Aprendizagem Aplicada à Segurança

Kubernetes Attacks Detection MLOps Approach

Relatório

Mestrado em Engenharia Informática

Pedro Sobral, 98491

Conteúdo

1 Introdução

Este trabalho é feito no âmbito da Unidade Curricular de Aprendizagem Aplicada à Segurança, e tem por objetivo a elaboração de um sistema de deteção de ataques num cluster de Kubernetes, contando com uma abordagem de MLOps, para um mais rápido e eficaz desenvolvimento de todo o processo de Machine Learning, tanto na criação de features e dos modelos, bem como no deployment de modelos para que possam ser testados num ambiente o mais próximo da realidade possivel.

Os dados utilizados neste projecto, foram retirados das seguintes fontes [1] [2], sendo os mesmos inicialmente coletados atraves do NetFlow de um cluster de Kubernetes que estava a correr uma aplicação de microserviços. Foram consultados as seguintes referências: [3][4][5][6]. Tal como numa boa arquitetura de MLOps, uma Feature Store foi implementada pois é uma componente importante, para a criação, manutenção e validação de features através de um robusto sistema de controlo de versões. Dado estes aspetos, foi utilizado a Hopsworks, uma Feature Store open-source.

Primeiramente foi tida uma abordagem "normal" de desenvolvimento de Machine Learning, através de Jupyter Notebooks, onde foram feitas analises referentes aos dados, e a que abordagens de treino de modelos usar, e foi tambem feita uma abordagem com um AutoEncoder.

A implementação da pipeline de ML deu-se sob o Kubeflow, que é uma plataforma end-to-end de orquestração direcionada para Machine Learning, esta plataforma corre sob um cluster de Kubernetes.

Seguindo uma linha de abordagem de MLOps, a parte de validação/monitorização é fulcral daí ter sido implementado uma prova de conceito capaz de fazer esse tipo de analise.

Os resultados foram satisfatórios, sendo sido possivel desenvolver um sistema de MLOps completo, no contexto da deteção de ataques direcionados a um cluster de Kubernetes.

2 Objetivos

Este projeto tem como objetivos:

- Desenvolvimento de um sistema de Machine Learning de deteção de ataques relativo à cibersegurança.
- Desenvolvimento de uma pipeline de ML integrada numa plataforma de MLOps.
- Utilização de uma Feature Store para lidar com o pre-processamento dos dados e das várias versões que este pre-processamento pode ter.
- Mecanismo de deploy automatico de modelos.

- Validação das previsões do modelo selecionado.
- Realização uma prova de conceito que mostre que a deteção de ataques por parte do sistema é rubosto e que permite uma rapida adaptabilidade a mudanças na distribuição dos dados.

3 Arquitetura

Dado tratar-se de uma implementação de um sitema de MLOps, a arquitetura do sistema desenvolvido pode ser observada na Figura 1.

