A6 REMLA Project Report – Group 4

Smruti Kshirsagar, Lucian Negru
Razvan Popescu, Ming Da Yang

1 INTRODUCTION

Phishing represents a cyber attack where malicious users impersonate legitimate organizations or individuals through misleading communications, typically messages, emails, and websites, to trick victims into revealing sensitive information such as passwords, banking information, or personal identification details. Moreover, URL phishing is a specific type of phishing attack where attackers create malicious websites that mimic legitimate ones to deceive users into providing such sensitive information. These attacks involve the use of modified URLs that closely resemble those of trusted organizations, often using slight misspellings or variations in domain names to trick users. This project[1] explores the use of Convolutional Neural Networks (CNNs) in detecting this type of phishing URLs.

2 RELEASE PIPELINE DOCUMENTATION

In this section, we document the process of making a new or updated version of a software package and a container image.

2.1 Software Package Release Pipeline

We are documenting the release pipeline for the Python package $lib-ml$. It hosts a library responsible for handling the preprocessing logic applied to data utilized in model training, tokenization, and queries. The pipeline is divided into two jobs to maintain a clear separation of concerns and facilitate better organization and management of the release workflow. The first job focuses on code quality assurance through linting, testing, and coverage analysis, and it is triggered on every push or pull request targeting the *main* branch. The purpose of the second job is to handle versioning, building, and publishing of the library, and it is triggered only when a new version tag ($v*$) is created and pushed, ensuring that the release process is initiated only for new version releases.

2.1.1 Pipeline Steps. The pipeline steps of the quality assurance job are the following:

(1) **Checkout code:** Retrieves the codebase.
(2) **Set up Python:** Configures the Python environment using Python version 3.x.
(3) **Install Poetry:** Install Poetry for Python dependency management.
(4) **Install dependencies:** Installs project dependencies using Poetry.
(5) **Lint with Pylint:** Analyzes Python files for errors and coding standards violations using Pylint.
(6) **Lint with Flake8:** Checks Python code against coding style conventions using flake8.
(7) **Run tests with coverage:** Executes tests with Pytest and generates coverage reports using Coveragepy.
(8) **Check coverage threshold:** Ensures test coverage meets a defined threshold (100% coverage in this case).
(9) **Upload coverage report and badge:** Uploads HTML coverage report and coverage badge artifacts.
(10) **Clean Up:** Clean up unnecessary files and resources.

When it comes to the building and publishing job, these steps are used:

(1) **Checkout code:** Retrieves the codebase.
(2) **Set up Python:** Configures the Python environment using Python version 3.x.
(3) **Install Poetry:** Installs Poetry for Python dependency management.
(4) **Install dependencies:** Installs project dependencies using $Poetry$.
(5) **Update lib version:** uses $update\_version.sh$ script to fetch the latest Git tag version and adjust the version in the $pyproject.toml$ file accordingly.
(6) **Build lib:** Compiles the source code and packages it into distributable formats using Poetry.
(7) **Publish lib:** Publishes the built library to Python Package Index (PyPI) using twine.
(8) **Clean Up:** Clean up unnecessary files and resources.

2.1.2 Purpose & Implementation. The first job starts by retrieving the latest version of the codebase from a version control repository to ensure that the most recent updates are available for subsequent stages. This is done using $actions/checkout@v4$. Following this, the Python environment is configured using $actions/setup-python@v4$, with Python version 3.x. This ensures that the correct Python version is consistently utilized during the pipeline execution. Python dependency management is streamlined through $Poetry$, a tool that simplifies package installation and versioning, which can be obtained using $snok/install-poetry@v1$. Once dependencies are installed, the environment is prepared for both development and testing purposes. To uphold code quality, Python files undergo a thorough analysis for errors and adherence to coding standards using $pylint$. Similarly, code is analyzed against style conventions to ensure uniform formatting and readability with $flake8$. Test coverage reports are generated using $coveragepy$ to evaluate how thoroughly the code is tested using unit testing. A coverage threshold of 100% is set to ensure that every executable line of code is exercised by the test suite. Upon completion of testing, artifacts are created using $actions/upload-artifact@v2$ to document comprehensive test coverage results, facilitating complete review and analysis. Following that, unnecessary files and resources are removed to maintain an efficient and clutter-free development environment.

Regarding the second job in the pipeline, the primary differences lie in steps 5, 6, and 7 when compared to the previous job. Step 5 determines the version number for the library. It uses the $update\_version.sh$ script to fetch the latest Git tag version and

[1] GitHub organization that contains the whole project can be found using this link: https://github.com/Release-Engineering-4

adjust the version number in the project's configuration file *pyproject.toml*. This step ensures that the library's version follows semantic versioning conventions and reflects the latest changes. The next step compiles the source code and packages it into distributable formats (e.g., wheel, source distribution) using *Poetry*. It generates the final artifacts that can be published to the package registry. The following step uses the built artifacts and publishes them to *PyPI* using the *twine* utility tool. This makes the library available for installation and use by other Python developers and projects.

2.1.3 Pipeline Data Flow. For the first job, the dataflow starts with the retrieval of the codebase, followed by the configuration of the Python environment. Project dependencies are installed next, creating a virtual environment with the necessary libraries. The code is then analyzed for errors and coding standards violations, producing linting reports. Testing generates test results and coverage data, which are subsequently evaluated against a predefined coverage threshold. If the threshold is met, HTML coverage report artifacts are uploaded. The process concludes with the cleanup of unnecessary files and resources.

