# ZADÁNÍ DIPLOMOVÉ PRÁCE

## I. OSOBNÍ A STUDIJNÍ ÚDAJE

Příjmení: **Hriadelová** Jméno: **Anna Mária** Osobní číslo: **466067**

Fakulta/ústav: **Fakulta elektrotechnická**

Zadávající katedra/ústav: **Katedra počítačů**

Studijní program: **Otevřená informatika**

Specializace: **Softwarové inženýrství**

## II. ÚDAJE K DIPLOMOVÉ PRÁCI

Název diplomové práce:

**Aplikace strojového učení ve webové aplikaci pro sociální zdrženlivost**

Název diplomové práce anglicky:

**Machine Learning Pipeline inside of the Web Application for Social Distancing**

Pokyny pro vypracování:

1) Seznamte se se strukturou webové aplikace a metodami které využívá.
2) Prostudujte si softwarový stack webové aplikace.
3) Analyzujte výkon vybrané webové aplikace.
4) Upravte architekturu webové aplikace, tak aby byly jednotlivé části správně zapouzdřeny.
5) Implementujte procesy pro nasazování, testování a verzování statistických modelů ve webové aplikaci.
6) Zhodnoťte dopad naimplementovaných změn v architektuře aplikace.

Seznam doporučené literatury:

1) Krajnik, Tomas, Tomas Vintr, Sergi Molina, Jaime Pulido Fentanes, Grzegorz Cielniak, Oscar Martinez Mozos, George Broughton, and Tom Duckett. “Warped Hypertime Representations for Long-Term Autonomy of Mobile Robots.” IEEE Robotics and Automation Letters 4, no. 4 (October 2019): 3310–17. https://doi.org/10.1109/lra.2019.2926682.
2) Vural, Hulya, Murat Koyuncu, and Sinem Guney. “A Systematic Literature Review on Microservices.” In Computational Science and Its Applications – ICCSA 2017, 203–17. Springer International Publishing, 2017. https://doi.org/10.1007/978-3-319-62407-5_14.
3) Hapke, Hannes, and Catherine Nelson. "Building Machine Learning Pipelines." 2020. ISBN: 9781492053194

Jméno a pracoviště vedoucí(ho) diplomové práce:

**Ing. Zdeněk Rozsypálek, katedra počítačů FEL**

Jméno a pracoviště druhé(ho) vedoucí(ho) nebo konzultanta(ky) diplomové práce:

Datum zadání diplomové práce: **22.06.2021** Termín odevzdání diplomové práce: **04.01.2022**

Platnost zadání diplomové práce: **19.02.2023**

| | | |
|---|---|---|
| Ing. Zdeněk Rozsypálek<br>podpis vedoucí(ho) práce | podpis vedoucí(ho) ústavu/katedry | prof. Mgr. Petr Páta, Ph.D.<br>podpis děkana(ky) |

## III. PŘEVZETÍ ZADÁNÍ

Diplomantka bere na vědomí, že je povinna vypracovat diplomovou práci samostatně, bez cizí pomoci, s výjimkou poskytnutých konzultací. Seznam použité literatury, jiných pramenů a jmen konzultantů je třeba uvést v diplomové práci.

| | |
|---|---|
| . <br>Datum převzetí zadání | Podpis studentky |

Master Thesis

Czech
Technical
University
in Prague

F3

Faculty of Electrical Engineering
Department of Computer Science

# Machine Learning Pipeline inside of the Web Application for Social Distancing

Bc. Anna Mária Hriadelová

Supervisor: Ing. Zdeněk Rozsypálek
Field of study: Open Informatics
January 2022

## Acknowledgements

I would like to thank my thesis supervisor *Zdeněk Rozsypálek* for his guidance, feedback, and patience during the development of this thesis.

At last, my gratitude belongs to my family and friends for their support and care.

## Declaration

I declare that I worked out the presented thesis independently and I cited all references according to Methodical instruction about ethical principles for academic thesis writing.

In Prague, 4. January 2022

## Abstract

This thesis focuses on the analysis of the structure of the web application for social distancing, proposal, and implementation of changes in the application architecture and the integration of the Machine Learning Pipeline. During the work, the algorithm for estimating the occupancy of places from Google as the state of the art, its functionality, advantages, and disadvantages, was investigated. Within the work, the individual parts of the web application kdynakoupit.cz and the methodologies that are used in the implementation were described in detail. An important part of the research was the study of the implementation of the Machine Learning Pipeline using modern tools and techniques, from the ingesting of new data, through their validation to the evaluation of models. The practical part of the thesis deals with the implementation of this cycle of machine learning and the separation of individual parts of the application according to their responsibilities into several separate microservices. A database, which is used to store and version datasets and models, was also implemented. The last part deals with the evaluation of the implemented parts and the discussion of the achieved results.

**Keywords:** microservices, machine learning pipeline, docker, containers, social distance

**Supervisor:** Ing. Zdeněk Rozsypálek

## Abstrakt

Táto práca sa zameriava na analýzu štruktúry webovej aplikácie pre sociálnu zdržanlivosť, návrh a implementáciu zmien architektúry aplikácie a integráciu kanálu strojového účenia. Počas tejto práce bol preskúmaný algoritmus pre predpokladanie zaplnenosti miest od spoločnosti Gogle, jeho vlastnosti, výhody a nevýhody. V rámci práce boli detailne opísané jednotlivé časti webovej aplikácie kdynakoupit.cz a metodológie, ktoré sú využívané pri implementácii. Dôležitou časťou výskumu bolo študovanie implementácie kanálu strojového učenia pomocou moderných nástrojov a techník, od prijímania nových dát, cez ich validáciu po evaluáciu modelov. Praktická časť práce sa zaoberá práve implementáciou tohto cyklu strojového učenia a oddeleniu jednotlivých častí aplikácie podľa ich zodpovedností do viacerých samostatných mikroslužieb. Implementovaná bola aj databáza, ktorá slúži na ukladanie a verzovanie dátových sadov a modelov. Koniec práce sa zaoberá hodnotením implementovaných častí a diskusiou nad dosiahnutými výsledkami.

**Kľúčové slová:** mikroslužby, kanál strojového učenia, docker, kontajnerizácia, sociálna zdržanlivosť

**Preklad názvu:** Aplikovanie strojového učenia do webovej aplikácie pre sociálnu zdržanlivosť

# Contents

# Figures

# Tables

# Chapter 1

# Introduction

For the last two years, almost everyone started working or studying from home because of a worldwide pandemic. Everyday life changed from day to day, we wear face masks to protect ourselves and to protect others and we try to keep our distance from each other in public places so we don't get infected. Living a busy life and at the same time trying to keep a distance while being in a public place is very hard to achieve for most of us. A person that has a job and family does not have time to wait until the queue to the supermarket or pharmacy is gone, or an older person that his immunity is very fragile would appreciate knowing when is the best time to go, for example, shopping.

Google provides information about the busyness of the specific place so it can recommend to you when is the best time to go shopping while avoiding queues or busy places. This feature is available on Google Maps under the "popular times". The problem with calculating such predictions is that Google is using personal data such as users' location and time spent on a specific place. Not everybody is a fan of being "tracked" especially when nowadays *personal privacy* is the key for every company.

A team of scientists at the Laboratory of Chronorobotics at the Artificial Intelligence Center, CTU Prague, had the idea to create a web application that estimates the crowdedness of specific places with respect to user privacy by using the model that is built on the basis of artificial intelligence named FreMEn, which is used, for example, in long-term deployments of robots in environments. The idea was to create an application for volunteers that will be providing us the data about the crowdedness in a short questionnaire. The system can work even with a relatively small amount of data and can model and create predictions, based on probability and other cues, even for places for which it has no available data yet.

In this work, we focus on the analysis of applications structure and methodology, research of architecture of microservices and its implementation while following the rules and key concepts of that architecture style, and implementation of the machine learning pipeline into the application.

## 1.1 Outline of the Thesis

In the second chapter, we explain the purpose of the web application for social distancing kdynkoupit.cz and we describe the overall architecture and responsibilities of separate parts of the application. We also describe the state of the art, its features, advantages, and disadvantages.

In the third chapter, we dive into the methodology that we use in the implementation of the application, and we research methods for improvement. An important part of the research is studying modern technologies and tools for creating machine learning pipelines that create a workflow of automation of the model life cycle.

In the fourth chapter, we describe the proposed changes based on the analysis described in the previous chapter and the implementation of these changes.

In the fifth chapter, we discuss the impact of implemented changes and their benefits. An important part is the model evaluation by using different classification metrics, where we discuss the results we achieved.

The sixth chapter is dedicated to the proposal of future work on the application. We discuss what other improvements can be done to enhance the accuracy of the estimator and which steps can be improved in the workflow of automation in machine learning pipelines.

# Chapter 2

# Related work

Kdynakoupit.cz is a community website for sharing information about the crowdedness of shops (and other places). It can recommend you the best time to visit your favorite shop, so that you can avoid queues, crowds, and waiting outside the shop, all the while saving you time and lowering the risk of infection or collision. The idea of making such an application came when the pandemic of Coronavirus started to spread very fast and it was recommended to avoid crowded areas and places [9].

The plan was to gather necessary data for predicting how busy can be a certain shop or place in a certain time of the day. The main purpose of collecting these data is to estimate the crowdedness to help people decide when to visit a certain supermarket or retail while staying anonymous and protecting personal privacy which means that we are not acquiring data by checking the user's positions unconsciously and thus it makes our idea unique in the means of gathering the data. These data are supplied from volunteers and these are used as a dataset for creating precise statistical models that can create predictions about the crowdedness of a specific place even from a small amount of data. That's why we had to develop also an application for these volunteers so they can easily submit the data which is available on all operating systems and is called FreMEn Explorer. By sharing these predictions we are contributing to lowering the concentration of people in shops, restaurants, pharmacies, and other private or public spaces. The goal is to erase the need for people to meet simply due to a lack of information if they prefer to avoid meeting others.

## 2.1 State of the art

Google is one of the biggest corporates that use crowdedness information in specific places and thus I describe how its predictions work.

Google maps launched *popular times* and *live busyness information* which are helpful features that let you see how busy a place tends to be on a given day and time or in a specific moment which is again a very useful tool during the pandemic because it can help you with social distancing.

To calculate busyness information Google uses *Location History* which is a feature that saves the history of the places where you go and it creates for you more personalized maps based on the information gathered - this feature is by default turned off [10]. Google promises that the analyzed data are anonymous. This data is instrumental in calculating how busy a place typically is for every hour of the week. The busiest hour becomes Google's benchmark — and Google then displays busyness data for the rest of the week relative to that hour. This data can also show how long people tend to spend at certain shops, which is handy if you're planning a day with multiple activities and want to know how much time to allocate at each place [11]. Because of the Coronavirus, many business places changed their opening hours or some restriction were made, predictions may not be as reliable as before.

Google Maps also provide Live, Real-Time busyness information that can vary from its typical busyness level - again the data are provided from the Location History feature.

Google says that privacy is their top-level priority when calculating busyness information and they use *differential privacy*, which is an advanced statistical technique when working with the data to ensure anonymity. Differential privacy uses several methods, including artificially adding "noise" to the Location History dataset to generate busyness insights without ever identifying any person. And if the systems don't have enough data to provide an accurate, anonymous busyness recommendation, Google doesn't publish it [12].

*Latency Analysis System* is a Google's patent that refers to the understanding of popular times for a business [13]. This system determines a latency period, such as a wait time, at a user destination. To determine the latency period, the latency analysis system receives location history from multiple user devices. With the location histories, the latency analysis system identifies points-of-interest that users have visited and determines the amount of time the user devices were at a point-of-interest. For example, the latency analysis system determines when a user device entered and exited a point-of-interest. Based on the elapsed time between entry and exit, the latency analysis system determines how long the user device was inside the point-of-interest. By averaging elapsed times for multiple user devices, the latency analysis system determines a latency period for the point-of-interest. The latency analysis system then uses the latency period to provide latency-based recommendations to a user. For example, the latency analysis system may determine a shopping route for a user [14].

This feature provides wait times in restaurants, grocery shops, or any other merchandise. This mobile location information history is fine-grained and we can tell that our activity is processed very detailed in time.

Assuming that the system determines the occupancy level of a specific place by tracking users' devices when entering and leaving the point-of-interest, predictions don't have to be precise if the point-of-interest is located in a shopping mall.

People find these features very helpful but most of them are not aware of how depth in detail can Google track their location a how is Google using its data to create such predictions.

## 2.2 Kdy nakoupit - application for social distancing

In the subsections, I am describing the different parts of the application and their usage. The application is divided into four smaller applications that make it a whole.

In the Figure 2.1 is shown an as-is diagram that briefly describes the architecture of the application.

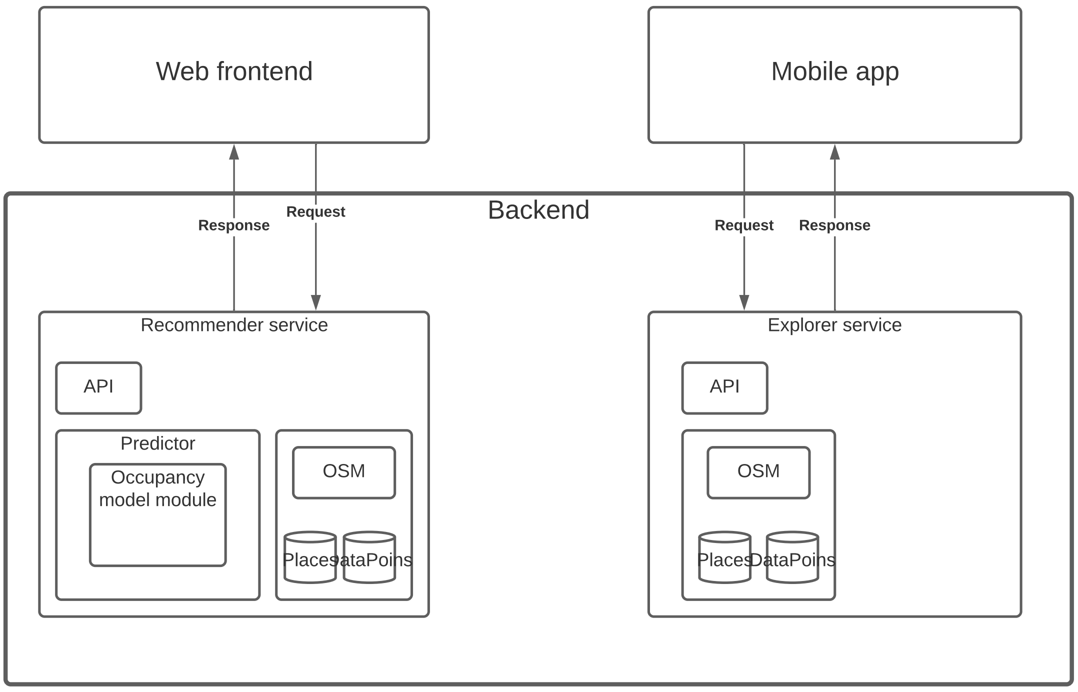
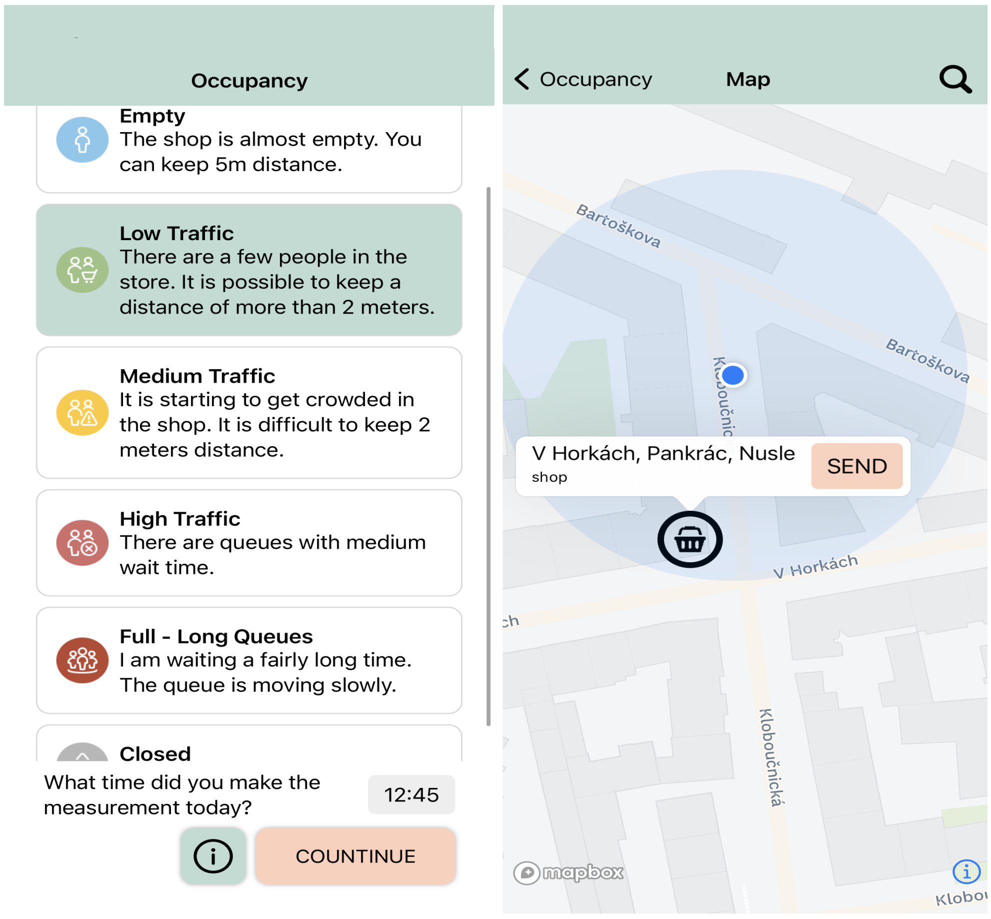


**Figure 2.1:** Application architecture

### 2.2.1 Explorer

FreMEn Explorer is an application that volunteers download and provide us the data about the busyness. The app is asking for the occupancy level, which scales from Empty (the lowest occupancy) to Full - Long Queues (the highest occupancy) and there is also the option Closed if the place is currently closed. This information is sent together with the time when providing this info. After that, the next step is to locate the place for that you provide this information. For locating the places, we are using OpenStreetMap [15] integrated into our applications. These steps are shown in the Figure 2.2.