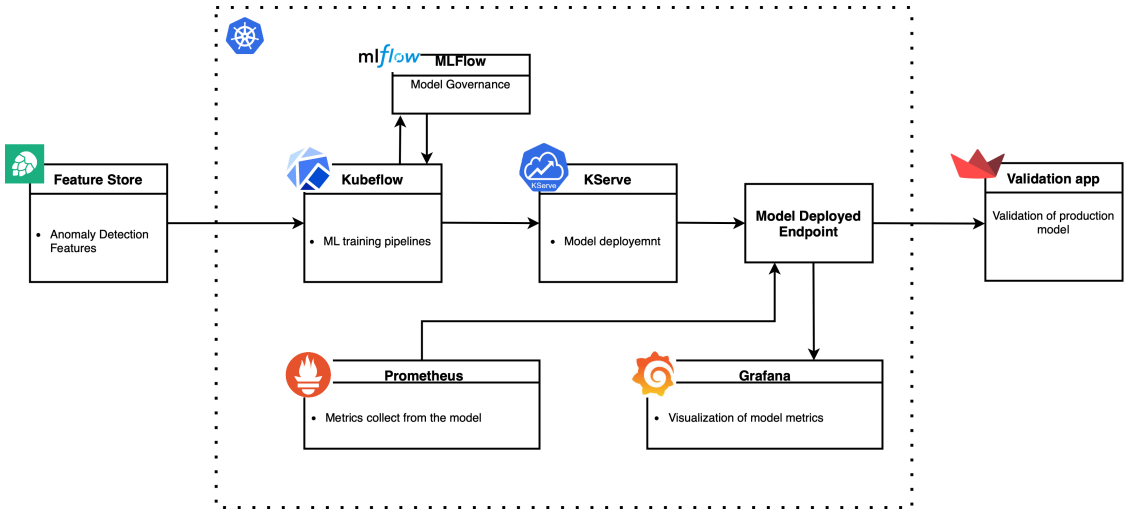

Figura 1: Arquitetura do Projeto

Numa visão mais geral, podemos ver o cluster de Kubernetes, onde se irá passar a maior parte do processamento, e a Feature Store e a aplicação de validação de fora.

Relativamente à Feature Store, no caso a Hopsworks. Este componente é responsavel pela criação de features que serão consumidas pelos modelos de ML. Nesta componente é feito todo o pre-processamento dos dados, desde conversões de tipos, a validações de dados, e pela criação de training dataset. A Feature Store está instanciada na cloud, nos próprios serviços da Hopsworks, estando associada a uma conta gratuita.

Dentro do cluster de Kubernetes, temos então a parte de treino, de deployment e validação dos modelos. Começando pela parte de treino, é utilizado o Kubeflow, que é uma plataforma end-to-end MLOps, aqui é definida uma pipeline de ML, e o resultado da pipeline será um modelo, modelo esse que será deployed através do KServe (ferramenta que o Kubeflow usa para dar deploy dos modelos). Durante a parte de treino para que haja uma melhor "governace" dos dados, foi utilizado o MLFlow, de modo a ter guardados todos os metadados dos modelos para quando for feita uma avaliação dos modelos, este seja de forma rápida e fácil.

Estando o modelo deployed, acessivel através de um endpoint (KServe faz esta parte automaticamente), o mesmo pode ser alvo de pedidos e pode ser testado, com por exemplo dados previamente identificados (com malware ou não) para se testar se o modelo está a ter uma boa performance. Para isso, foi utilizado o Streamlit, que periodicamente faz pedidos aos modelos com dados de input com o target definido, comparando o resultado do modelo com o que de facto o input representa.

Para aceder à infraestrutura é necessário estar dentro da rede do IT, e o Kubeflow pode ser acessido através do seguinte link: http://10.255.32.77

As credenciais de acesso são as default, e podem ser consultadas no seguinte link: https://github.com/kubeflow/manifests

4 Feature Store

Os dados são referentes ao trafego de rede dentro de um cluster de Kubernetes, estando estes identificados como maliciosos ou benignos. Oa dados foram extraídos da seguinte fonte: [1]

Com a utilização da Feature Store todo o processo de pre-procesasmento passa a ser feito na Feature Store, e foi exatamente isso que foi feito. (Figura 2)

