YRKESHÖGSKOLAN
NOVIA

Reproducible Machine Learning Models and Experiments

A platform for hosting and managing machine learning projects

Niklas Wallin

Degree Thesis

Thesis for a Master of Engineering (UAS) degree

Automation Technology

Vaasa 2023

DEGREE THESIS

Author: Niklas Wallin

Degree Programme and place of study: Automation Technology, Vaasa

Specialisation: Intelligent Systems

Supervisor(s): Johan Westö, Kaj Wikman

Title: Reproducible Machine Learning Models and Experiments

Date: 10.12.2023 Number of pages: 58 Appendices: 7

Abstract

This thesis concerns finding suitable open-source tools and a platform for managing and hosting machine learning projects. The platform should keep track of the different versions used of the code, datasets, and hyperparameters during the project's life cycle, by using version control tools. The platform should also be able to keep track of the machine learning experiments made when developing a model. By keeping track of what was used for creating a model, it is possible to reproduce older experiments and machine learning models.

The thesis is also about creating a pipeline to run the experiments and create machine learning models. A demo task is included in the thesis, where the pipeline was able to fetch data from the web or use the current data on the PC to create and validate a machine learning model. Finally, the model created is deployed to a website, where it can be tested.

Selected models with good performance metrics can be added to a model registry. This is to better keep track of the state of the selected models e.g., which models should be further tested and which are ready to be deployed to production.

Language: English

Key Words: Machine learning, version control, data version control, model registry, open-source

Table of Contents

Table of Figures

1 Introduction

This master's thesis is part of the master's degree programme in Automation Technology at Novia University of Applied Sciences. The employer for this thesis is the Intelligent Systems Institute at Novia.

The Intelligent Systems Institute brings together knowledge and competence in the field of applied intelligent systems. This is done by bringing together international students and the personnel in the Intelligent Systems Institute. This cooperation tries to respond to different research questions regarding autonomous systems and AI in industrial applications, with expertise in the following areas: AI, machine learning, IoT, data analyses, big data, digitalization, automation, and digital twins. (Yrkeshögskolan Novia, 2023)

1.1 Machine learning

Machine learning is a part of artificial intelligence. Machine learning is used to solve complex problems and make accurate predictions from data. A machine learning model is created by analysing a lot of data. The model is trained on the data to find different correlations between the inputs and the output result. Some of these correlations may even be impossible for humans to find. The accuracy of the models' predictions gradually increases with the amount of data. This is because it has more data to use when learning to find the different patterns between the inputs and outputs. Some common applications, which use machine learning models, are auto-completion of sentences, estimation of travel time, summarization of articles and generation of new images. (Developers.Google, 2023)

Machine learning and artificial intelligence have become more powerful and important over time. This is because machine learning can solve complex problems, which may be too complex for humans to solve. One of the reasons for the rise of machine learning and artificial intelligence is the amount of data available. There are several publicly available databases on the web, as well as corporations who may collect data through websites or IoT devices to their databases. The data can be used for developing a machine learning model to solve or make predictions of a certain task. Another key factor in the popularity of machine learning is hardware availability. The hardware has become more powerful and cheaper. Because of this, it is no longer required to use a super-computer to create

powerful machine learning models. Therefore, it is not only corporations who can create successful machine learning models, but anyone can. There is a wide range of tasks that can benefit or be solved by using machine learning. (Data-flair, 2023; Flovik, 2019)

1.2 Challenges with machine learning

As with any new technology, they also have their challenges. This is also true for machine learning. There are a few challenges, some bigger than others. One of the challenges is the quality of the data, since data is one of the stone pillars of machine learning. If the data is of poor quality, for example, the data collected has a mix of different units for the same thing or the data is not scaled between a max and minimum value. Without standardizing and normalizing the data collected, it will cause the model to be trained and validated against the poor-quality data. This will cause the model to perform worse than it should have if the data was of good quality. (Ataman, 2023)

Another challenge is the over- and underfitting of training data. In simple terms, underfitting means that the model is unable to detect patterns in the training data, which results in bad predictions of real data. Underfitting may be the result of model complexity, feature selection, insufficient training data and preprocessing to name a few possible reasons. In opposition to underfitting, overfitting means that the machine learning model is performing perfectly on the training data but is performing poorly on new data. Overfitting might for instance appear if the model is too complex, uses a low amount of data or is trained with irrelevant features. (Kumar, 2023)

Another big challenge with machine learning is to keep track of all the different versions in the project. When creating a machine learning model there are several different factors, which has an impact on how the model is created and how it will perform. These factors consist of the data used for training the model and the code that specifies how the model is trained along with hyperparameters. In addition to the data, code and hyperparameters, other things might have to be tracked. This can for example be the different model versions, model parameters, model metrics and other potential artifacts. (Barazida, 2021)

1.3 DevOps and MLOps

DevOps and MLOps are two different workflows when developing software or machine learning projects. DevOps is a workflow that can be used when developing traditional software projects. It focuses on assisting the developers with the general software development processes. MLOps is the workflow that brings more benefits when working on machine learning projects, as it focuses on assisting with developing machine learning models. (Maayan, 2022)

1.3.1 DevOps

DevOps is a set of practices and processes that aims to assist the developers through the developing, releasing and monitoring of the product. This is done by close collaboration between the developers and the operations team. There are a few concepts that make DevOps an efficient workflow. The workflow with DevOps consists of four stages for the developers and four stages for the operations team as seen in Figure 1. The first stage for the developer team is to plan a task, this can for example be adding a feature. The next stages are to code the feature and to validate that it is working as planned, before merging the changes into the development branch. It is during the test stage that automated tests are running. This is to quickly make sure that nothing of significance is getting broken before merging the code into the develop branch. Once all planned features have been added to the development branch and all tests have passed, it is time to create a new release and deploy it. The operate stage consists of making sure that the end-to-end delivery to the users is working as well as handling the IT infrastructure required for it. Monitoring is the last stage of the DevOps cycle. This is where feedback about the product is gotten. This feedback helps with future development cycles and improving the product in upcoming releases. (Atlassian, 2023; Gitlab, 2023)

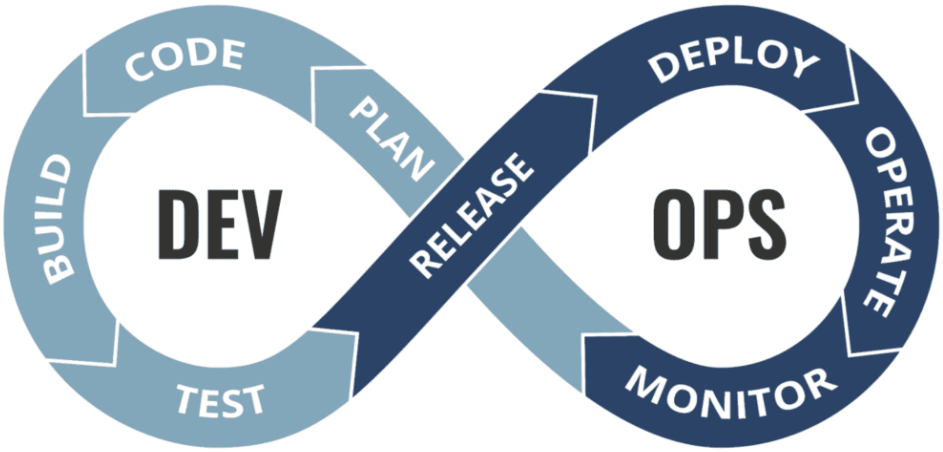

Figure 1: DevOps cycle. (Islam, 2022)

Another big concept with DevOps is automation using pipelines. A pipeline is used for splitting a task into smaller stages, where each stage may use inputs and produce outputs. The output from a stage is usually used in the following stage as an input. This creates a pipeline, where the stages run in a sequence. Once all stages have run successfully, the task has been completed. Pipelines are typically used to replace manual tasks, which also minimizes human-made errors and improves efficiency. Another typical task that gets automated with a pipeline is testing of the software each time a developer adds new changes to the project. This is called continuous integration testing. By running tests automatically each time new changes are added to the project, it increases the chance of detecting flaws and errors quickly. This, in turn, is necessary to maintain a high-quality product and to be able to deliver software releases regularly. The collaboration between the developer team and the operation team contributes to better feedback on the whole project. The feedback helps to further automate tasks to be able to quickly find bugs, improve the code quality and steadily release new versions of the product. (Gitlab, 2023)

1.3.2 MLOps

MLOps, which stands for machine learning operations, is a set of practices and processes that aims to assist with creating and delivering machine learning models into production as well as monitoring their performance. MLOps is a specific implementation of DevOps, which focuses on machine learning tasks and processes. In MLOps, there is a collaboration between the machine learning engineers and the data scientists. Usually, the data scientists

work with the data and develop the model. This includes labelling of the data, data transformation and feature engineering, as well as deciding which algorithm to use in the machine learning model. The machine learning engineer focuses on the deployment and monitoring phase of the machine learning model. MLOps is also used to track different versions of code and artifacts. This includes code, data, hyperparameters, the model and its performance metrics. These artifacts are tracked for each experiment, which makes it possible to compare the performance between experiments and select the most suitable model. (Islam, 2022)

Similarly, to the DevOps cycle, there is also an MLOps cycle, see Figure 2. The main differences between them are the machine learning stages of the cycle, as well as the monitoring stage in the operations section. The data and the model stages consist of algorithm selection, data labelling, feature engineering and data transformation. (Islam, 2022)

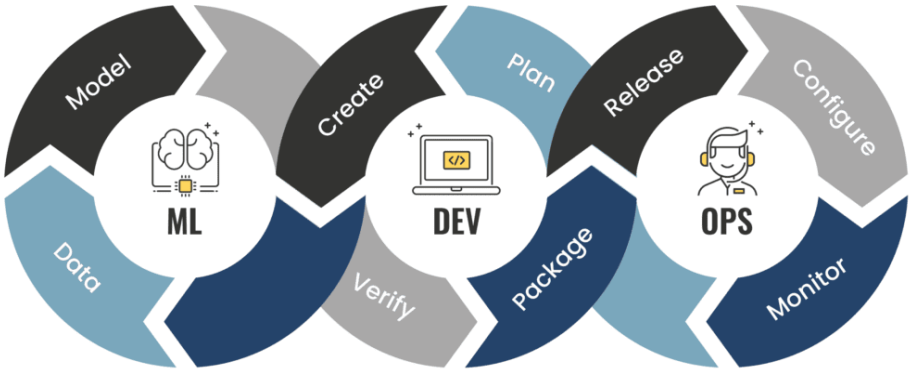

Figure 2: MLOps cycle. (Islam, 2022)

Depending on the algorithm chosen, there might be a requirement to have all the data labelled or partly labelled. During the data transformation, the data gets standardized and normalized. By standardizing the data, each cell of a column in a table is written in the same unit. For example, a column that contains data of lengths. All the values in this column should be written in meters. So, if some cells are written in millimeters, thumbs, or other units, they should either be converted to meters, or the whole row should be removed. By standardizing the data, the data quality is improved. Without standardizing the data, the data is corrupted by the unstandardized values, which will result with a bad performing model. (Islam, 2022; Stitchdata, 2023)

Data normalization is another important part of improving the quality of the data. Normalization of data can include setting limits of the maximum and minimum values of each cell in a column. This is usually done if a column or feature contains values with a big range between the lowest and highest values. By limiting the range of the values of a feature, when a model is created, the feature will not have a bigger impact than other features with a smaller range of values. Normalizing data can for example be done by limiting the minimum and maximum values of a feature between 0 and 1. This can be done with the min-max normalization formula shown in Figure 3. (Liu, 2022)

$$x_{\text{norm}} = \frac{x - \min(x)}{\max(x) - \min(x)}$$

Figure 3: Min-max normalization formula. (Liu, 2022)

Once the model has been released and is in production, it still needs to be monitored. This is done during the monitor stage in the MLOps cycle. One of the reasons why monitoring is important is that a model's performance may decrease over time. This can be because the real-world data may have gradually changed compared to the data that the model was trained on. This phenomenon is called data drift and will result in a decrease in the model's performance, which is known as model drift. Figure 4 shows how the data has changed over time, which decreases the model's performance.

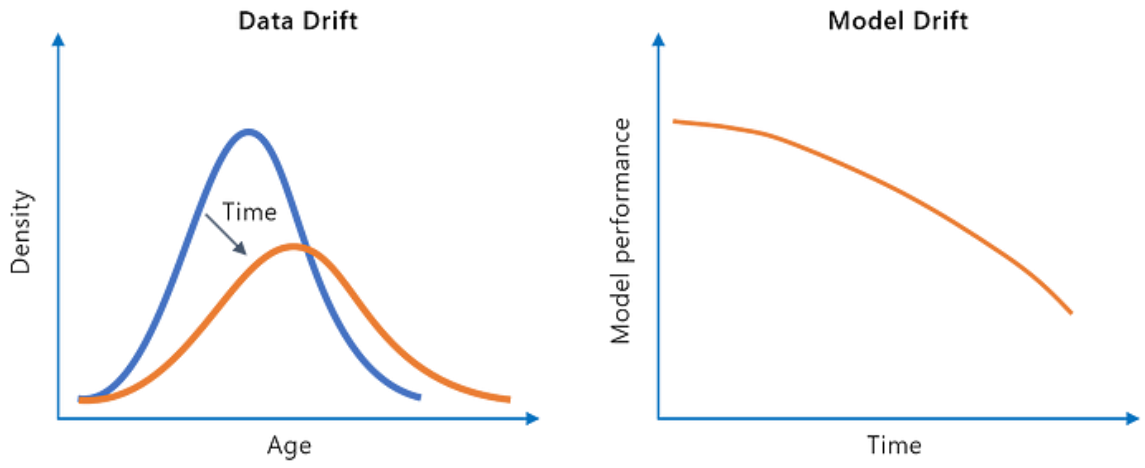

Figure 4: Input data changed over time, causing data drift and model drift. (Sharma, 2023)

By monitoring the model's performance, it is possible to catch the model drift issues on time before the model's performance gets so bad that it is not usable anymore. This is done by continuously developing and making improvements to new models with data that fits the real-world data. A new model can automatically be deployed to production if the current model's performance is decreasing to a certain level. Figure 5 shows how model drift can be solved by retraining and deploying a new model once the performance of the current model decreases over time. (Sharma, 2023)

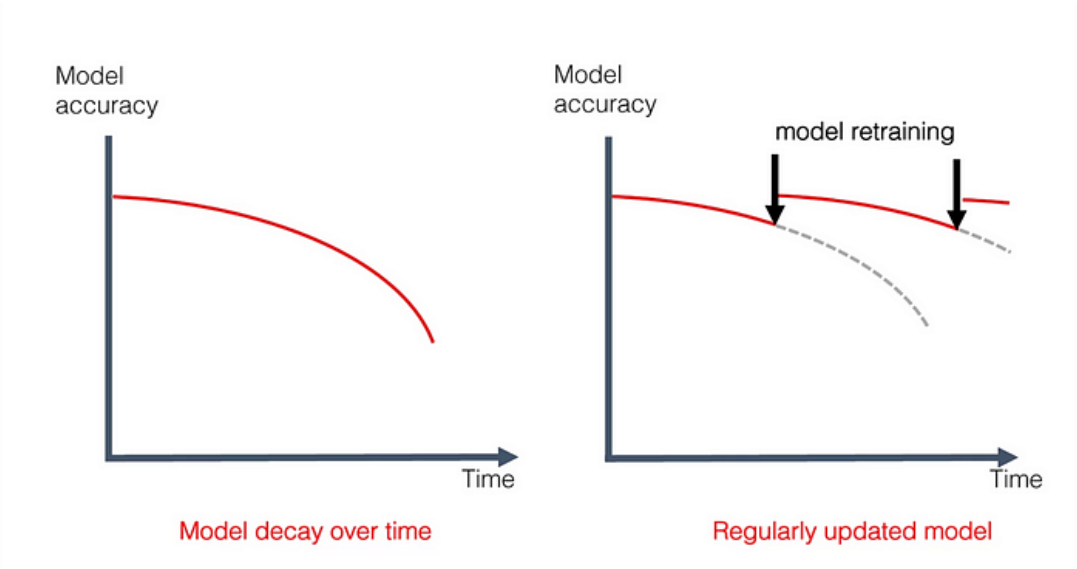

Figure 5: Model drift solved by retraining and deploying a new model. (Sharma, 2023)

Today, MLOps is a crucial set of practices when developing machine learning models. This is to ensure quick experimentation, faster deployment of models to production as well as assuring, through monitoring, that high-quality models are constantly being used. (Sharma, 2023)

2 Version control and model registry

When developing a new product, it is important to track all files used during the whole life cycle of the project. However, it is just as important to track the state and changes made to the files through the whole life cycle of the project. The reason why this is important is to be able to maintain different releases of the product. For example, fixing bugs or

potentially adding additional features to older releases. Tracking the different states of files is done with version control systems. (Şen, 2022)

In addition to tracking files, it is also important to keep track of the different models created and the models that are in production. One of the reasons why this is important is that different versions of the models can be used in different places or applications. It helps with managing the life cycle of the machine learning models produced. (Oladele, 2023)

2.1 Version control systems

Version control systems are a type of software used to keep track of changes made to files. These kinds of tools are common when developing software. Most of the time there is a team of software developers who are making changes to the files in the project. This would be a big challenge if no tool tracks each file's history as well as changes made to the files. The version control system helps the developers and the project to bring in new changes and keep track of who has made changes as well as when the changes were made. (Pubudu, 2021)

With version control systems it is possible to go back and forward to see the project state during a certain time. This is a very useful feature, since it allows the project to compare changes in the files at different times. It also allows the project to roll back to a previous state. This can be used when a fatal bug has been introduced and there is no quick fix for the bug. If this happens, a rollback to a previous working state without the bug can be taken into use instead. There are two main types of version control systems used today, centralized and distributed version control systems. (Chacon & Straub, About Version Control, 2014)

2.1.1 Centralized version control systems

The centralized version control system uses a single server where the whole repository is stored, according to Figure 6. Each user can connect to the server to check out working copies of the files that the user will make some changes to. The connection to the server is required during the whole process of checking out, making the changes and pushing the file back to the server. (Cheshire, 2023; Gitlab, 2023)

While a user has the files checked out on their computer, other users cannot access these files until they have been pushed back to the server. This will help with making sure that there will not be any merge conflicts with this file, since only one user at a time can add changes to it. However, this might result in the project proceeding slower if several people are waiting to add changes to the same file. (Gitlab, 2023; Lamb, 2021)

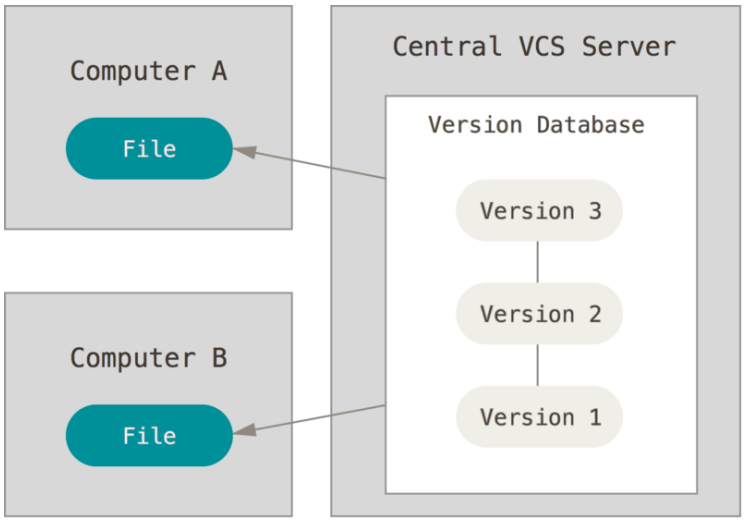

Figure 6: Centralized version control system. (Chacon & Straub, Centralized Version Control Systems)

