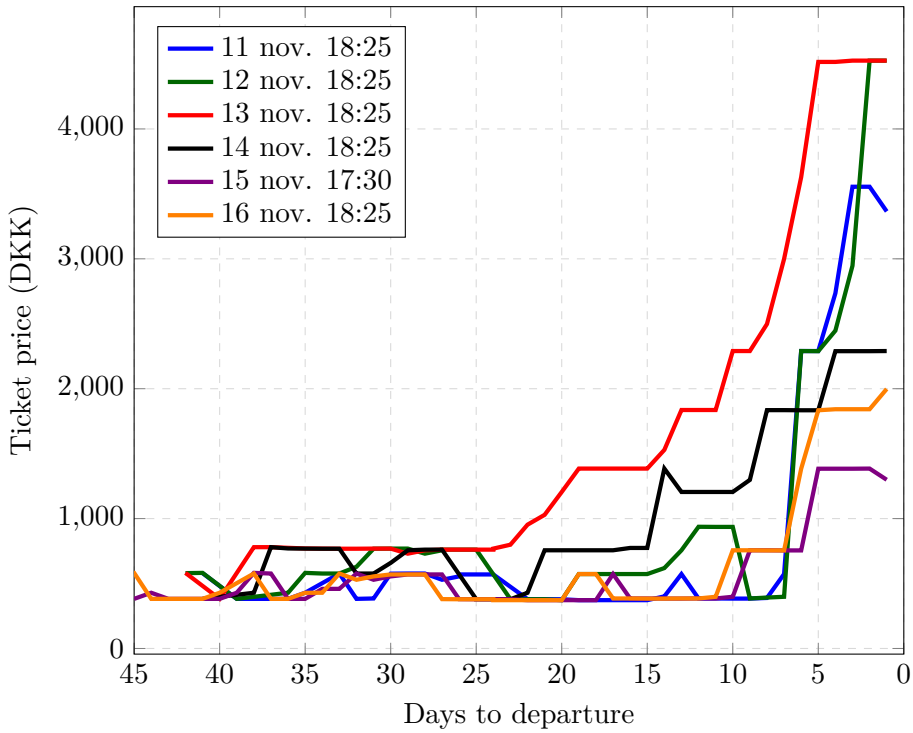


Figure 2.1: **Flight prices for a selection of SAS flights from CPH to LHR**

Generally, it seems that there is a lower bound of about 400 DKK and it seems that most flights reach this lower bound some time before departure. We also see that the closer we get to departure, the larger the spread in price. We can do the same kind of graph on a per airline basis as seen in Figure 2.2.

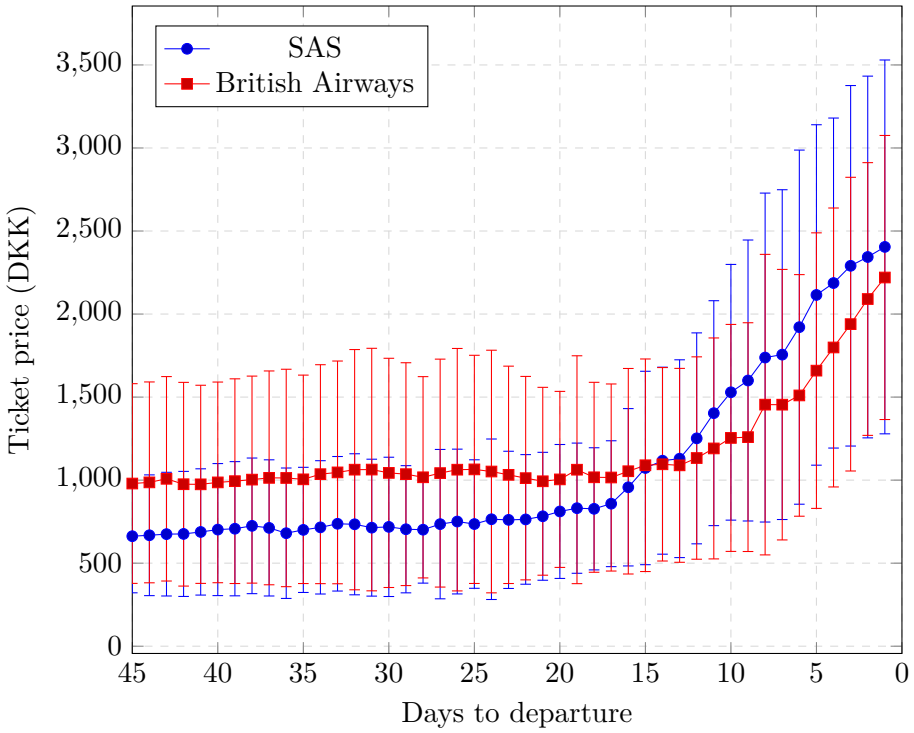


Figure 2.2: **Average flight prices from CPH to LHR**

We can see that SAS and British Airways follow a somewhat similar pricing strategy. They both increase prices when approaching the departure date. The standard deviation (error bars) increase for SAS, but are largely constant for British Airways. This indicates that SAS might have larger price extremes when approaching the departure date.

While these two airlines are largely similar, other airlines have completely different pricing strategies. Air France, for example, has a largely spanning pricing range until about 37 days before departure being quite low on average. At 37 days until departure, the prices are suddenly stabilized (see Figure 2.3) and finally the trend gets more uncertain in the last 10 days before departure. If a customer wanted to buy a ticket for an Air France flight, it would most likely be beneficial to buy it more than a month before departure.

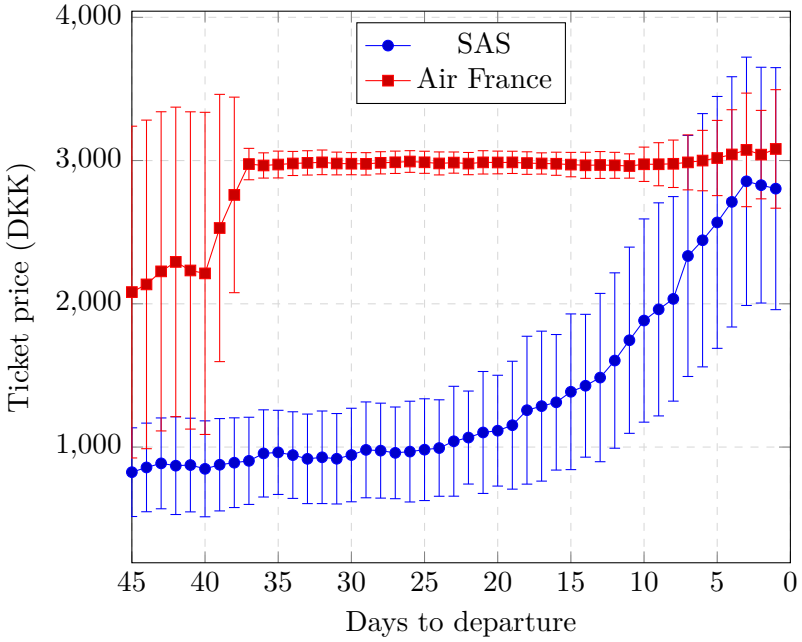


Figure 2.3: **Average flight price from CPH to CDG**

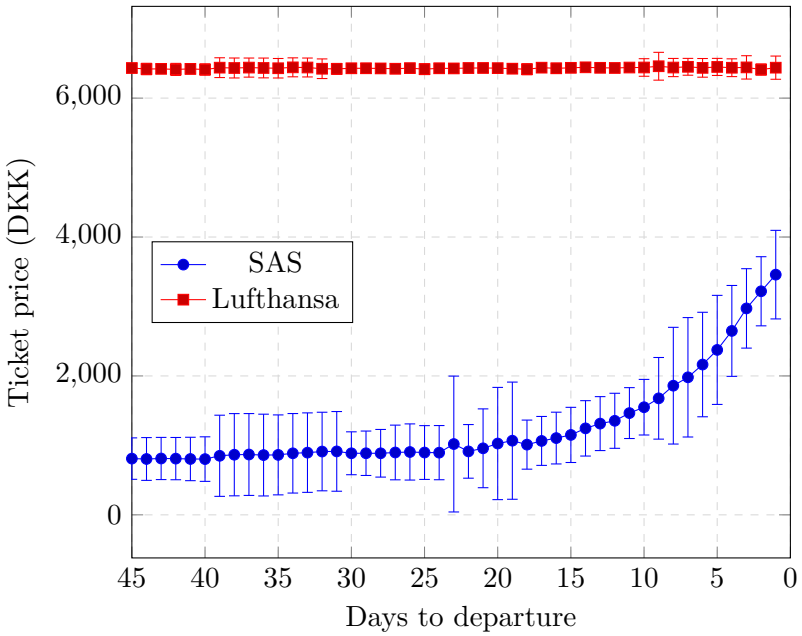


Figure 2.4: **Average flight price from CPH to FRA**

The most remarkable pricing strategy we found was Lufthansa's. Lufthansa uses an almost entirely constant but very high price, only deviating a bit from it when approaching the departure date (see Figure 2.4). We suppose that this is largely caused by the fact that Lufthansa is headquartered in Frankfurt and has a lot of connecting flights to other destinations from Frankfurt. They therefore do not need to be as price competitive on their flight to Frankfurt, as most passengers will probably book a connecting flight to elsewhere.

These different pricing strategies has a large influence on whether it is possible to save money by waiting to buy a certain ticket. A stable price would mean low benefits from waiting, but also few disadvantages. A more chaotic price could have more opportunities for saving money, but also larger penalties for incorrect classification. We can see in Figure 2.1 that while there are some cases close to the departure that are relatively cheap, the penalty for incorrect waiting could result in a 11-fold price increase.

It is very likely that these different pricing strategies will mean that certain prediction models will fit certain companies better than others. It might be better to consider an ensemble of prediction models, where a specific model is chosen for each company. These models will be explored in further details in Chapter 4.

3 System design

This chapter describes how the entire system was envisioned, detailing how the data is gathered and how the system stores the data in the database. Later on in Chapter 5 there will be given a detailed description of how the ideas in this chapter was implemented. The prediction design will be further discussed in Chapter 4.

3.1 Language choice

To keep the source code of the project from spanning several code-bases that needs to be kept in revision, we have chosen to only work with a single programming language. As the group uses different operating systems on their work computers and because we would like to use a common Interactive Development Environment, we choose to use Java as a programming language with Eclipse as the IDE. Although the subsystems of our project could be made using different programming languages it would require extra resources to manage several projects and code-bases.

3.2 Data analysis

In order to predict the future price or trends for a flight ticket some data is necessary. All the needed data is mostly available on the internet, e.g. on the different airline companies' web pages. However, there exist a wide variety of search engines that allows users to find and compare prices on flight tickets such as Momondo and Skyscanner, where Skyscanner will be described below. We assess the optimal obtainable analysis data is:

Ticket price The price for a direct flight as a single adult

Log time The time when the data was gathered

Departure airport The origin airport from where the flight departures

Destination airport The airport the flight is going to

Carrier The airline company responsible for the flight

Flight number The specific flight number for the planned flight

Departure time The time and date the flight is scheduled to leave the departure airport

Duration time The estimated total flying time for the flight

Travel form Differs between direct or round trips

Available seats The amount of seats still available for purchase on the flight

All of this data would be useful for either storing in a database to ensure the identity of the different flights or for predicting the future price or trend.

To get these data one can either go directly to each airline's website or use an online search engine which provide a collection of airline ticket prices from a variety of airline companies.

An alternative way to obtain this data would be through an external API, e.g. Google's QPX Express API which provide prices on global airline tickets. However access to an API often requires payment, this is also the case with Google's API. The first 50 queries per day are free whereafter it costs $0.035 US per query [11], which makes this a very expensive solution when it is necessary to collect large amounts of data.

3.2.1 Skyscanner

Skyscanner [2] is a search engine for airline tickets, hotels, and car rentals. Skyscanner does not sell these services, but gathers information from other sources and refers the users to buy them from these sources.

Skyscanner requires the following input to search for a flight route:

–Departure airport – Destinaiton airport – Travel date – Travel form

In Figure 3.1, a screenshot is shown from skyscanner.dk with the following input:

- Departure airport: **CPH**
- Destination airport: **LHR**
- Travel date: **1st November 2014**
- Travel form: **Direct flights only**

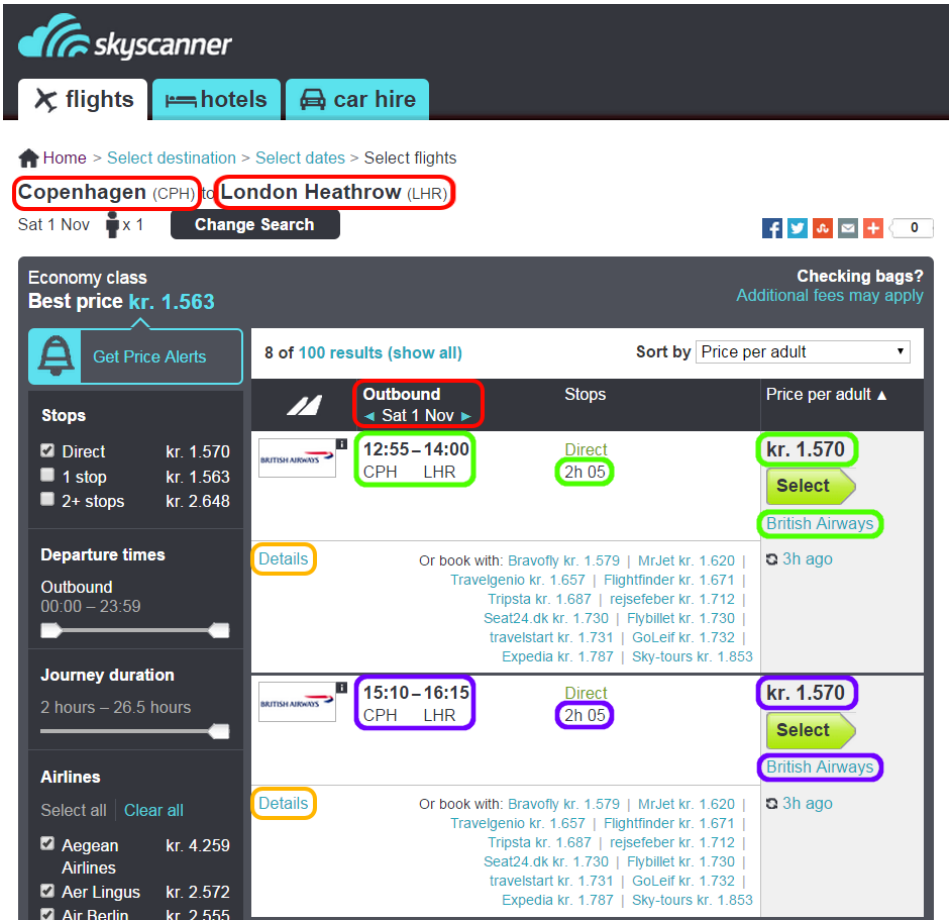


Figure 3.1: **Screenshot from skyscanner.dk** [2]

The screenshot shows the search result of the given input. The input to Skyscanner outlined with red and the output for each found flight is outlined respectively green and purple.