After sending the data from the user, the backend of the Explorer receives a request that contains the data and it will save it to our measurements database (DataPoints database). These data contain crowdedness ratio (occupancy level) information, location (longitude, latitude), timezone, time of the day, the id of the place, category of the place (supermarket, park, pharmacy, etc.), hashed key, and other information. These data are then used for creating models for estimating predictions.


Occupancy

**Empty**
The shop is almost empty. You can keep 5m distance.

**Low Traffic**
There are a few people in the store. It is possible to keep a distance of more than 2 meters.

**Medium Traffic**
It is starting to get crowded in the shop. It is difficult to keep 2 meters distance.

**High Traffic**
There are queues with medium wait time.

**Full - Long Queues**
I am waiting a fairly long time. The queue is moving slowly.

**Closed**

What time did you make the measurement today? 12:45

COUNTINUE

Occupancy Map

V Horkách, Pankrác, Nusle
shop

SEND

**Figure 2.2:** Explorer iOS app

#### OpenStreetMaps

OpenStreetMap is built by a community of mappers that contribute and maintain data about roads, trails, cafés, railway stations, and much more, all over the world [15]. People are responsible for putting data to OSM which means that everyone can manage maps. On our website, we also provide instructions on how to change data in OSM in case the user cannot see some specific place he is looking for or the data provided for a place (for example opening hours or address) are incorrect.

### 2.2.2 Recommender

Another part of the backend is Recommender. Recommender is responsible for receiving requests from a user that is searching for a certain place or places in a specific area. If a user is searching for a certain place it means that the user wants to see how crowded is the place at different times of the day during the week mostly for the reason of avoiding busy hours. If a user is looking up a place in a specific area (e.g. Prague - Vinohrady) or without specifying the area, the result is points-of-interest based on the category (pharmacy, supermarket, restaurant, etc.) that the user has specified. If the user did not specify the category, by default all points-of-interest are displayed in this area. The area is bounded with a bounding box.

The searching request provides user data retrieved from the OSM database of places in this area if it finds some. Request for detailed information about a specific place provides data that the machine learning model estimated for that place.

In Figure A.2 we can see the detailed view of a specific place with a traffic graph. The use case is that a user searched for places in a specific area (Vinohrady, Prague), bounding boxed area is highlighted on the map with points-of-interest. After that, the user checked the specific place (Žabka, Rumunská) and detailed information was shown on the left side with the address of this place, opening hours (if provided), and traffic graph containing information about busyness in different timelines of the day (every 15 minutes during the day). Users can also see predictions of busyness for the next five days. This graph is generated by our machine learning model.

### 2.2.3 Occupancy model

The occupancy model is a machine learning model that is generated every day at a certain time because of constant changes in OSM mentioned in section OpenStreetMaps. To generate an occupancy model, we require sufficient measurements provided, to be model precise. If that condition is met, dataframe is created and saved into a dataframe comma-separated value (CSV) file as a dataset for our model.

Dataset is made of several crucial features that our model uses when creating predictions. Several features that we care about are: coordinates, opening time, city, city population, highway (whether the place is located nearby highway), parking (does the place have a parking lot?), living (is it located in the living area?), public transport (is there available public transport nearby?) and category (supermarket, pharmacy, park, etc.). These features are so far very important in creating our model but we can still improve the result of prediction by adding new features to the dataset (such as place area). We also include information gathered from volunteers to the dataset, such as crowded ratio and time of the measurement.

The more diverse measurement data we gather from users, the better predictions we can provide for different places. For example, if users send us measurements for the small local supermarket located in the living area, without public transport and parking lot nearby we can apply this measurement to similar small local supermarkets in the same or even different cities based on their area size. Another example is a supermarket that belongs to a well-known supermarket chain with a variety of products and items, located near the highway, with a parking lot and in the living area, based on the measurements provided for this place we can predict that this data can be applied also for other supermarkets with this features.

The more detailed information about how the model works is described in the next section.

## FreMEn

The method used in the model is built on the basis of artificial intelligence named FreMEn = Frequency Map Enhancement, which is used, for example, in long-term deployments of robots in environments that change under the influence of human action. The goal of using this artificial intelligence is to ensure the inclusion of robots into human-inhabited areas ( such as offices, hospitals, etc.) that is as non-invasive as possible. This method can be applied even beyond the domain of robotics. FreMEn can learn, understand, and evaluate, in which hours and at which places there is a heightened number of people in a single area, i.e. of potential spreaders of the virus which is the case we use in our application [16].

FreMEn is based on an assumption that from a mid-to long-term perspective, some of the environment dynamics are periodic such as crowdedness of the place during the day (e.g. local supermarket has few people in the morning before people go to work, it is almost empty before noon, medium traffic during the lunch time and highest traffic is between 4pm - 6pm when people leave work). To reflect that, FreMEn models the uncertainty of elementary environment states by a combination of periodic functions rather than by a constant probability. Modeling the uncertainties as probabilistic functions of time allows the integration of long-term observations of the same environment into memory-efficient Spatio-temporal models. These models can predict the future environment states with a given level of confidence [17].

The concept is based on the idea of frequency transforms, which represent functions of time by the frequencies that make them up. FreMen simply takes a given sequence of long-term observations of a particular environment state, calculates its frequency spectra by the Fourier transform, and stores the most prominent spectral components. These components correspond to the observed periodicities of the given environment state. Knowledge of the spectral components allows calculating the probability of the environment state for a given time.

In Figure 2.3 we can see the method overview. The data points (a,t) observed over time (top, black) are first processed by frequency analysis FreMEn to determine a dominant periodicity $\boldsymbol{T}$. Then, the time $\boldsymbol{t}$ is projected onto a 2d space (called hypertime) and the vectors $\boldsymbol{(a,t)}$ become $\boldsymbol{(a,cos(2t/T),sin(2t/T))}$ (bottom, left). The projected data are then clustered (bottom, center, blue) to estimate the distribution of a over the hypertime space (green). Projection of the distribution back to the uni-dimensional time domain allows to calculate the probabilistic distribution of $\boldsymbol{a}$ for any past or future time.

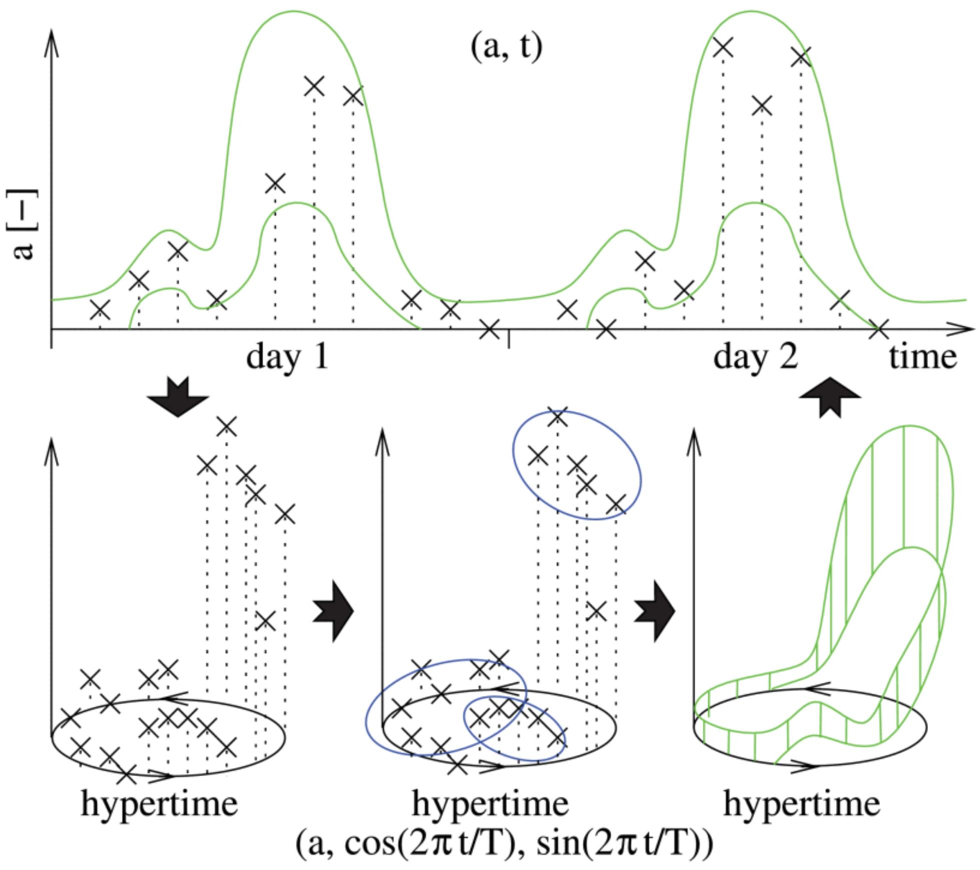


**Figure 2.3:** Method overview [1]

For our project to successfully commence and gain universal usability, we need a large number of volunteers who will be willing to supply data about the numbers of people in certain areas at particular times. This is the only way in which we can supply FreMEn with sufficient information to create a continuous time-space map as I mentioned in the example in the previous section.

## 2.3 Summary

In this chapter, I described the *popular times* feature that is implemented in Google Maps as state of the art. Popular times or live busyness information lets you see how busy a place tends to be on a given day and time. Google uses Latency Analysis System that determines a latency period, such as a wait time, at a user destination. For example, it determines when a user device entered and exited a point-of-interest. Based on the elapsed time between entry and exit, the latency analysis system determines how long the user device was inside the point-of-interest and by averaging elapsed times for multiple user devices, it determines a latency period for the point-of-interest. Providing these data, it can create a "traffic graph" for specific places on Google Maps.

Furthermore, I described different parts and their responsibilities of kdynakoupit.cz application. It’s an application for social distancing that predicts how busy can a specific place be at different times of the day, but with a respect to user privacy. The measurement data are collected from volunteers that provide us the information about the location of the place, time of the measurement, and crowded ratio information (such as empty, low, medium, or high traffic) through the separate mobile application. These data are used as a dataset for creating precise statistical models that are able to create predictions about the crowdedness of a specific place even from a small amount of data.

# Chapter 3

# Methodology

In this thesis, my goal is to investigate points for improvement in the backend of application for social distancing. In this chapter, I am describing methods that we currently use in the project and research methods that could be a potential improvement. In the first part, I describe the differences between microservices and monolithic architectures, their pros and cons and I explain the reason why we have chosen microservices architecture. In the next part of this chapter, I describe how we build and run our application, what technology we use and what other technologies are useful in deploying and developing microservice applications. The last part is about Machine Learning Pipelines, a method for creating processes to accelerate, reuse, manage, and deploy machine learning models.

## 3.1 Microservices vs. Monolith

In our application for social distancing, we use the microservices architecture as I described in the previous chapter. The reason for choosing this architecture style is because the project is complex and it has to provide us a possibility to scale because of the possibility of adding new capabilities and functionalities. Building such a complex application is not trivial at all. It requires a team that has some experience in containerization, DevOps, domain modeling, etc. I have chosen monolithic architecture for comparison so I can prove the reason for using microservices architecture in our project.

### 3.1.1 Monolithic architecture

A monolithic architecture is one in which all the components of a given application are located together in one unit. This drive is usually limited to one instance of the application's runtime. Traditional applications often consist of a web interface, a service layer, and a data layer. In a monolithic architecture, these layers are combined in an instance of the application (see Figure 3.1) [2].

The most common example that comes to mind when discussing monoliths is a system in which all of the code is deployed as a single process. We can have multiple instances of this process for robustness or scaling reasons, but fundamentally all the code is packed into a single process. In reality, these single process systems can be simple distributed systems in their own right because they nearly always end up reading data from or storing data into a database, or presenting information to web or mobile applications.

Modular monolith, on the other hand, is a subset of a single process monolith in which a single process consists of separate modules. Each module can be worked on independently, but all still need to be combined for deployment [3].

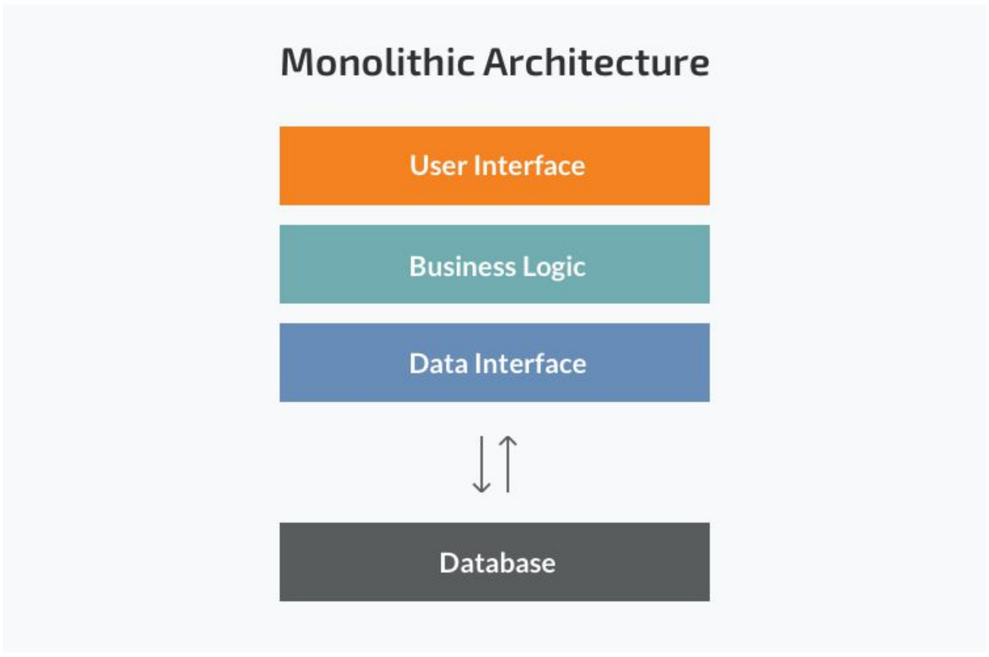


**Figure 3.1:** Monolithic architecture [2]

## Advantages

1. **Easy to deploy** - because of deploying just one application, deployment should not be a difficult process.

2. **Simple to develop** - developing a monolithic application requires basic knowledge of developing any information system, code can be reused

within the monolith itself and it is easy to comprehend.

3. **Simple to manage** - we have to take care of only one project.
4. **Debugging and testing** - indivisible unit, running end-to-end testing or debugging is much faster as well as monitoring, troubleshooting is greatly simplified.

#### Disadvantages

1. **Language lock** - once we choose a certain language and framework for an application that is the best at that time, it will be hard to switch or change the whole implementation to a newer version or completely new technology.
2. **Difficult to digest** - as the project grows, the complexity grows as well and understanding of the codebase becomes hard to digest.
3. **Scaling** - difficult to scale services independently.
4. **Developement of new feature** - complexity of development and deployment management as the code base grows, which slows down releases and the implementation of new features.

### 3.1.2 Microservices architecture

The following section is based on the book Building microservices [3].

Microservices have become an increasingly popular architecture choice in the half-decade. Microservices are independently releasable services that are modeled around a business domain. A service encapsulates functionality and makes it accessible to other services via networks. One microservice might represent an inventory, another order management, and yet another shipping, but together they might constitute an entire system. Microservices are an architecture choice that is focused on giving you many options for solving the problems you might face.

From the outside, a single microservice is treated as a black box. It hosts business functionality on one or more network endpoints, over whatever protocols are most appropriate. Consumers, whether they're other microservices or other sorts of programs, access this functionality via these networked endpoints. Internal implementation details (such as the technology the service is written in or the way data is stored) are entirely hidden from the outside world. This means microservice architectures avoid the use of shared databases in most circumstances; instead, each microservice encapsulates its own database where required.

Microservices embrace the concept of information hiding. Information hiding means hiding as much information as possible inside a component and exposing as little as possible via external interfaces. This allows for a clear separation between what can change easily and what is more difficult to change.

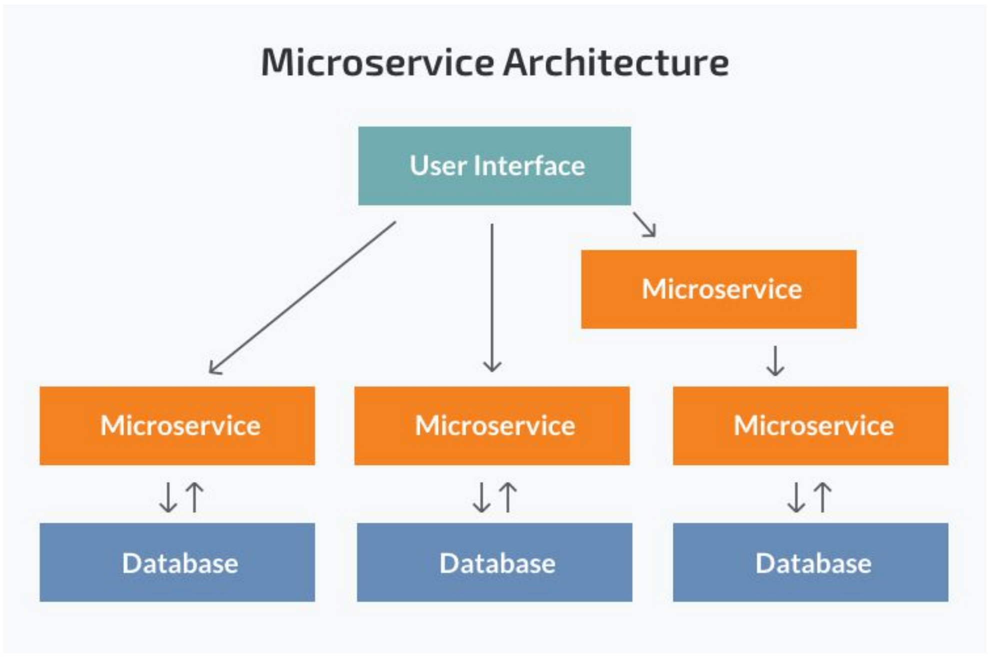


**Figure 3.2:** Microservices architecture [2]

## Key concepts

1. **Independent Deployability** - every service is deployed on its own, which means that when a change is made to a certain service, only this service is deployed without the need of deploying the whole application. To ensure this concept, microservices should be *loosely coupled*. We should be able to change one service without having to change anything else.

