Estimating landing time based on historical data

Aerlabs

Jari Bervoets
Eric Dammeyer
Tim Polderdijk
Boris Schrijver
Yin Wu

Combining historical weather data and forecasting, a runway prediction model and live data into a proof of concept for airports and their environment

TU Delft

Estimating landing time based on historical data

Aerlabs

by

Jari Bervoets
Eric Dammeyer
Tim Polderdijk
Boris Schrijver
Yin Wu

to obtain the degree of Bachelor of Science
at the Delft University of Technology,
to be defended publicly on Thursday July 2, 2020 at 16:00 PM.

Project duration:	April 20, 2020 – July 2, 2020	
Thesis committee:	Ir. R. Koster	AerLabs, Client
	Ir. T. Overklift	TU Delft, Coordinator
	Dr. J.S. Rellermeyer	TU Delft, Coach
	Ir. O.W. Visser	TU Delft, Coordinator
	Dr. Ir. H. Wang	TU Delft, Coordinator

TU Delft

1 Foreword

This is the report of our bachelor end project called **Estimating landing time based on historical data**. During this project, we were tasked with creating a web application that can accurately predict the landing times and runways for arriving flights at an airport, including a front end in the form of a graphical user interface. The client for this project is AerLabs, a start-up founded in 2018 that builds digital solutions for the aviation industry. This report is part of the requirements to pass the bachelor end project at the TU Delft.

We feel that this project was a useful learning experience in several areas. Namely, dealing with time pressure, communicating with each other and with the client, and working remotely. Also, the project took place during extraordinary times. The Covid-19 disease impacted both our personal and professional lives. We faced challenges such as working together while we have never met each other physically and working alone from home. Learning to cope with these challenges made it so that this project will not be soon forgotten. Overall we feel that this project was good preparation for our future work.

We would like to thank our TU Delft coach, Dr. Jan Rellermeyer, and our client, Ir. Robert Koster, for the help and supervision they have provided over the past few months. - *23/06/2020*

Contents

2 Summary

In this bachelor end project, a system is developed for the client AerLabs that predicts the estimated landing time of airplanes heading towards Amsterdam Airport Schiphol (AMS) with a mean average error of 39 seconds. In addition to this, a novel method is developed to predict the runway an airplane will land on. The model predicts the correct runway over 80% of the time. The project was initiated to determine the feasibility of a lightweight cost-effective Airport Collaborative Decision-Making (A-CDM) system, using data that is publicly available. This project successfully validates the initial steps of creating such a system by generating the A-CDM milestone Estimated Landing Time.

At the beginning of the project requirements and design goals were set in accordance with our TU Delft coach and the client. We divided the final requirements into three components: an application, machine learning and data.

The application developed consists of a map and a table view showing flights in real-time. In the map, the flight paths are visualized and flights can be inspected for their live information. Additionally, the map is designed to give the least amount of distractions possible, making the application suited for an environment such as an airport operations centre. The predictions and flight information are fetched by a RESTful API server that processes the data. All of this runs on Google Cloud Platform leveraging their services, reliability and scalability.

The prediction models are both implemented as sequential neural networks using regression and classification methods respectively. The models are based on the proposed model by Wang et al. and the trajectory clustering method from Gariel et al.. We extended the proposed methods by adding meteorological data to the feature input set. The models are trained with the last 900 seconds of data from every flight before arriving at AMS. On average over the descending trajectory, the model predicts the correct runway over 80% of the time and predicts the landing time with a mean absolute error of 39 seconds. The work done by Wang et al. scored on average a mean absolute error of 69.19, which gives us a ballpark figure on the performance of our landing time prediction model.

The data component contains data acquisition and processing. The data used is acquired in the form of Automatic Dependent Surveillance-Broadcast (ADS-B) messages from the OpenSky Network [37]. The flight trajectories are extracted from the ADS-B messages with a Traffic library [31]. After extracting the trajectories the flights are processed by cleaning errors and removing noise. Moreover, the flights are augmented with input features that are necessary for the prediction models. One of these features is the cluster of flights with a similar flight path. This is computed by first performing a principal component analysis, and then applying the Density-Based Spatial Clustering of Applications with Noise (DBSCAN) algorithm to cluster the trajectories.

Over a period of 10 weeks, the system was designed, the models tested and the was developed. The team consists out of five participating members. Supervision was given by a TU Delft coach and the final deliverable is this report.

3 Introduction

This report is a document that presents the project **Estimating landing time based on historical data**.

We want to succeed in designing and implementing a system based on requirements posed by our client AerLabs. The main goal we are researching is whether we can accurately predict the landing time and runway of arriving flights at Amsterdam Schiphol Airport (AMS). For an end-user to interact with these predictions and flight's information, a graphical user interface needs to be developed.

As seen in the index, the document is structured in 9 chapters. The detailed core of the report starts in chapter 4 where the requirements are analysed, detailed and structured. The design and implementations are described in chapter 5, here the methodology of every part of the system is presented. Results are presented in chapter 6, after which we discuss whether we've met the requirements in chapter 7. Later there will be a discussion and recommendation section meant for possible future improvement or extension of our work. The potential ethical implications of our final product are also covered in this section. The research paper, info sheet, and Software Improvement Group (SIG) evaluations are included as appendices.

4 Requirement Analysis

The client presented a well-defined set of initial requirements. After doing research and having discussions with both our supervisor and client the requirements were finalized and presented in the research paper.A.4.1 However, these requirements deserve further explanations, require deeper analysis, and need to be structured to meet them systematically.

In this chapter, the requirements formalised in the research A.4.1 are analyzed from a high-level perspective. For this analysis the requirements are divided into three components, respectively *Application*, *Machine Learning* and *Data*. Each component is discussed separately and its corresponding requirements are analyzed. Some requirements might be trivial, but others cause serious implications which lead to additional requirements and design choices that must be defined. The decision was made on discussing the components in top-down order. This way requirements that follow from original requirements can be intuitively introduced and then passed on to the subsequent component for analysis.

4.1. Motivation

The 13 "Must haves", 3 "Should haves" and 3 "Could haves" requirements are spanning the domains of data independence, performance, user interaction, and aerospace specifics. In this data-driven project, we looked at the steps the data passes and identified that we can classify this in 3 parts. Data is sourced from a third party and then has multiple iterative ETL cycles before it is used to train a prediction model. The model, when trained, is used to predict the estimated landing time and runway which are visualized in the front end. Hence we end up with the split *Data*, *Machine Learning* and *Application*.

4.2. Application

The web application is the part of our product that users will interact with. It is the part that clients connect to, and it shows the user all the necessary information. The distinction between what can be considered as part of the application is not strict. Therefore we decided to include the front end, back end and some things we thought best fitted here. In this section, we will first consider the implications of the design goals for the application. How does this affect our design process and what is important to keep in mind? Then we take a look at the requirements of the front end and what the application should look like. Next, we formalize the back end needed to serve the application. The section is wrapped up by explaining how the components of the application work together.

4.2.1. Design goals

Performance One of the design goals we had during this project was performance. For the application, this means we want to make sure the application performs. In other words, it has to keep working smoothly even when there are many concurrent flights. There should be no lag and the loading time of the application should be close to none.

Data integrity Data integrity is another design goal we set. For the application this is interpreted as follows: no matter how many clients there are, any two clients will get to see the same information, and this has to remain consistent across the application as well. For example, a flight that is on the map must be part of the table. Also whenever data is shown to multiple different concurrent users, this must be the same data. This is important because the application is a decision-support tool. Users should make decisions based on exactly the same information.

User experience The main design goal having to do with the application is the user experience. We want to make sure that for any user, it is intuitively clear how to control the application. Ideally, there should be no need for an instruction manual or help. Furthermore, the location where the application will be used is likely an operations centre. Therefore there should not be unnecessary visual distractions. Although this application is not mission critical, it can certainly make the job of an operator easier.

4.2.2. Front end

The front end is the web application that users get to see and interact with. Thus, it is concerned about how to visualize the fetched data in a way that is easy to understand and quick to use for users. Because of that, most of the requirements for the front end are centered around the user experience.

From our client, we already get an idea of how the application should look like. For example, it should have a map and a table to display the data. Furthermore, it should have a playback mode and it can deal with 1000 flights concurrently. In this subsection, these requirements are explained in details. After considering the design goals that we had, they led us to the following requirements for the front end.

REQ 1. The front end must visualize flight paths.
In line with the user experience, the application should show flights visually. To be able to give users a visual representation of arriving flight paths, the front end must have a map, that shows airplanes and their paths. A monochrome map is preferred to a coloured map because it will cause less distraction to users. It is highly recommended to highlight the runways of airports. The airplane that renders on the map could have a yellow colour to make it stand out on the map and a line should be following the airplane to indicate the path. The movements of airplanes should be fluent, without changing position suddenly or unexpectedly. The information about an airplane will only be displayed when the user is hovering over it. In this way, we obtain a clear view for the user. Of course, we need to keep data integrity in mind for this view.

REQ 2. The front end must have a table view with the data and milestones of the tracked flights.
If a user is looking for more information on a specific flight or wants to find it on the map, the front end must have a table view to make this possible. The rows in the table should be at least selectable and filterable. The table must contain columns about the data of the tracked flights. The table should at least display the call sign/name of the airplane. Other useful data about the flight should be also added e.g. whether an airplane is going to land on AMS or not. The table must contain columns about the milestones of the tracked flights. The table must have columns of A-CDM milestones: ELDT and ALDT. This table view is important for the user experience since it is not possible to find all this information in an organized way without such a view. Naturally, we need to make sure that the data displayed in the table is consistent with the data which the map view uses to ensure data integrity.

REQ 3. The architecture must be designed such that the front end can have a playback mode.
In case a user wants to watch the incoming flights from a past timestamp again, both the map and the table should have a playback mode with a date picker. There needs to be an option to query data from a given timestamp. The front end needs to have a clear way to choose a timestamp and use a playback mode. This functionality was required by the client and is one of the things that brings the user some of the needed value. Once again, we need to make sure that in playback mode, the flights that are on the map and in the table come from the same data and do not have discrepancies.

REQ 4. The front end needs to be able to handle 1000 concurrent flights.
If there is a lot of air traffic around the given airport, the application still needs to be able to do its job without

a big drop in performance. When the number of flights grows to 1000, the front end should still be able to visualize all the flights without any visible delay. It also needs to show the flights in a smooth way, without stuttering or seemingly ’teleporting’ flights. If we simply display all the data we get, there are a lot of flights on the map and table that will never land at our airport. So one of the improvements for the performance could be to remove those flights and only show the ones that will actually land at our airport. The user will not have a good experience with the application if we cannot achieve this.

REQ 5. The front end could have a performance dashboard, visualizing the performance of the milestone predictions.
A feature we would like to add is to implement a performance dashboard to display the error rate of the predicated milestones. The user will have more knowledge about how well the predictions have been made rather than just the estimated value. In addition to that, this would improve the user experience by making the application better looking.

4.2.3. Back end

The back end is the part of our application that users cannot see or interact with. It hosts a server that provides correct flight data for the front end to fetch while it ensures the request time will not take too long. This includes the data source and prediction models.

REQ 6. The back end needs to provide correct flight data with added predictions.
In order to show the users the flights, it is important that it receives correct data from the back end. To achieve this the back end will have to be able to ingest data from multiple sources, transform this data, and perform predictions on it. After this, it will have to serve this data through a RESTful API that the front end can call. This includes how accurate the predictions most likely are. For example that a given flight will have 80% probability of landing on runway 36C. The application must get the same data when multiple separate calls to the server are made at the same moment in time. This is necessary to ensure data integrity. In other words, the information provided by the back end should be consistent across our applications. The fetched data should also be complete and correct.

REQ 7. The back end needs to be able to handle 1000 concurrent flights.
Just like the front end performance, the back end containing the prediction models needs to be able to handle the same load, otherwise, the application will lose its use. That means the prediction model server needs to be able to give predictions for up to 1000 concurrent flights. This performance requirement is needed for the design goal of performance.

4.2.4. Automation

In our research report, we did some research on potentially using ’infrastructure as code’ because it is useful when you have to make different components work together. But in our case, due to time pressure, we made a simple API instead, which gathers data, processes it, runs the predictions on it, and can send all of it to the front end by using HTTP requests.

In order to connect the front end, the collected data, and the prediction models we have trained we’ll need to set up some form of infrastructure. This infrastructure will have to be efficient enough to process and serve hundreds of flights at the same time. This means the application will have to be scalable. Thankfully many cloud platforms offer to auto-scale deployed apps on their platforms as a service, so we won’t have to worry about that. Using this API is perfectly in line with our design goals of data integrity and performance. It will ensure the consistency of data in our application, and because it is hosted in the cloud the performance for the back end will be good enough too.

4.3. Machine Learning

Machine learning algorithms to predict traffic patterns and arrival times is an ongoing field of research. Trajectory clustering plays a key role in both fields since it naturally groups flights with the same arrival routes which is key in predicting their landing time. Recently several methods are proposed combining supervised machine learning [45],[20] and Density-based spatial clustering of applications with noise (DBSCAN) [24] for trajectory clustering. There are two main requirements which are in the prediction domain, "*The system must make a correct prediction on at which runway a flight is going to land 90% of the time.*" and "*The system must continuously compute the A-CDM milestone ELDT - Estimated landing time.*". In recent times machine learning algorithms have become increasingly well known as a solution for these problems. We follow the path presented by Wang et al. [45] and use supervised machine learning in this research.

In order to train the prediction models, several steps need to be performed. First and foremost the necessary data needs to be gathered and processed, this is discussed in the next section. After the data is processed the prediction models must be trained. When trained the models can be incorporated in the *Application* so that predictions for a given flight can be made based on its current state.

The core data-set used is based on ADS-B data. This data is represented as a time-series. Each data point holds the state of an airplane for a specific moment in time. All data points sequentially in time for a specific airplane will form a trajectory. This trajectory includes departures, flights, arrivals and on-ground manoeuvres. For this research, we are only interested in the flights and arrivals part of a trajectory. The trajectory needs to be split as such so that only flight time and arrivals are kept. This requirement to the data is further explained in the paragraph below **Extracting trajectories from ADS-B messages**.

The need for a place to train the prediction models other than a laptop to arise from the sheer size of the data we need to process. We hence need to investigate on which cloud platform we will run our training and prediction jobs. This is further explained in requirement 11.

In this section, we will first discuss related design goals. Then challenges that arise from the requirements.

Design goals The *Machine Learning* domain has the following design goals: training time, model size and inference time. The inference time is the time the model needs to provide the result of a single prediction. These three goals are subgoals of the larger *Performance* design goal and offer a more detailed look into what this means for the machine learning domain. In order to make the model iteratively better by using different models sizes or by improving the training set, and combined with the fact that the project needs to be completed within a short time frame of 10 weeks, the time the models needs to train needs to be within bounds. Aiming for a training time within the range of 1 to 3 hours makes that multiple runs can be done per day.

Model sizes can grow really large when using a large number of nodes. This is not uncommon when training highly complex models in the video recognition domain where the feature set is in the millions. Our number of features is relatively small. However it still is good hygiene to keep to model small too.

The inference time is implicitly determined by the requirement 10. The inference time on the model should be low enough to achieve the 1000 flights every set interval. Having a lower inference time also means fewer costs as compute resources are less in use.

REQ 8. The system must make a correct prediction on at which runway a flight is going to land 90% of the time. This will be verified using factual data from the airport. The first model we will need to build is a model that predicts on which runway an airplane is going to land. This is a classification problem. A fixed (small) number of runways are present at any airport. Using ADS-B flight data, weather data and certain static information about the airport these jobs need to be completed. For this supervised learning job, we will process the data to combine ADS-B and weather data at corresponding time-points and use them as features. In the data processing step, we will also need to determine at which runway the airplane has landed. This runway will then be added as a label to all the data points that the track of the flight consisted of.

REQ 9. The system must continuously compute the A-CDM milestone ELDT - Estimated landing time. The second model we will need to build is a regression model in which the Estimated Landing Time (ELDT) is calculated. This is the second stage in the two-stage rocket. That is, we will use the runway as a feature in this model. When doing predictions this runway information comes as a result of the first model. When doing processing the data for the training step we need to specify at which time the airplane is going to land. And add that as a label. In order for a ML model to understand this correctly, the ELDT will actually be the number of seconds before it lands. This will be a continuous number on a scale of 0 to X. With X being the maximum amount of seconds we can predict before it is going to land.

REQ 10. The architecture must be designed such that it can process 1000 flights concurrently at any moment. In line with the requirements, the architecture we use for the models needs to be able to serve predictions for 1000 flights concurrently. Implementing neural networks ourselves to achieve this, is not necessary and would waste a lot of our time and energy. Multiple complete frameworks are readily available to use for creating and training prediction models. We need to determine which one suits our requirements the most. The inference time needed to predict a single flight in line with the requirement to process a 1000 flights.

REQ 11. The system must be designed to use Cloud Technology. After a deep-dive into the requirements surrounding data and data processing the need arose to use Cloud Technology to run this system. The large quantity of data made working on a local device like a laptop unfeasible. Following from 10 we had to acknowledge that both the training and inference steps are not possible on a local device.

4.4. Data

Data is the cornerstone of this project. Data must be extracted, cleaned and transformed for usage in the prediction models. Although requirements on data acquisition and processing are not explicitly mentioned, they are a prerequisite for the success of the project. In this section, we first consider the relevant design goal "performance". Then the data acquisition is taken into consideration. The nature of the data causes challenges which also needs to be addressed. Furthermore, the data must be augmented for the usage in prediction models. However, this is not straightforward and some implicit requirements follow that need analysis. At last in the final section of this chapter, the implications of storing the data at multiple stages for different usages are taken into account.

Performance The design goal related to data is performance. Performance starts at the source. To see why to consider that the data processing propagates through the system. An inefficient step earlier could end up as the bottleneck, slowing the whole system down. It's easy to say that we would like to have a good performance. But what are the steps you can take in order to guarantee a result? In our case, we already know beforehand that there are several critical steps we can take. For instance, we need data, lot's of it. However, ADS-B messages are in its raw form not indexable. In other words, before processing the data we can not select the flights on features such as the arriving airport. Then how can we prevent looking for a needle in a haystack? (flights going to AMS, from all flights at that moment). A solution to this problem will greatly reduce the overhead in the amount of data that needs to be processed.
Furthermore, we know that at some stage we are going to use python libraries such as Pandas and NumPy. Although Pandas provide a great intuitive way of interacting with data, native methods can be slow. Too slow considering the years of flight data we plan to process. A solution would be to use NumPy's vectorization over Pandas vectorization. This can result in a speedup of roughly 8. Finally, we reach for high performance by the process of not stopping when the methods produce results, but by critical examination and performing multiple iterations on the working code.

4.4.1. Data acquisition

The main challenge is that we don't have immediate access to a reliable source of clean data. The data we can get access to is from the OpenSky Network, a non-commercial crowd-sourced ADS-B gathering platform. The OpenSky Network mostly consists of uncertified receivers gathering and relaying the messages. The OpenSky Network gathers these messages, processes them and serves it to its users. The OpenSky Network offers two ways of accessing the data. First, by use of an HTTP API. Here data can be retrieved in near real-time. However, only the most recent data can gather this way. Second, via the Impala Shell. The Impala Shell is a SQL interface on top of a Hadoop cluster. This method of access allows for retrieving the full history of data available from the OpenSky Network. Since we need as much data as possible, we decided on using the Impala shell. Furthermore, the prediction models use meteorological data input features, this must be acquired from a trustworthy source. In this section, the requirements about the data sources and the Impala API are further discussed.

REQ 12. The flight data interface must be independent of the data source e.g. it shouldn't matter if the data comes from a static data import or streaming data.
The core data set that is eligible for this project is Automatic dependent surveillance-broadcast (ADS–B) data. This is a surveillance technology in which an airplane periodically broadcasts information about itself, enabling it to be tracked. ADS-B is an open protocol which can be received and decoded by everyone, given you've got the right equipment. On a protocol level, this independence has hence been arranged. Whether you collect the data yourself or get it from a third party, both are just as viable for use in the project. Whether the data needs to be streamed or delivered in batch depends on the use case. For training ML models batch driven data will be used. When doing real-time interference it will be streaming data.

REQ 13. The data must be enriched with meteorological data at that current time
As mentioned before, the prediction models need this as an input feature. However, this is not part of ADS-B. An algorithm to derive meteorological conditions from raw ADS-B data is recently presented in research by Sun et al. [40]. However, it is more practical to retrieve the meteorological data from a source such as AMS or the KNMI.