Skyscanner offers the following output:

- Carrier (e.g. British Airways or SAS)
- The price of the flight (in DKK)
- Departure time, the time the flight is scheduled to be in air, the timezone is the same as the departure airport
- Travel duration, in hours and minutes
- Travel form (e.g. direct trip or round trip)

Skyscanner also provides the flight numbers but only if you click the "Details" button (which is outlined with orange on Figure 3.1). Unfortunately Skyscanner does not provide any information about the number of available seats available on a flight, and the flight number of the flights would not provide any helpful data for prediction. Because of this we chose not to let the web scraper waste time on clicking the "Details" button for every flight.

Some search engines, like Skyscanner, protect their service and data from exploitation by blocking certain IP addresses with high query volume which they deem as malicious access. This sets a clear limit for the number of requests the web scraper can make with Skyscanner as the source. Due to the the rate of requests necessary for round trips this project only considers direct trips.

Skyscanner was selected as the search engine for this project as their website uses a calendar widget that supports plain-text editing, making it easier to use by a web scraper.

3.3 Web scraper

To extract flight information data from the internet, a web scraper (also called a web data extractor) is needed. A web scraper is a piece of software simulating a human exploring the internet while gathering data. There are multiple open source web scraping tools available.

Many search engines, such as Momondo and Skyscanner, use large amounts of JavaScript to make the website interactive. Unfortunately this makes it problematic for a machine to interpret and navigate a website, as it requires compiling and executing the JavaScript code which is a much larger task than just reading the HTML. In addition, things like cookies and modern CSS style sheets can also change the behavior of the web page. Most web scraping libraries only support scraping plain HTML sites. This works well for most static pages, but is often terrible for more interactive pages. It so happens that most flight search engines, Skyscanner included, use these interactive features.

Some web design patterns designed to make the website easier to use for humans, can make it significantly more difficult to use for web scrapers. A common example of this is the use of a calendar widget which makes it more manageable for a human to select a date, but makes it difficult for a computer to navigate individual months.

The following sections will describe two scraping libraries, Scrapy and Selenium, both able to handle interaction with websites through JavaScript.

3.3.1 Scrapy

Scrapy is a simple open source scraping library which among others, promote itself as simple, fast, and portable. This would work well with the nature of our project and as such it is a good choice. Unfortunately Scrapy only supports Python and it would therefore compromise the previous programming language choice for our project, which is Java.

3.3.2 Selenium

Selenium is an automated browser and testing framework for web applications. Selenium supports several languages such as Java, JavaScript, C#, PHP, and Python. It runs in any mainstream browser and is open source. As described above traditional web scrapers can have problems scraping websites using JavaScript, however Selenium opens a specific web driver (for instance a Google Chrome or a Mozilla Firefox web browser) in which the testing or scraping will be executed. This ability makes Selenium suitable for this project as it is able to do anything a human could, using a web browser, the speed of the scraping however will be reduced compared to that of traditional web scrapers.

Selenium was chosen for this project because of its JavaScript support, its support for implementation in Java, and lastly its capability to control an actual browser and thereby simulate human actions as input to the browser.

3.4 Database

To be able to store the information about flight tickets and the periodic development of their price, a storage facility of some kind is needed. This storage facility, or database, is required to cope with a large amount of data to ensure scalability of the system.

3.4.1 SQLite

SQLite is great for small development and testing as the whole database is contained in a single file that is easy to move using the file system [30]. Based on the scope of the project SQLite would be sufficient, but the project also focus on scalability so that it might use strong distribution software such as Hadoop. This made SQLite a poor choice for the project as it does not work well with data sizes that Hadoop is made for.

3.4.2 MongoDB

NoSQL databases, such as MongoDB, excels in environments where there is a big write load on the database as the insert function is very efficient, although not very secure [19]. The safety issue is not a big concern as all the data used could be re-scraped from the web just as we have been doing throughout the project period. The biggest problem with potential data loss would be to assert what, or if any, data is missing as it would require the web scraper to run again at least once to make sure that every entry found already exists in the database and if not then insert it. The project would not gain much from the insert throughput of MongoDB as most of the database access consists of retrieving the data from the database. The time consumed by the web scraper loading web pages also largely outweighs that of the database's write/read time.

3.4.3 PostgreSQL

PostgreSQL [26] is an open source SQL implementation offering large data capacities [15], up to 1 GB per field and 32 TB per table, as well as the support for several users accessing the database concurrently. PostgreSQL complies with the ACID (atomicity, consistency, isolation, and durability) principle as well as supporting a large variety of data types including large binary objects. This makes PostgreSQL a good choice for our project as the system requires a lot of data and will continue to grow for as long as it is used.

3.5 Object-relational mapping (ORM)

An ORM is a mapping tool for converting objects from object-oriented programming (OOP) into relational data entries and vice versa, e.g. between fields in Java objects and SQL database entries in tables. This means that the ORM handles all communication with the database through objects that the programmer can create, update, and delete without the need for using or even knowing SQL statements or queries. This is one of the ways to deal with the object-relational impedance mismatch [8], which is a set of problems concerning the use of the two different systems, i.e. a object oriented language and a relational database. As such most of the work of handling the impedance mismatch will be taken on by the ORM layer.

The project group agreed to use an ORM layer between the functionality layer and the database layer of the system, and as the database of choice for this project is PostgreSQL the chosen ORM library would need to support communication between Java and PostgreSQL. The following section will describe the Hibernate ORM library that we chose.

3.5.1 Hibernate

Hibernate is a widely used ORM implemented in the Java programming language. In this project its task would be to convert Java objects created by the web scraper into entries in a relational PostgreSQL database, and later convert the entries back into new Java object that can be used to perform prediction on flight prices.

Hibernate was chosen mainly because of its high performance and scalability [6]. It was originally designed for an application server cluster, making it very useful for our structure [6]. The Hibernate ORM is capable of handling the PostgreSQL database types [4] and as such would be a fine choice for our project. The Hibernate ORM also specifies its own querying language called Hibernate Query Language (HQL) which is a powerful language similar to SQL but fully object-oriented and able to understand notions like inheritance, polymorphism and association [5].

3.6 Prediction

A significant part of the system is the prediction of the flight prices which will try to predict whether to buy a given ticket at the current time or if you should wait for a later time. Two different types of prediction will be discussed in this report: classification and regression. These two types will be further explained and discussed in Chapter 4 where different methods of classification (see Section 4.2) and regression (see Section 4.3) are explored. Later in Chapter 6 it is evaluated which method is best suitable for our problem of determining whether to buy a ticket now or wait.

3.7 System model function

This section describes how the implementation of the system will function taking a mathematical perspective using tuples and functions. The system consists of the previously mentioned parts: a web scraper outputting to a database, and a predictor making predictions based on the data stored in the database.

The web scraper, S, for Skyscanner takes the following as input:

Two strings, a_{depart} and a_{dest}, which specifies the departure and destination airport of the flight as well as two dates t_{start} and t_{end}, specifying the first and last departure date to scrape respectively. The start date must be before or equal to the end date. The web scraper will scrape all flights for the defined route on all days from t_{start} to t_{end}. As output the web scraper gives a list of tuples each containing the carrier c, price p, departure time dt, duration d, and the log time l, of each flight for the searched routes and dates.

$$S\left(a_{depart}, a_{dest}, t_{start}, t_{end}\right) = \left\{\left(c_1, p_1, dt_1, d_1, l_1\right), \left(c_2, p_2, dt_2, d_2, l_2\right), ..., \left(c_k, p_k, dt_k, d_k, l_k\right)\right\}$$

This data is then stored in a database along with the original input for the web scraper in several single units called flights, denoted by f. A flight consists of the following elements; origin airport a_{depart}, destination airport a_{dest}, carrier c, departure time dt and duration d. A flight denotes a list of flight prices consisting of a price p and a log time l defining when the associated price was found. As such f will be a tuple:

$$f\left(a_{depart}, a_{dest}, c, dt, d\right) = \left\{\left(p_1, l_1\right), \left(p_2, l_2\right), ..., \left(p_k, l_k\right)\right\}$$

The flight objects can then be used in a predictor P, that when given a list of flight prices will be able to create a classifier function F that can predict whether to buy a ticket for a specific flight at a specific time, e.g. from CPH to LHR on the 1st of November 2014, the created function returns a Boolean value indicating whether to buy now or wait for the specific date. As such the list of input will be on the following form:

$$\left\{\left(a_{depart_1}, a_{dest_1}, c_1, dt_1, d_1, p_1, l_1\right), ..., \left(a_{depart_k}, a_{dest_k}, c_k, dt_k, d_k, p_k, l_k\right)\right\}$$

Each input is basically a list of unions of flights and associated flight prices. As such the input differs on flights, where the input is most likely static in a lot of cases, e.g. multiple flight prices for a single flight, as well as the price and the log time. This data is what the predictor will use to make the function F using prediction methods.

The function F takes a union of a flight f, a timestamp, t, and the current price at that timestamp, p, as input and will return a Boolean value indicating whether to buy the ticket at that time or wait.

$$F\left(a_{depart}, a_{dest}, c, dt, d, t, p\right) = \left\{\top, \bot\right\}$$

This sets the limitation that it is only possible to predict whether to buy now or wait for days which data have been scraped for, i.e. for the current day or a day prior to this. It is of course possible to predict the result for a day in the future if you are able to estimate or predict the price for that day, however that is another problem, which would also make the task of classifying a correct result obsolete as predicting the price can be used to answer the classification directly. Another possible way is also to leave the current price out of the predictor, making the timestamp the only variable, however this have been found to be less efficient as examined in Subsection 4.1.4.

3.7.1 Input / output data

To clarify the mathematical models the input and output data to the web scraper is illustrated respectively in Table 3.1 and Table 3.2.

String Departure Airport a_{depart}	String Destination Airport a_{dest}	Time StartDate t_{start}	Time EndData t_{end}

Table 3.1: **Web scraper input data**

String Carrier c	BigDecimal Price p	Time DepartureTime dt	Time Duration d	Time LogTime l

Table 3.2: **Web scraper output data**

3.8 Task distribution

A secondary part of the system is the distribution of several different tasks; the system needs a way to manage and distribute time-consuming tasks between several machines.

There are two tasks that would be beneficial to distribute: the scraping of Skyscanner [2] and the creation of predictors. Scraping is primarily limited by the speed of the network and the browser it simulates. Scraping is therefore not very CPU intensive, whereas predictor training is a much more CPU intensive task. It would be useful for the receiver to be able to handle both types of tasks, as to utilize the power of the machine as much as possible. If a receiver can only handle one type of task and there is no messages for that task available on the server the receiver unit would be idle while there might still be pending messages for the other type of task. For distribution of these tasks a message queue would be a useful tool. There exist several different open source libraries, such as ActiveMQ, RabbitMQ and ZeroMQ, that all implements message queues.

A message queue system is a system that allows different programs to communicate with each other through a message server. The general idea is that some subsystems can publish messages to the server while other subsystems can receive the messages from the server and act on the contents of the messages. Then the receiver can acknowledge the messages if they were able to act according to the contents of the messages. This will mark the messages as “handled” by the server and the messages can no longer be received by other subsystems.

This can be used for distribution of tasks. Say a task is comprised of a message that contains the necessary input to the web scraper. When the task/message is published onto the server only one of many "web scraper" receivers will get the message and start to perform scraping. If the contents of the task could be split up into smaller scraping tasks, e.g. split into two tasks each of half size, two receivers could work on the task simultaneously. Therefore the message queue system would not be handling the distribution of tasks into sub-tasks but would only handle the distribution of the sub-tasks to several receivers.

Based on observation and reviews from other programmers we have concluded that the difference between these open source libraries were so negligible that we will only focus on RabbitMQ. The following section will briefly describe RabbitMQ and the more advanced Hadoop.

3.8.1 RabbitMQ

RabbitMQ has high reliability due to having built-in persistence, delivery acknowledgments, and publisher confirms. Furthermore, RabbitMQ is language independent and includes a friendly web-driven management user interface [20].

RabbitMQ was chosen for this systems because of its rapid deployment, robustness, and ease of use.

3.8.2 Hadoop

Hadoop is a piece of infrastructure software based on several open source frameworks and tools. The typical problem Hadoop is used for is to solve storage and handling of huge amount of data [14]. Hadoop handles both big data and the computation of data by breaking it into smaller pieces, making it easier for individual machines to handle. The storage of data is organized in clusters using the Hadoop Distributed File System. The file system is, as the name states, a distributed file system which makes it secure and easy to scale. The handling/processing of the data is done using the principles of MapReduce. The principle of MapReduce is to map a task to different computers and after processing reduce the result so it is more easily accessible [16].

Due to the complexity of analyzing different prediction methods and a time-limited project, we refrained from using Hadoop even through the project might benefit from its comprehensive properties.

3.9 Development tools

In this section, the tools utilized for development will be described. Alternatives to the tools used will also be discussed. The tools that where used include: Git (via GitHub), Gradle, and Jenkins.

3.9.1 Revision control

Git and SVN are two popular revision control systems used for source code management, which is especially useful when developing software in groups. Ultimately the difference between the two, and a lot of other similar software, is preference as they are not radically different from each other.

In this project Git was decided upon preference, and to host the repository the popular hosting service GitHub was used. The amount of features it provides on the website is vast with a great graphical user interface making it simple to use [1].

3.9.2 Build automation tool

Using build automation tools eases the building process when compiling. Some of the tasks that build automation tools can automate are: compiling source code to binary, packaging binary code, and running automated tests.

For this project the automation tool Gradle was used. Gradle is developed combining the advantages from the popular build tool Apache Ant and Apache Maven as well as a few additional features, such as reliable incremental builds [3].

3.9.3 Continuous testing

For ensuring correctness of the program, continuous testing was implemented. This was done with the extendable open source continuous integration server called Jenkins. It is an award-winning [21] piece of software that monitors the execution of repeated jobs, such as building and testing software projects continuously.

Installation is effortless as it is as simple as running a simple command, and generally exhibits ease of use.

It was setup so that every commit pushed to the project's GitHub repository would trigger a run of the automated tests in Jenkins. The results would be displayed on the servlet, running on one of the servers in the project group room.

3.10 Summary

The system modules described in the previous sections are all summed up in Figure 3.2, where it is shown how information flows between the components.

The input (departure airport, destination airport, and date) used in Skyscanner is given to the web scraper which process the information, collects the output and passes it to the ORM. The ORM (Hibernate) encapsulates the database, so that the web scraper and predictor only deal with Hibernate. The predictor then takes the data that the web scraper collects and turns it into something meaningful that users will be able to use when deciding whether they should buy their flight tickets. Reflections on the predictor will be described in Chapter 4.