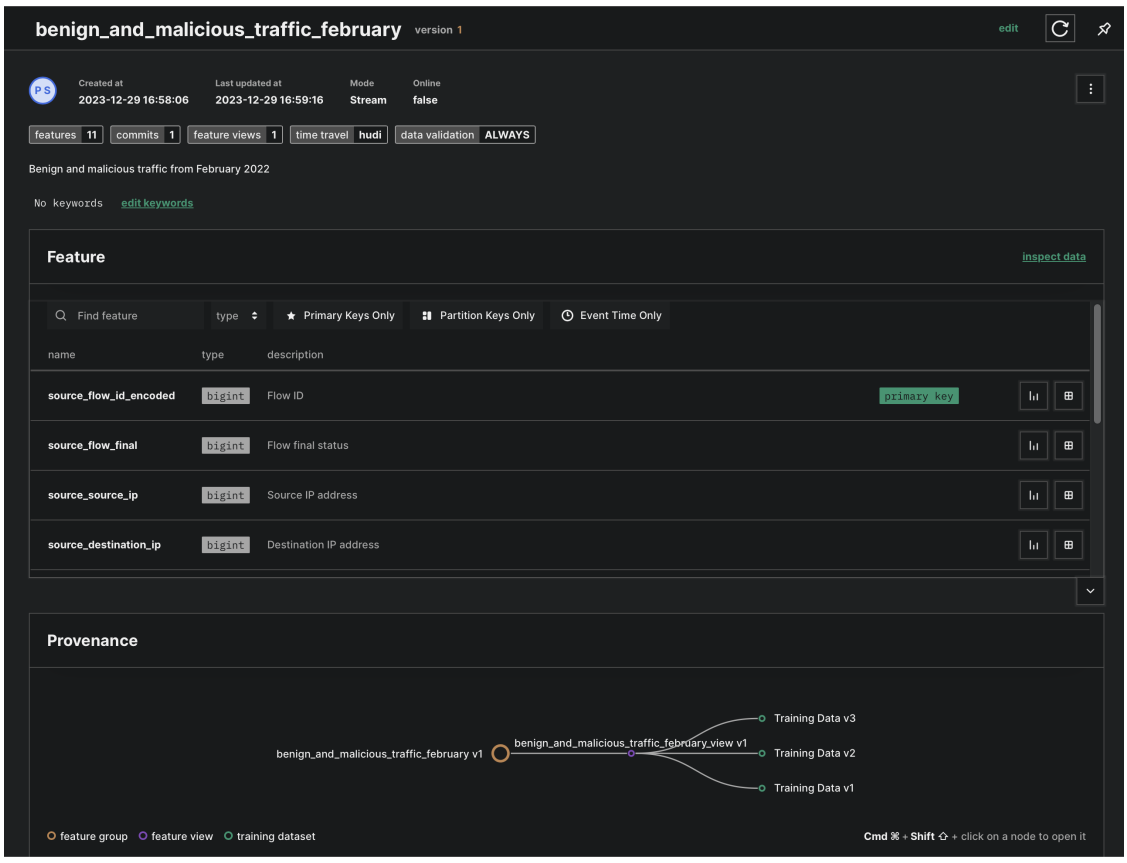

Figura 2: Interface da Feature Store

O primeiro passo a ser feito, foi ler os dados e fazer algum rename de colunas. A partir daí foi feito um preprocessamento, ao nivel do tipo de dados que cada coluna tinha, como por exemplo codificar o network transport de tcp e udp para numérico, converter o ip do fromato xxx.xxx.xxx.xxx para valor numerico com recurso à biblioteca ipaddress, entre outros tipos de pre-processamento.

Antes ainda de os dados irem para a Feature Store, são criadas validations, através da biblioteca great expectations, que serve para validar intervalos nos dados, valores minimos, ou até validações mais complexas.

Estando tudo isto definido, os dados podem ser inseridos na Feature Store, e é adicionada uma pequena descrição a cada features que é adicionada.

Neste ponto é criado o dataset de treino (os dados sao normalizads neste momento), onde é feita uma divisão entre treino, teste, e validação com uma porpoção de 70, 20, e 10 por cento, respetivamente. Os dados de validação serão posteriormente usados no modulo de validação com um modelo já em produção.

5 Machine Learning

Dado que existe a implementação da Feature Store, a parte de pre-processamento já está toda feita, portanto os dados já chegam limpos e cuidados, no entanto é feito ainda visualização de dados. Posto isto, é carregado o dataset de treino e teste, para alimentarem os modelos de treino.

Importante referir que foram feitas duas abordagens, uma sem AutoEncoder (Baseline) e outra com AutoEncoder.

Para o AutoEncoder, a rede criada teve as seguintes dimensões:

- Encoder: 10 e 8
- Latente Space: 6
- Decoder: 8 e 10

Sendo o Latente Space utilizado como os dados que alimentam os modelos.

Os modelos utilizados foram os seguintes: Logistic Regression, SVC, KNN, e Decision Tree.