For the second job, the initial part of the data flow remains the same as in the previous job. Then, if a new Git tag has been pushed, the library version is updated accordingly. The source code is then compiled and packaged into distributable formats. These packages, representing the main artifact, are subsequently published to the package manager. Finally, unnecessary files and resources are cleaned up.

2.2 Container Image Release Pipeline

We are documenting the release pipeline for $model - service$. This repository automates the process of building and deploying a containerized wrapper service for our pre-trained ML model. The service exposes the model via a REST API, ensuring scalability and accessibility to other components. This release pipeline is also divided into multiple distinct jobs to have a clear separation of concerns. The first job ensures code quality by analyzing Python files for errors and adherence to coding standards, while the second one focuses on testing and coverage reports. The last job automates the process of building and deploying container images for the model service. Triggered by new version tags ($v*$), it logs into the GitHub Container Registry (GHCR), builds the container image, and pushes it, facilitating the release of the model service. Together, these jobs ensure a streamlined and reliable release process for the containerized model service. Since the quality assurance and testing jobs follow similar steps as those in the $lib - ml$ pipeline, we will specifically focus on the image-building job in the following subsections.

2.2.1 Pipeline steps. The following steps are followed to build and push the container image:

(1) **Checkout code:** Retrieves the codebase
(2) **Login into GHCR:** Logs into the GitHub Container Registry (GHCR) with GitHub actor and token.
(3) **Build & push image:** Build the Docker image, tag it with the Git version, and push it to the GHCR.
(4) **Clean Up:** Clean up unnecessary files and resources.

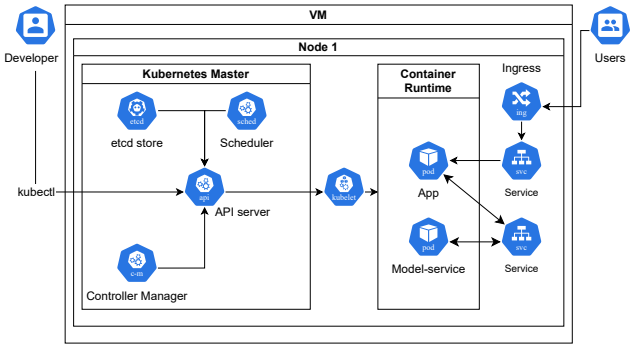

Figure 1: Diagram of Kubernetes deployment using Minikube

2.2.2 Purpose & Implementation. The job starts by retrieving the codebase, as explained in the software package release pipeline. This is crucial as it allows the workflow to work with the most recent changes and updates in the repository. The following step involves logging into the GHCR using the GitHub actor and token, accomplished through $docker/login - action@v1$. This authentication process is necessary to gain the appropriate permissions to push Docker images to the registry, ensuring secure and authorized access. Finally, the Docker image is built and tagged with the Git version, providing a unique and traceable identifier for the image. Once the image artifact is built, it is pushed to GHCR, making it available for deployment and use in various environments. After this, a cleanup step is performed to remove unnecessary files and resources.

2.2.3 Pipeline Data Flow. The workflow begins by retrieving the codebase, ensuring the latest version of the source code is available for the build process. It then logs into the GHCR for secure access. Following this, the Docker image is built, tagged with the Git version, and pushed to GHCR, creating a uniquely identifiable artifact. Finally, the workflow performs a cleanup, removing unnecessary files and resources.

3 DEPLOYMENT DOCUMENTATION

The current deployment of the application is done through a local Kubernetes cluster using Minikube. Currently, a skeleton framework exists on GitHub using Vagrant and Ansible playbooks to set up a multi-machine Kubernetes network, however, the setup for a multi-machine Kubernetes network was left out since it was not feasible within the timeframe. Looking at figure 1, we can observe the structure of the deployment. Minikube hosts its Kubernetes cluster in a single-node virtual machine. The Developer interacts with the API server of the Kubernetes Master using kubectl, which in turn interacts with the Container Runtime. The Ingress exposes the service to the users, which serves the application. The application consists of two container images, the frontend app and the model-service which the app queries.

The `deploy.yml` file uses a deployment of the app using the app image from the package registry of the *app* repository, which is set to have 2 replicas. It also creates a service for the app. Alongside it spawns an Ingress which exposes the app to the outside of the

cluster. Then it creates a pod which hosts the model-service container image from the *model-service* repository. The file also creates a service for the model-service pod. Lastly, we spawn a ServiceMonitor which monitors the app service and collects metrics for Prometheus.

4 EXTENSION PROPOSAL

In this section, we critically reflect on the current stage of the project, and identify shortcomings and improvements that can be made.

4.1 Project state

The code base of all repositories is in one GitHub *organization*. The GitHub organization contains 6 repositories, namely *operation*, *model−service*, *model−training*, *app*, *lib−ml* and *lib−version*. The operation repository deploys the whole application to a Kubernetes cluster as can be seen in figure 2. The pipeline and interaction between the different repositories are explained in section 7. In short, the app image is the web-app that the user interacts with. The app queries a model-service instance to fetch predictions. Model-service is a Flask that hosts REST API endpoints for prediction. Model-training creates and trains models and the trained model is packaged into the model image.

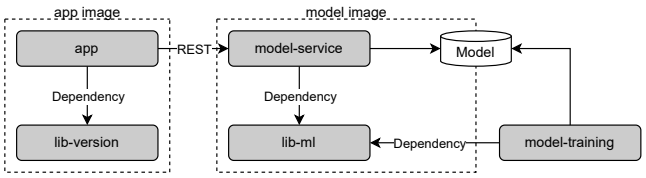

Figure 2: The architecture of the application. It shows how the different repositories interact with each other.

