

EDAS 011**Uso de API para consumir modelos de *machine learning*****Aluno:****Rodrigo Lacerda****Professor:****Giorgio de Tomi****29/01/2023**

Sumário

Resumo	3
Introdução.....	4
Revisão da Literatura	5
Método	6
1. Fastapi.	7
2. Docker (Conteirização)	8
3. API gateway (aws)	11
Resultados e Discussão	13
Conclusões	14
Referencias.....	15

Resumo

Criar um modelo de *machine learning* é sem dúvidas o trabalho principal de todo cientista de dados, porém, é preciso ir além para que o modelo seja usado em ambientes profissionais e de produção de alta escalabilidade para que realmente se tire valor dele.

Para isso, precisa ser desenvolvido todo um planejamento e desenvolvimento, para que o modelo seja hospedado em uma plataforma de fácil implementação, escalável e com baixo custo. Assim, pode-se aproveitar ao máximo os benefícios oferecidos pela ciência de dados.

Esse artigo visa expor um processo para a criação de uma API (*application plataforma interface*) que possibilitará o consumo de um modelo de *machine learning* de classificação de empréstimos, bastando para isso, de uma internet.

Introdução

A área de ciências de dados cresceu bastante nos últimos anos, sendo proposto diversos estudos e análises para os mais variados problemas enfrentados tanto pelos negócios e, como também, pela sociedade.

Isso, sem dúvidas, enriqueceu bastante o leque de atuação dos cientistas de dados, e proporcionou uma crescita acelerada no número de vagas de emprego.

Imediatamente, com o rápido crescimento dos usos de modelos de ciência de dados, surgiu uma alta necessidade de profissionais para juntar esse programa junto com outras áreas das empresas, e prestar a constante manutenção e monitoramento desses fluxos e modelos, após serem colocados em produção.

Antigamente, tal função era própria para desenvolvedores back-end ou engenheiro de dados, entretanto, em empresas de rápido crescimento ou com estruturas enxutas, nem sempre pode-se dispor de um profissional a mais para essa função.

Por consequência, outra área está em alta, que é o MLOps (junção do inglês de *Machine learning* e *DevOps*), que é responsável por toda parte de implementação do modelo na nuvem e criação de pipelines para a automação desses fluxos de implementação e testes, mas também pelo seu monitoramento.

Desse modo, os cientistas de dados podem focar em exploração e testes com os grandes conjuntos de dados, e os engenheiros de *machine learning* podem se preocupar em criar os melhores modelos, deixando todas as etapas de automação, monitoramento e implementação desses pelos profissionais de MLOps.

Será detalhado nesse artigo todas as etapas que um profissional de MLOps deverá saber para que consiga exercer esse papel de elo entre cientistas de dados e a equipe de tecnologia.

Com o intuito disso, será desenvolvido e implementado um modelo de *machine learning* utilizando a biblioteca de criação de API chamada FastAPI, desenvolvida para a linguagem de programação Python. E, depois, será implementado na infraestrutura de nuvem da AWS® (*Amazon Web Services*), onde poderá ser consumido por qualquer pessoa com acesso à internet.

Revisão da Literatura

Para ser possível que seu modelo seja consumido por qualquer pessoa ou aplicação é necessário seguir padrões e protocolos conhecidos como API (do inglês *Application Programming Interface*), que em simples modos, é uma forma de integrar sua própria aplicação com outras de terceiros ou internas.

Como exemplo, a API do *Google Maps* é bastante utilizada por diversos outros sites para conseguir informações de localização e mapas atualizados. Nesse caso, o google possui servidores com milhares de registros e ele disponibiliza para acesso através de sua API, onde terceiros podem criar uma chave de acesso e usar para serem usadas em suas próprias aplicações.

Ela funciona de várias maneiras, mas a mais flexível e popular é a arquitetura Rest (*Representational State of Transfer*), que usa métodos HTTP (*hypertext transfer protocol*), como POST, UPDATE, GET, DELETE e outros.

Nesses métodos, podem ser usados parâmetros no próprio corpo requisição para se criar lógicas mais complexas, como informar valores para serem inseridos como entrada em um modelo de machine learning, para gerar uma resposta.

Por exemplo, existem muitos modelos de ciências de dados sobre classificação de fraudes de cartão de crédito, a partir de uma compra qualquer, dependendo de vários critérios, como o valor da compra, horário ou tipo de produto, poderia ser enviado no corpo de uma requisição HTTP para um modelo de machine learning na nuvem, que depois devolveria a resposta imediatamente se é uma compra fraudulenta ou não.