2. **Modeled around a Business domain** - the idea is to define service boundaries. By modeling services around business domains, we can make it easier to roll out new functionality.

3. **Flexibility** - because microservices are deployed separately, they have easier error correction management and feature issuance. We can update the service without redeploying the entire application, and if something goes wrong, roll back the update.

4. **Owning Their Own State** - it is very common in microservices that each service has its own database. If one service needs data from another service database, this service can "ask" for this data. This gives the microservices the ability to decide what is shared and what is hidden.

## Advantages

1. **Technology Heterogenity** - microservices system is composed of multiple services and we can decide to use different technologies inside each one. That gives us freedom in choosing the right tool for each service instead of selecting one standardized all-in approach. We are also able to choose the right data storage depending on what kind of data we need to store in the database. In Figure 3.3 we can see different parts in a social network system. Posts that users make can be stored in a document-oriented data store, pictures could be stored in Blob storage and for example, users interactions in a graph-oriented database.

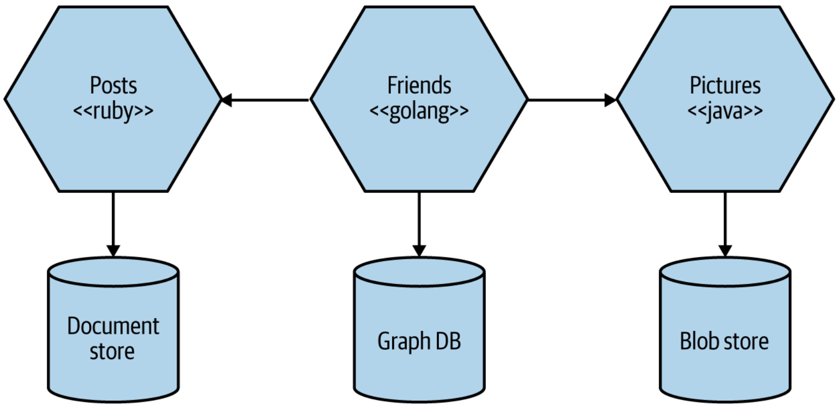


**Figure 3.3:** Different data storage based on the requirement [3]

2. **Robustness** - a component of a system may fail, but as long as that failure doesn't cascade, we can isolate the problem, and the rest of the system can carry on working. This is a huge advantage compared to monolithic service, where if a failure occurs, everything stops working. With microservices, we can build systems that handle the total failure of some of the services and degrade functionality accordingly. It is important to know how to handle such failures and the impact those failures will have on the end-users of the software.

3. **Scaling** - with smaller services, we are able to scale these services that need scaling, and that allows us to run other parts of the system on smaller, less powerful hardware.

4. **Ease deployment** - with microservices, we can make a change to a single service and deploy it independently of the rest of the system. If

a problem occurs, it can be quickly isolated to an individual service, making fast rollback easy to achieve. It also means that we can get the new functionality quicker to users. This is one of the main reasons organizations like Amazon or Netflix use these architectures.

5. **Team development** - because the team of developers is distributed depending on the service that is being implemented, we can extract problems like large codebase and less productivity due to waiting for others to finish their job. Microservices allow us to better align the architecture to the organization or team.

There is one of these advantages that I want to describe more - **scaling**. Scaling is very important in every system, the reason is that it allows us to improve the performance of the system, for example by handling more load or by improving latency. There are four different types of scaling based on the Scale Cube from The Art of Scalability. The first one is *vertical scaling* which means improving the hardware by faster CPU, better I/O for improving latency, etc. *Horizontal duplication* on the other hand, means having multiple things capable of doing the same work. We can duplicate parts (e.g., by creating replicas) of the system to handle more workloads. Next, *data partitioning* is diving the work based on some attributes of the data. That can be for example diving users into two different databases based on their surname - by that, we can distribute the load. The last type is *functional decomposition*, which is a separation of work based on the type, e.g., microservice decomposition. By that, we can extract existing functionality from an existing system and create a new microservice.

### Disadvantages

Microservices architecture, despite all of the benefits I named in the previous section, has a lot of pain points. Building such a system can become a very complex task that requires some level of seniority and skills from developers.

1. **Technology Overload** - each service in the system can be implemented in a different framework, language and can be using different tools or technologies. That can lead to technology overload. It is important to carefully balance the breadth and complexity of the technology we use against the costs.

2. **Cost** - the more processes, the more computers, networks, storage, and supporting software are needed the more costs. Not only does hardware and software components cost money but also learning new technologies can have an impact on the costs because of man-hours spent on this.

3. **Reporting** - compared to monolithic architecture, where all the data are stored in one primary database or its replica, for microservices it can get more difficult to get reports across all the data because of scattering the data across multiple isolated schemas.

4. **Monitoring and Troubleshooting** - it is more difficult to monitor several applications at the same time instead of monitoring just one.

5. **Security** - because more information flows over the network between services, data are becoming more vulnerable. These data could be potentially part of man-in-the-middle attacks. We have to be sure that only authorized parties are able to reach the endpoints.

6. **Testing** - the scope of testing end-to-end tests becomes very large. We have to run tests across multiple processes, all of which need to be deployed and configured for the test scenarios.

7. **Latency** - information that flows within a single process now needs to be serialized, transmitted, and deserialized over the network. All of this can result in worsening latency of the system.

8. **Data consistency** - as I mentioned before data in microservices are scattered across the database, which can cause a problem in data consistency. The use of distributed transactions in most cases proves to be highly problematic in coordinating state changes.

Nowadays, microservices are often associated with containerization. Containerization offers developers more freedom e.g., each component encapsulated in a separate container, may have different dependencies and libraries, without affecting other parts of the application. In addition, containerization and microservice architecture have the same purpose - transfer of classic heavy application into light, flexible, scalable, and easy to operate. I am describing containerization more accurately in the next section.

## 3.2 Containerization and container orchestration

In this section, I describe how we have deployed our application, whether in a production or development environment. In the first part, I describe the ways of deploying microservices applications, and the reason why we chose containerization. In the next part, I write about Docker and docker compose technology, which we use in our application. In the last part, I will introduce Kubernetes technology, which is becoming part of almost every production microservice application, and its benefits, why we should use it.

### 3.2.1 History

The first concept of application containers was created in 1979 and used in Unix version 7. It was a chroot command that was used to isolate the process by changing its root directory on the file system. Further progress in virtualization and containerization was made as early as 2000 as Free BSD Jails was developed, a concept that is quite similar to the current Docker containers. Jails provide more isolation than chroot, in that they virtualize not only the file system but also the set of users and the network. The same as the predecessor of Unix, Jail allows you to change the root directory for a process. In addition, they provide the ability to choose a different IP address and for each container hostname, set your own set of users and their rights. In 2001 a similar technology has emerged for Linux users, called LinuxVServer. Linux Containers technologies have contributed to the further development of containerization in 2008 and Let Me Contain That For You (LMCTFY) in 2013 by Google. These two platforms were the basis for the present Docker, which was founded in 2013. Originally, Docker used the Linux Containers platform, then switched to its own library, libcontainers, which is based on LMCTFY. Just after the founding of Docker, containerization gained its greatest popularity [18].

There are big differences between classical architecture, virtualization, and containerization. An application developed for deployment on a traditional server architecture is dependent on the target computer's operating system. This fact means migration on another server with a different OS forces changes in the application.

Virtualization, whether using virtual machines or containers, addresses this problem, as well as a large number of others. In the case of a VM, above the OS layer, we find the hypervisor, the purpose of which is to emulate real HW for use in virtual machines. Then each VM appears to be a real computer with its own OS. VMs needs an operating system that takes up space and resources of the physical computer. To solve this problem we can use containers that have a slightly different approach to virtualization. Containers use a virtualized OS and have only the application with its libraries and dependencies, so containers are light weighted and more flexible. The main purpose of containerization is for applications to be written once and run anywhere [19]. Containers are an abstraction at the app layer that packages code and dependencies together. Multiple containers can run on the same machine and share the OS kernel with other containers, each running as isolated processes in userspace. Containers take up less space than VMs, can handle more applications, and require fewer VMs and Operating systems [4].

Using containers we solve the following problems:

1. Application transfer between different environments and operating sys-

tems.

2. Speed and efficiency compared to virtualization.
3. Application and host isolation.

On the other hand, it brings up the following disadvantages:

1. Docker and other container platforms are relatively new technologies, they can face security issues.
2. Storing data in a containerized application is much more complicated. Containers are designed so that when they are turned off, the data stored in the container itself loses. For this reason, storage is a common practice data outside the container, which in itself violates the principle of isolation and is a security hole.
3. Containers are slower compared to the classic server architecture due to host system interconnection overheads and containers.

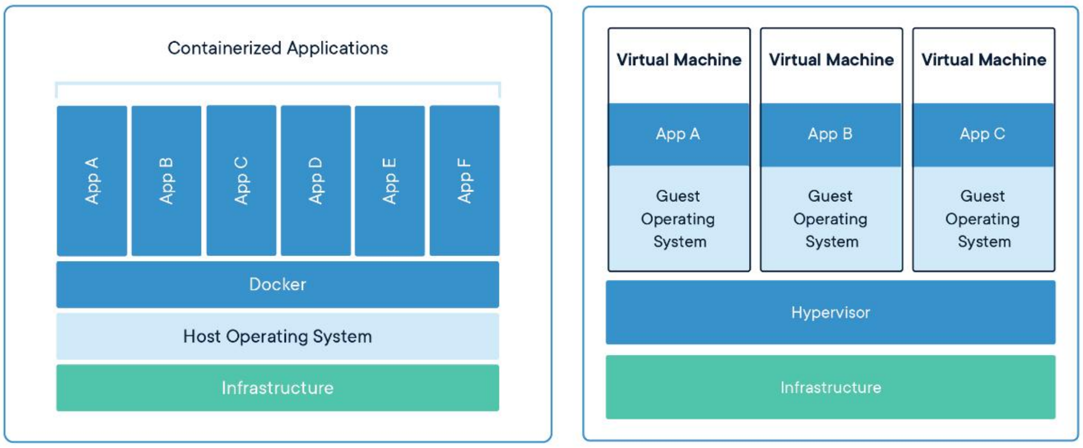


**Figure 3.4:** Comparison of virtualization and containerization [4]

Last but not least part of containerization is container orchestration. Container orchestration is the automation of much of the operational effort required to run containerized workloads and services. This includes a wide range of things software teams need to manage a container's lifecycle, including provisioning, deployment, scaling, networking, load balancing and more [20].

### 3.2.2 Docker

This section is based on Docker documentation [5].

Docker is an open platform for developing, shipping, and running applications. Docker enables us to separate applications from the infrastructure so we are able to deliver software quickly. With Docker, infrastructure can be managed in the same ways as we manage our applications. By taking advantage of Docker's methodologies for shipping, testing, and deploying code quickly, it is possible to significantly reduce the delay between writing code and running it in production. Docker provides the ability to package and run an application in a loosely isolated environment - *container*. Before I go further with the description of Docker's abilities, I describe docker architecture in short.

Docker uses a client-server architecture. The Docker *client* talks to the Docker *daemon*, which does the heavy lifting of building, running, and distributing Docker containers. The Docker client and daemon can run on the same system, or a Docker client can be connected to a remote Docker daemon. The Docker client and daemon communicate using a REST API, over UNIX sockets or a network interface. Another Docker client is *Docker Compose*, which lets us work with applications consisting of a set of containers.

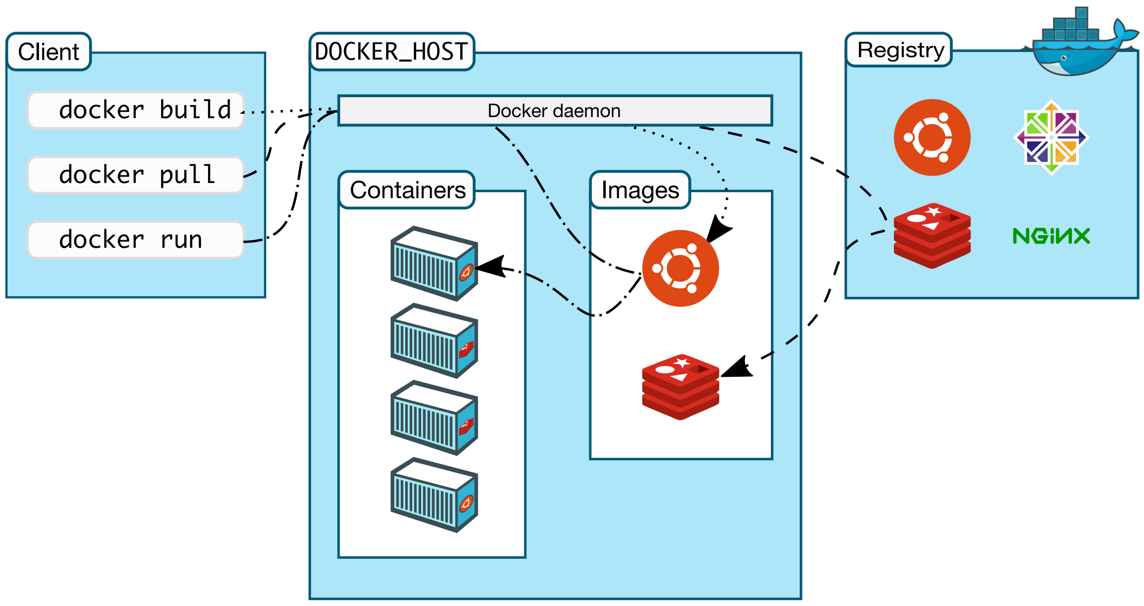


**Figure 3.5:** Docker architecture [5]

**Docker deamon** or `dockerd`, listens for Docker API requests and manages Docker objects such as containers, images, volumes, and networks. It is also able to communicate with other daemons to manage Docker services.

**Docker client** is the way that Docker users interact with Docker. It provides us commands such as `docker run`, the client sends these commands to `dockerd`. The `docker` command uses the Docker API.

**Docker registries** store Docker images. Docker Hub is a public registry and Docker is configured to look for images on Docker Hub by default. Commands such as `docker pull` or `docker run` pull required images from registry.

**Docker objects** are objects such as image, container, volume, network, or plugin. The most important ones are image and container.

An **image** is a read-only template with instructions for creating a Docker container. Images can be created by ourselves or we can use already existing ones, that are published in the registry by others. To build our own image, the *Dockerfile* must be created first. Dockerfile consists of a set of instructions, that have to be made in order to create an image and run it. Each instruction creates a layer in the image and when we change the Dockerfile, only the layers which have changed are rebuilt - that makes images lightweight, small and fast when compared to other virtualization technologies.

A **container** is runnable instance of an image. We can create, start, stop, move or delete a container using the Docker API or CLI. The container has the ability to connect to one or more networks and it can have its own storage. We are able to control how isolated a container's network, storage, or other subsystems are from other containers or from the host machine. A container is defined by its image as well as any configuration options we provide to it when we create or start it. When a container is removed, any changes to its state are not stored in persistent storage disappear. Containers are lightweight and contain everything needed to run the application, so we do not need to rely on what is currently installed on the host. We are also able to share containers while we work and be sure that shared containers work in the same way.

Docker provides tooling and a platform to manage the lifecycle of containers e.g. developing applications and their components using containers, distributing and testing the application as a unit, deploying into the production environment as a container, or an orchestrated service. Because Docker is a container-based platform it allows highly portable workloads. Containers can run on local computers or laptops, physical or virtual machines in data centers, cloud providers, or mixed environments. Docker also makes it easier to dynamically manage workloads, scaling up or down applications and services in near real-time. Docker is a cost-effective alternative to hypervisor-based virtual machines, so we can use more computer compute capacity to achieve certain business goals. It is suitable for small, medium deployments as well as for high-density environments.

### Docker Compose

Docker Compose is a tool for defining and running multi-container Docker applications. With Compose and YAML files we can configure the application's services. YAML is a human-readable data-serialization language. It is commonly used for configuration files and in applications where data is being stored or transmitted [21]. With a single command, we create and start all the services from the configuration. Compose works in production, development, testing, and staging environment. When we are developing an application, the ability to run this application in an isolated environment and interact with it is important. The Compose file provides a way to document and configure all of the application's service dependencies such as database, caches, web service APIs. Using the Compose command-line tool we are able to start more containers for each dependency with a single command `docker-compose up`.

The disadvantage of Compose is that it was originally focused on the development and testing workflows, but with new releases, Compose has gained more production-oriented features. Docker Compose can be used only on single host deployments.

Another feature of Compose is that it preserves volume data when containers are created. Volumes are the way of storing container data. Volumes are stored in the part of the host filesystem which is managed by Docker. Compose preserves all volumes used by the services. When `docker-compose up` runs, if it finds any containers from previous runs, it copies the volumes from the old container to the new container - this process ensures that the data won't get lost. Compose also caches the configuration used to create a container. When we restart a service that has not changed, Compose reuse the existing containers [22].

For using Docker Compose following steps are required:

1. Define app's environment with a Dockerfile so it can be reproduced anywhere.
2. Define the services that make up the app in docker-compose.yml so they can be run together in an isolated environment.
3. Run docker compose up command for starting all the services.