There are some challenges with centralized version control systems. The main challenge, to be able to work on the project is, that there must be a steady connection to the server. Another challenge or disadvantage is that if there is a power failure and the server goes down, the users cannot fetch new files or push their changes out to the server. This means that it might not be possible to continue to work on the project until the server is up and running again. (Pubudu, 2021)

There is also a big risk when using a single server for storing the entire project. For example, there may be some issues with the hardware or software that might crash or break the server. If this happens without a backup of the project, the whole project will be lost. However, with a backup, it is possible to restore the project to the latest backed up version. (Pubudu, 2021)

Centralized version control systems have started to lose their popularity as the use of distributed version control systems is getting more popular. Some of the more common

centralized version control systems are the following: CVS, Subversion and Perforce. (Gitlab, 2023)

2.1.2 Distributed version control systems

A distributed version control system is quite different compared to a centralized version control system. The main difference is that each user has their own copy of the whole project on their computer, including the history of all the changes made to the project. By having a local copy of the project on the computer, it allows the user to see all the changes ever made to the project and by whom. (Gitlab, 2023)

Even though all users have their own local copy of the project, there is a master version of the project located on a remote server as shown in Figure 7. This is where the users will integrate the changes made to the project. (Intland Software, 2022)

With a local copy of the project on the user's computer, there is no requirement for an internet connection to make changes to the files or to see who has made earlier changes to the project. An internet connection is only required when the user wants to push out their changes or fetch new changes made to the project from the remote server. (Pubudu, 2021)

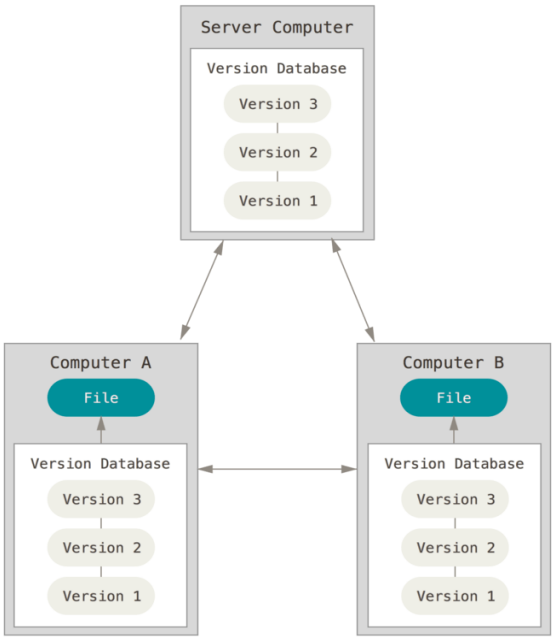

Figure 7: Distributed version control system. (Chacon & Straub, Distributed Version Control Systems, 2014)

There are some benefits with distributed version control systems. The main benefit is that the user has a full copy of the project. This prevents the project from being destroyed if the main server for some reason gets broken and there are no backups available. Since all users have their own copy of the project and its history, the project can easily be restored to the newest or nearly newest state from a user's local copy of the project. (Chacon & Straub, Distributed Version Control Systems, 2014)

Another advantage is speed. There is no need to connect to a server to check out certain files, which the user will make changes to. All files can already be found in the local copy of the project. Another speed related advantage is that the user can instantly run several different commands or operations on their local copy of the project. This can for example be, to create a branch or commit their changes. An internet connection to the server is only required when running commands or operations that fetch the new changes made to the project or pushing new changes made to the project into the remote server. (Pubudu, 2021)

Lastly, when users have created a branch and made some commits to it, the user can push their local branch to the remote server. Even though the branch has been pushed to the server, the branch will not be visible to other users until they fetch the newest state of the project. Also, the branch has not changed the state of the main development branch of the project. To update the development branch, the user will have to merge their own branch into the development branch. When trying to merge the branches, there might be some merge conflicts between them. This happens if both branches have been editing the same file. The user will then have to resolve these merge conflicts before the merge of the branches can be completed. To resolve the merge conflicts, the user must review the conflict and modify the conflict according to what should be merged into the development branch. Should the changes in the new branch be merged or should the code currently in the development branch be kept, or a combination of the changes in the new branch and the code in the development branch. (Pubudu, 2021)

Distributed version control systems have become more popular since it is more powerful and richer in features than centralized version control systems. However, distributed version control systems are most likely more complex than centralized version control systems, which might be more challenging for new users. The most common distributed version control systems are Git and Mercurial. (Cheshire, 2023)

2.2 Version control for code, data and hyperparameters

In machine learning projects there is a need to track several different files and artifacts. This is because machine learning projects are different compared to traditional coding projects. Usually, coding projects consist mainly of text files, which contain the source code of the project. In a machine learning project, there are several different types of files, for example, code, datasets, hyperparameters, metrics and model artifacts. All of which should be tracked during the life cycle of the project. In addition to this, when developing a model, it takes several experiments with small changes to develop the most suitable model for the task it should solve. Each experiment may consist of a change in either the code, the dataset or the hyperparameters. A change in one of these types of files may have a direct impact on how the model will be created or how it will perform. By tracking the different experiments as well as the artifacts and model performance, it is possible to choose the best model once the experiments have been completed. (Hashesh, 2023)

2.3 Model registry

When a good model candidate has been created, it can be placed and stored in a model registry. The model registry is used for managing and tracking the whole life cycle of models. The model registry consists of different stages. At first, the model is placed into a default stage once it is registered. The next stage of the model is to transfer it into the staging stage. This is where the model will be further tested in a similar environment as where it will be used in the end. The model's performance is monitored and validated before it may be transferred into the next stage, called production. At the production stage, the model is ready to be integrated into the application where the model will be used. The previous model at the production stage will be moved to the archive stage. This is where earlier used models are stored for trackability. (Oladele, 2023)

3 Aim of the study

The task for this thesis was received from the Intelligent System Institute. This thesis aims to investigate and find suitable free and open-source tools and a platform for hosting and managing machine learning projects. The tool or the platform should be able to keep track of the different versions of the code, the data, the hyperparameters, the metrics and other artifacts, such as machine learning models.

The task also consists of finding a suitable way to reproduce earlier made machine learning models or experiments. In addition to this, a pipeline, consisting of a few stages should be created. The pipeline should be able to automatically create, train and validate a machine learning model and finally deploy it to an application.

It should also be possible to store models in a model registry to manage and track each registered model's whole life cycle. In the registry, the models can be placed into different stages, such as staging, production and archive. This is to increase the trackability of the state of each model and which stage they are currently in.

A solution to a demo task should also be presented in the study. The demo task consists of creating, training and validating a model and finally deploying it to an application where it can be tested. The model should be created by the pipeline according to the versions of the code, the data, and the hyperparameters. By checking out another version of the code, the data, and the hyperparameters another type of model should be created by the pipeline.

Research objectives

1. Find a free platform for hosting and managing machine learning projects and experiments.

2. Find free and open-source tools that can be used for tracking, code, datasets, hyperparameters and other artifacts, such as machine learning models.

3. Find free and open-source tools for managing and tracking the whole life cycle of models.

4 Tools selected

There are a set of different tools used in this thesis. Git is used for tracking the files containing the code. DVC is mainly used for tracking datasets, experiments and parameters as well as creating a pipeline. MLflow is used for experiment tracking and model registry. Dagshub is the platform selected for hosting the machine learning project and combining the use of Git, DVC and MLFlow. The FastAI Python library is used for creating the machine learning models.

4.1 Git

Git is an open-source distributed version control system. It is used for tracking all changes made to the tracked files in a project. Each Git user has their own copy of the project on their PC, where they can make changes to the files without having a direct impact on the other user's copy of the project. Today, Git is an essential tool for developing software and it is the most used version control system used for new software projects. (Stackoverflow, 2022)

A Git project or repository consists of several branches, where the master or the main branch is where all new changes finally get merged to. When making a change to a project, such as adding a new feature. By convention, a feature branch should be created. This feature branch should consist of all the changes required for the feature. Once all changes for the feature have been implemented into the feature branch, it can be merged into the master branch. Once the feature has been merged, other users can fetch new changes made to the project. This allows the other users to see the new changes made, as well as use them. (Nobledesktop, 2023)

4.1.1 Git under the hood

When creating a Git repository, a hidden directory called ".git" is created in the project's root directory. This is where all the logic of how Git tracks the project is located. The hidden directory contains several different files and directories as shown in Figure 8.

```
$ ls -F1
config
description
HEAD
hooks/
info/
objects/
refs/
```

Figure 8: Files and folders when initializing a git repository. (Git-scm, 2014)

The “config” file contains project specific settings, such as the remote server's address. The “description” file is an extra file that is created by default. However, it is only used by a program called Gitweb, which is a web-based front-end program for Git. The HEAD file contains a reference to the branch or commit SHA-1 hash value that is currently checked out. (Git-scm, 2014)

In the “hooks” directory, it is possible to add scripts that can be automatically triggered when running Git commands. Initially, the directory contains a few sample scripts of the most common Git commands, see Figure 9. These sample scripts are not triggered at all by default. For example, there is a sample script called “pre-commit.sample”. By changing the name of the script to “pre-commit”, it will automatically be triggered before executing the Git command “git commit”. The “pre-commit” script can for instance verify that there are no trailing whitespace changes at the end of each row that has been modified in the staging area. If trailing white space changes are found, the “git commit” command will not pass. It will instead give an error message pointing to the file with the trailing white space. (Atlassian, 2023; Git-scm, 2014)

```
applypatch-msg.sample       pre-push.sample
commit-msg.sample           pre-rebase.sample
post-update.sample          prepare-commit-msg.sample
pre-applypatch.sample       update.sample
pre-commit.sample
```

Figure 9: Sample scripts in the “.git/hooks” directory. (Atlassian, 2023)

The “info” directory contains information about which files not to track with Git. Files that are not wished to be tracked, can be added to a file called “**.**gitignore”. The “.gitignore” file can contain file names, directory names or patterns such as “*.bin”, which would ignore all files that have a “.bin” extension. (Git-scm, 2014)

The “refs” directory contains information about all the branches currently in the repository. For each branch, there is a file which has the name of the branch. The file itself contains the reference to the latest commit of the branch. (Atlassian, 2023)

Finally, the “objects” directory is where all the data for each commit is stored. Every file that is in a commit will be stored as a type called BLOB, binary large object. Every directory in a commit will be stored as a type called tree. The tree may contain BLOB files or other trees, depending on whether the directory consists of files or more directories. The reason why files are stored as BLOB files instead of normal files is to remove all meta-data of the file. Meta-data is always stored in regular files. Meta-data can for example be the name of the file or the date when the file was created. However, a BLOB file does not contain any meta-data. Instead, it contains only the binary data of the file. (Rosenbaum, 2020)

Git uses a SHA-1 hash value to identify each commit, tree and each BLOB file. The SHA-1 hash value is a hexadecimal string of 40 characters that represents 20 bytes. These bytes are generated by the SHA-1 hash algorithm. The SHA-1 algorithm is used by Git to create a unique SHA-1 hash of each commit, each tree and each BLOB file. According to Figure 10, the root directory to the left consists of a file and a directory. The file called “TEST.JS” has a SHA-1 hash value of “F00D1” and the directory called “DOCS” has a SHA-1 hash value of “CAFE7”. The root directory itself has a SHA-1 hash value of 841B9. The “DOCS” directory consists of two files, “PIC.PNG” and “1.TXT”, which has their own SHA-1 hash values of “F92A0” and “73D8A”. (Rosenbaum, 2020)

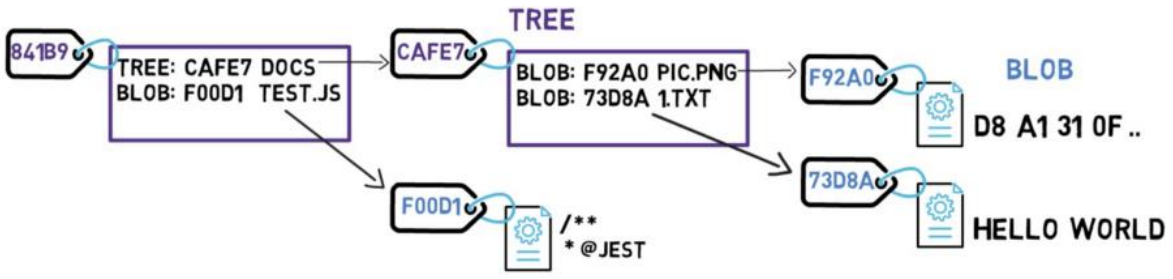

Figure 10: Structure of an example project. (Rosenbaum, 2020)

When creating a commit, Git takes a snapshot of the directories and the files that the user wants to commit. As shown in Figure 11, the commit object has a reference to the root directory in the project. In this case, the directory with the SHA-1 hash value of “841B9”. Along with the reference, the commit object also consists of meta-data, such as the time of the commit, the commit message, the author of the commit and a reference to the parent commit. This is the first commit made to the project, which means that there is no parent commit at this point. The whole commit object has its own SHA-1 hash value, which is “A1337”. (Rosenbaum, 2020)

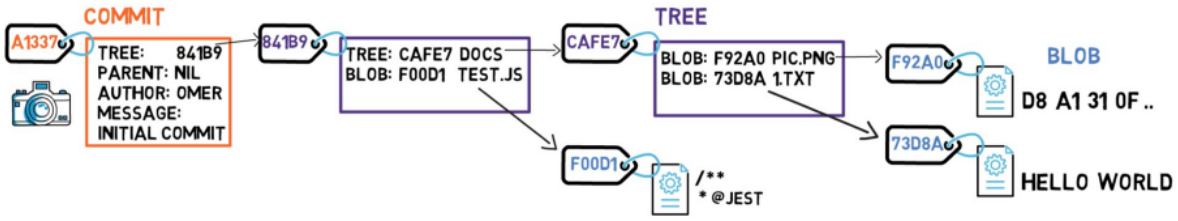

Figure 11: Initial commit of the project. (Rosenbaum, 2020)

The next phase is an example of what happens when a file is changed in the project and the change is added to a new commit. In Figure 12, a change has been made to the file called “1.txt”, which is in the “DOCS” directory. The content of the file “1.txt” has been changed from “HELLO WORLD” to “HELLO WORLD!”. Because of this change, the SHA-1 hash value of the file “1.txt” must be regenerated. The newly generated SHA-1 hash value is “62E7A”. This change will have an impact on the “DOCS” folder where the file was located. This causes a chain reaction on all the directories that will get a newly generated SHA-1 hash value. All the files, that have not been changed, will keep their current SHA-1 hash values. Finally, the commit object will also receive a new SHA-1 hash value. (Rosenbaum, 2020)

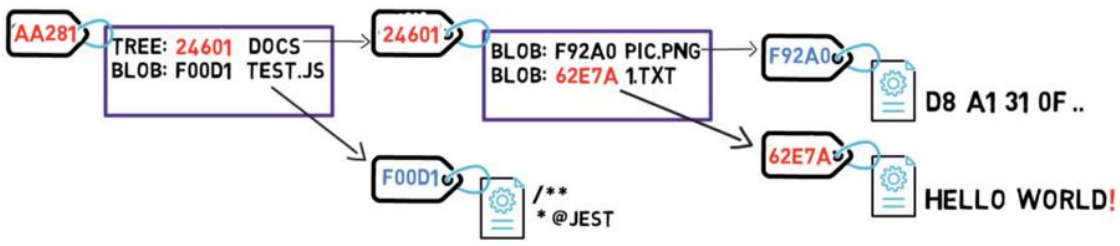

Figure 12: Chain impact of new SHA-1 hash values generated. (Rosenbaum, 2020)

Finally, when creating the commit with the modification to the file “1.txt". The commit will get a reference to its parent commit, according to Figure 13. The files or directories that did not get a new SHA-1 hash value will not be added to the new commit object. Instead, the new commit will use the untouched files and directories SHA-1 hash values from the previous commit. By doing this, each unique version of a file or directory will just be stored once. This saves storage by not storing several versions of the same file or directory. (Rosenbaum, 2020)

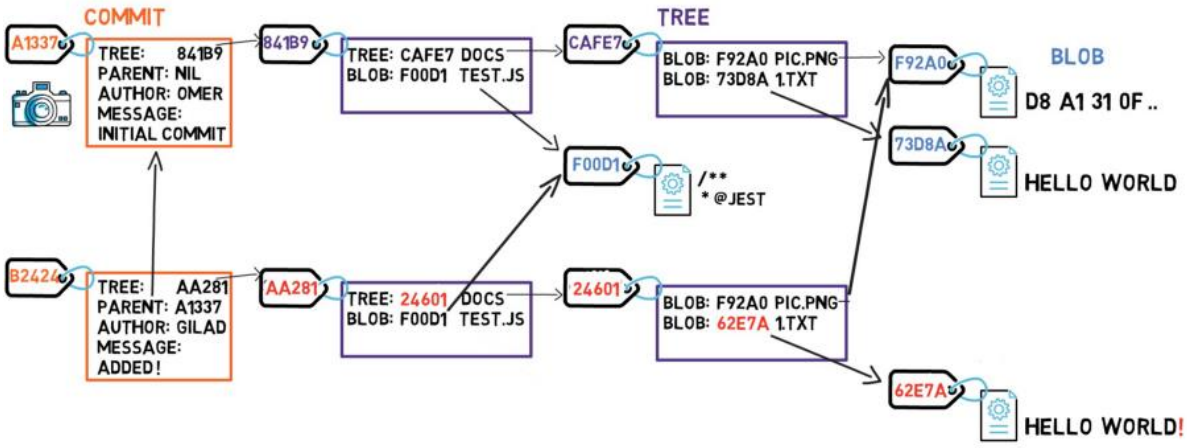

Figure 13: Cross reference from older commits. (Rosenbaum, 2020)

4.1.2 Git workflow

There are several workflows or ways on how to work with Git, more specifically how to use the branch feature in Git. Gitflow is a workflow of standardizing how to work in a project when it comes to branch usage in the project. Usually, there are at least two branches, the main branch and the develop branch, see Figure 14. The main branch is the branch that is used for creating official releases of the software or product. The develop branch is where developers add changes to the project in between releases, such as new features. (Atlassian, 2023)

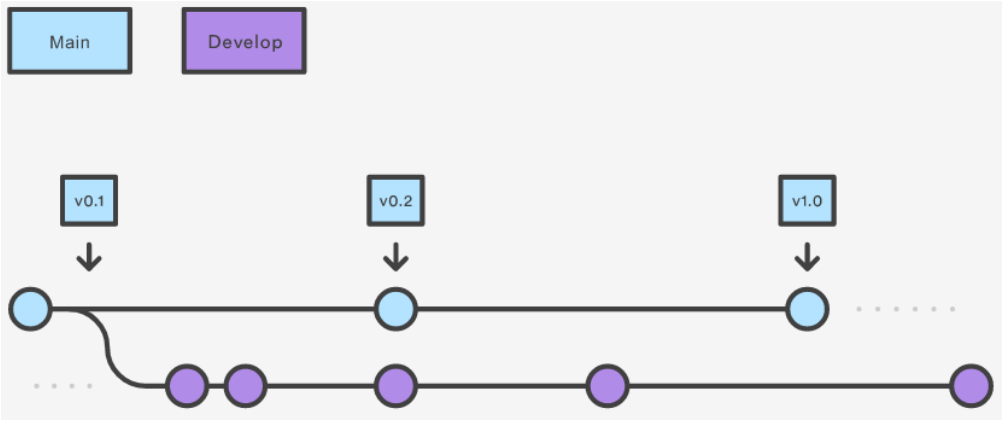

Figure 14: Gitflow, main and develop branch. (Atlassian, 2023)

