

Assignment 1: Design and Implementation of MLOps for an ML Application

Group 11:
Sietse Smole
Kenneth de Wolff
Sander Schuitemaker
Ibrahim Zayani
Mats Pierik

GitHub:
<https://github.com/Sanderror/Data-engineering.git>

1. Overview of the ML Application

Phishing is one of the most common cybersecurity threats today (Jang-Jaccard & Nepal, 2014). Phishing, which involves tricking users into revealing sensitive information by disguising malicious links as legitimate, can have far-reaching consequences. Examples are identity theft, financial loss, or unauthorized access to critical systems (APWG, 2024). Real-time threat detection and blocking have become crucial for both individuals and companies due to the increasing complexity of phishing attempts (Sahami et al., 1998).

Machine learning (ML) can provide a powerful solution for identifying phishing links by analyzing patterns that traditional security systems might overlook. By examining features such as the number of dots and the length of the URL, ML models can detect subtle, yet decisive differences between legitimate and phishing links (Abdelhamid et al., 2014). Because of the ability to adapt and learn from new data, ML is very effective when dealing with evolving phishing tactics.

The ML application described in this report is designed to classify URLs as phishing or legitimate based on specific features. The success of this application relies on high accuracy in distinguishing between phishing and legitimate URLs. An adaptable model is crucial as phishing tactics continuously evolve (Gupta et al., 2017). Additionally, the model must process and classify these URLs in real time to alert users quickly and prevent potential security risks.

A comparison was made between the performances of three state of the art classification models, and the best model was deployed in the application. The models compared were the Support Vector Machine model (SVM), Logistic Regression model (LR), and Random Forest model (RF). The best model was chosen based on recall, rather than accuracy. This decision was made because in the case of phishing links, type II errors (predicting no phishing link, when in reality it is) are worse than type I errors (predicting phishing link, when in reality it is not). If the user trusts our application, and uses a phishing link which our application predicted to not be a phishing link, this could have bad consequences (as noted before). Therefore, we prefer a high recall over a high accuracy or precision. However, we also do not want a model with a recall of 1.0 that simply always predicts “it is a phishing link”. That is why the accuracy of the chosen model should be at least 0.7.

To maintain the high performance of the application, an MLOps framework was implemented. This enables consistent monitoring, retraining, and deployment of the application (Kreuzberger et al, 2022). Regular retraining is necessary to keep the model updated with new data. As phishing patterns could potentially change, continuously providing the most recent data into the training set helps the model learn emerging patterns.

The dataset used to train and test the the application has been obtained from Kaggle under the name “Phishing Dataset for Machine Learning”¹ (Tan, 2018). This dataset contains 48 features on the URLs of 5.000 phishing webpages and 5.000 legitimate webpages. We manually selected 12 of these features to be used in our model. The features were chosen based on

¹ <https://www.kaggle.com/datasets/shashwatwork/phishing-dataset-for-machine-learning>

interpretability, as there was no explanation available of the features accompanied with the dataset, so we chose 12 features that we understood and we knew how to scrape from a URL. Those are the following 12 features:

- NumDots: the number of dots (.) in the URL
- UrlLength: the length of the full URL
- NumDash: the number of dashes (-) in the URL
- NumDashInHostname: the number of dashes (-) in the hostname of the URL
- AtSymbol: the number of at signs (@) in the URL
- TildeSymbol: the number of tilde signs (~) in the URL
- NumUnderscore: the number of underscores (_) in the URL
- NumPercent: the number of percentage signs (%) in the URL
- NumAmpersand: the number of ampersand signs (&) in the URL
- NumHash: the number of hashtags (#) in the URL
- NumNumericChars: the number of numbers in the URL
- NoHttps: whether the url starts with "https" (1 if no, 0 if yes)

To track the model's performance continuous monitoring is crucial. Focusing on metrics like prediction accuracy and recall allows for the comparison of models to select the best-performing model to detect phishing links before they could impact users (Sculley et al., 2015).