Outra analogia bem popular, seria comparar o padrão API Rest como o processo de pedir ordem em um restaurante, por exemplo: o cliente, após ler o cardápio (que poderia ser considerado a documentação da API que será consumida) e escolher seu prato, informa ao garçom o seu prato escolhido, (o garçom funciona como o método *Get* do *HTTPs*), que deverá acionar o cozinheiro sobre o que deverá cozinhar (aqui o cozinheiro funciona como o servidor da empresa onde estão armazenados todos dados, desse modo, assim que ele recebe a ordem do método *Get*, ele busca pelos produtos necessários, conclui o prato e passa para o garçom devolver a 'resposta' para o cliente.)

Método

Esse artigo foi desenvolvido a partir de um trabalho passado (para acessar todo código e análises acesse esse [link](#)) em que criamos um modelo de machine learning para a classificação de um empréstimo, o objetivo era limpar o conjunto de dados, com valores faltantes e outliers, identificar os atributos que mais significaria para a previsão do empréstimo, e desenvolver e comparar diversos modelos para escolher um que performance melhor no teste.

O conjunto de dados tem as seguintes colunas:

Variável	Descrição
Loan_ID	ID único do empréstimo (numérico)
Gender	Masculino/Feminino (Male/Female)
Married	Casado - sim ou não (Y/N)
Dependents	Número de dependentes
Education	Escolaridade (Graduate / Under Graduate)
Self_Employed	Auto empregado - sim ou não (Y/N)
ApplicantIncome	Renda do aplicante
CoapplicantIncome	Renda do fiador
LoanAmount	Montante do empréstimo, em milhares
Loan_Amount_Term	Prazo do empréstimo, em meses
Credit_History	Histórico de crédito corresponde aos critérios (1 ou 0)
Property_Area	Localização da propriedade (Urban/Semi Urban/ Rural)
Loan_Status	Empréstimo aprovado - sim ou não (Y/N)

(imagem: colunas do conjunto de dados de empréstimo).

```
df.head(10)
```

	Loan_ID	Gender	Married	Dependents	Education	Self_Employed	ApplicantIncome	CoapplicantIncome	LoanAmount	Loan_Amount_Term	Credit_History	Property_Area	Loan_Status
0	LP001002	Male	No	0	Graduate	No	5849	0.0	NaN	360.0	1.0	Urban	Y
1	LP001003	Male	Yes	1	Graduate	No	4583	1508.0	128.0	360.0	1.0	Rural	N
2	LP001005	Male	Yes	0	Graduate	Yes	3000	0.0	66.0	360.0	1.0	Urban	Y
3	LP001006	Male	Yes	0	Not Graduate	No	2583	2358.0	120.0	360.0	1.0	Urban	Y
4	LP001008	Male	No	0	Graduate	No	6000	0.0	141.0	360.0	1.0	Urban	Y
5	LP001011	Male	Yes	2	Graduate	Yes	5417	4196.0	267.0	360.0	1.0	Urban	Y
6	LP001013	Male	Yes	0	Not Graduate	No	2333	1516.0	95.0	360.0	1.0	Urban	Y
7	LP001014	Male	Yes	3+	Graduate	No	3036	2504.0	158.0	360.0	0.0	Semiurban	N
8	LP001018	Male	Yes	2	Graduate	No	4006	1526.0	168.0	360.0	1.0	Urban	Y
9	LP001020	Male	Yes	1	Graduate	No	12841	10968.0	349.0	360.0	1.0	Semiurban	N

(imagem: dataframe do conjunto de dados de empréstimo).

Aonde ***Loan_Status*** é a variável dependente que queremos classificar.

O modelo com melhor performance foi o Random Forest, utilizando somente 4 atributos: ***Total_Income*** (Combinação entre *ApplicationIncome* e *CoapplicationIncome*), ***LoanAmount***, ***Credit_History*** e ***Property_Area***).

```
model_randomforest = RandomForestClassifier(n_estimators=25, min_samples_split=25, max_depth=7, max_features=1)
predictor_var = ['TotalIncome', 'LoanAmount', 'Credit_History', 'Property_Area']
model_random_forest_final = classification_model(model_randomforest, df, predictor_var, outcome_var)
```

Acurácia : 84.853%
Score da Validação Cruzada : 82.114%

(imagem: score de performance do modelo de machine learning).