The Impala Shell ADS-B data is retrieved as time series data from the Impala Shell and is given in the appendix in Figure A.1. We are especially interested in analyzing several features, each which will be discussed here.

icao24	*ICAO24 address of the transmitter in hex string representation.*
callsign	*callsign of the vehicle. Can be None if no callsign has been received.*

The *ICAO 24* and *callsign* are used for identifying airplanes and flights. ADS-B data does not contain uniquely identifiable flight numbers for every leg. Although the *icao24* and *callsign* address seemingly provide this. However, both are not sufficient for unique flight identification in general.
The airplane's callsign is mission-related and assigned by the airline or military. Because missions, i.e. destinations or purposes of a plane, can change over time. The callsign will change as well. The ICAO 24-bit address is uniquely bounded to the transmitter of an airplane and is part of the airplane's certificate of registration. This doesn't mean that the address stays the same. Airplane's can be sold or leased to other companies, which then reprogram the ICAO 24 address. Flights could be uniquely identified by integrating with external sources. These are however commercial such as FlighRadar 24, or restricted such as the B2B network manager from Eurocontrol. Our goal is to use open data and publicly available sources. Thus our method for trajectory extraction and unique flight identification must rely on processing the data as is.

time_position	*seconds since epoch of last position report. None if no update last 15s*
last_contact	*seconds since epoch of last received message from this transponder*

The time position and last contact are the basis of the time series. This information is fundamental to create flight trajectories from individual messages.

longitude	*in ellipsoidal coordinates (WGS-84) and degrees. Can be None*
latitude	*in ellipsoidal coordinates (WGS-84) and degrees. Can be None*
geo_altitude	*geometric altitude in meters. Can be None*

The position of an airplane is obviously key in the flight data processing. WGS-84 coordinates are the standard for GPS systems and are also used by ADS-B. In WGS-84 the altitude generally corresponds with the mean sea-level of the closest ellipsoid. In the Netherlands, this roughly corresponds with the "Normaal Amsterdams Peil" NAP, which is the reference height used in the local coordinate projection "Rijksdriehoeks Coordinaten" [1]. From the research, it followed that using a locally projected coordinate system is used preferably to preserve distances and angles during the data augmentation process.

baro_altitude	*barometric altitude in meters. Can be None*
on_ground	*true if the airplane is on ground (sends ADS-B surface position reports).*

The barometric altitude in combination with the on_ground attribute may serve as the most important indicator of whether a flight has landed. The difference is that the barometric altitude is actually the altitude in terms of atmospheric pressure measured at a static port outside the airplane. While the geo altitude is the transmitted height above mean sea level from the closest ellipsoid. The barometric altitude often stops transmitting after the airplane's landing gear are down. This roughly happens at the 5KM mark during the final approach and differs for every type of airplanes. Also, it might not stop transmitting at all.

heading	*in decimal degrees (0 is north). Can be None if the information is not present.*

If we consider only one entry from the time series, i.e. one ADS-B message that contains the position of an airplane, airplane flowing in opposite directions could be interpreted the same since there is no notion of direction. Also heading values that change at a fast rate over consecutive messages indicate (sharp) turns which are important for analysing and clustering the trajectory.

4.4.2. Data transformation

The ADS-B messages retrieved from the OpenSky Network contain errors, noise is unfiltered and contains duplicated messages. This data is not reliable enough for use in machine learning and statistical models. As it is said, garbage in, garbage out. First, the data must be transformed into something we can work with. Then the data must be filtered, cleaned. And finally, the data needs to be augmented for the usage in the prediction models. In this section, we will discuss these steps accordingly.

Extracting trajectories from ADS-B messages The data is extracted from the Impala Shell as a time series. To work with this the retrieved ADS-B messages must first be transformed from time series to flight trajectories corresponding with single flights.[2].[3]. One challenge is that the ADS-B data messages are not uniquely identifiable. This is an issue because in order to do trajectory extraction we need to decide: which messages correspond to a flight, when a flight starts and when a flight ends. Notice that this is not unambiguous since an airplane with the same ICAO 24 and callsign will often do multiple legs in one day. We combine and formalize the above in the following requirement:

REQ 14. Flights must be uniquely identifiable and their trajectories must be extracted from ADS-B.

Data filtering ADS-B messages are unfiltered. i.e. if we would acquire a stream, we would collect all messages of the receiver or network in question. This could be great if your goal is to collect as much data as possible. However, we are economically bounded by a budget and therefore need to decide on which data we are interested in. In our project, the models are trained for AMS. Therefore it is required to filter the flights retaining only flights at AMS. Additionally, for the purpose of this project, we are only interested in arriving flights. We formalize this in the requirement stated next:

REQ 15. The data must be filterable on arriving airport, specifically AMS.

In order to do the filtering on the airport, the initial query must also incorporate geographical bounds. To see why consider that an ADS-B message does not contain information about the landing airport. Thus in order to determine the destination of a flight efficiently the following must happen: First the data-set must be reduced by only considering messages in certain geographic bounds around AMS. Then based on the trajectory of an airplane around AMS it can be determined whether the airplane is actually destined for AMS.

[1]https://zakelijk.kadaster.nl/rijksdriehoeksstelsel
[2]i.e. a flight departing from location A to B, without any stops in between
[3]This is also called a leg

Problems with ADS-B ADS-B messages retrieved from the OpenSky Network as is contain multiple errors and noise. To start with ADS-B is still in the implementation phase. Meaning that there is no strict oversight and regulations [4] Some airplanes send out erroneous codes or are inconsistent while transmitting. Second ADS-B messages are 112 bit long messages. The messages can be wrongly decoded, or bit(s) can be flipped during transmission. This may result in spikes and drops or plain false values. In our research, it was suggested to apply a Low-Pass filter given by: (A.2). We formalize this requirement as follows:

REQ 16. The data must be cleaned removing errors and noise must be corrected by applying a low-pass filter.

ADS-B limitations The most common ADS-B data links operate on a frequency of 1090 MHz. This means that the receiver must be in the line of sight in order to receive the message. For airplanes this is often not an issue since receivers in the low-cost range (50 - 250$) could potentially receive messages within the optimal and maximum range of 450 km (which is bounded by the curvature of the earth). Also, airplanes that approaching airports, not even considering on the airport, could be undetected because of a simple line of sight limitations. Recall that the OpenSky Network is crowd-sourced, so there is no guarantee the coverage is 100, in fact, it is most probably not.

Geo spatial analysis for determining actual A-CDM milestones and landing runway In order to determine actual A-CDM milestones and the landing runway geo-fencing must be applied against spatial bounds that describe features on the airport. The following requirements is fundamental:

REQ 17. The data processing architecture must enable efficient spatial, time and attribute queries. The system needs to be able to handle data efficiently. This requirement seems obvious but considering the sheer size of the datasets that we are working within this project is it an important one.

From the requirement to predict the runway a flight lands on the following can be derived:

REQ 18. The system must detect on which runway an airplane has landed

Additionally the requirements about the actual A-CDM milestones and landing runway are stated as:

REQ 19. The system must generate the A-CDM milestone ALDT - Actual landing time.

REQ 20. The system should generate the A-CDM milestone AIBT - Actual in-block time.

REQ 21. The system should generate the A-CDM milestone AXIT - Actual taxi time.

4.4.3. Data Augmentation

The input set for the prediction models contain features that are not part of the ADS-B messages. Therefore the data must be augmented. Some augmentation steps are explicitly mentioned in the research paper A.5.2. However, from the requirements on predicting the ELDT and the expected runway that the airplane will land on several additional requirements can be derived. In this section, we will bundle all requirements together that are about augmenting the data. Also, the last part of the data augmentation process, data scaling, will be discussed.

Augmenting the location of an airplane The first requirement is about the coordinate system used. ADS-B data uses the WGS-84 coordinate system. However, for our usages, we prefer to transform the coordinates into a local projection, the RijksDriehoeksCoordinate. This is done because in the projected coordinate system there is a linear relationship between the position, distance and angles.

REQ 22. WGS84 coordinates must be projected to RijksDriehoeksCoordinaten

Next, the data must be augmented such that the distance is computed from the reference location and AMS, to the actual location of the plane.

REQ 23. The data must be augmented with the distance to AMS from every point.(A.5)

REQ 24. The data must be augmented with the distance to the Reference location from every point. (A.4)

[4]yet..., ADS-B is gradually getting enforced legislative over the world.

Finally, the data must be augmented with the position of the airplane in relation to AMS. This is formalized as:

REQ 25. The data must be augmented with the angular position of every point in relation to AMS. (A.6)

Augmenting similar trajectories The ELDT prediction model uses a cluster assigned to every flight as an input feature. Trivially after actually clustering the trajectories, the initial data must be augmented with the result of the clustering. For completeness, this is also formalized as:

REQ 26. Every flight must be augmented with the assigned trajectory cluster.

Data augmentation: Normalization vs. Standardization The final stage of data processing and augmentation is data scaling. it is important to scale the data before the trajectories are clustered and the prediction models are learned. Otherwise arbitrary features might have a larger impact on the result as wanted. This is especially important when it comes to variables concerned to spatial data.[28] Two scaling methods discussed here are normalization and standardization. Normalization rescales features into a range of 0 and 1. Standardization rescales features to have a mean of 0 and a standard deviation of 1. After scaling, all the features get equal weight so that redundant or noisy objects can be eliminated while valid and reliable data will be obtained.[43] Standardization is applied to standardize residuals while normalizing is applied to normalize vectors in common. In our case, normalizing is more favourable than standardization since we are dealing with spatial vector data. The above is formalized as follows:

REQ 27. Data features that are used in the models must be normalized.

4.4.4. Storage

To satisfy most of the requirements, the data must be stored somewhere. This is primarily used for training the prediction models in our final product. Also, it is important to retain the data as-is, and after each processing step. This way the data can be processed and augmented independently and improved incrementally. Something we need to keep in mind is which architecture for storing data is useful for geospatial and time-related features. Also, all A-CDM milestones must be encapsulated. Furthermore, this architecture should also be able to be used for playback of the data. In this section, cloud storage is suggested as a viable option. Also, a requirement about the data retrieval infrastructure is further analyzed.

Cloud storage Cloud storage is a preferred option because it has good maintainability and is more practical than local storage due to a large amount of data. It also allows us to use a cloud computing database. This combines the functionality of storing and computing large data in the cloud. In the end, we decided to use Google's cloud platform for this project. It includes many computation functionalities such as 'BigQuery' and has easy to use storage buckets. Essentially, by using this platform we could combine all the functionalities we needed to handle our data. This made it the most practical option.

REQ 28. The flight data interface must encapsulate all A-CDM milestones.
This requirement is very specific and not very well-phrased. It should be interpreted as such that in later stage information about flights, i.e. A-CDM milestones, can be directly integrated. This requirement comes from the current situation in aviation where parties often face legacy issues. A simple thing as adding a data feature or getting information that does not come out of the box often can't be done or is extremely hard. In our case, adding A-CDM milestones is fundamentally the same as augmenting the data with features. Therefore we will consider this as trivial.

5 Methodology

In the previous chapter, we defined and analysed the challenges and problems we had to overcome, to be able to create a final product that the client will be satisfied with. After the task had been clearly defined, we need to actually find solutions for the problems imposed by the requirements and the implementation challenges.

In this chapter, our development process is explained. It describes our process in satisfying the requirements for each component of the project. We will discuss the components of the project in the same manner as the previous chapter, respectively: *Application*, *Machine learning*, and *Data processing*.

For the *Machine learning* and *Data processing* part we have used an iterative process combing Jupyter Notebooks as a scratch pad and a source controlled Python application.

5.1. Application

The application consists of a front end visual interface, and a back end API server to fetch data and run the prediction models on that data. Based on the requirements and design goals defined in the previous chapter in section 4.2, we will explain how we went about creating the application to satisfy those requirements. In this section, we describe the implementation details of the application. Also, we describe the process of creating architecture. It is separated into front end and back end just like the requirement definitions.

5.1.1. Front end

The front end consists of everything that the user gets to see or interact with directly. The app has mainly two views: a map view and a table view. Both views are initialized with the same state. The data about the status of airplanes is fetched from the back end every five seconds to the front end. Then, the data is mapped to satisfy the information needed to visualize lines, airplanes, and table accordingly. Both views satisfied the performance requirement 4. They contain the same navigation bar at the top. This visual layout is meant to be easy to use and understand for any user.

At the center of the navigation bar, we can see five elements as shown in Figure 5.1. The first element is a clock which shows the current time or the time of playback in the format of “year-month-day hour:minute”. The second element is the link to the map view, the third element is the link to the table view, and the fourth element is composed of a date picker and a playback button for a playback mode. They are explained in detail in the following subsections. The fifth element is the option of whether to hide the airplane that is already on the ground or to show all the airplanes that are received from the server. The change may have some delay because the data is fetched every 5 seconds.

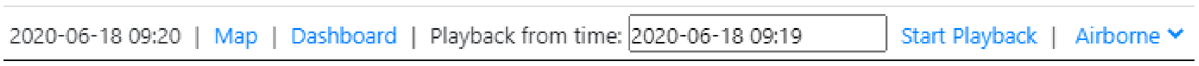

2020-06-18 09:20 | Map | Dashboard | Playback from time: 2020-06-18 09:19 Start Playback | Airborne

Figure 5.1: The navigation bar at the top of the screen

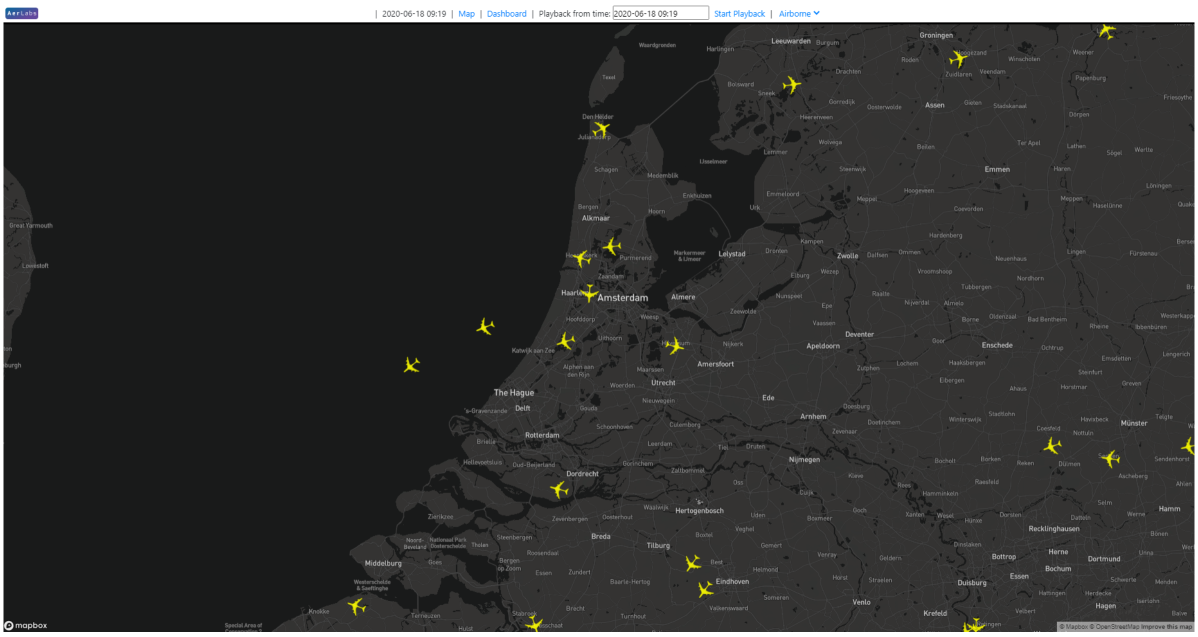

AerLabs
| 2020-06-18 09:19 | Map | Dashboard | Playback from time: 2020-06-18 09:19
Start Playback | Airborne
mapbox
© Mapbox © OpenStreetMap Improve this map

Figure 5.2: Default view of the app: Map

Map view

User interface Figure 5.2 shows the default view of the application: the map view. This map is one of the two parts required to satisfy requirement 1. The map view is built with a map from the react-mapbox-gl. Mapbox can apply five professionally designed maps: light, dark, streets, outdoors, and satellite. The map is designed to improve usability and user experience by design goal A.4.2. The map we have used is the adoption of a dark theme map. Important features such as runways are highlighted (see Figure 5.3). Furthermore, jitter and noise (unimportant features such as city names etc.) are removed as much as possible. The center point of the map is set to longitude and latitude of AMS and a zoom level of 8 is set as initial view state.

When the user hovers the mouse over an airplane, the user will see the call sign, the predicated runway, and the estimated landings time of the airplane as displayed in Figure 5.4. If the airplane is not going to land on AMS, then the user will see the phrase “Does not land on Schiphol” instead of the predicted value from our model.

The other part needed to satisfy requirement 1 is the actual flight paths. To show them, three deck.gl layers are added to the map. The first layer is the line layer. The line layer is used to visualize flight paths. It stored the last 30 data points for each receiving airplanes. The path has a white color because the map has a black color. The second and third layers are the icon layer. One icon is a yellow airplane, another icon is the same airplane but with a circle around it and has a blue color. The first icon layer with a yellow airplane denoted an unselected airplane. The second icon layer with the blue airplane denoted a selected airplane. To achieve this, the third layer is only visible when the airplane has been selected. The layout of these three layers is shown in Figure 5.5.

Performance In requirement 4 the needed performance for the front end is specified. We have chosen deck.gl to visualize airplanes and flight paths because deck.gl has a layered approach to visualization with larger datasets. And it renders datasets with unparalleled accuracy and performance. This is exactly what we needed to visualize a large number of flights. Thanks to the high-performance deck.gl layers, the map can easily visualize more than 1000 concurrent flights, under the condition the flight path only stored last 30 data points. Not all the points are stored because it will slow down the application when time passes.

Because the data is fetched every 5 seconds, the known position of the airplane will have a time interval of 5 seconds. If we refresh the position of airplanes after 5 seconds, The movement of the airplane will

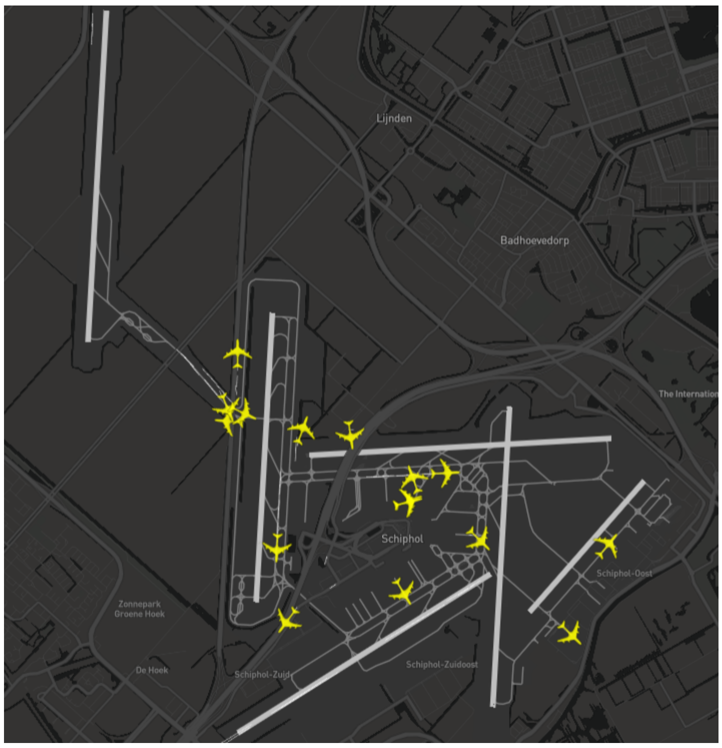
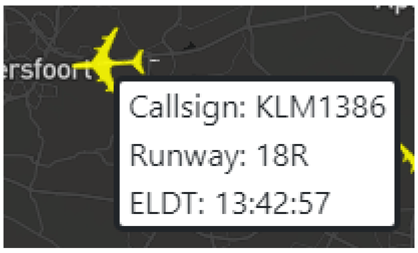


Figure 5.3: All flights at AMS at a certain moment with highlighted runways



Callsign: KLM1386
Runway: 18R
ELDT: 13:42:57

Figure 5.4: Pop-up on hovering over an airplane

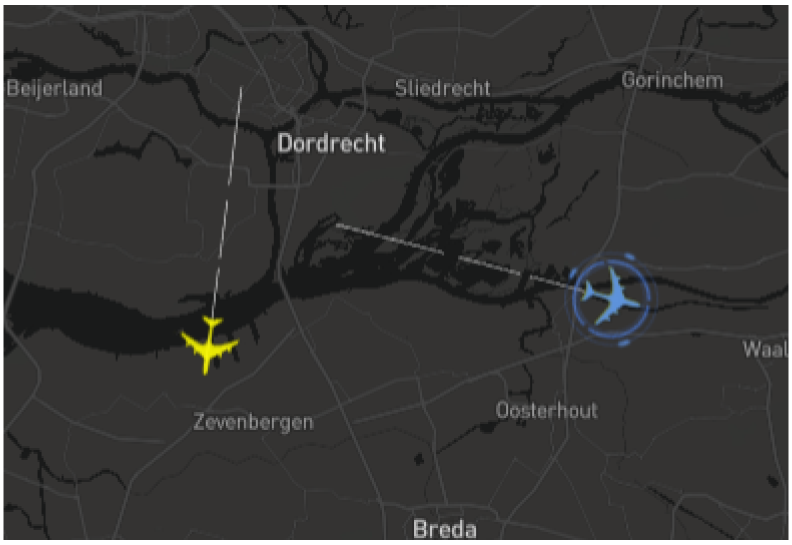
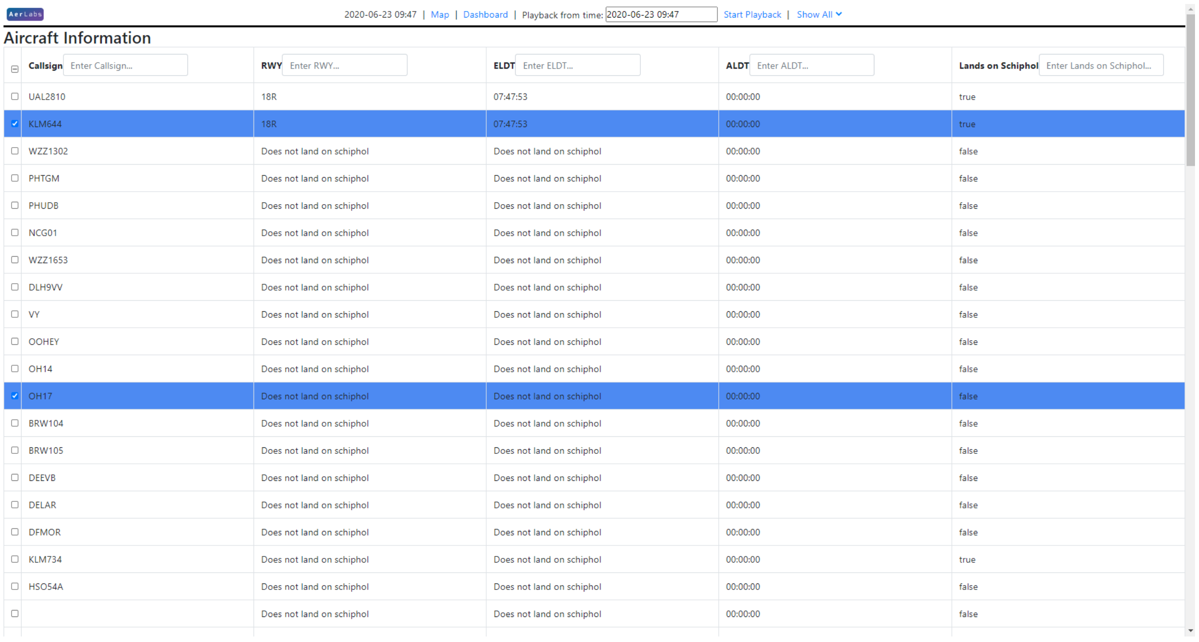


Figure 5.5: The visuals of paths, airplanes, and selected airplanes

be turn out to be abrupt. This is against our design goal A.4.2. Therefore, we estimated the position of airplanes in these 5 seconds. At first, the distance the airplane flights in 5 seconds is calculated with its speed. At second, the destination point of the airplane it should be after 5 seconds is calculated with the function *destinationPoint*. This function takes the longitude, latitude, speed, and bearing of the airplane as its input parameters. The calculated longitude and altitude of the destination point are normalized using the formula $(x+540)\%360-180$ so that the returned value is between -180 and 180 in the map. Then we use the function *geoInterpolate* from d3 library. This method creates a function that accepts input between 0 and 1 and interpolates the original position and target position for us. The current frame divided by the frame rate becomes the input for the *geoInterpolate*.