The deployment of the model needs to prioritize low latency and scalability, ensuring that the system can handle increased traffic without declining performance. This requires containerization and orchestration to ensure flexible and on-demand scalability (Restack, 2024). Additionally, CI/CD (Continuous Integration/Continuous Deployment) pipelines are implemented to automate testing, versioning, and deployment. This reduces the risk of errors during updates and streamlines the release of model modifications (Restack, 2024).

This report outlines the the design and implementation of the MLOps framework, and reflections on the development process, followed by an overview of individual contributions.

2. Design and Implementation of the MLOps System

The full MLOps system can be divided into two main components: the ML pipeline and the serving application. Each of these two components are built, tested, and deployed using CI-CD pipelines. Furthermore, there are CI-CD pipelines implemented for the compiling and publishing of the pipeline, for the re-execution of the pipeline and for the redeployment of services. The full design of the MLOps system is demonstrated in Appendix A. Let's now dive into the individual components in more detail.

2.1 ML pipelines

The ML pipeline is created in a Jupyter Notebook using Google Cloud's Vertex AI functionality. Before being able to create the ML pipeline, we needed the data. We first had to manually clean the full phishing dataset as obtained from Kaggle into a dataset with 12 variables, which were

described in section 1. Introduction. This cleaned dataset was manually uploaded as a CSV-file to a Google Cloud Cloud Storage data bucket. Then, the first section of the pipeline could start: data preparation. First, the dataset was downloaded from the data bucket into a CSV-file which can be used throughout the pipeline. This dataset was then forwarded to the second component, which created a train set and test set to be used for the second section of the pipeline: the modeling. The dataset was split into a train set of 80% and a test set of 20%.

In the second section of the pipeline, the modeling, three different models were trained and tested to be compared later. The three models were a Support Vector Machine model (SVM), a Random Forest model (RF), and a Logistic Regression model (LR). Each of these models had its separate component within the pipeline. Within each of these components, the same steps were performed. First, the train and test sets, as created in the previous component, were loaded into a Pandas DataFrame. Then, the train and test sets were split into the dependent variables (X_{train} , X_{test}) and the independent variable (y_{train} , y_{test}) using the 'Class' attribute (indicating whether an instance is phishing or not). Then, the models were trained with default parameters using the X_{train} and y_{train} data. Afterwards, the models' predictions on X_{test} were generated and stored in y_{pred} . This y_{pred} was used together with y_{test} to obtain the accuracy and recall score of the models. These scores were stored in three dictionaries (for each model one), which were forwarded to the next section: the model comparison. Furthermore, the models were stored to a Pickle-file and some metadata, on the file type (.pkl) and algorithm used (e.g. SVM), were stored.

In the next section, the model comparison, the three models were compared on their performance metrics calculated in the previous section. The best model was chosen as the model with the highest recall, which had an accuracy of at least 0.7. As explained in section 1. Introduction, the type II error is more dangerous than the type I error in this case, which is why we value the recall as the most important metric. At the same time, we do not want a model that always predicts a link to be a phishing link (recall of 1.0, but low accuracy), which is why we want the model to have an accuracy of at least 0.7. If none of the models obtained an accuracy score higher than 0.7, simply the model with the highest accuracy score was chosen.

The final section of the ML pipeline simply uploads the best model, as decided in the previous section, to a Google Cloud Cloud Storage models bucket. After this step, there is one additional step concerning CI-CD pipelines, but that will be explained in section 2.3.

2.2 Serving application

As can be seen in Appendix A, the serving application consists of 2 components: a UI component and an API component. The UI component allows the user to provide a potential phishing link URL in a text box as input (see Appendix B). The user can submit this URL, and then the API component comes into play. The UI component sends a POST request to the API component. The content of this POST request contains the 12 variables described in section 1. Introduction. These variables have been automatically extracted from the URL the user

submitted using regular expressions. The API component then uses these variables as input to the model, which is extracted from the Google Cloud Cloud Storage models bucket (and which was uploaded through the ML pipeline). The model then predicts either 0/False (no phishing link) or 1/True (phishing link), and posts the prediction back to the UI component. The UI component then parses the prediction and renders the response page, which displays 'It is likely that this is a phishing link.' if the prediction is 1/True (see Appendix C) and 'It is unlikely that this is a phishing link.' if the prediction is 0/False (see Appendix D).