Assim, com o modelo treinado e escolhido, foi usado a biblioteca Pickle para ser exportado e possibilitar seu uso no futuro:

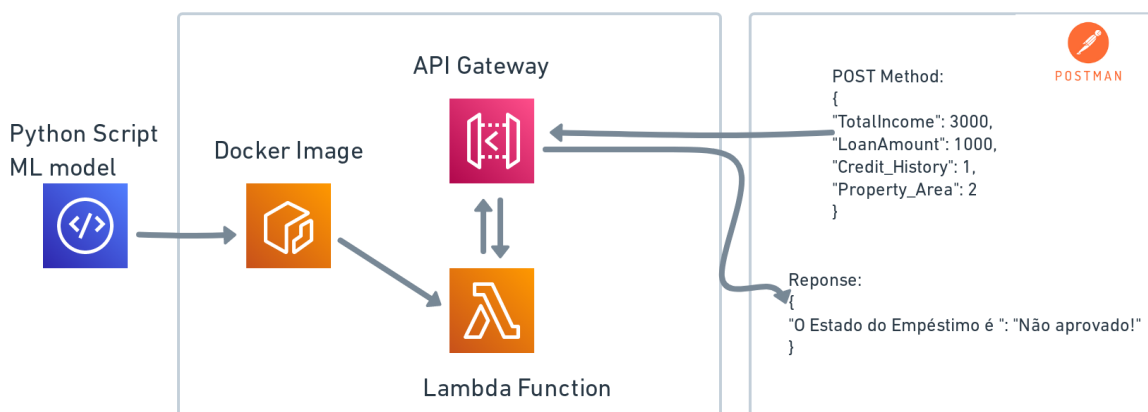
```
# Salvando os modelos
import pickle
import os
if not os.path.exists('models'):
    os.mkdir('models')

#salvando o random forest classifier
with open("fast_api_folder/ML_models/randomforestclassifier.pkl", "wb") as file:
    pickle.dump(model_random_forest_final, file)
```

✓ 0.1s

(imagem: salvando modelo de machine learning em formato Pickle).

1. Project Design.



(imagem: esquema completo do projeto).

O design do projeto começa com a criação do modelo de machine learning com o Python e depois do ajuste do modelo com o FastAPI, após essa parte de desenvolvimento, é criada uma imagem (ou container) com o Docker, que depois será armazenado em uma instância do AWS ECR.

Assim, consegue-se ligar a lambda function com essa imagem e configurar a API através da AWS API Gateway. Com essa configuração pronta da api, já retornado um endereço URL com o endpoint para consulta.

Com esse endereço pode-se testar com um programa de consultas API, como o postman ou insônia, nesse caso, foi optado pelo postman.

2. FastAPI / Docker (Containers)

FastAPI é uma estrutura de desenvolvimento de APIs com a linguagem de programação Python.

Ela obteve uma forte aceitação na comunidade de desenvolvedores por ter uma implementação simples e rápida, mas com uma estrutura bastante estável e quase sem bugs.

Além disso, ela apresenta a melhor performance de rapidez comparados com outros frameworks com a mesma finalidade para Python, como Flask e Django.

No nosso caso foi desenvolvido somente um endpoint com o propósito de enviar diferentes valores de *Total_Income*, *LoanAmount*, *Credit_History* e *Property_Area*) e depois saber se dependendo desses parâmetros será aprovado ou não o empréstimo.

Então, foi pensando em um método HTTP Post, que consiga ser enviado no body da requisição todos esses parâmetros:


```

st_api_folder > house_price_main.py > predict_loan_status
1  from fastapi import FastAPI
2  import pickle
3  import pandas as pd
4  from mangum import Mangum
5  from schemas.schemas import Predictor
6
7
8  # Carregando Pickle file
9  with open("ML_models/randomforestclassifier.pkl","rb") as file:
10     pickle_model = pickle.load(file)
11
12  app = FastAPI()
13
14  handler = Mangum(app)
15
16  # @app.get("/")
17  # def root():
18  #     return {"msg":"working!"}
19
20  @app.post("/predictions")
21  async def predict_loan_status(prediction: Predictor):
22
23     predictions = {"TotalIncome": int(prediction.TotalIncome),
24                  "LoanAmount": int(prediction.LoanAmount),
25                  "Credit_History": int(prediction.Credit_History),
26                  "Property_Area": int(prediction.Property_Area)}
27
28     df_predictions = pd.DataFrame([predictions])
29     result = pickle_model.predict(df_predictions)[0]
30
31     if result == 0:
32         message = "Não aprovado!"
33     else:
34         message = 'Aprovado!'
35
36     return {"O Estado do Empéstimo é " : message}

```

(imagem: código especificando o uso do FastAPI para desenvolver api por Python).

