UNIVERSITÀ DEGLI STUDI DI SALERNO

Enhancing Online Security: Implementing a URL Analysis Tool for Phishing Prevention using Kubeflow and Kubernetes

Ingegneria Informatica

Corso di Laurea Magistrale LM-32

Secure Cloud Computing a.a. 2023/24

Giulia Minichiello	0622702127
Lucia Senatore	0622702089

Sommario

Overview

The aim of this project is to explore, train, and serve a machine learning model for either Classification or Time Series Data forecasting, leveraging the main components of Kubeflow. We are developing a containerized application that runs on a Kubernetes cluster.

Specifically, we have developed an application that predicts whether a URL contains phishing threats or is secure. The prediction model is trained using a Kubeflow pipeline, which will be described in detail in the following chapters.

To accommodate 50,000 customers and manage 5,000 simultaneous requests, the application is supported by a robustly designed Kubernetes cluster.

Introduction to the problem

Detecting phishing URLs is crucial for maintaining online security and safeguarding sensitive information. In phishing attacks, deceptive websites try to mimic legitimate ones to steal user data. These malicious sites cause the disclosure of personal and confidential information by unsuspecting users. Phishing is a growing threat in the digital era.

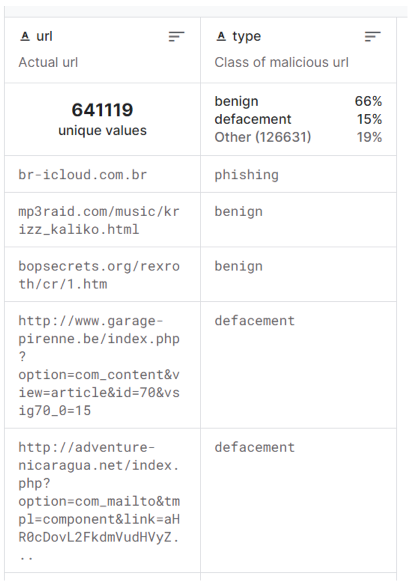
The threat can be reduced by identifying and reporting phishing URLs. This approach not only protects individual users, but also preserves the integrity and reliability of the Web. Effective detection mechanisms contribute to a safer environment, where users can navigate safely and know that their personal and professional information is not accessible by malicious attackers.

The distributed application is easy to use because it has been designed to meet the needs of computer security experts and users who are not as experienced. The application examines the characteristics of the URL that are commonly associated with phishing sites by simply entering it in a text field, making it easy for anyone to evaluate the potential dangers of a URL.

Dataset

The dataset used to train the model is "Malicious URLs dataset". It includes 651,191 URLs, categorized into benign (428,103), defacement (96,457), phishing (94,111), and malware (32,520) types. Sourced from five different origins, including the ISCX-URL-2016, Malware Domain Blacklist, Faizan Git repository, Phishtank, and PhishStorm datasets, it provides a comprehensive overview of various URL threats.

Due to the meticulous process used to compile this dataset and the substantial number of URLs collected from diverse sources, we consider it a reliable training resource for our model.



url Actual url	type Class of malicious url
641119 unique values	benign 66% defacement 15% Other (126631) 19%
br-icloud.com.br	phishing
mp3raid.com/music/krizz_kaliko.html	benign
bopsecrets.org/rexroth/cr/1.htm	benign
http://www.garage-pirenne.be/index.php?option=com_content&view=article&id=70&vsig70_0=15	defacement
http://adventure-nicaragua.net/index.php?option=com_mailto&tmpl=component&link=aHR0cDovL2FkdmVudHVyZ...	defacement

Figure 1 - Extact of the Dataset

Kubeflow Pipeline

To train the model, we used a Kubeflow Pipeline composed of 15 components, each contributing to the training process and tuning of the hyperparameters for two selected models: 'Logistic Regression' and 'Decision Tree'. This pipeline aims to deliver the best-trained model.

In our approach to phishing URLs detection, we have chosen to adopt Logistics Regression and Decision Tree models because of the advantages they offer.

First, the choice of these models, which turn out to be simpler, is in line with our goal of obtaining a high interpretability compared to other models, such as neural networks. Given the critical nature of phishing detection, having a clear understanding of how our model makes decisions is essential for effective threat analysis.

Moreover, the simplicity of these models allows us to avoid the risk of overfitting, ensuring that our model generalizes well and provides solid performance. The faster training times associated with Logistic Regression and Decision Tree accelerate the model development process, allowing us to iterate and adapt quickly. In the constantly evolving field of cybersecurity, speed is a key factor in staying ahead of emerging threats. By using these models, it is possible to achieve a balance between interpretability, avoid overfitting, fast training, and reliable detection of phishing URLs.