2.3 CI-CD pipelines

The final part of the whole MLOps system are the CI-CD pipelines. CI-CD pipelines have been implemented for multiple goals: the compiling and publishing of the ML pipeline, the (re)deployment of the serving application, and the retraining of the ML pipeline. All CI-CD pipelines have been created using Google Cloud Cloud Build triggers. The first trigger that is implemented, compiles and publishes the ML pipeline as described in section 2.1. It creates a pipeline executor and pushes that image to Google Cloud's Artifact Registry. This image is then used to publish the actual pipeline, which uploads the updated model to a Google Cloud Cloud storage models bucket.

However, we do not want to publish this pipeline just once, but rather we want to be able to retrain the ML pipeline to get the best, most up-to-date model continuously. Therefore, for the (re)training of the ML pipeline, a trigger had been created that automatically reruns the whole ML pipeline as described in section 2.1. This trigger runs on a schedule, where it will retrain the pipeline every week on Monday at 9:00 a.m. CEST time zone. Furthermore, the trigger can also be run manually whenever needed.

We already mentioned at section 2.1 that there was one additional step to the ML pipeline, which would be discussed in section 2.3. This additional step is a CI-CD pipeline that automatically triggers the redeployment of the serving application, after the model has been retrained. This is needed, because we need to test if the application still functions correctly with the new model. The Docker containers of the prediction UI and API components are rebuilt and afterwards redeployed in Google Cloud's Cloud Run.

Finally, this redeployment of the serving application is not only performed after the retraining of the ML pipeline, but also after any code change committed to the UI or API component in the GitHub. After a code change, the application has to be rebuilt to see if everything still functions correctly or if errors have been introduced due to the code changes.

3. Reflection

This current design and implementation as described in the previous chapter, allows people to check unknown links and get an indication of whether they could trust that link. Individuals have to copy and paste the link into the indicated field on the web application. After submitting the link a python function using regular expressions will extract information about the link such as the

length, number of specific characters, etc.. Then a prediction is made based on these variables whether the link is a phishing link or not. The user is then given an immediate response which indicates whether they should trust the link or that it is likely a phishing and that they should not interact with it. This application can make the internet a safer place especially for less tech-savvy people like elderly. As elderly are often the target of phishing, they could use this application to protect themselves.

The front-end of the ML application is very basic with a simple html template. At first we had a form which required the user to extract the information from this link and manually fill it in, but this was a tedious task and made the application less useful as it required significantly more time and left room for human error, as people can easily miss a couple of characters. The form is now only 1 question which requires the user to paste the link and submit this. Extracting all the relevant information from the link is done automatically making it user-friendly and very simple and quick to use.

Besides this adjustment the front end could also be more informative as the goal of the application is to prevent phishing. Some rule of thumbs could be added for people to identify phishing links quickly themselves and maybe some help option like a phone number or link to a website of some authority so that people can contact the right authorities immediately in case of a phishing incident. Not too much should be added however as this would make the application too crowded and maybe too complicated for less tech-savvy users which is the target audience. These simple additions and adjustments could make the application more user friendly and effective in preventing phishing.

In the pipeline we opted for three different models as we found that these were used in similar projects and got the best results. We chose SVM, Random Forest, and Logistic regression. To compare the model performance we first tested if the accuracy was above a certain threshold of 0.7, to ensure it was effective and not only predicted one class. Then, we selected the model with the highest recall, as in this use case of detecting phishing links False negatives are (or: the type II error is) more harmful than False positives (the type I error). False negatives in this case are when a phishing link is labeled as a safe link, this can be dangerous as the user is then more likely to use that link. The accuracy has to be above the threshold of 0.7 to prevent the pipeline selecting a model that just labels all links as phishing as this would give a perfect recall., but would make the application useless. This is why we opted to select the model with the highest recall and sufficient accuracy.

Regarding the triggers, there are currently three. The first trigger is a scheduled trigger, scheduled each week on monday around 9:00. This trigger initiates the model retraining as shown in Appendix A. After this training is done and a model is selected and uploaded to the modelbucket, the serving application is automatically redeployed with the new model using the second trigger. The third trigger is on the GitHub repository. Whenever a change is made in one of the serving application files the serving application will be redeployed, updating the API and UI components and making sure users have access to the most recent model.