4.2 Shortcomings

The biggest shortcoming of the current project is the missing multi-VM Kubernetes deployment using Vagrant. Due to time constraints and making sure other parts of the project were working we decided to move forth on the development of other aspects. The development and deployment of the application were done on a local Kubernetes cluster using Minikube. In the current state, the Vagrant setup of multi-VM's and provisioning the environment with Ansble is working correctly. The recommended Kubernetes distribution, Minikube and k3d do not have support to spread out the Kubernetes cluster over multiple machines. The impact of this means that we cannot test and develop the application in an isolated setting without affecting live systems or simulating a real-world cluster environment for testing Kubernetes' scalability and behaviour.

Currently, the front-end of the web-app application is built using basic HTML templates. Every time the user interacts with the site it reloads the whole page, even if only 2 elements are updated. This is bad practice. If the web app grows in size it may lead to longer load times and unnecessary data transfer.

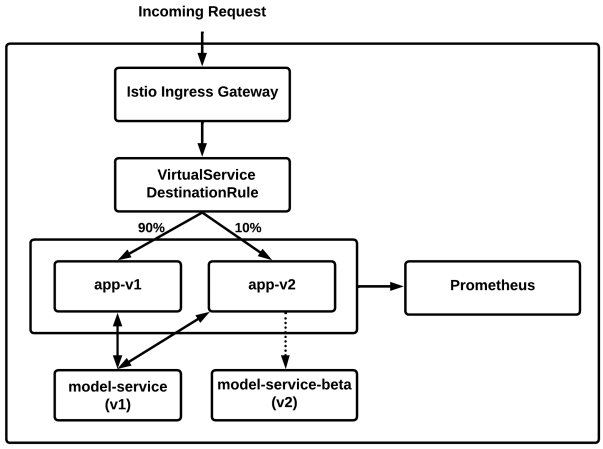

Figure 3: Experiment Design

4.3 Improvements

To allow provisioning and to create a development environment that mimics a real-world cluster environment, a correct Kubernetes distribution must be selected that allows multi-machine clusters to be set up. One Kubernetes distribution that allows for this could be *k3s*. According to the article, this distribution allows for multi-machine clusters, thus solving our previous problem. To test whether it would work, we would need to modify the Ansible playbooks to install the distribution, create a script that appoints the control node to be the controller plane and make the worker nodes join the controller plane.

To solve the problem of the web-app reloading the whole page to update its elements, a solution would be to refactor the project to use ReactJS. This allows the page to be dynamically updated so the user does not have to reload the whole page when interacting with the page [1]. If implemented, the page would not be reloaded once the user interacts with it, i.e. checking if a link is a phishing link should display the result of the API call without reloading the page.

5 ADDITIONAL USE CASE FOR ISTIO

The Istio service mesh can be used for traffic management and continuous experimentation involves leveraging several of its key features such as routing, load balancing, traffic mirroring, and canary deployments. To implement this for our Phishing web app, we introduce an extension where the user of the app can provide feedback on whether the prediction was correct or incorrect. This enables us to collect metrics which can help ascertain the model's accuracy in real-time.

The experiment type selected is Shadow Launch. A shadow launch is a technique where production traffic and data are run through a newly deployed version of a service or machine learning model, without that service or model actually returning the response or prediction to customers/other systems. In our case, it is done for a portion of the traffic to *app* (10%). Two versions of model-service are used, which run as separate services. The app fetches

the response using the original model, which is the production model (Model A) and asks the user for feedback. If a user provides feedback, the accuracy metrics are updated accordingly and this URL is sent to the model to be tested (Model B). The prediction of the model is compared to the prediction given by the original model and feedback, and accuracy metrics are updated for the beta model. This implementation is configured using Istio.

6 EXPERIMENTAL SETUP

The flow of the experimental setup is shown in Figure 3. Two instances of model-service are created, one which contains the live production version (Model A) and another which has the model which is to be tested (Model B). Both of these are deployed using separate services: model-service and model-service-beta. The app code is modified to accommodate user feedback. A flag is used to indicate whether the app should be used for the experiment. This flag is set using an environment variable which can be passed with the deployment. The service endpoints are also passed as environment variables. Hence, the test environment for the app is configurable through the deployment yaml. If the flag is set to *false*, the app will function as originally intended, without the extension.

The test user interface shows that it will be used for beta model testing. Upon submitting the URL to be classified, the app will show whether the URL is phishing or not based on Model A's prediction. Users will be asked to provide feedback about whether the prediction was correct. Counters are set up in Prometheus to capture correct and incorrect predictions to evaluate the accuracy of Model A on real-time data. In the experimental web app instance, upon submitting the feedback, the app flask service will send the request to Model B. Metrics for this model will also be updated accordingly. This will happen in the background, hence it is a shadow launch.

Istio is used to set up two versions of the app, one of which will send requests to just Model A and one which will send requests to both Model A & B. These are created as Kubernetes Deployments. Traffic management is done using weights so that only 10% of the traffic is routed to the experimental app. However, as mentioned above, the deployment can be easily modified to have both instances of the app sending requests to both models by setting the flag in the container's environment variables, or varying the weights as well. Both versions of the app are configured to send metrics to Prometheus. Two Deployments and Services are created for the Model A and Model B. The Deployment for Service *model-service-beta* is created using an image that utilizes Model B. Istio Ingress gateway is created to serve the app. VirtualService and DestinationRule are used to create a load balancer that routes the incoming traffic to specific instances of the app. The setup is shown in Figure 4. Kiali Dashboard in Figure 5 shows that the configurations are correct and the Gateway setup is working with incoming traffic being recorded. Replicas can be created to have more pods serving the app.