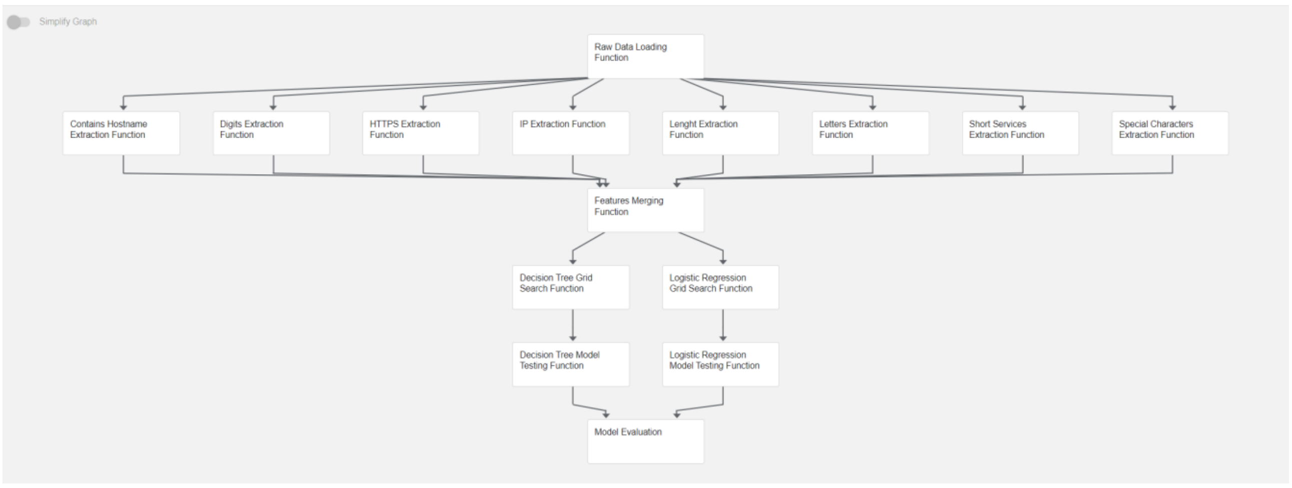

Simplify Graph
Raw Data Loading Function
Contains Hostname Extraction Function
Digits Extraction Function
HTTPS Extraction Function
IP Extraction Function
Lenght Extraction Function
Letters Extraction Function
Short Services Extraction Function
Special Characters Extraction Function
Features Merging Function
Decision Tree Grid Search Function
Logistic Regression Grid Search Function
Decision Tree Model Testing Function
Logistic Regression Model Testing Function
Model Evaluation

Figure 2 - Kubeflow Pipeline

Data Preprocessing and Features Extraction

Raw Data Loading

In this component, the dataset is loaded from a local CSV file into a pandas DataFrame object. Next, the string 'www.' is removed from each URL, and the string representing the URL category is replaced with a number in a range from 0 to 3, where 0 represents 'benign', 1 represents 'defacement', 2 represents 'phishing', and 3 represents 'malware'. After these operations, the newly modified dataset is saved in a file that is passed to the other components for feature extraction from the data.

```
def _load_data(args):
    data = pd.read_csv('malicious_phish_copy.csv')

    data['url'] = data['url'].replace('www.', '', regex=True)

    rem = {"Category": {"benign": 0, "defacement": 1, "phishing":2, "malware":3}}
    data['Category'] = data['type']
    data = data.replace(rem)

    print(data)

    data.drop(['type'], axis=1, inplace=True)

    #creates a csv structure with the data
    data.to_csv(args.data, index=False)
```

Snippet 1 - Load Raw Data Function

Count Digits

In this component, all the digits in the URL are counted, and the resulting data is stored in a new label of the DataFrame. The DataFrame is then saved in a CSV file, making it ready to be merged with the other extracted features.

```
def _url_count_digit_ext(args):

    print("reading data")
    # Open and reads file "data"
    data = pd.read_csv(args.DataIn)

    print("data read")
    def digit_count(url):
        digits = 0
        for i in url:
            if i.isnumeric():
                digits = digits + 1
        return digits

    print("counting digits")
    data['digits'] = data['url'].apply(lambda x: digit_count(x))

    print("saving data")
    # Saves the json object into a file
    data.to_csv(args.DataOut)
```

Snippet 2 - Count Digit Function

Count Letters

In this component, all the letters in the URL are counted, and the resulting data is stored in a new label of the DataFrame. The DataFrame is then saved in a CSV file, making it ready to be merged with the other extracted features.

```
def _url_count_letters_ext(args):

    print("reading data")
    # Open and reads file "data"
    data = pd.read_csv(args.DataIn)

    def letters_count(url):
        letters = 0
        for i in url:
            if i.isalpha():
                letters = letters + 1
        return letters

    print("counting letters")
    data['letters'] = data['url'].apply(lambda x: letters_count(x))

    print("saving data")
    data.to_csv(args.DataOut)
```

Snippet 3 - Count Letters Function

Has IP Address

This function reads data from a CSV file, checks if an IP address is present in URLs, and adds a 'Has_IP_Address' column to the dataframe with binary values (1 if an IP address is found, 0 if not). Finally, it saves the modified DataFrame to a new CSV file, making it ready to be merged with the other extracted features.

```
def _url_hasippaddr_ext(args):

    print("reading data")
    # Open and reads file "data"
    data = pd.read_csv(args.DataIn)

    def Has_IP_Address(url):
        match = re.search(
            '(([01]?\\d\\d?|2[0-4]\\d|25[0-5])\\.([01]?\\d\\d?|2[0-4]\\d|25[0-
5])\\.([01]?\\d\\d?|2[0-4]\\d|25[0-5])\\.'
            '([01]?\\d\\d?|2[0-4]\\d|25[0-5])\\/)|'  # IPv4
            '(([01]?\\d\\d?|2[0-4]\\d|25[0-5])\\.([01]?\\d\\d?|2[0-4]\\d|25[0-
5])\\.([01]?\\d\\d?|2[0-4]\\d|25[0-5])\\.'
            '([01]?\\d\\d?|2[0-4]\\d|25[0-5])\\/)|'  # IPv4 with port
            '((0x[0-9a-fA-F]{1,2})\\.(0x[0-9a-fA-F]{1,2})\\.(0x[0-9a-fA-F]{1,2})\\.(0x[0-
9a-fA-F]{1,2})\\/)' # IPv4 in hexadecimal
            '(?:[a-fA-F0-9]{1,4}:){7}[a-fA-F0-9]{1,4}|'
            '([0-9]+(?:\.[0-9]+){3}:[0-9]+)|'
            '((?:(?:\d|[01]?\d\d|2[0-4]\d|25[0-5])\.){3}(?:25[0-5]|2[0-
4]\d|[01]?\d\d|\d)(?:\/\d{1,2})?)', url)  # Ipv6
        if match:
            return 1
        else:
            return 0

    print("checking if url has ip address")
    data['Has_IP_Address'] = data['url'].apply(lambda x: Has_IP_Address(x))

    print("saving data")
    data.to_csv(args.DataOut)
```

Snippet 4 - Has IP Addres Function

Contains Hostname

This function reads data from a CSV file, checks each URL to see if its hostname is present within the URL itself, and adds a new column to the DataFrame, that contains binary values (0 if the URL contains its hostname, 1 if it doesn't). Finally, it saves the modified DataFrame to a new CSV file, making it ready to be merged with the other extracted features.

```
def _url_hostname_ext(args):

    print("reading data")
    # Open and reads file "data"
    data = pd.read_csv(args.DataIn)

    def has_hostname(url):
        hostname = urlparse(url).hostname
        hostname = str(hostname)
        match = re.search(hostname, url)
        print(hostname)
        if match:
            return 1
        else:
            return 0

    print("checking host name")
    data['has_hostname'] = data['url'].apply(lambda x: has_hostname(x))

    print("saving data")
    # Saves the json object into a file
    data.to_csv(args.DataOut)
```