Table view

User interface In accordance with the requirement 2, the application has a table view we call 'Dashboard'. Dashboard view shows a table of the flights within the area covered by the map. Figure 5.6 is an image of the table view.


AerLabs
2020-06-23 09:47 | Map | Dashboard | Playback from time: 2020-06-23 09:47 Start Playback | Show All

Aircraft Information

	Callsign	RWY	ELDT	ALDT	Lands on Schiphol
☐	UAL2810	18R	07:47:53	00:00:00	true
✓	KLM644	18R	07:47:53	00:00:00	true
☐	WZZ1302	Does not land on schiphol	Does not land on schiphol	00:00:00	false
☐	PHTGM	Does not land on schiphol	Does not land on schiphol	00:00:00	false
☐	PHUDB	Does not land on schiphol	Does not land on schiphol	00:00:00	false
☐	NCG01	Does not land on schiphol	Does not land on schiphol	00:00:00	false
☐	WZZ1653	Does not land on schiphol	Does not land on schiphol	00:00:00	false
☐	DLH9VV	Does not land on schiphol	Does not land on schiphol	00:00:00	false
☐	VY	Does not land on schiphol	Does not land on schiphol	00:00:00	false
☐	OOHEY	Does not land on schiphol	Does not land on schiphol	00:00:00	false
☐	OH14	Does not land on schiphol	Does not land on schiphol	00:00:00	false
✓	OH17	Does not land on schiphol	Does not land on schiphol	00:00:00	false
☐	BRW104	Does not land on schiphol	Does not land on schiphol	00:00:00	false
☐	BRW105	Does not land on schiphol	Does not land on schiphol	00:00:00	false
☐	DEEVB	Does not land on schiphol	Does not land on schiphol	00:00:00	false
☐	DELAR	Does not land on schiphol	Does not land on schiphol	00:00:00	false
☐	DFMOR	Does not land on schiphol	Does not land on schiphol	00:00:00	false
☐	KLM734	Does not land on schiphol	Does not land on schiphol	00:00:00	true
☐	HSO54A	Does not land on schiphol	Does not land on schiphol	00:00:00	false
☐		Does not land on schiphol	Does not land on schiphol	00:00:00	false

Figure 5.6: Table view with two selected rows

The first column of the table shows checkboxes which indicate which row/rows has/have been selected. The second column is the call sign of the flight if given. Call sign (or used to be call signal) is a unique designation for a transmitter station. Aviation call signs are communication call signs assigned as a unique identifier for an airplane. The third column is the predicated runway of the airplane. The fourth column is the estimated landings time. The fifth column is the airplane's actual landings time. The sixth column is a Boolean value which indicates whether the airplane is going to land on AMS or not. All the columns have a filter. The user can search for the desired airplane(s) at the header of each column.

Performance The front end table also needs to adhere to the performance requirement 4. We imported BootstrapTable from 'react-bootstrap-table-next' because of its easy to use built-in functionalities for a table. Airplanes in the table can be identified by the unique 24-bit identifier ICAO 24. The value of the icao24 is not displayed, but it is used as a key field in the table. To avoid a long list, the data from the server is first filtered in the App class before it is sent to the child class Table. The data is filtered such that only the flights in the bounding box of the map view are displayed in the table.

Each time the view state of the map is changed, it will be propagated to the parent class App. The App updates the state and passed the new state to the child classes Map and Table. Table class get all the selected airplanes from the class App. The checkbox of the corresponding row(s) of the selected airplane(s) will be checked and the row(s) will be highlighted with the same blue color as the icon of a selected airplane. When a row is clicked, the corresponding airplane will be added to the list ‘selected’ if it is not in the list before. Else, it will be removed from the list ‘selected’. All these changes will be propagated the parent class to ensure the design goal A.4.2 data integrity within the application.

Playback

User interface The playback function plays back the flight data from any chosen date and time. That way operators can see a replay of the flights on the map and table as required by the requirement 3. In the navigation bar, there is a simplistic date picker which a user can click on to open a small calendar where a date and time can be chosen in an intuitive way (see Figure 5.7). The date picker can also work as a text field if a user types date and time in the given format. The default value for this text field will be the date and time of starting the application, making the format clear. The way the playback mode looks and is initiated was made keeping the design goal of the user experience in mind.

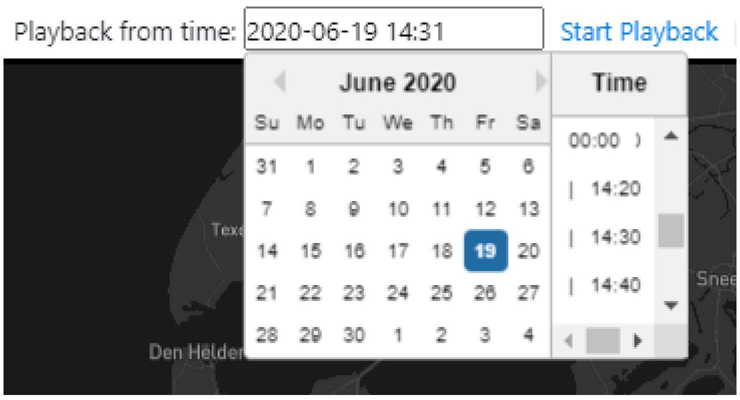


Figure 5.7: A date picker for playback

Performance To make sure the playback mode does not ruin the performance of the front end stated in 4, we have implemented it in a simple way by storing the time difference from the current time. In the React code, a boolean is kept in the states of the App and NavBar to indicate whether the application is currently playing back or not. In the same way, an integer is kept containing the time difference in seconds between the current unix time and the unix time of the playback. Upon starting playback, first, the given date and time are verified to be valid, if the timestamp is not valid then the user will be alerted of this and playback will not start. If the timestamp is valid then the unix timestamp is calculated and the aforementioned boolean and integer are set to ’true’ and the calculated difference respectively.

After that, two propagated function calls are made to the Map and App classes. The function in ’App’ sets the boolean and integer there as well. It also clears the ’lines’, ’airplanes’ and ’table_data’. In the function that sends the HTTP requests to fetch data, it checks whether the application is in playback mode using the boolean, and if it is, then it requests data from the current unix time minus the time difference integer. If it is not in playback mode, it fetches current data. What makes this feature possible is the fact that our API has the option of giving a timestamp to query the OpenSky Network database with.

5.1.2. Back end

To achieve the final product we have, an architecture was required to connect the front end visuals to the flight data. Keeping maintainability and possible extension in mind, we made a simple API server which we can use to query the OpenSky Network database at a certain point in time. This way, both the real-time view and playback view can be done by calling to the same API, just with a different unix timestamp. This also makes it possible to filter the data at the API server, before it gets sent to the front end. That way the front end will take up less memory, and it is not performing a task better suited for a back end component. There is also a paragraph explaining how our API server makes use of the Schiphol API to adhere to the design goal of performance at the end.

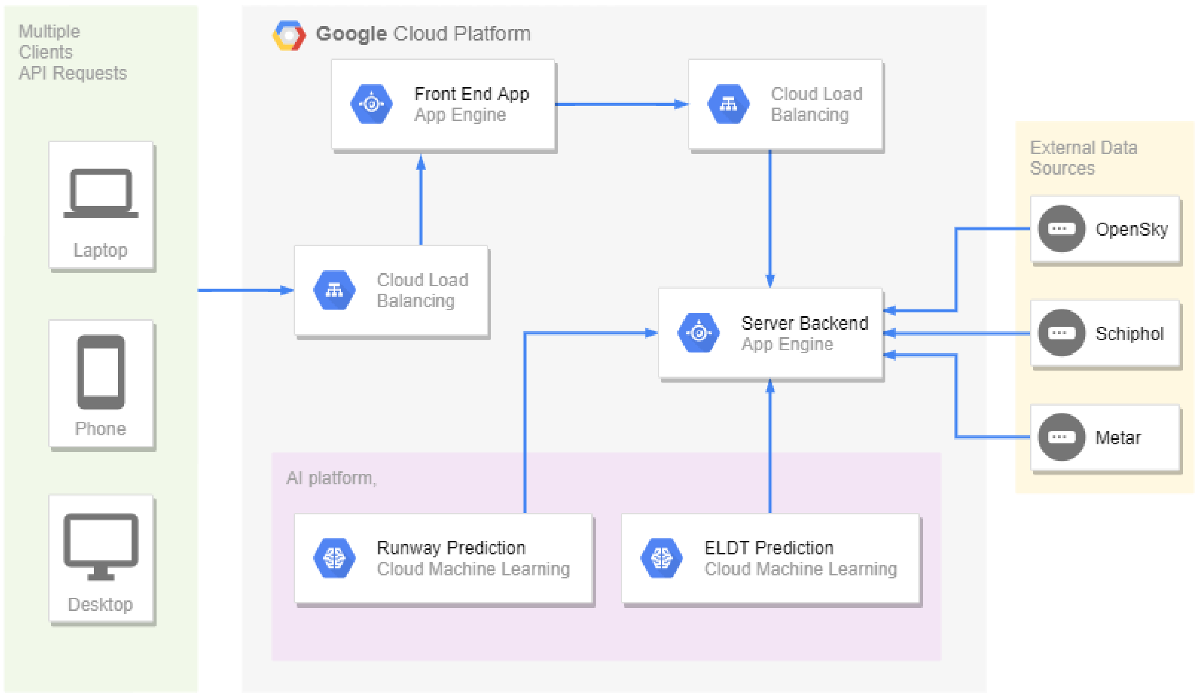


Figure 5.8: Overview of the back end architecture

Figure 5.8 is an overview of the back end architecture. The server continuously collects data from all 3 external sources. It does so at different rates per source: every 15 minutes from the AMS API, every 10 minutes from the Metar API, and every 5 seconds from the OpenSky Network API. We do this as the AMS data and the weather data do not need to be updated as frequently as the flight data from OpenSky Network (and also to not exceed the number of API calls we can make with our free accounts to any of these services).

The server then filters the flights on whether they land at AMS or not by matching the OpenSky Network flight data to the AMS arriving-flight data. It adds the weather data and then transforms the data of the relevant flights to make the format suitable as input to the prediction model we've trained and hosted in Google's AI Platform in the cloud. It then first makes a request to the runway prediction model to get the runways and then adds this data for the request to the ELDT prediction model. Finally, this data is added to the flight data and the flight data is served through an API. The same API call at the same moment will result in the same result, which satisfied the data requirement 6 and data integrity design goal in A.4.2.

We decided to use the Flask framework to create the API of the server back end as it is a well known and capable framework for building APIs in python. We did not see the need to consider other options. The RESTful API we have made works well for current data and is a bit slower when in playback mode. However, this is due to the OpenSky Network database being slower when queried with a certain timestamp, and it has nothing to do with our API. When querying current flights, the API first runs the prediction model on the data it gets, and then sends the data along with the predictions to the front end.

Because these services are hosted in the Google Cloud Platform we will let Google take care of scaling our application up and down when needed. This way, the performance requirements in 7 for the back end are satisfied.

Schiphol API To fully satisfy requirement 4 and 7 we need to remove all the flights that are not relevant for the user. In order to only show the flights that are going to land at AMS, we first need to filter all of the flights provided by OpenSky Network on whether or not they actually land on AMS. This is done by contacting the Schiphol API with a request for all arriving flights and matching these flights with the data from OpenSky Network. The Schiphol API, unfortunately, implements an extremely impractical version of pagination so you have to make sure to get all pages instead of just the first one. Also, one has to keep in mind that the callsign as supplied by OpenSky Network is constructed from PrefixICAO and flightNumber as supplied by the Schiphol API. This is something that happens at the back end, but is also good for the front end.

5.2. Machine Learning

In this section, we describe the implementation of the prediction models, the architecture, and the use of a cloud provider. The two prediction models we have created are largely similar. As in they are both sequential neural networks. However, the number of hidden layers and the number of nodes at every layer is different. Using Keras, a high-level API for Tensorflow we implemented the networks.

In a sequential model, layers are stacked sequentially. Each layer has a unique input and output. Those inputs and outputs then have a unique input shape and output shape. In the below image 5.9 this is visually represented. The neural network shown has 3 input nodes, 1 hidden layer with 4 nodes, and two output nodes.

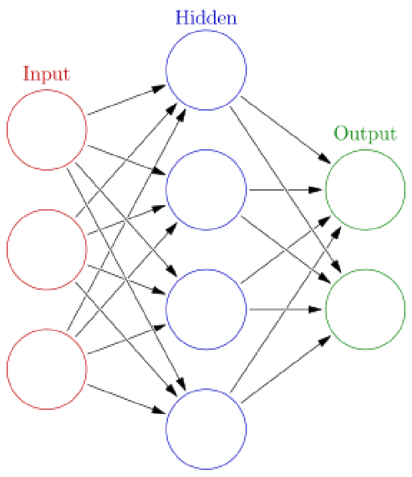


Figure 5.9: A sequential neural network

The sheer volume of the data that needed processing and subsequently training was too large to fit properly on a single device such as a laptop. The choice has was therefore made to run the entire operation on a cloud platform. Combining both the clients and our preference we choose Google Cloud Platform (GCP). In the specified topic below we detail what we used for training the model.

5.2.1. Classification model

Empirically we have determined the size of the model. We have settled for a 3 layer model. Input, hidden and output layer. The input layer has 16 nodes, corresponding to the features used to train the model. The hidden layer has 32 nodes, and the output layer has 12 nodes corresponding to the number of runways. Various sizes of the hidden layer have been tested, trying to eliminate under- and overfitting in the process.

Features The following list of features has been used to train the runway classification model.

- altitude_standardized
- dewpoint
- groundspeed_standardized
- hour_of_day
- loc_x

- loc_y
- loc_z
- pressure
- temp
- track_x
- track_y
- vertical_rate_standardized
- visibility
- winddir_x
- winddir_y
- windspeed

Label The label we use for this classification model is the runway. During the *feature engineering* phase we have determined the runway for every flight landing at AMS. The model only allows a *int64* to be used as a label for a classification problem. Therefore, we need to map the runways to a number. AMS has 6 runways which can be used in both directions, hence the 12 runways listed below.

1. 04
2. 22
3. 06
4. 24
5. 09
6. 27
7. 18C
8. 36C
9. 18L
10. 36R
11. 18R
12. 36L

Activation function The hidden layer has a generic ReLu activation function. However, the last layer has a SoftMax activation function. Together with the shape of the output later, which is equal to the number of classes this model can determine, makes that this is a classification model. When predicting a sample each of the 12 output nodes will hold a value on how much the model thinks it is landing on the runway corresponding with the output node. These numbers cumulative are 100%.

5.2.2. Regression model

Empirically we have determined the size of the model. We have settled for a 4 layer model. Input, hidden and output layer. The input layer has 16 nodes, corresponding to the features used to train the model. The hidden layer has 32 nodes, and the output layer has 12 nodes corresponding to the number of runways. Various sizes of the hidden layer have been tested, trying to eliminate under- and overfitting in the process.

Features The following list of features has been used to train the ELDT regression model.

- altitude_standardized
- dewpoint
- distance_to_runway
- groundspeed_standardized
- hour_of_day
- loc_x
- loc_y
- loc_z
- runway_04
- runway_06
- runway_09
- runway_18C

- runway_18L
- runway_18R
- runway_22
- runway_24
- runway_27
- runway_36C
- runway_36L
- runway_36R
- pressure
- temp
- track_x
- track_y
- vertical_rate_standardized
- visibility
- winddir_x
- winddir_y
- windspeed

Label The label in this prediction model is the time to land in seconds.

Activation function The activation function used in all layers is ReLu. It is the most commonly used activation function in neural networks.

5.2.3. Cross-validation

We used cross-validation as a technique for testing the performance and validity of the prediction models. Cross-validation is illustrated in Figure 5.10. It opposes the risk of over-fitting a model to a certain training and validation split by dividing the entire data-set in a predefined number of parts and using each part as a validation set for a training run.

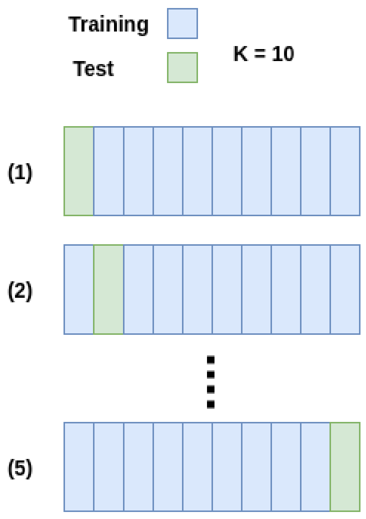


Figure 5.10: Cross validation

5.2.4. Google Cloud Platform

The data processing, training of the prediction models and the hosting of the two prediction models run in the Google cloud platform. This ensures that the performance requirement for serving predictions 10 and 11 are met.

5.3. Data

In this section, we will describe the flow from extracting the data from the Impala API up to the final augmentation steps. First, the process of acquiring the data is given. Then the method for trajectory extraction is discussed, which turns out to be a determining factor in our data processing workflow. Next, the steps taken to filter the data, significantly reducing the overhead in the amount of data to process, are explained. After the data is filtered, correcting the data for errors and noise is treated. Finally, it is shown in detail how the data is transformed from noisy, erroneous data into a quality data set.

5.3.1. Data sources

In this project, several data sources were used. An overview is given of the OpenSky Network which is the source of flight data. Next, it is discussed where we acquire the meteorological data around an airport (METAR). Lastly, we discuss the Schiphol API, which was used as an additional data source to retrieve current flight information and verify which flights are going to land at Schiphol Airport (AMS).

OpenSky Network The main source of data for our model is ADS-B. We collect this from an initiative called OpenSky Network. Acquiring data from the OpenSky Network satisfies requirement 12. The OpenSky Network started in 2012 as a research project between armasuisse (Switzerland), University of Kaiserslautern (Germany), and University of Oxford (UK). The project aims at improving the security, reliability and efficiency of the air space usage by providing open access to real-world air traffic control data to the public. The OpenSky Network consists of a multitude of sensors connected to the Internet by volunteers, industrial supporters, and academic/governmental organizations. All collected raw data is archived in a large historical database. The database is primarily used by researchers from different areas to analyze and improve air traffic control technologies and processes.

METAR To be able to use weather data for our models, we need to have weather data first. The most conventional weather data for the aviation industry are METAR reports. These reports are in raw text strings and need to be parsed before we can use them. They are publicly available and for this project, we will use reports from the weather station at AMS (ICAO code = EHAM). A METAR report contains things such as observation time and date, temperature, wind speed and direction, and horizontal visibility. An example of a METAR report looks like this: "METAR EHAM 171925Z 02012KT 350V060 9999 VCSH FEW040CB 19/14 Q1010 TEMPO FM2000 3000 TSRA=". Information on how to decode such a report can be found on the METAR Wikipedia page [13]. Naturally not every single data field in a METAR report is used for our model, and not all fields are always included in a report so this data that we gathered is not consistent. All of that means we still needed to process the weather data to only contain the useful fields, such that we could use it to train our models. However, this results in the meteorological feature we require for the prediction models, satisfying requirement 13. These METAR reports are consistent and universal per measuring station, so using them conforms to the design goal of data integrity.

5.3.2. Data Extraction

In this section, we will describe how we extracted the data from the Impala API and processed the data accordingly. First, the process of acquiring the data is given. Then the method for trajectory extraction is discussed, which turns out to be a determining factor in our data processing workflow. Next, the steps taken to filter the data, significantly reducing the overhead in the amount of data to process, are explained.

Acquiring the data The data from OpenSky Network is retrieved by using the Impala Shell. A virtual machine was used on which a long-running Python script was present. The script logged in over SSH to the OpenSky Network server where it fetched a one hour span of data. It will then store that data in a parquet format on Google Cloud Storage. In this fashion, it looped over all the hours in 2019. Due to time and memory restrictions, it was not possible to extract more than one hour of full data at a time.

Flight encapsulation: Identification and extraction of trajectories When every ADS-B message is interpreted as a row in a matrix. The resulting matrix corresponds with a time series containing information about all airplanes during that given time. This doesn't enable us to do data cleaning and processing. A good way of representing the data would be if every unique flight has it's own time series, only showing its own ADS-B messages. The well-known framework Pandas is ideal for working with time series. Chained Pandas' methods enabled us to intuitively interact with the data. Applying vectorization on the Dataframes with NumPy realizes the efficiency needed to perform the (big) data processing.

