FRAMEWORK PARA SERVIÇOS DE APRENDIZADO DE MÁQUINA

PEDRO HOLLANDA BOUEKE

Projeto de Graduação apresentado ao Curso de Engenharia de Computação e Informação da Escola Politécnica, Universidade Federal do Rio de Janeiro, como parte dos requisitos necessários à obtenção do título de Engenheiro.

Orientador: Flávio Luis de Mello

Rio de Janeiro

Março de 2019

FRAMEWORK PARA SERVIÇOS DE APRENDIZADO DE MÁQUINA

PEDRO HOLLANDA BOUEKE

PROJETO DE GRADUAÇÃO SUBMETIDO AO CORPO DOCENTE DO CURSO DE ENGENHARIA DE COMPUTAÇÃO E INFORMAÇÃO DA ESCOLA POLITÉCNICA DA UNIVERSIDADE FEDERAL DO RIO DE JANEIRO COMO PARTE DOS REQUISITOS NECESSÁRIOS PARA A OBTENÇÃO DO GRAU DE ENGENHEIRO DE COMPUTAÇÃO E INFORMAÇÃO

Autor:

PEDRO HOLLANDA BOUEKE

Orientador:

Flávio Luis de Mello, DSc

Examinador:

Diego Leonel Cadette Dutra, DSc

Examinador:

Manoel Villas Boas Junior, MSc

Rio de Janeiro

Março de 2019

Declaração de Autoria e de Direitos

Eu, *Pedro Hollanda Boueke* CPF *108.243.966.50*, autor da monografia *Framework para Serviços de Aprendizado de Máquina*, subscrevo para os devidos fins, as seguintes informações:

1. O autor declara que o trabalho apresentado na disciplina de Projeto de Graduação da Escola Politécnica da UFRJ é de sua autoria, sendo original em forma e conteúdo.

2. Excetuam-se do item 1. eventuais transcrições de texto, figuras, tabelas, conceitos e idéias, que identifiquem claramente a fonte original, explicitando as autorizações obtidas dos respectivos proprietários, quando necessárias.

3. O autor permite que a UFRJ, por um prazo indeterminado, efetue em qualquer mídia de divulgação, a publicação do trabalho acadêmico em sua totalidade, ou em parte. Essa autorização não envolve ônus de qualquer natureza à UFRJ, ou aos seus representantes.

4. O autor pode, excepcionalmente, encaminhar à Comissão de Projeto de Graduação, a não divulgação do material, por um prazo máximo de 01 (um) ano, improrrogável, a contar da data de defesa, desde que o pedido seja justificado, e solicitado antecipadamente, por escrito, à Congregação da Escola Politécnica.

5. O autor declara, ainda, ter a capacidade jurídica para a prática do presente ato, assim como ter conhecimento do teor da presente Declaração, estando ciente das sanções e punições legais, no que tange a cópia parcial, ou total, de obra intelectual, o que se configura como violação do direito autoral previsto no Código Penal Brasileiro no art.184 e art.299, bem como na Lei 9.610.

6. O autor é o único responsável pelo conteúdo apresentado nos trabalhos acadêmicos publicados, não cabendo à UFRJ, aos seus representantes, ou ao(s) orientador(es), qualquer responsabilização/ indenização nesse sentido.

7. Por ser verdade, firmo a presente declaração.

_____

Pedro Hollanda Boueke

UNIVERSIDADE FEDERAL DO RIO DE JANEIRO
Escola Politécnica - Departamento de Eletrônica e de Computação
Centro de Tecnologia, bloco H, sala H-217, Cidade Universitária
Rio de Janeiro - RJ CEP 21949-900

DEDICATÓRIA

Dedico todo o esforço relacionado ao trabalho aqui exposto, bem como todos os meus anos de estudo e formação, à minha mãe, que fez de tudo em sua vida para me fornecer a melhor educação que pôde, mas se foi antes de presenciar esse momento. Serei eternamente grato pelo seu amor.

AGRADECIMENTO

Agradeço a todas as pessoas presentes durante a minha jornada acadêmica, em especial ao professor Flávio Luis de Mello. Agradeço também ao povo brasileiro, que contribuiu de forma significativa à minha formação e estada nesta Universidade. Este projeto é uma pequena forma de retribuir o investimento e confiança em mim depositados.

RESUMO

O trabalho aqui exposto tem como finalidade propor uma plataforma para execução de modelos de aprendizado de máquina de forma escalonada e de fácil gerência, se tratando de um sistema configurável e acessível a usuários por meio de uma interface *web*. O trabalho aborda a arquitetura desse sistema e uma implementação e implantação, apresentando suas componentes, funcionamento e um protótipo para sua validação. Também são realizadas conclusões em cima do trabalho desenvolvido e são apresentados trabalhos futuros em cima da proposta exposta.