Snippet 5 - Has Hostname Function

HTTPS

This function reads data from a CSV file and checks each URL to determine if it uses HTTPS (secure HTTP). It adds a 'https' column to the DataFrame, assigning binary values (1 if the URL uses HTTPS, 0 if it doesn't), and then saves the modified DataFrame to a new CSV file, making it ready to be merged with the other extracted features.

```
def _url_https_ext(args):

    print("reading data")
    # Open and reads file "data"
    data = pd.read_csv(args.DataIn)

    def httpSecure(url):
        htp = urlparse(url).scheme
        match = str(htp)
        if match=='https':
            return 1
        else:
            return 0

    print("checking if url has https")
    data['https'] = data['url'].apply(lambda x: httpSecure(x))


    print("saving data")
    data.to_csv(args.DataOut)
```

Snippet 6 - HTTPS Function

URL Length

This function reads data from a CSV file, calculates the length of each URL, adds a new column to the DataFrame, and then saves the modified DataFrame to a new CSV file, making it ready to be merged with the other extracted features.

```
def _url_len_ext(args):

  print("reading data")
  # Open and reads file "data"
  data = pd.read_csv(args.DataIn)

  print("counting url length")
  data['url_len'] = data['url'].apply(lambda x: len(x))
  print("saving data")
  data.to_csv(args.DataOut)
```

Snippet 7 - URL Lenght Function

Shortening Services

This function reads data from a CSV file, checks if URLs contain popular URL shortening services, assigns 1 if a shortening service is detected, 0 if not, and then saves the modified DataFrame to a new CSV file, making it ready to be merged with the other extracted features.

```
def _url_shortserv_ext(args):

  print("reading data")
  # Open and reads file "data"
  data = pd.read_csv(args.DataIn)

  def Shortining_Service(url):
    match = re.search('bit\.ly|goo\.gl|shorte\.st|go2l\.ink|x\.co|ow\.ly|t\.co|tinyurl|tr\.im|is\.gd|cli\.gs|'
            'yfrog\.com|migre\.me|ff\.im|tiny\.cc|url4\.eu|twit\.ac|su\.pr|twurl\.nl|snipurl\.com|'
            'short\.to|BudURL\.com|ping\.fm|post\.ly|Just\.as|bkite\.com|snipr\.com|fic\.kr|loopt\.us|'
            'doiop\.com|short\.ie|kl\.am|wp\.me|rubyurl\.com|om\.ly|to\.ly|bit\.do|t\.co|lnkd\.in|'
            'db\.tt|qr\.ae|adf\.ly|goo\.gl|bitly\.com|cur\.lv|tinyurl\.com|ow\.ly|bit\.ly|ity\.im|'
            'q\.gs|is\.gd|po\.st|bc\.vc|twitthis\.com|u\.to|j\.mp|buzurl\.com|cutt\.us|u\.bb|yourls\.org|'
            'x\.co|prettylinkpro\.com|scrnch\.me|filoops\.info|vzturl\.com|qr\.net|1url\.com|tweez\.me|v\.gd|'
            'tr\.im|link\.zip\.net',
            url)
    if match:
      return 1
    else:
      return 0

  print("checking if url has shortening service")
  data['Shortining_Service'] = data['url'].apply(lambda x: Shortining_Service(x))

  print("saving data")
  data.to_csv(args.DataOut)
```

Snippet 8 - Shortening Services Function

Special Characters

This function reads data from a CSV file, checks if contain special characters, counts each character in an URL, and then saves them in additional columns representing the counts of each special character to a new CSV file, making it ready to be merged with the other extracted features.

```
def _url_spechar_ext(args):

    print("reading data")
    # Open and reads file "data"
    data = pd.read_csv(args.DataIn)

    feature = ['@','?','-','=','.','#','%','+','$','!','*',',','//']

    print("checking special characters")
    for i in feature:
        data[i] = data['url'].apply(lambda x: x.count(i))

    print("saving data")
    data.to_csv(args.DataOut)
```

Snippet 9 - Special Characters Function

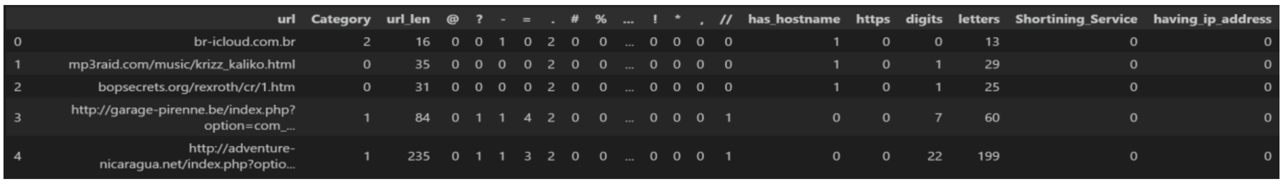
Features Merging

The function initiates its process by loading data from multiple CSV files. As part of the data consolidation step, it identifies and eliminates duplicate columns to maintain data integrity. Moreover, it filters out any columns that are considered irrelevant or unnecessary for subsequent machine learning training.

Following data cleaning, the function proceeds to extract the features (independent variables) and the target variable (dependent variable) essential for training machine learning models. This demarcation distinguishes between what the model should predict (target) and what it should employ for making predictions (features).

To assess model performance, the function partitions the combined dataset into two subsets: a training set (80%) and a validation set (20%).