Currently, in our application, we defined two docker compose files. One is for the development environment, the other one is for the production environment.

The following fragment shows one service called *recom* defined in our docker-compose file.

```
recom:
    build:
        context: ./Recommender
        dockerfile: ./Dockerfile
    image: recom:latest
    container_name: recom
    restart: unless-stopped
    ports:
      - "8001:8000"
    volumes:
      - ./Recommender:/src
      - ./models:/src/predictor/occupancy_model/models
    env_file:
      - .env.recom
      - .env.osm
      - .env.db_common
    networks:
      - backend-dev
    depends_on:
      - db_measure
      - db_place
    command: bash -c  '/src/setup.sh'
```

Recom service uses an image that's built from `Dockerfile` specified in the context. On the next line, we specify the image name, container name, and restart policy. Then it binds the container and the host machine to the exposed ports. Next, we define volumes where container data will be stored. We also define environment files that are needed for our service. Then the network container is specified. Depends_on express dependency between services, recom service depends on measure and place databases. Lastly, we define the command that runs the service.

### 3.2.3 Kubernetes

The previous section was devoted to the description of containerization and its benefits. Other important parts of the application's lifecycle are maintenance, support, and development. Over time, the application tends to expand with new functionalities, and with increasing popularity, more users are using the application and at the same time must provide reasonable response speed and service availability. Application load is solved by scaling, either vertical or horizontal. Vertical scalability is achieved by improving the hardware on which the application runs. Horizontal scaling is about adding system elements by creating replicas or splitting data. Vertical scalability is more limited but easier to maintain. Horizontal scalability on the other hand has problems associated with load balancing between replicas. The external service that communicates with the replicated element must know which address to send the query to. In this case, the address means the IP address, which uses can cause another problem. When deploying the application to another environment it is not ensured that the network of IP addresses will remain the same, which means that at any time the application must be intercepted during the migration.

After solving the above problems, another problem arises, such as defying the behavior of the system in case of failure of one of the replicated units or generally some services. In this case, the previous algorithm for load balancing must rebuild the entire network and at the same time, the service must be restored. Another problem can be deploying a new version of the service, especially for critical systems.

A container orchestrator is a tool that solves the above problems. It provides the user the ability to configure system behavior in the above cases and automates container management by configuration. Besides that, it provides a lot of other useful features which I describe further in this section.

**Kubernetes**, also known as K8s, is a portable, extensible, open-source platform for managing containerized workloads and services, that facilitates both declarative configuration and automation. It has a large, rapidly growing ecosystem. Kubernetes services, support, and tools are widely available. Kubernetes was first developed by engineers at Google before being open-sourced in 2014. It is a descendant of Borg, a container orchestration platform used internally at Google [23].

Kubernetes provides a framework to run distributed systems resiliently. It takes care of scaling and failover for your application, provides deployment patterns, and more.

Besides that, Kubernetes schedules and automates container-related tasks throughout the application lifecycle, including [23]:

1. Deployment - Deploy a specified number of containers to a specified host and keep them running in the desired state.
2. Service discovery and load balancing - Kubernetes can automatically expose a container to the internet or to other containers using a DNS name or IP address. If traffic to a container is high, Kubernetes is able

to load balance and distribute the network traffic so that the deployment is stable.

3. Rollouts - A rollout is a change to a deployment. Kubernetes lets us initiate, pause, resume, or roll back rollouts.

4. Storage orchestration - Kubernetes allows us to automatically mount a storage system of your choice, such as local storage, public cloud providers, and more.

5. Self-healing - When a container fails, Kubernetes can restart or replace it automatically to prevent downtime. It can also take down containers that don't meet your health-check requirements.

6. Secret and configuration management - Kubernetes lets us store and manage sensitive information, such as passwords, OAuth tokens, and SSH keys. We can deploy and update secrets and application configuration without rebuilding your container images, and without exposing secrets in your stack configuration.

To understand Kubernetes better, it is good practice to know its architecture.

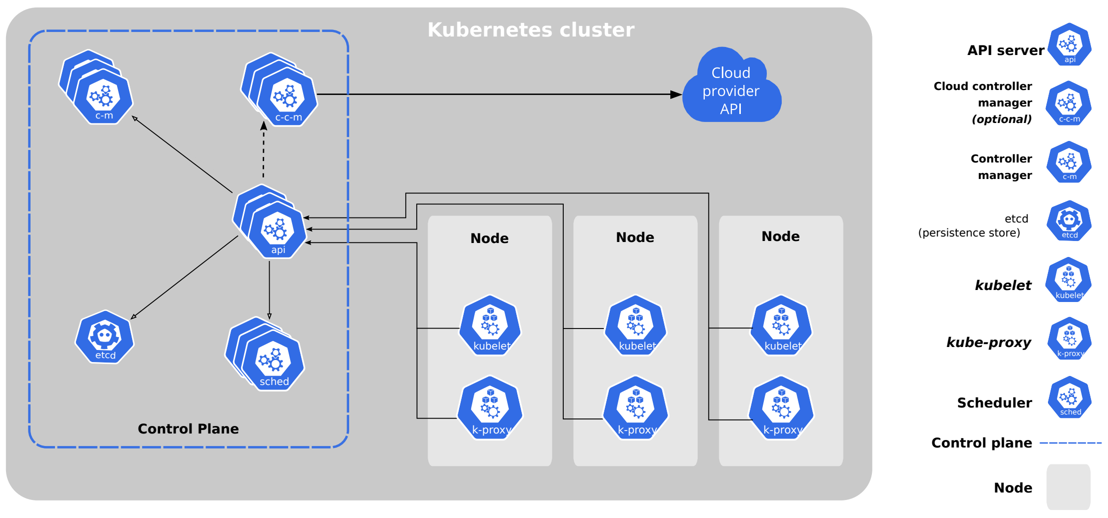


**Figure 3.6:** The components of a Kubernetes cluster [6]

1. **Cluster** - set of worker machines, called nodes, that run containerized applications. Every cluster has at least one worker node.

2. **Container** - a lightweight and portable executable image that contains software and all of its dependencies.

3. **Node** - a worker machine in Kubernetes.

4. **Pod** - the smallest and simplest Kubernetes object. A Pod represents a set of running containers on your cluster.

5. **Control plane** - the container orchestration layer that exposes the API and interfaces to define, deploy and manage the lifecycle of containers.

6. **Service** - an abstract way to expose an application running on a set of Pods as a network service [24].

A cluster consists of a set of nodes, in which pods are placed. In each node, more than one pod can be placed and in each pod, more containers can be placed. Furthermore, the control plane manages the worker nodes and the pods in the cluster. In production environments, the control plane usually runs across multiple computers and a cluster usually runs multiple nodes, providing fault-tolerance and high availability.

## Control Plane components

The control plane's components make global decisions about the cluster such as scheduling, as well as detecting and responding to cluster events.

**Kube-apiserver** provides a REST API that allows users, parts of the cluster, and external systems to communicate with each other. The interface is documented with OpenAPI. JSON or Protobuf is used for communication. Another way to use the apiserver is `kubectl`, which allows calling the K8s interface via the command line.

Kubernetes backing store for all cluster data is stored in **etcd**, which is a key-value store for distributed environments.

Another important part of the control unit is the scheduler. It assigns new pods to nodes that correspond with the conditions of deployment. The default scheduler for K8s is a **kube-scheduler**.

**Kube-controller-manager** is a control plane component that runs controller processes. In Kubernetes, controllers are control loops that watch the state of your cluster, then make or request changes where needed. Each controller tries to move the current cluster state closer to the desired state. Logically, each controller is a separate process, but to reduce complexity, they are all compiled into a single binary and run in a single process.

Some types of these controllers are:

1. Node controller - responsible for noticing and responding when nodes go down.

2. Endpoints controller - populates the Endpoints object (that is, joins Services and Pods).
3. Job controller - watches for Job objects that represent one-off tasks, then creates Pods to run those tasks to completion.
4. Service Account and Token controllers - create default accounts and API access tokens for new namespaces.

A similar unit is the **cloud-controller-manager**. It is also a controller manager, but unlike kube-control-manager it manages controllers, which are specific to a given cloud platform. It is used for K8s integration to cloud platforms (such as AWS or Google Cloud).

#### Node components

Node components run on every node, maintaining running pods and providing the Kubernetes runtime environment.

**Kubelet** is an agent that runs on each node in the cluster. It makes sure that containers are running in a Pod. This element reads the configuration files, defined in either YAML or JSON format and runs containers accordingly, and checks their health. When the configuration file is deleted from the monitored folder, kubelet stops the container.

**Kube-proxy** is a network proxy that runs on each node in a cluster, implementing part of the Kubernetes Service concept. It maintains network rules on nodes. These network rules allow network communication to Pods from network sessions inside or outside of the cluster. Kube-proxy uses the operating system packet filtering layer if there is one and it's available. Otherwise, kube-proxy forwards the traffic itself.

## 3.3 Machine Learning Pipelines

The following section is based on the book Building Machine Learning Pipelines [7].

Machine learning pipelines implement and formalize processes to accelerate, reuse, manage, and deploy machine learning models. Software engineering went through the same changes a decade or so ago with the introduction of continuous integration (CI) and continuous deployment (CD). Back in the day, it was a lengthy process to test and deploy a web app. These days, these processes have been greatly simplified by a few tools and concepts.

Previously, the deployment of web apps required collaboration between a DevOps engineer and a software developer. Today, the app can be tested and deployed reliably in a matter of minutes. Data scientists and machine learning engineers can learn a lot about workflows from software engineering.

Pipelines also become more important as a machine learning project grows. If the dataset or resource requirements are large, the approaches we discuss allow for easy infrastructure scaling. If repeatability is important, this is provided through the automation and the audit trail of machine learning pipelines.

A machine learning pipeline starts with the ingestion of new training data and ends with receiving some kind of feedback on how a newly trained model is performing. It includes a couple of steps such as data ingestion, data validation and preprocessing, model training and analysis, evaluation, and lastly model deployment. This pipeline is described as a model life cycle shown in Figure 3.7. A very important part of the machine learning pipeline is its automation. These steps could be manually very error-prone and the solution is automation with a variety of tools. One of the tools is TensorFlow Extended, which is a library that supplies all of the components that are needed for building a machine learning pipeline.

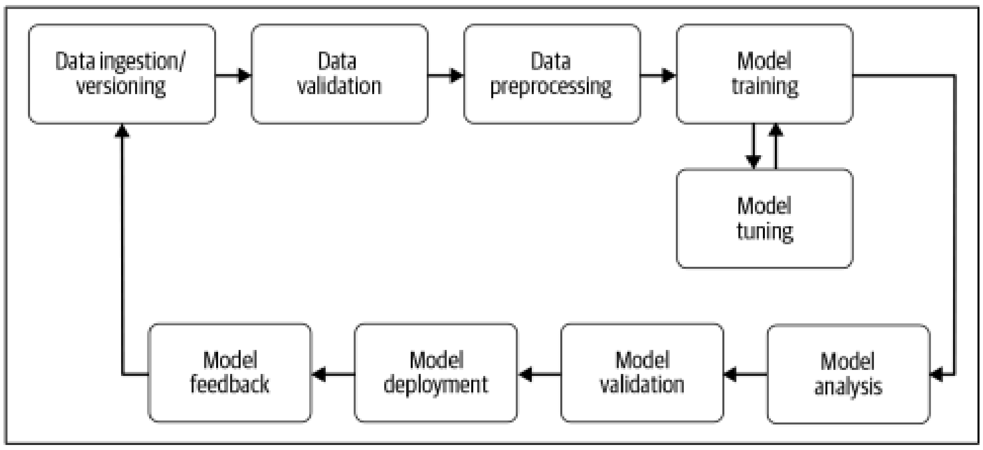


**Figure 3.7:** Model Life Cycle [7]

### 3.3.1 Data ingestion and versioning

In this step, the data are processed into a format that the next component can "digest". This is the point where we also version the incoming data. In this step of our pipeline, we read data files or request the data for our pipeline run from an external service. Before passing the ingested dataset to the next component, we divide the available data into separate datasets (e.g., training and validation datasets) and then convert the datasets into TFRecord files containing the data represented as tf.Example data structures. TFRecord is a lightweight format optimized for streaming large datasets.

### 3.3.2 Data Validation

Before training a new model version, we need to validate the new data. Data validation focuses on checking that the statistics of the new data are as expected (e.g., the range, number of categories, and distribution of categories). It also alerts the data scientist if any anomalies are detected. For example, if we are training a binary classification model, our training data could contain 50% of Class X samples and 50% of Class Y samples. Data validation tools provide alerts if the split between these classes changes, where perhaps the newly collected data is split 70/30 between the two classes. If a model is being trained with such an imbalanced training set and the data scientist hasn't adjusted the model's loss function, or over/under sampled category X or Y, the model predictions could be biased toward the dominant category.

### 3.3.3 Data Preprocessing

It is highly likely that we cannot use our freshly collected data and train the machine learning model directly. In almost all cases, we will need to preprocess the data to use it for our training runs. Labels often need to be converted to one or multi-hot vectors. The same applies to the model inputs. If we train a model from text data, we want to convert the characters of the text to indices or the text tokens to word vectors. Since preprocessing is only required prior to model training and not with every training epoch, it makes the most sense to run the preprocessing in its own life cycle step before training the model.

### 3.3.4 Model Training

The model training step is the core of the machine learning pipeline. In this step, we train a model to take inputs and predict an output with the lowest error possible. With larger models, and especially with large training sets, this step can quickly become difficult to manage. Since memory is generally a finite resource for our computations, the efficient distribution of the model training is crucial.

### 3.3.5 Model analysis

Generally, we would use accuracy or loss to determine the optimal set of model parameters. But once we have settled on the final version of the model, it's extremely useful to carry out a more in-depth analysis of the model's performance This may include calculating other metrics such as precision, recall, and AUC (area under the curve), or calculating performance on a larger dataset than the validation set used in training.

Another reason for an in-depth model analysis is to check that the model's predictions are fair. It's impossible to tell how the model will perform for different groups of users unless the dataset is sliced and the performance is calculated for each slice. We can also investigate the model's dependence on features used in training and explore how the model's predictions would change if we altered the features of a single training example.

### 3.3.6 Model deployment

Machine learning models can be deployed in three main ways: with a model server, in a user's browser, or on an edge device. The most common way today to deploy a machine learning model is with a model server, which we will focus on in the next chapter. The client that requests a prediction submits the input data to the model server and in return receives a prediction. This requires that the client can connect with the model server.

### 3.3.7 Pipeline Orchestration

All the components of a machine learning pipeline described in the previous section need to be executed or, as we say, orchestrated, so that the components are being executed in the correct order. Inputs to a component must be computed before a component is executed. The orchestration of these steps is performed by tools such as Apache Beam, Apache Airflow, or Kubeflow Pipelines for Kubernetes infrastructure.

### 3.3.8 TensorFlow

Machine learning pipelines can become very complicated and consume a lot of overhead to manage task dependencies. At the same time, machine learning pipelines can include a variety of tasks, including tasks for data validation, preprocessing, model training, and any post-training tasks. Google decided to develop a platform to simplify the pipeline definitions and to minimize the amount of task boilerplate code to write. The open source version of Google's internal ML pipeline framework is TensorFlowExtended (TFX).

TFX provides a variety of pipeline components that cover a good number of use cases. Following components were available:

1. Data ingestion with ExampleGen
2. Data validation with StatisticsGen, SchemaGen, and the ExampleValidator
3. Data preprocessing with Transform
4. Model training with Trainer
5. Checking for previously trained models with ResolverNode
6. Model analysis and validation with Evaluator
7. Model deployments with Pusher

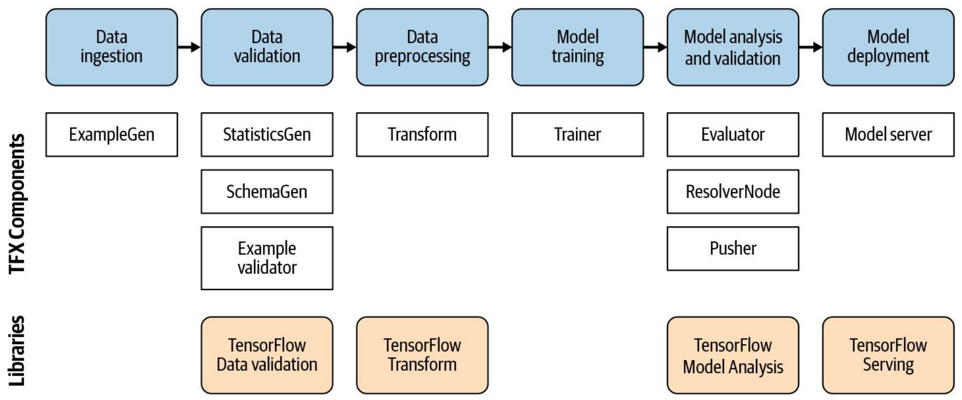


**Figure 3.8:** TFX components and libraries [7]

All machine learning pipeline components read from a channel to get input artifacts from the metadata store. Examples of artifacts include raw input data, preprocessed data, and trained models. Each artifact is associated with metadata stored in the MetadataStore. The data is then loaded from the path provided by the metadata store and processed. The output of the component, the processed data, is then provided to the next pipeline components.

The components of TFX are connected through metadata, instead of passing artifacts directly between the pipeline components, the components consume and publish references to pipeline artifacts. Metadata that are created during pipeline run are saved using Machine Learning Metadata (MLMD) API. MLMD saves the metadata to a MetadataStore, based on a storage backend (In-memory database, SQLite, or MySQL).

There are multiple alternatives to TFX such as AeroSolve by Airbnb, Luigi by Spotify, or Metaflow by Netflix and each of them is designed with specific engineering stacks in mind.