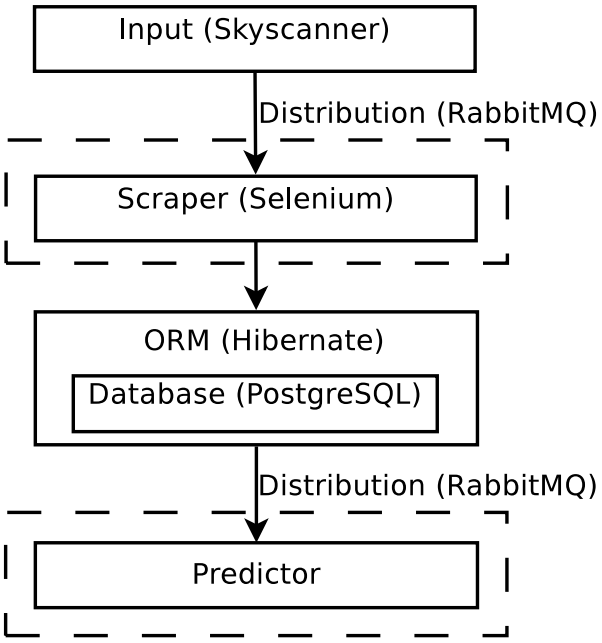


Figure 3.2: **A visualization of the system modules and how the data moves through the system**

4 Prediction models

Predictions are used to make an educated guess on how certain values will evolve in the future, in this case how the airfare prices will fluctuate. At first some common attributes will be described in Section 4.1, then the classification models will be discussed in Section 4.2. The last discussed prediction model is regression in Section 4.3. This chapter will be evaluated in Chapter 6.

We will consider two different machine learning approaches: classification and regression. In both cases the trained machine learning algorithm takes a vector of input variables and outputs a vector of output variables. The major difference between the two, is that the output of a classification function is discrete, while the output of the regression function is continuous.

A regression function can often be used as a classification function and vice versa:

- A classification function can be trained with continuous data if the values are discretizied beforehand.
- A regression function can be trained with discrete data if each possible value is assigned a continuous output.

This is often the case with neural networks, but may not be effective for other regression methods.

4.1 Attributes

Before any predictions can be made, the attributes used for determining the future prices has to be determined. These attributes are the base for how the machine learning algorithm learns and predicts.

4.1.1 Ways to partition the data

There are a number of ways which we might partition the data into smaller subsets. Partitioning the data into multiple independent sets would enable us to train multiple machine learning systems in parallel. This might make the system significantly faster to train, and as long as we ensure that prices are independent across partitions, it should also be just as precise.

These are the possible ways we have considered partitioning the data:

- Not at all (train a single predictor with all data)
- By airline
- By route (origin, destination)
- By route and airline (origin, destination, airline)

The project has refrained from partitioning the data, as we have observed better overall prediction when no partitioning is done (see Section 6.3). This is most likely because the machine learning algorithms are able to generalize across different airline companies. For example the price is likely to rise when approaching the departure date no matter the airline company. Since we do not partition the data, the machine learning algorithm will have to include the origin, destination and airline as attribute inputs as well as the attributes discussed in the following sections.

This solution is not scalable; a single computer needs to handle all the data. If it were to analyze a large data set (such as all flights on Earth), it would be necessary to partition the data in some way in order to distribute the workload onto multiple machines.

4.1.2 Price influencing attributes

In addition, there are a number of attributes that might influence whether a price at a given time is optimal. For example the number of days before departure may have a significant impact on whether there will be any lower future prices. We have considered a number of different possible attributes:

- Days/hours to departure
- Week/day/hour of departure
- Price
- Base price, the price a certain number of days prior to departure (e.g. price 30 days before departure)
- First price, the earliest observed price
- Price relative to base price or first price
- Price at a certain time before current price (e.g. price 1 day ago, a week ago, etc.)
- Difference between current price and a certain amount of time before now
- Relative difference between current price and the above

4.1.3 Class variable

The class variable or class attribute is the attribute that has its value predicted either by regression or classification. As with the different price influencing attributes to select from, there are also a number of class attributes to look at. We might want to estimate a future price curve (regression) or just whether now is a good time to buy (classification). Estimating the future price is a finer prediction than just estimating whether the current price is future optimal; whether a price is future optimal can be seen from the price curve, but not the other way around. Here are some of the different class variables we have considered:

- Whether the price is increasing, stable, or decreasing (classification)
- Whether the current price is future optimal (classification)
- Price for a given time (regression)
- The difference between the current price and the future optimal price (regression)
- Above, but relative to the current price (regression)

Some of these predictions contain more information than others. For example if we can predict the future price trend, we can also predict whether the price is going to increase or decrease and whether the current price is future optimal. Thus it must be easier to classify whether a price is future optimal than it is to predict a future price curve, as the former reduces to the latter.

We will therefore largely focus on estimating whether the current price is future optimal as this is the most coarse estimation. This, of course, means that the entire price until departure is necessary in order to train the system. This means that we can only train the system with flights that already have departed.

4.1.4 Attribute selection

We could use all of the attributes mentioned in Subsection 4.1.2, but it is likely that some of these are heavily correlated or even irrelevant. For example the difference and relative difference between two time points are naturally heavily correlated. These might not have a lot of possible information gain, therefore it might be beneficial to ignore these attributes.

A general rule is that the best attributes have high correlation with respect to the class variable, but low correlation between each other [17]. Based on this, we use Correlation based Feature Selection (CFS) by Hall [17] to select the best set of attributes. CFS is only an evaluator of attributes; it can tell if one set of attributes is better than another set. Due to our relatively small number of attributes, we can perform a simple exhaustive search over all possible sets of attributes in order to find the best set of attributes. We used the Weka machine learning tool to do this analysis and found the following attributes to be best:

- Days to departure
- Carrier
- Current price

The remaining attributes only provide little information gain compared to these. The results are largely what we would expect with the only surprise being that the destination is not included. It would seem obvious that the destination has a large influence on the price since a further away destination means a larger cost, but this does not necessarily mean that it has a influence on whether the current price is future optimal.

These attributes are therefore the most necessary attributes and are the ones we have chosen to focus on.

4.1.5 Weight

In addition to the aforementioned attributes it is beneficial to assign a weight to each training instance. The reason for this is that there are times where it is much more important to classify correctly.

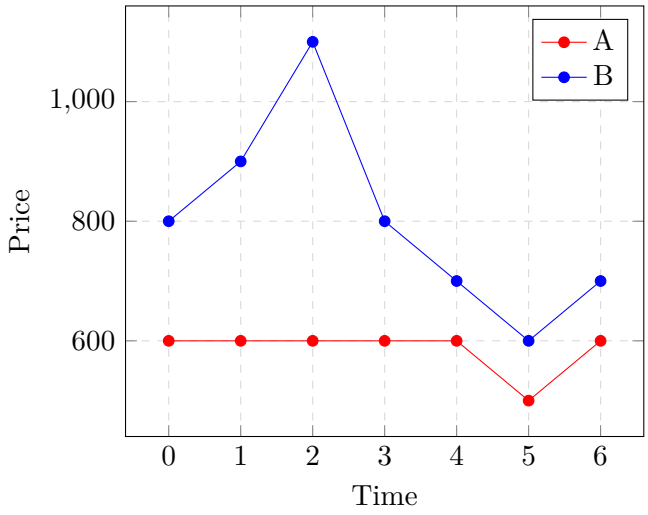


Figure 4.1: **Example of two different price curves where neither is price optimal at $t = 2$ but have different weights**

For example consider Figure 4.1. While neither A nor B is price optimal in $t = 2$, we can much better tolerate an incorrect classification in A than in B due to the smaller difference between the price at $t = 2$ and the future optimal price at $t = 5$. The machine learning algorithm should therefore prioritize classifying B at $t = 2$ correctly over classifying A at $t = 2$ correctly.

We therefore assign a weight w to each instance based on the current price p and the future optimal price p_f:

$$w = \frac{\max(p, p_f)}{\min(p, p_f)} - 1$$

We chose this weight because of a number of properties we wanted:

- If the difference between the price and the future price is 0, then the weight should be zero as there are no benefit or disadvantage for waiting or buying.
- If the future minimum price is n times the current price p where $n \geq 1$, then the weight is $n - 1$.
- If the future minimum price is n^{-1} times the current price p where $n \geq 1$, then the weight is $n - 1$.
- The larger the difference between the price and future price, the larger the weight.

For example the weight of A in $t = 2$ would be $600/500 - 1 = 1/5$. Likewise the weight of B in $t = 2$ would be $1100/600 - 1 = 5/6$. Thus B is weighted $25/6 \approx 4$ times more than A in $t = 2$.

This ends up weighing nodes with the same price as the future optimal price at 0. This is theoretically not a problem: when the price is the same as the future optimal price it makes no difference whether we wait or buy immediately. In practice we would however like the system to buy if this difference is 0. We do this because price increases are often much larger than price decreases and it would therefore be preferable to buy earlier than risk a large price increase. To achieve this behavior we add a small base weight to each instance so that no instance is totally ignored by the machine learning algorithms.

In Section 6.4 we evaluate the effects of using these weights.

4.2 Classification

There are a multitude of different methods that can be used for classification in machine learning. In this chapter we will be focusing on decision trees and rule learning and their applications for this project. As mentioned in Subsection 4.1.3, we will mainly look into predicting whether the current price is future optimal (see Section 1.2). This is because a future optimal price means that the user should buy now and a non future optimal price means that the user should wait.

4.2.1 Decision trees

A decision tree has a similar structure to that of a flowchart, however certain rules ensures a specific structure. The relations between elements are drawn into a hierarchical network, where the initial node is the root node and then the tree is traversed by following the decision nodes until a leaf node is reached. An example of this can be seen in Figure 4.2 where a decision is made on whether the current price is future optimal. A decision tree is used for classification. As the output is a leaf node it can only output one element from a finite set of elements, in our case either True or False.

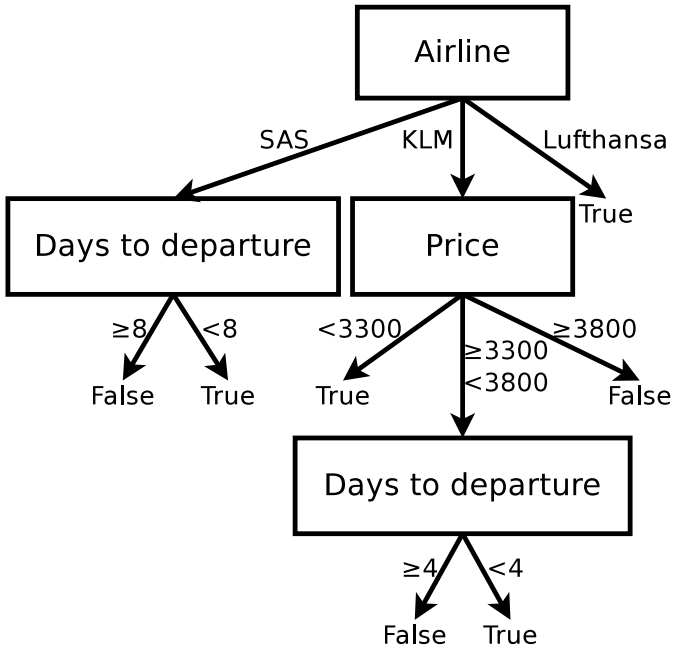


Figure 4.2: **A simple decision tree exploring various factors that may have an influence whether a price is future optimal. The data depicted is not necessarily real and is merely an attempt to display how a decision tree works.**

There is several different algorithms for constructing a decision tree from a set of data. These algorithms have different strengths and weaknesses which we will explore in the following sections.

4.2.2 C4.5

One of the more known algorithms for building building decision trees is the C4.5 algorithm by Quinlan [28]. The C4.5 algorithm is a divide and conquer algorithm that continuously splits on the attribute with the highest information gain. It is an improvement on Quinlan's earlier ID3 algorithm with added support for continuous inputs. The information gain, also known as the Kullback-Leibler divergence, is a measure for the reduction of information entropy. This means that we always chose the node that leads to the lowest entropy.

It continues to do this until all of the samples belong to the same class or a few number of classes. In this case we just generate a leaf node with that class. If we do not have conflicting instances this is always possible to reach. The basic pseudocode for the C4.5 algorithm can be seen in Algorithm 4.1.

Algorithm 4.1 Basic C4.5 pseudocode [28]

```
1: procedure C4.5(S)
2:     if stopping criteria met then terminate
3:     a_best ← arg max(information-gain(attribute a ∈ S))
4:     D_v ← the subsets from splitting on a_best
5:     Tree ← decision tree that tests a_best in root
6:     for all D_i ∈ D_v do
7:         Tree_i ← C4.5(D_i)
8:         Attach Tree_i to the corresponding branch of Tree
9:     return Tree
```

In cases where there are continuous inputs, the algorithm splits the input into two intervals. This means that the algorithm for C4.5 always produces a binary decision tree, although there are variants that produce decision trees with higher branching factors. Afterwards the decision tree created by C4.5 is often pruned. Pruning is a process to reduce overfitting of a decision tree by removing sections that provide little predictive power or tries to describe noise in the input data. There are several ways to prune a decision tree with one of the simplest methods being *reduced error pruning*, where each node is replaced with its most popular class if it does not worsen the predictive accuracy of the decision tree.

We use J48, an open-source C4.5 implementation, as our decision tree algorithm. J48 can optionally use the aforementioned reduced error pruning method to prune the tree after it has been built.

4.2.3 RandomTree

The implementation of the RandomTree algorithm in Weka uses a modified version of REPTree which is a fast decision tree learner [7]. It is modified to get the desired randomness where at each node k randomly chosen attributes are considered. No pruning of the data is done by the algorithm [33]. The RandomTree algorithm is mainly used in the Random forest algorithm but we have found that it is surprisingly good at classifying.

4.2.4 RandomForest

Another variant of decision trees are the random forest algorithm. Random forest is a method for either classification [22] or regression, that operates by creating a forest of decision trees where it then utilizes several learning algorithms (ensemble) to attain a better prediction accuracy. The forest of decision trees are constructed at training time. Class assignment is made by the number of votes from all the trees (each tree being a vote) and for regression the average of the results is used.

A subset of the training data is selected (~2/3) to train each tree in the forest. The remainder of the data is used for error estimation and variable importance.

By randomly selecting the features to split on for each decision tree the random tree is better than most machine learning algorithms at reducing incorrect classification due to noise, but at the cost of more calculations.

The forest error rate depends on two factors. First, the correlation of any two trees in the forest, as increased correlation will increase the forests error rate. Secondly, the error rate of each individual tree (often referred to as "strength"). Decreasing the error rate of a single tree decreases the overall error rate as its vote gets more precise (classification) or it pushes the average result in a more precise direction (regression).