The experiment can be performed on a local machine by running the following steps:

- Download Istio
- Run Minikube cluster and install Istio on it
- Apply addons and enable Istio injection

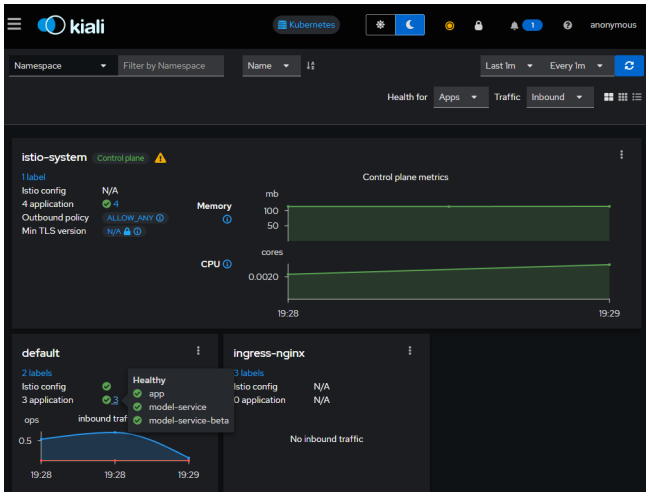

Figure 4: Kiali Validation

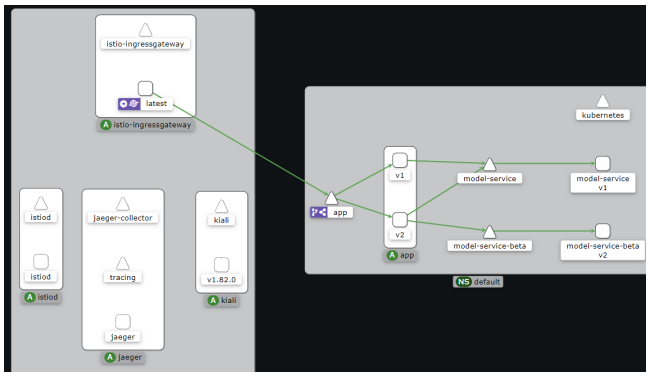

Figure 5: Traffic Flow on Kiali

- Apply the istio-canary.yml
- App will run and can be accessed through the tunnel and be monitored using Kiali and Prometheus

Prometheus is configured to scrape the app for metrics. These metrics are collected and can be viewed on the Prometheus Dashboard. Figure 6 shows the results of the experiment. Prometheus visualises the accuracy values for both Model A and B. This can be used to compare the performance of the new model to the original on real-time traffic. In our case, the new model makes similar predictions to the original model and hence the graphs are overlapping. We can see that *model_accuracy* and *beta_model_accuracy* are being recorded and compared using a PromQL query. Due to time constraints, we were not able to run the experiment on a large number of varying requests, which would provide a better view of model accuracies. This experiment can be expanded to test response time and other metrics and also inspect issues within both models if accuracy values drop.

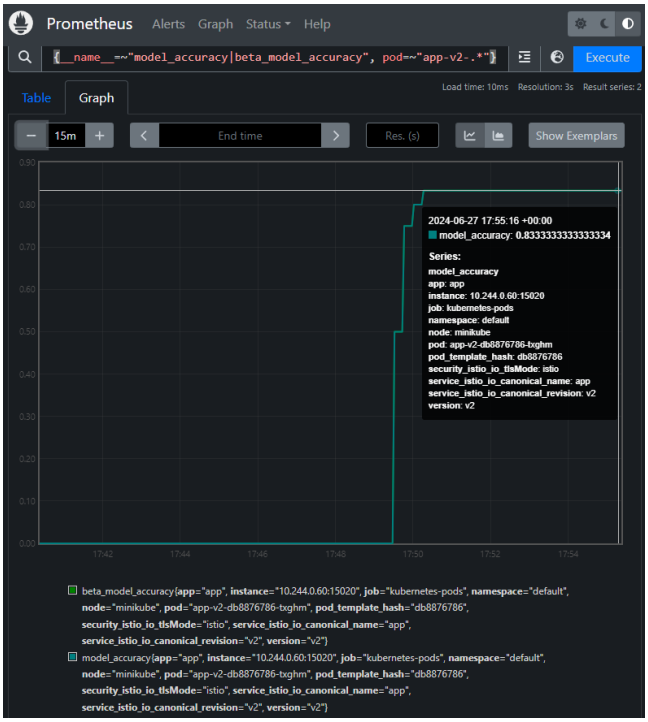

Figure 6: Istio Experiment Results

7 ML PIPELINE

This section outlines the setup and configuration of our ML pipeline, including the tools chosen, design considerations, and automated tasks. We describe the pipeline's steps and the artifacts it generates, highlighting the decisions made to optimize project configuration. Specifically, we are documenting the pipeline for *model – training*. This repository handles the processes of training, predicting, and evaluating the model, ensuring they occur reproducibly while also versioning the generated artifacts and trained models.