Finally, the training and testing datasets are saved into a data dictionary, which is then transformed into a JSON object.



	url	Category	url_len	@	?	-	=	.	#	%	...	!	*	,	//	has_hostname	https	digits	letters	Shortining_Service	having_ip_address
0	br-icloud.com.br	2	16	0	0	1	0	2	0	0	...	0	0	0	0	1	0	0	13	0	0
1	mp3raid.com/music/krizz_kaliko.html	0	35	0	0	0	0	2	0	0	...	0	0	0	0	1	0	1	29	0	0
2	bopsecrets.org/rexroth/cr/1.htm	0	31	0	0	0	0	2	0	0	...	0	0	0	0	1	0	1	25	0	0
3	http://garage-pirenne.be/index.php?option=com_...	1	84	0	1	1	4	2	0	0	...	0	0	0	1	0	0	7	60	0	0
4	http://adventure-nicaragua.net/index.php?optio...	1	235	0	1	1	3	2	0	0	...	0	0	0	1	0	0	22	199	0	0

Table 1 - Final Pandas DataFrame after Features Extraction

Model Training

Decision Tree

The Decision Tree component tunes hyperparameters for the Decision Tree model using a grid search approach.

Firstly, the function defines a parameter grid that includes various combinations of hyperparameters to be evaluated. In this case, the hyperparameters are:

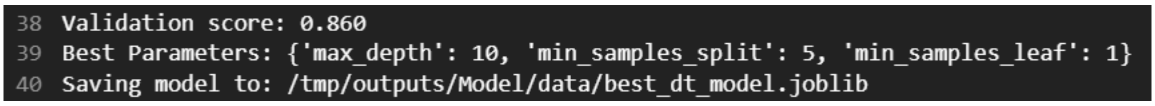
- **max_depth**: Maximum depth of the decision tree.
- **min_samples_split**: The minimum number of samples required to split an internal node.
- **min_samples_leaf**: The minimum number of samples required to be in a leaf node.

Then the function uses nested loops to iterate over all combinations of the hyperparameters, trying every combination of the specified values.

After fitting the model, it evaluates the model's performance on the validation data by computing accuracy.

During the process, the function tracks of the best validation score obtained and compares the obtained validation score with the best obtained score. If the current validation score is higher than the best score, it updates ‘best_score’ to the new score and stores the hyperparameters used for this model in ‘best_params’.

Once all combinations have been tried, the function updates the Decision Tree model with these best hyperparameters and saves this best model to a file using the joblib library.



```
Validation score: 0.860
Best Parameters: {'max_depth': 10, 'min_samples_split': 5, 'min_samples_leaf': 1}
Saving model to: /tmp/outputs/Model/data/best_dt_model.joblib
```

Figure 3 - Decision Tree Grid Search Results

Logistic Regression

The Logistic Regression component tunes hyperparameters for the Logistic Regression model using a grid search approach.

Firstly, the function defines a parameter grid, which contains various combinations of hyperparameters to be evaluated for the Logistic Regression model. The hyperparameters being tuned are:

- **C**: Regularization strength, which controls the inverse of the regularization term (smaller values mean stronger regularization).
- **penalty**: The type of regularization term to apply ('l1' for L1 regularization, 'l2' for L2 regularization).

The function uses nested loops and iterates over all combinations of these hyperparameters as specified.

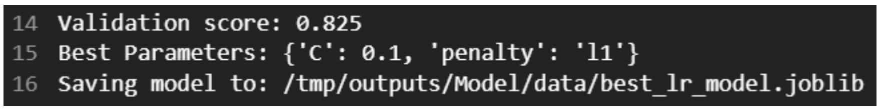
After setting the hyperparameters for the Logistic Regression model, it proceeds to train the model on the training data. The Scikit-Learn pipeline fit both the Scaler and the Logistic Regression Model.

The model's performance is evaluated on the validation data using the score method. The validation score is computed to measure the accuracy of the model's predictions.

During the hyperparameter tuning process, the function maintains a record of the best validation score. The function keeps track of the hyperparameters that resulted in the best score.

After each model is trained and validated with a specific combination of hyperparameters, the function compares the current validation score with the best score. If the current score is higher (indicating a better-performing model), it updates ‘best_score’ to the new score and stores the hyperparameters used for this model in ‘best_params’.

The Logistic Regression model is updated with the best hyperparameters, the best model is then trained on the entire training set with these hyperparameters. Finally, the best model is saved to a file using the joblib library.



```
Validation score: 0.825
Best Parameters: {'C': 0.1, 'penalty': 'l1'}
Saving model to: /tmp/outputs/Model/data/best_lr_model.joblib
```

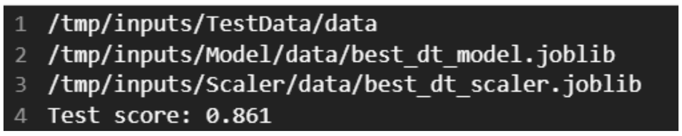
Figure 4 - Logistic Regression Grid Search Results

Model Assessment

Decision Tree

This component is responsible for testing the Decision Tree model using test data and evaluating its performance. The model and scaler are loaded from their respective files using the ‘joblib.load()’ function.

The Scikit-Learn pipeline, consisting of the loaded scaler and the Decision Tree model, is used to make predictions on the test data. The accuracy of the model's predictions is computed using the score method and is saved into a file.



```
/tmp/inputs/TestData/data
/tmp/inputs/Model/data/best_dt_model.joblib
/tmp/inputs/Scaler/data/best_dt_scaler.joblib
Test score: 0.861
```

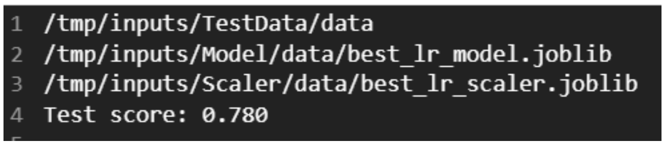
Figure 5 - Decision Tree Testing Results

Logistic Regression

This function is responsible for testing the Logistic Regression model using the test data and evaluating its performance. The model and scaler are loaded from their respective files using the 'joblib.load()' function.

A Scikit-Learn pipeline is created, consisting of the loaded scaler and the Logistic Regression model, and is used to make predictions on the test data.

The accuracy of the model's predictions is computed using the score method and is saved into a file.