### 3.3.9 Apache Beam

A variety of TFX components and libraries (e.g., TensorFlow Transform) rely on Apache Beam to process pipeline data efficiently. Apache Beam offers us an open-source, vendor-agnostic way to describe data processing steps that then can be executed in various environments. Since it is incredibly versatile, Apache Beam can be used to describe batch processes, streaming operations, and data pipelines. In fact, TFX relies on Apache Beam and uses it under the hood for a variety of components (e.g., TensorFlow Transform or TensorFlow Data Validation).

## 3.4 Summary

In this chapter, I focus on the methodology. In the first part, I describe the difference between two software architectures. Monolithic architecture, which is suitable for the software that is being developed in a small team and serves usually a certain purpose such as a library or school information system with a database. They are easy to develop, deploy, test and monitor but in case of adding new functionalities or scaling out the system, several problems can occur. These issues can be solved by using a microservices architecture. In microservices, the system is broken into more independent services and each of the services lives on its own. This can give us a huge degree of flexibility in choosing technology, handling robustness, and scaling. On the other hand, this brings up a certain degree of complexity. When core concepts of microservices are well implemented, the system becomes very powerful and effective.

The next part is focused on containerization and container orchestration. Containerization is a form of operating system virtualization, through which applications are run in isolated containers, all using the same shared operating system. A container is a fully packaged and portable computing environment. The purpose is to be able to run the application anywhere. This is a must-have process in the case of using microservices architecture because of building, running, and deploying all the services. When a system gets bigger and scales out, container orchestration comes in handy. Kubernetes is a container orchestration platform for managing containerized workloads and services. It takes care of scaling and failover for the application, provides deployment patterns, and more.

The last part of this section is about Machine Learning Pipelines. ML Pipeline implements processes to accelerate, manage and deploy machine learning models. It starts with the ingestion of new training data, validating and preprocessing the data, continues with the machine learning model training, analysis, and validation, and ends with the model deployment. For a building such a pipeline we can use TensorFlow Extended, which is a library that supplies all of the components that are needed.

All of the methods that I described in this chapter are the basis for the implementation in this thesis. With the knowledge I gained, I can reconstruct the architecture, including Kubernetes, and implement Machine Learning Pipeline into our application.

# Chapter 4

# Implementation

In this chapter I describe proposed changes to the application and the implementation of these changes. The goal is to refactor the application based on the researched and described methodology in the previous section. The first half of the chapter describes proposed changes and the other half deals with implementation description. The benefits and results will be discussed in the next chapter.

## 4.1 Proposed changes

The main goal of this thesis is to analyze the software architecture, research methodology, propose, and implement changes that can boost application performance, and encapsulate different services in a microservices architecture.

The main proposed change is in the application architecture. In the Figure 4.1 is shown to-be architecture diagram. For applications that implement machine learning and gather data for improving this model every day, it is useful to create a Machine Learning (ML) Pipeline. Application in current version has machine learning logic implemented in Recommender service, as we can see in the Figure 2.1. One of the last steps in the ML Pipeline is Model Deployment. As previously described, the ML model should have its own server. The server should listen to requests from the client and respond with the prediction. This is how we can also achieve encapsulation of each service and we also improve the microservices architecture - in this case, it is scaling up by microservice decomposition (splitting one service into two separate ones).

In this case, the Recommender service will be split into Recommender and Predictor. Recommender handles requests from a web application (client) either for searching a certain place or area, or request for estimating the occupancy for the place. Request for estimating the occupancy level will be handled by the new Predictor service, which will respond with the occupancy levels for every 15 minutes during the day for the next 5 days.

Another useful feature of the ML pipeline is having a database for generated models and datasets. Each record in the Model database is described by model name, dataset file, encoding class file, object_ids model file, groups model file, and groups in NumPy array format file. Each of these files will be described in the next sections. Storing and versioning datasets can help us compare statistics of datasets or evaluations of models. With this information, we can improve the model or dataset. My idea is to create a Postgres database that will be storing these files and it will be attached to the Predictor service.

Furthermore, I create a pipeline that will handle all of the ML steps. Predictor service will include the whole pipeline process, from receiving a new dataset, through validation and processing to generating and evaluating the model with each run. If the pipeline is successful, a new model will be deployed to the production environment.

The last step is to create a container orchestration by using Kubernetes for easier deployment and monitoring, managing, or replicating services.

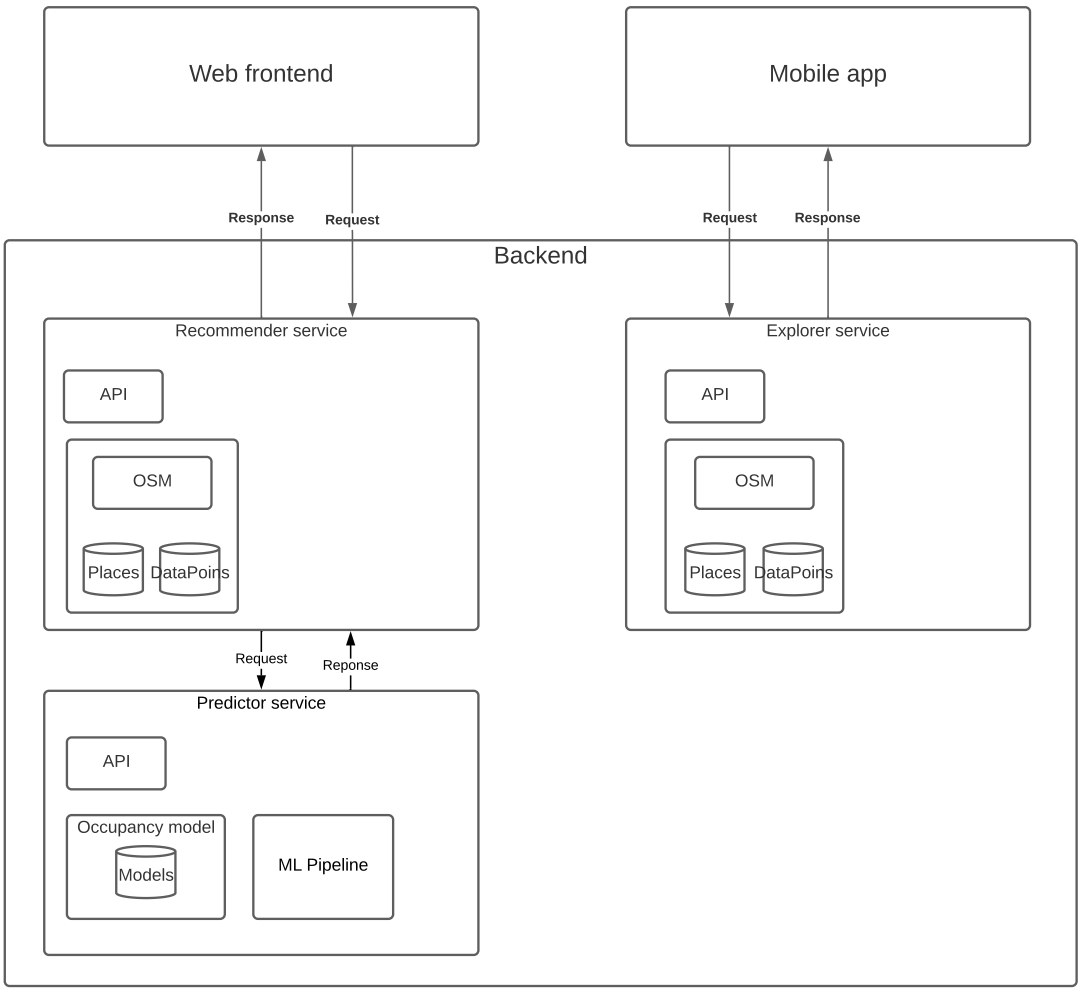


**Figure 4.1:** New application architecture

## 4.2 Machine Learning Pipeline

In this section, I describe the implementation of the Machine Learning Pipeline based on the knowledge I gained from research in the previous chapter. Each step of the pipeline is described in standalone sections with a code snippet. The first three steps are implemented using *TensorFlowExtended* (TFX) library and the rest is our custom code or components from *sklearn* library. I implemented the pipeline in the Jupyter Notebook. The implementation was based on TensorFlow documentation [25].

### 4.2.1 Data ingestion

In this step of the pipeline, I read data from the request body that is sent from the Recommender service. Recommender has a scheduler that every 24 hours (approximately at 2:00 am) create a new dataset. Data are loaded from the DataPoint database that contains measurements from volunteers, but before loading, we have to check if there are places that have been updated in OSM recently. If a place has been updated, we update this place in our database of places. After loading measurements data, dataframe is created by appending data from measurements and places records. Each measurement contains information about the place in which the measurement was made, and this information is used for finding this specific place in the Place database. When this place was found and retrieved from the database, we read the features we need for creating the dataframe.

When the creation of dataframe is done, we serialize it to JSON format and send it as a request for generating new models to the Predictor service. Predictor handles the request, deserializes the request data back to dataframe. Because new models are about to be created, a new record in the Model database is created as well. This dataframe is stored in the database as comma-separated value format file.

In the following code snippet is shown an example record of dataframe stored in comma-separated value format.

```
id,time,object_id,type,coords,crowded_ratio,people_num,name,
    address,opening_time,city,population,highway,parking,living,
    public_transport,hashed_key,server_ts,timezone,coords_device
0,2021-07-21 06:20:21.432000+00:00,39
    a60347739693e3c5645ca7fd68fd20,shop,"(49.8325153,
    18.2637946)",2,,"Alza, Novin  sk  , Moravsk  Ostrava","{'
    place_name': 'Alza', 'road': 'Novin  sk ', 'suburb': '
    Moravsk  Ostrava', 'city': 'Ostrava', 'municipality': '
    okres Ostrava-m sto', 'county': 'Moravskoslezsk  kraj', '
    state': 'Moravskoslezsko', 'postcode': '702 00', 'country':
    '  esk   republika', 'country_code': 'cz'}","{""mo"": [480,
    1200], ""tu"": [480, 1200], ""we"": [480, 1200], ""th"":
    [480, 1200], ""fr"": [480, 1200], ""sa"": [480, 1200], ""su"
    ": [480, 1200]}",Ostrava,304880.0,False,False,False,7,
    $2a$12$IVepg05nfcjCSK9s9ueD308Y51PxZKstzxDh4vo5W5DesT4uijQUO
    ,2021-07-21 06:20:22.169100+00:00,Europe/Prague,
```

The next step is to create a `TFRecord` format file that will contain the data from dataframe represented as `tf.Example` data structures. In tf.Example, each record contains one or more features that would represent the columns in our data. The reason for using such a format is that some TFX components work only with this format. Before creating TFRecord, the dataframe is modified by adding new columns (such as retail chain and timestamp) and filtering out records that have empty values in columns that we require having them as non-empty.

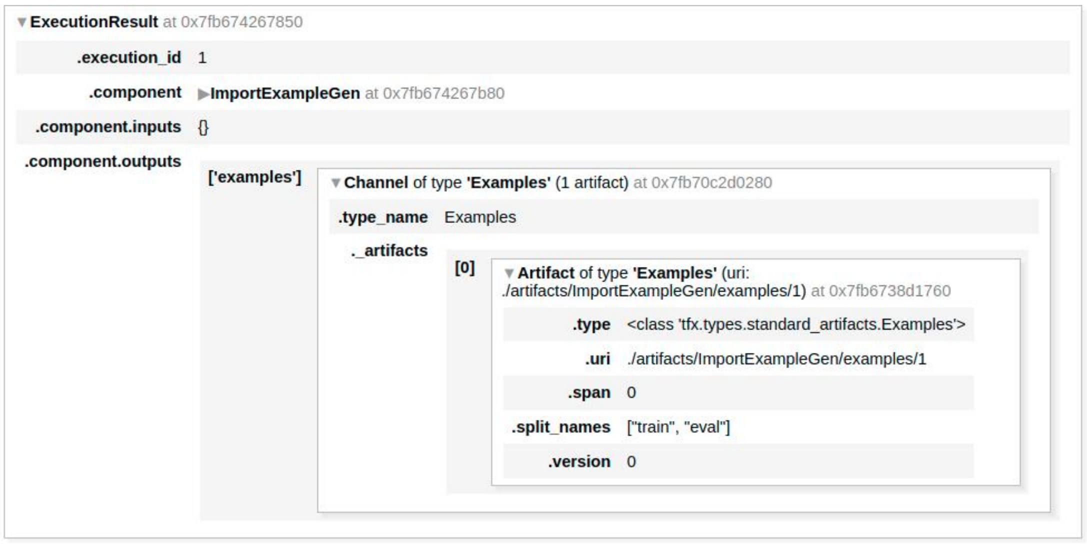
The following code snippet shows an example of how to create TFRecord from dataframe in python.

```python
tf_record_writer = tf.io.TFRecordWriter(tfrecords_filename)
    for index, row in df.iterrows():
        example = tf.train.Example(
            features=tf.train.Features(
                feature={
                    "type": _bytes_feature(row["type"]),
                    "coords": _bytes_feature(row["coords"]),
                    "crowded_ratio": _int64_feature(row["crowded_ratio"]),
                    "people_num": _float_feature(row["people_num"]),
                }
            )
        )
    tf_record_writer.write(example.SerializeToString())
tf_record_writer.close()
```

Generated TFRecord we can now load in ExampleGen component. This component handles the process of ingesting, splitting, and converting the datasets. Datasets can be read from local or remote folders. In our case, we read the local TFRecord by using `ImportExampleGen` component. Each of the ExampleGen components allows us to configure input settings (input_config) and output settings (output_config) for our dataset. We can configure how the data should be split. Often we would like to generate a training set together with an evaluation and test set. We can define the details with the output configuration. In the code snippet below, while data ingesting, I defined the output config to split the dataset in two-way split: training and evaluation sets of ratio 2:1. The ratio settings are defined through the `hash_buckets`.

```python
1  output = example_gen_pb2.Output(
2      split_config=example_gen_pb2.SplitConfig(splits=[
3          example_gen_pb2.SplitConfig.Split(name='train', hash_buckets=2),
4          example_gen_pb2.SplitConfig.Split(name='eval', hash_buckets=1)
5      ]
6          ))
7
8  example_gen = ImportExampleGen(input_base=_data_root,
9                                 output_config=output)
10 context.run(example_gen)
```

On line 10 I execute the component as part of an interactive pipeline, the metadata of the run is shown in Jupyter Notebook. The outputs of the component are shown in Figure 4.2. In the folder structure we can find the outputs in `ImportExampleGen` folder, where input dataset is split into `Split-train` and `Split-eval` folders, containing compressed TFRecords.



```
▼ExecutionResult at 0x7fb674267850
        .execution_id  1
           .component  ▶ImportExampleGen at 0x7fb674267b80
    .component.inputs  {}
   .component.outputs  ['examples']  ▼Channel of type 'Examples' (1 artifact) at 0x7fb70c2d0280
                                       .type_name  Examples
                                      ._artifacts  [0]  ▼Artifact of type 'Examples' (uri:
                                                        ./artifacts/ImportExampleGen/examples/1) at 0x7fb6738d1760
                                                                .type  <class 'tfx.types.standard_artifacts.Examples'>
                                                                 .uri  ./artifacts/ImportExampleGen/examples/1
                                                                .span  0
                                                         .split_names  ["train", "eval"]
                                                             .version  0
```

**Figure 4.2:** `ExampleGen` component output

## 4.2.2 Data validation

In this section, I describe the process of validating ingested data from the previous ML step. The data validation step checks that the data in a pipeline is what our feature engineering step expects. It assists us in comparing multiple datasets. For example, we can check statistics in the training dataset against the evaluation dataset or statistics in current and previous datasets still align. These statistics are able to highlight whether a feature contains a high percentage of missing values or if features are correlated.

It also checks for data anomalies or whether the data schema hasn't changed. Data anomalies could be values that do not match the expected pattern or other requirements in the dataset. A data schema is a form of describing the representation of our datasets. A schema defines which features are expected in the dataset and which type each feature is based on (float, integer, bytes, etc.). The schema definition of the dataset can then be used to validate future datasets to determine if they are in line with our previous training sets.

Because we can compare schemas, we can quickly detect if the data structure in newly obtained datasets has changed (e.g., when a feature is deprecated).

It can also detect if the data starts to drift. This means that our newly collected data has different underlying statistics than the initial dataset used to train the model.

A potential problem with the input dataset is data bias. Data bias is a type of error in which certain elements of a dataset are more heavily weighted or represented than others. A biased dataset does not accurately represent a model's use case, resulting in skewed outcomes, low accuracy levels, and analytical errors [26].

The TensorFlow library offers a tool that does data validation, TensorFlow Data Validation (TFDV) and it is part of the TFX project. TFDV allows us to perform the kind of analyses I described previously (e.g., generating schemas and validating new data against an existing schema). It also offers visualizations.

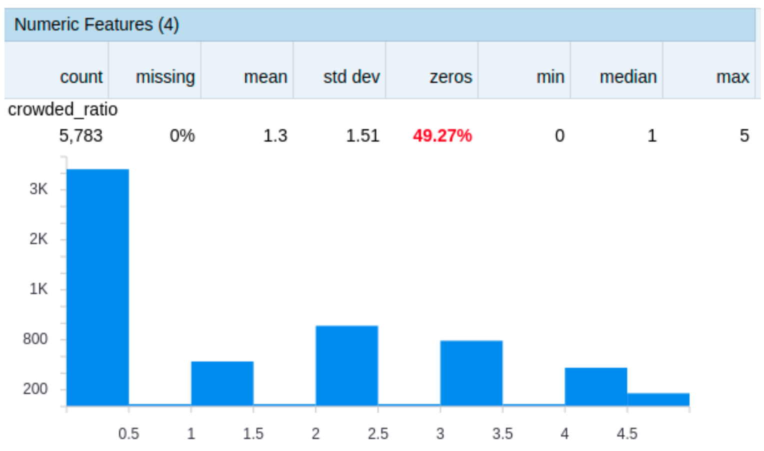
The first step in our data validation process is to generate summary statistics for our data. In the code snippet below, I load the TFRecord dataset with TFDV and generate statistics for each feature:

```
1 stats = tfdv.generate_statistics_from_tfrecord(
2     data_location=_data_root + '/data.tfrecords')
3 tfdv.visualize_statistics(stats)
```

For numerical features, TFDV computes for every feature:

1. The overall count of data records
2. The number of missing data records
3. The mean and standard deviation of the feature across the data records
4. The minimum and maximum value of the feature across the data records
5. The percentage of zero values of the feature across the data records

In addition, it generates a histogram of the values for each feature as we can see in Figure 4.3below by calling *visualize_statistics* function on line 4.