Usually, developers do not directly work on the develop branch when adding their changes. Instead, a feature branch is used. A feature branch is a branch that is based on some commit in the develop branch, usually the newest commit at the time when the feature branch was created, but it is not mandatory. As the name says, a feature branch is used by developers when they are adding a new feature or other changes to the project. Once the feature has been developed in the feature branch, the branch will be merged into the develop branch. As shown in Figure 15, there are usually several different feature branches at the same time, since each developer that is adding new changes or features has their own feature branches. (Atlassian, 2023)

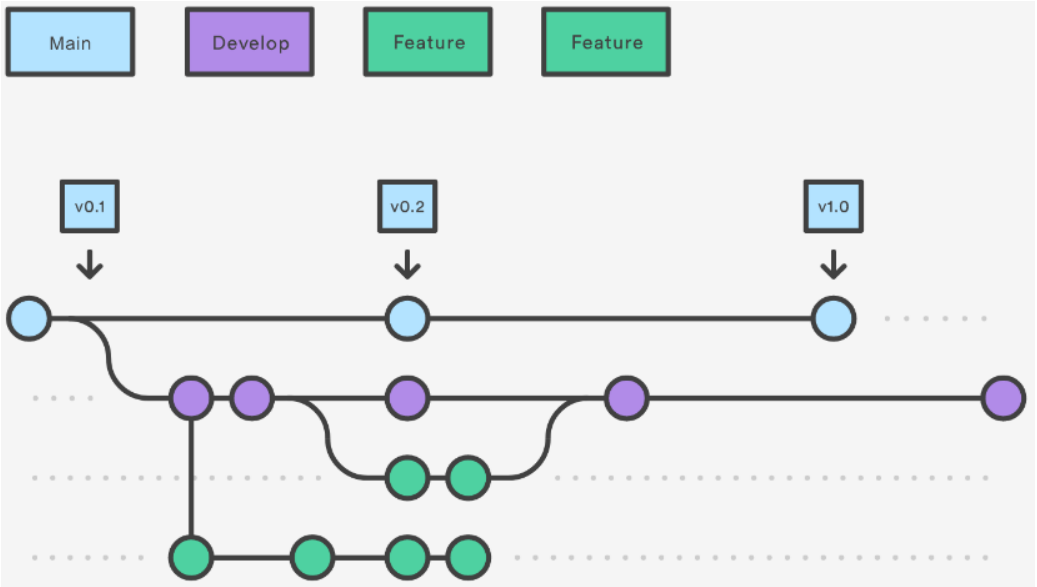

Figure 15: Gitflow, main, develop and feature branches. (Atlassian, 2023)

When a set of features has been merged into the develop branch, it may be time to add the new features in the develop branch into the main branch. In other words, publish a new version of the software or product. Instead of merging the develop branch with the main branch directly to accomplish this, a release branch can be used, see Figure 16. (Atlassian, 2023)

When creating a release branch, a code freeze usually takes place as well. The code freeze means that no more features should be added to the release branch. This is to minimize the risk of new bugs getting into the release, which could cause delays of the new release. During the code freeze, only additional documentation and potential bug fixes could be added to the release. Once the release is proven to be stable and ready, the release branch gets merged with the main branch, creating a new version of the software or product. After this, the release branch also gets merged with the develop branch. This is to add the additional commits after the code freeze, e.g., documentation and bug fixes. (Atlassian, 2023)

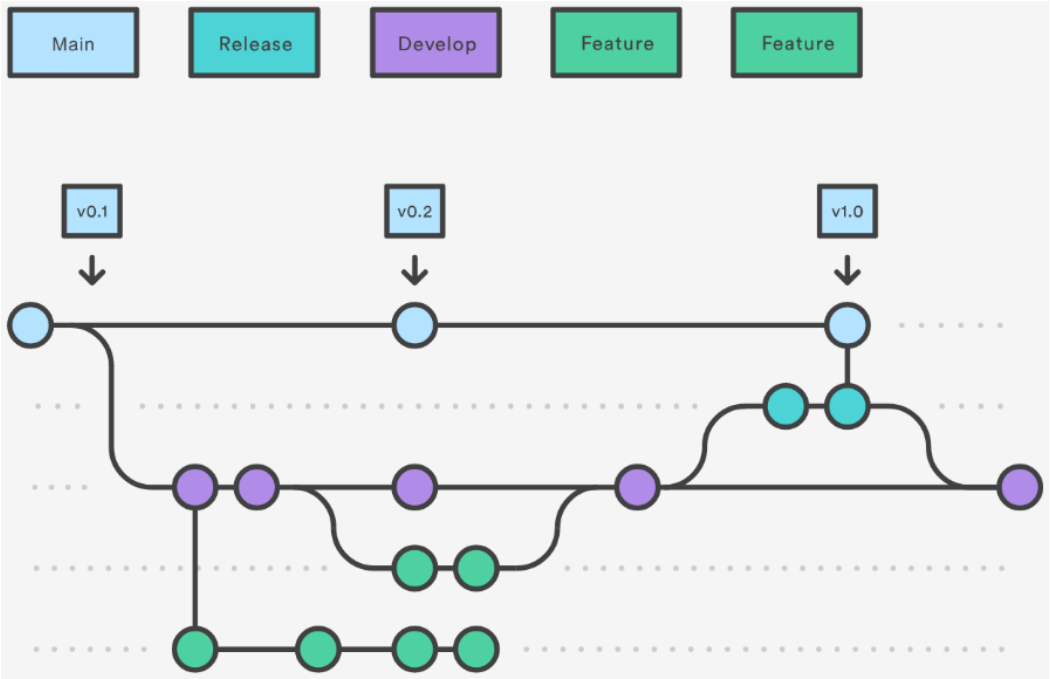

Figure 16: Gitflow, main, release, develop and feature branches. (Atlassian, 2023)

Now there is still one type of branch that is left in the Gitflow workflow, it is the hotfix branch. The hotfix branch is a branch for fixing bugs or issues that have been published through the main branch. In other words, it fixes some issues in a version of the software

or product that potential customers are using. The hotfix branch is created from the main branch where the issue has been found, see Figure 17. Once the issue has been fixed in the hotfix branch, the hotfix branch gets merged into the main branch, the develop branch and the current release branch if there is one at the time. When merging the hotfix branch into the main branch, a new version of the main branch has been created, which means that a new version of the software or product has been made. (Atlassian, 2023)

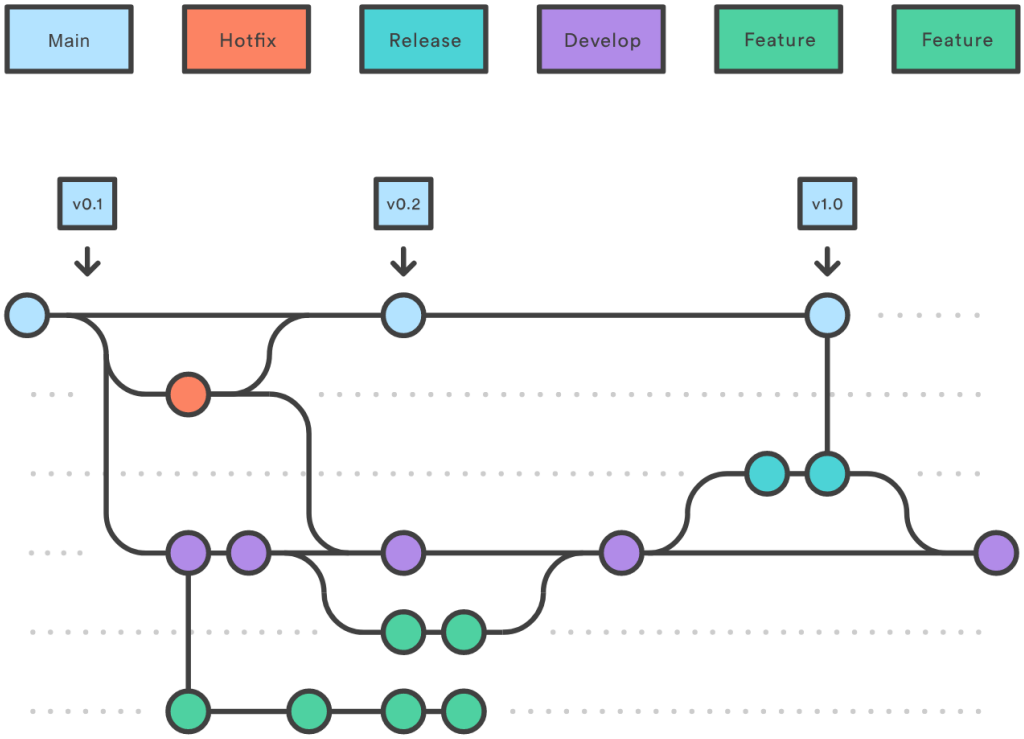

Figure 17: Gitflow, main, hotfix, release, develop and feature branches. (Atlassian, 2023)

4.2 DVC

DVC, which is short for data version control, is an open-source version control tool for tracking large files, machine learning experiments, creating pipelines and model management. The tool was created to help machine learning engineers and data scientists manage their different data versions, experiments and models. DVC can be configured to store the tracked data and models at different cloud services, on the local machine or other devices through the SSH protocol. Another key feature of DVC is the possibility to create pipelines. The pipelines can for example be used for reproducing machine learning models, which consist of several different stages. The pipeline will run the different stages one after the other to complete the experiment. (Dvc, 2023)

4.2.1 DVC data versioning workflow

The first action needed to be able to start tracking data and other artifacts with DVC, is to initialize DVC in a Git repository. The initial state of the repository is shown in Figure 18. This command will generate a hidden folder called ".dvc". This folder contains the configuration of DVC as well as a folder called cache. (Ivancic, 2020)

To initialize DVC, run the following command in the root folder of the Git repository.

```
dvc init
```

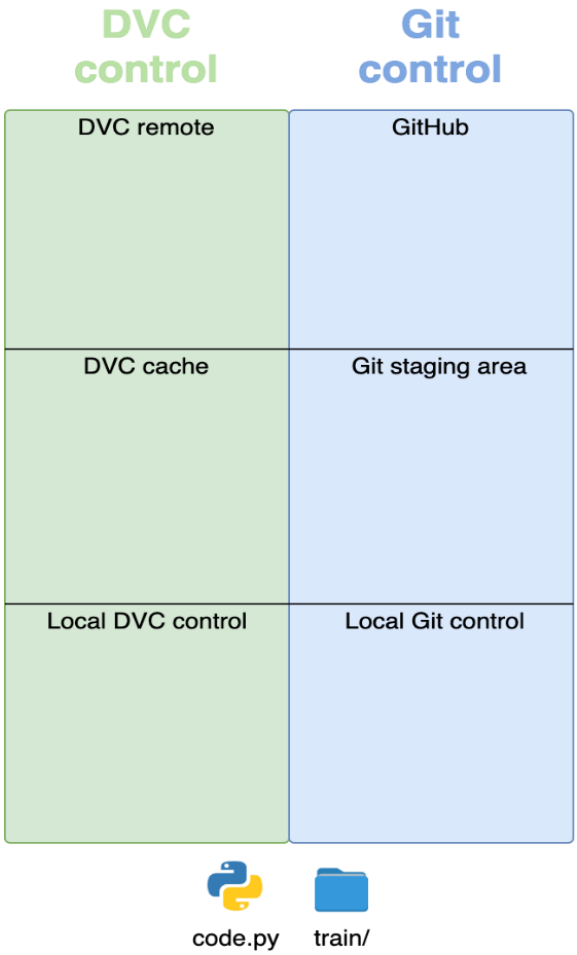

Figure 18: Initialized Git and DVC repository. (Ivancic, 2020)

Once DVC has been initialized, it is possible to configure the remote storage of DVC. The remote storage can be located on a cloud service, on a server accessible with SSH or on the PC itself. To setup the remote with the default remote settings, run the following command:

```
dvc remote add -d <remote_name> <path_to_remote>
```

Now, DVC has been setup and is ready to start tracking files and artifacts. To track a file or a folder with DVC, run the following command:

```
dvc add train/
```

Once this command has run, it will do a few things. First, a copy of the file or folder to track will be placed in the DVC cache, which is located at ".dvc/cache" in the root directory of the repository. The cache folder contains the different versions of the files tracked by DVC. The reason for keeping a copy of the tracked files in the cache, is to quickly be able to use different versions of a file without having to check out the file from the remote storage. This can take several minutes depending on the number of files and their sizes. It is however not mandatory to keep copies of the tracked files in the cache directory. If the file version that should be used cannot be found in the cache directory, it will be fetched from the remote storage instead. (Ivancic, 2020)

DVC will also generate two files, if they are not present. The first generated file will have the same name as the tracked file or folder with an additional file extension of ".dvc" appended to the file. The generated file is a text file containing the information about the file tracked by DVC. The file is written with YAML syntax and contains key-value pairs with information about the file added by DVC, see Figure 19. One of the keys is called "md5", where the value is the hash value of the file or folder added by DVC. The hash value is used for identifying the version of the file or folder tracked by DVC. Another key is the "path", which contains the relative path to the file or folder tracked. The "description" key contains the description of the file or folder tracked. The remote key contains the remote name for pushing or fetching the file or folder from the remote storage. It is also possible to add custom key-value pairs to the file. The custom key-value pairs should be added to the meta stage. (Dvc, 2023; Ivancic, 2020)

```
outs:
  - md5: a304afb96060aad90176268345e10355
    path: data.xml
    desc: Cats and dogs dataset
    remote: myremote

# Comments and user metadata are supported.
meta:
  name: 'Devee Bird'
  email: devee@dvc.org
```

Figure 19: Sample "data.xml.dvc" file. (Dvc, 2023)

The second file that is generated is a file called ".gitignore". This file contains names of files or folders which Git will ignore to track. In this case, the ".gitignore" will contain the name of the file or folder that was added by DVC. This will prevent Git from tracking the dataset file or folder, which should only be tracked by DVC, as shown in Figure 20.

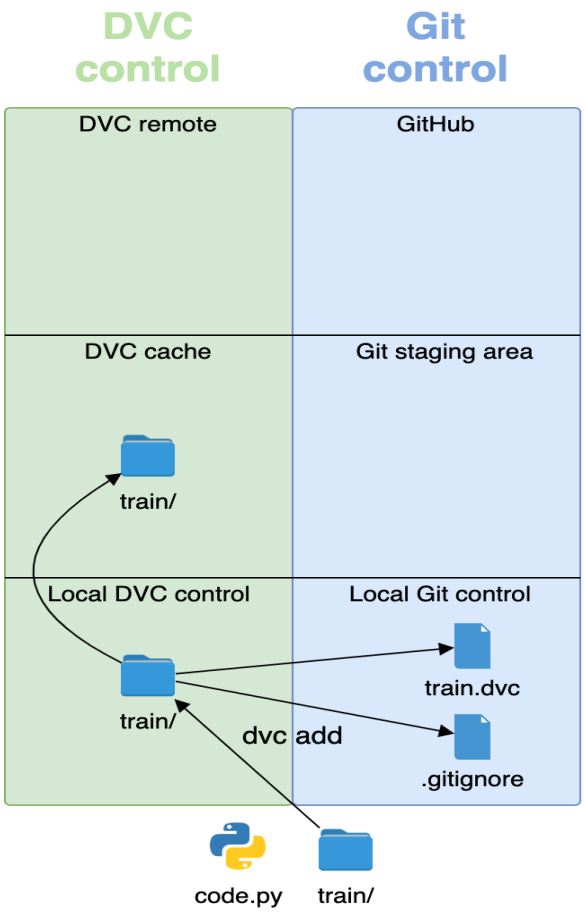

Figure 20: Repository state once DVC added the train folder to be tracked. (Ivancic, 2020)

After DVC has added the files to track, Git can now add the rest of the files, including the generated files with an extension of ".dvc". When adding files to track with Git, they are moved to Git's staging area, ready to be committed, see Figure 21. (Ivancic, 2020)

To add the files to track and commit the changes with Git, use the following commands:

```
git add  train.dvc
git add .gitignore
git add code.py
git commit -m "Added code.py, train.dvc and the .gitignore."
```

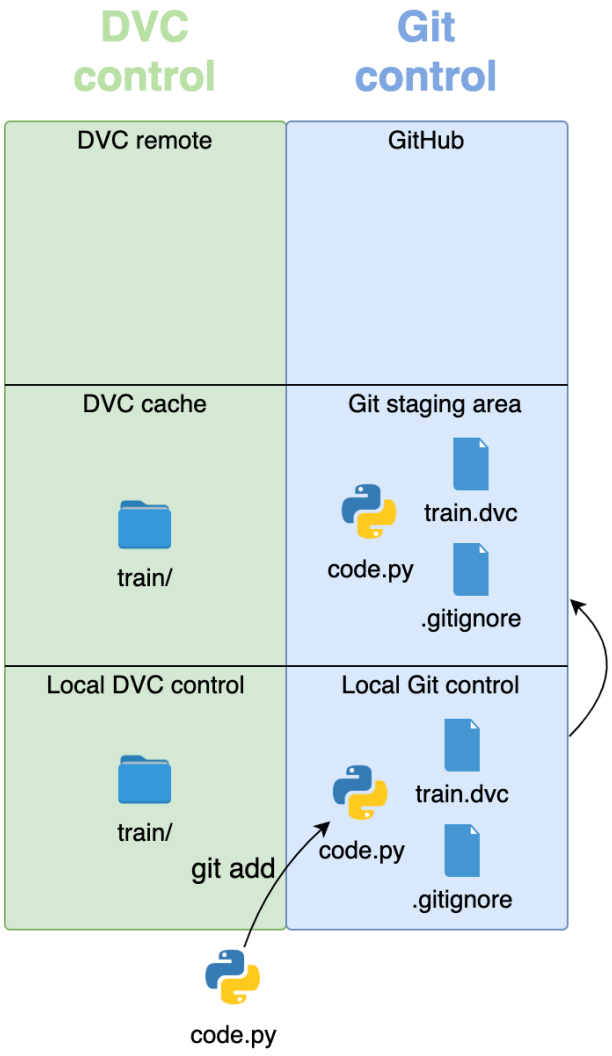

Figure 21: Repository state after Git started to track the rest of the files. (Ivancic, 2020)

Once the files have been added either by Git or DVC, it is possible to push these changes to the remote storage, as shown in Figure 22. Once the files have been pushed to the remote storage everyone with access to the repositories can download the same versions of the files that got pushed to the remote storage. (Ivancic, 2020)

To push the files to the remote storage of DVC and Git use the following commands:

```
dvc push -r <remote_name>
git push origin <branch_name>
```