- 1) Aqui especifica-se o tipo de método que será usado, aqui usamos o prefixo “/prediciton” que será usado como o endpoint para esse método;
- 2) Aqui especifica-se o que está sendo esperado pelo método no corpo da requisição, que seriam todas as variáveis necessárias para que o modelo de machine learning precisa para retornar um resultado;
- 3) Por fim, dependendo da resposta que o modelo enviar, (voltando para a analogia do restaurante, seria o prato de comida pronto para ser consumido), e enviamos uma resposta se fora ou não aprovado o empréstimo.

Após esse último desenvolvimento em Python, precisamos organizar tudo em um container que sirva para várias aplicações e sistemas diferentes, para isso foi utilizado o Docker que cumpre essa função de organizar todo o código e dependências em um arquivo que possa ser usado para diferentes sistemas

operacionais ou aplicações, nesse caso, foi enviado como imagem para o serviço da Amazon chamado ECR (*Elastic Container Registry*).

```
FROM public.ecr.aws/lambda/python:3.8

# Install the function's dependencies using file requirements.txt
# from your project folder.

COPY ML_models ./ML_models
COPY schemas ./schemas

COPY requirements.txt .
RUN pip3 install -r requirements.txt --target "${LAMBDA_TASK_ROOT}"

# Copy function code
COPY house_price_main.py ${LAMBDA_TASK_ROOT}

# Set the CMD to your handler (could also be done as a parameter override outside of the Dockerfile)
CMD [ "house_price_main.handler" ]
```

(imagem: código para transformar o projeto em uma imagem do Docker).

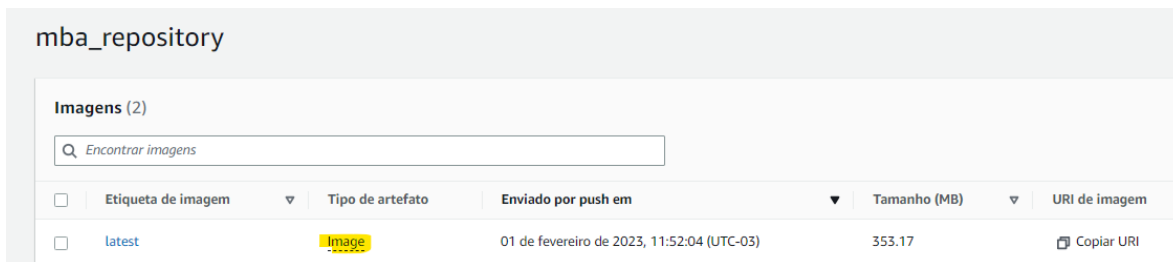
Nesse script é especificado quais os diretórios do projeto deverão ser copiados (pasta modelos, aonde se encontra o modelo Pickle, e também schemas, que é aonde está a classe construída para especificar os campos do corpo da requisição), instalar as dependências ou bibliotecas necessárias para rodar o modelo (que estão escritas no arquivo requirements.txt) e por fim define o nome da função que o serviço da Amazon Lambda deverá atender quando for chamado, nesse caso

Como observado, Brasil, Bélgica e França são as seleções que estão mais recentemente no pódio do rank FIFA.

```
-----
failed to compute cache key: "/house_price_main.py" not found: not found
PS H:\Meu Drive\02) MBA\MBA_Projects\ProjetoFinal_ML_API_AWS\fast_api_folder> docker build -t mba_repository .
[+] Building 28.9s (11/11) FINISHED
=> [internal] load build definition from Dockerfile
=> => transferring dockerfile: 534B
=> [internal] load .dockerignore
=> => transferring context: 2B
=> [internal] load metadata for public.ecr.aws/lambda/python:3.8
=> CACHED [1/6] FROM public.ecr.aws/lambda/python:3.8@sha256:b34bd5e42c111aaa23f05c5440064c0254e3b6085e83af7772c7f9356216ab
=> [internal] load build context
=> => transferring context: 1.33kB
=> [2/6] COPY ML_models ./ML_models
=> [3/6] COPY schemas ./schemas
=> [4/6] COPY requirements.txt .
=> [5/6] RUN pip3 install -r requirements.txt --target "/var/task"
=> [6/6] COPY loan_predictions.py /var/task
=> exporting to image
=> exporting layers
=> writing image sha256:868340e777750d7bea4339222d7345ee91cfc865c1b802f06147ccd5cf00c481
=> naming to docker.io/library/mba_repository
```

(imagem: comando no terminal mostrando o momento que é construído uma imagem Docker a partir do código especificado).