Em ambas as abordagens, foi feita uma pesquisa pelos melhores hyperparametros através de uma GridSearch. Por cada modelo foram obtidos dois resultados, um com os parametros default e o outro com os melhores parametros encontrados. Para ilustrar os resultados foi implementado a visualização da matrix de confusão, e a métrica selecionada para avaliação dos modelos foi a accuracy.

6 Pipeline de ML

Para o desenvolvimento da pipeline de ML, usei o Kubeflow, um sistema end-to-end que permite a criação de pipelines de ML. Uma pipeline de ML, é uma ferramenta que permite fazer orquestração dos vários passos que um workflow de ML tem. Deste modo cada "passo" é chamado de componente, e um componente é um container. Por cada componente ser um container a reproducibilidade é mais facilmente alcançada, poupando assim recousos computacionais e tempo em estar a criar um container igual a um que já fora anteriormente criado.

Como é possível observar na Figura 3, uma componente é uma função em python, que usa um decorador para indicar uma container image base e que packages são necessários instalar, dentro da função é desenvolvido código de uma forma natural, no caso da Figura 3 a criação de uma experencia no MLFlow, que é uma ferramenta que permite ter um maior controlo e "governance" sobre os metadados e os modelos gerados.

```python
@component(
    packages_to_install=['mlflow'],
    base_image='python:3.8',
)
def create_mlflow_experiment(mlflow_user: str, mlflow_password: str, mlflow_tracking_uri: str) -> str:
    import mlflow
    import os
    from datetime import datetime

    os.environ['MLFLOW_TRACKING_USERNAME'] = mlflow_user
    os.environ['MLFLOW_TRACKING_PASSWORD'] = mlflow_password
    mlflow.set_tracking_uri(mlflow_tracking_uri)

    client = mlflow.tracking.MlflowClient()
    current_time = datetime.now().strftime("%Y%m%d-%H%M%S")
    experiment_name = f"kubeflow-pipeline-{current_time}"

    experiment_id = client.create_experiment(experiment_name)

    print(f"Created a new MLflow experiment: {experiment_name} with ID: {experiment_id}")

    return experiment_name
```

Figura 3: Exemplo de um componente da pipeline de ML

A pipeline final fica definida da seguinte forma (Figura 4) , e como é possivel ver, a pipeline está definida em 6 fases:

- Carregamento dos dados
- Criação de um MLFlow Experiment
- Treino dos modelos
- Seleção do melhor modelo

- Upload do melhor modelo
- Deploy do melhor modelo

Quanto à primeira fase, há um pormenor que precisa de ser mencionado, dentro da rede da UA, e consequentemente na rede onde o cluster de Kubernetes está a correr não é possível acessar a Feature Store, certamente a firewall deve estar a bloquear tal conecção. De modo a culmatar o problema, todo o pré-processamente de dados é de igual modo realizado do lado da Feature Store, no entanto aqui na pipeline os dados são lidos de um repositório do GitHub criado por mim, onde coloquei lá o mesmo output que a Feature Store daria já com os dados pre-processados.