7.1 Technology Stack

We use a *Cookiecutter* template to structure our project, ensuring a consistent and standardized project setup that promotes best practices and simplifies initialization. Our pipeline operates on Python version >= 3.10 and < 3.11, benefiting from its extensive library support and compatibility with the most recent tools and technologies. Furthermore, we use *Poetry* to manage dependencies and package our project. *Poetry* simplifies dependency management by providing a single tool for handling package installation, virtual environments, and project packaging. It ensures reproducibility by locking dependencies in a *poetry.lock* file, enabling consistent builds across different environments.

We also employ *Pylint* and *Flake*8 for code quality and style checks in our development process. *Pylint* analyzes Python code for errors and potential bugs and adheres to coding standards, providing detailed feedback to ensure code consistency and quality. On the other hand, *Flake*8 focuses on enforcing coding style guidelines, to maintain clean and readable code. In addition, we utilize *Pytest* and *Coveragepy* for testing and code coverage analysis in our pipeline. *Pytest* is a powerful testing framework for Python that allows us to write simple and scalable tests. It provides a wide range of features for test discovery, fixtures, and assertions, making it highly flexible and easy to use. *Coveragepy*, on the other hand, measures the extent to which our codebase is covered by tests. It generates reports that show which parts of our code have been tested and which have not. This helps us identify areas that require additional testing and ensures that our tests effectively validate the functionality of our code.

Lastly, we use Data Version Control (DVC) for managing and versioning our data and models in the development lifecycle. DVC helps us automate the pipeline and track changes to datasets, models, and experiments, facilitating reproducibility. DVC is advantageous because it separates data and model management from code, enabling efficient versioning of datasets and models without bloating version control systems like Git. Moreover, it provides robust support for remote storage via *Google Drive*, allowing us to store and access large datasets and models efficiently. By utilizing DVC, we make sure that our project maintains traceable and reproducible workflows, necessary for collaboration and scalability.

7.2 Design Decisions

In designing our ML pipeline, we prioritized reproducibility, modularity, scalability, automation, data and model versioning, and environment consistency. Reproducibility was a key focus, achieved through consistent environments and DVC for versioning data, models, and experiments. Our pipeline is modular, meaning each stage is a distinct unit that can be managed independently. This modularity simplifies maintenance and updates while ensuring smooth data flow through the pipeline. We designed the pipeline with scalability in mind, which allows for adjustments in data processing and model training to accommodate larger datasets and more complex models. Automation was integrated through DVC and Git workflows, improving efficiency. Lastly, we used Poetry for managing dependencies and creating consistent environments across development stages, ensuring that the pipeline behaves uniformly and reliably throughout its lifecycle.

7.3 Pipeline Stages

Figure 7 presents our pipeline which consists of five stages, each playing a crucial role in the overall workflow. Initial data and other interim results can be fetched from and stored on Google Drive using DVC.

(1) **Data pre-processing:** This stage prepares the raw data (URLs and corresponding labels) for the subsequent stages of the pipeline, using the *lib – ml* library. This involves tokenizing, padding, and encoding the data to ensure it is in the correct format for training and evaluation. The process includes loading the raw text data, splitting it into training, testing, and validation sets, fitting the tokenizer on it, and then applying tokenization and padding. The outputs of this stage are the pre-processed data and the trained tokenizer together with the character index.
(2) **Model creation:** This stage involves defining the architecture of the URL phishing detection model, using the *Keras*

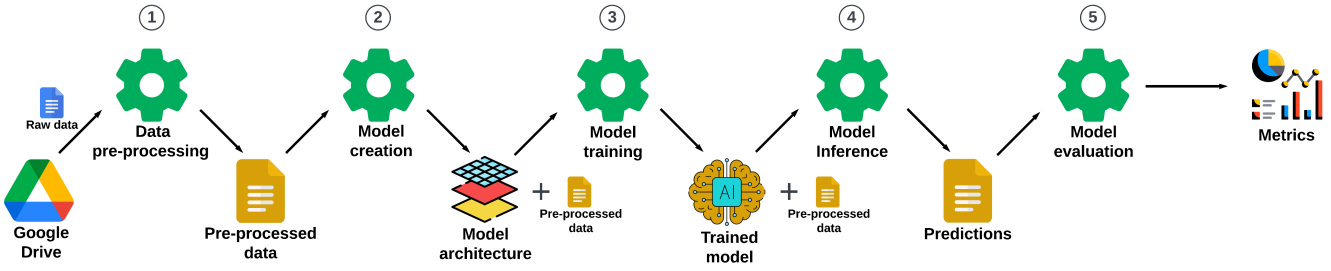

Figure 7: ML pipeline

library. The process begins by loading the character index, a mapping of characters to their corresponding integer indices, which is essential for the embedding layer. The model is then constructed by incorporating several layers such as Embedding, Conv1D, MaxPooling1D, Dropout, Flatten, and Dense layers with a sigmoid activation function. The output of this stage is the defined model architecture.

(3) **Model training:** This stage begins by compiling the model with specified parameters including binary cross entropy as loss function, adam optimizer, and evaluation metrics such as accuracy. After compilation, the model is trained for 30 epochs using a batch size of 5000 and validation data to monitor performance. The output of this stage is the trained model.

(4) **Model inference:** This stage uses the trained model to generate predictions on the test dataset. After prediction, the probabilities are converted into binary labels using a threshold of 0.5, and the true labels are reshaped for consistency. The outputs of this stage are the reshaped true labels and the binary predictions.