These triggers are useful to keep the model and serving application up to date, but it is not done optimally right now. The scheduled trigger makes sure that the model is updated and trained on the new data each week. However, it could also be that there is no new data over a couple of weeks resulting in unnecessary model retraining and wasted resources. To solve this problem, the trigger could be adjusted to be dependent on changes in the data bucket. The retraining could for example be triggered whenever 50 new rows of data are added to the data bucket. This also ensures that the used model is up-to-date to the newest available data while also making sure no resources are wasted. During this project we tried to configure this trigger but ran into some problems, therefore we opted for the scheduled trigger.

One struggle we encountered was when creating the serving application. We kept running into the problem where the outcome kept being the same regardless of the input. The response page kept returning the same text of "It is unlikely that this is a phishing link.". We did not know what was causing the problem. To solve this we added a couple of different logs into our app.py file to get a better idea of what was happening as we could not see this when running the code in the virtual machine. With this method we found that the problem was not in our model or in transferring the input data which we initially thought the error would be in. The problem was in sending the predicted value to the response_page.html, we sent a string "True" or "False" to the html template instead of a boolean value. Since we sent a string each time it would print the same output because the if statement only worked for a boolean true value and not string, therefore always printing the other option of the if statement. After changing the predicted value to a boolean, the problem was solved and the application worked.

4. Individual Contributions

Dividing the tasks effectively for this assignment was essential for maximizing productivity, leveraging each member's strengths, learning new skills, and ensuring timely completion. Our team consists of a mix of different backgrounds and varying levels of experience in data engineering and different strong points. The distribution of tasks was determined so that everyone could learn new skills while also being able to use their strengths.

The individuals task division looked like this:

- Sander: Main focus was creating the CI/CD triggers. Created the triggers for the retraining of the model (with the pipeline executor included) and the triggers for the service redeployment (manual, invoked by retraining trigger, or through code changes). Also added the automatic trigger of the service redeployment in the ML pipeline. Wrote the design and implementation section.
- Mats: Main focus was on designing and creating the complete ML pipeline (except for the trigger of the service deployment at the end). He also assisted in preparing our relevant dataset. He also wrote the reflection section of this paper.
- Ibrahim: Worked on dataset selection, preparing the data through EDA, and eventually determining what the relevant subset of the dataset is. He also contributed to the writing of the Design and Implementation section.

- Kenneth: Created the dockerized images for the prediction-ui and prediction-api component, containerized the components, and made sure they were connected with each other through a network. Also worked on finding relevant papers for the introduction/overview, and wrote the introduction/overview of the paper.
- Sietse: Created the UI and API app components, as well as the final flowchart when the project was complete. The design of the UI was created by him, as well as making it work using regular expressions to extract the relevant data values. He also assisted in the EDA of our dataset, and model selection after Ibrahim and Kenneth evaluated the various options in our field.

As can be seen in the eventual task division, we all did our part, and worked a lot together on location. Because only some of us had experience with these kinds of applications, we met up and worked together on the project to learn the functions of such a MLOps system.

References

Abdelhamid, N., Thabtah, F.A., & Abdel-jaber, H. (2017). Phishing detection: A recent intelligent machine learning comparison based on models content and features. *2017 IEEE International Conference on Intelligence and Security Informatics (ISI)*, 72-77.

Anti-Phishing Working Group (APWG). (2024). *Phishing Activity Trends Report*.
<https://apwg.org/trendsreports/>

Gupta, B.B., Arachchilage, N.A., & Psannis, K.E. (2017). Defending against phishing attacks: taxonomy of methods, current issues and future directions. *Telecommunication Systems*, 67, 247 - 267.

Jang-Jaccard, J., & Nepal, S. (2014). A survey of emerging threats in cybersecurity. *Journal of Computer and System Sciences*, 80(5), 973-993. <https://doi.org/10.1016/j.jcss.2014.02.005>

Kreuzberger, D., Kühl, N., & Hirschl, S. (2022). Machine Learning Operations (MLOps): Overview, Definition, and Architecture. *IEEE Access*, 11, 31866-31879.