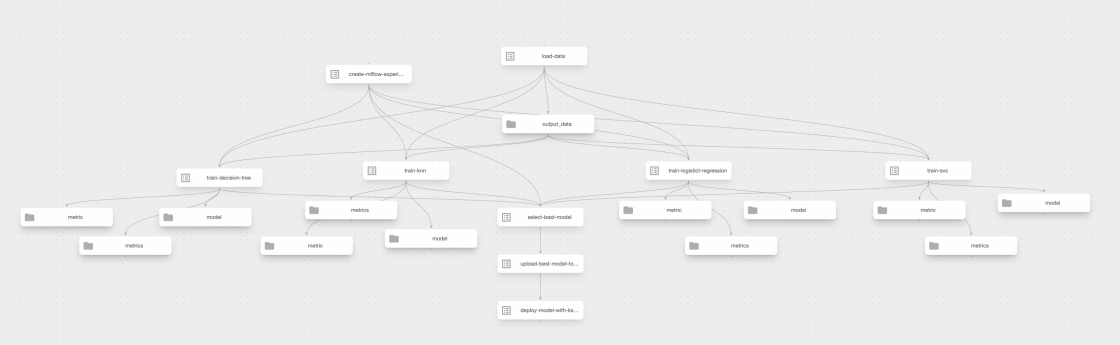

Figura 4: Pipeline Final

Não foi desenvolvida nenhuma pipeline que fizesse uso do AutoEncoder, falado na secção anterior, dado que os CPU's onde o cluster de Kubernetes estava a correr não suportavam instruições AVX2, necessários para a execução do Tensorflow.

A pipeline depois de correr tem o seguinte aspeto (Figura 5), todas os componentes a verde, significa que não houve nenhum erro na execução da pipeline e o melhor modelo do conjunto de modelos treinados foi deployed e encontra-se acessível atraves de um endpoint.

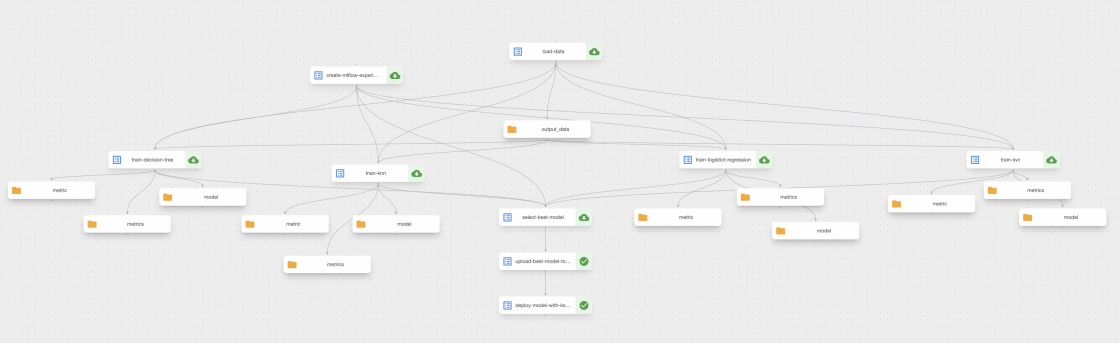

Figura 5: Aspeto da Pipeline depois de correr

O deployment do modelo de ML é feito com recurso ao KServe, que é a ferramenta que o kubeflow disponibilizaa nativamente para este efeito. O modelo depois de estar deployed fica

acessivel através do seguinte URL: internamente pelo http://[nome-do-deployment].svc.local.cluster, e pode também ter um acesso externo com a utilização de um ingress, ou no caso mais simples atraves de se fazer port-forward do endpoint em que o modelo de ML está deployed.

Na interface do Kubeflow, é possível observar os modelos que foram deployed, na aba "Endpoints", a Figura 6, ilustra essa feature que a plataforma disponibiliza.

Na Interface do Kubeflow, é possível também verificar métricas e os logs do modelo, como as Figura 7 e Figura 8 mostram, criando assim uma forma rápida e acessível de verificar o estado do modelo, quer através da visualização de métricas através do Grafana integrado no Kubeflow, quer da visualização dos logs dos modelos.

