# Plataformas Cognitivas – Trabalho final

Disclaimer: Para rodar o projeto é necessario colocar os arquivos sensiveis na raiz do projeto. Estamos enviando um zip com esses arquivos mais o link do repositório.

## Parte 1 - Obtenção dos dados

A obtenção dos dados foi feita de forma programática no arquivo **get\_data/get\_data.py.** É uma classe, para poder ser reutilizada em outros lugares.

Metodos da classe:

* \_\_init\_\_ : No instanciamento da classe procura nas variaveis de ambiente (.env) o nome do bucket onde o csv vai ser salvo e o arquivo de credenciais do GCE (para subir de forma programatica sem precisar de autenticação no navegador)
* get\_big\_query\_data: Metodo que busca no BIG Query a base de dados. É parametrizado, que posteriormente vai chamar a base 'emf-teacher.'trabalho\_loans.loan\_default'
* write\_in\_storage\_as\_csv: Escreve no bucket do GCE tanto o dado como CSV tanto quanto um arquivo de metadado em JSON
* get\_url\_file: Retorna a url do arquivo posteriormente salvo no bucket
* get\_file: Retorna o arquivo posteriormente salvo no bucket
* get\_to\_temp\_folder: faz download do arquivo do bucket e salva em uma pasta temporaria local.

Além da classe por si só, fizemos um script **get\_data/update\_data\_on\_gce.py** que caso chamado pelo terminal vai atualizar o arquivo no bucket e devolver a url tanto do arquivo de metadado quanto do CSV.

Além disso criamos um nootbook para fazer uma melhor exploração dos dados antes de partir para os modelos utilizando diretamente o link do bucket como fonte de dados. O Nootbook está localizado em **explore\_data/explore.ipynb.**

Depois de explorar os dados, decidimos criar uma classe para normalização dos dados: **normalization/normalization.py** que baiscamente trata a base para deixar ela toda por igual, para podermos usar os dados da mesma forma nos três modelos.

Além da classe há um scriptem **normalization/generated\_normalize\_csv.py** que vai chamar a classe e gerar um arquivo local na pasta temporaria em **tmp/data\_normalized.csv.** A motivação desse aquivo é principalmente para podermos usar os dados normalizados em R.

## Parte 2 - Propensão de inadimplência em R

A obtenção dos dados em R foi igual aos outros modelos, após a execução e extração dos dados do big query e normalização pelo script python, um arquivo csv é salvo em uma pasta temporária dentro do projeto, por onde fizemos a leitura no R utilizando o método read.csv.

Para transformarmos e servir nosso modelo em uma API utilizamos o Plumber. O código é bem simples, apenas jogamos o modelo que foi salvo no passo anterior em uma pasta temporária do projeto para um objeto utilizando o método readRDS e então setamos os parâmetros da API, como o método post e o serializer.

Para a criação da função, ela recebe apenas 2 parâmetros. O primeiro parâmetro req é responsável por conseguirmos pegar o body do JSON que será enviado no método POST, já o segundo são os dados em si. Definimos um dataframe dentro da função e iteramos dentro dele para cada variável que o modelo precisa para predizer.

O output do modelo irá retornar 0 ou 1, de acordo com o threshold definido de 0.5, caso maior que o threshold será 1 para a inadimplência.

* Também criamos um arquivo para normalização dos dados em R. Caso seja necessário, o arquivo deve estar local e a normalização é exatamente igual a feita em python.
* Temos um arquivo requirements.R para instalar as bibliotecas necessárias para rodar tanto o modelo quanto o endpoint.
* Já no arquivo server.R ele apenas irá chamar os requirements antes de checar se o modelo existe na pasta temporária do projeto. Depois, caso o modelo não existe, ele irá executar o arquivo do modelo.

## Parte 3 - Clusterização e classificação em Python

TODO

## Parte 4 - Modelo de redes neurais

O treinamento em redes neurais é realizado pelo arquivo **federation\_model/train.py.** Primeiro pegamos os dados normalizados usando 80% de treino e desses 80%, 20% para validação. O modelo foi treinado em 5 epocas, que deu uma acuraria de mais de 95%.

Usamos uma rede neural sequencial com:

* Uma camada de 128 neuronios com ativação relu
* Uma camada de dropout
* Uma camada de 1 neuronio com ativação sigmoid

Depois o modelo é salvo na pasta temporaria e tudo enviado para o Azure ML.

O serving do modelo é feito com Flask no arquivo **federation\_model/server.py**, apensar esperando um JSON no body da requisição no formato **`{ data: [] }`** onde o Array precisa ser uma Matriz de 1x51 contendo os valores númericos esperado pelo modelo. O Modelo só consegue interpretar números então normalizamos as variaveis categorias em colunas de 0 e 1.

Por ultimo o Docker em que ele é servido é bem simples. É o arquivo **dockerfile** na raiz do projeto, ele basicamente chama o server.py expondo em uma porta passada via argumento.

Video da explicação sobre o modelo: [Link](https://www.youtube.com/watch?v=dsbTUi0LT0Y)

## Parte 5 - Criação da Plataforma Cognitiva

Para a criação da plataforma Cognitiva criamos o model manager dentro do GCE.

A criação do model manager está no arquivo **model\_manager/server.py.** Também é um arquivo de Flask com duas rotas:

* Path: ‘/predict’

Method: ‘POST’

Body: ‘{data: […], model: ‘federation\_model | r\_model | cluster\_model’}’

Função: Faz a predição do modelo selecionado atraves do valor de model, mandando uma requisição HTTP e retornando o seu valor como JSON

* Path: ‘/download’

Method: ‘GET’

Função: Faz o download dos Logs gerados pelo model Manager

O Model manager le o arquivo **config/microservices.json** gerado pelo script **generate\_config.sh** que le e salva o IP de quando os Docker’s dos modelos estão de pé para usar de referencia nas chamadas HTTP.

Com o server funcional, dockerizamos ele no arquivo **dockerfile.model\_manager** basicamente expor o Flask de forma simples.

Com tudo isso pronto criamos um script para a criação da VM de forma programatica no GCE, localizado em **vm/ create\_vm.py.** Esse script cria a VM e um Firewall no GCE, e apos isso roda diversos commandos dentro da maquina para copiar os arquivos locais. Depois de copiado é executado o arquivo **vm/ setup\_vm.sh** para fazer o Docker funcionar corretamente e por fim os docker’s são buildados e subidos na mesma rede do Docker para haver comunicação entre eles. No final o script retorna o IP publico da máquina.

## Parte 6 - Transformação em Solução cognitiva

Aqui decidimos criar um front-end com React, está tudo contido dentro da pasta front. Fizemos uma tela de formulário com todas as features onde o usuário seleciona o tipo de classificação e depois de enviar tem o retorno do model\_manager interpretado na tela, dizendo que há propensão de inadimplencia ou não.

As instruções para rodar o front-end estão no arquivo **front/README.md. É necessario node.js instalado na máquina.**

Publicamos ele no firebase, que é um host gratuito do Google para conteudo estatico.

Aqui só tivemos um problema. Não conseguimos fazer o IP da VM funcionar com HTTPS, e como o firebase é HTTPS é preciso seguir os passos do arquivo **web-tutorial.docx** na raiz do repositório para poder fazer a chamada das predições. Acredito que iria precisar de instalação de algum certificado no Firewall, mas confesso não soubemos exatamente como fazer, mesmo com o Docker exposto na porta 443.

O deploy do front-end foi automatizado dentro do GH Actions dentro do proprio repositório.

[URL para o front-end publicado](https://prediction-ia.web.app/)