Numeric Features (4)

| | count | missing | mean | std dev | zeros | min | median | max |
|---|---|---|---|---|---|---|---|---|
| crowded_ratio | 5,783 | 0% | 1.3 | 1.51 | 49.27% | 0 | 1 | 5 |

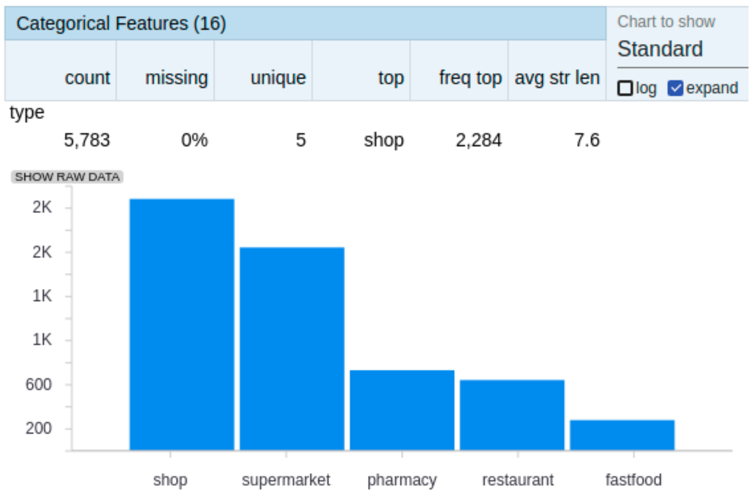
**Figure 4.3:** `TFDV` numeric feature statistics visualization on our dataset

As we can see from the generated statistics for the `crowded_ratio` feature, a lot of zeros are present in the dataset but in that case, zero means that in the specific time that measurement was made, the place was either closed or it was empty. Because we get rid of missing values before ingesting the data, the missing percentage remains zero.

For categorical features, TFDV provides:

1. The overall count of data records
2. The number of missing data records
3. The number of unique records
4. The average string length of all records of a feature
5. For each category, TFDV determines the sample count for each label and its rank

In Figure 4.4 is shown generated statistics for the `type` feature from our dataset, again the missing percentage is zero because we filtered out missing categories before ingesting, there are five unique categories and the most popular is shop.



Categorical Features (16)

| | count | missing | unique | top | freq top | avg str len |
|---|---|---|---|---|---|---|
| type | 5,783 | 0% | 5 | shop | 2,284 | 7.6 |

Chart to show: Standard
☐ log ☑ expand

**Figure 4.4:** `TFDV` categorical feature statistics visualization on our dataset

Once we have generated our summary statistics, the next step is to generate a schema of our dataset. As shown in the following code snippet, I generate the schema information from the generated statistics with a single function call *infer_schema* and display the result by calling *display_schema*:

```
schema = tfdv.infer_schema(stats)
tfdv.display_schema(schema)
```

And the results are shown in Table 4.1. In this visualization, Presence means whether the feature must be present in 100% of data examples (required) or not (optional). Valency means the number of values required per training example. In the case of categorical features, single would mean each training example must have exactly one category for the feature. The schema that has been generated here may not be exactly what we need because it assumes that the current dataset is exactly representative of all future data as well. If a feature is present in all training examples in this dataset, it will be marked as required, but in reality, it may be optional.

The schema of the dataset I generated, now becomes handy. TFDV can validate any data statistics against the schema, and it reports any anomalies. The following code snippet shows how to generate anomalies and display them. If there are no anomalies found, the result message is "No anomalies found." - which is our case. If an anomaly was found, it is shown in a table with information about features such as feature name, anomaly short description, and anomaly long description. With this information, we are aware of the potential issues and we can update the schema and dataset.

```
anomalies = tfdv.validate_statistics(statistics=stats, schema=
    schema)
tfdv.display_anomalies(anomalies)
```

| Generated Schema from our dataset | | | | |
|---|---|---|---|---|
| Feature Name | Type | Presence | Valency | Domain |
| address | BYTES | required | - | - |
| city | BYTES | required | - | - |
| coords | BYTES | required | - | - |
| coords_device | BYTES | required | - | - |
| highway | STRING | required | - | highway |
| living | STRING | required | - | living |
| name | BYTES | required | - | - |
| object_id | BYTES | required | - | - |
| opening_time | BYTES | required | - | - |
| parking | STRING | required | - | parking |
| retail_chain | BYTES | required | - | - |
| server_ts | BYTES | required | - | - |
| time | BYTES | required | - | - |
| timestamp | BYTES | required | - | - |
| timezone | STRING | required | - | timezone |
| type | STRING | required | - | type |
| crowded_ratio | INT | required | - | - |
| people_num | FLOAT | required | - | - |
| population | FLOAT | required | - | - |
| public_transport | INT | required | - | - |

**Table 4.1:** Data Schema

| Domains | |
|---|---|
| Domain | Values |
| highway | 'False', 'True' |
| living | 'False', 'True' |
| parking | 'False', 'True' |
| timezone | 'Europe/Prague', |
| type | 'shop', 'supermarket', 'fastfood', 'pharmacy', 'restaurant' |

**Table 4.2:** Data Domain

A very important step in validation is to determine how representative the validation set is in regards to the training set. As shown in the following code snippet, I load both datasets and then visualize both of them.

```
train_stats = tfdv.generate_statistics_from_tfrecord(
    data_location=example_uri + '/Split-train/data')
val_stats = tfdv.generate_statistics_from_tfrecord(
    data_location=example_uri + '/Split-eval/data')

tfdv.visualize_statistics(lhs_statistics=val_stats,
   rhs_statistics=train_stats,
                              lhs_name='VALIDATION_DATASET',
   rhs_name='TRAIN_DATASET')
```

The results are shown in Figure 4.5 and in Figure 4.6. These visualized results can give us a good overview of how the data looks like in both datasets. For example, we can now see that the zeros percentage in both datasets in Figure 4.5 is very similar and if we look at the generated histogram the rest of the values are also distributed well. In the case of categorical features in Figure 4.6, we can observe similar results.

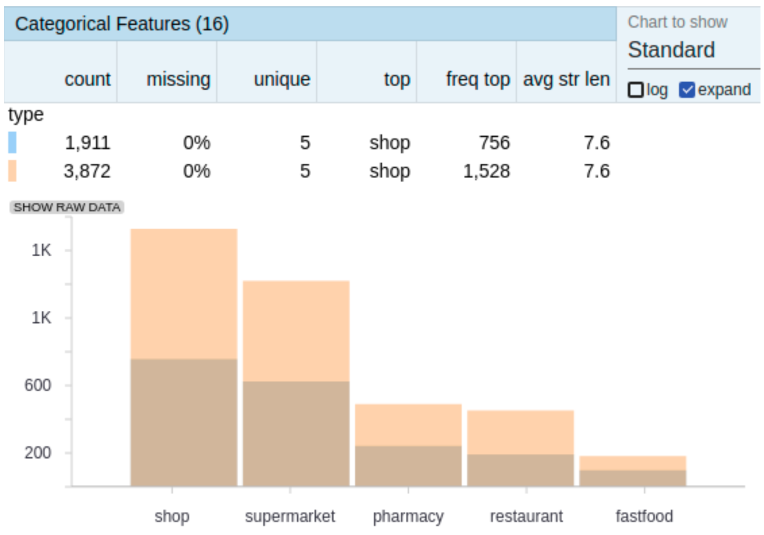
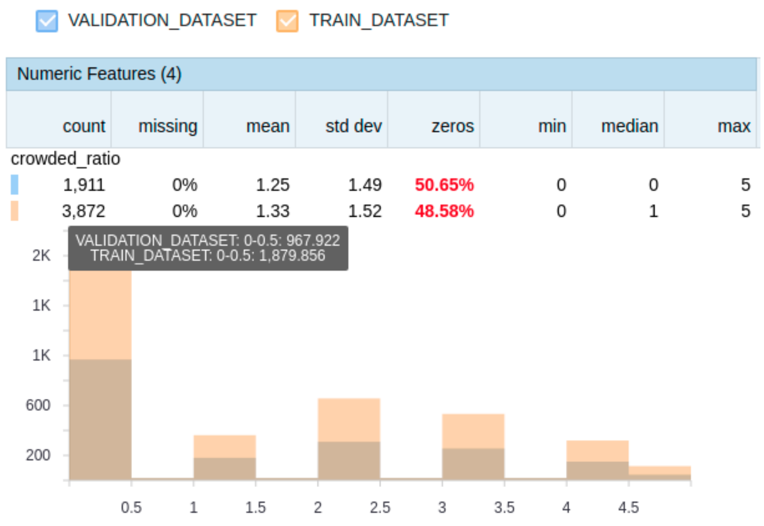

Categorical Features (16)

Chart to show: Standard ☐ log ☑ expand

| type | count | missing | unique | top | freq top | avg str len |
|---|---|---|---|---|---|---|
| 🟦 | 1,911 | 0% | 5 | shop | 756 | 7.6 |
| 🟧 | 3,872 | 0% | 5 | shop | 1,528 | 7.6 |

**Figure 4.5:** Generated validation dataset against traning dataset - detailed on `crowded_ratio` feature


☑ VALIDATION_DATASET ☑ TRAIN_DATASET

Numeric Features (4)

| crowded_ratio | count | missing | mean | std dev | zeros | min | median | max |
|---|---|---|---|---|---|---|---|---|
| 🟦 | 1,911 | 0% | 1.25 | 1.49 | 50.65% | 0 | 0 | 5 |
| 🟧 | 3,872 | 0% | 1.33 | 1.52 | 48.58% | 0 | 1 | 5 |

VALIDATION_DATASET: 0-0.5: 967.922
TRAIN_DATASET: 0-0.5: 1,879.856

**Figure 4.6:** Generated validation dataset against traning dataset - detailed on `type` feature

If we want to integrate the validation step into the ML pipeline, TFX provides a component name `StatisticsGen`, which accepts the output of `ExampleGen` components as input and then performs the generation of statistics. For generating schema, we use SchemaGen component and if we want to check the dataset for anomalies, we use ExampleValidator. The following snippet shows the code that uses all of the mentioned components.

```
statistics_gen = StatisticsGen(
    examples=example_gen.outputs['examples'])
context.run(statistics_gen)

schema_gen = SchemaGen(
    statistics=statistics_gen.outputs['statistics'],
    infer_feature_shape=True)
context.run(schema_gen)

example_validator = ExampleValidator(
    statistics=statistics_gen.outputs['statistics'],
    schema=schema_gen.outputs['schema'])
context.run(example_validator)
```

### 4.2.3 Data preprocessing

The data we use to train our models are often provided in formats our models can't consume. For example, in our application, a feature we want to use to train our model is available only as 'True' and 'False' values. Any machine learning model requires a numerical representation of these values (1 and 0). In this section, I describe how we convert features into numerical representations so that our machine learning model can be trained with the numerical representations of the features.

In this step of ML Pipeline, I implement the TFX component Transform (TFT) to show how can we convert data using the TFX library, and the rest of preprocessing is done using sklearn preprocessing library.

The key to TFT is the preprocessing_fn function that defines all transformations we want to apply to the raw data. When we execute the Transform component, the preprocessing_fn function will receive the raw data, apply the transformation, and return the processed data. Previously mentioned function is implemented in Python file (e.g., module.py in our case). When we execute the Transform component, TFX will apply the transformations defined in our module.py module file to the loaded input data. The component will then output our transformed data, a transform graph, and the required metadata. The only preprocessing I do in TFT is converting sparse features to dense - by filling the missing values with default values (zero for numeric values and empty string for bytes values). At the same time, I filter out features, we don't need when training the models. Features we care about are highway, living, parking, public_transport, population, crowded_ratio, retail_chain, timestamp, type, and object_id. The following code snippet shows how I implement TFT in the pipeline.

```
transform_file = os.path.join(os.getcwd(), './scripts/module.py'
    )
transform = Transform(
    examples=example_gen.outputs['examples'],
    schema=schema_gen.outputs['schema'],
    module_file=transform_file)
context.run(transform)
```

TFT is capable of many useful operations for preprocessing such as one-hot encoding, normalization, bucketing, etc. We may be able to include the whole preprocessing data process in the TFX environment but for now, the rest of the process is done using sklearn preprocessing library.

After the transformation is done, processed data are again stored as compressed TFRecords. Records are stored in `Transform/transformed_example` folder. From this point, we need to transform TFRecord to pandas dataframe, because the format that preprocessing methods consume is dataframe.

I pass the dataframe as an argument to the encoding function, that we implemented in the Predictor. The first step is to create weights for each feature (type, retail_chain, highway, living, parking, population, and public_transport). Weights are important in calculating the distance between categories in variables while estimating the model for prediction (detailed description in Models evaluation section). The next step is to transform features into one-hot encoding. Each of the categorical features (type, living, highway, retail_chain, parking) is preprocessed using `OneHotEncoder`. OneHotEncoder encodes categorical features as a one-hot numeric array. The input to this transformer should be an array-like of integers or strings, denoting the values taken on by categorical (discrete) features. The features are encoded using a one-hot encoding scheme. This creates a binary column for each category and returns a sparse matrix or dense array [8]. These encoders for each categorical feature are saved in a dictionary. The following code snippet shows the example of a one-hot encoding of the 'type' feature.

```
categories = ['fastfood', 'pharmacy', 'restaurant', 'shop', '
    supermarket']
# Let's say that some random record from the dataset has a 'shop
    ' category, after encoding it will be transformed into an
    array with binary values. 0 represents that it doesn't
    belong to the category and 1 represents the fact, that it
    belongs to that category mapped by categories array.
one-hot-transform = [0. 0. 0. 1. 0.]
# After multiplying one-hot with its weight.
weighted-transform = [0 0 0 100 0]
```

All of the weights, encoders dictionaries, and features are saved into `Python Pickle File` (PKL) format that enables objects to be serialized to files on disk and deserialized back into the program at runtime. This file is also stored in a database together with the dataset mentioned in the Data ingestion section.

After saving the encoding classes to the database, another transformation is done. We create another dataframe column called `'ohe'` that concatenates all of the one-hot encoded features and numerical features such as population and public transport multiplied by their weight. This column creates a unique definition of a place based on its characteristics. One more step before creating models is to create a set of 'groups' - newly created ohe column and set of 'oids' which is a set of column object_id - unique id of each place from OSM.

### 4.2.4 Creation of models

In this section, I describe the creation of models for predicting crowdedness. By this point, data has been ingested, validated, and preprocessed. This ensures that all the data needed by the model is present and that it has been reproducibly transformed into the features that the model requires. We want to ensure that the training proceeds smoothly because it is often the most time-consuming part of the entire pipeline.

In this step of the ML pipeline, we use our custom-created algorithms for creating statistics models. This step is hard to implement into the TFX environment because TFX uses components that require specific data structures. Because of that, our pipeline doesn't use TFX anymore.

We create two types of object models for predicting. The first one is storing object_ids (oids) and its crowdedness predictions for a week. Second object model stores groups and their predictions. The process of creating predictions is the same for both of them.

Note that the sizes of the set of groups and object_ids aren't necessarily the same. Object id is unique for each place but the group is described as encoded characteristics of the place which means that more places can have the same group (or a very similar one).

When creating predictions for a set of object_ids, only required data for training are filtered first. We filter out `timestamp, crowded_ratio` features that belongs to the specific object_id. We required at least two measurements for creating predictions. By using the algorithm described in the first chapter, we are able to create crowdedness predictions for different times of the day. Predictions for that place (described by object_id) are saved to array of models together with these information: `object_id, length` - number of measurements for that place, `position` - array representation of transformed features and `prediction_week` - matrix representation of crowdedness predictions. The length of the array of models depends on the number of unique object_ids in the dataset. Each model is then saved to the dictionary data structure, where the key is the object_id and values the rest of the data that the array contains. In the end, after we iterated over all of the records, the dictionary of models is saved to the database as 'models_object_id' PKL.

Creating predictions for a set of groups is the same process as for object_ids. An array of models is now represented by group name but the rest remain the same. While appending new values to the array of models, the NumPy array of groups ('numpy_array_ohes') is created and described as the number of measurements and position - representation of an array of transformed features. The length of the array again depends on the number of unique groups in the dataset. After creating the dictionary data structure, this dictionary is as well saved to the database as 'models_ohe' PKL. Besides saving the models, we also save created NumPy array of groups to the database as a NumPy file.

After explaining the creation of models, we can now assume how the predictions are estimated. Usually, one model is able to predict the desired state but in our case, the more models we have, the more accurate estimation can be.

### 4.2.5 Models evaluation

At this point in our ML pipeline, we have checked the statistics of our data, we have transformed our data into the correct features, and we have created trained models.

Our model analysis process starts with the choice of metrics. It's good practice to pick multiple metrics that make sense for our business problem because one single metric may hide important details. In this section, I will review some of the most important metrics.

Once the creation of models is finished, the pipeline continues with evaluation. Firstly, we have to transform an evaluation dataset to the form that the estimator can handle. I load the evaluation dataset from TFRecord to pandas dataframe and parse the dataframe. Parsed dataframe is sent as data request to 'predict' endpoint. The predictor estimates the model based on the object_id of the place that requires predictions if sufficient measurements were provided for this place, otherwise, it estimates the model by using the distance between object parameters and group parameters. The estimated model then estimates the crowdedness using mathematical operations. The output is a list of 672 estimated values. Each value represents 15 minutes time gap during the day. The output is then parsed for the next five days also counting today's day.