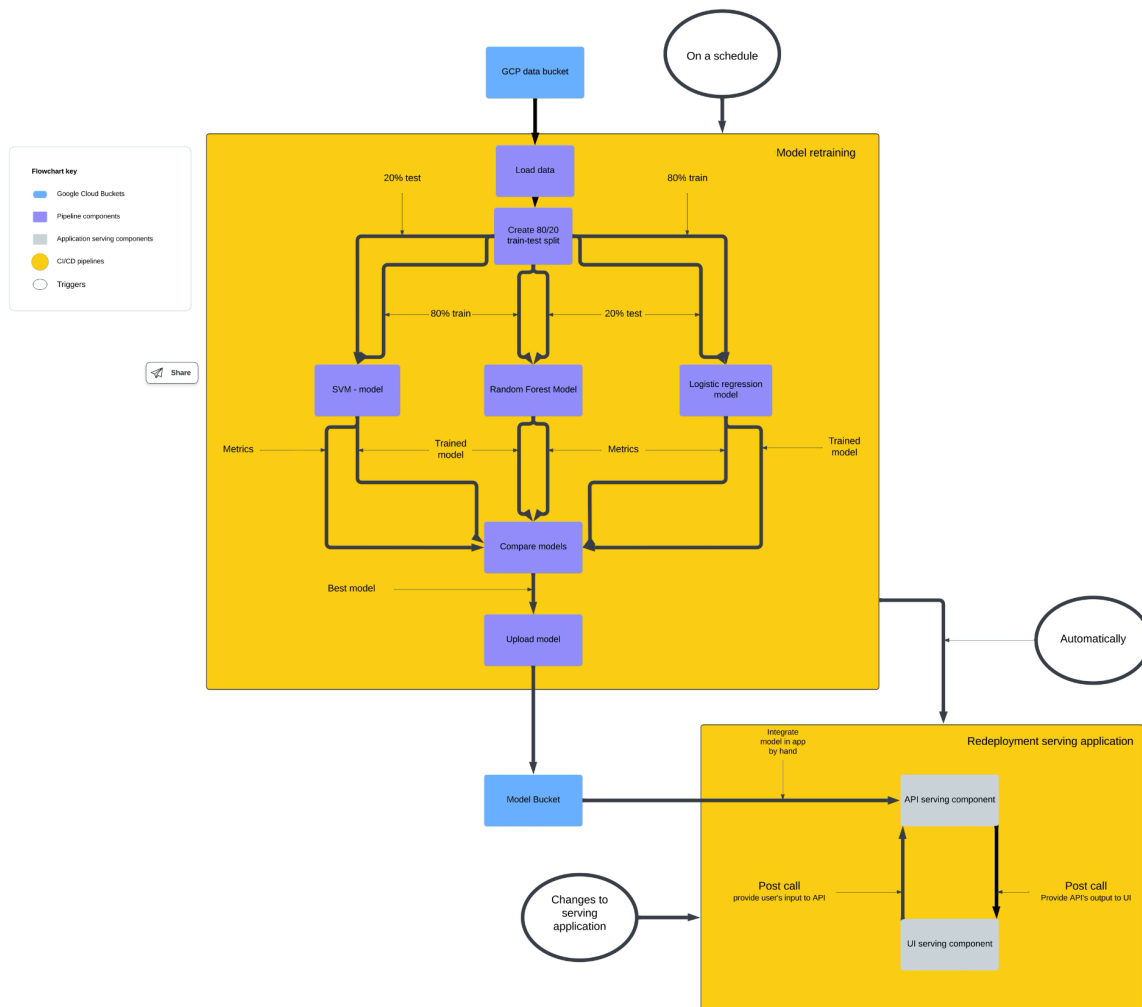
Restack. (2023). MLOps Strategies for Scalable Machine Learning. Retrieved from <https://www.restack.io/p/mlops-answer-scalable-machine-learning-strategies-cat-ai>

Sahami, M., Dumais, S.T., Heckerman, D.E., & Horvitz, E. (1998). A Bayesian Approach to Filtering Junk E-Mail. *AAAI Conference on Artificial Intelligence*.