The Traffic library [36] can be used to convert the ADS-B messages in Pandas data frames. The good thing about this library is that it solves the problem of trajectory extraction by combining the callsign, ICAO 24 address and the timestamp to determine single legs of flights. Thus using the traffic library gives us the desired result of a unique flight data frame only containing its own related ADS-B time series, satisfying requirement 14.

An additional convenient benefit of using the Traffic library is that it contains (outdated) geospatial information about some airports and their infrastructure. Since AMS is included, it saves us the trouble of geo-encoding all airport features by hand.

Filtering arriving flights at AMS Our goal is to only retain flights that are arriving at AMS. Unfortunately, data retrieved from the OpenSky Network is not filtered as such. However, the impala API enables us to perform a geospatial query by defining the bounds on longitude and latitude. We choose the bounds to be the terminal manoeuvring area (TMA) of EHAM described in the Aeronautical Information Publication, published by LVNL. We pick the TMA because this is controlled airspace, meaning traffic is directed to and from AMS. Thus flights arriving at AMS must enter this airspace (see Figure 5.11).

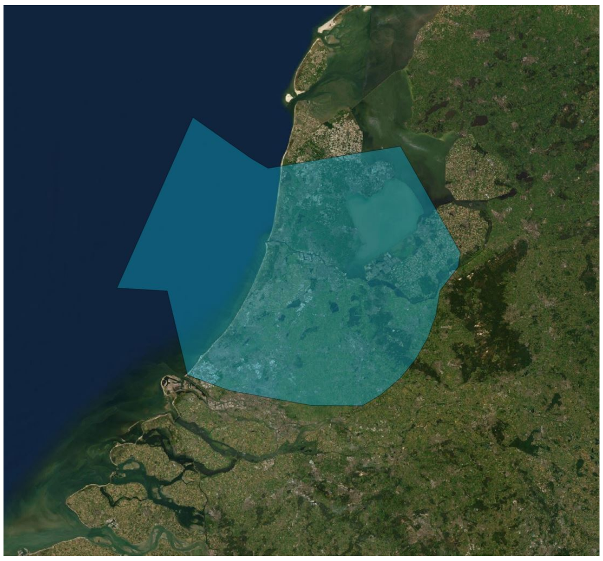


Figure 5.11: The Terminal Manoeuvring Area of Amsterdam Airport Schiphol

After filtering the data still contains both arriving and departing flights from AMS, and possibly noise (fly over's). In order to satisfy requirement 15 we still need to determine which flights are actually destined to land at AMS. Our method for doing so is as follows:

- Take the last point of each trajectory and compute the distance to AMS.
- Discard flights with a distance larger then 8000 meters [1]

The method described above works because we are only considering flights are inside the TMA. If flights are not landing at AMS, it means that they have another destination. This, in turn, means that at some point the airplane must exit the TMA or land at another airport inside the TMA. In both cases, the distance from AMS would be at least a magnitude larger than 8000 meters.

5.3.3. Data processing

The data processing is done using cloud technology. From various sources, the data is stored on Google Cloud Storage. Using Google Cloud Dataflow, a managed Apache Beam service, the data is processed iteratively to

[1]This distance is based on a radius that takes into account all the runways and possible line of sight limitations which may result in dead spots.

clean, combine and add features (see Figure 5.12).

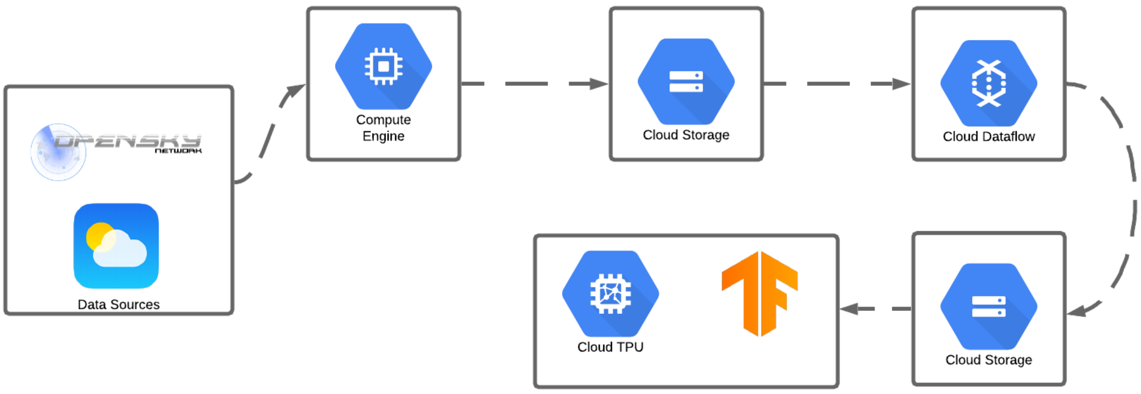


Figure 5.12: The data pipeline

OpenSky Network Data retrieved from the OpenSky Network often contains false callsigns due to decoding errors. These errors will have to be removed to satisfy requirement 16. Unfortunately, this can not easily be corrected. We can, however, discard flights with a callsign that contain not enough messages. It turned out that a threshold of 10 worked well. Then the trajectories are cleaned by applying the following steps:

1. Remove flights resulting from decoding errors by removing flights that only have a few entries (i.e. messages) in the data frame. [2]
2. Remove duplicate messages by resampling to 1 message per second for every flight.
3. Applying several filters to remove noise and other (decoding) errors.

Removing noise by applying filters Spikes and drops are not corrected by the above method. In order to do this, we need to apply several filters, depending on the data attribute. In the previously done research a low-pass filter (A.2) was suggested. However, the traffic library contains a filtering method that is optimized for most of the attributes. It would make no sense to reinvent the wheel [3], thus we used the median-filter from the traffic library wherever applicable. The altitude is correctly smoothed as seen in Figure 5.13.

However, not all data can be corrected using filters. The Figure 5.14 illustrates an example of false negative value. In the first figure, an arriving airplane is observed.[4] However, after cross-validating it against the altitude of a airplane we discovered that false negatives also occur. As shown in Figure 5.14, the airplane never transmits the on-ground flag, while it most certainly must be.

Upon inspecting Figure 5.14, one must also notice the noise after the airplane supposedly landed. These sudden spikes are most probably because of a lack in line of sight and interference by other signals. Oddly once the signal is restored, the airplane transmits an altitude of over 7 KM. This is obviously nonsense, the airplane is actually on the ground. Dealing with this problem is difficult, especially with the dependence and restriction of only using crowd-sourced data from the OpenSky Network. In order to satisfy requirement 16, we discard the ground segments of a flight and leave processing operational flight data on the ground for future work.

[2]Data retrieved from the OpenSky Network often contains false callsigns. This is due to decoding errors. Because of how trajectories are extracted, these errors result in flights containing only multiple entries. Although possibly this could, in theory, be corrected, the best way to handle this situation is by discarding the false flights. Likely the next message will be correctly decoded anyway.

[3]We actually did reinvent the wheel, since we implemented the filters before we started using the traffic library. However, it's (almost) always preferred to use a wheel instead of an edgy one.

[4]Something curious is happening. After reaching an altitude of 0 (landing), the altitude shows sudden spikes and drops. Most probably this is due to the line of sight limitations and interferences with other signals on the airport. Oddly the altitude stabilizes at 5000 meters, which is clearly false since the airplane is on the ground.

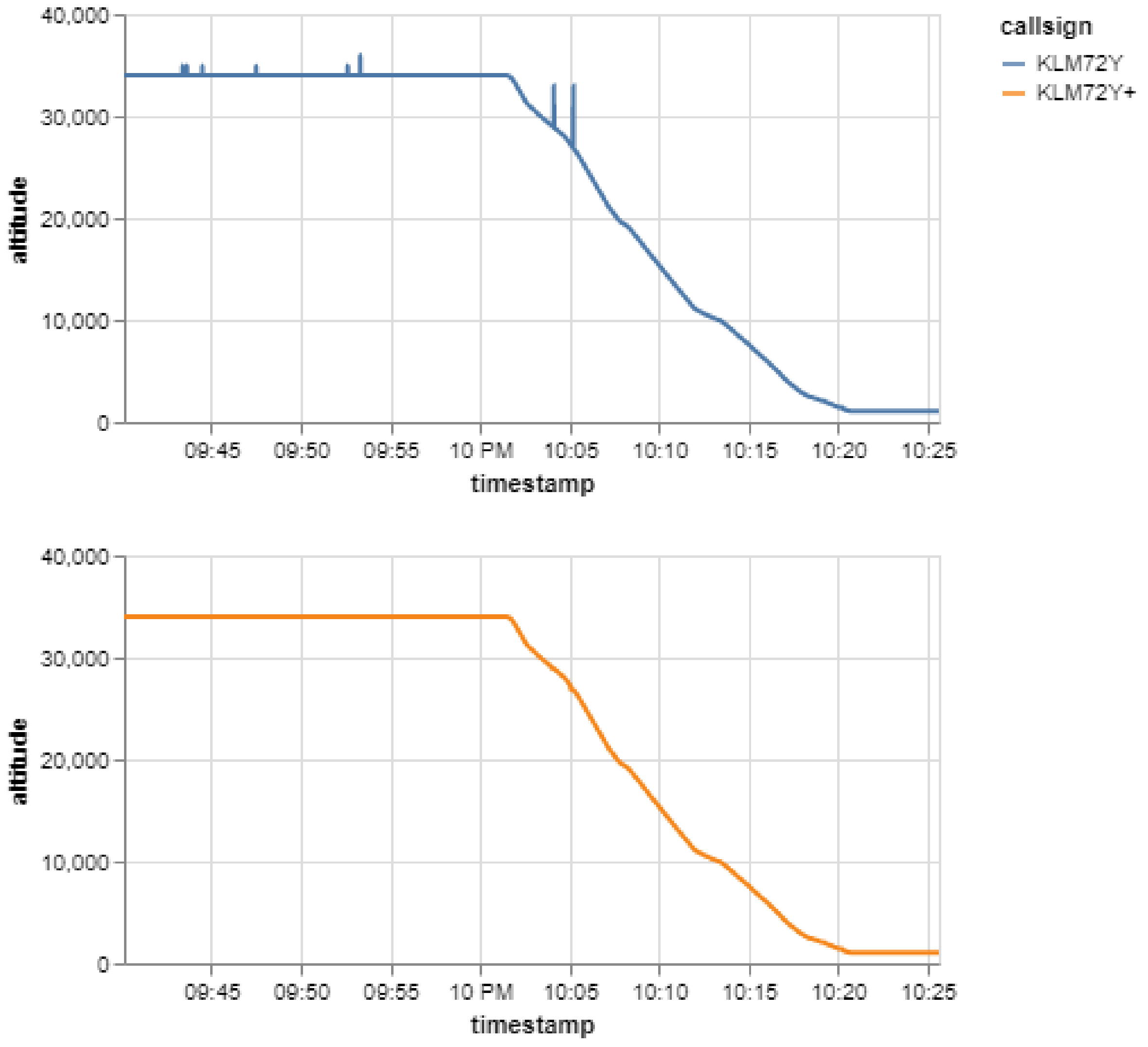


Figure 5.13: Erroneous altitude messages

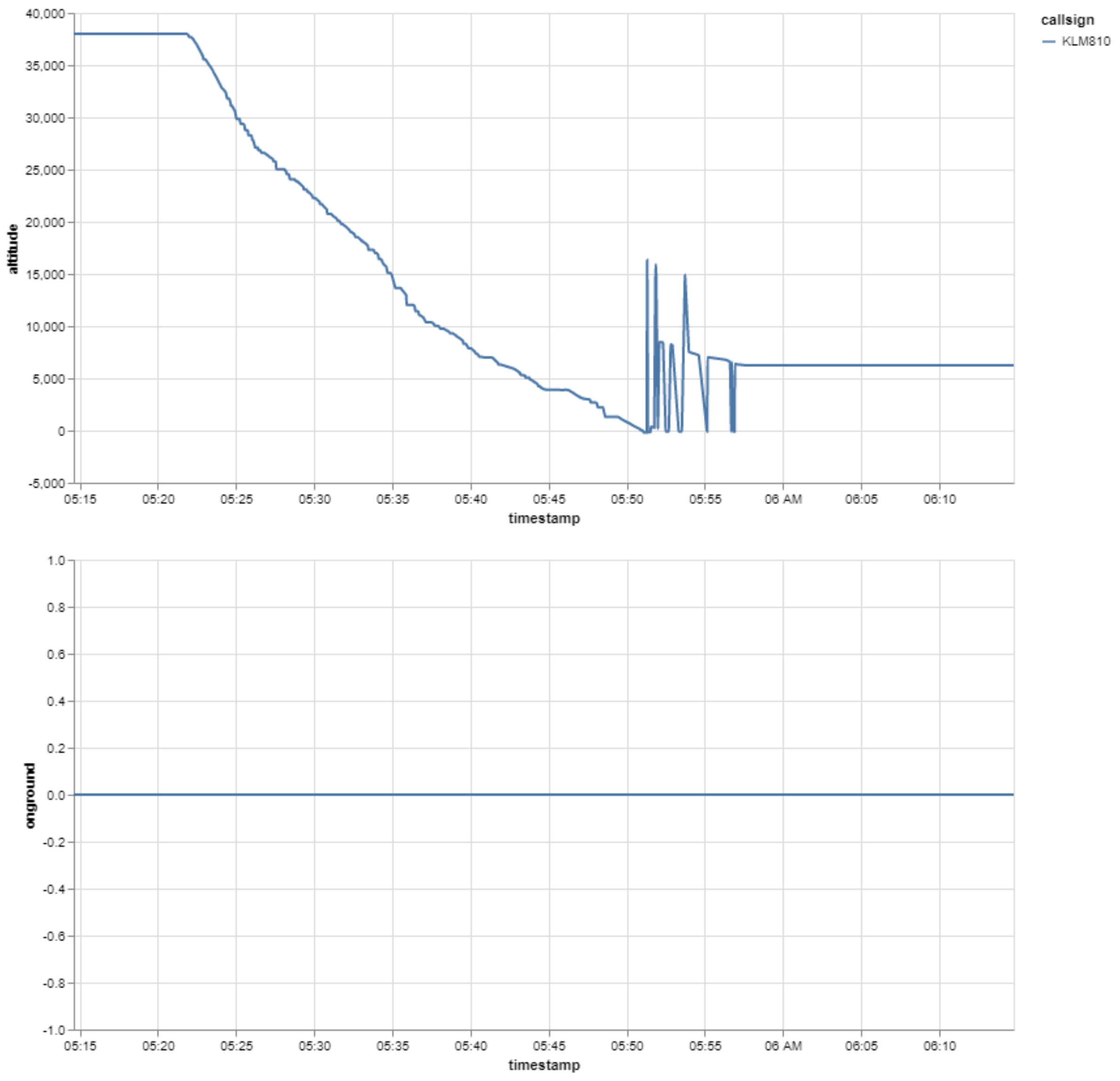


Figure 5.14: Faulty messages on ground

Reliable on-ground information is useful in validating whether an airplane has actually landed. However, as seen previously with the noise in data and the false negatives which we can not repair (it's probably an error at the transmitting side) any single of the above named features is not reliable enough to make strong inference's about the data. Combining them however would in most cases enable us to say with certainty if and when an airplane has landed. This is evidently the problem of determining the ALDT 19, one of the main requirements of our project.

5.3.4. Data Augmentation

Recall from the problem analysis that we required to augment the following:

- Actual landing time (19)
- Actual arriving runway (8)
- Project coordinates into RijksDriehoeksCoordinaten (22)
- Distance to AMS (23)
- Distance to reference point (24)
- Angular position in relation to AMS (25)
- Assignment to cluster containing similar flown trajectories

In this section we will first discuss how we computed both the actual landing time and arriving runway. Then the methodology for coordinate transformation into the local projected coordinate system RijksDriehoeksCoordinaten and the remaining requirements is explained. Evidently computing the distance between points and determining the angular position is closely related with the coordinate system that is used. Finally it is discussed how flights are actually clustered using DBSCAN.

Actual Landing Time and Arriving Runway The actual landing time is the actual date and time when the airplane has landed (touch down). This could be determined by looking at the on-ground, or altitude information in an ADS-B message. However as explained in the processing, there are multiple problems with such an approach. Although spikes and drops can be smoothed out while processing, we still can not account for false negatives or plain faulty transmissions. But by combining the information we do have, we can say with near certainty that airplanes have landed, satisfying requirement 19. Our approach is as follows: First, we investigate the barometric altitude. The barometric altitude stops transmitting when the landing gear is lowered unless the barometric altitude is bugged this results in a value of 0. While this is a good indication the airplane is actually landing, this is not the exact moment of touch down. The exact time of landing gear extensions varies and could be influenced by the type of the airplane, weather or pilot preferences. We use the coordinates of the runways to determine the actual landing time. In Figure 5.15, the airport can be seen with geo encoded features plotted with the Altair library.

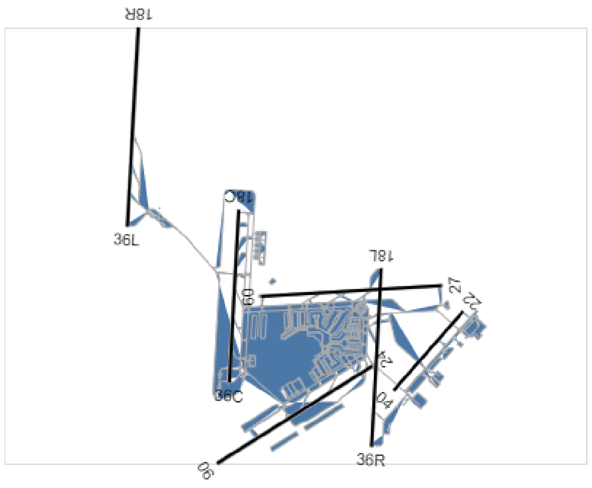


Figure 5.15: Geo encoded visualization of Amsterdam Airport Schiphol

To determine the exact landing time of an airplane, we take the first intersection of the trajectory with any runway boundary below an altitude of 1000 meter. We do this because a flight can cross a runway multiple times if we only look at the x-, and y-axis boundaries. This is illustrated in Figure 5.16, the intersection is the transition from the green to the red line.

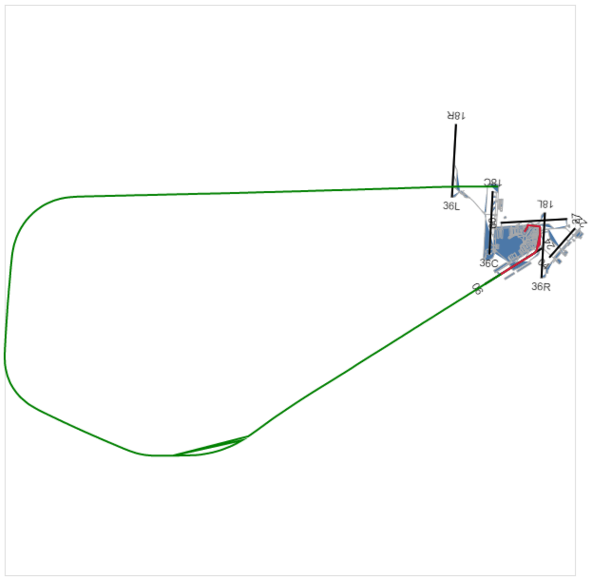


Figure 5.16: Intersection of trajectory with runway

At last, we use the geo encoded information about the runways to determine the runway the airplane used for landing. This is done by aligning the Instrument Landing System (ILS) with the orientation of the runway. The arriving runway is determined by requiring that both the alignment matches and that three consecutive messages are within the Geo-Fence of that runway.

Coordinate Transformation The WGS84 coordinates are transformed as follows:

```
from pyproj import Transformer
wgs84_to_rd = Transformer.from_crs(4326, 28992).transform

def wgs84_to_rd(x):
  lat, lon = x
  return wgs84_to_rd(lat, lon)
```

Since the projected coordinate system is based on airplane linear units, most importantly meters can directly be used. This easily enables us to augment the distances by computing the respective Euclidean distance. [5]

The angular position is calculated with the formula given in (A.6). At last, the cosine and sine values of the angular position and heading are chosen instead of the angle in degrees to avoid a discontinuity at 360°. Vectorization from the NumPy library is applied to optimize the augmentation for big data sets. [6] Given is the following code snippet of applying vectorization to augment the data with the cosine value of the heading.

```
df['cos_heading'] = np.apply_along_axis(lambda x: np.cos(x), 1,
 ↪  df[['heading']].values)
```

Trajectory Clustering

Clustering the trajectories consist of two parts. First, a principal component analysis must be done. Then the data can be clustered using the DBSCAN algorithm. We will discuss parts in this section.

Principal component analysis Before applying PCA the data first needs to be re-sampled and normalized. Re-sampling can be achieved by efficiently grouping and reshaping the original Pandas dataframe.

[5]In the WGS84 coordinate system distance is computed with the Haversine Distance

[6]This is explained in detail here, and here. Panda's native methods are to slow to apply on big data sets.

```
resampled =  df.groupby('flight_id').apply(lambda x:
 ↪  x.sample(sample_size).sort_values(by='timestamp'))
augmented_matrix = resampled[["x", "y", "altitude", "distance_to_schiphol",
 ↪  "distance_to_reference_point", "cos_angular", "sin_angular", "cos_heading",
 ↪  "sin_heading"]].values.reshape(-1, 400)
```

The principal component analysis described by (A.13) and the data normalization required 27 can be directly performed with the scikit-learn package. We choose the number of principal components such that 95% of the variance is retained.

```
from sklearn.preprocessing import MinMaxScaler
from sklearn.decomposition import PCA
x = MinMaxScaler().fit_transform(augmented_matrix)
pca = PCA(n_components=.95)
result = pca.fit_transform(x)
```

Trajectory clustering with DBSCAN The DBSCAN algorithm is part of the sci-kit learn package. The main difficulty with DBSCAN is tuning its parameters. Both rely on a solid distance metric. However, the distance in a n-dimensional space [7] is not easily interpreted. To get an intuitive feeling an exploratory data analysis is done where we plot random sample containing arriving flights at AMS during 1 day[8].