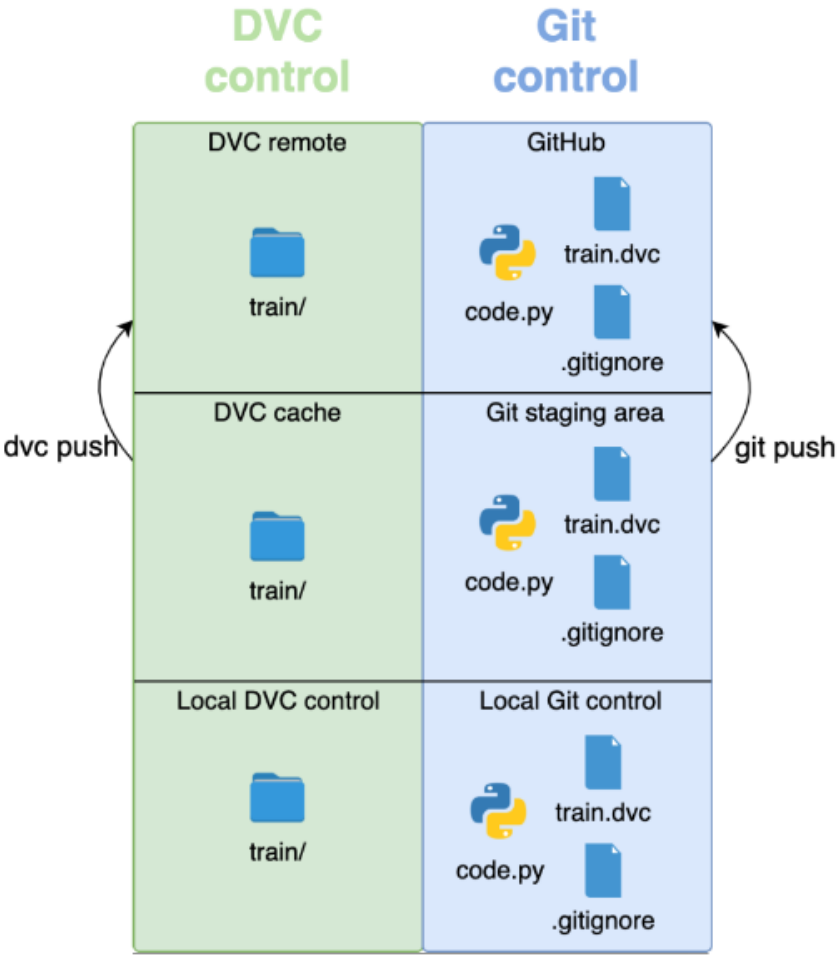

Figure 22: Repository state after pushing changes to the remote storage. (Ivancic, 2020)

At this point, all the files in the repository can now be found either in the DVC remote storage or in Git's remote storage. Let's say that another user has created another branch where more training data has been added and some changes to the code have been made. To be able to use the new dataset and changes to the code. The new changes must first be

fetched. The fetch command fetches all new changes made to the repository, but it does not check out anything to the current branch in use. The “git checkout” command is used to check out the new changes of a specific commit or branch. At this point, it is important to fetch and check out the new changes with Git first. This is because Git will check out the code as well as the “.dvc” files, which are used to describe which version of the data DVC should check out. If DVC was to run its fetch and checkout command first, it would have checked out the current version of the dataset instead of the new data. (Ivancic, 2020)

To fetch and check out the changes with Git, run the following commands:

```
git fetch
git checkout <branch_with_new_changes>
```

After this run the DVC fetch and checkout command or the pull command for doing both the fetch and checkout with a single command:

```
dvc fetch && dvc checkout
or
dvc pull -r <remote_name>
```

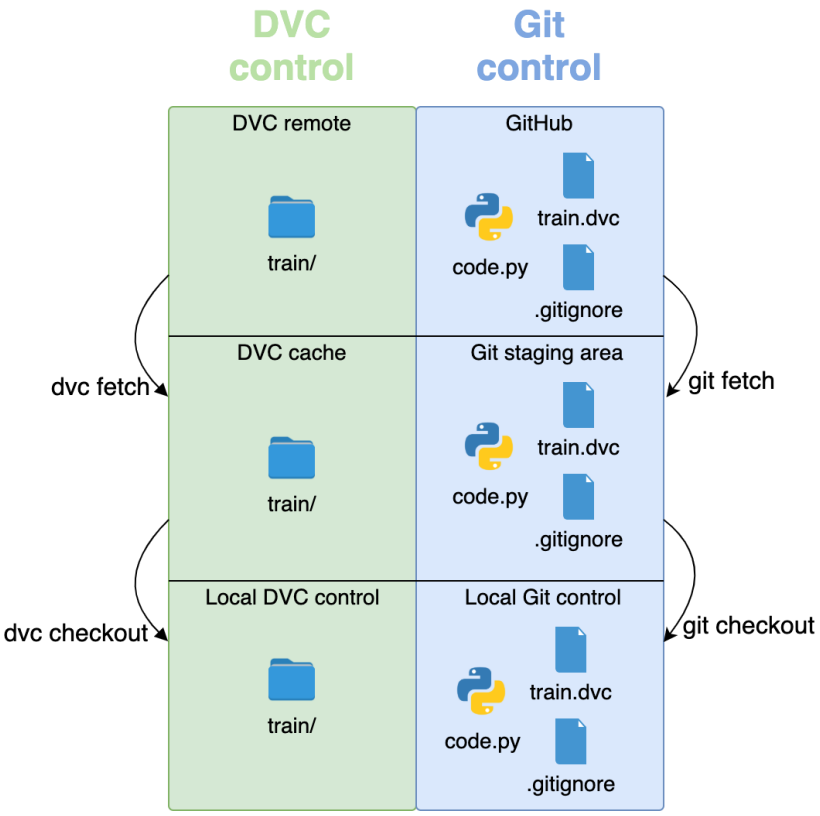

Figure 23: Fetching files from the remote storages. (Ivancic, 2020)

After the new changes have been fetched and checked out, according to Figure 23, the project will have the same versions of the files according to the new branch, which has added more training data and made some changes to the code. (Ivancic, 2020)

4.2.2 DVC pipeline workflow

Creating a machine learning model consists of several stages and a lot of experimentation. These stages can for example be cleaning data, transforming the data, training the model and validating it. By using a pipeline, it is possible to automate the different stages, which will in turn ease the experimentation part. Each of the stages can be split into its own dedicated Python script and executed automatically, one after the other. This workflow can be done by using DVC pipelines. (Feki, 2023)

The pipeline configuration is defined in a file called ”dvc.yaml”, see Figure 24, in the root of the repository. This is where the different stages of the pipelines are defined along with their inputs, outputs, dependencies, parameters and the command the stage will run. Usually, in a pipeline, the stages create some sort of output, which the next stages are depending on. These outputs are then defined as dependencies in the stage, which are using it. The idea with this is that if a stage fails and therefore does not produce the output, it will cause the other stages not to run. (Feki, 2023)

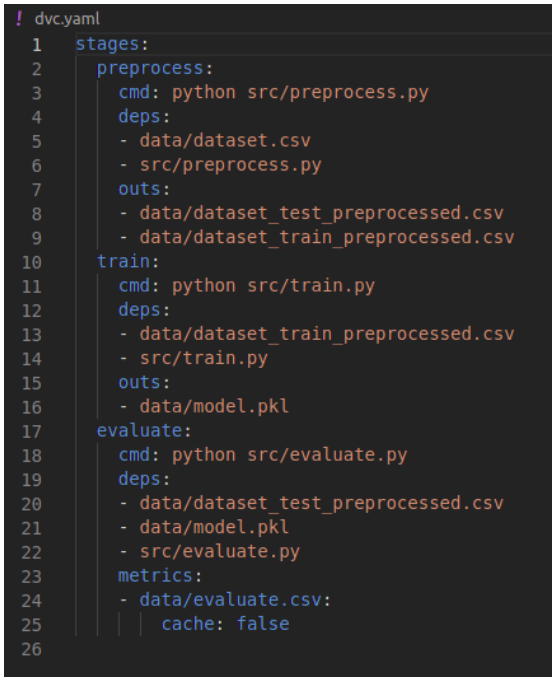

Figure 24: Sample “dvc.yaml” file. (Štetić, 2022)

Another feature of the DVC pipeline is that it knows which stages are required to run. DVC does this by tracking everything that is required for each stage. Each time the pipeline is triggered, DVC calculates the md5 hash value for everything required per stage, which usually consists of inputs, outputs, parameters, dependencies and the script to execute. All of this information is stored in a file called “dvc.lock”, see Figure 25. This file is updated after each stage when the pipeline is running. By tracking this information, DVC knows which stages are required to run and which stages can be skipped. For example, a pipeline consisting of five stages, where the fourth stage fails because of a typo in the Python script that the stage runs. Once the typo is fixed and the pipeline reruns, DVC will notice that the first three stages have not been modified since the last pipeline run. However, DVC will notice that the Python scripts in the fourth stage have been modified. Because of this, the pipeline will skip the three first stages and start running the pipeline from the fourth stage. (Feki, 2023)

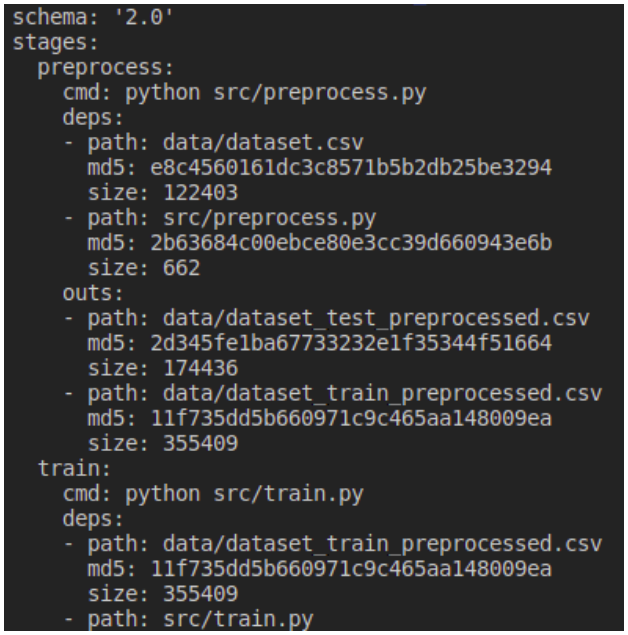

Figure 25: Sample “dvc.lock” file. (Štetić, 2022)

The parameters that are defined in the stages of the pipeline are parameters that are used in the Python script executed for the stage. DVC has a dedicated file for the parameters, called “params.yaml”, see Figure 26. It is a convention to use this file for setting the values of the parameters instead of hard coding the parameters and their values in the pipeline stage. In the parameters file, it is possible to set general parameters and stage specific parameters for better maintainability. (Feki, 2023)

```yaml
base:
  project: bank_customer_churn
  random_state: 0

data:
  raw_data_dir: data/raw
  data_file_name: Churn_Modelling.csv
  cat_cols:
  - Geography
  - Gender
  num_cols:
  - CreditScore
  - Age
  - Tenure
  - Balance
  - NumOfProducts
  - HasCrCard
  - IsActiveMember
  - EstimatedSalary
  target_col: Exited

data_split:
  processed_data_dir: data/processed
  test_size: 0.2

train:
  model_dir: models
  model_type: XGBClassifier
  train_params:
    learning_rate: 0.2
    max_depth: 5
    n_estimators: 200

eval:
  model_path: models/model.pkl
  reports_dir: reports
  metrics_fname: metrics.csv
```

Figure 26: Sample "params.yaml" file. (Feki, 2023)

4.3 MLflow

MLflow is an open-source tool for managing the life cycle of machine learning models. MLflow consists of four different components: tracking, projects, models and model registry, according to Figure 27. Each of the components has its own features and capabilities. The MLflow tools can run on the local machine or online on different cloud services. (MLflow, 2023)

Figure 27: MLflow core components. (Patel, 2023)

4.3.1 MLflow tracking

The tracking component is used for logging metrics, parameters, code versions and output files of the machine learning code as well as the start and end time of the experiment. These logged metrics and artifacts are used to show the performance of the model. MLflow tracking can for example be used with Python through the Python mlflow library. The library provides different logging functions, such as autolog, which automatically logs all artifacts, metrics and parameters it is capable of logging. All the experiments that have been tracked can be visualized and managed in the MLflow user interface through a web browser. The experiments can be stored both locally on the PC, or on a cloud service provider. (MLflow, 2023)

4.3.2 MLflow projects

MLflow projects is a component, which assists with organizing machine learning projects in a standardized way based on conventions. By using the MLflow projects convention of organizing and packaging the files in a project, it is possible to easily setup virtual environments for your projects and reproduce machine learning models. All of this is done through a YAML file called “MLproject.yaml”, according to Figure 28. This file describes the specifications of the project and how it is set up. What kind of environment should be set up and which dependencies are required for the code to run. It also defines different stages, like a pipeline, where each stage may have its own set of parameters, inputs, outputs and the script file to be executed. By executing the YAML file, it will automatically set up the environment for the project and install the required dependencies and execute the pipeline. Once the pipeline run is completed, a machine learning model should have been created. By using MLflow projects, it is possible to reproduce the whole project and machine learning model automatically through a single command. (MLflow, 2023)

```yaml
name: My Project

python_env: python_env.yaml
# or
# conda_env: my_env.yaml
# or
# docker_env:
#    image:  mlflow-docker-example

entry_points:
  main:
    parameters:
      data_file: path
      regularization: {type: float, default: 0.1}
    command: "python train.py -r {regularization} {data_file}"
  validate:
    parameters:
      data_file: path
    command: "python validate.py {data_file}"
```

Figure 28: Sample "MLproject.yaml" file. (MLflow, 2023)

4.3.3 MLflow models

The MLflow models component is used for packaging machine learning models in a standardized format, for the model to be used with different services, such as a REST API. One of the key features of the models component, is that it is capable of packaging the model so it can be used by different languages, or flavors, as it is called in MLflow. All the most popular flavors, such as sklearn, fastai, keras, tensorflow, pytorch, to name a few, are all supported. It is also possible to use the model directly through Python, as a function call. When packaging a model, a YAML file called "MLmodel.yaml" is generated, according to Figure 29. This file contains information about which flavors can use the model. (MLflow, 2023)

```yaml
time_created: 2018-05-25T17:28:53.35

flavors:
  sklearn:
    sklearn_version: 0.19.1
    pickled_model: model.pkl
  python_function:
    loader_module: mlflow.sklearn
```

Figure 29: Sample "MLmodel.yaml" file with two flavors. (MLflow, 2023)

4.3.4 MLflow model registry

The model registry component is used for storing and versioning models produced and managing them in a centralized storage. Once model experimentation has provided a suitable model that could be used for deploying to production, it is possible to register that version of the model to the MLflow model registry. By registering a model, it is easier to maintain and manage the different versions of the models. Registered models can be stored in different stages, such as staging, production or archive, as shown in Figure 30. The staging stage is usually used for further testing of the model before finally moving it to the production stage. Once, the model is moved to the production stage, it is ready to be deployed into the final product. The archive stage consists of model versions that have been in the staging or production stage but have been placed in the archived stage since the model did not pass the testing or the model has been deprecated by newer model version. (MLflow, 2023)

Figure 30: MLflow model registry with the registered models. (MLflow, 2023)

4.4 Dagshub

Dagshub is a web platform for hosting and managing machine learning projects. Figure 31 shows the initial view of a project. The platform is used to allow machine learning engineers to cooperate and collaborate on machine learning projects. The Dagshub platform supports the use of Git, DVC and MLflow components for tracking the different versions of the code, datasets, experiments, models and other artifacts. (Dagshub, 2023)

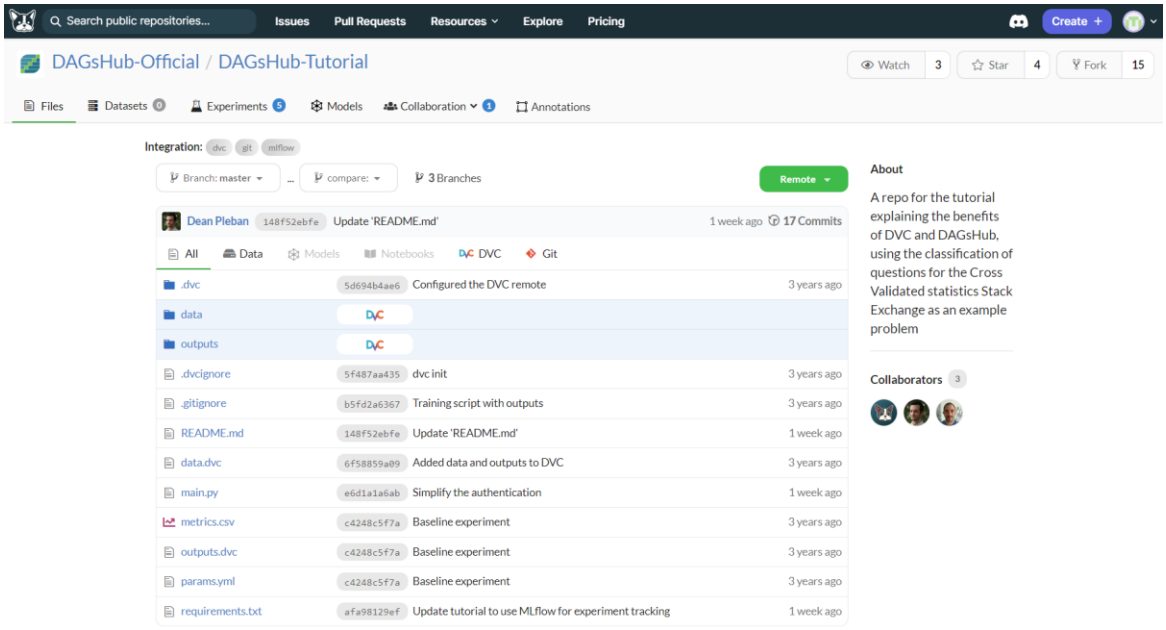

Figure 31: Initial view of a project hosted on Dagshub. (Dagshub, 2023)

4.4.1 Experiments view

Dagshub has a dedicated view of experiments made. This is where all the experiments are stored. There is an overview of each experiment, such as the Git commit used, when the experiment was created and a few hyperparameters and metrics, as shown in Figure 32. It is also possible to label experiments to be able to filter the view according to the label of different types of experiments made. (Saboo, 2021)

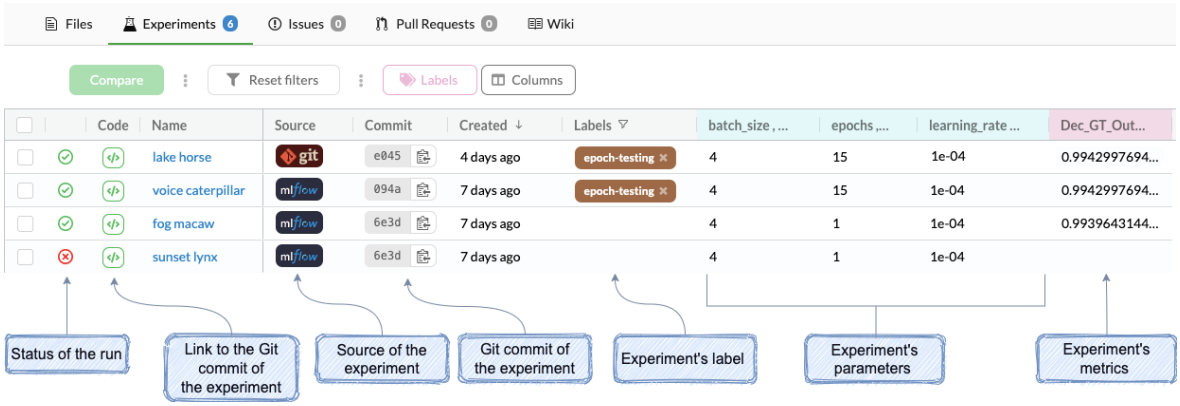

Figure 32: Dagshub experiment view. (Dagshub, 2023)

The experiment view also supports the possibility of looking at each experiment individually or selecting multiple experiments and comparing their differences and results, as shown in

Figure 33. This may help the user to figure out the different behaviours between the experiments and select the best model or further add improvements for the next set of experiments. (Saboo, 2021)