Palavras-Chave: aprendizado de máquina, Django, agendador de tarefas, aplicação *web*.

ABSTRACT

The work here shown has the aim of proposing a framework for the execution of machine learning models in a easily managed and scalable manner, constituted by a configurable system accessible by users by the way of a web interface. The work approaches this system's architecture, an implementation and a deployment, going through its components, its functioning and a prototype for validation of the project. It also elaborates conclusions over the developed work and apresents future work propositions.

Key-words: machine learning, Django, job sheduler, web application.

SIGLAS

AWS - Amazon Web Services

HTML - Hypertext Markup Language

MTV - Model Template View

ODBC - Open Database Connectivity

UFRJ - Universidade Federal do Rio de Janeiro

Conteúdo

Lista de Figuras

Lista de Tabelas

Capítulo 1

Introdução

1.1 Tema

O trabalho apresenta uma solução para uma das questões comumente negligenciadas dentro da área de tecnologias de Aprendizado de Máquina: a implantação e uso de modelos em ambientes não acadêmicos. A proposta executada se reserva à criação de uma plataforma para gerência e execução de modelos de Aprendizado de Máquina de forma a automatizar os processos relacionados ao uso de tais modelos, da coleta dos dados a serem analisados, passando por sua execução até a entrega dos resultados.

1.2 Delimitação

O trabalho apresentado se destina a atender as demandas de agentes de todas as esferas do círculo de usuários de modelos de Aprendizado de Máquina. Enquanto necessárias execuções recorrentes de modelos, a plataforma desenvolvida apresenta uma solução compatível aos principais contextos em que tais modelos se apresentam. São exceções notáveis a esse conjunto de usuários parte dos desenvolvedores de modelos de Aprendizado de Máquina que, possivelmente, não estejam interessados em praticar a execução de seus modelos em ambientes de produção. Também se excluem aqueles que não possuem requisitos técnicos para a execução dos componentes necessários ao sistema, como usuários do sistema operacional Windows, usuários de bancos de dados incompatíveis com a linguagem SQL e usuários impossibilidatos de

fazer uso da linguagem de programação *Python 3*.

1.3 Justificativa

Enquanto que algoritmos e modelos matemáticos são apresentados frequentemente pelo ambiente acadêmico como soluções aproximadas de problemas reais e complexos condizentes ao que se diz respeito da área de Aprendizado de Máquina, a sua implantação em ambientes não acadêmicos ou de produção poucas vezes é abordada. Dado esse cenário, nota-se que existe espaço e demanda para o estudo e desenvolvimento de novas metodologias e plataformas que ambicionem soluções para as dificuldades envolvidas em trazer um modelo de Aprendizado de Máquina para um ambiente não acadêmico, dentre as quais destacam-se: gerência de execuções e processamento, gerência de arquivos e modelos, visualização e acompanhamento de resultados.

1.4 Objetivos

Objetiva-se o desenvolvimento de um *framework* que permita o agendamento e acompanhamento de execuções de modelos de Aprendizado de Máquina. Uma execução caracteriza-se pela instanciação de um modelo, coleta dos dados a serem processados, inferência do modelo sobre os dados coletados e persistência tanto dos resultados quanto das informações relativas à execução. Por acompanhamento, refere-se à possibilidade de usuários do *framework* visualizarem os resultados das execuções, bem como de configurarem todos os aspectos da execução, como o escalonamento, o modelo e as bases de persistência.

Esse objetivo será alcançado com a implementação bem sucedida de um *framework* que permita que todos os detalhes descritos sejam executados, fazendo uso de um sistema de agendamento de tarefas para agendamento de execuções do modelo; uma plataforma de desenvolvimento de aplicações *Web* para permitir o acompanhamento e gerência das execuções; bases de dados que contenham as informações dos usuários do sistema, registros detalhando as etapas das execuções do modelo e os dados a serem processados. Dessa forma, serão objetivos parciais para se alcançar o fim

desejado:

1. Desenvolvimento de uma aplicação *Web* caracterizada por:
   (a) Permitir agendamento de execuções de programas para processamento de modelos de Aprendizado de Máquina.
   (b) Permitir a visualização de resultados e eventos gerados pelos processos agendados.
   (c) Permitir a execução de modelos de Aprendizado de Máquina, com a visualização dos resultados em tempo real.
   (d) Se comunicar com bases de dados externas para coleta e persistência de dados.
   (e) Ser reconfigurável a fim de atender ambientes diversos.
2. Desenvolvimento de uma base de dados relacionais para armazenamento de eventos do *framework*.
3. Desenvolvimento de um programa para execução de modelos de Aprendizado de Máquina.