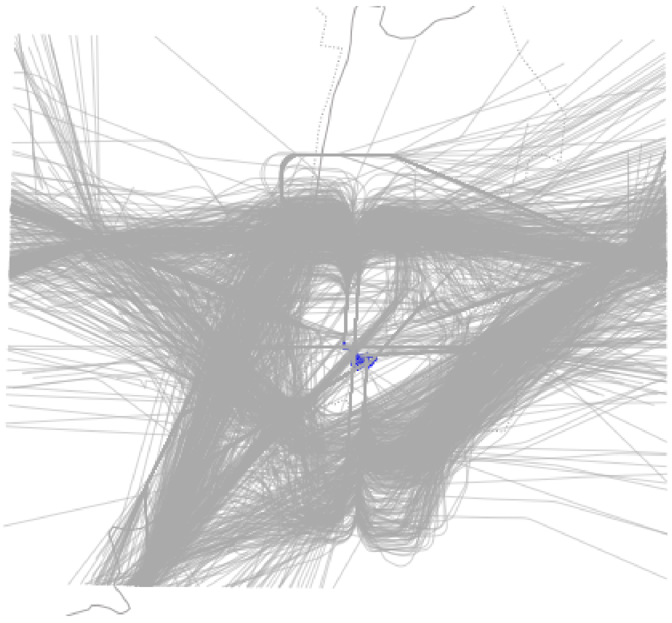


Figure 5.17: Arriving flights for one random day at AMS

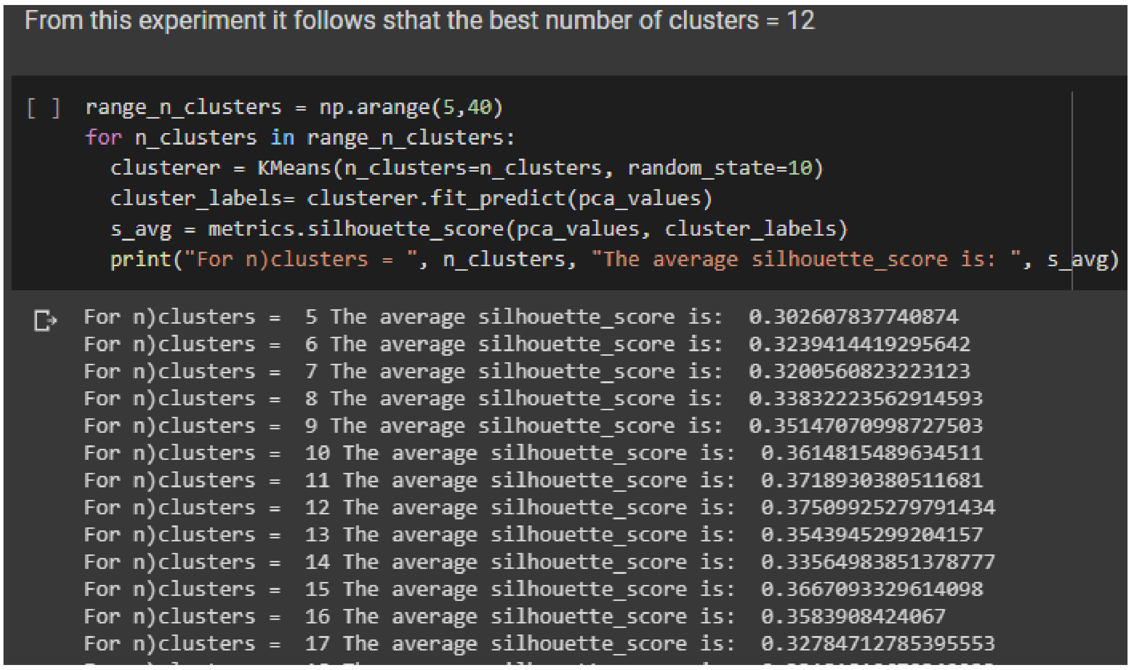
The above Figure 5.17 will serve as an intuition for the clustering parameters. Our goal is to find the optimum parameters for the DBSCAN algorithm resulting in the "best" clustering of the trajectories. To do this we use the knee detection method described in [35].

This method is put into practice by using k-means clustering with a-priori determined cluster sized and computing the average silhouette efficient of the resulting clusters. Based on the traffic plotted above around we set the range of clusters between 8 and 20. For each cluster, the silhouette score is calculated (see Figure 5.18). Based on this method the number of clusters generated should be around 12 for the input data set.[9] This value is interpreted as a guideline, i.e. we consider the number of clusters around 12 to be a "good" representation of the data.

[7] n is the number of principal components

[8] Note that the runway configuration at AMS is heavily dependent on the meteorological conditions at a given time, this could have an impact on the traffic pattern

[9] This is based on the highest silhouette average. Note that this is not a real optimum for the number of clusters since it could be that features other than distance between points play a role when determining cluster membership.


From this experiment it follows sthat the best number of clusters = 12

```
[ ] range_n_clusters = np.arange(5,40)
    for n_clusters in range_n_clusters:
      clusterer = KMeans(n_clusters=n_clusters, random_state=10)
      cluster_labels= clusterer.fit_predict(pca_values)
      s_avg = metrics.silhouette_score(pca_values, cluster_labels)
      print("For n)clusters = ", n_clusters, "The average silhouette_score is: ", s_avg)
```

```
For n)clusters =  5 The average silhouette_score is:  0.302607837740874
For n)clusters =  6 The average silhouette_score is:  0.3239414419295642
For n)clusters =  7 The average silhouette_score is:  0.3200560823223123
For n)clusters =  8 The average silhouette_score is:  0.33832223562914593
For n)clusters =  9 The average silhouette_score is:  0.35147070998727503
For n)clusters =  10 The average silhouette_score is:  0.3614815489634511
For n)clusters =  11 The average silhouette_score is:  0.3718930380511681
For n)clusters =  12 The average silhouette_score is:  0.37509925279791434
For n)clusters =  13 The average silhouette_score is:  0.3543945299204157
For n)clusters =  14 The average silhouette_score is:  0.33564983851378777
For n)clusters =  15 The average silhouette_score is:  0.3667093329614098
For n)clusters =  16 The average silhouette_score is:  0.3583908424067
For n)clusters =  17 The average silhouette_score is:  0.32784712785395553
```

Figure 5.18: The silhouette scores for the different clusters

Now there is an indication of the preferred cluster size, the parameters of the DBSCAN algorithm are left to determine. By again using the knee detection method we decide the parameters to be: $eps = 1.26$ and $min\_points = 5$. The next three figures show our results taken over 5 different samples.[10] The first plot (Figure 5.19) is a sample containing 4 days of data. In the second plot (Figure 5.20), the clusters are extracted and shown individually. The third and last plot (Figure 5.21) shows the result of running the algorithm on 4 different samples of one day. For all samples, we used the same settings and on average 13.47% of the trajectories were outliers. i.e. the percentage of trajectories not assigned to a cluster.

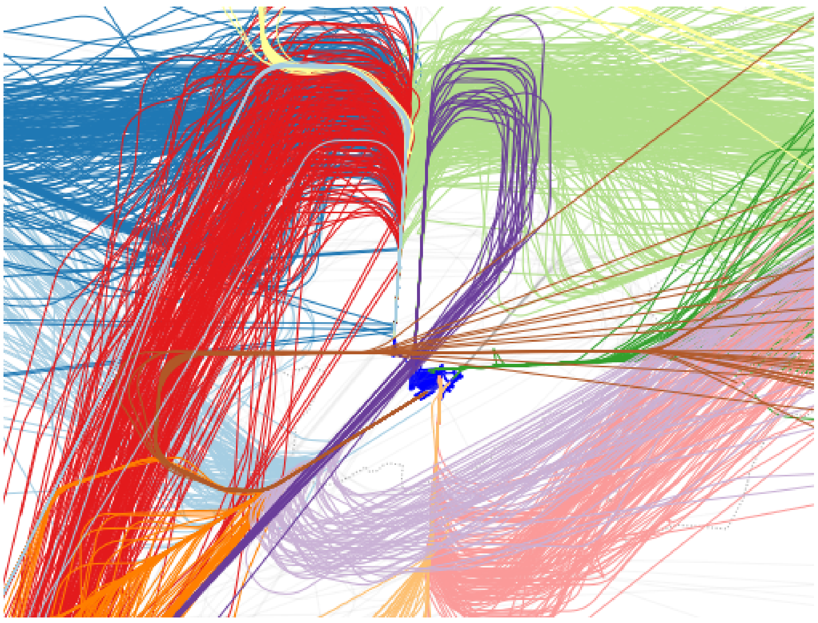


Figure 5.19: Clustering of a 4 day sample.

[10]Unfortunately the parameters scale with the input and density of the graph. We found that over several 1-day samples the results were consistent, however, this could vary depending on the traffic on that current day.

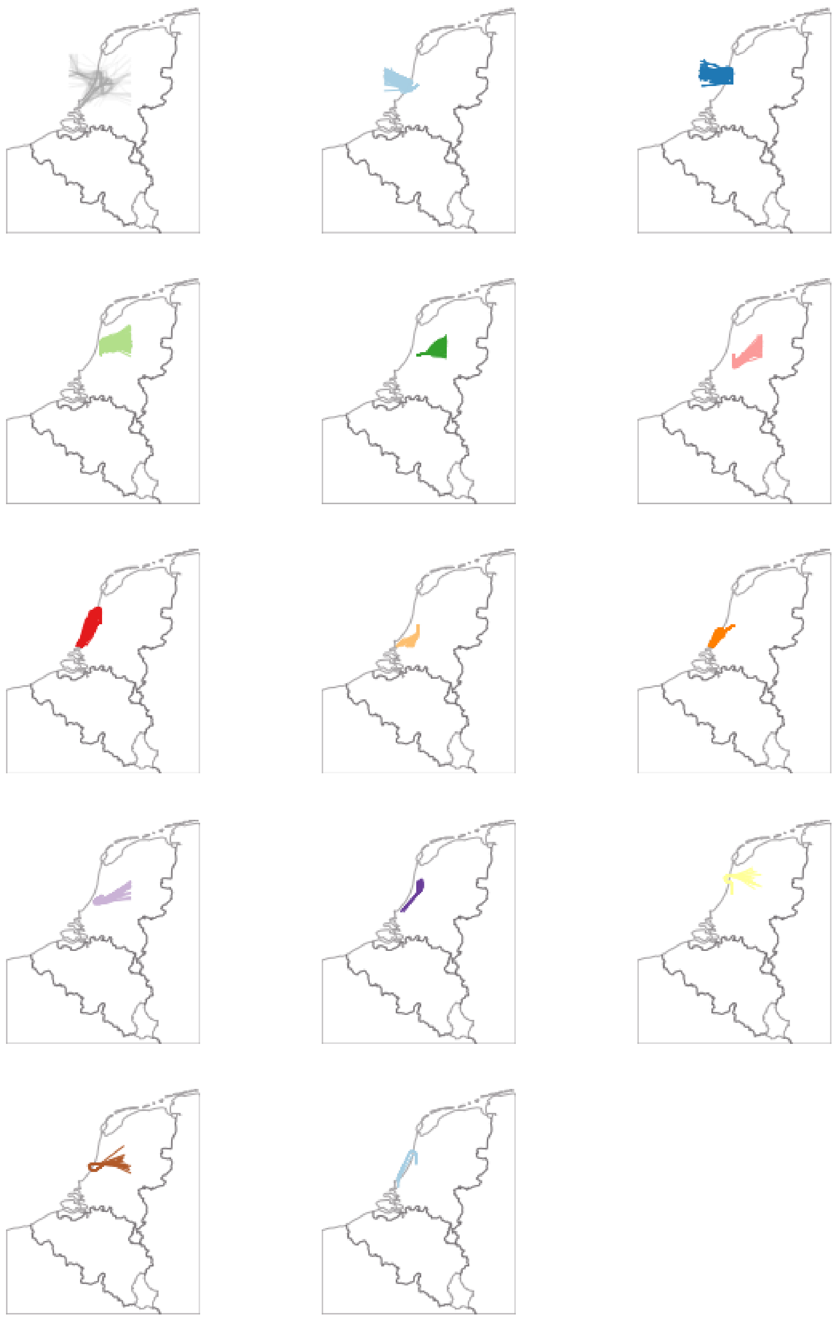


Figure 5.20: Single clusters of 4 day sample.

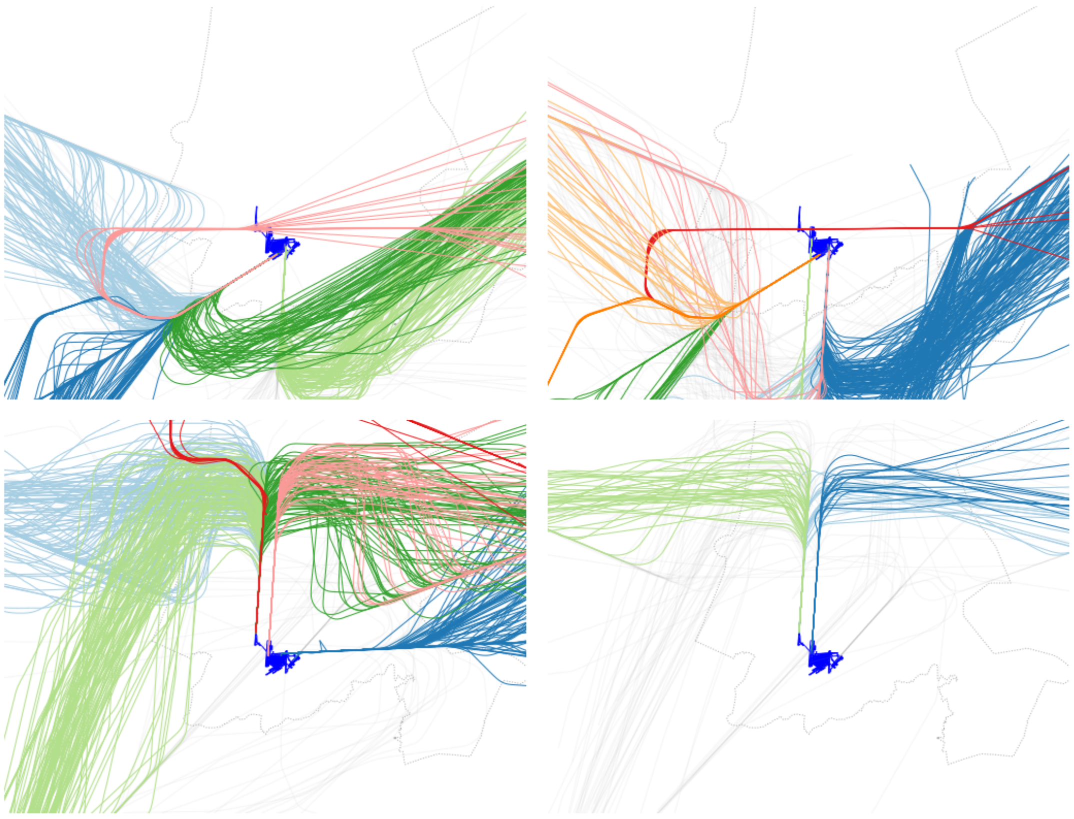


Figure 5.21: Clustering results on 4 random days of traffic.

6 Results

This chapter shows the results of our prediction models in terms of accuracy, supported by graphs.

6.1. Machine learning

The Machine learning section will present the result with regards to model accuracy. During the project two models were trained, the *Runway classification model* and the *ELDT regression model.*

6.1.1. Runway classification model

The runway classification model is a sequential neural network consisting of three layers. An input layer, one hidden layer, and an output layer. The number of input nodes is equal to the number of features, in case 16. For the hidden layer, through empirical methods, we found that 32 nodes served our model best. The output layer has 12 nodes corresponding to the runways.

The accuracy is determined by the *Sparse Categorical Accuracy* which describes the percentage of good predictions it has made. With the current model, we can determine the runway on which it lands with over 80% accuracy (see Figure 6.1). And determine the top 2 with over 92% accuracy (see Figure 6.2). AMS is a complex airport when compared with other airports. It features 6 runways on which airplanes can take off in any direction. Some runways are also parallel to each other. This makes predicting the runway harder, as weather data doesn't differentiate at that point.

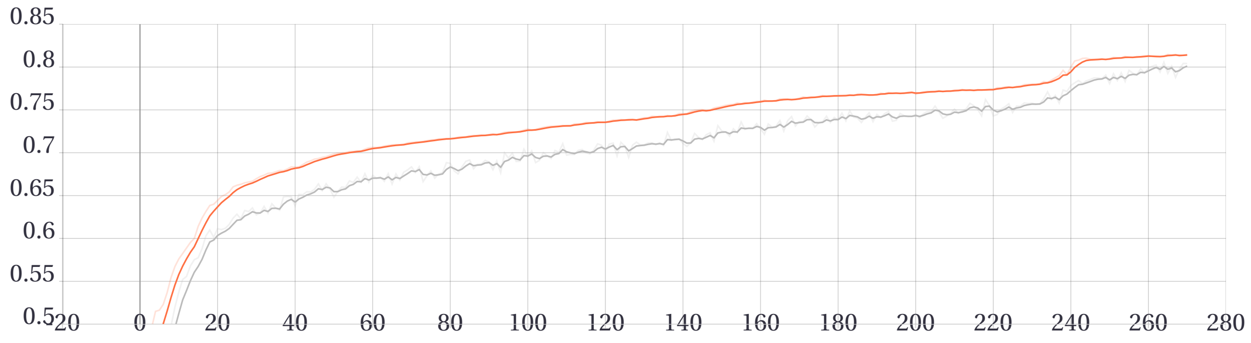


Figure 6.1: The resulting Sparse Categorical Accuracy of a training and validation set

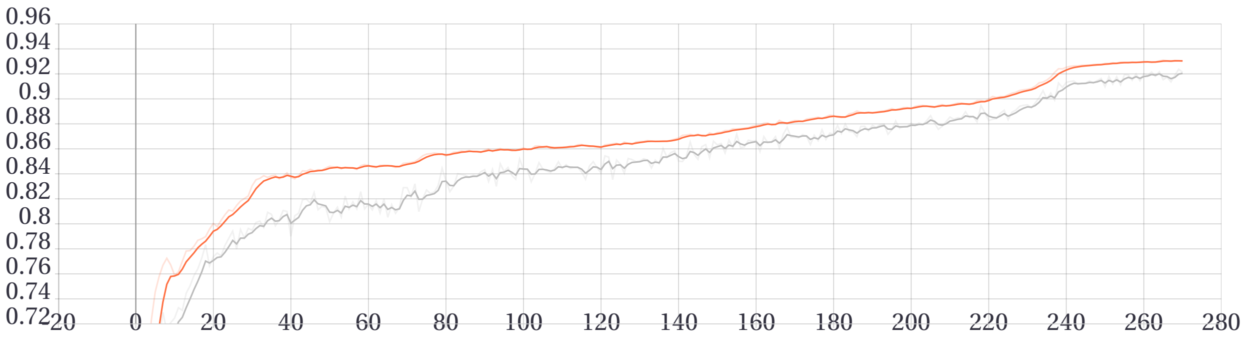


Figure 6.2: The resulting Top 2 Sparse Categorical Accuracy of a training and validation set

6.1.2. ELDT regression model

The ELDT regression model is a sequential neural network consisting of four layers. An input layer, two hidden layers, and an output layer. The number of input nodes is equal to the number of features, in case 30. For the hidden layers, through empirical methods, we found that two layers of 64 nodes each served our model best. The output layer one node which outputs the estimated time to land in seconds.

After training the model through various scenarios we've reached a current optimum with a Mean Absolute Error of 39 seconds (see Figure 6.3). And a Root Mean Squared Error of 52 seconds (see Figure 6.4). As the RMSE isn't significantly higher we can conclude that the amount of outliers is less significant and the model can be trained and predicted reliably.

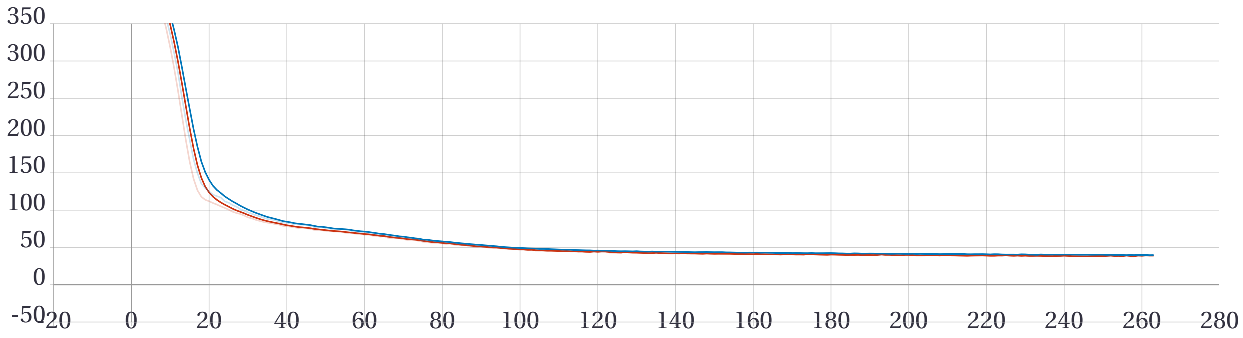


Figure 6.3: The resulting Mean Absolute Error (MAE) of a training and validation set

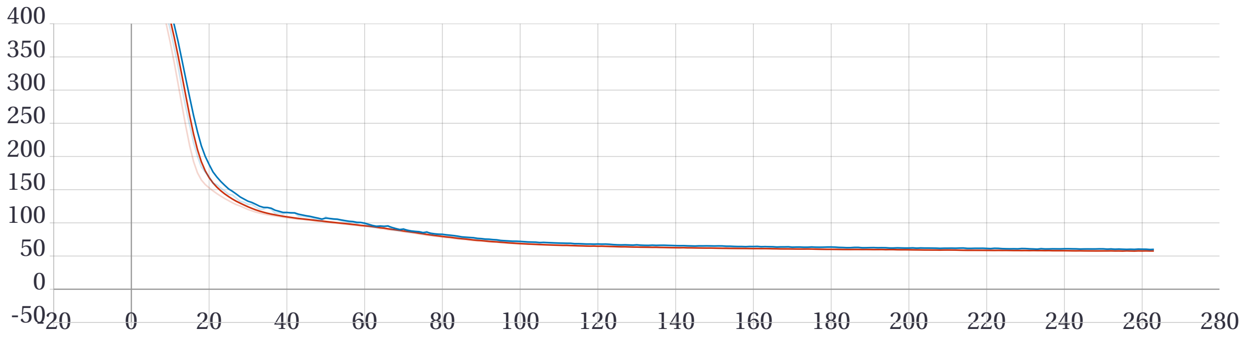


Figure 6.4: The resulting Root Mean Squared Error (RMSE) of a training and validation set

7 Discussion & Recommendations

In this chapter, the requirements and design goals set are evaluated. Then we discuss the results of the prediction models. Subsequently, we consider the implications of our work, especially for the client. Finally, we end the chapter with recommendations to improve our work.