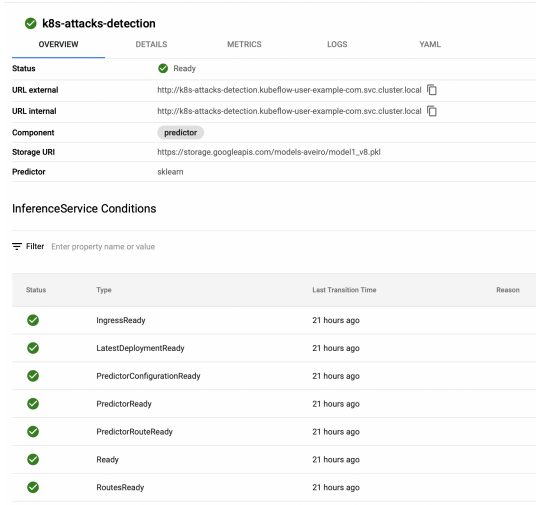

Figura 6: Visão geral da aba Endpoints

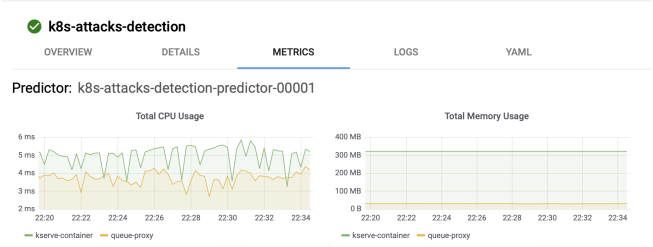

Figura 7: Métricas (CPU/RAM) do modelo em produção

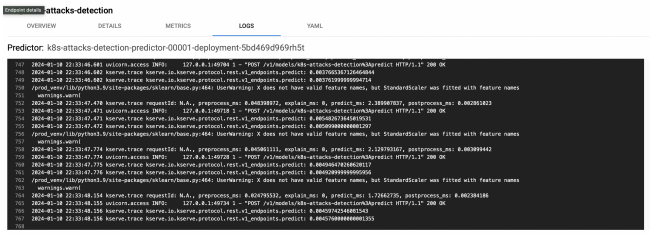

Figura 8: Logs do modelo em produção

7 Resultados

Relativamente aos resultados, os mesmos encontram-se nas seguintes tabelas, onde para a abordagem Baseline os resultados estão na Tabela 1, e para a abordagem com AutoEncoder os resultados estão na Tabela 2. Cada tabela mostra também os melhores parametros selecionados pela GridSearch.

Sobre a abordagem Baseline os resultados são os seguintes (Tabela 1 e Figura 9):

Model	Default - Accuracy	GridSearch - Accurtacy	Parameters
SVC	0.974	0.978	C: 1, gamma: 1, kernel : rbf
Logistic Regresion	0.807	0.809	C: 10, solver: newton-cg
KNN	0.997	0.998	metric: manhattan, n-neighbors: 3, weights: distance
Decision Tree	1.000	0.999	criterion: gini, max-depth: 5, min-samples-leaf: 1

Tabela 1: Resultados dos modelos - Baseline

O modelo com melhor performace é a Decision Tree, alcançando uma accuracy de 1, a sua matrix de confusão pode ser consultada na Figura 9

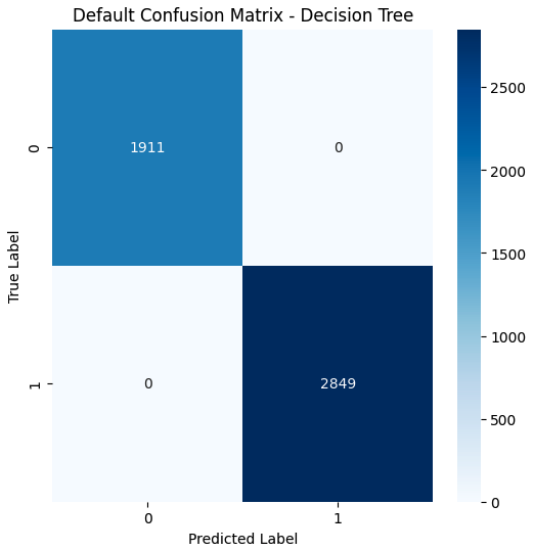

Figura 9: Matrix de confusão do melhor modelo - Baseline

Sobre a abordagem AutoEncoder os resultados são os seguintes (Tabela 2 e Figura 10):

O modelo com melhor performace é o KNN, alcançando uma accuracy de 0.998, a sua matrix de confusão pode ser consultada na Figura 10

Model	Default - Accuracy	GridSearch - Accurtacy	Parameters
SVC	0.883	0.887	C: 0.1, gamma: 1, kernel: rbf
Logistic Regresion	0.767	0.762	C: 0.1, solver: newton-cg
KNN	0.997	0.998	metric: euclidean, n-neighbors: 3, weights: distance
Decision Tree	0.995	0.995	criterion: entropy, max-depth: 20, min-samples-leaf: 2