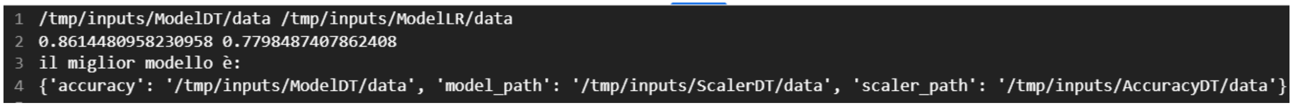


```
/tmp/inputs/TestData/data
/tmp/inputs/Model/data/best_lr_model.joblib
/tmp/inputs/Scaler/data/best_lr_scaler.joblib
Test score: 0.780
```

Figure 6 - Logistic Regression Testing Results

Model Evaluation and Identification of The Best Model

This component determines the best model based on their accuracy scores and prints its details, such as the path and the accuracy of the model and the path of the associated scaler.



```
/tmp/inputs/ModelDT/data /tmp/inputs/ModelLR/data
0.8614480958230958 0.7798487407862408
il miglior modello è:
{'accuracy': '/tmp/inputs/ModelDT/data', 'model_path': '/tmp/inputs/ScalerDT/data', 'scaler_path': '/tmp/inputs/AccuracyDT/data'}
```

Figure 7 - Model Evaluation Results

Pipeline Creation

After the definition of each component, the entire process, from data preprocessing to displaying results, is automated within the pipeline. The output of each component serves as the input for the following one, ensuring a seamless flow of data. Finally, the pipeline is compiled into a YAML file, making it ready for deployment in a Kubeflow environment.

```
import kfp
from kfp import dsl


@dsl.pipeline(name='First Pipeline', description='Applies Decision Tree and Logistic Regression for a '
                                                  'classification problem.')
def first_pipeline():
    # Loads the yaml manifest for each component
    load_raw_data =
kfp.components.load_component_from_file('./data_preprocessing/load_raw_dataset/load_raw_dataset.yaml')
    url_count_digits_ext =
kfp.components.load_component_from_file('./data_preprocessing/url_count_digit_ext/url_cou
nt_digit_ext.yaml')
    url_count_letters_ext =
kfp.components.load_component_from_file('./data_preprocessing/url_count_letters_ext/url_c
ount_letters_ext.yaml')
    url_hasipaddr_ext =
kfp.components.load_component_from_file('./data_preprocessing/url_hasipaddr_ext/url_hasip
addr_ext.yaml')
    url_hostname_ext =
kfp.components.load_component_from_file('./data_preprocessing/url_hostname_ext/url_hostna
me_ext.yaml')
    url_https_ext =
kfp.components.load_component_from_file('./data_preprocessing/url_https_ext/url_https_ext
.yaml')
    url_len_ext =
kfp.components.load_component_from_file('./data_preprocessing/url_len_ext/url_len_ext.yam
l')
    url_shortserv_ext =
kfp.components.load_component_from_file('./data_preprocessing/url_shortserv_ext/url_short
serv_ext.yaml')
    url_spechar_ext =
kfp.components.load_component_from_file('./data_preprocessing/url_spechar_ext/url_spechar
_ext.yaml')
    load = kfp.components.load_component_from_file('load_data/load_data.yaml')
    decision_tree =
kfp.components.load_component_from_file('decision_tree/decision_tree.yaml')
    logistic_regression =
kfp.components.load_component_from_file('logistic_regression/logistic_regression.yaml')
    test_decision_tree =
kfp.components.load_component_from_file('test_decision_tree/test_decision_tree.yaml')
    test_logistic_regression =
kfp.components.load_component_from_file('test_log_regr/test_log_regr.yaml')
    show_result = kfp.components.load_component_from_file('show_result/show_result.yaml')

    # Run load_data task
    load_raw_data_task = load_raw_data()
    url_count_digits_ext_task = url_count_digits_ext(load_raw_data_task.output)
    url_count_letters_ext_task = url_count_letters_ext(load_raw_data_task.output)
    url_hasipaddr_ext_task = url_hasipaddr_ext(load_raw_data_task.output)
    url_hostname_ext_task = url_hostname_ext(load_raw_data_task.output)
    url_https_ext_task = url_https_ext(load_raw_data_task.output)
    url_len_ext_task = url_len_ext(load_raw_data_task.output)
    url_shortserv_ext_task = url_shortserv_ext(load_raw_data_task.output)
    url_spechar_ext_task = url_spechar_ext(load_raw_data_task.output)

    load_task = load(url_count_digits_ext_task.output,url_spechar_ext_task.output,

url_hasipaddr_ext_task.output,url_shortserv_ext_task.output,url_count_letters_ext_task.ou
tput,

url_hostname_ext_task.output,url_https_ext_task.output,url_len_ext_task.output)
    decision_tree_task = decision_tree(load_task.output)
    logistic_regression_task = logistic_regression(load_task.output)
    test_decision_tree_task = test_decision_tree(decision_tree_task.outputs['TestData'],
decision_tree_task.outputs['model'],
                                                 decision_tree_task.outputs['scaler'])
    test_logistic_regression_task =
test_logistic_regression(logistic_regression_task.outputs['TestData'],logistic_regression
_task.outputs['model'],

logistic_regression_task.outputs['scaler'])
    show_result_task = show_result(test_decision_tree_task.outputs['ModelDT'],
                                   test_decision_tree_task.outputs['ScalerDT'],
                                   test_decision_tree_task.outputs['AccuracyDT'],
                                   test_logistic_regression_task.outputs['ModelLR'],
                                   test_logistic_regression_task.outputs['ScalerLR'],
                                   test_logistic_regression_task.outputs['AccuracyLR'])
```

Snippet 10 – url_prediction_pipeline.py

Once the YAML file associated with the Kubeflow pipeline is created, it is uploaded to Kubeflow and its execution starts. The pipeline conducts the full process of tuning and training the selected models. Once this process finishes and the best model is identified, it must be downloaded manually. This manual intervention is necessary because, in our setup, the Kubeflow pipeline is not directly connected to the application deployment, due to the absence of tools that would facilitate this linkage.