Experiment ID:	0ad1dce6eb	6c5b4ec4fb	af94ca334b
Author:	Tolstoyevsky	Tolstoyevsky	Tolstoyevsky
Created:	14 days ago	14 days ago	14 days ago
Commit Message:	Lower LR=0.002	Even lower LR	Run with default params
Parameters			
max_nb_epochs	2	2	2
batch_size	32	32	32
learning_rate	0.002	0.0002	0.02
Metrics			
loss	0.007823823019862175	0.013818355277180672	0.03044593334197998
epoch	1	1	1
avg_val_loss	0.07953067123889923	0.16655410826206207	0.2730430066585541

Figure 33: Comparison between three different experiments. (Dagshub, 2023)

4.4.2 Pipeline overview

If the project contains a DVC pipeline, an overview of the pipeline is shown on the project page. The pipeline view gives a good overview of the different stages in the pipeline, as shown in Figure 34. Here, it is also possible to see each stage's inputs and outputs as well as how the stages are connected. The overview also shows which version control tool is managing the files used in the pipeline. This can be very useful for users who are new to the project. (Dagshub, 2023)

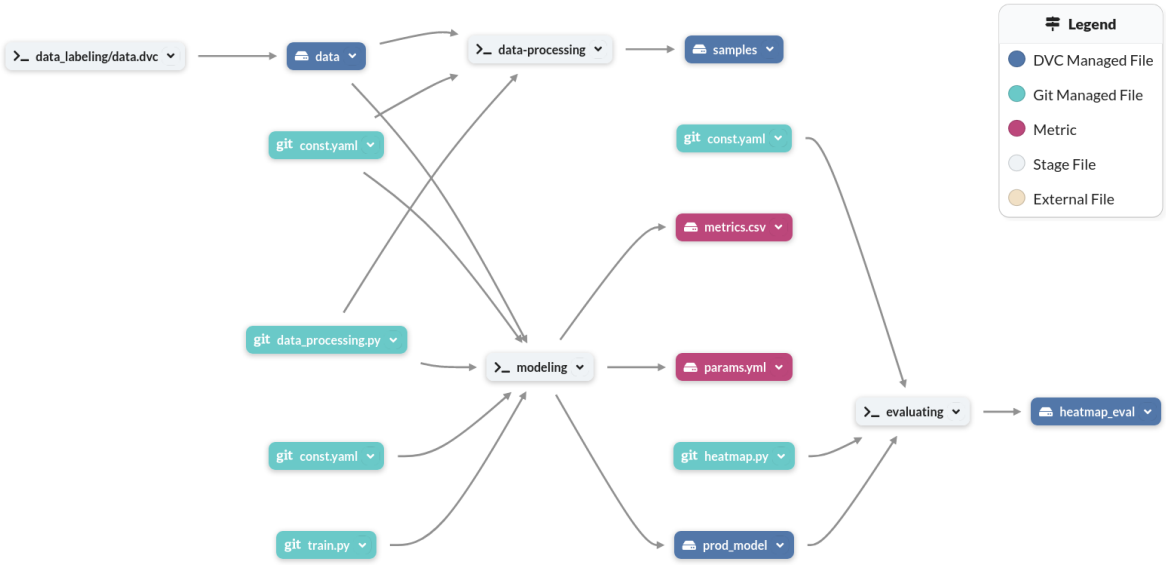

Figure 34: Overview of a DVC pipeline. (Dagshub, 2023)

4.5 FastAI

FastAI is an open-source deep learning library for creating machine learning models. The FastAI library focuses on productivity and ease of use. According to Figure 35, the architecture of FastAI is built in several different layers to allow different kinds of users to benefit from the library and create successful models. For example, it is possible to create high performance machine learning models by only using the high-level functions of the library. The high-level functions will take care of the lower lever function calls. This may limit the configurability of how the model is created, but it allows users with limited knowledge to create powerful machine learning models. However, it is not limited to only being used as a high-level deep learning library. It is also possible to use the low-level functions, which may increase the configurability of how the model is created. (Howard, 2023)

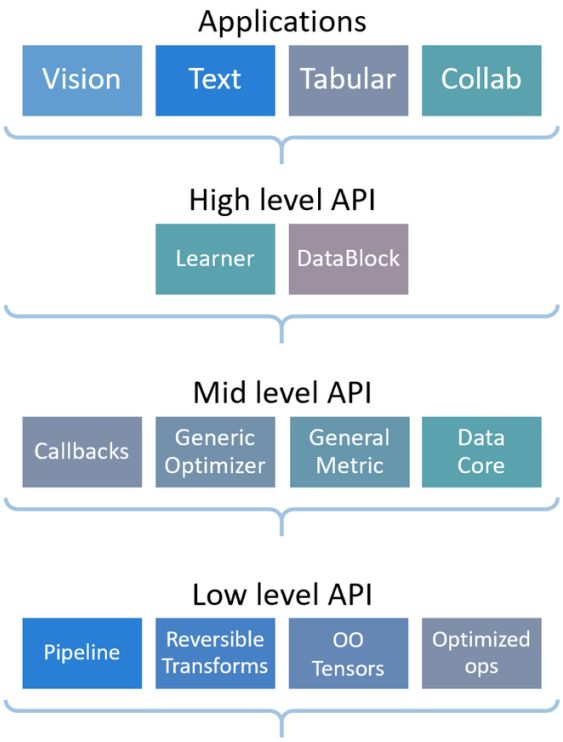

Figure 35: FastAI layer architecture. (Howard, 2023)

The FastAI deep learning library is common for a set of different applications such as computer vision, medical imaging and language processing. FastAI also provides a set of pre-trained models for different applications. These models can be fine-tuned on datasets to increase the accuracy on the real data. By using a pre-trained model, it is possible to quickly develop a high performing model instead of having to develop it from scratch. (Frąckiewicz, 2023)

5 Result

This chapter consists of two sub-chapters. The first chapter gives an overview of the task to solve. The second chapter describes how the task was approached and which steps were taken to solve the task.

5.1 Demo task

To better demonstrate how the Dagshub platform works with data science and machine learning projects, a demo task was to be included in the thesis. A suitable solution for the demo task should be found and implemented. The task consists of creating a pipeline, which can create a machine learning model to make valid predictions according to the task it has been trained for. It also consists of keeping track of the different versions of the code, the data, the hyperparameters and the experiments used for creating a model as well as tracking the model artifact and metrics. With the different versions tracked, it should allow the users to check the performance of all the models and experiments created. It should also be possible to check out an earlier version of the project and recreate a model according to the versions of the code, the data and the hyperparameters used during that time. When a model has been created, it should also be possible to deploy it somewhere so that a user can test the performance of the model in a real-world application.

5.2 Demo task solution

The following chapters will describe which steps were taken to complete the demo task and which tools were used to accomplish this. The first thing that was required was to find a suitable task that could be solved by using machine learning. The task selected was image classification. The plan for the demo task is to create and train a machine learning model capable of predicting what is shown in an image, according to the class labels the model has been trained on. The model will be deployed to a website so that a user can test the model and see the result of the prediction made by it. The user would give the model an image as input and the model would predict what is shown in the image. The website would show the image that the user gave as input to the model. The model's prediction would be shown as a label displayed under the image and a confident score of the model's prediction. The score would indicate how sure the model is that the prediction is correct.

5.2.1 Setting up the repository

The first step was to create a new repository on Dagshub for tracking the project. A private repository called ImageClassification was created with the available "Jupyter Notebook + DVC" template, as shown in Figure 36. The template sets up the initial folder structure of the project. It also contains a sample pipeline and a sample Jupyter notebook which contains a few cells that explain the projects' structure and the workflow of DVC. The initial file and directory structure were quite good, but small modifications were made while the project proceeded.

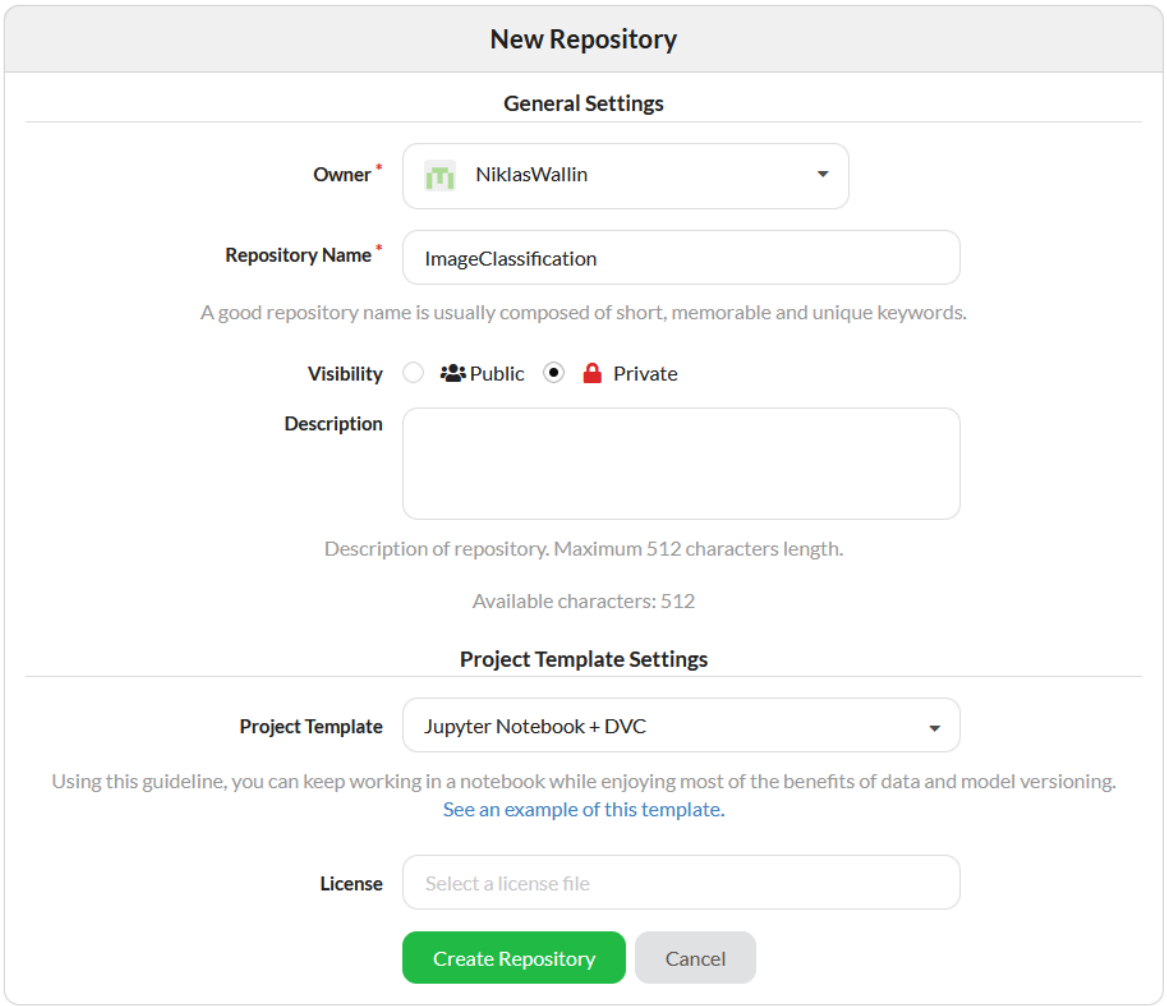

Figure 36: Creating the ImageClassification repository on Dagshub.

Once the repository was created on Dagshub, it was possible to clone it to the PC by using Git. This was done with the following command:

```
git clone https://dagshub.com/NiklasWallin/ImageClassification.git
```

The next stage was to create a virtual environment to be used when working with the repository. The virtual environment was set up and configured with Conda and the Python version selected was “3.9”. With the virtual environment activated, it was possible to start installing the required Python packages, without affecting or installing packages to the default environment of the PC. With the virtual environment activated, DVC was installed with the following command:

```
pip install dvc
```

Once DVC was installed, it was possible to initialize DVC in the cloned repository. This was done by executing the following command in the root directory of the cloned repository:

```
dvc init
```

After the DVC was initialized, it was possible to set up the remote and the required credentials for pushing and fetching data from the remote. The required remote address and credentials are provided under the remote button at Dagshub, as shown in Figure 37.

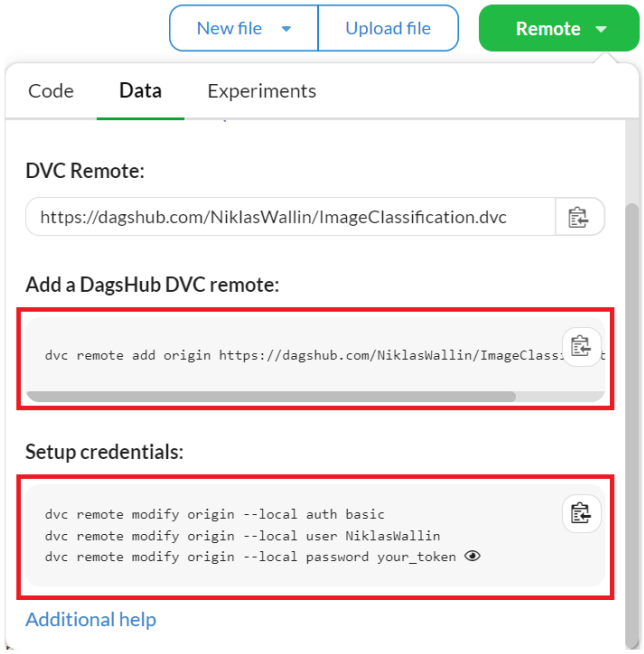

Figure 37: Commands of the DVC credentials and remote.

The following commands were executed for adding the DVC credentials and setting up the remote in the DVC configuration file:

```
dvc remote modify origin --local auth basic
dvc remote modify origin --local user NiklasWallin
dvc remote modify origin --local password <TOKEN_VALUE>
dvc remote add origin https://dagshub.com/NiklasWallin/ImageClassification.dvc
```

The final stage of setting up the repository was to install and configure MLflow. To install the MLflow Python library, the following command was executed:

```
pip install mlflow
```

Once MLflow was installed, Dagshub required a few variables to be set before it was possible to log experiments and the experiment's metrics, parameters and artifacts to MLflow. Dagshub provides the variables and their values from the repository remote, as shown in Figure 38.

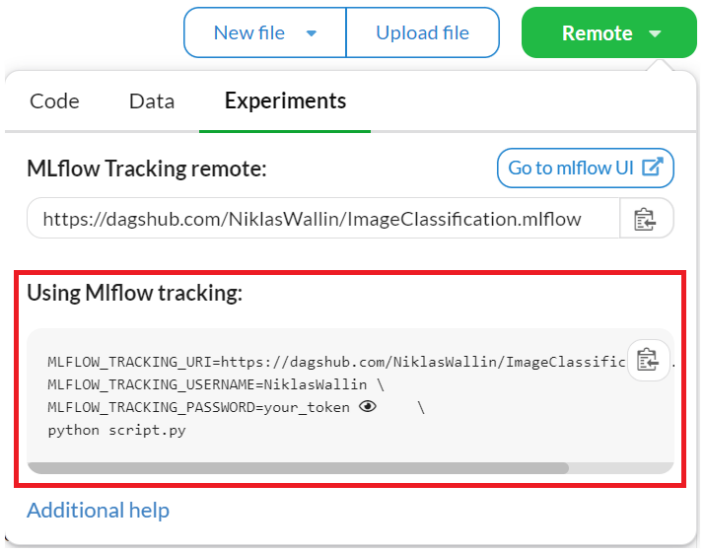

Figure 38: Variables required for MLflow logging.

These variables were added as environment variables to the virtual environment in order for the variables to only be available when the virtual environment was activated.

```
MLFLOW_TRACKING_URI=https://dagshub.com/NiklasWallin/ImageClassification.mlflow
MLFLOW_TRACKING_USERNAME=NiklasWallin
MLFLOW_TRACKING_PASSWORD=<TOKEN_VALUE>
```

At this point, the configuration of the repository is completed. It is now possible to push and fetch changes made to the project. This includes code and data and logging experiments with MLflow. Figure 39 shows the initial view of the project once it was set up.

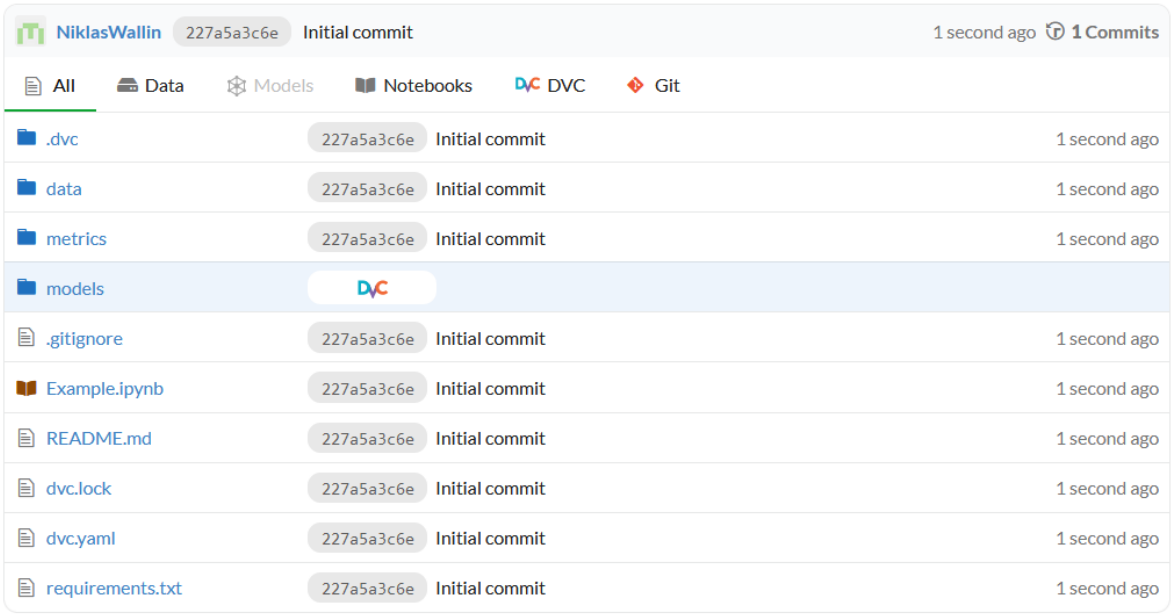

Figure 39: Initial state of the ImageClassification repository.

5.2.2 Initial prototyping

Jupyter notebook was used for the initial development of the machine learning model. The reason for this is that during the early stage of development, the code is executed several times with small modifications made between the iterations. Jupyter notebooks allows the user to split code into different sections, called cells. Each cell can be triggered on its own or select multiple cells to run multiple cells. By doing this, it is easy to follow the code and see the values of the variables. It is also possible to make changes in a specific cell and just rerun that cell to instantly get the new variable values. During the development phase, a few different libraries for making image classification models were investigated and the FastAI library was selected. The reason for this was that the library was quite popular for these kinds of tasks.

The main parts of the project were first developed in a notebook. This included downloading images from Bing according to the class labels the user has put in. Once the images have been downloaded to the computer, the notebook will use FastAI´s

augmentation capabilities to artificially grow the size of the dataset. Next, a pre-trained Resnet18 model was fine-tuned on the dataset created. The fine-tuned model was validated on the test data and its performance was shown in a confusion matrix. Once the Jupyter notebook was able to create a solid model, capable of solving image classification tasks, it was time to split the notebook into smaller scripts for further development and improvements.

5.2.3 Python scripts

Once the initial development and testing of the model creation, training and validation were completed in the notebook, it was time to implement the similar logic into dedicated Python scripts. This is to be able to maintain the code more easily in their own dedicated scripts, which can run through the terminal. The scripts were designed in a way that each Python script produces an output that the next script uses. This was decided for it to be easier to implement a DVC pipeline at a later stage. More on this in the next chapter. In total, there are four Python scripts used in the solution. Below is a brief explanation of each of the scripts.