4.2.5 RIPPER rule learner

The RIPPER (Repeated Incremental Pruning to Produce Error Reduction) rule learner is a propositional learner that is designed to classify new data based on a training set. This approach fits well into what this project tries to accomplish as the data already gathered can be used as the training set and then classify the new data.

In rule learning each classification method uses a rule learning algorithm to generate rules from a set of data, this rule set is then applied to a new set of prediction data. Many of the different divide-and-conquer algorithms use biases to make it better at certain problems. RIPPER uses an *over fitting avoidance bias* meaning that it prefers using simpler rules rather than complex rules even if the complex rule have a better accuracy on the training set. This is especially an advantage on noisy training sets so that the rule learner does not make rules based on the noise but rather generalize the tendency of the learning data.

Algorithm 4.2 Extract from RIPPER pseudocode [12]

```
1: procedure RIPPER(P,N,k)        ▷ P is positive examples, N is negative examples
2:     RuleSet ← BUILDRULESET(P,N)
3:     for i = 0 to k do
4:         OPTIMIZERULESET(RuleSet,P,N)
5:     return RuleSet
```

The pseudocode in Algorithm 4.2 shows how RIPPER builds the rule set and then optimizes it. The full pseudocode of the RIPPER algorithm (which includes the `BuildRuleSet` and `OptimizeRuleSet` procedures) can be found in [12, page 11].

4.3 Regression

In this section the report will take a closer look on the regression algorithms utilized, including neural networks and time series.

4.3.1 Neural networks

Neural networks [9] as a computer science concept is inspired by the central nervous system in the brain, in the way that the nodes in the artificial neural network resemble that of its biological counterpart. Biological neurons receive signals through synapses, and when the signals received are strong enough (above certain threshold) the neuron is activated and generates a signal output via the axon. The complexity of real neurons is highly abstracted when modeling artificial neurons: they consist of inputs, which are multiplied by weights ("strength" of the given signals), and then computed by a mathematical function which determines the activation of the neuron, another function computes the output of the artificial neuron.

There are several types of neural networks, this project have mainly used the multilayer perceptron. The multilayer perceptron is a feed-forward neural network consisting of a set of input nodes for each attribute, a number of hidden layers each with a number of hidden neurons, and one or more output nodes corresponding to the result for the given input. The number of hidden layers and neurons affect the overall speed and precision of the multilayer perceptron, but the optimal numbers is almost always different from dataset to dataset. Multilayer perceptrons uses back-propagation [29], a supervised learning technique, for training the neural network and can distinguish non-linearly separable data [10].

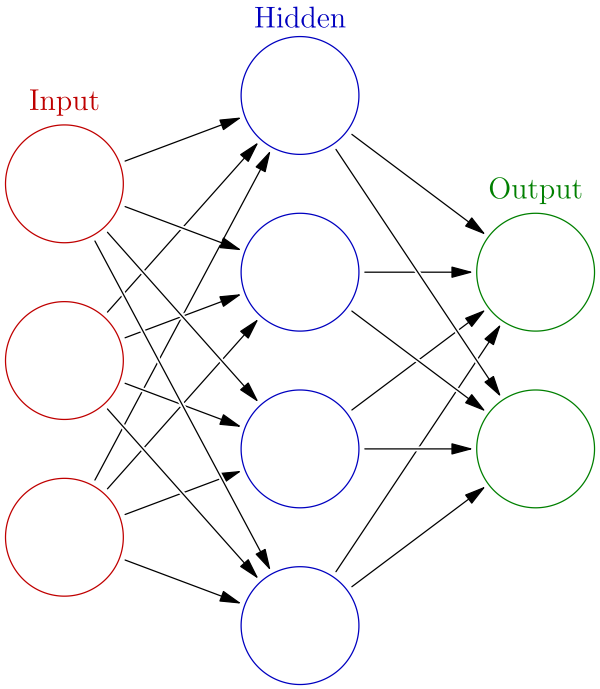


Figure 4.3: **An artificial neural network is a group of connected nodes with weighted connections. [32]**

4.3.2 Time series

A time series [31] is a set of numeric measurements of the same entity taken at equally spaced uniform intervals over time, often plotted via line charts as in Figure 4.4. The vertical axis is the time axis where the time is split in equally spaced intervals in order to keep the chart consistent. Time series are frequently used in statistics, weather forecasting and earthquake prediction, and is used in forecasting because it attempts to predict future events and trends based on what has been observed previously.

There are five different variables when discussing time series, first is the trend. Trend is the long term development in the time series, and it can either have an upward, downward or stable tendency, in Figure 4.4 the trend clearly have an upward tendency. The second variable is seasonality, seasonality is a form of repeated data that happens at regular time intervals. Seasonality is often related to natural or human behavior, in Figure 4.4 the seasonality visibly as the peaks in every year of the series. Random variation may not be occurring at all or the variation can be clear as in Figure 4.4 the variation is high because the peaks is increasing with time. The last two variable is more difficult to see in the figure, e.g. cycles which occur when the series follows an up and down pattern that is not seasonality. And irregularities which sometimes is present due to a single event which makes the series having strange increasing or decreasing entries.

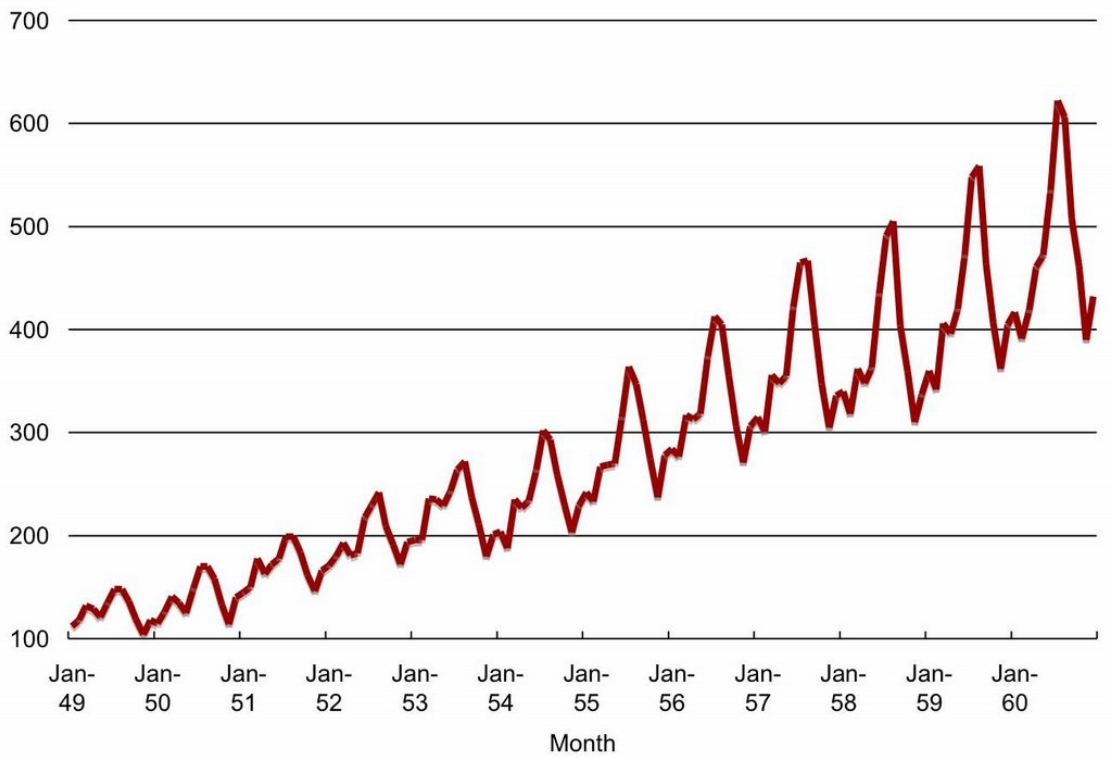


Figure 4.4: **A general example of time series plotted in a line chart [23]**

5 Implementation

In this chapter it is explained how the ideas from the previous chapters was implemented in the project. The chapter is concluded with the description of the tests performed to ensure that the implementation works as planned.

5.1 Scraping

Scraping is the most important task of the program, without data the predictor would not be able to say anything about flight prices. Therefore we decided, as described in Section 3.3, to use Selenium and benefit from it's advantages. Selenium is an advanced browser automation framework primarily designed for automated website testing, but is capable of web scraping as well. Selenium can automate several browsers such as Google Chrome, Firefox or Internet Explorer. We chose to automate Google Chrome, as this is the browser we are most familiar with and there are some problems with automating Firefox due to recent software updates to it.

Using Selenium has a few disadvantages. As the web pages in most browsers are loaded partially at first (i.e. that the page is displayed before it is fully loaded), it can be very easy to create race conditions in browser-based web scrapers, such as Selenium. This is due to web page elements not loading at the same speed every time the page is loaded. Another large disadvantage with scraping in general is that it relies on the implied structure of the web page. As HTML does not carry any semantics, it is necessary for us to interpret the website manually and build the web scraper from these observations. This means that the web scraper will be developed for a specific website design and any significant design changes will break the web scraper.

We locate elements on the web page using XPath, a language for selecting nodes in a XML or HTML document. This makes it possible to locate elements on the web page in a concise way. Unfortunately, it also means that we depend on the design of the website, as we use class names and element IDs that Skyscanner may change at any time. We use this to navigate, fill out forms, and read the web page elements.

Skyscanner's website has a button that from the list of flights for a single day advances to the flights the following day. We wanted to use this to more efficiently navigate between days for a select route. Our web scraper is therefore designed as an iterator of lists of flights, where each list represents the available flights for each day. The web scraper is initialized to a date, an origin, and a destination. The `next` method then returns the flights occurring on that day. Calling `next` again returns the flights the day after, and so on.

In Listing 5.1 is a sample of how the web scraper is implemented using Selenium. In this code we navigate to the Skyscanner website, waits for it to load, and then fills in the search information before pressing the search button. As mentioned before we need to be careful with our timing as it is easy to introduce race conditions into the system. Because of this, the first thing we do is to wait for a specific element to appear. We have observed that when this element appears, the preceding elements will have been loaded and thus it is safe to query the other elements on the page.

```
1  import org.openqa.selenium.*;
2
3  public class SkyScannerScraper implements
       Iterator<List<SkyScannerFlight>> {
4
5    [...]
6
7    private void search() {
8          driver.get("http://www.skyscanner.dk");
9
10         // Waiting for any string in departure to be present
               before it clears and writes in the first field.
11         new WebDriverWait(driver, 10).until(new
               Predicate<WebDriver>() {
12
13             @Override
14             public boolean apply(WebDriver input) {
15                 return !driver.findElement(
16           By.id("departure-input")).getText().equals(null);
17             }
18         });
19
20         select(driver.findElement(onewayElem));
21         select(driver.findElement(directElem));
22
23         writeSuggest(driver, By.id("departure-input"), from,
               By.xpath("//*[@id='from-autosuggest']/*"));
24         writeSuggest(driver, By.id("destination-input"), to,
               By.xpath("//*[@id='to-autosuggest']/*"));
25
26         WebElement date = driver.findElement(calendarElem);
27         date.clear();
28         date.sendKeys(when.toString("dd-MM-yyyy"));
29
30         driver.findElement(searchElem).click();
31     }
32
33   [...]
34 }
```

Listing 5.1: **Section of SkyScannerScraper.java**

5.2 Database

The database is implemented based on the considerations made in Section 3.4 and Section 3.5.

To better suit our needs with testing and evaluation two separate databases was setup; a scraping database and testing database. The scraping database contains all the information about flight prices gathered from Skyscanner and the testing database contains a static subset of the same data used for integration testing for various parts of our system that has to work with the scraping database.

5.2.1 Scraping database

The scraping database contains all the data collected by the web crawlers and is therefore updated with several thousand data point every day. To reduce redundant data (database normalization and normal forms [18]) the data was divided into four tables, each containing different data, as seen in Figure 5.1. The *Airlines* table contains the name of known airline companies. The *Airports* table contains all data concerning the airports. The *Flights* table contains all data concerning flight routes, however, when referencing the origin/destination airport and the airline it links to the *Airports* and *Airlines* table respectively. Lastly, *Flight_prices* stores the price for a given time of one of the entries in the *Flights* table.

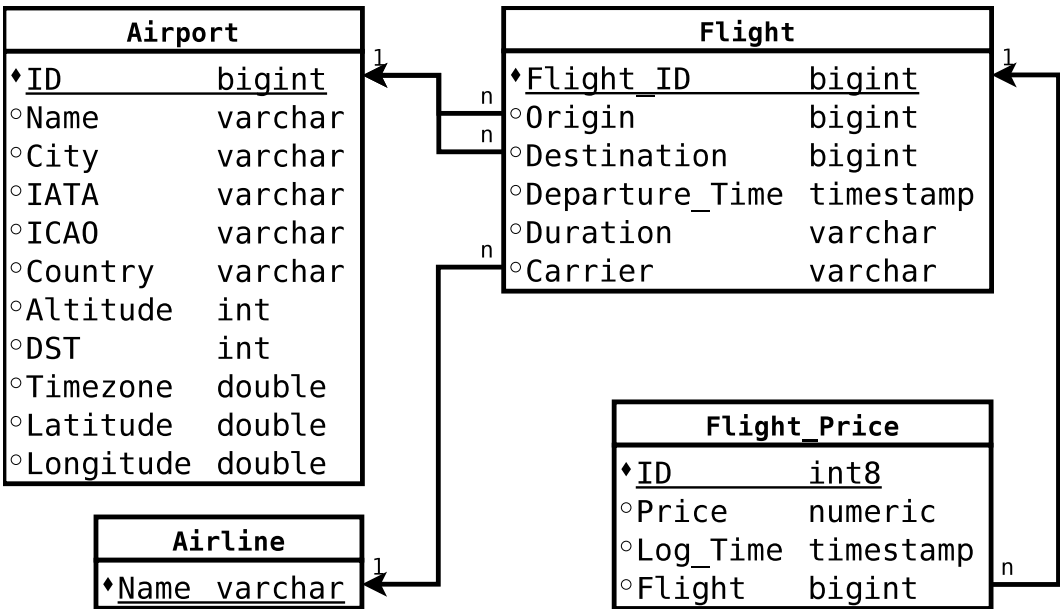


Figure 5.1: **Entity-relationship diagram over the database**

5.2.2 Testing database

The testing database has the same tables as the scraping database as the purpose of this database is to act like a static and limited version of the web scraper database. It contains a full copy of the *Airlines* and *Airports* tables, but only has a select set of entries from the *Flights* and *Flight_prices* tables, thereby the testing database is a partial copy of the scraping database. This is to ensure that we can always know the expected output from the testing database when running a specific query as it is not updated every day like the web scraper database. By knowing exactly what the database returns, we can perform correctness tests of all parts of our system that has to interact with the scraping database without having to use the very large scraping database itself or risking unintentionally modifying the scraping database.