(5) **Model evaluation:** Using the reshaped true labels and predicted binary labels, this stage calculates a classification report detailing precision, recall, F1-score, and support for each class. Additionally, it computes a confusion matrix to visualize true positive/negative and false positive/negative predictions, displaying accuracy as well. The output of this stage is a JSON file with the accuracy, ROC AUC score, and F1-score that can be used for further analysis.

7.4 Automated Tasks

Automation plays an important role in our ML pipeline through both DVC and Git workflow. DVC automates data management and versioning across stages such as data pre-processing, model creation, training, inference, and evaluation. It tracks datasets, model architectures, hyperparameters, and evaluation metrics, ensuring reproducibility and allowing comparisons between model iterations. Simultaneously, the Git workflow automates code quality through linting tools such as *Pylint* and *Flake*8, enforcing coding standards and enhancing code readability. Automated testing frameworks such as *Pytest* validate the functionality of our pipeline, ensuring that changes do not compromise existing features. Additionally, coverage reports generated by *Coveragepy* are automated within the workflow, providing insights into the codebase's test coverage. Together, these automated processes streamline development, improve reliability, and support efficient iteration in our ML project.

7.5 Artifacts

The ML pipeline generates essential artifacts across its stages to support development, evaluation, and deployment. These include raw data, pre-processed datasets, tokenizers, character indices, model architecture, trained models, binary predictions, and evaluation metrics (accuracy, F1-score, ROC-AUC). These artifacts, along with coverage reports from automated tests, ensure transparency, reproducibility, and efficiency throughout the pipeline's lifecycle.

8 ML TESTING DESIGN

This section delves into the ML testing pipeline we implemented, detailing our testing strategy and the automated tests we developed. Our approach includes tests for features and data integrity, model development and infrastructure, performance evaluation, and monitoring. We document the rationale behind our choices, analyze and evaluate the limitations of our current testing approach, and explore potential improvements. Furthermore, we propose additional tests to enhance the robustness and reliability of our model.

8.1 Design Decisions

In designing our ML testing pipeline, we aimed to ensure our model training and inference process's robustness, reliability, and efficiency. Our primary goals were to create a resilient system capable of handling non-deterministic behaviours, to verify the integrity of features and data, and to ensure that the entire ML infrastructure operates smoothly. We also aimed to meet non-functional requirements such as optimal memory usage and performance. By implementing comprehensive unit tests (using *pytest*), and evaluating test adequacy metrics (obtained using *coveragepy*), we sought to build a reliable framework that guarantees high-quality outcomes and facilitates continuous monitoring and improvement of our models. We chose unit tests because of their feasibility with our distributed system.

8.2 Pipeline Stages

We automated our testing process through GitHub workflows to ensure thorough validation for every push and pull request (to the *main* branch), maintaining high standards of code quality. This approach allows us to promptly validate changes and maintain code

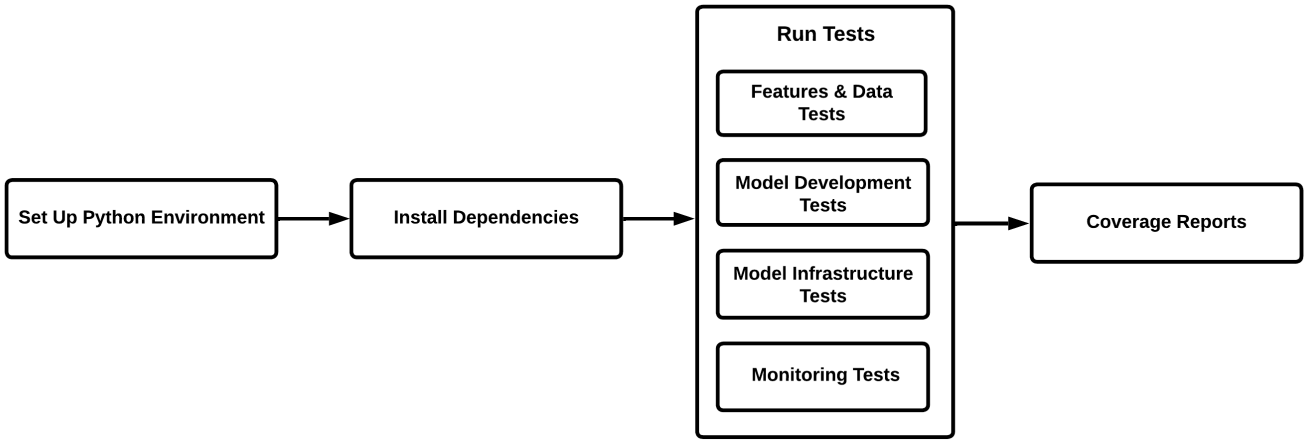

Figure 8: General overview of ML testing workflow

quality by running comprehensive tests automatically. Coverage reports are integrated into our workflow, providing detailed insights into the extent of code covered by our tests. This automation not only improves efficiency by quickly identifying potential issues but also enhances reliability by maintaining consistent testing standards across our model training pipeline. Overall, these practices enable us to deliver and maintain robust and efficient machine learning solutions confidently. Our automated testing workflow, illustrated in Figure 8, comprises the following types of testing:

(1) **Feature & Data Tests:** These tests ensure the integrity and correctness of input data and feature engineering processes, validating that the data used for training is accurate and consistent.
(2) **Model Development:** These tests ensure the complete development of the ML model by evaluating reliability, performance metrics, and potential biases through methods such as code reviews, hyperparameter tuning, baseline comparisons, and analysis across diverse data slices and timeframes.
(3) **Model Infrastructure Tests:** These tests ensure the stability and functionality of the entire machine learning pipeline, from data ingestion to model serving, validating that each component integrates seamlessly and performs as expected under various conditions and inputs.
(4) **Monitoring Tests:** These tests ensure that the ML model and its infrastructure maintain stability, reliability, and performance by checking data consistency, model staleness, errors, performance issues, and resource utilization.