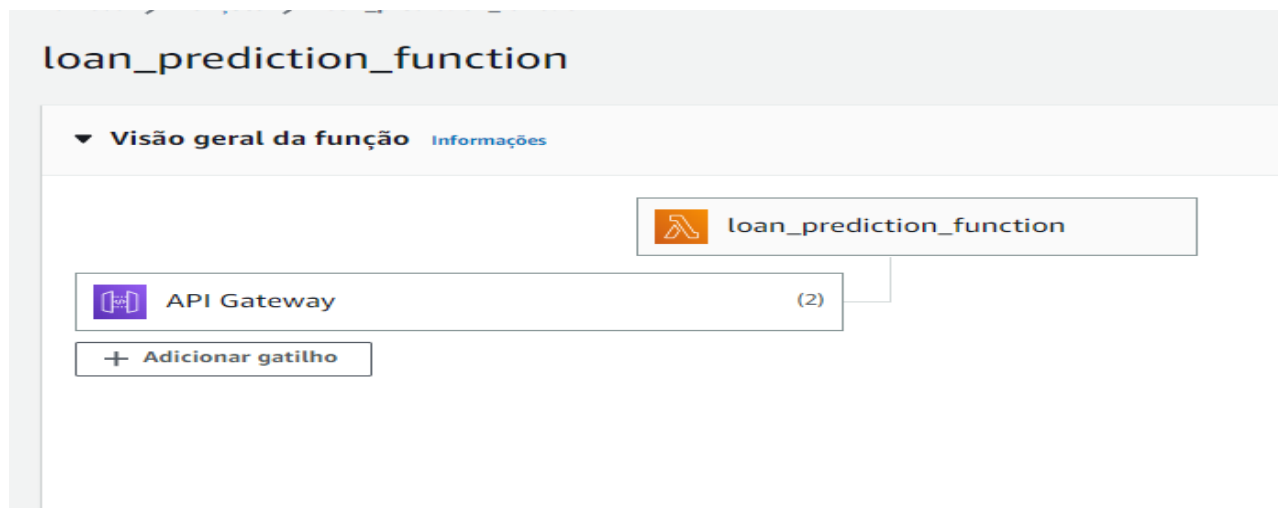
Por fim, ele é lançado (*Push* do inglês) para o ECR:



(imagem: repositório ECR com a imagem do Docker file).

3. AWS Lambda Function / AWS API gateway

Por fim, foi criada uma função lambda que é encarregada de rodar o modelo de machine learning dentro da imagem Docker.



(imagem: esquema da aws lambda function, com gatilho da api gateway).

Ela é configurada com Python 3.8 e tem tempo de até 3 min para rodar o script.


Após configurar o Lambda Function, basta criar uma API Rest pela AWS api gateway, aonde com algumas simples configurações, conseguimos fazer o deploy (implementação) da nossa API na nuvem.

Na imagem abaixo, ainda tem um ambiente teste que mostra todo o fluxo de informação dessa API, começando pelo Cliente (cliente, nesse caso agente externos e terceiros) faz uma solicitação HTTP, nossa api envia para a lambda function (Solicitação de integração) que ativa nossa função lambda, aonde vai rodar nosso modelo de machine learning e depois devolver para API a resposta.



(imagem:fluxo da api pela aws api gateway).

Após implementar a API, será gerado nossa URL que poderá ser usada por qualquer pessoa ou aplicação.