5.2.3 Index

To speed up the lookup of values in the database, indexes [27] can be used. This can be used to decrease the lookup time for larger queries substantially. Indexes work by adding information on certain values, much like that of a book, where you can look up frequent topics and find all the pages where this topic is used. This of course sacrifices some performance when updating the database tables, but instead gains performance when looking up values.

For example, a query for a single item might have to look through the entire table to find an item ($\mathcal{O}(n)$ time complexity), while a similar lookup with an index can do a binary search through the index ($\mathcal{O}(\lg n)$ time complexity). This can especially make a huge difference when dealing with more complex queries. For example a query taking $\mathcal{O}(n^2)$ time might be possible in $\mathcal{O}(n \lg n)$ time.

This project uses PostgreSQL's B-tree implementation of indexing, which enables a lot of comparison operators, including `<`, `<=`, `=`, `>=` and `>` as well as constructs equivalent to combinations of these, including `BETWEEN` and `IN`. Indexes have been used as the web scraper only updates the database with a few hundred entries at a time, where the predictor might have to use all of the data at a time. As such we expect the gain in read performance to outweigh that of the loss in write performance.

Implementing indexes for flight_prices, using b-tree with flight_id, log_time and price as attributes, resulted in a huge speedup for certain queries: we had queries that took more than 20 minutes to execute before implementing indexes, that afterwards only took about 10 seconds.

5.2.4 Hibernate

As mentioned in Section 3.5, we implemented Hibernate on top of our database. Hibernate uses a customized SQL-like language called HQL that is the translated SQL suitable for whatever database is in use. This has the benefits of making it easy to use the database in our object-oriented program but also introduced a few problems. For example we had at first used the HQL query in Listing 5.2.

```
1 SELECT new ClassifierDataRecord(record.flight, record,
2   (SELECT MIN(price) FROM FlightPriceRecord AS d WHERE
       d.timestamp > record.timestamp AND d.flight = p)
3 )
4 FROM FlightPriceRecord AS record
5 WHERE record.flight.departureTime < :lateDate
```

Listing 5.2: **Original HQL query**

The `new ClassifierDataRecord` syntax is a simple construct that makes the results return as a list of objects, instead of a list of arrays. This makes it more ergonomic to use in our code.

Unfortunately, Hibernate does not handle this well. Executing this HQL query results in Hibernate first executing the outer select statement, and then for each row returned it will query the database for the subquery. As the outer query can return more than 200000 rows, this will in turn execute just as many queries on the database which will take a very long time to execute.

If we instead structure it like in Listing 5.3 where we do not cast the results to an object, Hibernate will instead include the subquery in the outer query, resulting in the database only receiving a single query.

```
1 SELECT record.flight, record, (
2   SELECT MIN(price) FROM FlightPriceRecord AS d WHERE d.timestamp
       > record.timestamp AND d.flight = p)
3 FROM FlightPriceRecord AS record
4 WHERE record.flight.departureTime < :lateDate
```

Listing 5.3: **Modified HQL query**

The modification of the HQL query result in far better performance, but means we have to cast the HQL results to the desired objects in our Java code instead which is slightly less ergonomic.

These two queries are semantically almost identical, but the performance differences are much more than one would expect. We consider this to be either a performance bug or a bad result of the query planner.

5.3 Prediction

Predicting when to buy a ticket for a specific flight is a central part of the project, and has throughout the development been explored in many directions. The following section will describe how the different methods of prediction have been implemented. This implementation is based on the conclusion from the experiments done in the Weka explorer application as further discussed in Subsection 6.1.3.

5.3.1 Classification

Weka is a Java library containing a large collection of machine learning algorithms that can be used for data mining tasks. It is developed at the University of Waikato, New Zealand [24]. In this project, the Weka library was mainly used to implement classification based prediction, a group that decision trees falls into.

All of the Weka classifiers use a common class called `Instances` that can be used for both training and predictions, making it easier to test a lot of different methods once we found a way to convert our data to this specific class. Setting it up was easy: Convert the data you want to use into Weka `Instances`, then pick a method from the Weka library and train it with the converted data. You now have a classification predictor that you can use to predict the outcome of future events. All you have to do is convert the data point you want to classify/predict to a Weka Instance and ask the classifier to perform the classification.

Although the use of the Weka library is easy, getting good and accurate results is hard. Many different combinations of methods and settings has to be tested to get good results for a given set of data. Although Weka's `Instances` class can be used by almost all of the different prediction methods it is still a time-consuming task to find the good results.

The main use of Weka in this project has been during the empirical study (see Section 6.2) where the simulation performed interfaced with Weka as seen in Figure 5.2.

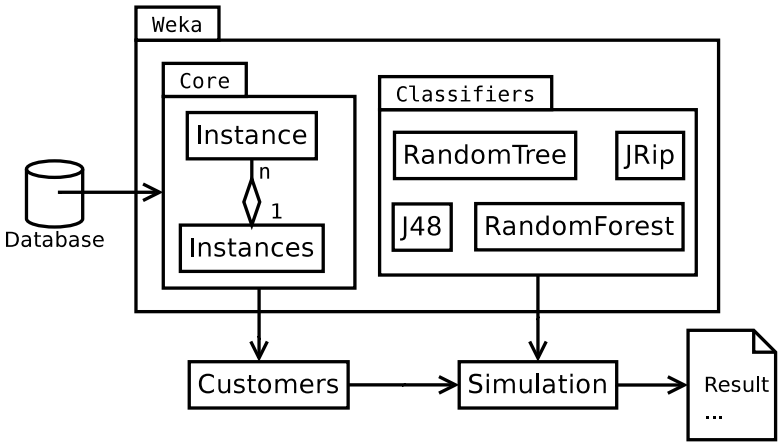


Figure 5.2: **Implementation of Weka in simulation study**

We convert a collection of prices from the database to a Weka Instances object. This object is used to create a collection of virtual customers that also holds some more information about how the simulation should be performed. These customers are parsed to a `Simulation` together with a Weka classification method which produces a result of the simulation. Using the same set of customers with different classification methods we were able to determine the most effective prediction method for that specific set of customers. By instead partitioning customers (e.g. limiting them to a specific route or airline carrier) we can evaluate under which circumstances the different classification methods are most useful and what kind of data to use to get persistently positive results.

5.4 Task distribution

As discussed in Section 3.8, distribution of scraping and prediction tasks are needed. The implementation of task distribution is done using RabbitMQ as described in this section. Ideally we want to distribute both scraping and prediction training as seen in Figure 5.3, here is also included a predictor database which should contain all predictors. Due to the reasons described in Section 6.3, it has been decided not to partition the data and therfore only a single predictor is needed. Thus the realized architecture of our system is more reminiscent of Figure 5.4.

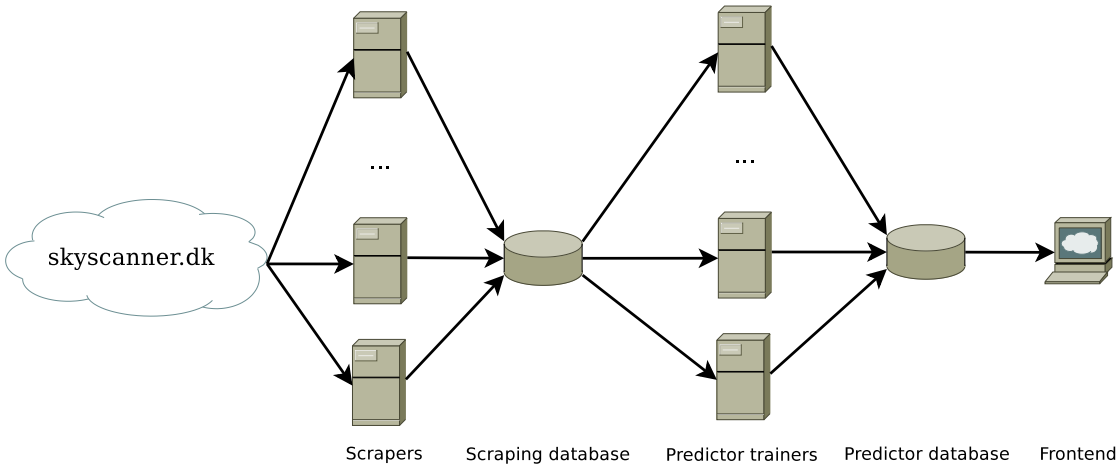


Figure 5.3: **Idealized system architecture**

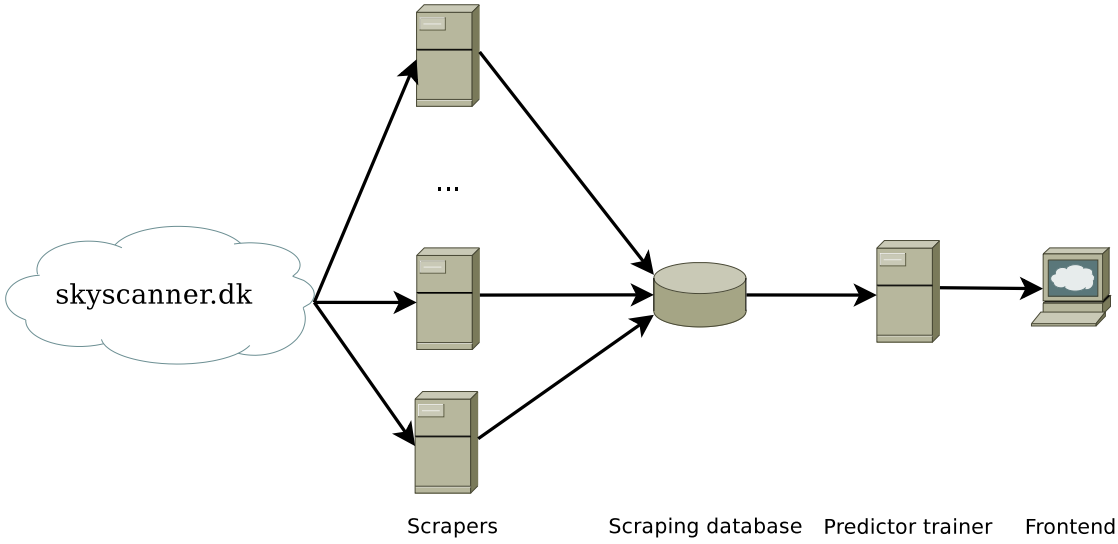


Figure 5.4: **Realized system architecture**

Note that these computers in the previous two diagrams are not necessarily separate physical machines. It is possible to run the entire system on a single machine, but more efficient to distribute it onto multiple machines.

```
1  import com.rabbitmq.client.*;
2
3  public class Sender {
4      private static Channel channel;
5      private static Connection connection;
6
7    public Sender(ConnectionFactory factory) throws IOException {
8        connection = factory.newConnection();
9        channel = connection.createChannel();
10       channel.basicQos(1, true);
11   }
12
13   public void sendMessage(byte[] message, String QUEUE_NAME)
         throws IOException {
14       isEmpty(message);
15
16       channel.queueDeclare(QUEUE_NAME, true, false, false, null);
17       channel.basicPublish("", QUEUE_NAME,
             MessageProperties.PERSISTENT_BASIC, message);
18   }
19   [...]
20 }
```

Listing 5.4: **Section of the Sender.java class**

As we had access to two server machines, we used one of them as the RabbitMQ server host. Setting the server up was not difficult, but reworking a lot of our code to work with the server took some time. We ended up creating a `Sender` and a `Receiver` class. A section of the Sender.java class is shown in Listing 5.4.

A class inheriting from `Sender` is able to send messages to the server. In Listing 5.4, line 6, where the channel variable has been declared the quality of service is set to 1 by `basicQos`, which sets the maximum number of messages that the server will deliver to each consumer to 1 which prevents starvation of a consumer.

In line 9, the function `sendMessage` is declared. This function takes the message as a byte array and the queue that the message will be delivered to as the `String QUEUE_NAME`. In line 12 the queue is declared and line 13 the message is published with the message property persistent, which ensures durability.

The receiving part of the message queue is available from inheriting the `Receiver` class which is able to pull tasks from the server.

The purpose of the `Sender` is to split a big task (e.g. scraping all flights from Copenhagen to London Heathrow for the next six months) into smaller independent tasks (e.g. six tasks scraping flights from Copenhagen to London Heathrow for a month each) and push these tasks onto the server. The purpose of the `Receiver` is to just pull a single task from the server queue, process it, tell the server that the task has been processed and then repeat by pulling the next task from the queue, if any.

As we had only two machines to distribute the work between it was clear that they both should be able to handle both types of tasks in our project, as just designating one machine to do scraping and the other to do prediction would defeat the purpose of having a distributed system as discussed in Section 3.8.

5.5 Testing

This section contains a detailed description of the tests used to ensure that the program works the way it was intended. The tests are divided into testing the following packages: `controller`, `dbstorage`, `messageQueue`, `predictor`, and `scraper`. Each package is tested to ensure that it executes correctly and is compatible with the other packages that it needs to interact with.

5.5.1 Controller

The `controller` package is one of the smaller packages of the system containing the `Controller` class that controls the web scraping by defining what routes to scrape and saves the results to the database. Because it is so small it is only tested by a single integration test that creates a controller and makes it interact with the web scraper and the test database. As the scraping task cannot be simplified and relies on web content that is controlled by a third party, we cannot check that the results are handled correctly, but only check that no errors or exceptions occurred during the interaction.

5.5.2 Dbstorage

The `dbstorage` package provides an access layer to the database used to store all the scraped flights. Testing the correctness of saving, updating, and extracting data from the database is the main focus when testing the package as a whole. Unit tests were created to check that the provided methods for accessing the database do what they should and tests have been created to ensure that the Hibernate ORM has been implemented correctly. It is also tested that data stored in the database can be retrieved correctly.

5.5.3 MessageQueue

The `messageQueue` package implements the access to RabbitMQ and handling of distributed tasks. In testing this functionality we used the already running RabbitMQ server, but used separate message queues made specifically for testing. A simple test ensures that messages can be parsed from a `Sender` to a `Receiver` through the server and that the `Receiver` can acknowledge to the `Sender` that it received and processed the message correctly.

There is also a test to make sure that the `Receiver` understands different messages and that it can receive, decode, and execute a scraping task. This involves starting a controller with the settings provided in a message and acknowledge the message.

Because we had a limited number of computers available, it was not possible to test a physical distribution of tasks between a large group of “slave” machines (workers). Instead we created some tests that emulates a larger number of workers by starting a lot of threaded `Receivers` on a few physical machines. This way we were able to ensure that the task distribution between a “cluster-like” setup of workers would work as well as a single worker. This threaded test was carried out with both 2 and 10 virtual workers.