The ”prepare_data.py” Python script, Appendix D, is used for preparing the data used for creating the model. The script can delete the current images and download new ones from Bing according to a query containing the class labels, if specified to do so through an argument. If there is no need to download new images, the script will proceed with the current dataset and create data-blocks of each class label from the current data available. The data-blocks are used to artificially grow the amount of data, by using augmentation. Once the augmentation is done, a data-loader object is created and saved to the PC. The data-loader object is used as an input in the following script, for creating the machine learning model.

The “modeling.py” Python script, Appendix E, is used for creating the model as well as validating and logging its performance. The script takes the data-loader object as input from the previous script, which is required to be able to create the model. The script starts with activating MLflow’s auto-log logger. The logger is used for tracking how the model is performing during validation and the parameters used when creating the model. The logger is also used for logging the model object to MLflow, once the model has been created. The

model is created by using transfer learning of a ResNet18 model. The ResNet18 is a pre-trained model of a neural network, which consists of 18 layers. The pre-trained ResNet18 model is fine-tuned with the data-loader object, which contains the data. Once the model has been fine-tuned, the model is validated. After the validation has been completed, a confusion matrix is created and the metrics from the validation are logged to MLflow. By logging the metrics and confusion matrix, it is easy to validate the model's performance and the class labels that the model had issues with. After the validation metrics and artifacts have been logged, the model object is logged to MLflow and saved to the PC. The saved model object is used in the final script when it is deployed to a website.

The ”app.py” Python script, Appendix F, is used for deploying the model to a website. This script uses the model as an input and deploys it to a website hosted on the local PC. The website consists of a simple user interface where the user can upload an image to the model. The model predicts what is displayed on the image according to the class labels that the model has been trained on. Once the image has been submitted, the website will display the image and the prediction made by the model. The prediction consists of a label and a number indicating the confidence level of the model's prediction.

The “common.py” Python script, Appendix C, is a file that consists of common libraries and variables used by the different scripts. The idea with this file is to have a variable defined once and for it to be imported to the scripts that are using them. This makes it easier to maintain the code by not having the same type of variables with the same value defined in several files across the whole project. Another important thing about this file is that most of the values of the variables are parsed from the file “params.yaml”. The “params.yaml” file is used for configuring the parameters used by the pipeline. These values are imported and set in the “common.py” file. More on the **“**params.yaml” file in the next chapter.

5.2.4 DVC pipeline

DVC has the capability of creating pipelines. In this project, a pipeline was created for running experiments and creating models. With the pipeline, it is simple to run new experiments and log them automatically to Dagshub. The pipeline consists of different stages, where each stage runs one of the Python scripts. Each stage may use several inputs and parameters to produce outputs. The stages may also have some dependencies and the

stage will not run if any of the stage's dependencies are not met. The pipeline created in this solution uses the previous stage's output as a dependency among other things. This causes the pipeline to stop if any of the prior stages fail.

The pipeline is defined in the file called "dvc.yaml", Appendix A, which is found in the root of the repository. This file consists of the different stages of the pipeline as well as their dependencies, inputs, parameters, outputs and the command executed in the stage. The values of these variables have been split into another file called "params.yaml", Appendix B. This was done to increase the maintainability as well as the readability of the pipeline. Figure 40 shows an overview of the pipeline created.

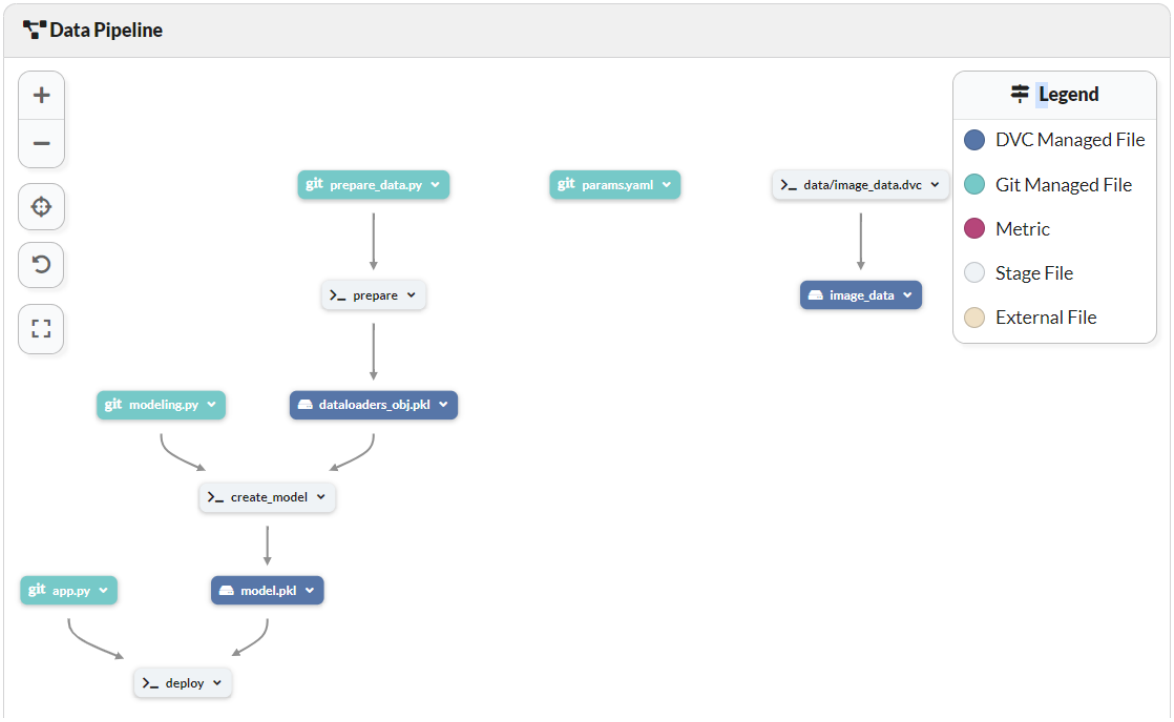

Figure 40: Overview of the DVC pipeline.

Another key feature of the DVC pipelines is that the pipeline knows which stages are required to run and which stages can be skipped. It is possible to skip a stage if nothing has changed from previous runs. The pipeline knows if a stage should run or not by tracking the state of the pipeline's parameters, dependencies, inputs and output artifacts. A checksum for each of these is saved in a file called "dvc.lock. This file is automatically generated and updated after each stage when running the pipeline.

The first stage in the pipeline is called “prepare”. This stage runs the “prepare_data.py” Python script, Appendix D. The stage has several parameters and a single dependency, which is the Python script to execute for this stage. The parameters are used for preparing the data as well as deciding if the current images on the PC should be used or if new images should be fetched from the web. This stage produces a data-loader object as an output, which is used as an input in the next stage.

The second stage is called “create_model”. This stage uses the data-loader object as an input from the previous pipeline stage and will not run if it is missing. The stage runs the “modeling.py” Python script, Appendix E. The stage uses three parameters. The first parameter defines how many iterations the model will be fine-tuned on the data. The second parameter is used for deciding if artifacts should be logged by MLflow or not. The third parameter contains the class labels, which the model has been fine-tuned on. This stage produces a model object as an output.

The last pipeline stage is called “deploy”. This stage runs the “app.py” Python script, Appendix F. The stage has a dependency on the model object from the previous pipeline stage. This stage uses one parameter, which is used for deciding if the website where the model is deployed, should be automatically opened or not. This stage does not produce any outputs.

5.2.5 Demo website

To allow users to test and see how the model is performing, a website was developed to which the model was deployed. Figure 41 shows the initial view of the website, Appendix G. The website consists of a button to browse for files to be given as input to the model and a submit button to trigger the model to make its prediction. There is also a button for showing the different class labels that the model has been trained on.

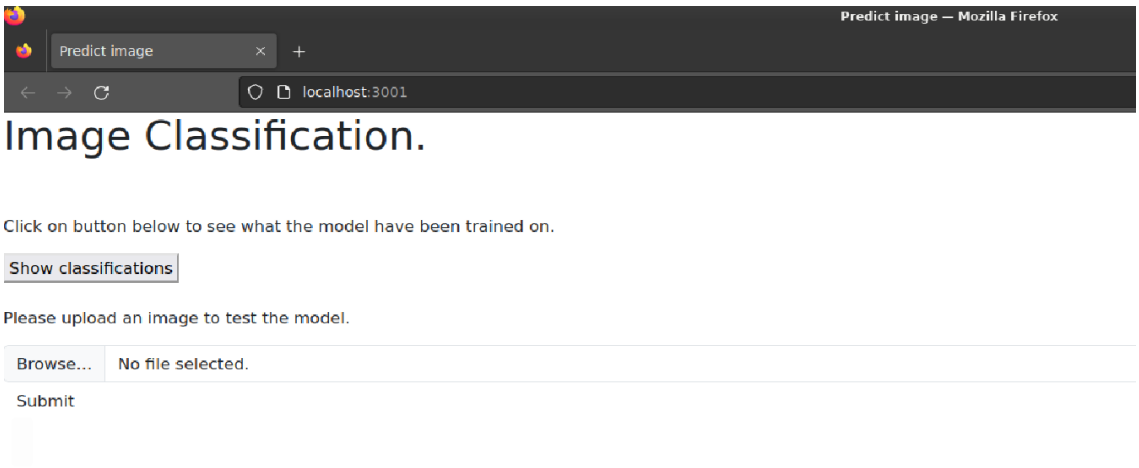

Figure 41: Demo website used to test the models.

Once the user has selected and submitted the image, the model predicts what is shown on the image according to what it has been trained on. The website will display the input image and the prediction of the model as well as the confidence level of the model's prediction as shown in Figure 42.

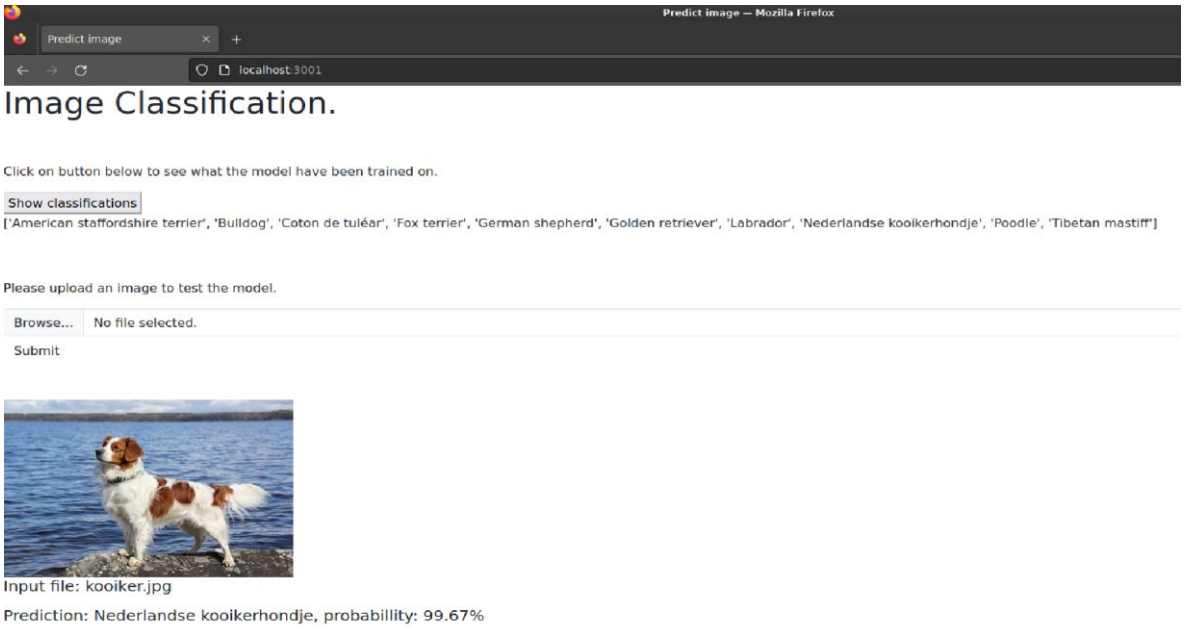

Figure 42: The view once a user has submitted an image.

By changing the Git branch to a branch used when developing a model for predicting vehicles displayed on an image and checking out the corresponding datasets tracked by DVC, it was possible to re-run the pipeline to create the model. The model created is deployed to the website, where the user can test it by submitting images of vehicles as shown in Figure 43.

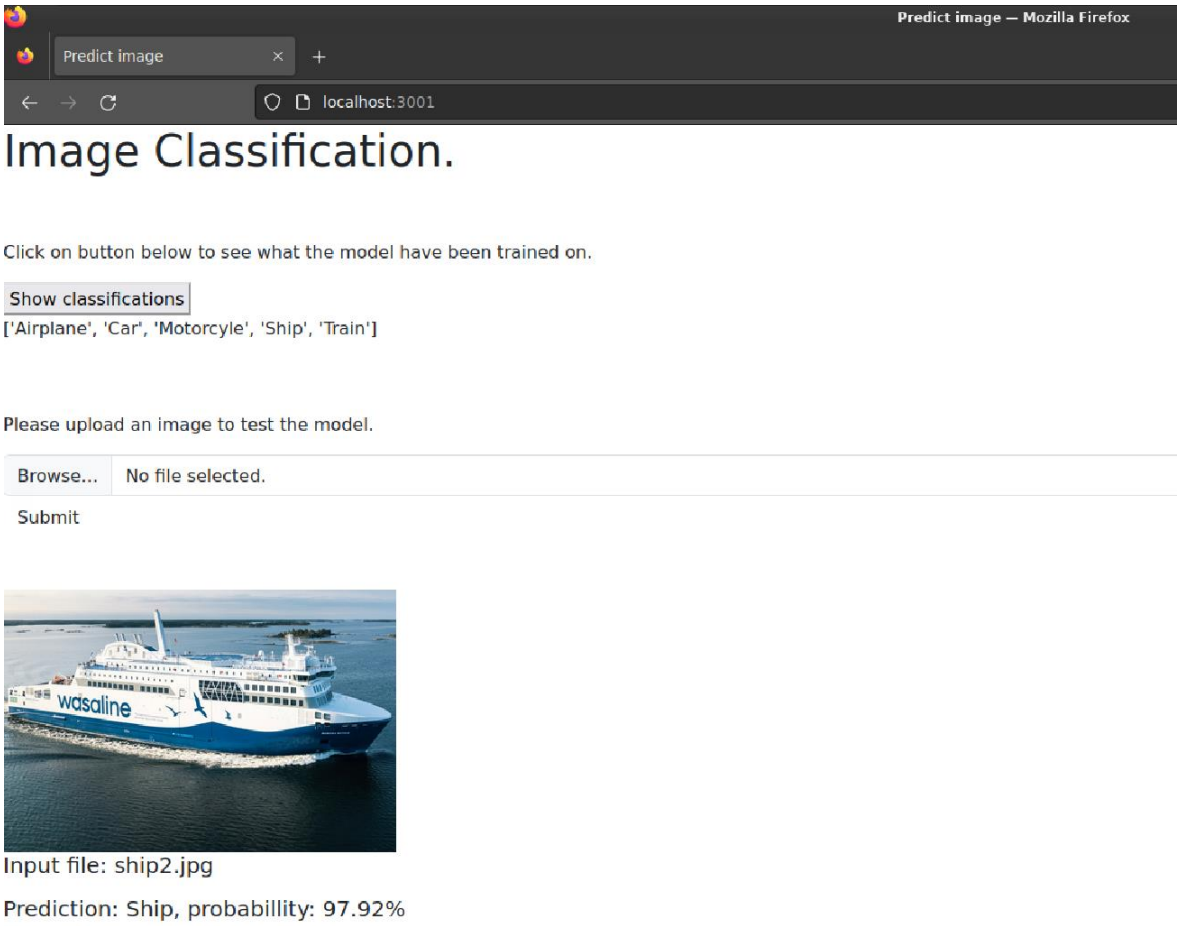

Figure 43: Vehicle model making its prediction on the submitted image.

When submitting a random image that does not represent anything that the model has been trained on, it will still make a prediction, but the confidence level will be much lower, as expected. For example, when submitting an image of a dog to the model that is trained to predict vehicles, the result will be poor, see Figure 44. In this case, the model most likely thinks that the background of the dog is water and therefore thinks it might be a ship that is displayed in the image.

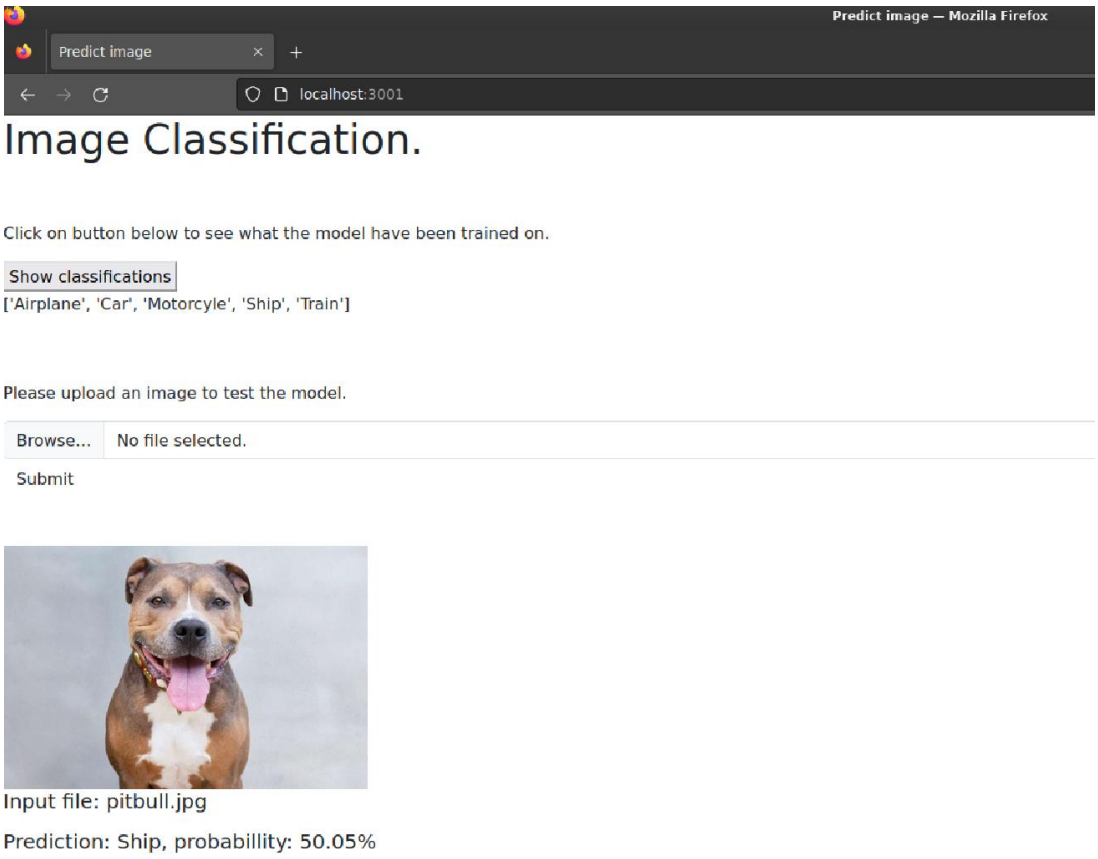

Figure 44: Bad probability with a random image as input.

5.2.6 Model registry

Once a suitable model has been found by running several experiments, it is possible to add them to the MLflow model registry. This is done through the MLflow UI, experiments view, which is found under the remote button at Dagshub, as shown in Figure 45.