1.5 Metodologia

Atendendo às expectativas de um projeto de *Software*, iniciou-se o projeto a partir de diagramas e modelos de Engenharia de Software que definem o funcionamento do sistema projetado.

Com o planejamento teórico concluído, foram definidas as ferramentas de desenvolvimento do sistema. Para controle de versão foi utilizada a ferramenta *Git*, com a manutenção de um repositório para todo o código produzido. Como linguagem de desenvolvimento, foi escolhida a linguagem *Python 3*, que se destaca por ser uma das principais linguagens de programação dentro da comunidade de praticantes de tecnologias de Aprendizado de Máquina [1]. Quanto ao desenvolvimento da aplicação *Web*, foi selecionada a plataforma *Django*, que é caracterizada por depender da mesma linguagem de programação que a escolhida para o *framework* e possuir um

sistema de autenticação de usuários embutido. Por fim, quanto aos bancos de dados, foram selecionados: o *SQLite3* para controle de acesso dos usuários, em uma cópia local à aplicação *Web*, o *MySQL* para armazenamento dos eventos produzidos pelo *framework* e, para o banco de coleta e persistência dos dados necessários aos modelos de Aprendizado de Máquina, um banco qualquer que possua compatibilidade com o conector genérico *ODBC*, permitindo consultados *SQL* genéricas.

O desenvolvimento do projeto e artefatos de software relacionados se deu de maneira incremental ao longo de um período de pouco mais de quatro meses, sendo guiado por reuniões e revisões frequentes entre os colaboradores envolvidos para discussão de detalhes técnicos. A validação dos resultados foi feita por meio de inspeção dos artefatos produzidos.

1.6 Descrição

No segundo capítulo será abordada a fundamentação teórica do trabalho desenvolvido, com a elaboração em cima das funções e histórico das ferramentas desenvolvidas. Serão abordados os temas: sistemas de aprendizado de máquina em ambiente de produção, ferramentas para agendamento de tarefas e desenvolvimento de software na plataforma *Django*.

A proposta de plataforma, sua arquitetura e implementação serão abordados no terceiro capítulo. O quarto capítulo se dedicará à conclusão e trabalhos futuros.

Capítulo 2

Fundamentção Teórica

2.1 Sistemas de Aprendizado de Máquina em Ambiente de Produção

Sistemas de Aprendizado de Máquina são sistemas que combinam algoritmos e modelos matemáticos da área de Aprendizado de Máquina em soluções que visam a implantação do uso de tais algoritmos e modelos em ambientes reais. Aprendizado de máquina, por sua vez, se refere a área de estudos voltata ao desenvolvimento e compreensão de modelos matemáticos caracterizados pelo aprimoramento de seus resultados por meio da ingestão de dados de treino, de forma que esses modelos possam realizar predições e decisões sem que sejam explicitamente programados para o fazerem, como aponta Christopher Bishop [2].

Com o surgimento de equipamentos de alto poder computacional a preços acessíveis, o campo de Aprendizado de Máquina foi capaz de ser remodelado a partir de princípios e modelos que anteriormente não eram praticáveis por conta das limitações técnicas da época. Com essa mudança, o campo se tornou um foco para o surgimento de novas tecnologias que se apresentam, em grande parte, como inovações da academia e de núcleos de pesquisa. O surgimento de novos paradigmas no campo não se viu acompanhado, em mesma escala, pelo surgimento de novos sistemas e metodologias de implantação das novas tecnologias em ambientes não acadêmicos.

Enquanto que modelos tomadores de decisões dominam a área, sistemas que permitiriam a implantação de tais modelos em ambientes de produção se encontram pouco difundidos. Atualmente, os principais sistemas desse tipo se encontram hospedados em soluções prontas por provedores de infra-estrutura gerenciada, como é o caso do *Amazon SageMaker* e do *Machine Learning Studio*, soluções pagas providenciadas pela *AWS* e pela *Azure*, respectivamente. Em contrapartida, soluções abertas e gratuitas ainda dependem de um grande trabalho de desenvolvimento por parte dos responsáveis pela implantação, requerendo um esforço similar ao desenvolvido no trabalho aqui exposto.

2.2 Agendadores de Tarefas

Para a implantação de um sistema de *stream* de dados para processamento constante, é necessária uma forma para a inicialização da execução de programas e tarefas correlatas ao trabalho sendo executado. Nesse sentido, considera-se úteis ferramentas agendadoras de tarefas, ou *job shedulers*. Essas são ferramentas responsáveis pela execução agendada de programas, viabilizando metas de frequencia de processamento e execução.