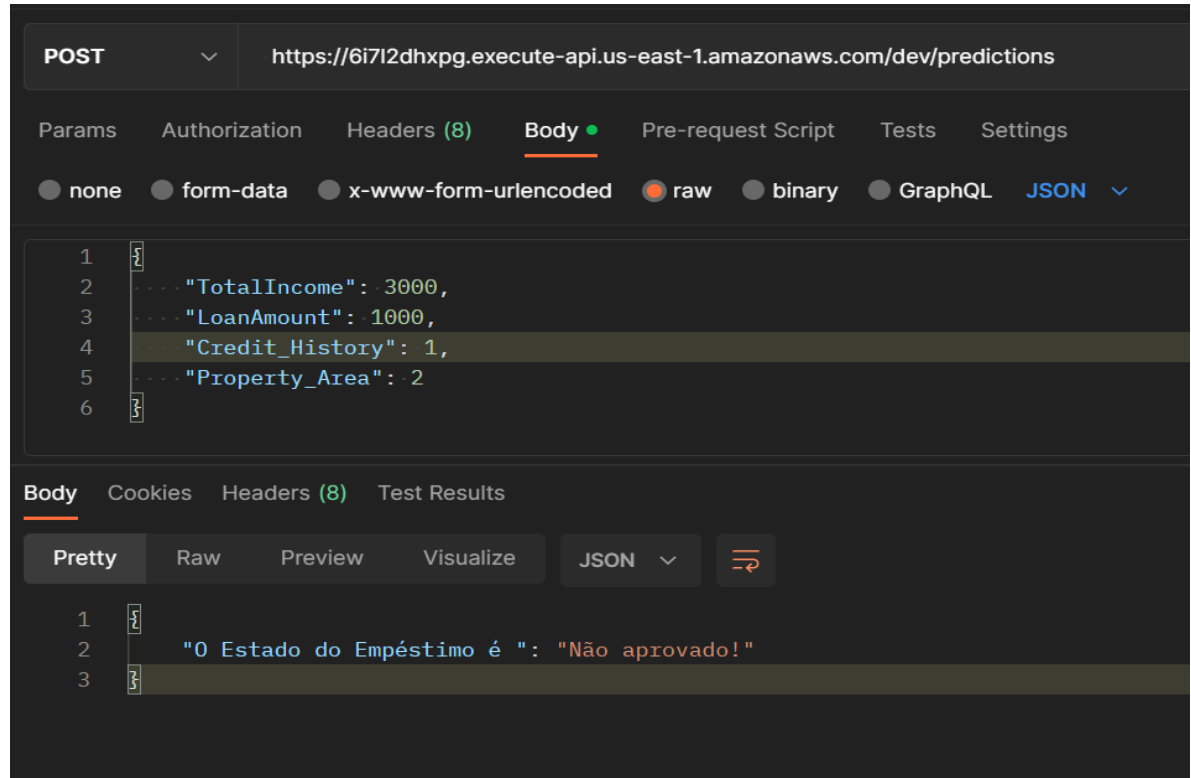
 Invocar URL: <https://a2czf6h0j0.execute-api.us-east-1.amazonaws.com/dev>

(imagem:fluxo endpoint da nossa API).

Resultados e Discussão

1. Postman tests

Para testar o modelo foi utilizado o programa Postman, onde podemos criar uma requisição HTTP e colocar os parâmetros no corpo da mensagem:



(imagem: teste da API com Postman).

Aqui podemos ver:

- 1) que o método do tipo usado é POST,
- 2) O endpoint <https://a2czf6h0j0.execute-api.us-east-1.amazonaws.com/dev/predictions>
- 3) O corpo da requisição com todas as variáveis, importante ressaltar que precisar estar no formato JSON.
- 4) A resposta enviada pela API.

Agora, podemos consultar diferentes tipos de parâmetros para entender se devemos ou não aprovar o empréstimo, a melhor parte é que isso poderia ser integrado com uma aplicação mobile ou web, aonde várias pessoas poderiam se beneficiar desse programa.

Conclusões

Desenvolver toda parte de implementação de um modelo de machine learning na nuvem é a última parte de todo ciclo das ciências de dados, e cada vez mais empresas estão precisando de profissionais com experiências de *machine learning*, mas também de implementação, manutenção e monitoramento deles em ambientes de produção.

Desse modo, entender as principais plataformas e serviços na nuvem e, principalmente, o fluxo e planejamento de como implementar esses modelos é essencial para que o modelo desenvolvido possa ser facilmente e de maneira escalada, consumida para usuários terceiros ou outras aplicações.

Para trabalhos futuros, pode-se melhorar a parte teórica do funcionamento de cada tecnologia aplicada nesse projeto, também criar uma rotina para entender o limite que essa api suporta de requisições por segundos sem cair, por fim, testar o método com diferentes opções de implementação na nuvem, para comparar custo, facilidade e manutenção.

Referencias

<https://colab.research.google.com/drive/1P2Ao5k5a3p13pg2WmBOI7XEnd4nL7MvM>

<https://medium.com/@selfouly/mlops-done-right-47cec1dbfc8d>

<https://aws.amazon.com/pt/what-is/api/>

<https://medium.com/p/3b9a1cec5e13>

<https://docs.docker.com/desktop/install/windows-install/>

<https://docs.aws.amazon.com/lambda/latest/dg/python-image.html>