Kubernetes Cluster

The application should serve 50,000 customers, with a peak concurrency of 5,000 requests; for this reason, the cluster has been designed to meet these requirements. However, the setup of a Kubernetes cluster needs to be flexible and able to change in response to updates in the Service Level Agreement (SLA) or requirements. Changes to the SLA might mean adjustments are needed in areas like how available the system is, how well it performs, and how secure it is. So, it's important to look over the current setup and make any needed changes, which could include upgrading hardware, changing security policies, or improving performance.

Cluster Configuration and Port Mapping

We designed a Kubernetes cluster with 1 Control Plane node and 3 Worker nodes, in this way we assure:

1. **Scalability and Load Handling**: The worker nodes are responsible for running the actual applications and services. Having 3 worker nodes provides the capacity to handle a large number of concurrent requests. The load is distributed across multiple nodes, enhancing the ability to scale the application and manage peak loads effectively.
2. **High Availability for Application Workloads**: If one worker node fails or becomes overloaded, the other nodes can continue to handle requests, minimizing downtime and ensuring a reliable user experience.
3. **Resource Allocation and Utilization**: The worker nodes provide the necessary compute, memory, and storage resources for application pods. By having multiple nodes, we can allocate these resources more efficiently, ensuring that the application has enough resources to handle peak demand without over-provisioning. At the same time, multiple worker nodes allow for more efficient utilization of resources. Pods are scheduled across these nodes based on resource availability and requirements, leading to optimal utilization of the hardware resources.
4. **Control-Plane Isolation**: A dedicated control-plane node ensures that the critical cluster management functions are isolated from the application workloads. This isolation improves the security and stability of the cluster, as the control-plane node can focus on managing the cluster's state, scheduling, and orchestration tasks.
5. **Fault Tolerance and Reliability**: If a worker node encounters an issue, the other nodes can take over its workload, ensuring continuous service availability. The control-plane node oversees this process, making sure that pods are rescheduled appropriately.
6. **Future Expansion**: As the demand grows, more worker nodes can be added to the cluster to increase its capacity.

The Kubernetes cluster, which includes one control plane node and three worker nodes, is a great example of how careful planning can lead to a highly efficient system in terms of both performance and cost. This balanced and flexible setup ensures efficiency and resilience and is ready to adapt to different workloads and future requirements.

The cluster is designed to handle many users and can deal with many simultaneous requests. This setup ensures the application remains stable and fast, providing a smooth user experience and uninterrupted service.

The use of three worker nodes is especially suitable for training simpler machine learning models like decision trees and logistic regression. However, the setup is flexible. If more complex models like deep neural networks are introduced, the cluster can be upgraded, for example by adding GPUs, through vertical scaling, to handle the increased computational complexity effectively.

For deploying the application, an important feature is the horizontal autoscaling of pods. This allows the system to automatically adjust the number of pods based on the actual workload, optimizing resource use, and improving the management of traffic peaks. This autoscaling ensures that the application remains responsive and high-performing even under variable workloads, significantly contributing to the overall efficiency of the system.

In conclusion, the cluster is set up to meet both current needs and more demanding future scenarios. This balanced approach to reliability, flexibility, and efficient resource use is key to maintaining a responsive, scalable system, ready to dynamically meet operational and technological challenges.

```yaml
kind: Cluster
apiVersion: kind.x-k8s.io/v1alpha4
nodes:
- role: control-plane
  extraPortMappings:
  - containerPort: 30020
    hostPort: 30010
- role: worker
- role: worker
- role: worker
```

Snippet 11 - multinode-config-with-port-mapping.yaml

Extra port mapping in Kubernetes allows us to expose ports of the containers running inside the cluster to the external network. It has been used in our configuration as we used Kind for local development. In this case, port 30020 of the container maps port 30010 of the local host. With this configuration traffic sent to port 30010 on the host will be forwarded to port 30020 on the container.

Dashboard Admin User

The following snippet creates a ServiceAccount named ‘admin-user’ in the kubernetes-dashboard namespace. ServiceAccounts are used in Kubernetes to provide an identity for processes that run in a Pod.

```yaml
apiVersion: v1
kind: ServiceAccount
metadata:
  name: admin-user
  namespace: kubernetes-dashboard
```

Snippet 12 - dashboard-adminuser.yaml

Cluster Role Binding

The following snippet creates a ClusterRoleBinding named ‘admin-user’. It assigns the *cluster-admin* role to the ‘admin-user’ ServiceAccount in the kubernetes-dashboard namespace.

The *cluster-admin* role is a predefined ClusterRole in Kubernetes that grants full administrative access to the entire cluster. This means the ‘admin-user’ ServiceAccount will have the permissions to perform any action on any resource within the cluster.

```yaml
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRoleBinding
metadata:
  name: admin-user
roleRef:
  apiGroup: rbac.authorization.k8s.io
  kind: ClusterRole
  name: cluster-admin
subjects:
- kind: ServiceAccount
  name: admin-user
  namespace: kubernetes-dashboard
```

Snippet 13 - cluster_rolebinding.yaml

Deploy URL Prediction App

Deployment

Deployment kind manages the deployment and scaling of a set of Pods. Ensures that the specified number of Pods are running and updates them, as necessary. Each pod is configured to request certain CPU and memory resources, based on the image dimension, and has defined limits.

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: url-detection
  labels:
    app: url-detection-app
  namespace: default
spec:
  replicas: 5
  selector:
    matchLabels:
      app: url-detection-app
  template:
    metadata:
      labels:
        app: url-detection-app
    spec:
      containers:
      - name: url-detection
        image: luciasenatore/url_detection
        resources:
          requests:
            cpu: "0.5"
            memory: "1.5Gi"
          limits:
            memory: "1.5Gi"
```

Snippet 14 - url_app_kubernetes.yaml

To manage container resources, guidelines have been followed. When creating a pod in Kubernetes, it is possible to specify the amount of resources a container needs. Kubernetes allows us to set resource requests and resource limits. In our case, we manage CPU and memory:

- For CPU, best practices suggest that we should set a request, but it is not necessary to set limits. By doing this, our pods have a guaranteed amount of CPU, but if more is needed, excess CPU is available and not wasted, as it is allocated to whoever needs it.
- For memory, best practices recommend setting both requests and limits to the same value.

This difference between CPU and memory management is because memory is non-compressible. Once memory is allocated, it cannot be released without terminating the process.

Service

Service kind provides a stable endpoint for accessing the Pods in a Deployment. Our service is named 'url-detection' and it is of type NodePort. It exposes the application running in the pods on cluster nodes' port 30020, which forwards to port 8501 on the pods. This setup is crucial for external access to the application.