5.5.4 Predictor

The `predictor` package contains everything relevant to making predictions of future flights and is one of the biggest packages in the system. Although it is large in size, it relies heavily on referenced libraries and does not have a lot of functionality to test. Instead a lot of time has been used to test the efficiency of different prediction methods in certain situations. One test was made to specifically examine the potential and actual savings for a collection of virtual customers using different methods for the predictions. The results and findings of this test can be seen in Section 6.2.

5.5.5 Scraper

The `scraper` package allows us to use Selenium to scrape data from Skyscanner. It has a collection of specific test cases that ensures that the web scraper works as intended when meeting specific criteria, e.g. a flight search returning no matching results. These tests are dependent on the website Skyscanner which might vary from time to time, and as such the tests are based on observations made before making the test cases, e.g. that there are never any direct flights between Aalborg (Denmark) and Beijing (China). This might of course not be the case always, which might "break" the test if one of these assumptions does not hold. The test cases are, as well as the web scraper, of course dependent on Skyscanner's website layout and as such the tests will also fail if the layout should change drastically.

6 Empirical study

To be able to choose whether the project should use regression or classification, experiments had to be made to determine how well both methods could predict values based on the collected data. After choosing a method it is also important to see how well this method performed in practice. This is because prediction performance cannot be measured only in correctly classified instances or how close the prediction is to the actual value. In practice the "performance" is measured in the amount of money saved as well as how many users end up paying extra for their tickets. The initial testing was performed in Weka explorer and explained in Section 6.1 and the more detailed experiments are described in Section 6.2.

6.1 Weka explorer

To evaluate and test which of the methods, if any, would fit our problem of either classifying or predicting price trends for flight tickets we used a tool called Weka. It enables us to input a set of data in comma-separated values format (CSV), as well as other formats, and from these create a predictor using different methods available in the Weka program. We use Weka as it allows us to evaluate a lot of different prediction and classification methods on our data. This makes it possible for us to try out a lot of different configurations for prediction methods as well as how much of our data we should use and in what form we should use it. We have mainly focused on the multilayer perceptron for regression and classification mostly looking at decision trees.

For all different kinds of prediction we used similar datasets of different kind and size as well as different configurations of the individual prediction methods options. We also tried using the dataset differently when testing and training:

- Training and testing on all the data.
- Training only with a subset of the data ranging from 50 to 99% and testing with the remainder.
- Defining a specific training and test dataset in two separate files.

6.1.1 Regression

To evaluate prediction using regression we looked at neural networks (multilayer perceptron) and time series. Time series is used to apply several regression methods, such as linear regression, to time-successive data points, whereas the multilayer perceptron can be used to describe any continuous function. This makes the Multilayer Perceptron a very powerful method in theory.

6.1.1.1 Multilayer perceptron

The multilayer perceptron is a neural network implementation in Weka which can utilize multiple hidden layers, each with a specific number of neurons as described in Subsection 4.3.1. The network can be configured in a lot of ways and have the following options in Weka.

- Hidden layers - The number of hidden layers as well as the number of neurons in them, denoted by a, b, c where a is the first hidden layer with a neurons, b is the second and so forth.
- Learning rate - The rate with which the weight of the neurons will be updated upon finding an error.
- Momentum - Adds speed to training, as it adds a part of the already occurred weight changes to the current weight change. If too large, however, the network will not converge.
- Training time - The number of iterations of training to complete.

The multilayer perceptron method could both be used for classification and for predicting specific values. For predicting specific values we had a relative absolute error of 60-120% using the methods described above. This amount of error is too large for us to use in an actual implementation.

As for classification we could for certain datasets get slightly better results. We found that we were able to predict about 80% cases correctly for most datasets, which seems pretty good. However our data shows that for ~50% of the flights it's better to just buy as early as possible. So even with 80% of cases correctly classified the result is only about 30 percentage points better than a naive predictor that just tells you to buy instantly.

6.1.1.2 Time series

Time series in Weka rely on different regression methods available in Weka to predict future entry points. This ranges from Linear Regression to Multilayer Perceptron as well as a number of other prediction methods. Time series uses a series of entries with equally spaced timestamps. The Weka implementation does not however support using data with more than one entry per timestamp, as such the data have to be structured such that it is a continuous line of data with one entry per timestamp (e.g. a single flight where there is a single price point per day). This way it only considers a single flight alone and does not use any data from other flights. Because of this the results below are very limited as only a selected set of flights where used.

The best results for time series was obtained using data for flights with little or no price fluctuation. Here time series, using Multilayer Perceptron regression methods, could estimate the last 5 days with a rather high precision.

For other flights with larger fluctuations in prices time series was largely off however. During the experiments the error rate was at 80%, or more, most of the time. We think that this is due to the prediction method only taking a single flight into consideration and as such cannot relate the current data to similar data from earlier flights.

6.1.2 Classification

For classification we mainly evaluated decision tree algorithms using the J48, RandomTree, and RandomForest methods. The Ripper rule learning was also evaluated using the Weka implementaion JRip. These methods are a bit simpler in configuration and did not have as many relevant options for us to configure. We used these with data specifying for each timestamp, a certain price and such, whether the customer should buy now or wait. This value would be true if the current price is the lowest price for any given timestamp after the current and false if there would be a cheaper price in the future.

6.1.2.1 J48

J48 uses the C4.5 algorithm described in Subsection 4.2.2 and the following options was explored in Weka.

- binarySplits - Forces binary splits on all nominal attributes
- confidenceFactor - Factor used for pruning
- numFolds - Determines amount of data used for pruning, one fold is used for pruning and the rest for training
- unpruned - If set to true no pruning is performed

For classification with the J48 method we were able to obtain a rate of 84% correctly classified instances in the best case.

6.1.2.2 RandomTree

RandomTree uses the methods described in Subsection 4.2.3 and the following options was explored in Weka.

- KValue - Sets the number of randomly chosen attributes. If 0, $log_2(number\_of\_attributes)$ is used.
- maxDepth - The maximum depth of the tree, 0 for unlimited
- numFolds - Determines the amount of data used for backfitting. One fold is used for backfitting, the rest is used for training. 0 for no backfitting.

For classification with the RandomTree method we were able to obtain a rate of 81% correctly classified instances in the best case.

6.1.2.3 RandomForest

RandomForest uses the method described in Subsection 4.2.4 and the following options was explored in Weka.

- maxDepth - The maximum depth of the tree, 0 for unlimited
- numFeatures - The number of attributes to be used in random selection
- numTrees - The number of trees to be generated

For classification with the RandomForest method we were able to obtain a rate of 81% correctly classified instances in the best case.

6.1.2.4 JRip

JRip uses the Ripper rule learning method described in Subsection 4.2.5 and the following options was explored in Weka.

- folds - Determines amount of data used for pruning, one fold is used for pruning and the rest for training
- optimizations - The number of optimization runs

For classification with the JRip method we were able to obtain a rate of 83% correctly classified instances in the best case.

6.1.3 Conclusion

As a conclusion to the experiments conducted in Weka explorer we conclude that regression prediction is too inaccurate to be useful. Therefore the project group have chosen to implement classification as we believe that this would ensure us bigger savings than a regression based predictor. This choice is made on the fact that regression predictions with an error of even just 50% could result in a customer waiting for a predicted lower price, where the actual price might actually be much higher than the predicted. As such a classification method should be more accurate as the tested correctly classified instances are above 80% for all the methods we tested. Because of this it is fair to assume that the predictor will get a correct result most of the time and incorrect results would result in the predictor choosing to buy a couple of days early or late.

6.2 Simulation

To evaluate the actual utility of the prediction methods several simulations have been made. The purpose of the simulations was to simulate the price search of a lot of virtual customers. These customers would use the different prediction methods to predict the price and trust the predictor 100%. When the prediction method would tell the customer to buy the ticket the customer would. Every customer has the following attributes:

startLooking The date on which the customer starts looking for tickets for the flight

departureTime The time of departure for the flight the customer wishes to book. Must always be after the startLooking.

from The departure airport of the flight that the customer is interested in

to The destination airport of the flight that the customer is interested in

carrier The selected carrier which the customer will book the flight from

relevantData All data from the database related to the flight that the customer wants to buy. This data is needed when simulating the customer as we need the actual price from a specific day to give the predictors as input. This will also tell us what the actual price was when the predictor tells the customer to buy the ticket.

6.2.1 Approach

The simulation will run through every customer one by one and for each date from the startLooking date ask the predictor whether to buy the ticket that date. This continues until the predictor tells the customer to buy now or until the departureTime date has been reached. The price that the customer pays for the ticket is stored. This price can be found in the relavantData for the customer. The minimum potential price as well as the initial price for each customer is also stored so that it may be compared to the actual buy price later. The minimum price is the lowest recorded price, following the startLooking date, for the flight that we know of. The initial price is the actual price when the customer starts looking. The simulator also calculates a Boolean variable indicating whether the price is a "good" price, being true if the buy price is within 1% of the minimum potential price.

For the simulation we generated just over 3,000 virtual customers. For flights with a departure time on or in between 19-11-2014 and 26-11-2014 a customer was generated for each price record that we have obtained in the period 28-10-2014 and 06-11-2014. In other words customers were generated for flights with departure from 19-11-2014 to 26-11-2014 where each customer would start looking for a ticket on a date between the 28-10-2014 and 06-11-2014. We had to make sure that no two customers were looking for the same flight on the same date, as that would invalidate the results of the simulation. The number of virtual customers for each airline carrier can be seen in Table 6.1.

Airline	British Airways	KLM	SAS	Lufthansa	Air France
Customers	470	470	1390	413	320

Table 6.1: **The virtual customer distribution used in the simulation**

As it is shown in Table 6.1, 45% of all customers is buying SAS tickets because most of the data collected using the web scraper is SAS flight. It also makes sense since one of SAS headquarters is CPH.

6.2.2 Results

To find the best classification method several simulations were made using different implementations of decision trees such as J48, RandomTree, RandomForest, and one with the rule learning algorithm JRip.

The results of the simulations are show in Table 6.2, Table 6.3, Table 6.5, and Table 6.4. A more visual representation of the results can also be seen in Figure 6.1 and Figure 6.2. Generally the simulation results showed that no method was as good as we had hoped. For our virtual British Airways and SAS customers no method could average better than the initial price, while our KLM customers were able to save money on average with 3 out of 4 methods. The Lufthansa and Air France customers always saved money on average, with Air France customers having the largest average savings of 139 DKK per. customer when using the JRip method.

The percentage of customers that received a "good" price with the different prediction models are shown in Table 6.2. The green entries indicates that the method is on average better than always buying straight away. The red entries indicates that the method is worse and that it would on average be better to buy at the initial price.

Airline \ Method	*Initial price*	RandomTree	RandomForest	J48	JRip
British Airways	*76.60%*	73.40%	73.62%	72.13%	73.40%
KLM	*81.70%*	95.53%	91.28%	80.64%	88.09%
SAS	*87.55%*	84.24%	84.24%	84.60%	86.55%
Lufthansa	*13.56%*	20.34%	20.34%	40.92%	60.29%
Air France	*19.06%*	75.00%	75.00%	96.88%	87.19%
Customer average	***64.88%***	**71.46%**	**73.31%**	**74.09%**	**77.74%**

Table 6.2: **The percentage of customers receiving a "good" price when using the different classification models**

As seen in Table 6.2 none of the methods was able to get the customers better prices for SAS and British Airways. The initial price for British Airways, KLM, and SAS flights are "good" prices for the majority of the customers, which indicates that they have few opportunities for price savings. This is more visible in Table 6.3 where the average customer savings and losses are seen.

Airline \ Method	RandomTree	RandomForest	J48	JRip
British Airways	7.83	7.34	14.91	27.42
KLM	-34.53	-24.41	3.13	-14.98
SAS	1.05	2.02	1.41	0.75
Lufthansa	-4.07	-4.22	-50.38	-63.97
Air France	-104.37	-104.37	-139.09	-115.80
Customer average	**-15.07**	**-13.18**	**-17.92**	**-18.47**

Table 6.3: **Average difference from initial price and purchase price in DKK. Negative numbers indicates money saved, positive numbers indicates money lost.**

In Table 6.3 numbers below zero indicates that the amount of money saved by the customers traveling with the specified carrier is greater than the amount of money lost. Numbers above zero indicates that customers lost more money on average than they saved resulting in an overall loss for customers traveling with that specific airline carrier.

Although the average amount saved or lost per customer should be a good indication of what method is best to use in practice, it is also relevant to see how much better they could be. One way of evaluating this is to look at how many customers received a price reduction and compare the amount to how many customers that could potentially have saved money. This result can be seen in Table 6.4 where the green entries indicates the method that has the most customers flying with the specific airline carrier who saved money and the red entries indicates the method with the fewest customers who saved money.

Airline \ Method	RandomTree	RandomForest	J48	JRip
British Airways	7.32%	7.32%	19.51%	14.63%
KLM	44.57%	33.14%	0.57%	20.57%
SAS	14.56%	12.86%	12.38%	8.50%
Lufthansa	30.85%	30.85%	81.59%	89.05%
Air France	80.82%	80.82%	81.45%	79.56%
Average	**36.92%**	**35.03%**	**46.36%**	**48.95%**

Table 6.4: **Percentage of customers that saved money out of all possible customers that could save money**

Note that while the J48 method has a decent overall score it is exceptionally bad at generating any price saving customers traveling with KLM. It should also be noted that while JRip is the worst method to use with Air France customers its result is not bad; making sure that almost 80% of all the customers that could have saved money actually does. Generally no method excels at generating price saving customers for Air France which might indicate that it is relatively easy to predict Air France flights compared to, say, SAS or British Airways.

To fully evaluate the methods it is also necessary to know not only how many customers that saved money, or what the average saving/loss per customer was, but also how many customers were misclassified so badly that they ended up buying their ticket for more than the initial price. The percentage of customers that lost money as a result of using the different classification methods are seen in Table 6.5 where the best and worst method for a given airline carrier are marked with green and red respectively.

Airline \ Method	RandomTree	RandomForest	J48	JRip
British Airways	5.32%	5.11%	8.94%	6.38%
KLM	2.55%	2.55%	1.49%	1.49%
SAS	8.42%	7.77%	5.47%	3.88%
Lufthansa	20.10%	19.85%	19.61%	11.38%
Air France	3.13%	3.13%	3.13%	9.69%
Average	**8.06%**	**7.70%**	**7.05%**	**5.52%**

Table 6.5: **Percentage of customers losing money when using the different prediction methods**

Interestingly all four methods seems to be worst for Lufthansa customers, with one in five customers losing money when using RandomTree, RandomForest, and J48, and one in ten customers losing money when using JRip. All four methods were able to generate a positive result in regards to the average savings per customer as seen in Table 6.3. Again we see a generally good result for Air France customers which further supports that it might be easy to predict when the price for Air France flights are optimal.