7.1. Evaluation of requirements & design goals

In this section, the requirements and design goals er evaluated. First we will discuss the requirements that we did not meet. Then we will discuss whether we reached the design goals. The table below contains the requirements that were formalised during the project:

Requirement	**Fulfilled**	**Handled in**
1	☑	5.1.1
2	☑	5.1.1
3	☑	5.1.1
4	☑	5.1.1
5	☐	Not handled
6	☑	5.1.2
7	☑	5.1.2
8	☐	5.2.1
9	☑	5.2.2
10	☑	5.2.4
11	☑	5.2.4
12	☑	5.3.1
13	☑	5.3.1
14	☑	5.3.2
15	☑	5.3.2
16	☑	5.3.3
17	☑	A.6.2
18	☑	5.3.4
19	☑	5.3.4
20	☐	Not handled
21	☐	Not handled
22	☑	5.3.4
23	☑	5.3.4
24	☑	5.3.4
26	☑	5.3.4
25	☑	5.3.4
27	☑	5.3.4
28	☑	5.2

As seen in the above table requirements 5, 8, 20 and 21 were not met. Below we discuss why not. The last two requirements are handled together because they are closely related.

- Requirement 5 was not critical for the application (labeled as could have), and we simply did not have any left over time to spend.
- Requirement 8 about the runway prediction is depending on the interpretation not satisfied. However, both we and the client feel that our results are a success. In hindsight, this requirement is not clearly defined. For instance, it doesn't include information on what the time requirements are for predicting with a 90% accuracy. Also, the time horizon is not defined. Predicting on which runway an airplane is going to land on one second before touchdown is something entirely different than when an airplane is half an hour out.
- Requirements 20 and 21 are not satisfied. This is due limitations on available ground data at AMS which is discussed in the next paragraph. One paragraph later we discuss how the system would be able to satisfy the requirements if there were no limitations on the data.

Data limitations on ground movements AMS is among the largest airports in the world. The area covered by AMS is large and there is an extensive runway infrastructure in place that processes large amounts of flight movements.[1] Factors like this result that receivers have a limited line of sight. This is especially true if the receivers are not situated at the airport, effectively resulting in no reliable ground movement data. Since the OpenSky Network is crowd-sourced and run by private data collectors, this is often the case. Therefore the amount of quality data collected on ground movements is extremely limited. In addition, the amount of traffic equipped with an ADS-B transmitter is large. This is not only due to the obvious fact that there is a high amount of traffic. Also, ground vehicles are more likely equipped with transmitters[2], resulting in a non-linear increase in the number of transceivers in relation to the traffic. More signals cause more signal interference, which limits the data.

Satisfying ground movement requirements without data limitations Would there be no data limitations our system would be able to satisfy the requirements? The architecture built enables efficient geospatial and attribute queries 17. This is for instance used for deciding the actual landed runway by:

1. Determining the alignment of the instrument landing system (ILS) with the runway
2. Geo fencing the location of the airplane against the bounds of the runway

Analogous to the method described above the actual taxi time and in-block time could be generated. The result is dependent on the quality of the data which is being impacted by obstructions or line of sight limitations. If the quality is reasonable, errors and noise can be mitigated by creating redundancy in the geofencing methods. For instance, the generated milestone would not be definite until 5 consecutive messages are within the bounds. This will effectively filter out errors, which cause the position to jump all over the region. [3]

Design goals Performance is a design goal that plays a key role in every component of the development process. The most important contributions for each component of the project are listed below:

- At the application side steps were taken to ensure that airplanes are displayed without lag and transition smoothly.
- The achieved inference time is small resulting in efficient usage of computation resources.
- The data processing is optimized for big data set by using vectorization and distributed methods.

[1]496.833 flight movements AMS 2019

[2]The larger the airport, and the more budget an airport has, the more likely it is those ground vehicles are equipped with ADS-B transmitters. Small to medium-sized airports often lack the budget or the necessity.

[3]It is observed that a flight's position jumps within two consecutive messages from Amsterdam to Moscow. This is obviously due to some sort of transmitting or processing error, where a latitude or longitude value is misinterpreted or transmitted

The design goals user experience and data integrity are solely for the application. The following points summarize why we achieved our goals.

- The front end is designed cleanly and in a minimal way.
- The map is styled such to not distract the user.
- Concurrent users see the same data, always.

7.2. Prediction model

The prediction models performed in line with our expectations. As seen in the results chapter we achieved a model reaching over 80% accuracy for predicting the right runway along its entire flight path, and the top two with over 92% accuracy. We have not found a benchmark to compare with this result. We can thus not assume whether this is good or bad in the research field. We can, however, determine the implications for the client as explained below.

The model estimating the landing time reached a Mean Absolute Error of 39 seconds. A similar study on estimating landing times has been done by Wang et al.[45]. This research is however based around the airport of Beijing Capital International Airport (BCIA). The results can't be compared directly. Both studies used a different location, data set and feature set. The average Mean Absolute Error Wang et al. achieved is 69.19 seconds. Multiple factors can be attributed to our improved accuracy. In this study weather data was added as a feature which has a significant impact on the prediction results. Locality advantages of Schiphol versus Beijing need to be explained in order to make a fair, reliable and trustworthy comparison.

Adding additional features could make the model more accurate and reliable. We already know certain factors that impact runway selection. Larger planes only have a subset of runways to land on, including the airplane model and type will presumably add value to the model. Multiple of these features have been identified and are part of the future works section.

7.3. Implications of results

The results in general and the locality of the results have implications for our client and our client's ambitions.

Importance of Amsterdam Airport Schiphol for results The main goal of this project was to investigate whether we are able to predict the landing time and arriving runway of an airplane. The client strongly suggested doing this for AMS. The runway configuration of AMS, consisting of six runways, is in terms of the possible number of runway combinations is the largest in the world. The configuration combined with multiple approach procedures for each runway[4] result in complex traffic patterns. Therefore it is a difficult challenge to predict the runway an airplane is going to land on. This must be derived from the approach procedure identified by the trajectory clustering. However, approach procedures following the same flight path are not unambiguous in the resulting landing runway. Similar procedures and flight paths can, up until the last moment, land on three different runways. This is due to the symmetrical layout of the runways at AMS shown in figure 5.15.

Predicting the landing runway based on trajectory clustering and machine learning is, as of now as far as we know, not done in any other publicly available research. Combined with the fact that AMS is among the airports with the most complex runway configurations and traffic patterns. Our results suggest that the runway- and estimated landing time prediction model could be feasible for all over the world.

Implications for client The goal of AerLabs is to build a lightweight A-CDM tool which uses publicly available data sources. Our project shows the feasibility of an A-CDM tool capable of determining the milestones related to the airborne segments of flights. However, the most important part of A-CDM is the turn-around process. In order to have a commercially viable tool, the entire turn-around process must be incorporated.

[4]For every runway configuration there are multiple approach procedures used for different meteorological conditions, times of the day and to mitigate noise.

In order to do this ground-coverage at the airport is a necessity. Although our project shows that this can not be done on AMS without acquiring additional resources, this does not mean that this would not hold in general. For instance, the research done by [36] shows that a similar tool is feasible that uses solely data from the OpenSky Network. This suggests that the feasibility of an A-CDM lightweight tool using only public data (i.e. OpenSky) strongly depends on the geographic location, size and randomness of the airport chosen. Since the intended market segment is small to medium-sized airports and those airports are more likely to have ground coverage. The premise of only using publicly available data holds not in general.

Ethical implications The tool is to be used as a decision support tool. The decision that can be made with our tool is based on the ELDT prediction and sending the ground handlers to the arriving airplane. The whole project is about timing. The goal is to be on-time, and save minutes, sometimes seconds, of the available resources. We do not store any personal data, nor is there any ethical implication when the prediction is off. In that case, the ground handling team will arrive earlier, or late, or not at all. But in any case, we do not consider this as an ethical implication of our work.

7.4. Recommendations

In this section, we list our recommendations for future work on the project.

Ground coverage Ground coverage can be a limitation if the A-CDM lightweight system is supposed to work out of the box. Open Sky's ground data coverage will not be sufficient for all airports. We recommend that a study is done to determine the coverage per airport that the client is interested in to add to its clientele. Furthermore, ADS-B receivers can be purchased cheaply, ranging from 50 to 1000 euro. We can imagine that upon demonstrating this tool as a proof of concept, receivers may be placed on the airport in collaboration with the airport.

Extra input parameters One thing that could potentially increase the accuracy of our models, is to add the airplane's size of arriving flights that come before the flight we are trying to predict. This is useful because the separation distance between two arriving airplanes is determined by the size of the first airplane. The reason for that is something called 'wake vortex' which every airplane leaves behind in the air. The bigger the airplane, the bigger the wake vortex will be. A small airplane, therefore, has to keep extra distance if it lands after a bigger airplane. But the other way around, an A380 does not have to keep extra distance if it lands after a small airplane because such a small wake vortex will not cause problems to such a big airplane. This separation distance can differ by several nautical miles, which of course affects the landing time.

Longer flight paths As for this project, we did not visualize the whole real-time flight paths from the starting point. We only stored the last 30 fetched position of each airplane. Otherwise, it slowed down the performance significantly as time passes. We had thought about other solutions, but none were really useful after some time. One possible alternative is to drop the real-time path. We could draw an arc between the fixed start point and the dynamic current point. We did not choose this alternative, because the last 30 data points tell more about the exact movement of a flight and retained a clean screen on the map. For the future, we can look for another solution that can visualize the whole flight path while the map is still clear and have high performance.

Hyperparameter tuning Currently, we have empirically determined the parameters for the prediction models. However, an automated way of finding an optimum exists for determining these parameters, which is called Hyperparameter tuning. This would allow for optimizing the model without the need to change the data.

Playback function As of now, unfortunately, we can only playback data from up to one hour ago, because the OpenSky Network does not let us query data with an earlier timestamp. A way to solve this would be to store all the 'current' airplanes' data in the cloud and connect the playback mode to our cloud storage bucket instead of our API. Even though this increases the amount of storage space required drastically, for playbacks from a long time ago it is a solution.

Flexibility Our final product was made with AMS in mind, so can not be used properly at other airports currently. A way to improve upon this is to store static data about airports worldwide in a simple relational database and make a feature where users can select their airport. Then you would have to store and connect things such as airport codes, starting coordinates on the map, runway coordinates, schedules for each airport, etc. to the prediction models and the front end. The models will need to be trained separately for each airport using historical data from that airport. This will require a huge amount of extra data from new airports, and for the sake of practicality, we did not attempt this in our project.

8 Conclusion

For this bachelor end project, AerLabs presented a set of requirements for developing a system that predicts the landing time and the runway an airplane will land on. Initially, we formulated design goals and divided the project's requirements into three components application, machine learning and data.

The development of the prediction models was based on the latest research in the field of airplane trajectory prediction [45] and trajectory clustering [24]. We extended the proposed models by adding meteorological data as an input feature with the aim of improving the prediction accuracy. The most important part of the process was processing the data consisting of trajectory extraction, cleaning noise and errors and augmenting the required features for the prediction models. The final iteration of the runway classification model can correctly predict the runway for 80% of arriving flights, in 92% of the cases the correct runway is one of the two highest probability runways. This prediction is then used to augment the data.

Although the requirements set the bar on 90 %, we consider the result as a success. In hindsight, the requirement was defined too generally, mainly because a concrete time horizon for the prediction was lacking. The landing time prediction model scored a mean absolute error of 39 seconds. This is an improvement over the 69.19 seconds achieved by Wang et al.[45]. However, this is no true comparison and serves as a benchmark because the data, location and feature set is different. Given that the achieved error from the landing time predication model is smaller than errors in related work; and that AMS is among the most complex airports in the world regarding traffic patterns and runway configurations, and considering that our approach for predicting the landing runway of an airplane was novel, both results look very promising.

The goal of AerLabs for is to determine the feasibility of a lightweight cost-effective A-CDM system. Our results successfully validated the initial steps of creating such a system by generating the estimated landing time of an airplane. However, we were not able to compute the additional milestones concerning ground movements because the ground coverage of the Open Sky Network on AMS is not sufficient. Therefore we conclude that by using solely public available data for AMS generating the additional A-CDM milestones can't be realized. This implicates that additional resources such as data from the airport or authorities are needed in a later stage.

However, the premise does not necessarily hold for other airports. Schultz et al. showed that it is possible to generate nearly all A-CDM milestones using only data from the OpenSky Network. The quality of ground data retrieved from the OpenSky Network is strongly dependent on the airport. Since AerLabs intents to supply small to medium-sized airports with this tool AerLabs should investigate the ground coverage on target airports if they want to pursuit using publicly available data. Also, the accuracy of the models could be improved by additional input features such as the airplane's size and model parameter optimization by hyper tuning.

To conclude, all requirements are satisfied but the ones mentioned. The design goals were achieved and the result of the models suggest that it is feasible for AerLabs to develop lightweight A-CDM system.

A
Research Report

A.1. Introduction

The goal of this project is to implement a system that is able to process positional flight data and calculate A-CDM milestones in realtime. We do this for AerLabs, a start-up founded in 2018 that builds digital solutions for the aviation industry. In this section we will briefly introduce some of the general concepts that we have encountered.
The systems includes all steps that are needed to eventually process the data, we have split these steps over the domains. The research document follows the different domains that we have identified. Out of the domains that we have identified the “Prediction Models” and the “Front-end / Visualisation” domain will contain the core of the problem statement. This is where the actual business value is generated and the algorithms for calculating A-CDM milestones live. The other domains are supporting of the solution.
In this project we will work with data provided by the client and other public and/or private data sources. In this research report we will discuss what an optimal solution to this problem would be.

A.2. Problem definition

AerLabs is a start-up founded in 2018 and builds digital solutions for the aviation industry. At the moment their digital solutions are mostly focused on environmental compliance. However AerLabs wants to make the step towards process management & information systems for Airports. One particular interesting process is Airport Collaborative Decision-making (A-CDM). Systems for A-CDM are built by a few multinationals who dominate the market and are only affordable for the largest international airports. However nearly all airports could benefit from using some form of A-CDM. The current advancements in (cloud) computing power, AI and open data, specifically Automatic Dependent Surveillance-Broadcast (ADS-B), provide opportunities for AerLabs to build a low cost and scalable A-CDM system. This system is aimed to be a light-weight A-CDM variant for airports with smaller budgets and less complex operations.

A.3. Background information

In this section we will provide background information necessary to understand the problem.

A.3.1. Airport Collaborative Decision-making (A-CDM)

A-CDM is described by eurocontrol as follows: "As the name implies, Airport CDM is about partners working together and making decisions based on more accurate and higher quality information, where every bit of information has the exact same meaning for every partner involved. More efficient use of resources, and improved event punctuality as well as predictability are the target results." [1] A-CDM consists of the following six concept elements:

- (Airport CDM) Information sharing
- The Milestones Approach (Turn-round Process)
- Variable Taxi Time

- (Collaborative) Pre-departure Sequence
- (CDM in) Adverse Conditions
- Collaborative Management of Flight Updates

For the purpose of our BEP we are only interested in the Milestones Approach. For an in depth discussion of the concepts the eurocontrol A-CDM manual can be used as a reference. [1]

The Milestones Approach describes the progress of a flight from the initial planning to the take off by defining milestones to enable close monitoring of significant events. The aim is to achieve a common situational awareness and to predict the forthcoming events for each flight. [2] In the following graph (Figure A.1) a overview of the milestones is given:

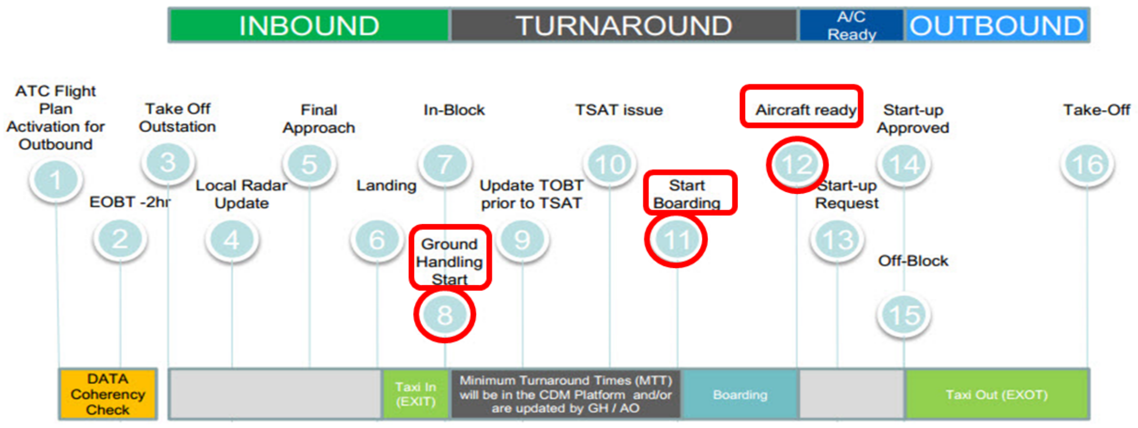


Figure A.1: A-CDM milestones

Our research will be focused on the estimated landing time (ELDT) defined in Milestone 5: Final Approach.

A.4. Scope of problem

In our BEP project we will show the feasibility of a low-cost, scalable, lightweight A-CDM system for AerLabs by creating a architecture that is able to process large amounts of unstructured flight data with The Milestones Approach. We will implement the ELDT, proving the feasibility of our proposed architecture. Furthermore we will develop a proof of concept A-CDM application by developing a GUI visualizing (live) flights and flight milestones.

A.4.1. Requirements

The following requirements are acquired by discussion with the client and TU Delft coach, as well as our research. They are listed below with the MoSCoW method.[46]

Must haves

- The flight data interface must be independent of the data source e.g. it shouldn't matter if the data comes from a static data import or streaming data.
- The data processing architecture must enable efficient spatial, time and attribute queries.
- The architecture must be designed such that it can process 1000 flights concurrently at any moment.
- The architecture must be designed such that the frond-end can have a playback mode.
- The frond-end must visualize flight paths.

- The front-end must have a table view with the data and milestones of the tracked flights.
- The flight data interface must encapsulate all A-CDM milestones
- The system must be able to correctly detect at which runway a flight has landed 99.9% of the time. This will be verified using factual data from the airport.
- The system must make a correct prediction on at which runway a flight is going to land 90% of the time. This will be verified using factual data from the airport.
- The system must generate the A-CDM milestone ALDT - Actual landing time.
- The system must continuously compute the A-CDM milestone ELDT - Estimated landing time.
- The system must be able to generate above stated milestones for all concurrent arriving flights at Schiphol for any given moment.
- The system must enforce data integrity.

Should haves

- The front-end of the system should have a playback mode.
- The system should generate the A-CDM milestone AIBT - Actual in-block time.
- The system should generate the A-CDM milestone AXIT - Actual taxi time.

Could haves

- The frond-end could have a performance dashboard, visualizing the performance of the milestone predictions.
- The system could generate the A-CDM milestone EIBT - Estimated in-block time.
- The system could generate the A-CDM milestone EXIT - Estimated taxi time.

A.4.2. Design goals

In this section we formulate design goals for the project. We will discuss data integrity, user experience and performance.

Data integrity Data integrity is important for several reasons. First, it is important that all stakeholders act on the same data. Secondly, sometimes decisions must be made in a split second, in this case you want to be sure the data is correct. Lastly, for regulation purposes it is important that there is one single source of truth. This means our flight data must be consistent at all times. In the context of our system this means that every instance of our system must serve the same data at all times. I.e. when a user A opens our web application from a different place or device than user B, user A and B must see the same data.

User experience The purpose of the GUI is to create a feasible design for a A-CDM system. In the design process we will think about which data must be displayed, and which shouldn't. Also the map where the flights are rendered on should be styled such that the user is not distracted and only sees relevant information. Furthermore the GUI should render concurrent arriving flights at Schiphol in a fluent motion, free from sudden and unexpected position changes.

Performance Performance is of great importance since the system must calculate the milestones from all concurrent flights at the same time. In order to process all concurrent flights around Schiphol at any given moment, we would approximately have a peak load of 300 flights. This is based on the daily fly movements at Schiphol airport. All these flights needs to be tracked, and their ELDT and possibly EIBT must be updated continuously in order to achieve accurate results. Not only will this be computationally heavy and put a strain on the available capacity. All flights must be rendered in the GUI as well. Because of both these requirements, we must put good thought in how we design the architecture. We want to be able to track and process 300 concurrent flights with ease. Also we want the system to be scalable in the amount of concurrent flights it can handle.