```yaml
apiVersion: v1
# Indicates this as a service
kind: Service
metadata:
  # Service name
  name: url-detection
  namespace: default
spec:
  type: NodePort
  selector:
    # Selector for Pods
    app: url-detection-app
  ports:
  # Port Map
  - port: 8501
    targetPort: 8501
    nodePort: 30020
```

Snippet 15 - url_app_kubernetes.yaml

HorizontalPodAutoscaler

HorizontalPodAutoscaler kind scales the number of Pods in a Deployment based on observed CPU utilization or another selected metric. Our autoscaler is named 'url-detection-autoscaler' and it adjusts the number of replicas within the range of 2 to 10, based on the CPU utilization reaching 70%.

It is essential for applications needing to handle variable loads efficiently, ensuring that the application scales as required while managing resource use effectively.

```yaml
apiVersion: autoscaling/v2
kind: HorizontalPodAutoscaler
metadata:
  name: url-detection-autoscaler
  namespace: default
spec:
  scaleTargetRef:
    apiVersion: apps/v1
    kind: Deployment
    name: url-detection
  minReplicas: 2    # Numero minimo di repliche
  maxReplicas: 10   # Numero massimo di repliche
  metrics:
  - type: Resource
    resource:
      name: cpu
      target:
        type: Utilization
        averageUtilization: 70 # Percentuale di utilizzo della CPU per lo scaling
```

Snippet 16 - url_app_kubernetes.yaml

Using Grafana for Monitoring Kubernetes Cluster

In managing a Kubernetes cluster, the Grafana dashboard is very important for monitoring resources. This platform allows real-time observation of parameters such as CPU usage, memory, network status, application performance, and especially the effectiveness of autoscaling.

Grafana enables understanding of how the cluster dynamically adapts to changing workloads, automatically scaling resources as needed. This continuous monitoring is essential to observe the efficiency of autoscaling and to implement targeted optimizations, ensuring the best possible performance.

Customization of the Grafana dashboard provides the ability to view and analyse the cluster's behaviour in relation to initial hypotheses, comparing expected data with actual observations. This process aids in identifying discrepancies and allows for the rapid implementation of necessary changes.

A big part of Grafana's usefulness is how it works together with Prometheus, a system that monitors and alerts in Kubernetes environments. Prometheus collects and saves data all the time, and Grafana allows us to view this data on the dashboard.

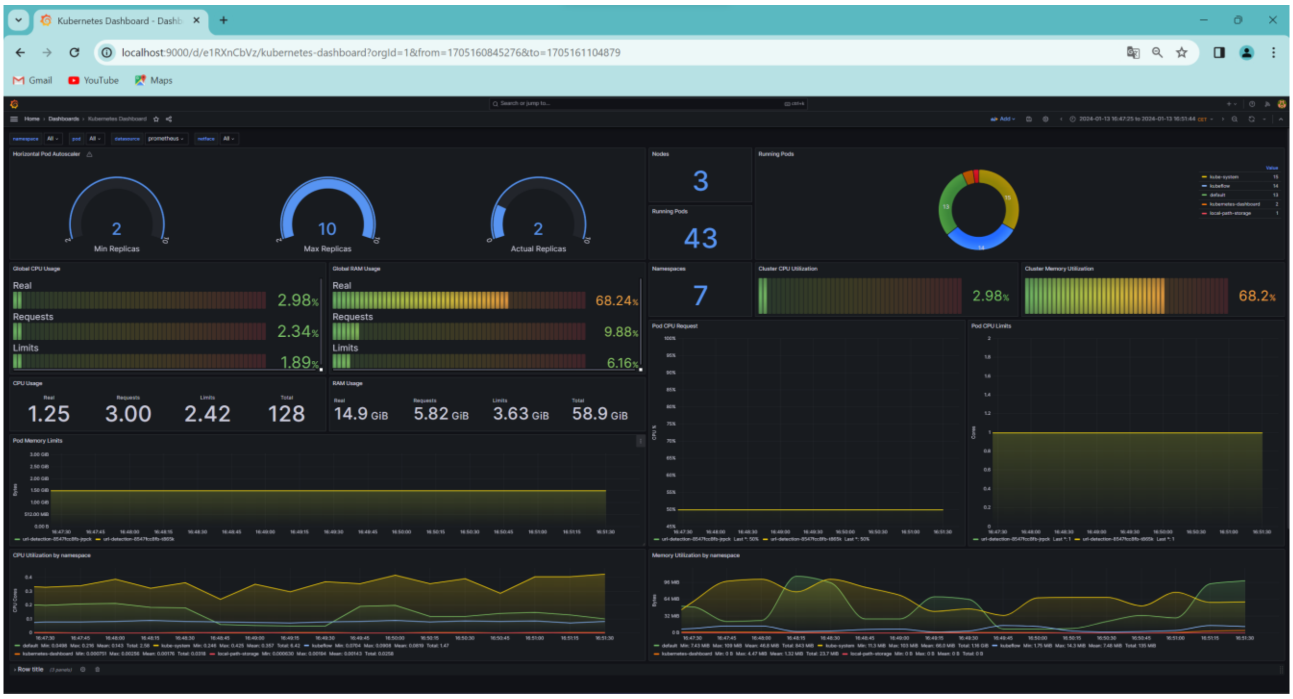


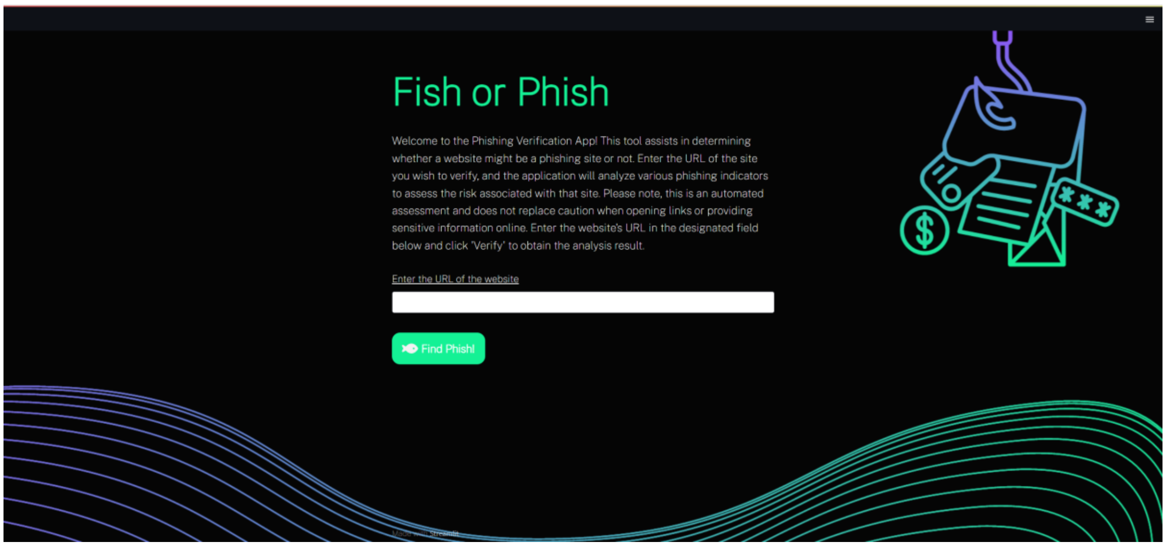
Figure 8 - Grafana Dashboard

App Description

The final application is built with Streamlit. When a user inputs a URL, the app performs various checks on the URL's characteristics commonly associated with phishing sites.

Key functionalities:

- **Feature Extraction**: The app extracts features from the URL, such as the number of digits, letters, whether it contains an IP address, its length, and if it uses a URL shortening service.
- **Machine Learning Model**: the pre-trained machine learning model, produced by the Kubeflow Pipeline, is loaded to classify the URL.
- **User Interface**: Users input a URL, after the prediction, the results are displayed on the webpage, indicating whether the URL is likely legitimate or a potential phishing threat.



Fish or Phish

Welcome to the Phishing Verification App! This tool assists in determining whether a website might be a phishing site or not. Enter the URL of the site you wish to verify, and the application will analyze various phishing indicators to assess the risk associated with that site. Please note, this is an automated assessment and does not replace caution when opening links or providing sensitive information online. Enter the website's URL in the designated field below and click 'Verify' to obtain the analysis result.

Figure 9 - Web Application UI

Conclusions