To better visualize the results of the simulation we created two histograms seen in Figure 6.1 and Figure 6.2. In Figure 6.1 the data from Table 6.3 has been converted to better get a overview of the different methods ability to generate savings.

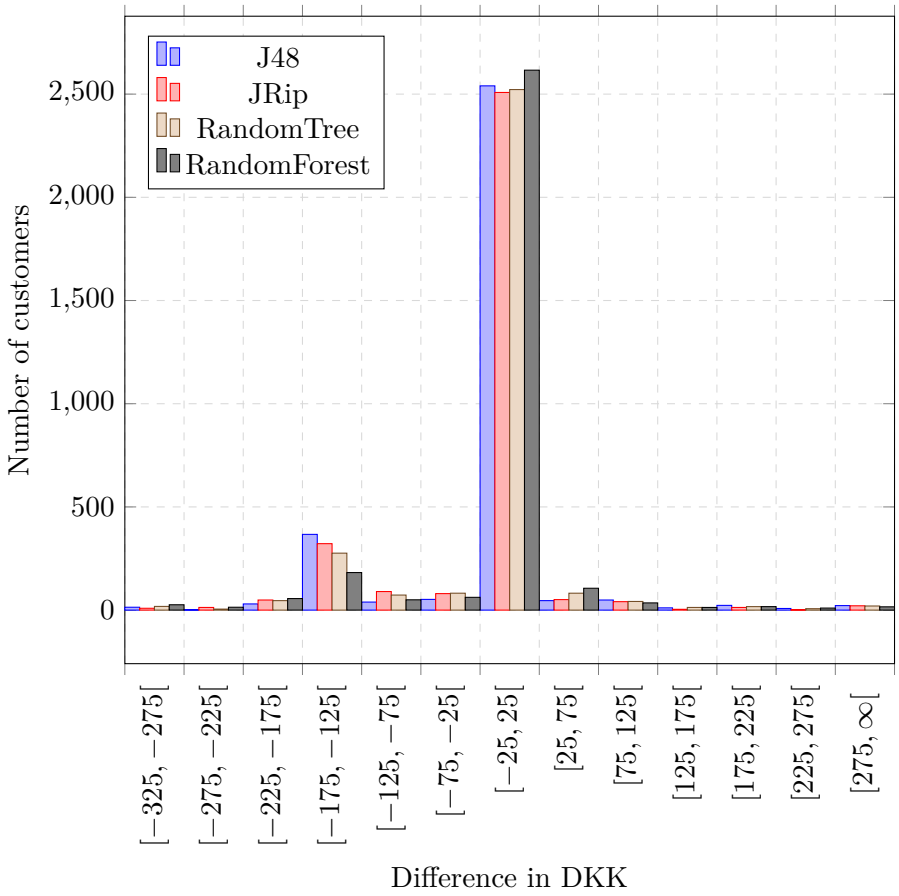


Figure 6.1: **Histogram showing the difference between the price when the methods suggests buying and the initial price (< 0 is a decrease in price, > 0 is an increase).**

With this data it is easier to see that there generally is not money to save. The four big pillars in the center of the graph shows the amount of customers who paid the initial price ±25 DKK. Also worth noticing is that there is a group of customers that saves around 125 to 175 DKK on their ticket. The right side of the histogram shows how many customers lost money and how much they lost. While it is a little hard to see from the figure the J48 method is, while good at generating savings, also the best method to use if you want to lose money. The JRip method is still very good at generating savings, and even better than J48 in some cases, but does not cause as many customers to lose money as J48 did.

While the absolute price difference between the initial price and the buying price is interesting to look at, it does not tell whether a price reduction of 100 DKK was for a ticket that costs around 500 to 800 DKK or if the price was closer to 2000 to 3000 DKK. In the first case the savings would be more noticeable than in the latter case, where a price reduction around 100 DKK might not matter much. To better visualize whether the reduction in price is noticeable we created a histogram similar to that of Figure 6.1, but with the savings/loses relative to the initial price instead of the absolute price difference. This histogram is found in Figure 6.2.

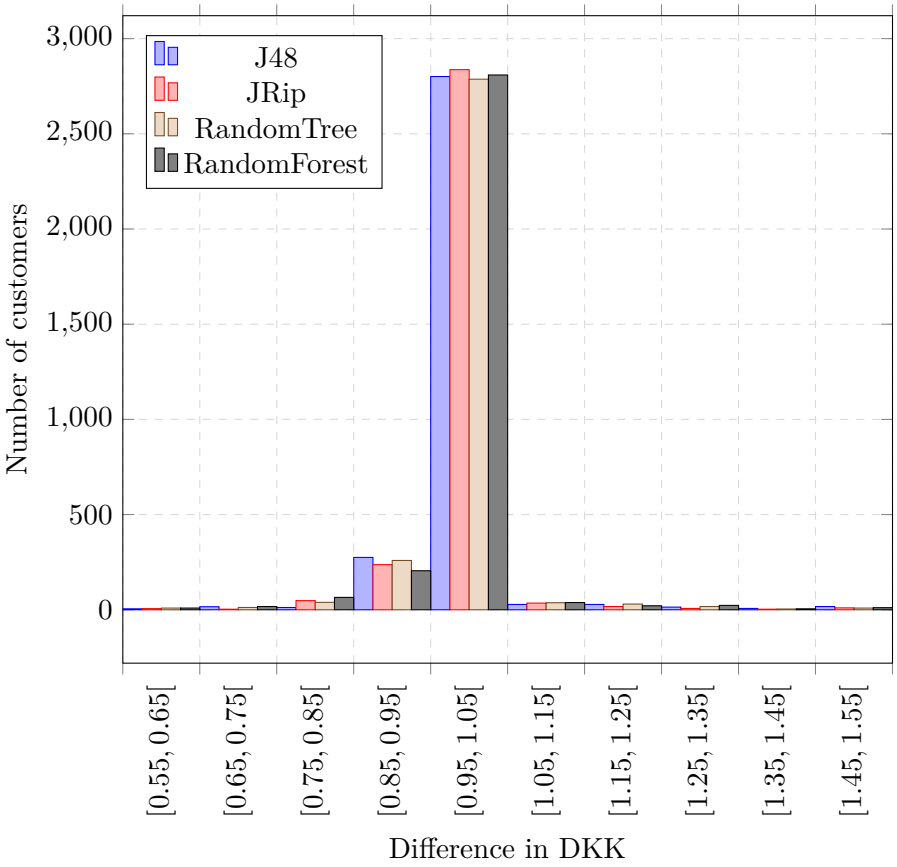


Figure 6.2: **Histogram showing the relative difference between when the methods suggests buying and the initial price (< 1 is a decrease in price, > 1 is an increase).**

6.2.3 Conclusion

The JRip method proved to be the best method overall by generating a total of 56,587 DKK in savings distributed between the 3000 customers, while the worst method, RandomForest, only managed to save 40,365 DKK.

Looking closer at the JRip results we see that the total is composed of 27,491 DKK in loses and 84,078 DKK in pure savings. This means that there is still quite a way to go to create the perfect flight price predictor. Furthermore it is possible to save an extra 88,333 DKK for the customer set used in the simulation. This means that while the JRip method is the best method we have found it still only manages to generate around 33% of the potential savings. This number varies greatly when looking at the savings produced for each airline carrier with Air France savings reaching 81% of what is possible to save. When combing the methods as further discussed in 7.1 and also choosing the right partitioning (see 6.3) it is possible to save 65% of all the money that can potentially be saved.

If you were to exclude the SAS and British Airways customers, as there does not seem to be a way for them to save a lot of money, the overall result gets slightly better. As a result of this there is little to no market potential for deploying the methods, that we have tested, to be used for predicting prices for either SAS or British Airways, as the customer would have a high chance of losing money. There is however also possibilities for savings when traveling with either SAS or British Airways, but when the risk of loss is greater than the chance of saving few customers would actually use the service.

Another idea is to offer the customer a compensation in the case that the predictor caused the customer to lose money. Whenever the predictor made a mistake the company offering the service could offer to pay the difference to the initial price. This way all customers would be guaranteed a maximum price corresponding to the initial price and in some cases a price saving. This would however require that the customers that actually saved money forfeited some of their savings to cover the loss of the customers who should be compensated, but as long as the average savings are above zero this is possible without loss of revenue.

6.3 Evaluation of partitioning

We claimed in Subsection 4.1.1 that not partitioning the data is better than partitioning the data. This claim is based on several simulations (as described in Section 6.2) where one partitioned on companies, while the others only took data from one company into account. We observed that while partitioning the data gave slightly better results for some airline carriers the best average result is achieved without the use of partitioning. The results of these simulations are seen in Table 6.6.

Airline \ Partitioning	No partitioning	Partitioning on companies
British Airways	27.42	16.34
KLM	-14.98	9.46
SAS	0.75	1.69
Lufthansa	-63.97	-79.92
Air France	-115.80	-62.03
Average	**-18.47**	**-12.53**

Table 6.6: **Average difference from initial price and purchase price in DKK when using JRip for prediction**

As we also mention in Subsection 4.1.1, we hypothesize that this is caused by the smaller amount of data available for the individual companies. This seems to be true for some of the airlines while it seems to make it worse for others. From this one might conclude that the predictors for some of the airlines benefits from knowing about the price trends of the others, while it to other predictors effectively is noise. In a similar fashion to what is further discussed in Section 7.2 an ensemble of prediction methods could be created to combine the best fit for the different airline carriers and achieve a better overall result.

The total savings for our 3000 customers when no partitioning was performed was 56,587 DKK, while always partitioning resulted in 38,386 DKK in savings. Combining the best results for JRip one can achieve a better results of 68,384 DKK in savings. If however one simple option should be chosen between either partitioning the data or not, the best result would be achieved by not partitioning the data.

6.4 Evaluation of weights

In Subsection 4.1.5 we hypothesized that giving each instance a weight might increase the predictive power of our application. We have run our simulation with and without these different weights in order to gain an overview over what effect these weights might have (see Table 6.7).

Airline \ Partitioning	Not weighted	Weighted
British Airways	5.90	4.66
KLM	4.57	-3.67
SAS	4.22	4.84
Lufthansa	-10.16	-35.36
Air France	-107.93	-114.97
Customer average	**-8.63**	**-13.65**

Table 6.7: **Average difference from initial price and purchase price in DKK when using J48 for prediction**

We observed that all airlines, except SAS, performed better or at least not significantly worse, but since the impact of the SAS results was so small we deem the case negligible. KLM and Lufthansa in particular saw the largest improvements. Using weights secured ~50% more savings on average, but the average impact on a single customer is still pretty small.

7 Conclusion

To conclude the project in general, now that the empirical study is finished, this chapter will contain three sections to sum up the results. Section 7.1 will describe the overall project results and Section 7.2 contains the discussion of the result. Lastly, Section 7.3 will contain an ending conclusion.

7.1 Project results

To measure the success of this project, one has to compare the results with the problem statement of the project. The goal of this project have been to solve the following problem statement:

How can flight price data be collected and used to predict future flight price trends, so that a customer can save money by using the predicted optimal buying time?

Throughout this report, the documentation of the development process of this system has been described, starting with system design and ending with the empirical study of the different prediction algorithms. The final result compares with the ambition of the 90% certainty as seen on Table 6.5 and in Section 6.2 which shows these results for the following prediction methods:

Random tree Using this method 8.06% of all customers lost money while 36.92% of all customers who could save money actually saved money using the predictor. This method is the best method to use when predicting KLM and British Airways flights.

Random forest Using this method 7.70% of all customers lost money while 35.36% of all customers who could save money actually saved money using the predictor. This method is the best method to use when predicting SAS flights.

J48 decision Tree Using this method 7.05% of all customers lost money while 46.36% of all customers who could save money actually saved money using the predictor. This method is the best method to use when predicting Air France flights.

RIPPER rule learning Using this method 5.52% of all customers lost money while 48.95% of all customers who could save money actually saved money using the predictor. This method is the best method to use when predicting Lufthansa flights.

Neural networks Neural Networks were not tested in a simulation as the other methods, however with an error rate of at least 50% in all our initial experiments with regression we found that Neural networks using regression would be nowhere close to reaching our goal of 90% certainty.

As such no overwhelmingly good results were found when using a single method, but combining the methods are possible and will give an overall better result. Some airlines such as Air France and Lufthansa proved to be much easier to predict than other airlines, but these are also airlines that are often more expensive than the others. On average, we could at maximum save about 30 DKK per customer, although we have seen savings on more than 700 DKK for some customers but also loses of far greater size, with some customers losing more than 1000 DKK. Fortunately, the bad results are generally related to the companies that are hard to predict (SAS and British Airways) and for this reason we do not consider our method suitable for these airlines. Using our methods with Air France, KLM, and Lufthansa only, a much better result can be achieved.

As we concluded in Subsection 6.2.3, our predictors saved 65% of the possible money there is to save for our virtual customers. The figure is much higher for e.g. Air France where the predictors could secure 81% of the possible savings. This means that there is definitely room for improvement for some airlines, while it could be hard to get better results for others.

In Chapter 2 we hypothesized that the different algorithms would perform better for certain airlines than others, due to their varying pricing strategies. In Table 6.3, we saw that this is indeed the case. Using an ensemble of different algorithms and partitionings for different airlines, we can achieve the most optimal prediction method. We can see the result of this in Table 7.1. Note that the airline British Airways is missing from this table. This is because we found no method that could give a positive average result for virtual customers traveling with British Airways, and thus the best course of action is to always buy immediately. The results in the table thus represents all but the British Airways customers as they have nothing to gain. A partitioning value of "Yes" means that the method should only be trained with data for the specific airline and a value of "No" means that the method should be trained with all the data available from all airline carriers. The ensemble result seen in Table 7.1 is the best we have found so far.

Airline	Best method	Partitioned	Average price diff.	Losing customers
KLM	RandomTree	No	-34.53	2,55%
SAS	RandomForest	Yes	-0.94	5,32%
Lufthansa	JRip	Yes	-79.92	7,51%
Air France	J48	No	-139.09	3,13%
Average			**-36.66**	4,15%

Table 7.1: **Ensemble prediction using a combination of the best algorithms in Table 6.3**

In addition to the aforementioned goal some more outlined demands for the system were:

– **The system has to collect flight information off the websites of airline travel companies or from a search engine.**

This functionality have been implemented as a scraping task (described in Section 5.1) using the Selenium library. The decision to use the comprehensive Selenium library turned out to be an appropriate decision, even though the Selenium library seemed to be an excessive choice, as it turned out that scraping data from websites using JavaScript gave some unpredicted troubles.

– **The system has to predict if a customer should wait or buy, based on earlier observations. This can be either an immediate purchase ("buy now"), when the system decides that the price is optimal now ("buy when I tell you"), or some time in the future (e.g. "buy in three days").**

This demand is met by the implementation of the classification prediction (described in Subsection 5.3.1), which works by predicting whether to buy now or wait, i.e. "buy when I tell you". The decision to solve this problem over the others was taken based on the observations in Chapter 6 as these problems would not be solvable in a satisfactory way when using regression methods. The regression approach did not seem precise enough to be used for reliable predictions on flight price trends.