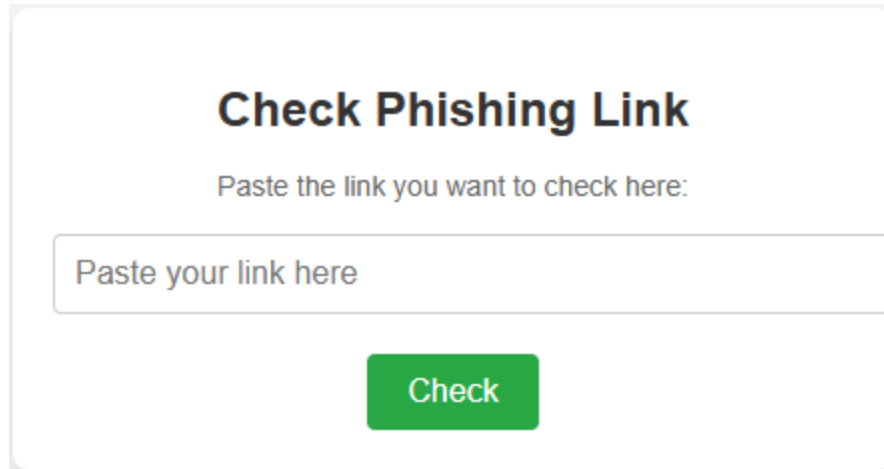
Sculley, D., Holt, G., Golovin, D., Davydov, E., Phillips, T., Ebner, D., Chaudhary, V., Young, M., Crespo, J., & Dennison, D. (2015). Hidden Technical Debt in Machine Learning Systems. *Neural Information Processing Systems*.

Tan, Choon Lin (2018), "Phishing Dataset for Machine Learning: Feature Evaluation", Mendeley Data, V1, doi: 10.17632/h3cgnj8hft.1

Appendices

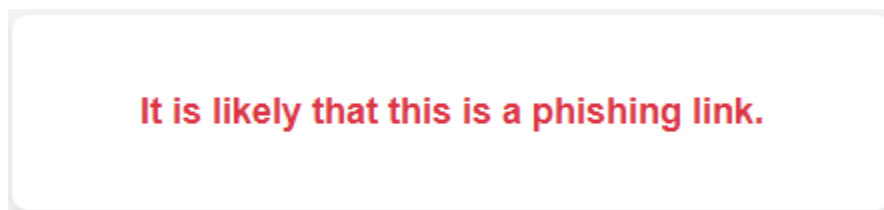


Appendix A: Flowchart of complete MLOps system



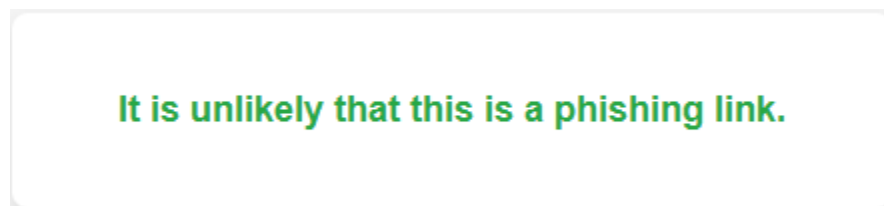
The screenshot shows a web interface for checking phishing links. At the top, the title "Check Phishing Link" is displayed in a bold, dark font. Below the title, a subtitle "Paste the link you want to check here:" is shown in a smaller, lighter font. Underneath the subtitle is a text input field with the placeholder text "Paste your link here". Below the input field is a green button with the text "Check" in white.

Appendix B: The input page of the serving application



The screenshot shows a response message in a light gray box with rounded corners. The text "It is likely that this is a phishing link." is displayed in a bold, red font.

Appendix C: The response page of the serving application if the model predicts the URL to be a phishing link



The screenshot shows a response message in a light gray box with rounded corners. The text "It is unlikely that this is a phishing link." is displayed in a bold, green font.

Appendix D: The response page of the serving application if the model predicts the URL to not be a phishing link