Training a machine learning model with Kubeflow and deploying an application based on this model in a Kubernetes cluster is a significant advancement towards a technology future that is both efficient and eco-friendly. Kubeflow stands out as a robust and adaptable environment for model training, accelerating the learning process while enhancing energy efficiency. In an era where resource optimization is crucial, Kubeflow represents a crucial step towards sustainable and responsible technology practices.

Adding Docker to this mix brings even more benefits. Its lightweight and portable containers simplify the deployment process, making applications easily accessible anywhere. This method reduces the time required for development and market introduction, ensuring that the application is reliable and consistent across different platforms.

Kubernetes is at the heart of this system, with its advanced orchestration system perfectly managing the application's distribution and scalability. Kubernetes' efficient utilization of available resources ensures optimal performance without wastage, aligning with the global shift towards efficiency and sustainability.

In summary, using Kubeflow, Docker, and Kubernetes together is not just a sign of technological progress, but it also represents a real commitment to more sustainable and environmentally friendly development, meeting the needs and expectations of our time.

Protect the App from Cyber-Attacks

Even though Kubernetes has a strong architecture, it's not completely safe from security threats. The complex nature of Kubernetes and the fast rate at which things are deployed can sometimes lead to small mistakes and security weaknesses. Since Kubernetes is open source, it's always being updated with new features and improvements, which can sometimes bring new security risks.

It's important to focus on security at every step of using Kubernetes. Implementing strong security practices and regularly checking for security risks are key to reducing threats. Moreover, setting up a Kubernetes cluster in the right way can make a big difference in making it safer. A well-planned setup not only makes things run better and more efficiently, but it also helps to close any gaps that bad actors could use to get in, making Kubernetes environment more secure and resilient.

Possible Threats

Deploying a web application involves navigating various difficulties, including substantial financial costs, complex system architectures, and a notable scarcity of qualified staff. However, the most crucial aspect to address is the security of the application. The foremost challenge in this context is safeguarding against vulnerabilities at the application layer, which are commonly targeted in cyber attacks to breach systems. Additionally, there is the looming threat of DDoS (Distributed Denial of Service) attacks, which aim to disrupt web application accessibility, often leading to severe operational disruptions and extensive harm. These challenges underscore the urgent need to implement robust security strategies in both the deployment and ongoing management of web applications.

NGINX Solutions

NGINX App Protect is a comprehensive web application firewall (WAF) and Layer 7 denial-of-service (DoS) defence solution designed for modern applications and APIs. It is a lightweight security solution that integrates into DevOps environments, providing robust protection without affecting performance.

NGINX App Protect offers protection against a range of threats, including DDoS attacks, SQL injection, cross-site scripting (XSS), and other common web application attacks.

- **Protection Against Exploits**: NGINX App Protect provides advanced WAF capabilities that exceed the basic OWASP Top 10 protection. It boasts over 7,500 advanced signatures, bot signatures, and threat campaign protection. This allows it to protect against a broad spectrum of exploits, including SQL injection, cross-site scripting (XSS), and other common web application attacks.
- **DDoS Mitigation**: For DDoS protection, NGINX App Protect DoS offers behavioural protection against DoS attacks for web applications. It's capable of mitigating sophisticated Layer 7 attacks. This module scales with applications and provides continuous performance and protection, ensuring that services remain unaffected by DDoS attempts.