Tabela 2: Resultados dos modelos - AutoEncoder

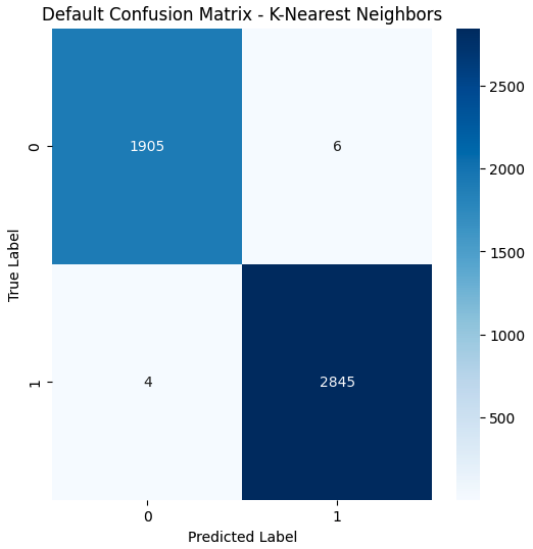

Figura 10: Matrix de confusão do melhor modelo - AutoEncoder

Em ambas as abordagerns os modelos conseguiram minimizar ao máximo os falsos positivos, e os falsos negativos, distinguindo bem quando algo era considerado benigno ou maligno.

Fazendo a devida análise dos resultados, podemos concluir que o AutoEncoder não foi significativo para a melhoria dos resultados final no que toca à métrica escolhida, a accuracy, e o modelo com melhor performance foi o Decision Tree na abordagem Baseline com uma accuracy de 1, no entanto a abordagem com o AutoEncoder não foi péssima uma vez que obteve um resultado de 0.998.

8 Validação

Para a validação do modelo, são usados os dados de validação que se encontram guardados na Feature Store, e foi criada uma interface com recurso ao *streamlit*. O objetivo é a cada

0.35 segundos ser feito um pedido ao modelo e comparar o resultado do pedido (o que foi previsto) com o que era expectável (uma vez que os dados de validação estão identificados se são malignos ou não). A interface no *streamlit* não é nada mais que um gráfico que é atualizado a cada 0.35 segundos com a informação se o modelo acertou ou não a previsão que fez.

Apesar do melhor modelo ter uma performance (accuracy) de 1, neste processo de validação, ou seja com o modelo em produção e com dados que nunca viu, o mesmo não se confirma, como é possivel ver na Figura 11, onde em 685 previsões o modelo acertou em pouco mais de 600 e errou certa de quase 100 vezes. Mostrando assim que o comportamento dos modelos em produção não é 100% igual ao comportamento na fase de treino.