A.5. Prediction models

The core of our project is the prediction of the landing time(ELDT) and runway for arriving aircraft. In order to do this the trajectory of an airborne flight must be predicted. We first discuss a literature survey about trajectory prediction concluding the proposition of a model. Then we discuss the methodology of the proposed model. Finally we propose a system architecture to implement the model. The necessary performance has been specified in the requirements.

A.5.1. Literature survey

In the field of trajectory prediction, there are two main disciplines. One is driven by aircraft performance models, which try to imitate nature by modelling the actual physics and dynamics of a flight. The other discipline is centered around machine learning approaches. A comprehensive review of the differences is outside of the scope of this research. Naturally as computer science students we will be focusing on the machine learning approaches. At the end of the 20th century, in 1999 Yann le Fablec et al. showed that aircraft trajectories can be predicted with neural networks.[22] This spiked interest in this research topic since it showed that machine learning models could outperform (kinetic based) aircraft performance models. In 2013 De Leege et al. introduced Generalized Linear Models for short-term trajectory prediction at Schiphol Airport.[21] Their model assumed fixed CDO[**?**]arrival routes and used open data that was readily available. A prediction was made along way-points on the fixed route. A limitation of the model is the dependence on the fixed CDO route. This makes the method not robust, especially for a large scale data set or streaming data that also contains overflights, holding patterns and other arrival procedures. Research by S. Trivedi et al. showed that clustering as pre-processing step improves the prediction accuracy significantly on large-scale clusterable data-sets [42]. This was put to test in the research by Tastambekov et al. In 2014 They made a short-term prediction of the flight trajectory which first clustered the data based on similar trajectories [41]. This method turned out to be robust and therefore deploy-able on an unstructured flight data set or streaming flight data. One year later S. Hong et al. used a similar method but enriched the data with the ATC intent information [25]. At last the final contribution in this field we are discussing is the model proposed by Z. Wang et al. in 2017. Their model uses the DBSCAN method by Gariel et al. to cluster flights combined with PCA for robust data pre-processing [45]. Then it uses a MCNN-based machine learning for 4D flight trajectory prediction. The model uses open available ADS-B data. However it does not include surface or altitude winds, nor the ATC intent used in previous papers. In the next section we will propose a model that extends the model proposed by Z. Wang et al. by adding the ATC intent, the surface- and altitude winds.

A.5.2. Methodology

This section described how the proposed model works. Most of it is in overlapping with the methodology introduced by Z. Wang et al. We will provide an overview of the whole model, followed by the detailed process from step to step.

Overview The whole model can be divided into two part: pre-processing and machine learning. In the part of pre-processing, there are mainly three steps: resampling and data augmentation, dimensionality reduction by Principle Component Analysis (PCA), and clustering by Density-Based Spatial Clustering of Applications with Noise (DBSCAN). In the part of machine learning, Multi-cells Neural Network (MCNN)-based machine learning is applied. MCNN is trained with Mean Absolute Error (MAE) and Root Mean Square Error (RMSE). They are illustrated in figure A.2. The more detail will be separately discussed.

Data preparation The starting point for the model is the dataset. We should make a choice from the entire dataset we've received because not all the data is needed for prediction. We also transform the longitude, latitude and altitude into 3D Cartesian coordinates. Apart from 4D trajectory information, three spatial dimensions and one time dimension, we also include data such as info about the aircraft, ATC intent, and the meteorological at atmosphere and ground. An example of required data:

1. Type of operation (departure/arrival),
2. Record beginning time t,
3. Time_position,
4. Time_velocity,
5. Aircraft number i,
6. Position (X, Y, Z),
7. Heading Ψ,
8. Horizontal velocity V_h,

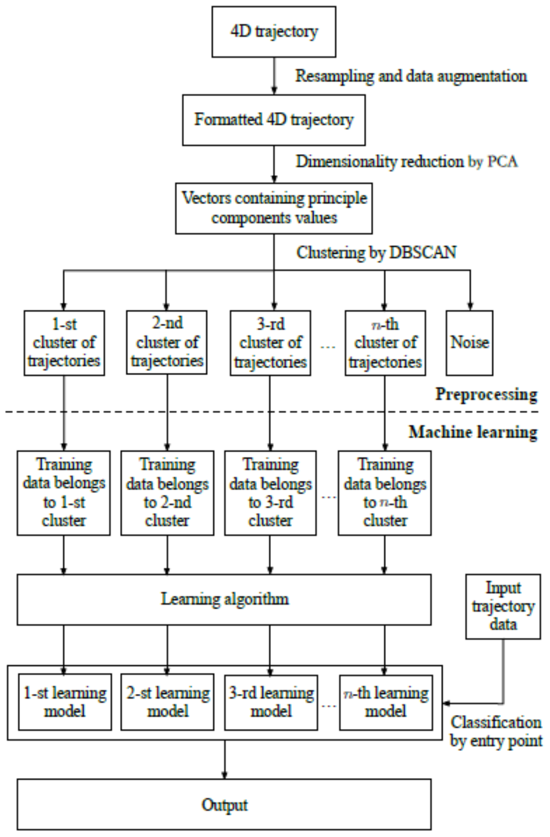

4D trajectory
Resampling and data augmentation
Formatted 4D trajectory
Dimensionality reduction by PCA
Vectors containing principle components values
Clustering by DBSCAN
1-st cluster of trajectories
2-nd cluster of trajectories
3-rd cluster of trajectories
...
n-th cluster of trajectories
Noise
Preprocessing
Machine learning
Training data belongs to 1-st cluster
Training data belongs to 2-nd cluster
Training data belongs to 3-rd cluster
Training data belongs to n-th cluster
Learning algorithm
Input trajectory data
1-st learning model
2-st learning model
3-rd learning model
n-th learning model
Classification by entry point
Output

Figure A.2: Methodology overview

9. Vertical velocity V_v,
10. Atmosphere temperature,
11. Atmosphere wind speed,
12. Atmosphere wind direction,
13. Atmospheric humidity,
14. Atmospheric pressure,
15. Ground temperature,
16. Ground wind speed,
17. Ground wind direction,
18. Ground humidity,
19. Ground pressure,
20. Ground wind gust, etc.

Each trajectory $T_i, i \in [[1, n]]$ are grouped together according to the aircraft number i. n is the total number of trajectories in the dataset. ADS-B data receivers do not receive all the records of trajectories. To compensate with the missing part, a low pass filter is applied to each trajectory. The 3D coordinates (X, Y, Z) and heading Ψ of the k-th point of i-th trajectory x_i^k. Each point is influenced by its previous point, except for the first point and heading of each trajectory point. A smoothing factor of 0.5 is chosen to prevent too much delay. The data is cleaned by replacing x_i^k with the new point $\tilde{x}_i^k$, that is calculated using following equations:

$$\tilde{x}_i^1 = x_i^1 \tag{A.1}$$

$$\tilde{x}_i^k = 0.5x_i^k + 0.5\tilde{x}_i^{k-1}, r \in [2, m_i - 1] \tag{A.2}$$

m_i is the total number of points in i-th trajectory. Trajectories with m_i less than 50 are eliminated, because too much data is lost. To sufficiently process the data, all points are then formatted into a vector with length 50 using the following re-sample method:

$$T_i = \{T_i^k | k = round(\frac{a \cdot m_i}{50}), a \in [1, 50]\} \tag{A.3}$$

Except from the features we have represented in the lists, more features can be derived from the list. This is how we enrich the existing feature with additional features: distance to the reference point R, distance to the corner point D, angular position from the reference point Θ. $(X_{ref}, Y_{ref}, Z_{ref})$ denotes the reference point. It can be seen as the geometric centre of runway configuration. R_i^k is the distance between each trajectory point and the reference point. $(X_{cor}, Y_{cor}, Z_{cor})$ denotes the corner point. It is the intersection points of runway configuration. D_i^k is the distance between each trajectory point and corner point. Angular position Θ_i^k is the angle between each trajectory point and the reference point. These three features are computed as shown in the following three formulas:

$$R_i^k = \sqrt{(X_i^k - X_{ref})^2 + (Y_i^k - Y_{ref})^2 + (Z_i^k - Z_{ref})^2} \tag{A.4}$$

$$D_i^k = \sqrt{(X_i^k - X_{cor})^2 + (Y_i^k - Y_{cor})^2 + (Z_i^k - Z_{cor})^2} \tag{A.5}$$

$$\Theta_i^k = arctan\left(\frac{Y_i^k - Y_{ref}}{X_i^k - X_{ref}}\right) \tag{A.6}$$

The angular position and heading are adopted with sine and cosine to avoid the discontinuity. After that, each original feature x is replaced with the normalized feature x^* in $[0,1]$ using the following formula:

$$x^* = \frac{x - min(x)}{max(x) - min(x)} \tag{A.7}$$

The final step of this dimensionality augmentation is to order the features of trajectories as shown with vectors below, where P_i^* is a vector of $[X^*\ Y^*\ Z^*]$:

$$T_i = [P_i^*\ R_i^*\ D_i^*\ cos(\Theta)^*\ sin(\Theta)\ cos(\Psi)^*\ sin(\Psi)^*\ etc.] \tag{A.8}$$

$$T = \begin{bmatrix} T_1 \\ \vdots \\ T_n \end{bmatrix} \tag{A.9}$$

This final step of data preparation is PCA. PCA is used to reduce the vector variable dimensions. In contrast to the previous step, the less significant feature will be removed from the data. We keep 95% of the dataset. C is the covariance matrix of T that shows the correlation between variables. The eigenvalues of C, λ_i, are computed to identify the principal components Y. The larger the eigenvalue, the larger the variance $var(Y)$.

$$C = \frac{1}{n-1} \cdot T \cdot T^T \tag{A.10}$$

$$Y = E \cdot T \tag{A.11}$$

$$var(Y) = E^T \cdot C \tag{A.12}$$

E is the rotation matrix. $\{\lambda_i | i \in [[1,m]]$ is ordered in such a way that $\lambda_1 > \lambda_2 > ... > \lambda_n$. The following equation shows up to which number q the variable will be kept:

$$G(q) = \frac{\sum_{i=1}^{q} \lambda_i}{\sum_{i=1}^{m} \lambda_i} \geq 95\% \tag{A.13}$$

DBSCAN cluster-based pre-proccesing This step divided dataset into clusters. At first, the neighbourhood radius ϵ and the minimum number of points required to form a cluster $MinPts$ should be chosen. Then, they are put into DBSCAN algorithm 1 and 2, together with the prepared dataset, to determine the respective cluster.

MCNN-based learning model The input layer consists of n independent variables $\{x_j | j \in [[1,n]]\}$. The hidden layer consists of m nodes $\{m_i | i \in [[1,m]]\}$. The output layer is the predicate node y, calculated as below:

$$f(z) = \frac{1}{1 + e^{-z}} \tag{A.14}$$

$$y = \sum_{i=1}^{m} w_i^2 f\left(\sum_{j=1}^{n} w_{ij}^1 x_j + b_i\right) + c \tag{A.15}$$

Algorithm 1: DBSCAN algorithm for forming clusters with respect to Eps and MinPts

```
function DBSCAN (setOfPoints, Eps, MinPts);
ClusterId := nextId(NOISE);
for i From 1 To SetOfPoints.size do
    Point := SetOfPoints.get(i);
    if Point.ClId = UNCLASSIFIED then
        if ExpandCluster(SetOfPoints, Point, ClusterId, Eps, MinPts) then
            ClusterId := nextId(ClusterId)
```

Algorithm 2: ExpandCluster for expanding the cluster

```
function ExpandCluster (SetOfPoints, Point, ClusterId, Eps, MinPts) : Boolean;
seeds := SetOfPoints.regionQuery(Point,Eps);
if seeds.size<MinPts then
    //no core point
    SetOfPoint.changeClId(Point,NOISE);
    return False;
else
    // all points in seeds are density-reachable from Point
    SetPofPoints.changeClId(seeds, ClId);
    seeds.delete(Point);
    while seeds <> Empty do
        currentP := seeds.first(); result := SetOfPoints.regionQuery(currentP, Eps); if result.size >=
          MinPts then
            for i From 1 TO result.size do
                resultP := result.get(i);
                if resultP.ClId IN (UNCLASSIFIED, NOISE) then
                    if resultP.ClId = UNCLASSIFIED then
                        seeds.append(resultP)
                    SetOfPoints.changeClId(resultP);
                // UNCLASSIFIED or NOISE
        // result.size >= MinPts
        seeds.delete(currentP);
    // seeds <> Empty
    return True;
```

b_i is the bias to the i-th hidden node. c is the bias to the output node. w^1_{ij} is the weight between the j-th input node and the i-th hidden node. w^2_i is the weight between the i-th hidden node and the output node. f is the activation function. The Sigmoid function is used to get a continuous output, which makes back propagation possible. The cross-entropy cost function J shows how well the prediction is made:

$$J = -\frac{1}{N}\sum_x [t\,ln\,y + (1-t)ln(1-y)] \tag{A.16}$$

N is the number of training data. t is the target output. The errors will be used to update the weights using the back propagation algorithm. The dataset is divided into a test set, training set, and validation set. They are trained and tested with nested cross validation. The performance of the prediction is evaluated with MAE(Mean Absolute Error) and RMSE (Root of Mean Squared Error).

A.5.3. Architecture

If we don't want to implement the entire neural net architectures ourselves (and we don't) we will have to choose a framework to use. Considering machine learning is becoming more and more popular we've seen many frameworks pop up in the last couple of years, the two main ones being Google's TensorFlow [17] and Facebook's PyTorch [15]. For our choice of framework we will limit ourselves to these two options. While either of these frameworks could be used each has its own strengths and weaknesses. TensorFlow's main advantages being its strong visualisation tools (TensorBoard), its production-readiness due to TensorFlow serving, and its large community, and PyTorch's being the fact that it is native to Python and the use of dynamic graph structure as opposed to TensorFlow's static one.

Both of these frameworks are entirely suitable to be used in this project however we decided it is best to go with TensorFlow as it is more friendly to being used in a production environment and therefore the most used deep learning framework in commercial applications.

Tensorflow's high level API, called Keras[9], will enable us to quickly and effectively implement the neural network as described above. Keras allows for the generation of neural net models in just a few lines of code and enables us to add layers just by using the tf.keras.layers.Dense() command with as its arguments the amount of neurons we want the layer to have and the activation function. The training of the network is don just by calling model.fit() with as its arguments the data on which we want to train and the amount of epochs to train it for. This enables us to focus on creating an efficient and accurate model without having to worry about the underlying implementation or the math behind the backpropagation.

This means that changing and tweaking the settings of the network will take next to no time or effort allowing us to spend more time on getting the network as accurate as possible.

A.5.4. Data Sources

An essential element for success in our project is the access of data. Nowadays most flights are equipped with ADS-B and/or Mode-S transmitters.[38] This data can be gathered by anyone with the right equipment. However this is raw data. And difficulties arise in the processing of this data which we will not address here. We are interested in raw flight data that is already processed into positional flight data. This way we can speed up our process and focus on the requirements of the system, and leave the flight data processing for future development.

The client provided us with a .parquet file that contains a Schiphol data set. This data set has the following headers:

track_point_id	flight_id	type_code	date_time	lon	lat	altitude(ft)

The above Schiphol data set might not be enough, also it is a static data set while flight data is streaming data by nature. Therefore we plan to also get a data-set and a live-stream from the Open Sky Network. They provide this data for free for non-profit organizations, governments and research institutions. The flight data they offer is modelled as state vectors, derived from ADS-B and Mode s messages. The state summarized all all tracking information at a certain point in time. There is extensive documentation [30] available and they have both a REST and Python API. The fields found in table A.1.

A.6. Application

In this section we will cover all the domains involved in making the final product (a web application). Next to the prediction model there are a lot of moving pieces to get an application running. We've identified the

icao24	*ICAO24 address of the transmitter in hex string representation.*
callsign	*callsign of the vehicle. Can be None if no callsign has been received.*
origin_country	*inferred through the ICAO24 address*
time_position	*seconds since epoch of last position report. Can be None if there was no position report received by OpenSky wit*
last_contact	*seconds since epoch of last received message from this transponder*
longitude	*in ellipsoidal coordinates (WGS-84) and degrees. Can be None*
latitude	*in ellipsoidal coordinates (WGS-84) and degrees. Can be None*
geo_altitude	*geometric altitude in meters. Can be None*
on_ground	*true if aircraft is on ground (sends ADS-B surface position reports).*
velocity	*over ground in m/s. Can be None if information not present*
heading	*in decimal degrees (0 is north). Can be None if information not present.*
vertical_rate	*in m/s, incline is positive, decline negative. Can be None if information not present.*
sensors	*serial numbers of sensors which received messages from the vehicle within the validity period of this state vecto*
baro_altitude	*barometric altitude in meters. Can be None*
squawk	*StartFragmenttransponder code aka Squawk. Can be None EndFragment*
spi	*special purpose indicator*
position_source	*origin of this state's position: 0 = ADS-B, 1 = ASTERIX, 2 = MLAT, 3 = FLARM*

Table A.1: Format definition of Open Sky network messages

domains according the representation in the figure A.3. This includes how we process our raw data, how we store data, the front end of the application, and a bit about configuring all the components and services we use to work together.

A.6.1. Data Ingestion

Problem definition / research question In order to process the data we first need to collect it and make it available for processing. This step we call Data Ingestion. The data mentioned in the data sources domain is available by various means. Data can be extracted from an API endpoint where for example JSON blobs are returned upon request. Next to that there will be cases where we need to gather the contents of static files hosted somewhere. Or we will need to expose an endpoint in order for a third party to push data to u sin a streaming manner.
Making sure we ingest this data in a reliable and easy to scale way is important. It needs to be easy for us to add or remove data sources dependent on the need for it by the models that use them.
The various sizes, shapes and formats of the data need to be channeled to standardised format to be used by the model. The model shouldn't have to worry about any of that because it is a separation of concern. Key to a good ingestion is defining a standardized format in which we transform the data. This is basically the contract that is created between the ingestion layer and the application layer where the model resides.
What should an implementation of this subsystem look like?

Existing market solutions Ingestion of the various sources defaults to a custom approach. There is unfortunately no silver bullet. A piece of software will need to be written to extract the data from the source and transform it to the required format. How this piece of code runs however is dependent on the platform of choice. Amazon Web Services provides a product called Lambda to allow pieces of code to be ran as serverless functions. This has the benefit that no infrastructure other than the Lambda configuration needs to be managed. Google provides a similar product called Cloud Functions. Dependent on customer preference one of these can be chosen, for the development of the subsystem there is no relevant difference to describe. The transformed data will be sent to the storage layer described in the storage domain.

A.6.2. Data Storage

Problem definition / research question In this section we focus on how to store, not what to store. Because all the data we use is already formatted appropriately and this section can disregard the content of the data, as the data ingestion domain takes care of the formatting and content. To be able to fulfill certain requirements like being able to replay past data, we will need to store the data we use somehow, such that we can use it

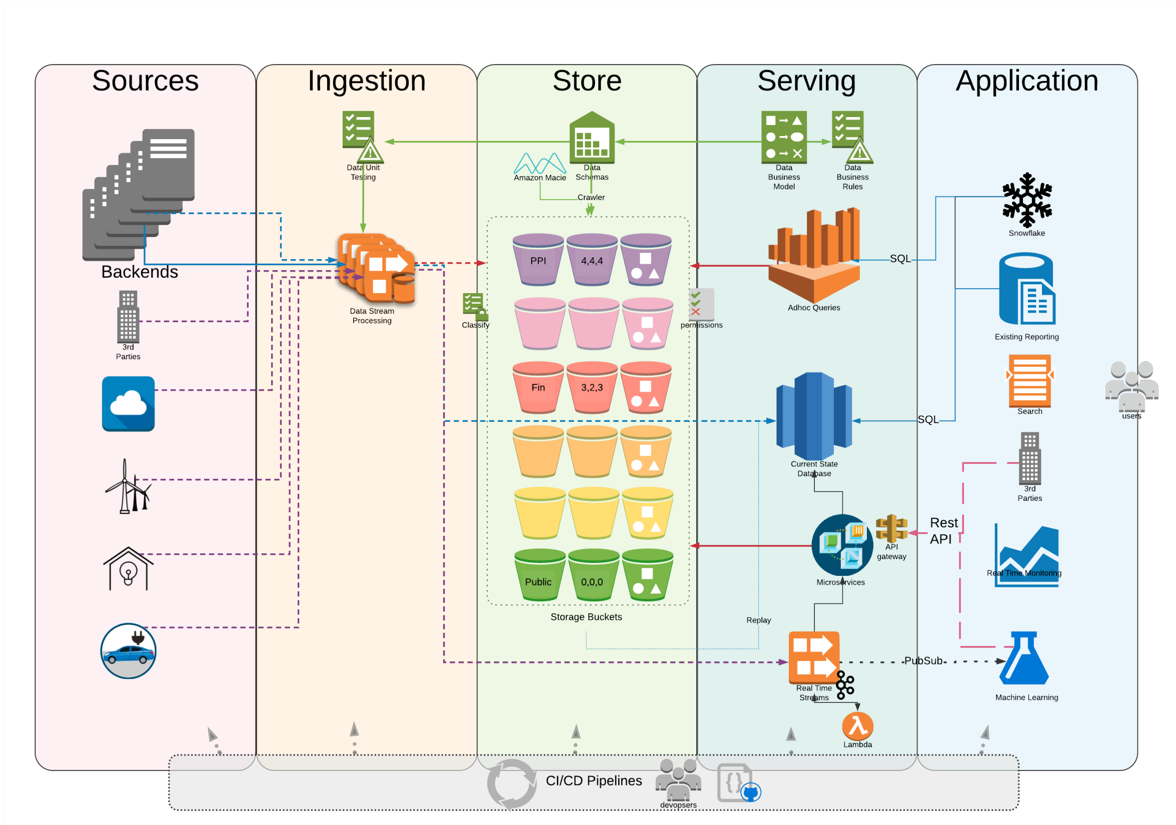


Figure A.3: Reference architecture of a data processing pipeline by Schuberg Philis B.V.

later for features like that. Storing data can also be useful for promising front end integrity, such that there is one data store from where separate clients/components will always get the same data. So it is necessary to store data, that brings us to the research question: which architecture for storing and querying data is best suited for Geo-spatial and time related queries? Another point that we need to consider is: how far into the future will data be kept? Naturally it can become expensive in terms of money and space to keep everything forever. We may need to delete data eventually, to keep those costs limited. In line with the requirements, an important subquestion to research is: how can this architecture enable a playback of the data?