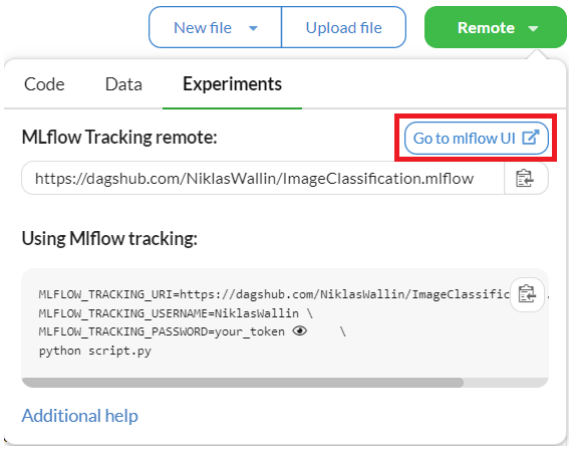

Figure 45: Link to MLflow UI.

At the MLflow experiments view, all the experiments that have been logged with the MLflow library will be shown as well as a short overview of them, as shown in Figure 46. Here, it is also possible to select multiple experiments to compare them with each other.

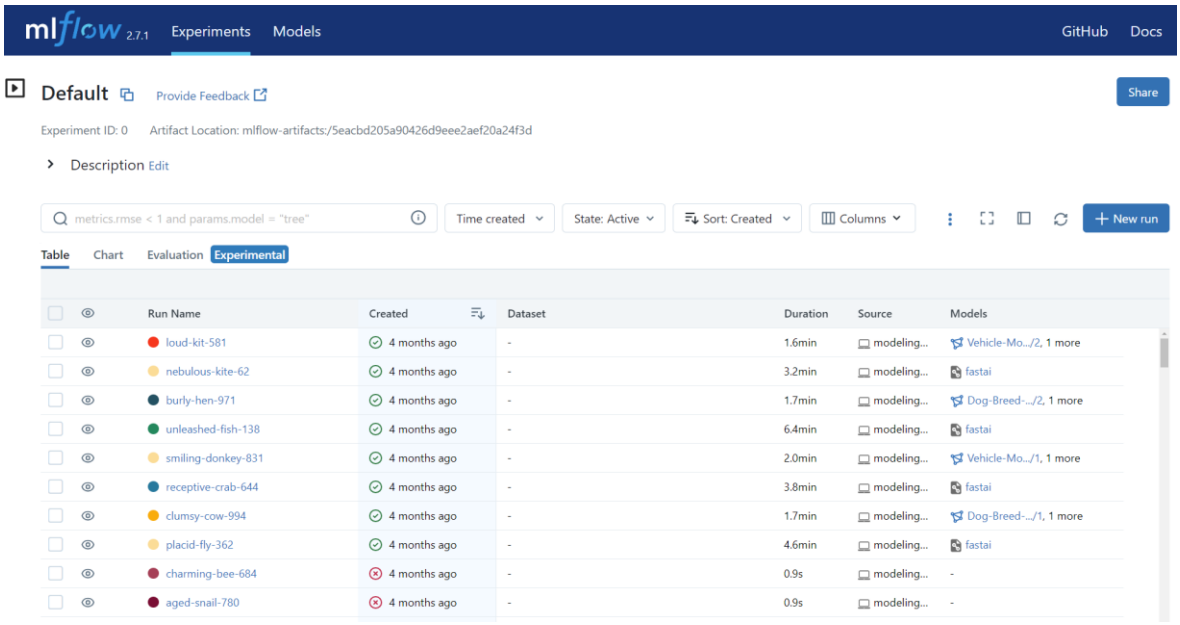

Figure 46: Experiments logged with MLflow.

By selecting one of the experiments, it is possible to have a closer look at it, see Figure 47. Here, it is possible to see everything regarding this experiment. For example, the run id and the Git commit used for this experiment, which is useful if the experiment should be reproduced.

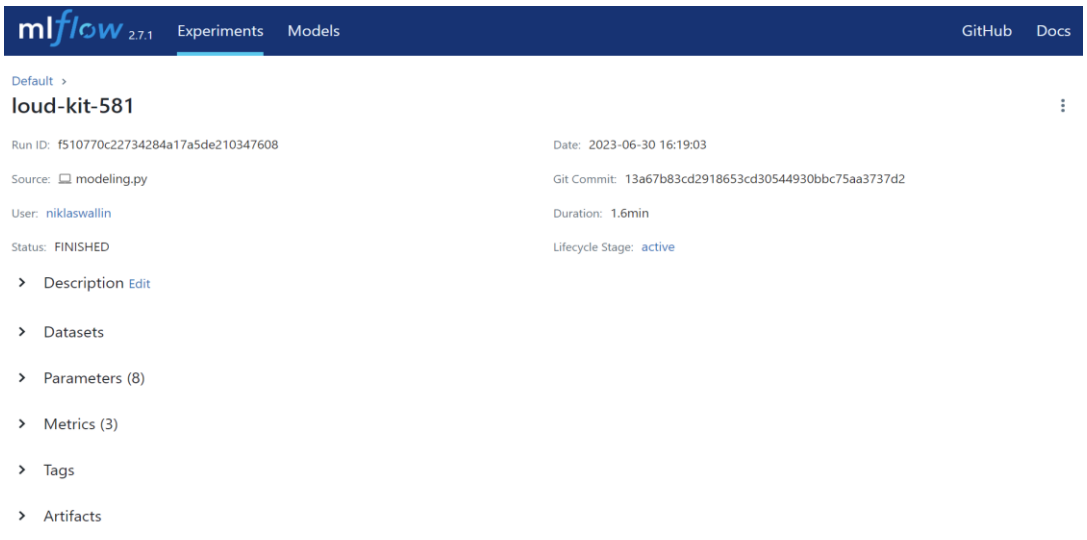

Figure 47: Closer overview of an experiment.

Parameters, metrics and artifacts can also be found under their own section. As shown in Figure 48, the artifacts section may contain images, virtual environments, requirements and dependencies needed for using the model.

Figure 48: Artifacts logged off an experiment.

In the artifacts section, it is also possible to register a model to the registry. This is done by pressing the register model button, which allows the user to create a new model registry or add the model to an existing model registry, according to Figure 49.

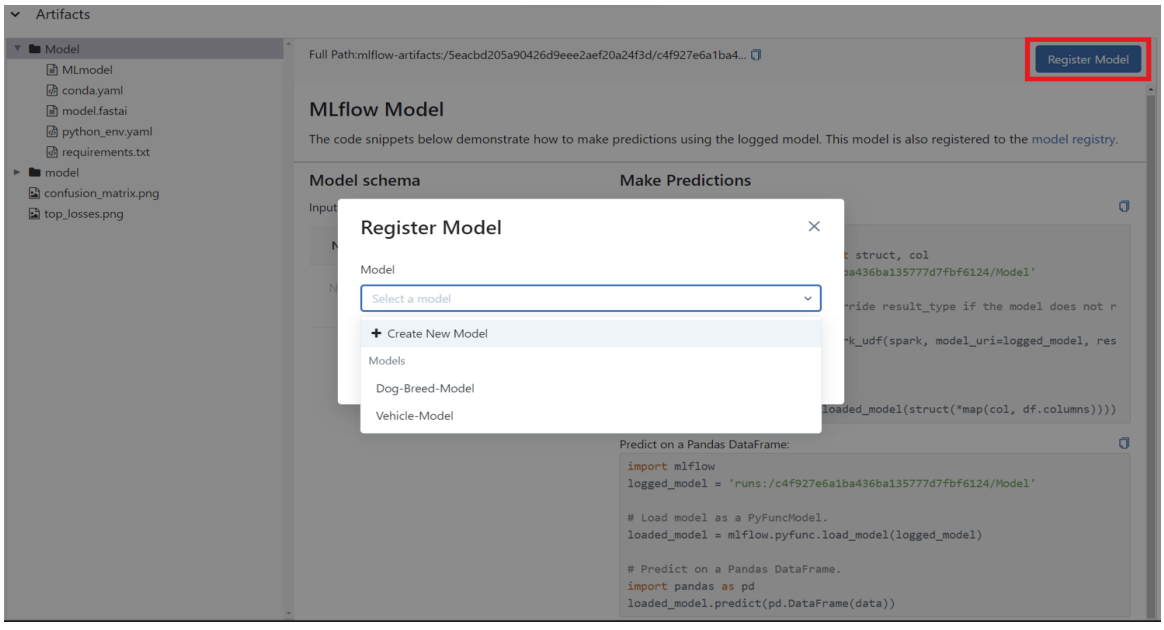

Figure 49: Adding a model to the registry.

Once the model has been added to a registry, it will appear under the models view, as shown in Figure 50. At this view, the user can see the different types of models registered, the model versions and their stage. By default, when a new version of a model is added, it will not be in any stage, it must be changed manually.

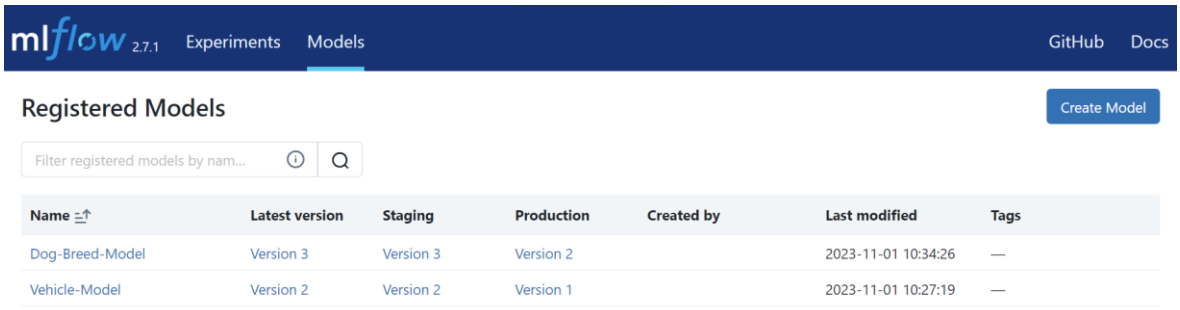

Figure 50: Registered models view.

By selecting one type of the registered models, it is possible to see all the different versions that have been registered of this type. This view shows when the versions of a model have been registered and which stage the different versions are at, either staging, production or archived, as shown in Figure 51.

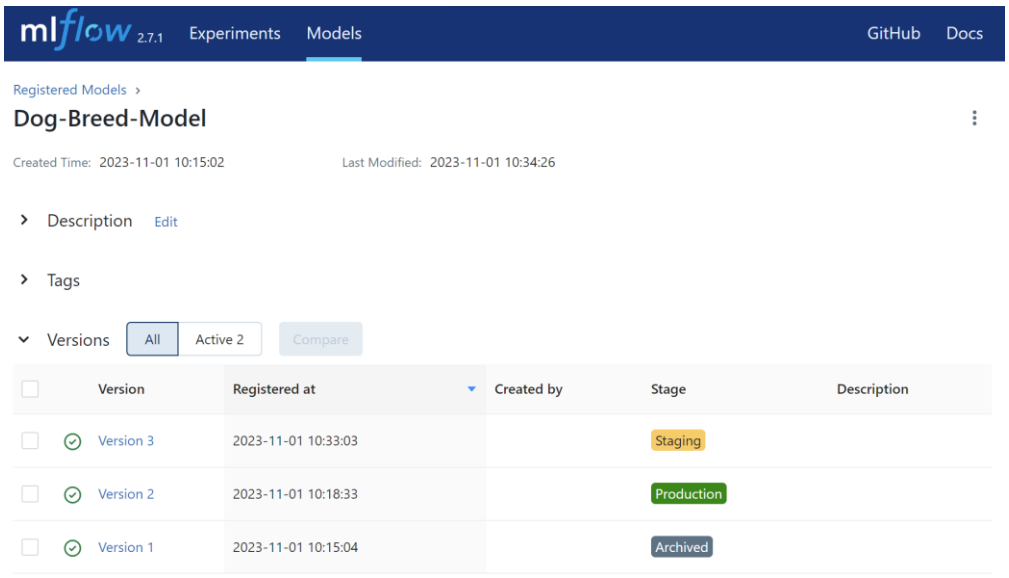

Figure 51: The versions of models registered.

To inspect a specific version further, it is possible to select it, which will open a view with more information, as shown in Figure 52. In this view, it is possible to change the state of

the model and find a direct link to the experiment via the source run link. This link allows the user to easily verify which Git commit was used for creating this version of the model and an overview of the experiment.

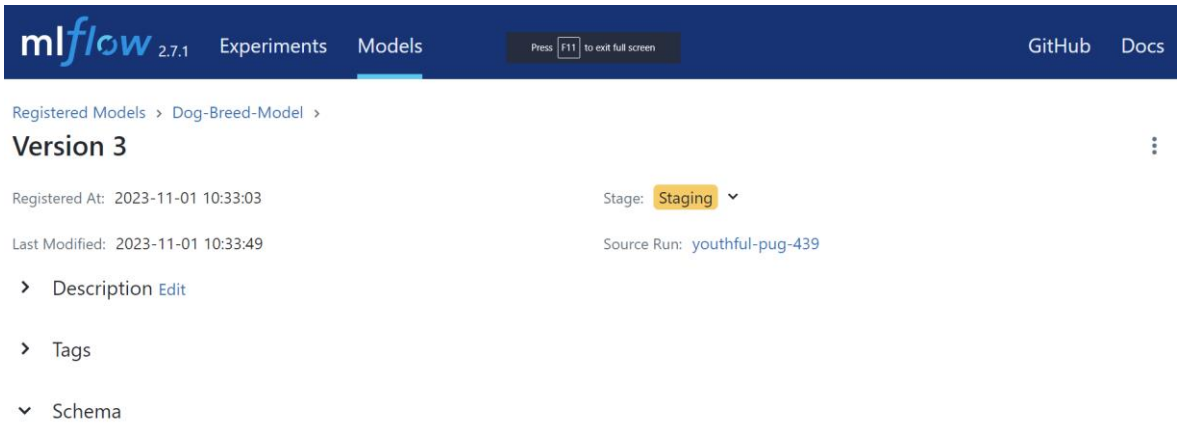

Figure 52: A closer look at version 3 of the model.

6 Discussion

The first objective of this thesis was to find a free platform and open-source tools for hosting and managing machine learning projects as well as creating a pipeline for reproducing earlier experiments and models. The second objective was to find open-source tools regarding the version control for all the different files required to be tracked, such as code, dataset, hyperparameters, metrics and other potential artifacts. The third objective was to find an open-source tool for managing the different versions of a certain machine learning model. All these three objectives were achieved in this thesis.

The platform that was selected for hosting and managing the machine learning projects was Dagshub. Dagshub combines the version control tools Git and DVC. Git is the version control tool for managing the source code files and other small sized files. Git also tracks files that were generated by DVC. DVC is the version control tool that is used for tracking the datasets used for training machine learning models.

DVC was also used for creating and running the pipeline. The DVC pipeline was able to run new experiments and create new models, as well as reproduce earlier experiments and machine learning models according to the Git branch and dataset version on the PC. The experiments were logged to both Dagshub and MLflow. MLflow contained a higher detail

level of the experiments. This was because of the MLflow logger, which logged everything about how the model was trained and created. MLflow was also used for tracking and managing different versions of certain models in a model registry.

With the MLflow model registry, it was possible to version models and manage their current state in a centralized repository. This gave a good overview of all versions of a model and their stages. With this overview, it was possible to know which models are ready to be deployed into production and which models require further testing. A closer look at a model provided some details about the model and a direct link to the experiment for more details about how the model was created.

Even though all the research objectives were accomplished, there were some limitations of the solution. For example, the free version of Dagshub has a limited storage capacity of 100GB. This may be an issue with projects that contain large datasets, where the storage limitation may be reached. Another limitation of the free version of Dagshub is that there can only be a maximum of three users who are collaborating on a private repository. However, on public repositories, there is no limit on the number of collaborators.

There can also be challenges with the version control tools, Git and DVC. At least if there is no prior experience with the tools. New users can forget to run some commands, which may result in wrong versions of either the code, the dataset or the hyperparameters. This may result with a mix of different versions of files that can end up with unwanted models or other issues connected to the version conflict.

Further research could include new features and improvements made to DVC version 3 or newer. This thesis used DVC version 2. There could also be a summarization of the visual studio code DVC extension. There could also be a comparison of different platforms, both commercial platforms and free platforms for hosting and managing machine learning projects. This could include benefits and flaws of each platform. Also, further research could be done to investigate other free tools that are useful for machine learning projects, as well as what are the best practices when developing a machine learning project.

7 References

Ataman, A. (2023, April 7). *Data Quality in AI: Challenges, Importance & Best Practices*. Retrieved from AIMultiple: https://research.aimultiple.com/data-quality-ai/

Atlassian. (2023). *Git hooks*. Retrieved from Atlassian: https://www.atlassian.com/git/tutorials/git-hooks

Atlassian. (2023). *Git refs: An overview*. Retrieved from Atlassian: https://www.atlassian.com/git/tutorials/refs-and-the-reflog

Atlassian. (2023). *Gitflow Workflow*. Retrieved from Atlassian: https://www.atlassian.com/git/tutorials/comparing-workflows/gitflow-workflow

Atlassian. (2023). *What Is DevOps?* Retrieved from Atlassian: https://www.atlassian.com/devops

Barazida, N. (2021, May 14). *Git Flow for Data Science*. Retrieved from Dagshub: https://dagshub.com/blog/git-flow-for-data-science/

Chacon, S., & Straub, B. (2014). About Version Control. In S. Chacon, & B. Straub, *Pro Git* (p. 10). Retrieved from git-scm: https://git-scm.com/book/en/v2/Getting-Started-About-Version-Control

Chacon, S., & Straub, B. (2014). Distributed Version Control Systems. In S. Chacon, & B. Straub, *Pro Git.* Apress.

Chacon, S., & Straub, B. (n.d.). Centralized Version Control Systems. In S. Chacon, & B. Straub, *Pro Git* (pp. 11-12). Apress.