To check if the evaluation is correct, I have to check the predicted crowdedness value to the actual value from the evaluation dataset. After the prediction was made, we need to find the correct value in the output array. If the actual crowdedness value was measured on Tuesday at 15:46, I have to find this specific day and the value that represent this time. I write a python script shown in the code snippet below, that counts that position and checks the day of the week when the measurement was made.

```python
def transform_date(time):
    days = {0: 'mo', 1: 'tu', 2: 'we', 3: 'th', 4: 'fr', 5: 'sa'
    , 6: 'su'}
    for fmt in ('%Y-%m-%d %H:%M:%S.%f%z','%Y-%m-%d %H:%M:%S%z'):
        try:
            datetime_object = datetime.strptime(time, fmt)
            break
        except ValueError:
            pass
    day = datetime_object.weekday()
    day_in_arr = days[day]
    hour = datetime_object.hour
    minutes = datetime_object.minute
    quarter = minutes / 15
    position = hour * 4 + math.floor(quarter)
    return position, day_in_arr
```

With this information, I am able to check the actual value against the predicted one. I store all of the actual values and all of the predicted values in separate arrays. Actual values are represent by `crowded_ratio`, which is a number from 0 to 5. Predicted values aren't integers, but floats. Floats round up or down depending on their value, so it becomes an integer that can be checked against actual value.

To calculate many classification metrics, it's necessary to first count the number of true/false positive examples and true/false negative examples in our evaluation set. *True positives* - Training examples that belong to this class and are correctly labeled as this class by the classifier. *False positives* - Training examples that do not belong to this class and are incorrectly labeled as this class by the classifier. *True negatives* - Training examples that do not belong to this class and are correctly labeled as not in this class by the classifier. *False negatives* - Training examples that belong to this class and are incorrectly labeled as not in this class by the classifier. With this matrix, called `confusion matrix`, we are able to calculate other metrics such as:

`Accuracy` - defined as (true positives + true negatives)/total examples, or the proportion of examples that were classified correctly. This is an appropriate metric to use for a dataset where the positive and negative classes are equally balanced, but it can be misleading if the dataset is imbalanced.

`Precision` - defined as true positives/(true negatives + false positives), or the proportion of examples predicted to be in a positive class that was classified correctly. So if a classifier has high precision, most of the examples it predicts as belonging to the positive class will indeed belong to the positive class.

`Recall` - defined as true positives/(true positives + false negatives), or the proportion of examples where the ground truth is positive that the classifier correctly identified. So if a classifier has a high recall, it will correctly identify most of the examples that are truly in the positive class.

Figure 4.7 gives us a visual idea of how Precision and Recall differ from each other.

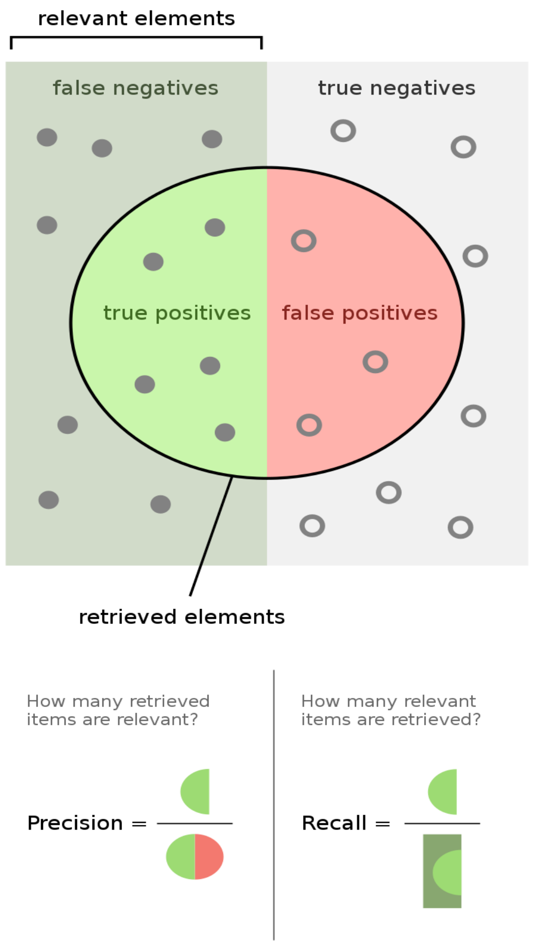


**Figure 4.7:** Precision and recall [8]

Another way to generate a single number that describes a model's performance is the AUC (area under the curve). The curve is the receiver operating characteristic (ROC), which plots the true positive rate (TPR) against the false positive rate (FPR).

One more important metric is calculating *Mean absolute error* (MEA). The numerical value for each training example is compared with the actual value. $\text{MAE} = \frac{1}{n}\sum|y - \hat{y}|$

where `n` is the number of training examples, `y` is the true value, and `ŷ` is the predicted value. For each training example, the absolute difference is calculated between the predicted value and the true value. In other words, the MAE is the average error produced by the model.

The results of all of these metrics will be discussed in the next chapter.

### 4.2.6 Model deployment

The deployment of your machine learning model is the last step before others can use your model and make predictions with it. The most common way today to deploy a machine learning model is with a model server, which I will focus on in this section. The client that requests a prediction submits the input data to the model server and in return receives a prediction. This requires that the client can connect with the model server.

I have created a Python web application using the Django framework (same as Recommender or Explorer services). I defined the endpoints that the Predictor will handle. The important endpoint is 'predict' that parse the request data and send them to the model prediction. The response is a response code with the data, that model predicted. Another endpoint is for generating new models. The request is sent from Recommender with requests data that contains dataframe that has been generated from the latest measurements. Newly generated models and datasets are stored in the Postgres database that is attached to this backend.

## 4.3 Summary

In this chapter, I discussed proposed changes to the application. The main goal was to implement Machine Learning Pipeline and split the logic of the Recommender service into two separate Recommender and Predictor services and create a database for saving generated models. In ML pipeline I work with *TensorFlowExtended* and *sci-kit learn* libraries.

I start with describing steps in ML Pipeline. Firstly, data has to be ingested into the pipeline. This step is called 'Ingesting data'. Taking our project as an example, I showed how to convert dataframe into TFRecord, which is a file format that accepts components from the TFX library. We load the data to `ExampleGen` data structure with the output configuration that split the input dataset into train and evaluation sets.

The next step is 'Data Validation'. In this step, we use the *TensorFlow-DataValidation* component, which has a few very handy functions. We are able to load the TFRecord into TFDV and the output is statistics with visualization. I discussed how to generate schemas and how to compare two different datasets based on their statistics and schemas. We are also able to validate the dataset by detecting any kind of anomalies.

The third step, 'Data Preprocessing' is effective preprocessing data in our machine learning pipelines with *TensorFlow Transform* (TFT). I described how to write the preprocessing_fn function, provided an overview of some available functions provided by TFT, and discussed how to integrate the preprocessing steps into the TFX pipeline. Only a validation of missing data was preprocessed using TFT, the rest of preprocessing is done in our project using *sklearn preprocessing* library. We load the important features that step into to training model and we create weights for each of the features. Categorical features are converted to one-hot encoders, which is a process of encoding categorical features as a one-hot numeric array. This creates a binary column for each category and returns a dense array. Output values are then multiplied by their weight. Before models are being created, a set of groups and a set of object_ids are created. Groups are defined as a string that is concatenated from each input transformed feature, defined by its characteristics (such as category, living, highway, parking, population, etc.).

Now that the data has been preprocessed, it is time to train our models. For each data in a set of groups and set of object_ids we generate a statistics model with predictions of crowdedness for each day in a week. These models are both saved with the results in the database.

Before deploying the model, we have to evaluate it first. Evaluation is done by predicting all places' crowdedness from the evaluation dataset. Each place is parsed to a form that the Predictors estimator can handle as an argument, and the output is an array of predictions of busyness for every 15 minutes for the next five days. We compare the actual value that was measured by a user from the testing dataset and the predicted value for that specific time and day of the week. I use several classification metrics such as confusion matrix, accuracy score, precision and recall curve, and ROC curve to determine the Predictor's success rate.

The last part is dedicated to 'Model deployment'. I create a web application with the Python framework Django. I create API endpoints in the web app, that handles several requests, e.g., predicting, generating model, generating record. I implemented a Postgres database that stores generated models for data and models versioning.

Chapter 5

# Impact of implemented changes

In this chapter, I discuss the results and benefits of implemented changes and the overall impact of the implementation on the application.

## 5.1 Microservices decomposition

When implementing an application with Machine Learning Pipeline it can be very handy to have a separate service for that. It gives us the freedom to choose what tool or framework is right for developing such a complex pipeline as well as choosing the right data storage for example, for saving models. In case of failure of generating new models from a new dataset or failure of prediction of some state, we are able to isolate it accordingly. Developing and testing such a service depends now on a smaller team of developers which eliminates problems such as waiting for others to finish and test the functionality of another use case, large codebase, and eventually less misunderstanding, which is very common in the development world. Another benefit and advantage mentioned in microservices is the ability to scale if needed in the future in case of the increasing popularity of the application.

I evaluate the separation of responsibilities from Recommender into Recommender service and Predictor (ML) service as a benefit to the application based on the believes mentioned in this section.

## 5.2 Data validation

During the implementation of the Machine Learning Pipeline, I encounter a very useful tool in the TensorFlow Data Validation (TFDV) component for visualizing the statistics and schema of the dataset. The visualization of features in graphs, together with information about its count, missing values, number of zeros, minimum or maximum numeric value, or number of unique values in categorical features, etc.

Providing these statistics (Figure A.1), I can tell that the dataset has too many measurements where the crowded ratio was zero and just a few measurements where the crowded ratio was 5. This can lead to inaccurate predictions in busy or rush hours. In the case of the population feature, which describes the population of the city place where the measurement was made, from the statistics I can see that many measurements were made in small cities up to 100k inhabitants and cities that have more than 1 million inhabitants and there are little examples of cities that are in between this range. In the case of categorical features, highway and parking features having a balanced number of examples (almost 1:1) is a good metric for training the model. On the other hand, the category of the place where measurement was made is again imbalanced, for example, the ratio of shops and restaurants is 3.6:1.

These statistics can be compared, for example, if we split the dataset into training and evaluation sets, we can check if the data still align.

Another handy function that the TFDV component provides, is generating the data schema. We are able to see, what data types describe features if the feature is required (if it's present in every record) and we can validate the schema against the generated statistics to see if there are any anomalies or outliers present. Again, we can compare, for example, schema from the previous dataset that has been used for training models with the current schema to check if data types still correspond.

## 5.3 Models versioning

Together with the implementation and deployment of ML service, I created a new data storage for saving and versioning models and datasets. We are able to look back into old models or datasets, compare them with new ones, or generate statistics against any data from the past. This is very beneficial when validating the data as I mentioned in the previous section. It is useful not only when storing newly created models but also for querying the data based on the attribute value (e.g., date of the creation, or name of the model).

The original implementation was through the file system, where we had to iterate through existing directories and folders and create new ones if we needed to. Searching and retrieving in such a way is difficult and unnecessarily demanding.

Implementing the database for saving and versioning models and datasets makes working and querying the models, datasets, or attributes easier.

## 5.4 Models evaluation

In this section, I discuss the results I received by using different classification metrics that evaluate ML models' success and accuracy.

As I described in the Implementation chapter in the Evaluation models section, predicted values from the estimator are floating points that are rounded to integers so they can be compared to the actual values from measurements. In reality, the traffic graph that represents the estimated predictions is using the range between two values - (0,1) as crowded ratio 1, (1,2) as crowded ratio 2, etc. That's why this evaluation isn't necessarily the most accurate.

First of all, I generate the confusion matrix that provides us the number of true positives, true negatives, false positives, and false negatives values required as an input for other metrics. Besides, by investigating the output values we can evaluate how accurate predictions are. The result is shown in Table 5.1.

| Confusion matrix | | | | | |
|---|---|---|---|---|---|
| 551 | 36 | 15 | 5 | 0 | 0 |
| 50 | 58 | 36 | 13 | 1 | 0 |
| 60 | 41 | 72 | 45 | 6 | 0 |
| 32 | 31 | 44 | 60 | 23 | 1 |
| 5 | 9 | 16 | 27 | 47 | 2 |
| 6 | 5 | 2 | 9 | 11 | 0 |

**Table 5.1:** Confusion matrix

Columns in this matrix represent predicted values from 0 to 5. Rows represent actual values. The diagonal represent true positive value, which means that predicted and actual value match. As we can tell, the most precise predictions are when predicts 0 value and the worst predictions are for value 5, which makes sense based on the statistics generated and discussed in the previous section. To provide a general view of how to read the matrix [row, column], I describe the predictions for value 0. By far we know that the first column represents predicted zeros on evaluation data. On [0,0] position, predicted and actual value matches. On [0,1] position, 50 values were predicted as 0, but the actual value was 1 - this is labeled as false positive. On [0,2] position, 60 values were predicted as 0 but the actual value was 2, and so on. On position [1,0], 1 was predicted but actual was 0 is labeled as a false negative. As the distance between indices grows, the number of incorrectly predicted values decreases, which is a good sign that the predictor is very close to the actual value.

In Table 5.2 we can see an overview of classification metrics and results, index of result arrays start from 0 and represent category number (0-5).

**The precision score** for each category is the number of true positives divided by the total number of elements labeled as belonging to the positive class. By looking at the result, we can tell that categories 0 and 4 have the most valid results because, in the case of category 4, 47/88 predicted values were predicted correctly.

**The recall score** for each category is the number of true positives divided by the total number of elements that actually belong to the positive class. By investigating the results, we can see that category 0 missed only less ten 10% actual values because it correctly matched 551/607 values. Category 5 is again 0 in the results because 0/33 were predicted.

**Accuracy score** is a single number, that calculates all correctly labeled values as true positives (actual and predicted value belongs to the same category) and correctly labeled values as true negatives so that the predicted value doesn't belong to the rest of the categories. Almost 60% accuracy is in my point of view a good number.

Calculating **mean absolute error** is a necessity in the evaluation. If we think about the number of categories that our predictor has to estimate values, the absolute value of an error can be 5. The best value for MAE is 0. The output of MAE is around 0.63, which means that the estimator made an average error just by 0.6 between actual and predicted values. If we get back to the part where I describe how the traffic graph plot the values, this error is still between the range in the same category. I consider this number as a good sign that the estimator is very close to the actual values.

| **Metric** | **Results** |
|---|---|
| Precision score | [0.78267045, 0.32222222, 0.38918919, 0.37735849, 0.53409091, 0.] |
| Recall score | [0.907743 , 0.36708861, 0.32142857, 0.31413613, 0.44339623, 0.] |
| Accuracy score | 0.5974222896133434 |
| Mean absolute error | 0.6322971948445792 |

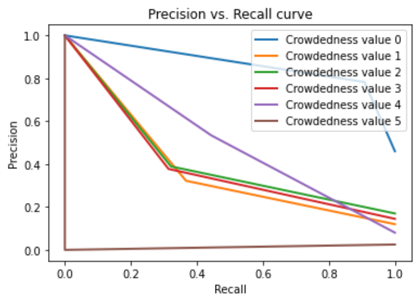
**Table 5.2:** Classification metrics results

In Figure 5.1 are shown classification metrics AUC and Precision vs Recall curve represented in graphs.

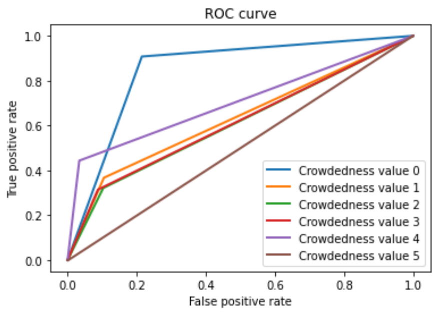
In Figure 5.1b is shown AUC. That is defined by True Positive Rate (TPR) against False Positive Rate (FPR). TRP is calculated as true positives / (true positives + false negatives) or Recall score and FPR is calculated as false positives / (false positives + true negatives). As the ROC moves further away from the x-axis toward the upper left of the plot, the model improves.

Ideal TPR is 1, on the contrary, ideal FPR is 0. The closest to these values is category 0, the worst is category 5.

In Figure 5.1a is shown precision score against recall score. Reviewing both precision and recall is useful in cases where there is an imbalance in the observations between the classes. For example in our case, there are many examples of category 0 and only a few examples of category 5. A no-skill classifier cannot discriminate between the categories, the curve in this case changes based on the distribution of the positive to negative classes but it tends to go towards the point (0,0). A skillful model is represented by a curve that goes towards (1,1), the closest to this point is again category 0.



**(a) :** Presicion vs Recall curve



**(b) :** ROC curve (AUC)

**Figure 5.1:** Classification metrics represented in graphs

## 5.5 Summary

In this chapter, I discussed the overall impact of implemented proposed changes to the application. I described that the decomposition of Recommender to Recommender and Predictor (ML) service was beneficial in our case because we separated the responsibilities and we allow both of the services to scale, deploy, monitor, and manage independently.

Another benefit is that I implemented a Postgres database for saving and versioning generated models and datasets. Few benefits I can name are: reducing the complexity of retrieving the models and datasets, querying data based on the different attributes, overall transparency of models.

When implementing Machine Learning Pipeline, several tools for validation of the dataset come in handy. I was able to see the statistics of the dataset and thus we can tell that the dataset has its issues, which can cause that predicted values aren't very accurate in some cases. These statistics can be compared against each other and we can check if data still align.

The last part is dedicated to models evaluation. I discussed the results of classification metrics that have been used. I can tell that the predictions are 60% accurate and the average error produced by the model is 0.6, which is a small number when the range of an error can be from 0 to 5. That means, that the predictions are inaccurate by 0.6 decimal points from the actual true value. Another observation I can see is that when the model is predicting category 0, it is almost 90% correctly matched because in the training dataset there is a huge imbalance in measurements of crowded ratio (thousand of measurements with crowded ratio 0 and just a few hundred with crowded ratio 5 or other). On the other hand, when predicting category 5, the model fails.