8.3 Features & Data Integrity Tests

Our feature and data tests focus on ensuring the integrity and correctness of input data and the processes involved in feature engineering. To achieve this, we implemented a series of tests that validate various aspects of the data pre-processing stage.

Ensuring the robustness and reliability of our data pre-processing is essential. We verify the distribution of labels within the dataset,

```
assert len(unique_labels) == 2, "Expected two unique labels"
assert 'legitimate' in unique_labels, "Expected 'legitimate' label"
assert 'phishing' in unique_labels, "Expected 'phishing' label"
```

Figure 9: Assertions verifying the correct distribution of labels in the dataset

as shown in Figure 9, to guarantee a balanced representation of *phishing* and *legitimate* classes. This balance is critical for training a model that generalizes well across different data categories, preventing bias. Moreover, we check the average length of URLs to ensure they fall within a reasonable range, setting thresholds between 10 and 80 characters, as illustrated in Figure 10. This prevents URLs from being excessively long, which could introduce noise and complexity, or too short, which might lack sufficient detail for effective modelling.

We also examine the dataset for potential null URLs to prevent disruptions during data processing, as null URLs can lead to errors during model training or inference. Ensuring there are no null values helps us maintain data integrity and consistency throughout the pre-processing stage. Similarly, we confirm the absence of null labels in the dataset. Null labels can make training data unusable, leading to ineffective model training. By securing data completeness in labels, we prevent errors and ensure our model learns effectively from the provided data, maintaining the overall quality and reliability of our dataset.

```
assert split_test_data["url_length"].mean() <= 80, "Average URL
length exceeds 80 characters"
assert split_test_data["url_length"].mean() >= 10, "Average URL
length is less than 10 characters"
```

Figure 10: Assertions verifying the average URL length in the dataset

Furthermore, we also assess the consistency and presence of data splits (train, test, validation) within the dataset. By verifying the proper partitioning into these distinct subsets, we ensure that our model can be evaluated accurately, leading to better generalization and performance. To further evaluate the robustness of our pre-processing stage, we test its handling of data shuffling. This confirms that the pre-processing steps, such as tokenization and encoding, produce consistent results even when the input data order is altered. Robust preprocessing is essential to maintain stable model performance, ensuring the model is not sensitive to the sequence in which data is presented.

Additionally, we validate the setup and existence of processed data and tokenizer directories, ensuring that necessary directories for storing processed data and tokenizer artifacts are properly created and accessible. Finally, we perform parametrized testing to confirm the presence and non-emptiness of specific data slices within the processed data. By making sure these slices are properly populated, we guarantee that the model is exposed to a diverse range of data, improving its ability to generalize and perform effectively across various scenarios.

8.4 Model Development Tests

The model development tests ensure the correct implementation and robustness of our model by validating its architecture, verifying resource loading, and ensuring proper model construction and saving. These tests guarantee reliable performance and accurate execution of the model training and inference processes.

We start by simulating the parameters managed by DVC through mocking, to ensure consistency and reproducibility during testing. This provides predefined parameters for the tokenizer path, input lengths, label categories, and model path. By doing this, we ensure that the model definition process receives consistent input parameters without relying on actual external configurations.

To test the loading of the tokenizer index, we mock the *load_pkl* method. Using $mock.patch.object$, we create a mock implementation of this method to return a controlled dictionary representing the tokenizer index. This allows us to verify that the model definition function correctly loads the necessary tokenizer index without interacting with the actual file system. Furthermore, we also mock the directory creation, path checking, and model saving functionalities, as previously explained, to verify the model is correctly saved to the specified path.

We also verify that the correct model architecture is created by checking the number of types of layers added to the model. Specifically, we make sure that the embedding layer is configured with the correct input dimensions, output dimensions, and input length, and that the dense layer is configured with the correct number of units and activation function. For example, we validate the embedding and dense layer configuration as in Figure 11.

$$assert\ isinstance(mock\_model.layers[0], Embedding)$$
$$assert\ isinstance(mock\_model.layers[-1], Dense)$$

Figure 11: Assertions verifying the correct types of the layers in the model

Furthermore, we assess how different hyperparameters influence our ML model training process and performance. This test is crucial in ensuring our model maintains robustness and consistent performance across various configurations enhancing its effectiveness. We utilize the *unittest.mock* to create controlled environments for our experiments. With the $train\_model$ function we simulate the model training process under different hyperparameter settings. Mock data and mock model instances are prepared using $pytest.fixture$, allowing us to simulate training and validation datasets as well as model behaviours. Parametrization plays a crucial role in this test as we vary hyperparameters such as batch size, number of epochs, optimizer type, and loss function across multiple iterations. Each configuration, as depicted in Figure 12, is tested to ensure our model's behaviour aligns with our expectations under diverse conditions.

```
    @pytest.mark.parametrize(
    "hyperparameter, values",
    [
        ("batch_train", [16, 32, 64]),
        ("epoch", [1, 2, 3]),
        ("optimizer", ["adam", "sgd", "rmsprop"]),
     ("loss_function", ["binary_crossentropy", "mse"]),
    ],
)
```

Figure 12: Hyperparameter configurations used for testing our ML model

During execution, we validate each configuration by asserting that the model is compiled correctly (*compile*), trained with the appropriate data and settings (fit), and saved to the designated location (*save*). These assertions, performed using *assert_called_once_with* on mock model methods, confirm the model's adherence to expected behaviours throughout the training process.