Cheshire, N. (2023). *Centralized vs Distributed Version Control Systems - What's Best for Notes Development?* Retrieved from Teamstudio: https://www.teamstudio.com/blog/distributed-vs-centralized-version-control-systems-for-lotus-notes

Dagshub. (2023). *DagsHub Pipelines*. Retrieved from Dagshub: https://dagshub.com/docs/feature_guide/pipeline/

Dagshub. (2023). *DAGsHub-Official / DAGsHub-Tutorial*. Retrieved from Dagshub: https://dagshub.com/DAGsHub-Official/DAGsHub-Tutorial

Dagshub. (2023). *Experiments Tracking*. Retrieved from Dagshub: https://dagshub.com/docs/feature_guide/experiment_tracking/

Dagshub. (2023). *What is DagsHub?* Retrieved from Dagshub: https://dagshub.com/about

Data-flair. (2023). *Why is Machine Learning so popular?* Retrieved from Data-flair: https://data-flair.training/blogs/why-machine-learning-is-popular/

Developers.Google. (2023, August 3). *What is Machine Learning?* Retrieved from Developers.Google: https://developers.google.com/machine-learning/intro-to-ml/what-is-ml

Dvc. (2023). *.dvc Files*. Retrieved from Dvc: https://dvc.org/doc/user-guide/project-structure/dvc-files

Dvc. (2023). *Get Started with DVC*. Retrieved from Dvc: https://dvc.org/doc/start

Feki, R. (2023, May 8). *ML experiments management with DVC*. Retrieved from Medium: https://rihab-feki.medium.com/ml-experiments-management-with-dvc-35c2ba78cdf8

Flovik, V. (2019, September 16). *Machine Learning: From hype to real-world applications*. Retrieved from Towardsdatascience: https://towardsdatascience.com/machine-learning-from-hype-to-real-world-applications-69de7afb56b6

Frąckiewicz, M. (2023, June 24). *Getting Started with FastAI: A Beginner's Guide*. Retrieved from Ts2: https://ts2.space/en/getting-started-with-fastai-a-beginners-guide/

Gitlab. (2023). *The benefits of a distributed version control system*. Retrieved from Gitlab: https://about.gitlab.com/topics/version-control/benefits-distributed-version-control-system/

Gitlab. (2023). *What is a centralized version control system*. Retrieved from Gitlab: https://about.gitlab.com/topics/version-control/what-is-centralized-version-control-system/

Gitlab. (2023). *What is DevOps?* Retrieved from Gitlab: https://about.gitlab.com/topics/devops/

Git-scm. (2014). *Customizing Git - Git Hooks*. Retrieved from Git-scm: https://git-scm.com/book/en/v2/Customizing-Git-Git-Hooks#_git_hooks

Git-scm. (2014). *Git Internals - Plumbing and Porcelain*. Retrieved from Git-scm: https://git-scm.com/book/en/v2/Git-Internals-Plumbing-and-Porcelain#ch10-git-internals

Hashesh, A. (2023, August 9). *Version Control for ML Models: Why You Need It, What It Is, How To Implement It*. Retrieved from Neptune: https://neptune.ai/blog/version-control-for-ml-models

Howard, J. (2023). *Welcome to fastai*. Retrieved from Github: https://github.com/fastai/fastai

Intland Software. (2022, March 9). *Your Crash Course in Version Control Systems*. Retrieved from Intland: https://content.intland.com/blog/your-2021-crash-course-in-version-control-systems

Islam, S. (2022, February 25). *MLOps vs. DevOps: What is the Difference?* Retrieved from PhData: https://www.phdata.io/blog/mlops-vs-devops-whats-the-difference/

Ivancic, K. (2020, August 19). *Data Version Control With Python and DVC*. Retrieved from Realpython: https://realpython.com/python-data-version-control/

Kumar, N. (2023, March 12). *Overfitting and Underfitting in ML*. Retrieved from SparkedByExamples: https://sparkbyexamples.com/machine-learning/overfitting-and-underfitting-in-ml/

Lamb, T. (2021, September 30). *Git vs SVN*. Retrieved from Gitkraken: https://www.gitkraken.com/blog/git-vs-svn

Liu, C. (2022, August 12). *Data Transformation: Standardization vs Normalization*. Retrieved from Kdnuggets: https://www.kdnuggets.com/2020/04/data-transformation-standardization-normalization.html

Maayan, G. D. (2022, August 10). *MLOps Vs. DevOps: What's the Difference?* Retrieved from Devops: https://devops.com/mlops-vs-devops-whats-the-difference/

MLflow. (2023). *MLflow Model Registry*. Retrieved from Mlflow: https://www.mlflow.org/docs/latest/model-registry.html

MLflow. (2023). *MLflow Models*. Retrieved from Mlflow: https://www.mlflow.org/docs/latest/models.html

MLflow. (2023). *MLflow Projects*. Retrieved from Mlflow: https://mlflow.org/docs/latest/projects.html

MLflow. (2023). *MLflow Tracking*. Retrieved from Mlflow: https://mlflow.org/docs/latest/tracking.html#

MLflow. (2023). *What is MLflow?* Retrieved from Mlflow: https://mlflow.org/docs/latest/what-is-mlflow.html

Nobledesktop. (2023, September 22). *What is Git and Why Should You Use it?* Retrieved from Nobledesktop: https://www.nobledesktop.com/learn/git/what-is-git

Oladele, S. (2023, October 20). *ML Model Registry: The Ultimate Guide*. Retrieved from Neptune: https://neptune.ai/blog/ml-model-registry

Patel, H. (2023). *How to use MLflow to Track and Structure Machine Learning Projects?* Retrieved from Censius: https://censius.ai/blogs/mlflow-machine-learning-projects

Pubudu, N. (2021). *Understanding Git and Version Control Systems*. Retrieved November 2022, from Medium: https://medium.com/nerd-for-tech/understanding-git-and-version-control-systems-e76da5ec8b34

Rosenbaum, O. (2020, December 14). *A Visual Guide to Git Internals — Objects, Branches, and How to Create a Repo From Scratch*. Retrieved from Freecodecamp: https://www.freecodecamp.org/news/git-internals-objects-branches-create-repo/

Saboo, S. (2021, September 7). *DagsHub → Github for Data Science*. Retrieved from Towardsai: https://pub.towardsai.net/dagshub-github-for-data-science-92e77adbc9a3

Şen, E. (2022, February 27). *Version Control*. Retrieved from Medium: https://senertugrul.medium.com/version-control-3f22d1bdeb52

Sharma, H. (2023, September 1). *The Crucial Role of MLOps in Enterprise Machine Learning*. Retrieved from Medium: https://medium.com/international-school-of-ai-data-science/the-crucial-role-of-mlops-in-enterprise-machine-learning-9bc9e79d4949

Stackoverflow. (2022). *2022 Developer Survey*. Retrieved January 8, 2023, from Stackoverflow: https://survey.stackoverflow.co/2022/

Štetić, F. (2022, June 15). *Versioning data using DVC*. Retrieved from Megatrend: https://www.megatrend.com/en/versioning-data-using-dvc/

Stitchdata. (2023). *Data transformation: A comprehensive guide to benefits, challenges, and tools*. Retrieved from Stitchdata: https://www.stitchdata.com/resources/data-transformation/

Yrkeshögskolan Novia. (2023). *Intelligent Systems Institute*. Retrieved December 2022, from Novia: https://www.novia.fi/en/intelligentsystems/

Appendices

Here are the appendices of the code and comments on the project.

A. Definition of the DVC pipeline.

B. Definition of the parameters used in the pipeline.

C. Library containing common variables across the Python scripts.

D. Python script for preparing the data to be used for creating the model.

E. Python script for creating and validating the model as well as logging data about the model.

F. Python script of the website where the model is deployed to.

G. Structure of the website.

Appendix A – dvc.yaml

```
stages:
  prepare:
    cmd: python src/model_creation/prepare_data.py
    deps:
      - src/model_creation/prepare_data.py
    params:
      - quaries
      - prepare.batch_size
      - prepare.valid_pct
      - prepare.seed
      - prepare.n_images_per_class
      - prepare.download_new_data
    outs:
      - ${prepare.data_loaders_pkl}

  create_model:
    cmd: python src/model_creation/modeling.py
    deps:
      - src/model_creation/modeling.py
      - ${prepare.data_loaders_pkl}
    params:
      - quaries
      - create_model.fine_tune_cycles
      - create_model.log_artifacts
    outs:
      - ${create_model.model_pkl}

  deploy:
    cmd: python src/webpage/app.py
    deps:
      - src/webpage/app.py
      - ${create_model.model_pkl}
    params:
      - quaries
      - deploy.open_browser
```

Appendix B – params.yaml

```
quaries:
  - Golden retriever
  - Poodle
  - German shepherd
  - Fox terrier
  - Tibetan mastiff
  - American staffordshire terrier
  - Coton de tuléar
  - Nederlandse kooikerhondje
  - Bulldog
  - Labrador

data_path: data/image_data

prepare:
  batch_size: 64
  valid_pct: 0.25
  seed: 1
  n_images_per_class: 100
  download_new_data: false
```

```
  #outputs
  data_loaders_pkl: models/dataloaders_obj.pkl

create_model:
  fine_tune_cycles: 2
  log_artifacts: true

  #outputs
  model_pkl: models/model.pkl

deploy:
  open_browser: true
```

Appendix C – common.py

```
import os
import sys
import git
import pickle
import numpy as np
import dvc.api
from fastai.vision.all import *

# Set repo root path
git_repo = git.Repo(__file__, search_parent_directories=True)
REPO_ROOT_PATH = git_repo.git.rev_parse("--show-toplevel")

# created params object for fetching parameters from params.yaml
params = dvc.api.params_show()

# path to where model is saved
MODEL_PATH = f"{REPO_ROOT_PATH}/{params['create_mod-
el']['model_pkl']}"
# path to dataloaders object
DLS_PATH = f"{REPO_ROOT_PATH}/{params['prepare']['data_load-
ers_pkl']}"


# Common parameters
# The classes used for the classification.
quaries = params["quaries"]
DATA_PATH = f"{REPO_ROOT_PATH}/{params['data_path']}"

# Prepare_data parameters
batch_size = params["prepare"]["batch_size"]
valid_pct = params["prepare"]["valid_pct"]
seed = params["prepare"]["seed"]
# Determine how many images of each class that you want to download
n_images_per_class = params["prepare"]["n_images_per_class"]
download_new_data = params["prepare"]["download_new_data"]


# Modeling parameters
fine_tune_cycles = params["create_model"]["fine_tune_cycles"]
log_artifacts = params["create_model"]["log_artifacts"]


# Deploy parameters
open_browser = params["deploy"]["open_browser"]
```

Appendix D – prepare_data.py

```
import shutil
import sys
import os
from bing_image_downloader import downloader

# import common script
# This is required so that the file both work with dvc pipeline
# as well as an independent script, possible to run from anywere in
the repo.
common_lib_path = __file__.split("/")[:-2]
common_lib_path = "/".join(common_lib_path) + "/common"
try:
    sys.path.insert(0, common_lib_path)
    from common import *
except Exception as e:
    print(f"Could not import library reason - {e}")
    exit()

def remove_data_folder():
    """Removes all files and folders of the directory where data is
stored."""

    if os.path.exists(DATA_PATH):
        shutil.rmtree(DATA_PATH)

def download_data():
    """Downloads data from bing."""


    if not os.path.exists(DATA_PATH):
        os.makedirs(os.path.abspath(DATA_PATH))

    # change to directory where images will be downloaded to
    prev_dir = os.getcwd()
    os.chdir(os.path.abspath(DATA_PATH))

    # Download images for each query
    for quary in quaries:
        downloader.download(quary,
                        limit=n_images_per_class,
                        output_dir="",
                        adult_filter_off=False,
                        force_replace=False,
                        timeout=10)

    # change back to the previous directory
    os.chdir(prev_dir)

def preprocess_data():
    """Creates Datablocks, grow dataset with by using augmenta-
tion."""

    if len(quaries)*n_images_per_class*0.75 < 64:
        global batch_size
        batch_size = 16

    # Define the data block
```

```
    img_classes = DataBlock(blocks=(ImageBlock, CategoryBlock),
# Classification task: image data and category labels
                            get_items=get_image_files,
# Function for converting file path to image data
                            splitter=RandomSplitter(valid_pct=0.25,
seed=1),  # Validation data fraction
                            get_y=parent_label,
# Use the direcotry name as the class label
                            item_tfms=Resize(224))
# Resize images to sizexsize

    dls = img_classes.dataloaders(DATA_PATH, batch_size=batch_size)
    # Visualize training images
    dls.train.show_batch(max_n=10, nrows=2, unique=False)

    # Create a new data block from the old one with default
    # FastAI augmentation transforms.
    img_classes = img_classes.new(item_tfms=Resize(224),
                                  batch_tfms=aug_transforms())

    # Create new dataloaders from the new data block
    dls = img_classes.dataloaders(DATA_PATH, batch_size=batch_size)
    # Show augmented example images
    dls.train.show_batch(max_n=10, nrows=2, unique=True)

    # save data-loader object
    try:
        with open(DLS_PATH, "wb") as f:
            pickle.dump(dls, f)
    except Exception as e:
        print(f"Could not save the data-loaders object. - {e}")

if __name__ == "__main__":

    if download_new_data:
        # Clean data folder and fetch new images from bing
        remove_data_folder()
        download_data()

    # preprocess data
    preprocess_data()
```

Appendix E – modeling.py

```
import mlflow
import sys
from PIL import Image

# import common script
# This is required so that the file both work with dvc pipeline
# as well as an independent script, possible to run from anywere in
the repo.
common_lib_path = __file__.split("/")[:-2]
common_lib_path = "/".join(common_lib_path) + "/common"
try:
    sys.path.insert(0, common_lib_path)
    from common import *
except Exception as e:
    print(f"Could not import library reason - {e}")
    exit()
```

```
def validate_learner(learn):
    """Get the metrics from the learner."""

    validation = learn.validate()
    validation_loss = validation[0]
    error_rate = validation[1]
    accuracy = validation[2]

    return validation_loss, error_rate, accuracy

def start_logger():
    """Start ml-flow autologger"""

    # MLFlow tracking uri from condas environment variables,
    # username and token also defined there.
    try:
        mlflow.set_tracking_uri(os.environ["MLFLOW_TRACKING_URI"])
        mlflow.fastai.autolog()
    except Exception as e:
        print(f"Something went wrong when trying to star MLFlow log-
ger - {e}")

def get_performance_figures(learn):
    """Function to get confusion matrix and top losses figures."""

    # Evaluate the results on the validation set
    # by for example plotting the confusion matrix
    interp = ClassificationInterpretation.from_learner(learn)
    confusion_matrix = interp.plot_confusion_matrix()
    plt.savefig("confusion_matrix.png")
    top_losses = interp.plot_top_losses(6, nrows=2)
    plt.savefig("top_losses.png")

    if log_artifacts:
        mlflow.log_artifact("confusion_matrix.png")
        mlflow.log_artifact("top_losses.png")

def tune_model(dls):
    """Use transfer learning to fine tuning a model."""

    # Fine tune the parameters of the network
    learn = cnn_learner(dls, resnet18, metrics=[error_rate, accu-
racy])
    learn.fine_tune(fine_tune_cycles)

    get_performance_figures(learn)

    val_loss, err_rate, acc = validate_learner(learn)
    print(f"{val_loss=}, {err_rate=}, {acc=}")

    if log_artifacts:
        mlflow.fastai.log_model(learn, "Model")
        mlflow.log_metric("Validation loss", val_loss)
        mlflow.log_metric("Error rate", err_rate)
        mlflow.log_metric("Accuracy", acc)
        mlflow.log_param("Classes", quaries)

    # save model
```

```
    learn.export(fname=Path(MODEL_PATH))

if __name__ == "__main__":

    # import the dataloaders object created by prepare_data.py
    try:
        with open(DLS_PATH, "rb") as f:
            dls = pickle.load(f)
    except Exception as e:
        print(f"Issues reading dataloader file. - {e}")
        exit()

    # Transfer learning fine tuning.
    if log_artifacts:
        start_logger()
    tune_model(dls)
```

Appendix F – app.py

```
import os
import sys
import webbrowser
import dvc.api
from flask import Flask, render_template, request
from fastai.vision.all import *
from PIL import Image
from numpy import asarray

# import common script
# This is required so that the file both work with dvc pipeline
# as well as an independent script, possible to run from anywere in
the repo.
common_lib_path = __file__.split("/")[:-2]
common_lib_path = "/".join(common_lib_path) + "/common"
try:
    sys.path.insert(0, common_lib_path)
    from common import *
except Exception as e:
    print(f"Could not import library reason - {e}")
    exit()

# path were user inputed images are saved.
inputed_images_path = __file__.split("/")[:-1]
inputed_images_path = "/".join(inputed_images_path) + "/user_images"

params = dvc.api.params_show()
app = Flask(__name__, static_url_path = inputed_images_path,
static_folder = inputed_images_path)

try:
    learn = load_learner(MODEL_PATH)
except Exception as e:
    print(f"Issues loading the model - {e}")
    exit()

# Create folder for storing images used for prediction.
if not os.path.exists(inputed_images_path):
    os.makedirs(inputed_images_path)
```

```
# sort the classes
quaries.sort()

@app.route("/", methods=["GET"])
def default():
    return render_template("index.html", classes=quaries)

@app.route("/", methods=["POST"])
def predict():
    try:
        file_to_predict = request.files["image_to_predict"]
        image_path = f"{inputed_images_path}/{file_to_predict.file-
name}"
        file_to_predict.save(image_path)

        img = Image.open(file_to_predict)
        numpydata = asarray(img)
        predicted_label, _, probs = learn.predict(numpydata)
        result = f"{predicted_label}, probabillity:
{probs.max()*100:2.2f}%"
        print(result)
        print(f"{image_path=}")
    except Exception as e:
        print(f"SOMETHING WENT WRONG! - {e}")
        no_img_text = "No image selected, or wrong type of file."
        return render_template("index.html", no_img_text=no_img_text,
classes=quaries)

    return render_template("index.html", result=result, the_image=im-
age_path, classes=quaries, picture_name=file_to_predict.filename)

def run_app():
    """Open browser and run the app"""

    url = "http://localhost:3000/"
    webbrowser.open(url)
    app.run(port=3000, debug=False)

if __name__ == "__main__":

    if open_browser:
        # deploy model to the browser
        url = "http://localhost:3001/"
        webbrowser.open(url)
        app.run(port=3001, debug=True)
```

Appendix G – index.html

```
<!DOCTYPE html>

<html>

    <head>

        <title>Predict image</title>

        <meta charset="utf-8">

        <meta name="viewport" content="width=device-width, initial-
scale=1">
```

```
        <link href="https://cdn.jsdelivr.net/npm/bootstrap@5.3.0-
alpha1/dist/css/bootstrap.min.css"
              rel="stylesheet"
              integrity="sha384-
GLhlTQ8iRABdZLl6O3oVMWSktQOp6b7In1Zl3/Jr59b6EGGoI1aFkw7cmDA6j6gD"
              crossorigin="anonymous">
    </head>
    <body>
        <H1>Image Classification.</H1>
        <br></br>
        <p>Click on button below to see what the model have been
trained on.</p>
        {% if classes %}
        <button id="button">Show class labels</button>
            <p id="testing" style="display: none;" >{{classes}}</p>
        {% endif %}
        <br></br>
        <p>Please upload an image to test the model.</p>
        <!---->
        <form action="/" method="post" enctype="multipart/form-data"
>
            <input class="form-control" type="file"
name="image_to_predict" >
            <input class="btn" type="submit" name="Predict image"
value="Submit" >
        </form>
        <!-- Display text when user have submitted no or invalid
file.-->
        {% if no_img_text %}
            <br></br>
            <h4>{{no_img_text}}</h3>
        {% endif %}
        <br></br>

        <!-- Show image used for prediction along with label and
probability.-->
        {% if result %}
```

```
            <img id="myImg" src={{the_image}} height="240"
width="360"
                    alt="Image used for prediction." title="Image
used for prediction.">
            <h5>Input file: {{picture_name}}</h5>
            <p><!--New line--></p>
            <h5>Prediction: {{result}}<h5>
        {% endif %}
    <!-- Script for showing/hiding the models class labels -->
    <script type="text/javascript">
        var button = document.getElementById('button');
        button.onclick = function(){
            var div = document.getElementById('testing');
            if (div.style.display == 'none') {
                div.style.display = 'block';
            }
            else {
                div.style.display = 'none';
            }
        };
    </script>
    <!--Bootstrap js-->
    <script
src="https://cdn.jsdelivr.net/npm/@popperjs/core@2.11.6/dist/umd/popp
er.min.js"
            integrity="sha384-
oBqDVmMz9ATKxIep9tiCxS/Z9fNfEXiDAYTujMAeBAsjFuCZSmKbSSUnQlmh/jp3"
            crossorigin="anonymous">
    </script>
    <script src="https://cdn.jsdelivr.net/npm/bootstrap@5.3.0-
alpha1/dist/js/bootstrap.min.js"
            integrity="sha384-
mQ93GR66B00ZXjt0YO5KlohRA5SY2XofN4zfuZxLkoj1gXtW8ANNCe9d5Y3eG5eD"
            crossorigin="anonymous">
    </script>
    </body>
</html>
```