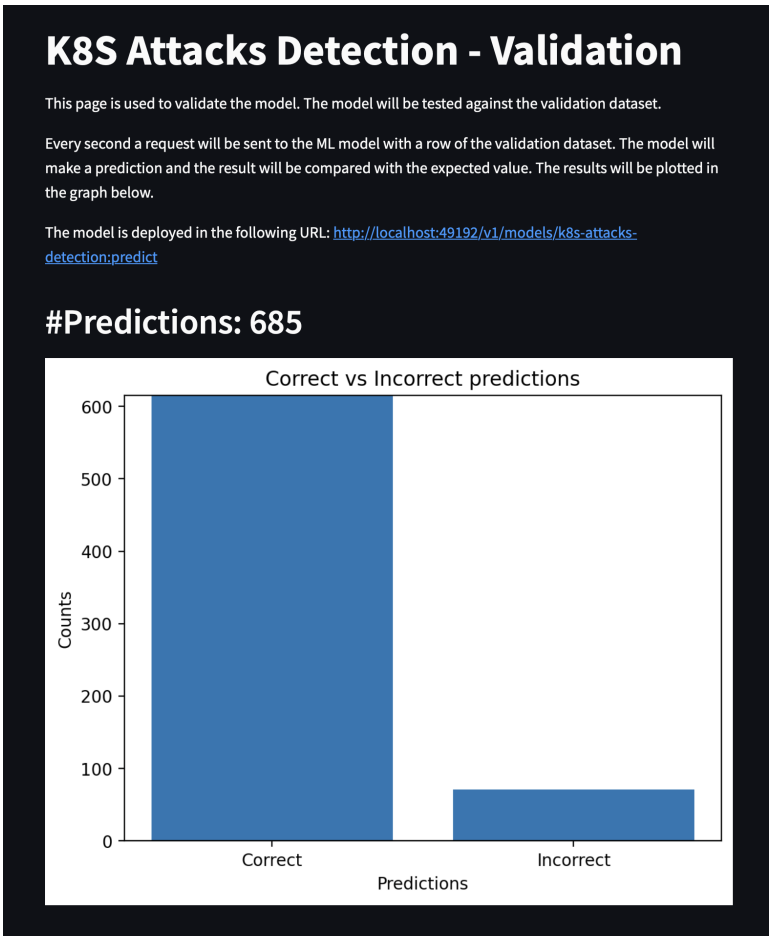

Figura 11: Visão geral da aba Endpoints

9 Conclusão

De um modo geral todos os objetivos propostos na Secção 2 foram atinguidos com sucesso. Com todo o desenvolvimento da pipeline de ML e com a utilização da Feature Store, consegiu-

se ter um sistema final suficientemente robusto e que permite uma rápida adaptabilidade a mudanças quer de pre-processamento, quer da pipeline de ML.

A Feature Store revela-se uma componente muito importante no sistema, pois permite separar a fase de pré-processamento dos dados, aumentado o foco neles e criando mecanismos que permitam no final ter uma melhor qualdiade na parte de engenharia de dados.

O desenvolvimento de uma pipeline de ML, permite então uma maior rapidez na implementação de novos modelos por exemplo, fazendo reduzir o intervalo de tempo que um sistema de ML demora desde que um novo modelo começa a ser implementado até que o mesmo seja avaliado para ser colocado em produçao.

O proceso de validação revela-se crucial para uma melhor analise do melhor modelo selecionado pela pipeline, pois como visto, apesar de ser um modelo com accuracy de 1 revelou que também falha pervisões, mostrando que o comportamento de um modelo em produção não é igual ao comportamento do modelo na sua fase de treino, pois os dados não são os mesmos.

10 Trabalho Futuro

Como trabalho futuro alguns pontos a considerar são:

- Melhoramento da construção da pipeline de ML.
- Implementação de AutoML
- Criação de triggers que permitam a implementação do conceito de Continuous Training

Referências

[1] "Assuremoss kubernetes run-time monitoring dataset," accessed: 2023-12-10. [Online]. Available: https://data.4tu.nl/articles/dataset/AssureMOSS_Kubernetes_Run-time_Monitoring_Dataset/20463687

[2] "Learning state machines to monitor and detect anomalies on a kubernetes cluster," accessed: 2023-12-11. [Online]. Available: https://dl.acm.org/doi/pdf/10.1145/3538969.3543810

[3] "Kubanomaly dataset on github," accessed: 2023-12-13. [Online]. Available: https://github.com/a18499/KubAnomaly_DataSet

[4] "Learning state machines to monitor and detect anomalies on a kubernetes cluster," accessed: 2023-12-10. [Online]. Available: https://dl.acm.org/doi/fullHtml/10.1145/3538969.3543810#BibPLXBIB0001

[5] "Aiops: Simple anomaly detection in kubernetes with active monitor and tensorflow," accessed: 2023-12-10. [Online]. Available: https://medium.com/keikoproj/aiops-simple-anomaly-detection-in-kubernetes-with-active-monitor-and-tensorflow-24727c1606e5

[6] "A kubernetes dataset for misuse detection," accessed: 2023-12-13. [Online]. Available: https://www.itu.int/dms_pub/itu-s/opb/jnl/S-JNL-VOL4.ISSUE2-2023-A26-PDF-E.pdf