– **The system can be distributed on several machines in order to scale to thousands of flight routes.**

The potential to scrape and predict thousands of flight routes makes distribution of web scraping and prediction tasks interesting. As discussed in Section 3.8, Hadoop Distributed File System and MapReduce handling could have enhanced the project's scalability, but due to among others, the choice of not partitioning the data, distribution was only implemented for scraping. The implementation of message queuing, using RabbitMQ is described in Section 5.4, and make scraping tasks scalable to several machines.

– **The system has to save the customer money in the average case.**

As we saw in 6.2.3 it was not possible to secure savings in the average case for British Airways. For the remaining airlines; Air France, KLM, Lufthansa, and SAS, it was however possible to select a prediction method to secure savings on average. In the case of SAS the average savings would however be too low to be used commercially. The predictions for Air France gave the best results, with an average of 139 DKK in savings (which is the equivalent of ~5% of the average ticket price) when using J48.

– **The prediction should be able to assert with 90% certainty that the customer does not pay more than they would without use of the predictor.**

As discovered in Table 7.1 the ensemble prediction method ensures at least 92% certainty for any airline company, as using JRip (the method with the most losing customers) gives an average loss in 7.51% of the cases. The average of the ensemble method however is almost 96% and so lies well above our demand for 90%.

7.2 Discussion

Scraping has been difficult to implement, due to the complexity of scraping from websites running JavaScript. When scraping with Selenium one have to be very careful with timing, as unexpected race conditions may occur. It is also heavily dependent on website design and if the design changes the web scraper would stop working immediately. Furthermore it is difficult to test the correctness of the scraping process as its results relies solely on third party website content. This has made scraping difficult to work with and generally unpleasant. It would have been preferable to use an API, providing the same information as Skyscanner, instead, however the cost of using an API is too expensive for this project.

For the predictors we would have liked to have more data; for a longer time and for more airports. We only managed to get data for two months and only from Copenhagen to four airports in Europe. International flights would have been interesting to look at as well, however we had to limit the number of requests made to Skyscanner (as discussed in Subsection 3.2.1), and as such these were left out.

Hibernate has been a bit problematic, due to the performance problems described in Subsection 5.2.4 among other things. An ORM layer is most useful when the operations on the database is mostly CRUD; Create, Read, Update, and Delete. The only operations we use are inserting rows in our database (creating) and running some advanced queries (reading). We therefore do not benefit as much from an ORM as an application with more CRUD operations would. The little benefits we got from the ORM does not outweigh the overhead of using it.

Initially we focused on creating a predictor using regression to predict the future price of a specific flight price. Being able to predict a future price would also mean that the other problems, being predicting a price trend or whether to buy now or wait, would be trivial. However, as discussed in Subsection 6.1.3 regression is not precise enough to be reliable for predicting prices. As such, we settled for a classification method predicting whether to buy now or wait, as it proved to have a much smaller number of wrong predictions.

Without an ensemble, we only managed to save about 18.5 DKK per flight, but this still only constituted about 33% of the maximum attainable savings. With an ensemble, we were able to save 36 DKK per flight constituting 65% of the maximum attainable savings (ignoring all British Airways flights). There might be more interesting routes where there is more money to be saved than the ones explored in this report. In addition, there might be machine learning methods that perform better than the ones used in this report. One example could be Q-learning that has been used by Etzioni et al. [13] to predict flight price trends.

7.3 Conclusion

During this project, we have utilized a number of technologies that we previously had no experience with. To gather data from Skyscanner we used the Selenium framework for automated browsing (see Section 5.1). The data we collected was added to a PostgreSQL database we designed using the Hibernate framework (see Section 5.2 and Subsection 5.2.4). This data was then used to analyze the pricing strategies of different airlines (see Chapter 2) which revealed that in some cases there were money to be saved by waiting. We considered different models for prediction (see Chapter 4) and evaluated them using a simulation of customers in Section 6.2.

We discovered that the different pricing strategies meant that some machine learning algorithms performed better for certain airlines. For example, JRip performed much worse than RandomTree for British Airways, even though JRip had the overall best results. This led us to believe that it is most beneficial to utilize an ensemble of machine learning algorithms where we select a specific algorithm and partition for each airline. Our evaluation in Section 7.2 supported this hypothesis and resulted in an average saving of 36.66 DKK per simulated customer.

In this project, we only use a few flight routes due to limitations on scraping. Getting more data than what we currently have using scraping is problematic, and other methods will have to be considered if this limitation is to be amended. We have not found any practical and affordable API that we can use to obtain this data.

All in all this project have successfully fulfilled the problem statement in a satisfying manner. The project resulted in some interesting points for further studies, even though the problem concerning prediction is complex.

8 Agile process

During this project period the group have been working on several workshops concerning the course Software Engineering. In these workshops the group have outlined some guidelines for the development and working process, which will be described in this chapter.

The projects working progress have been based on the agile method Extreme Programming, hereafter noted as "XP". The XP working method have not been utilized fully though, as some of the 12 principles of XP have been left out.

8.1 The 12 XP principles

The Planning Game The project was more or less specified beforehand and have a very narrow goal: finding whether you should buy a flight ticket now or wait to a later time and/or predict the future price of a flight ticket. The date of release was also specified by the university and as such the only things left were the scope and priorities of different tasks. The scope was initially set for a solution that could either classify or predict a specific price for any given flight on a lot of international flights, however it was soon realized that this was not possible due to limitations on getting data. Therefore as such the scope was limited to only 4 routes in Europe. The project involved two different tasks, either classifying, e.g. finding whether to buy now or wait, and predicting a future price for flight tickets. The priority of these tasks are firstly to be able to successfully classify a flight and secondly to predict the specific future price.

Small Releases The project have been developed in small stages and have more or less always been functional. All parts of the project software that could be developed separately have been so, e.g. the web scraper was developed first and then development on the predictor started, as such a working web scraper was released before starting the next stage. All of the code have been worked on and improved such that the effectiveness have been improved, without breaking the code.

Metaphor This is one of the principles we have not really utilized, as we could not find any logical metaphorical equivalent for the project. It is a classifier/predictor and as such could perhaps be associated with a weather forecast, e.g. is it going to rain, stay sunny or so, however we think that weather forecasting is much more complex and as such not really a valid metaphor to simplify the explanation on what this project is about.

Simple Design The system have been implemented and incremented iteratively meaning that for every day we have implemented new features these were able to comply with that days goals. Later on these parts of code might be refactored for various reasons, for example better efficiency.

Testing Every part of the code has been tested and experimented on to test the correctness and the efficiency of the individual parts. As the goal of this project is to find a way to predict prices for flight tickets. It is very important that this is done precise, as wrong predictions might lose customers money.

Refactoring The code have been refactored through the entire project. This has mostly been smaller things to improve efficiency of some algorithm or method, but bigger changes have also been made. One of the biggest changes that have been made is that the code at first build on a predictor which returned future price values. This predictor was later changed such that it instead classified whether the ticket should be bought now or not.

Pair Programming This have been utilized to maximize knowledge sharing as well as to produce code of higher quality. Even though we might lose some man hours we have found that we in turn rarely have to rewrite code. We have also found that it helps get more people intimate with the code.

Collective Ownership The project have been uploaded to GitHub (as mentioned in Subsection 3.9.1) which makes it easy to share and contribute to the code base. This makes it easy for everyone in the project group to make changes, if necessary.

Continuous Integration As everyone in the project group can change parts of their code continuous testing is important to make sure the code is stable and completes all tests as supposed upon every contribution to the code. For this a Jenkins (as mentioned in Subsection 3.9.3) server has been configured which takes care of testing the code for every commit made to the projects GitHub repository.

40-hour Week Every possibility for a group meeting with project work has been used, including almost all hours before and after lectures between 9 and 16 as well as days where no lectures would take place. The only exception being days were the group had to be divided as half of the group had to follow one lecture and the other half another, which would sometimes result in the group having no hours were the whole group was present.

On-Site Customer The project had no actual customers, and as such no on-site customer either. This is mainly due to the project being a university project, thus the project guidelines is defined by the university and not by a customer directly. In the project the supervisor has partly been used as a customer though, as material and current progress have been relayed to him and been evaluated on a supervisor meeting every week as well as through e-mail.

Code Standards In this project the Java code standard has been used to keep the code consistent and clean. The project has also used a formatting script which strictly formats the source code to use the code standard.

8.2 Conclusion

Working with XP has given the group a lot of good things, especially pair programming have been nice and helped the group to better understand and share knowledge of the code. Refactoring has also been utilized greatly in our project as we often have had to change parts of our code to use other libraries for prediction to test out different prediction methods.

Using the XP working method has not been entirely without issues though, as the project group have been split in two teams of three when concerning courses, as such the pair programming has not been utilized much in some parts of the code as the group have been working in two teams working on the prediction and distribution parts separately. This has also resulted in less work time as we have tried to work in the group room with the whole group present. This has been hard as half of the group have had to follow one course and the other half another resulting in days where only half of the group were able to be present for project work.

The biggest difficulty when working with XP has been the documentation. There is not much documentation in pure XP, but because this is a university project we have to make a report documenting the project. As such we have had to go beyond XP for writing the report which has resulted in the report being written almost entirely in the end of the project period.

Bibliography

[1] Github's homepage. URL https://github.com/. (Accessed: 2014.12.17).

[2] Skyscanner's website. URL http://www.skyscanner.dk/. (Accessed: 2014.10.07).

[3] Gradle homepage. URL https://www.gradle.org/. (Accessed: 2014.12.17).

[4] Hibernate supported databases, . URL https://developer.jboss.org/wiki/SupportedDatabases2. (Accessed: 2014.12.10).

[5] Hql: The hibernate query language, . URL https://docs.jboss.org/hibernate/orm/3.3/reference/en/html/queryhql.html. (Accessed: 2014.12.10).

[6] Hibernate orm, . URL http://hibernate.org/orm/. (Accessed: 2014.12.10).

[7] Exact implementation of randomforest in weka 3.7, October 2013. URL https://stackoverflow.com/questions/19137243/exact-implementation-of-randomforest-in-weka-3-7. (Accessed: 2014.12.14).

[8] Ambysoft. Impedance mismatch. URL http://www.agiledata.org/essays/impedanceMismatch.html. (Accessed: 2014.12.02).

[9] Jianchang Mao Anil K. Jain. Artificial neural networks: A tutorial. *Computer*, March 1996.

[10] G Cybenko. *Mathematics of Control, Signals, and Systems*. Springer Science+Business Media, 1989.

[11] Google Developers. Qpx express api: Pricing and usage limits. URL https://developers.google.com/qpx-express/v1/pricing. (Accessed: 2014.12.12).

[12] Thomas G. Dietterich. Machine learning research: Four current directions. *AI Magazine*, 18(4), 1997. URL http://aaaipress.org/ojs/index.php/aimagazine/article/download/1324/1225. (Accessed: 2014.12.05).

[13] Oren Etzioni, Rattapoom Tuchinda, Craig A Knoblock, and Alexander Yates. To buy or not to buy: mining airfare data to minimize ticket purchase price. In *Proceedings of the ninth ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 119–128. ACM, 2003.

[14] The Apache Software Foundation. Welcome to apache hadoop! *Hadoop.Apache.org*, 2014. URL http://hadoop.apache.org/. (Accessed: 2014.12.10).

[15] The PostgreSQL Global Development Group. About. *PostgreSQL.org*, 2014. URL http://www.postgresql.org/about/. (Accessed: 2014.12.10).

[16] Apache Hadoop. Mapreduce tutorial. URL https://hadoop.apache.org/docs/r1.2.1/mapred_tutorial.html. (Accessed: 2014.12.17).

[17] M. A. Hall. *Correlation-based Feature Subset Selection for Machine Learning*. PhD thesis, University of Waikato, Hamilton, New Zealand, 1998.

[18] Teach ICT. Database normalisation and normal forms. URL http://www.teach-ict.com/as_a2_ict_new/ocr/AS_G061/315_database_concepts/normalisation/miniweb/index.htm. (Accessed: 2014.10.07).

[19] MongoDB Inc. The mongodb 2.6 manual. *MongoDB.org*, 2014. URL http://docs.mongodb.org/manual/. (Accessed: 2014.12.10).

[20] Pivotal Software Inc. What can rabbitmq do for you? *RabbitMQ.com*, 2014. URL http://www.rabbitmq.com/features.html. (Accessed: 2014.12.10).

[21] Kohsuke Kawaguchi. Meet jenkins. *Jenkins*, May 2014. URL https://wiki.jenkins-ci.org/display/JENKINS/Meet+Jenkins. (Accessed: 2014.12.09).

[22] Adele Cutler Leo Breiman. *Random Forests*. Salford Systems. URL https://www.stat.berkeley.edu/~breiman/RandomForests/cc_home.htm. (Accessed: 2014.12.01).

[23] Oracle Data Mining and Analytics. Time series forecasting 2. Blogspot. URL http://oracledmt.blogspot.dk/2006/03/time-series-forecasting-2-single-step.html. (Accessed: 2014.12.09).

[24] University of Waikato. Weka 3: Data mining software in java. *The University of Waikato*, 2014. URL http://www.cs.waikato.ac.nz/ml/weka/index.html. (Accessed: 2014.11.24).

[25] Sorcha Pollak. 2012 was the safest year ever to travel by plane. TIME.com, 2 2013. URL http://newsfeed.time.com/2013/02/28/2012-was-the-safest-year-ever-to-travel-by-plane/. (Accessed: 2014.12.09).

[26] PostgreSQL. About, . URL http://www.postgresql.org/about/. (Accessed: 2014.07.13).

[27] PostgreSQL. Indexes, . URL http://www.postgresql.org/docs/9.1/static/indexes.html. (Accessed: 2014.12.01).

[28] J. Ross Quinlan. *C4.5: Programs for Machine Learning*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1993. ISBN 1-55860-238-0.

[29] David E. Rummelhart and James L. McClelland. *Parallel Distributed Processing: Explorations in the Microstructure of Cognition: Foundations*. Bradford Book, 1986.

[30] SQLite. About sqlite. *SQLite*, 2014. URL http://www.sqlite.org/about.html.

[31] Wikipedia. Time series, . URL http://en.wikipedia.org/wiki/Time_series. (Accessed: 2014.11.14).

[32] Wikipedia. Artificial neural network, . URL https://en.wikipedia.org/wiki/Artificial_neural_network. (Accessed: 2014.12.09).

[33] Yongheng Zhao and Yanxia Zhang. Comparison of decision tree methods for finding active objects. *Chinese Journal of Astronomy and Astrophysics*, 7:289, 2007. URL http://arxiv.org/pdf/0708.4274.pdf.