# Chapter 6

# Proposal for further application development

In this chapter, we describe suggestions that can be done to improve the estimator predictions, and what other steps can be done in enhancing the automation of the model life cycle.

Based on the results we achieved by evaluating the model we could see that the measurements dataset is imbalanced. If we could be able to acquire more balanced measurements such as measurements from different times of the day in various places (e.g., small local shops, big supermarkets, city parks, restaurants, etc.) in the cities with various populations, and during empty times, busy hours and medium traffic, we would be able to train the model with diverse data and thus, the model predictions would be more accurate for different times of the day and more diverse places. This is not an easy task to achieve, because it depends on the number of volunteers throughout the republic.

Another suggestion is in the step of evaluating the model, where instead of rounding predicted values, I would create intervals that the predicted value belongs to and transform in such a way also the crowded ratio number. This transformation would bring us better results in the evaluation of the model.

In the machine learning pipeline, we introduced the TensorFlow Transformer component, which is responsible for data preprocessing. In the implementation, we use the Transformer for filling in missing values, but it is capable of many other methods for preprocessing such as converting from categorical feature value to one-hot encoded value or normalizing numeric values. Because this preprocessing was originally done in the implementation, I didn't implement it in the Transformer component. In future work, I suggest implementing whole data preprocessing into the Transformer component.

The last proposal is to set up Kubernetes - container orchestrator tool. It provides the user the ability to configure system behavior and automates container management by configuration. This can be very beneficial for applications with microservices architecture when developing, testing, or deploying the application to production. It provides options for load balancing, storage orchestration, self-healing of the container, etc.

# Chapter 7

# Conclusion

In this thesis, the goal was to analyze the structure and methodology of the application for social distancing kdynakoupit.cz, design and implement changes for improving the performance of selected service, and implement machine learning pipeline that ingests, validates dataset, and evaluates the models.

The first part is dedicated to the description of the application structure, where we had to analyze and get to know the application architecture first. It is based on microservices architecture, composed of four independent services. We described the responsibilities and purpose of each service. Furthermore, we researched existing solutions for estimating occupancy levels.

In the next section, we explain the methodology and perform research on how to implement a machine learning pipeline using modern tools and technologies such as TensorFlow. The key benefit of machine learning pipelines is the automation of the model life cycle steps. When new training data becomes available, a workflow that includes data validation, preprocessing, model training, analysis, and deployment should be triggered.

In the third part, we propose a change to separate the responsibilities of the Recommender service to the Recommender service and Predictor service, which would serve as a standalone machine learning backend with a machine learning pipeline (see Figure 4.1). After proposing changes, we described their implementation. We created a separate backend server for the machine learning model called Predictor and achieved separation from Recommender. A Postgres database was also attached to this service, for saving and versioning new datasets and models. The last part of the implementation is creating a machine learning pipeline. I implemented steps for ingesting a new dataset, its validation, preprocessing, and evaluation.

The last part of this thesis deals with the impact of implemented changes.

We discussed the benefits of the implementation such as machine learning service independence and database creation. When implementing a machine learning pipeline, several tools for data validation were useful in determining whether the dataset is, for example, balanced or imbalanced. The results achieved in the evaluation of the model using different classification metrics were also discussed in this section.

The output of this thesis is the overall analysis of the application, implemented and tested the functionality of a new microservice that serves as a machine learning model server.

# Bibliography

[1] Tomáš Vintr etc. Tomáš Krajník. Warped hypertime representations for long-term autonomy of mobile robots [online], 2019, [cit. 2022-01-04]. Available on: https://ieeexplore.ieee.org/document/8754723.

[2] Romana Gnatyk. Microservices vs monolith [online], 3.10.2018 [cit. 2022-01-04]. Available on: https://www.n-ix.com/microservices-vs-monolith-which-architecture-best-choice-your-business/.

[3] Sam Newman. Building microservices, 2021. ISBN: 978-1-492-03402-5.

[4] Docker. What is container? [online], [cit. 2022-01-04]. Available on: https://www.docker.com/resources/what-container.

[5] Docker. Docker overview [online], [cit. 2022-01-04]. Available on: https://docs.docker.com/get-started/overview/.

[6] Kubernetes. What is kubernetes? [online], [cit. 2022-01-04]. Available on: https://kubernetes.io/docs/concepts/overview/components/.

[7] Hannes Hapke and Catherine Nelson. Building machine learning pipelines, 2020. ISBN: 9781492053194.

[8] Scikit-learn. sklearn.preprocessing.onehotencoder [online], [cit. 2022-01-04]. Available on: https://scikit-learn.org/stable/modules/generated/sklearn.preprocessing.OneHotEncoder.html.

[9] Kdynakoupit.cz. How it works? [online], 2021, [cit.2022-01-04]. Available on: https://kdynakoupit.cz/.

[10] Google Support. Manage your location history [online], [cit. 2022-01-04]. Available on: https://support.google.com/accounts/answer/3118687?hl=en.

[11] Matt D'Zmura. Behind the scenes: popular times and live busyness information [online], 2020, [cit.2022-01-04]. Available on: https://blog.google/products/maps/maps101-popular-times-and-live-busyness-information/.

[12] Google policies. How google anonymizes data [online], [cit. 2022-01-04]. Available on: https://policies.google.com/technologies/anonymization?hl=en&gl=de.

[13] Bill Slawski. Popular times for businesses learned by looking at location history [online], 2016, [cit. 2022-01-04]. Available on: https://www.seobythesea.com/2016/12/google-tracking-how-busy/.

[14] Google patents. Point-of-interest latency prediction using mobile device location history [online], 2016, [cit. 2022-01-04]. Available on: https://patents.google.com/patent/US9470538.

[15] About OpenStreetMaps. Openstreetmaps [online], [cit. 2022-01-04]. Available on: https://www.openstreetmap.org/about.

[16] Chronorobotics Laboratory. Fremen contra covid [online], 2021, [cit. 2022-01-04]. Available on: https://chronorobotics.fel.cvut.cz/cs/ai-vs-covid.

[17] Tomáš Krajník. What is fremen? [online], 2021, [cit. 2022-01-04]. Available on: https://github.com/gestom/fremen/wiki.

[18] Rani Osnat. A brief history of containers: From the 1970s till now [online], [cit. 2022-01-04]. Available on: https://blog.aquasec.com/a-brief-history-of-containers-from-1970s-chroot-to-docker-2016.

[19] IBM Cloud. Containerization [online], [cit. 2022-01-04]. Available on: https://www.ibm.com/cloud/learn/containerization.

[20] VMWare. Container orchestration [online], [cit. 2022-01-04]. Available on: https://www.vmware.com/topics/glossary/content/container-orchestration.

[21] Wikipedia. Yaml [online], [cit. 2022-01-04]. Available on: https://en.wikipedia.org/wiki/YAML.

[22] Docker. Docker composer overview [online], [cit. 2022-01-04]. Available on: https://docs.docker.com/compose/#features.

[23] Kubernetes. What is kubernetes? [online], [cit. 2022-01-04]. Available on: https://kubernetes.io/docs/concepts/overview/what-is-kubernetes/.

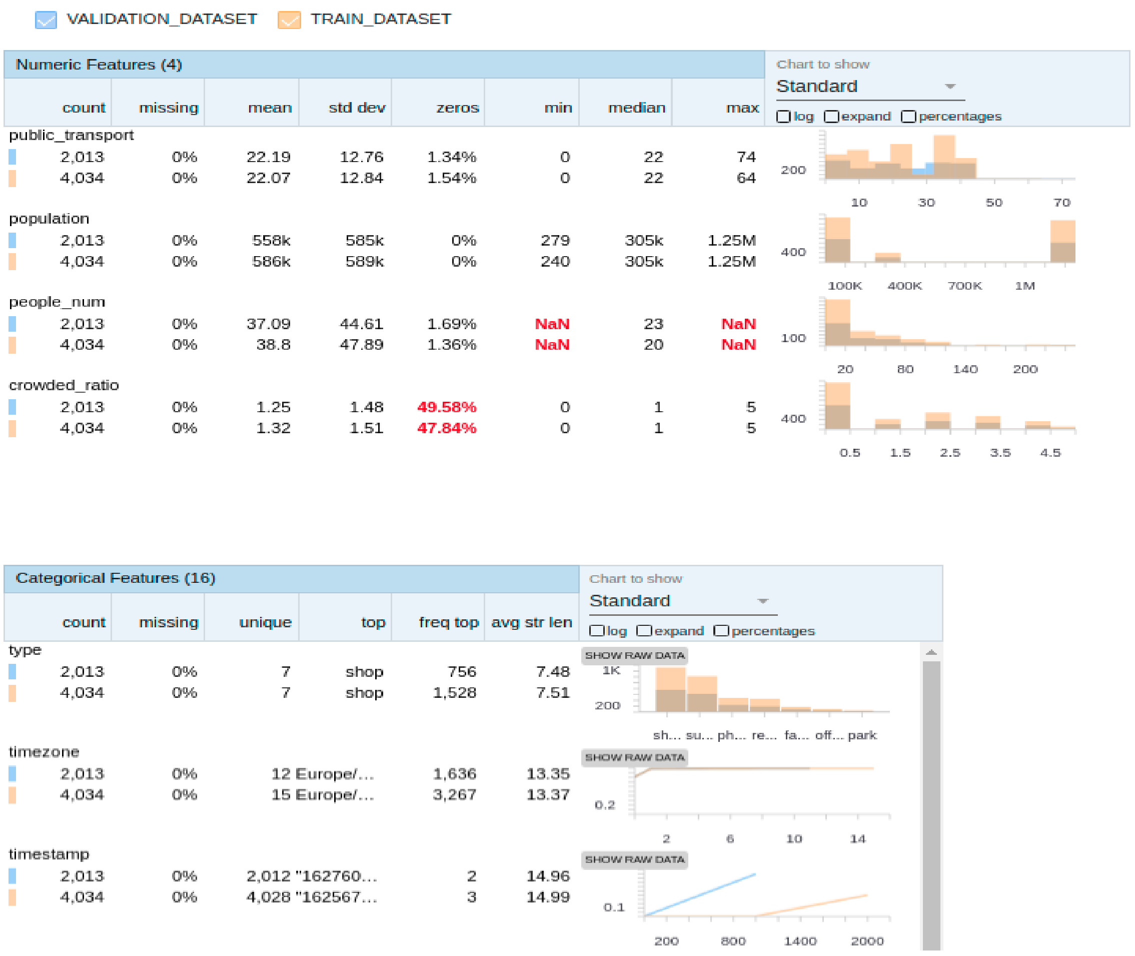
[24] Kubernetes. Glossary kubernetes [online], [cit. 2022-01-04]. Available on: https://kubernetes.io/docs/reference/glossary/?fundamental=true.

[25] TensorFlow. Tensorflow [online], [cit. 2022-01-04]. Available on: https://www.tensorflow.org/.

[26] Telus International. Seven types of data bias in machine learning [online], [cit. 2022-01-04]. Available on: https://www.telusinternational.com/articles/7-types-of-data-bias-in-machine-learning.

# Appendix A

# Additional Figures



🟦 VALIDATION_DATASET 🟧 TRAIN_DATASET

Numeric Features (4)

Chart to show: Standard ☐ log ☐ expand ☐ percentages

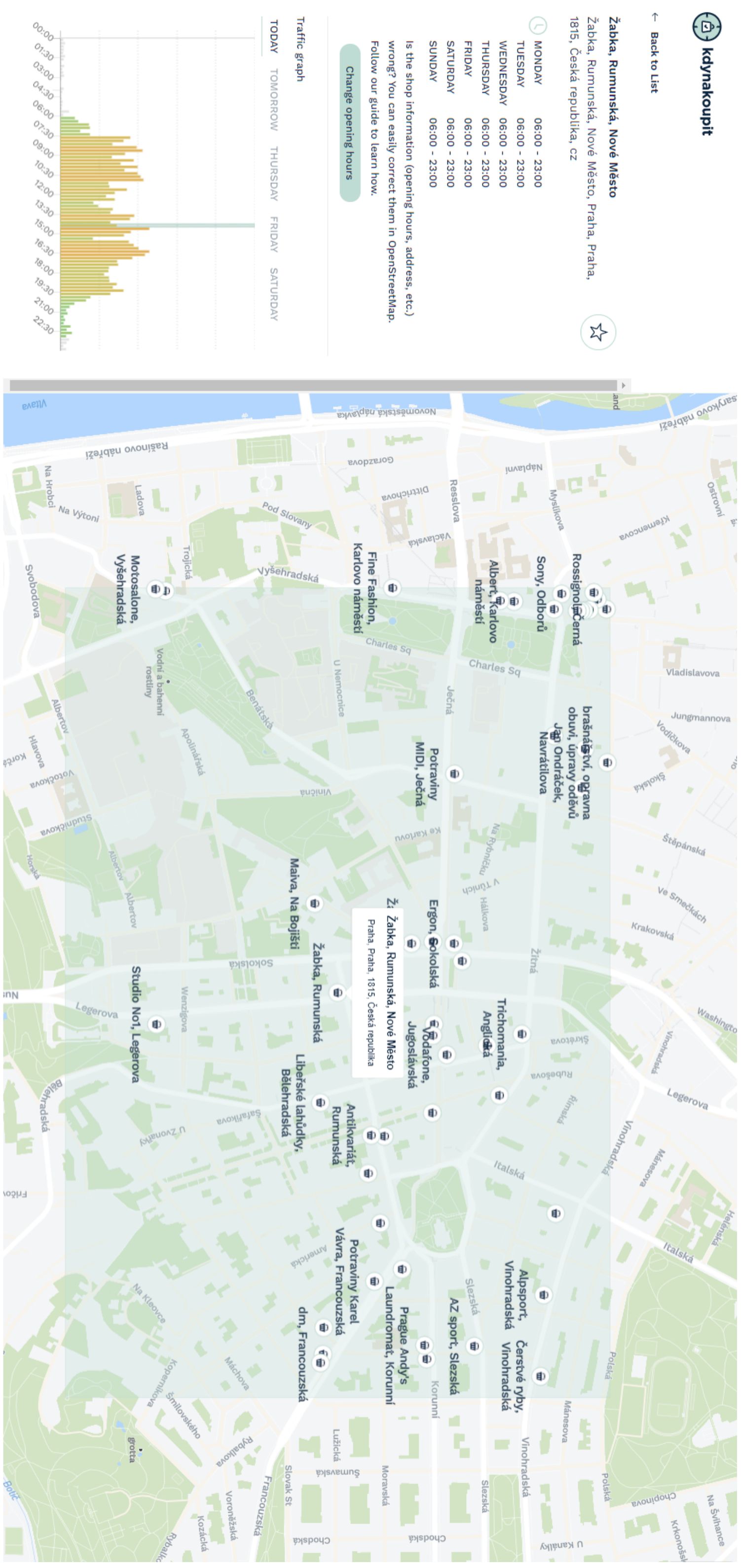
| | count | missing | mean | std dev | zeros | min | median | max |
|---|---|---|---|---|---|---|---|---|
| **public_transport** | | | | | | | | |
| 🟦 | 2,013 | 0% | 22.19 | 12.76 | 1.34% | 0 | 22 | 74 |
| 🟧 | 4,034 | 0% | 22.07 | 12.84 | 1.54% | 0 | 22 | 64 |
| **population** | | | | | | | | |
| 🟦 | 2,013 | 0% | 558k | 585k | 0% | 279 | 305k | 1.25M |
| 🟧 | 4,034 | 0% | 586k | 589k | 0% | 240 | 305k | 1.25M |
| **people_num** | | | | | | | | |
| 🟦 | 2,013 | 0% | 37.09 | 44.61 | 1.69% | NaN | 23 | NaN |
| 🟧 | 4,034 | 0% | 38.8 | 47.89 | 1.36% | NaN | 20 | NaN |
| **crowded_ratio** | | | | | | | | |
| 🟦 | 2,013 | 0% | 1.25 | 1.48 | 49.58% | 0 | 1 | 5 |
| 🟧 | 4,034 | 0% | 1.32 | 1.51 | 47.84% | 0 | 1 | 5 |

Categorical Features (16)

Chart to show: Standard ☐ log ☐ expand ☐ percentages

| | count | missing | unique | top | freq top | avg str len |
|---|---|---|---|---|---|---|
| **type** | | | | | | |
| 🟦 | 2,013 | 0% | 7 | shop | 756 | 7.48 |
| 🟧 | 4,034 | 0% | 7 | shop | 1,528 | 7.51 |
| **timezone** | | | | | | |
| 🟦 | 2,013 | 0% | 12 | Europe/... | 1,636 | 13.35 |
| 🟧 | 4,034 | 0% | 15 | Europe/... | 3,267 | 13.37 |
| **timestamp** | | | | | | |
| 🟦 | 2,013 | 0% | 2,012 | "162760... | 2 | 14.96 |
| 🟧 | 4,034 | 0% | 4,028 | "162567... | 3 | 14.99 |

**Figure A.1:** Generated statistics on training and validation sets


kdynakoupit

← Back to List

**Žabka, Rumunská, Nové Město**

Žabka, Rumunská, Nové Město, Praha, Praha, 1815, Česká republika, cz

| | |
|---|---|
| MONDAY | 06:00 - 23:00 |
| TUESDAY | 06:00 - 23:00 |
| WEDNESDAY | 06:00 - 23:00 |
| THURSDAY | 06:00 - 23:00 |
| FRIDAY | 06:00 - 23:00 |
| SATURDAY | 06:00 - 23:00 |
| SUNDAY | 06:00 - 23:00 |

Is the shop information (opening hours, address, etc.) wrong? You can easily correct them in OpenStreetMap. Follow our guide to learn how.

Change opening hours

Traffic graph

TODAY TOMORROW THURSDAY FRIDAY SATURDAY

Figure A.2: Detailed view on a specific place with traffic graph