2.2.1 Slurm

Uma dessas ferramentas é o programa conhecido por *slurm* [3], um *workload manager* capaz de realizar o agendamento de tarefas. Essa é uma das principais ferramentas de código aberto utilizada em sistemas *Unix-like*, capaz de funcionar em *clusters* computacionais de forma distribuída ou baseada em apenas um instância. O Slurm não requer modificações no *kernel* do sistema operacional, sendo relativamente auto contido por conta disso.

As três principais funções do *slurm* são a alocação de recursos de maneira exclusiva e não exclusiva a usuários por um período de tempo; prover um *framework* para iniciar, executar e monitorar tarefas paralelas em um conjunto de nós computacionais; arbitrar a contenção de recursos gerenciando filas de trabalhos pendentes. Sua arquitetura consiste de um daemon rodando em cada nó e um daemon central no

nó principal. Os daemons provém comunicação hierárquica tolerante a falhas.

Os nós da rede se dividem em partições, cada uma sendo uma fila de tarefas, cada tarefa é subdividida em subtarefas, que podem ser processadas pela partição em paralelo. Uma vez alocada a um conjunto de nós, o usuário é capaz de iniciar o trabalho em paralelo no forma de subtarefas em qualquer configuração dentro da alocação de nós.

2.2.2 Crontab

Outra ferramenta dessa categoria, a utilizada nesse trabalho, é o *crontab* [4]. O nome é uma abreviação de *cron table*, pois se refere ao arquivo *cron file* utilizado internamente para agendamento e execução de tarefas. Esse programa consiste de um arquivo no sistema operacional que periodicamente é lido pelo *cron* daemon, fazendo uso de expressões *CRON*, e suas linhas são interpretadas como entradas de uma tabela contendo em uma coluna a expressão que define sua agenda de execução e em outra coluna o comando a ser executado como um programa de linha de comando. Seu diferencial está no fato de ser extremamente simples e possuir suporte para manipulação por meio de diversas linguagens de programação, incluindo *Python 3*, a linguagem escolhida para o desenvolvimento do sistema aqui detalhado.

2.3 Desenvolvimento com Django

O *Django* [5] é um *framework* de desenvolvimento de aplicações e serviços *web* disponível nas linguagens de programação *Python 2 e 3*, gratuito e de código aberto. Sua estrutura interna é baseada no paradigma MTV, particularmente caracterizada por uma forte e nativa integração com bancos de dados a partir de uma interface própria. Todos os componentes de uma aplicação *web* são reproduzidos por meio de convenções próprias ao *framework*, baseados na reusabilidade de código e orientado à integração nativa com bancos de dados diversos.

O desenvolvimento com o *framework* é marcado pela pouca quantidade de código produzido e pelas diversas interfaces de programação providas pela plataforma.

Também se destacam a a integração nativa com um sistema de administração embutido nas aplicações produzidas que fornecem, sem custos adicionais de desenvolvimento, uma aplicação e interface de autenticação de usuários também utilizável em todas as camadas de serviço contruídas paralelamente no mesmo ambiente.

O *framework* disponibiliza ao desenvolvedor interfaces próprias para o acesso a bancos de dados, templates para renderização de HTML, classes extensíveis para programação orientada a objetos, encaminhamento de requisições configurável, dentre outras facilidades.

2.4 Amazon AWS

A *Amazon Web Services*, ou AWS, é a maior provedora de infraestrutura em nuvem do mercado. A empresa provê serviços para websites, aplicações cliente servidor, banco de dados, análise de dados, redes, dispositivos móveis, ferramentas de gerenciamento, armazenamento de dados, aluguel de servidores entre outros [6]. Seus serviços podem ser acessados em seu portal *web* e por meio de suas *API*s e *SDK*s.

A Amazon está disponível em diversas localidades, operando *datacenters* ao redor do mundo. Cada local é composto por uma região e zonas de disponibilidade, onde uma região é uma zona geográfica totalmente independente das demais, contendo diversas zonas de disponibilidade também isoladas entre si. A AWS adota, para a maioria de seus serviços, o plano de pagamento *pay-as-you-go*, fazendo necessário apenas o pagamento pelos recusrsos consumidos.