Possible solutions There are several ways to store data available as of now. In this case, the first decision that needs to be made is to use cloud storage or local storage. This is relevant for how the data is stored and for security as well. We have decided to use cloud storage for this project, because using a cloud database is preferable over local storage when the design goal of maintainability and practicality are considered. "Cloud computing databases are extremely popular for reducing IT complexities and operational costs. They prevent the hassle of licensing, traditional procurement, maintenance, and installation involving a huge number of IT staff. With the rapid increase in the business pace, cloud databases enable organizations to cut down on the in-house IT resources required to manage huge sets of data."[39] Now we need to decide whether we are going to use a database or not. To do this it is necessary to know how much data we are dealing with. The data comes from ADS-B exchange, and will add up to 10-20GB of data per day [3]. That means storing data of the past month (which we believe is a somewhat realistic desire for user companies) would take up roughly 300-600GB of space. Querying and performing the required computation on a database of this size for a real-time view while new data is also being added every minute, is not optimal. Instead of a database it is more suitable to use a streaming platform for the real-time view and prediction, because they separate the data from the computation. That will lead to increased scalability and performance [16], also making the replay feature more practical. But there are some things we want to store which fit nicely into a database: referencing tables. For example one of the tables contains which airport has which three letter acronym. Such information is required for a user friendly front end application, and is structured static data, so using a database is a good way to handle this data. Considering the fact that queries for this reference database will be very similar to each other, and joins/merges may be used in it, query optimizations can be used. For these reasons, a standard relational database fits the best here. So which streaming platform are we going to use to achieve the real time view? Google has one called BigQuery [4]. BigQuery has native support for geospatial analysis which is useful in our case. Especially considering that many of our queries will use a range for coordinates. A nice advantage of using BigQuery is that it comes with cloud storage. This combination of functionality makes it easier to use compared to a combination of two different platforms for cloud storage and streaming data. It also supports regular SQL so writing queries will not be a problem. Another option is Kafka [8], an open source streaming platform. There is a suite of tools that uses Kafka to provide geospatial functionalities as well [7]. We consider Kafka over Hadoop because for the real time view the most important functionality is processing a data stream, which is more suited to Kafka than to Hadoop.

Conclusion To answer the research questions: in our case using the cloud is very practical, and there are several options available that support geo-spatial and time related queries. To facilitate a real time view, and make a playback of the data possible, a streaming platform is more suitable than a regular database. We have decided to use Google BigQuery for this project. The reason is that all the mentioned options are usable for the real time view, but if we use BigQuery we can also use it for storing data to train a machine learning algorithm. So it is the most convenient option in this case. For the relational database a simple MySql database will suffice.

A.6.3. Front-end / Visualisation

Problem definition / research question The front-end provides users with the interface that enables the visualisation of the data and allows users to interact with the data. The goal is to develop a GIS application. We desire to deliver an application that can render 1000 flights concurrently and without visible delay. Thus, real-time visualisation of data is very important for the application. The choice is made to build a single-page application (SPA). SPA is a web interface that can update or replace the page's components without reloading the entire page.[27] Therefore, a dynamic web page can be attained with the SPA. The application should have mainly two views: a live view with visualized flight paths and a table view with the data and milestones of tracked flights. Other non-critical requirements for the front-end are a playback mode and a performance dashboard that displayed the performance of the milestone predictions. The most challenging part is the visualisation of flights. Since the SPA is often implemented using JavaScript,[23] this section will be focused on which GIS-related JavaScript libraries and how they can be used for our application. On the basis of this, we will decide which JavaScript frameworks we are going to use.

Existing market solutions Due to the limited time, we only researched three widely used mapping libraries. Because of their popularity, they are better documented and have the most example so we can pick it up more easily.

Mapbox GL JS[11] is a JavaScript library that uses *WebGL* to render interactive maps from vector tiles and styles. *WebGL* is a cross-platform, royalty-free web standard for a low-level 3D graphics API.[19] WebGL rendering under Reactive programming can handle the streaming input data. Vector tiles and styles followed Mapbox Vector Tile Specification and Mapbox Style Specification respectively. Styles are applied to the vector tile, that stores geospatial vector data.[12] Mapbox vector tiles and Mapbox styles form the most prominent features of Mapbox GL JS. The use of vector tiles instead of traditional raster tiles benefits the users with the zooming functionality and rendering of labels.[26] The user can choose from a range of styles to customize the maps at ease. The integration of Mapbox GL JS with Angular, React, or Vue can be installed via *npm*. React wrapper has by far the most releases and commits out of these three frameworks.

Similar to Mapbox, Leaflet[10] also stands for interactive maps. It is an open-source JavaScript library for mobile-friendly interactive maps. It stands for simplicity, performance, and usability. It is indeed light-weighted and simple to use. But to pay it, it does not work well with complicated GIS application. It is extended with lots of plugins. React is the leading trend in the *npm* download graph for Leaflet maps. OpenLayers[14] is an open-source JavaScript library for a dynamic map. In its official website, it has listed four features: (1) tiled layers, (2) vector layers, (3) cutting edge, fast, and mobile-ready, and (4) easy to customize and extend.[14] Tiled layers ensure that tiles from any source can be pulled. Vector layers can render vector data from any formats. OpenLayers uses Canvas 2D, WebGL, and HTML to access components with variant functionality. It also supports CSS and a range of useful third party libraries to customize and extend functionality. Angular has the most comprehensive wrapper of OpenLayers. React and Vue have only a minimal wrapper.

Findings If Mapbox GL JS is chosen, we would likely to install react-map-gl. This library can initialize and track the state of a Mapbox WebGL map. Not all the functionality of Mapbox GL JS' Map class is in this library because React is not built to manipulate the use of HTML5 canvases and WebGL.[44] Same for Leaflet, react-leaflet is what we want to install. If we apply OpenLayers, we will probably install ngx-OpenLayers. This wrapper has supported each OpenLayers class with a corresponding Angular component.[33] But all necessary functions are also available to React and Vue.

Based on analyses on syntax, architecture, data management, lifecycle, and third party package of these three frameworks, Angular has the steepest learning curve while Vue is the easiest framework to learn.[34] Justified performance is hard to measure. From the research of Prof. Ockelberg and Prof. Olsson, angular scored best at DOM-manipulation, memory allocation; React scored best at start-up time; they get the same score in build size.[32]. The scores don't show a big difference between these three frameworks. In general, all three frameworks showed similar performance.

Conclusion In the light of the analysed libraries and frameworks, React is best suited to our application. We will mainly use Mapbox for its WebGL-based advantage. OpenLayers can be used to visualize different kinds of map data from multiple sources. The leaflet is useful if we want to extend the application with a specific plugin. Along the way of development, more mapping libraries may to our sight than the libraries mentioned in this section. All things considered, we decided to choose React as our JavaScript framework for our application.

A.6.4. Automation

Problem definition / research question In this project certain pieces of software will be used that were not created by us. All of these components need to be able to work together properly for the final product.

Existing market solutions Infrastructure as code (IaC) is a good way to solve this problem. IaC means that there will be so-called 'configuration files'. These files contain all the necessary definitions that the software components use. The idea is that the configuration files contain what needs to be done, and how that is done remains the responsibility of the component/tool that is being used [29]. A benefit of IaC is that it can be used to create one single source of truth for all software components, which satisfies the front end integrity requirement. One existing platform to use IaC is Terraform by HashiCorp. "Configuration files describe to Terraform the components needed to run a single application or your entire datacenter. Terraform generates an execution plan describing what it will do to reach the desired state, and then executes it to build the described infrastructure. As the configuration changes, Terraform is able to determine what changed and create incremental execution plans which can be applied. The infrastructure Terraform can manage includes low-level components such as compute instances, storage, and networking, as well as high-level components such as DNS entries, SaaS features, etc." [18]. Another well-known platform for IaC is AWS(Amazon Web Services) cloudformation. "AWS CloudFormation provides a common language for you to model and provision AWS and third party application resources in your cloud environment. AWS CloudFormation allows you to use programming languages or a simple text file to model and provision, in an automated and secure manner, all the resources needed for your applications across all regions and accounts." [6]. After AWS cloudformation, AWS CDK was made. CDK is essentially a platform which addresses the weaknesses of Cloudformation. It uses constructs to represent architectures. These construct can be made into Cloudformation templates and easily used in an AWS environment. A useful comparison between Cloudformation and CDK can be found here: [5].

Conclusion All of the above platforms seem usable and sufficient for this project, so we will choose whichever one we feel is the easiest to use.

A.7. Conclusion

During the analysis of the problem statement we identified two main parts. At the core of our project is the prediction of the landing time(ELDT) and runway for arriving aircraft. This part will mostly be focused on implementing a machine learning model which will achieve set requirements. Supportive of the core is the implementation of the machine learning model in an application architecture so that it can be ran in a production-like environment.
Based on research by Z. Wang et al. we are going to implement an improved Multi-cells Neural Network

(MCNN) based machine learning model.
We decided it is best to go with TensorFlow as it is more friendly to being used in a production environment and therefore the most used deep learning framework in commercial applications. Python will be the programming language for the machine learning approach. Through exploration we will gather the details of the machine learning implementation.
Research and market analysis of the supportive components of the prediction model have revealed the state-of-the-art methods and tools available to achieve the set requirements. We have chosen to start our initial development on Google Cloud Platform where we will levarage Google BigQuery for data storage and Google Collab to work together on developing the machine learning model.

B
Info Sheet

General Information

Title of the project: Estimating landing time based on historical data
Name of the client organization: AerLabs
Date of the final presentation: July 2, 2020

Description

The core challenge of the project is to build the infrastructure needed to process and analyze flight data, deliver a web-application that shows (historical) flights and the analyzed/enriched data in a GIS and dashboard. We learned many concepts about this project from the research phase and it helps us to determine the suited choice for our project by the implementation. We met almost daily to track the whole process. We had built models to make the prediction and a web-application to visualise fetched data. The codes were mainly tested in unit testing. Due to the time pressure, we did not complete all our requirements. We had made a few recommendations e.g. use more parameters and extend the application to other airports for future works.

Members of the project team

Name: Jari Bervoets
Interest: Machine learning, cloud computing
Contribution and role: Back-end developer, cloud developer
Name: Eric Dammeyer
Interest: Machine Learning and Gardening
Contribution and role: Data processing
Name: Tim Polderdijk
Interest: Machine learning, programming
Contribution and role: Front-end developer, SIG correspondent
Name: Boris Schrijver
Interest: Solving problems
Contribution and role: Data Processing and Machine Learning
Name: Yin Wu
Interest: Calculus and programming
Contribution and role: Front-end developer and tester
All team members contributed to preparing the report and the final project presentation.

Client

Name: Robert Koster
Affiliation: AerLabs

Coach

Name: Jan S. Rellermeyer
Affiliation: Distributed Systems

Contact

Jari Bervoets, jaribervoets@gmail.com
Eric Dammeijer, ericdammeyer@gmail.com
Tim Polderdijk, timpolderdijk@gmail.com
Boris Schrijver, boris@radialcontext.nl
Yin Wu, jiayinysj@gmail.com

The final report for this project can be found at: http://repository.tudelft.nl

C
SIG Feedback

C.1. SIG Submission

For this project, we needed to submit our code to the Software Improvement Group (SIG) 2 times. Once at the end of week 6 and once at the end of week 8. The week 8 deadline was pushed back by more than a week in the end.

C.1.1. First submission

After the first submission, the SIG gave us a maintainability score of 4.4 / 5.5 stars. This score was based on metrics: code duplication, unit size, unit complexity, unit interfacing, module coupling and component balance. In the SIG portal, we could see exactly which parts of our code had bad maintainability.

The category in which we had the lowest score was the unit size. This was because some of the methods we had contained a lot of lines. Personally, for some of the indicated methods, we did not agree with their size being too big, since we had several cases where we used many lines for a single list assignment to make it more readable. So in execution, this is only 1 line of code, but in the source file, it could be more than 10 lines. For example, we had a method where we assigned data frames multiple times in such a readable way, in 15 lines of code that equate to 1 executable line of code. This meant that the method was already 28 lines longer for the feedback report.

The second category in which we had some maintainability issues was code duplication. In two files we had about 10 lines of code that occurred twice. One important thing we had to keep in mind was that the SIG does not take .ipynb files (python notebooks) into account for their evaluation and feedback. This was unexpected for us because we had several important parts of our code in python notebooks. So, in reality, we considered that our notebooks might also have some maintainability issues. These notebooks, however, are not very likely to be updated much in the future according to our expectations.

C.1.2. Changes

Naturally, we wanted to improve on the parts of our code that were highlighted by the SIG. So we changed the files such that there was no more code duplication in them and decreased the unit size of the large methods by moving some code into newly created smaller units. Apart from this we simply had a lot more code by the time we had to make the second submission. Because of that, we expected some maintainability issues to surface in the newer code that could have been removed already if we had worked faster and already had that code for the first submission.

C.1.3. Second submission

After the second code submission, the SIG gave us a maintainability score of 3.7 / 5.5 stars. This is lower than before, and the main category that dragged this score down is code duplication. However, there is a good reason for that.

The code duplication was almost entirely over multiple files because many lines used to define things are the same across several iterations of our models. We believe this does not require any changes because they are iterations, and we will create a new iteration if we want to update the model in the future. Apart from the equivalent lines in different iterations, there are only a few duplicated lines: 20 lines total across the whole project.

The unit size has many methods that have around 25 lines, which SIG colours yellow. Yellow means there are still somewhat acceptable. There are also red methods, which have more than 50 lines. This makes it look like we wrote huge methods, however, most of these lines are actually list assignments. This is just like the first submission, SIG counts every line as a line of code even if many lines are part of definitions and do not 'do' anything.

The biggest method of all according to the SIG is 107 lines long, except in the code it is not a method, but the list assignments at the start of the file. We do not know why SIG sees this as a method as they are simply assignments like: PROJECT_ID = '<projectid>'; list1 = {...}; list2 = {...}. We think this might be a bug in the automated process of the SIG.

In the category of unit interfacing we also got a low score for some of our methods, these methods have to do with the machine learning models, and simply need many parameters. There is one method which takes 9 arguments, and the rest is at 6 or lower. For machine learning models, we do not think it is necessary to reduce the number of parameters. For unit complexity, we have two 'red' methods with high complexity. Both of which are methods in the imported/cloned java code from OpenSky Network, so we can ignore them as we did not make them. They are a state deserializer method and an equals method for state vectors. All things considered, we decided it is not necessary to refactor many of the candidates that SIG gave us. We have already explained why some methods are seen as 'red' when they are not really that bad or are not even a unit at all. With these things in mind a score of 3.7 stars is acceptable.

Bibliography

[1] Airport Collaborative Decision Making. https://www.eurocontrol.int/concept/airport-collaborative-decision-making, . Accessed: 2020-04-24.

[2] Milestone approach. https://www.skybrary.aero/index.php/Airport_Collaborative_Decision_Making_(A-CDM), . Accessed: 2020-04-24.

[3] Ads-b exchange. URL https://www.adsbexchange.com/data/#.

[4] Google bigquery. URL https://cloud.google.com/bigquery#section-1.

[5] Aws cdk. URL https://searchaws.techtarget.com/tip/Compare-AWS-CDK-vs-CloudFormation-and-the-state-of

[6] Aws cloudformation. URL https://aws.amazon.com/cloudformation/.

[7] Geomasa. URL https://www.geomesa.org/documentation/user/introduction.html#what-is-geomesa.

[8] Kafka. URL https://kafka.apache.org/intro.

[9] Keras: The python deep learning library. URL https://keras.io/.

[10] an open-source javascript library for interactive maps. URL https://leafletjs.com/index.html.

[11] Api reference, . URL https://docs.mapbox.com/mapbox-gl-js/api/.

[12] Style specification, . URL https://docs.mapbox.com/mapbox-gl-js/style-spec.

[13] Metar documentation. URL https://en.wikipedia.org/wiki/METAR.

[14] Latest. URL https://openlayers.org/.

[15] Pytorch. URL https://pytorch.org/.

[16] Separate computation and storage. URL https://medium.com/@ajstorm/separating-compute-and-storage-59def4f27d64.

[17] Tensorflow. URL https://www.tensorflow.org/.

[18] Terraform. URL https://www.terraform.io/intro/index.html.

[19] Webgl - opengl es for the web, Jul 2011. URL https://www.khronos.org/webgl/.

[20] Richard Alligier, David Gianazza, and Nicolas Durand. Machine Learning Applied to Airspeed Prediction During Climb. In *ATM seminar 2015, 11th USA/EUROPE Air Traffic Management R&D Seminar*, Lisboa, Portugal, June 2015. FAA & Eurocontrol. URL https://hal-enac.archives-ouvertes.fr/hal-01168664.

[21] Arjen de Leege, Marinus M. Van Paassen, and Max Mulder. A machine learning approach to trajectory prediction. 08 2013. ISBN 978-1-62410-224-0. doi: 10.2514/6.2013-4782.

[22] Yann Fablec and Jean-Marc Alliot. Using neural networks to predict aircraft trajectories. 05 1999.

[23] David Flanagan. *JavaScript: the definitive guide.* " O'Reilly Media, Inc.", 2006.

[24] Maxime Gariel, Ashok Srivastava, and Eric Feron. Trajectory clustering and an application to airspace monitoring. *IEEE Transactions on Intelligent Transportation Systems - TITS*, 12, 01 2010. doi: 10.1109/TITS.2011.2160628.

[25] Sungkwon Hong and Keumjin Lee. Trajectory prediction for vectored area navigation arrivals. *Journal of Aerospace Information Systems*, 12(7):490–502, 2015.

[26] Kit CJA Macleod and Richard Hewitt. Technical report: progress with developing an outcome-based web application.

[27] Ali Mesbah and Arie Van Deursen. Migrating multi-page web applications to single-page ajax interfaces. In *11th European Conference on Software Maintenance and Reengineering (CSMR'07)*, pages 181–190. IEEE, 2007.

[28] Glenn W Milligan and Martha C Cooper. A study of standardization of variables in cluster analysis. *Journal of classification*, 5(2):181–204, 1988.

[29] Kief Morris. *Infrastructure as code: managing servers in the cloud.* " O'Reilly Media, Inc.", 2016.

[30] The Open Sky Network. Open-sky network. URL `https://opensky-network.org/apidoc/index.html/`.

[31] Xavier Olive. traffic, a toolbox for processing and analysing air traffic data. *Journal of Open Source Software*, 4:1518, 2019. ISSN 2475-9066. doi: 10.21105/joss.01518.

[32] Niclas Olsson and Nicklas Ockelberg. Performance, modularity and usability, a comparison of javascript frameworks, 2020.

[33] Quentin-Ol. quentin-ol/ngx-openlayers, Apr 2018. URL `https://github.com/quentin-ol/ngx-openlayers/tree/master/documentation`.

[34] Elar Saks. Javascript frameworks: Angular vs react vs vue. 2019.

[35] V. Satopaa, J. Albrecht, D. Irwin, and B. Raghavan. Finding a "kneedle" in a haystack: Detecting knee points in system behavior. In *2011 31st International Conference on Distributed Computing Systems Workshops*, pages 166–171, 2011.

[36] Michael Schultz, Judith Rosenow, and Xavier Olive. A-cdm lite: Situation awareness and decision-making for small airports based on ads-b data. 12 2019.

[37] Matthias Schäfer, Martin Strohmeier, Vincent Lenders, Ivan Martinovic, and Matthias Wilhelm. Bringing up opensky: A large-scale ads-b sensor network for research. pages 83–94, 04 2014. doi: 10.1109/IPSN.2014.6846743.

[38] Sesar. Ads-b. URL `https://ads-b-europe.eu/`.

[39] Simplilearn. Top 7 cloud databases - revolutionizing cloud computing, Dec 2019. URL `https://www.simplilearn.com/cloud-databases-across-the-globe-article`.

[40] Junzi Sun, Huy Vû, Joost Ellerbroek, and Jacco M. Hoekstra. Weather field reconstruction using aircraft surveillance data and a novel meteo-particle model. *PLoS ONE*, 13, 2018.

[41] Kairat Tastambekov, Stéphane Puechmorel, Daniel Delahaye, and Chistophe Rabut. Aircraft trajectory forecasting using local functional regression in sobolev space. *Transportation research part C: emerging technologies*, 39:1–22, 2014.

[42] Shubhendu Trivedi, Zachary A Pardos, and Neil T Heffernan. The utility of clustering in prediction tasks. *arXiv preprint arXiv:1509.06163*, 2015.

[43] Deepali Virmani, Shweta Taneja, and Geetika Malhotra. Normalization based k means clustering algorithm. *arXiv preprint arXiv:1503.00900*, 2015.

[44] Visgl. visgl/react-map-gl, Apr 2020. URL `https://github.com/visgl/react-map-gl`.

[45] Zhengyi Wang, Man LIANG, and Daniel Delahaye. Short-term 4d trajectory prediction using machine learning methods. 11 2017.

[46] Wikipedia. MoSCoW method — Wikipedia, the free encyclopedia. `http://en.wikipedia.org/w/index.php?title=MoSCoW%20method&oldid=952152213`, 2020. [Online; accessed 29-April-2020].