8.5 Model Infrastructure Tests

The model infrastructure tests validate the stability and functionality of the entire ML pipeline. These tests ensure seamless integration and reliable performance of each pipeline component, under diverse conditions and input scenarios, to support consistent and efficient model deployment and operation.

We focus on evaluating our trained model against a simpler baseline model. To establish this comparison, we begin by constructing a basic neural network (*simple_model*) using the *Keras* framework, consisting of input and output layers defined by Dense with specific activation functions, such as *relu* or *sigmoid*. This foundational model is designed to serve as a benchmark for our main trained model.

Initially, data undergoes the pre-processing steps necessary for training our baseline model, including splitting, tokenization, padding, encoding, but also slicing, to speed up processing. Then, the baseline neural network model is trained using the specified parameters, including the stochastic gradient descent optimizer and mean squared error as a loss function. Throughout training, we monitor key performance metrics (which also account for monitoring tests)

such as training duration and memory usage to ensure efficient execution. These metrics help us gauge the model's computational demands and responsiveness during training. Following training, we evaluate its performance against URLs from the test data. We also compare its predictions with those generated by our trained model. This comparative analysis allows us to assess the predictive accuracy and generalization capabilities of our model.

Assertions within the function validate critical metrics: the duration of the training process (*duration*), the efficiency of prediction generation (*duration2*), memory usage during execution (*memory_usage*), and the comparative accuracy between our trained model (*accuracy_model_1*) and the baseline model (*accuracy_model_2*). These checks, illustrated in Figure 13, ensure that our machine learning pipeline operates efficiently within predefined performance thresholds, delivering reliable results for our predictive models.

```
assert duration < 150, "Training took too long"
assert duration2 < 5, "Predictions took too long"
assert memory_usage < 150 * 1024 * 1024, "Memory usage high"
assert accuracy_model_1 >= accuracy_model_2
```

Figure 13: Assertions to verify duration and memory metrics and also model accuracy

In the *model-service* repository we perform also infrastructure tests. This is because we test the prediction of the model. The tests test the output of the model by checking if the received response contains correct data.

Lastly, we run a *manual* test locally with the artifacts to compare the production model to a simpler model as a baseline. The simpler model is a logistic regression model trained on the same data set as the production model. Since the ML testing pipeline does not carry over any artifacts this test is only applicable when run locally.

8.6 Monitoring Tests

Monitoring tests are essential for maintaining the stability, reliability, and performance of our ML models and infrastructure. These tests diligently assess data consistency, model integrity, error rates, performance benchmarks, and resource utilization. By continuously monitoring these aspects, we ensure our systems operate optimally and deliver consistent, accurate results over time.

Besides the monitoring tests discussed in the section above, here we start by verifying that all URLs in our dataset have a non-zero length. This step is essential as it ensures that every URL provides meaningful information necessary for our ML tasks. Additionally, we test the balance between the *legitimate* and *phishing* label classes in our dataset, as presented in Figure 14. This balance is important for preventing bias and enabling our models to generalize effectively across different classes.

Moreover, in the *test_label_validity* test, we confirm the correctness of our dataset labels by ensuring they are correctly categorized as either *legitimate* or *phishing*. This validation step is key in assuring our model training proceeds with accurate label definitions, which in turn contributes to robust prediction performance. Finally, the *test_url_format* test evaluates the format of URLs within our dataset using a regex pattern to ensure compliance with valid URL structures. This validation step is essential for maintaining data integrity and facilitating effective processing and analysis.

```
assert (np.abs(class_counts[1] - class_counts[0]) <= 1),
"Class distribution is not balanced."
```

Figure 14: Assertion to verify class balance

8.7 Suggestions for Improvement

According to the paper *What's your ML Test Score? A rubric for ML production systems* [2], the production-readiness of an ML system can be evaluated using a rubric system. The paper also describes multiple aspects of automating and testing your ML system. Using the paper we can grade how our ML system performs with regards to automation and testing, which aspects have been tested, and which tests are yet to be implemented. One point is awarded for executing a test manually and two points are awarded if the system that runs the test automatically on a repeated basis. Currently, we have the following aspects implemented:

(1) **Test against a simpler model as a baseline** – automatic
(2) **Test that the distributions of each feature match your expectations** – automatic
(3) **Unit test model specification code** automatic
(4) **Test that data invariants hold in training and serving inputs** – automatic
(5) **Test for NaNs or infinities appearing in your model during training or serving** – automatic
(6) **Integration test the full ML pipeline** – manual

We automate one or two tests in each category presented in the paper. According to the paper we got a score Improvements could be made to include all missing tests and automate them through environment deployment using a tool such as Jenkins[2]. Another missing feature of the ML testing is the lack of metamorphic testing, which was one of the requirements for the assignments. Without it, we cannot be certain about the model's (non-)determinism.

REFERENCES

[1] allrounddiksha. 2024. React.js: The Complete Guide. https://medium.com/@allrounddiksha/react-js-the-complete-guide-e203aa81ac64. Accessed: June 19, 2024.
[2] Eric Breck, Shanqing Cai, Eric Nielsen, Michael Salib, and D. Sculley. 2016. What's your ML test score? A rubric for ML production systems.

[2] https://www.jenkins.io/