Capítulo 3

Proposta de Plataforma

3.1 Arquitetura

O projeto desenvolvido se trata de uma aplicação *Django* que conta com três interfaces, uma para execução de um modelo preditivo em ocorrências controladas, outra para monitoramento dos serviços associados e outra para a visualização dos resultados das predições, chamadas, respectivamente, de *Showroom*, *Monitor* e *Disparo*. Essas três interfaces são controladas pela aplicação por meio de três classes, vistas na figura 3.1. Os componentes da aplicação podem ser visualizados na figura 3.2, onde se observa a existência de um arquivo *config.yaml*, responsável por agregar todos os parâmetros e configurações da aplicação.

Além da aplicação, existem as componentes *daemon.py* e *datalake.py* que são responsáveis, respectivamente, pela predição de resultados e conexão com o *datalake* onde estão os dados a sofrerem predições. O escalonamento de predições é gerenciado pelo *crontab*, que, por sua vez, é gerenciado pela aplicação.

3.2 Implementação

No modelo seguido pelo *framework Django*, o modelo *MTV*, todas as páginas e interfaces da aplicação são renderizadas a partir de templates HTML preenchidos pelo controlador da aplicação, que no caso se trata do arquivo *views.py*. Os templates HTML se encontram dentro do diretório *templates*. Os modelos se encontram no arquivo *models.py*, fazendo uso direto das classes de modelo do *framework*. As rotas

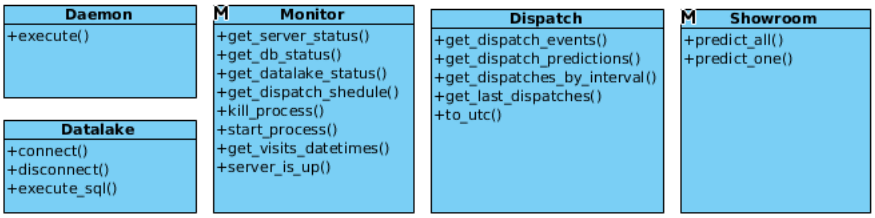


Figura 3.1: Diagrama de classes da aplicação.

da aplicação se encontram no arquivo *urls.py*. As componentes *monitor.py*, *showroom.py*, *dispatch.py*, *datalake.py* e *daemon.py* se encontram no diretório *scripts*.

3.2.1 Configuração

Toda a configuração da aplicação é feita a partir de dois arquivos, o *settings.py* e o *config.yaml*. No primeiro são configuradas as propriedades básicas de uma aplicação *Django*, se tratando de seu arquivo de configuração padrão. No segundo, estão as configurações específicas a essa aplicação. O caminho ao arquivo *config.yaml* é escrito como a variável *CONFIG_FILE*, dentro do arquivo *settings.py*, para que as configurações possam ser carregadas na aplicação durante sua inicialização.

O arquivo *config.yaml* contém todos os parâmetros necessários para o funcionamento da aplicação. Os parâmetros podem ser vistos na tabela 3.1.

3.2.2 View

Em aplicações que fazem uso do *framework Django*, a componente *view.py*, vista na figura 3.1, é responsável pelo controle e recebimento das requisições à aplicação. Essa componente é vista como um pacote da linguagem *Python* e contém as funções responsáveis por enviar o HTML a ser visualizado pelo usuário e responder todas as chamadas assíncronas requisitadas pelas interfaces.

É por meio dessa componente que o controle de acesso às páginas da aplicação é realizado, com o uso de decoradores especiais que restrigem o acesso a certas

Tabela 3.1: Parâmetros de configuração contidos no arquivo *config.yaml*.

Parâmetro	Descrição
monitor.controllerserver.address	Endereço (ip ou nome de domínio) para o servidor monitorado
monitor.controller.server.machanism	Mecanismo de conexão com o servidor monitorado (plain ou ssl para ips, dns para nomes de domínio)
monitor.controller.server.port	Porta de conexão com o servidor monitorado(apenas para ips)
monitor.view.refresh_rate	Segundos entre atualizações das informaçoes na view monitor
dispatch.view.filter.default_offset	Diferença temporal padrão para o filtro de disparos
showroom.controller.max_min_model_path	Arquivo de modelo usado para pré-tratamento das amostras
showroom.controller.prediction_model_path	Arquivo de modelo usado para a predição dos resultados
showroom.controller.samples_path	Arquivo contendo amostras em formato csv
showroom.controller.number_of_columns	Número de colunas entre as primeiras do arquivo a serem mostradas na view
mysql_events_db.name	Nome do banco de eventos
mysql_events_db.user	Nome do usuário do banco de dados
mysql_events_db.password	Senha do usuário do banco de enventos
mysql_events_db.host	Endereço do banco de eventos
mysql_events_db.port	Porta de acesso ao banco de eventos
datalake.connection_string	Cadeia de conexão ODBC para o datalake
datalake.table	Nome de uma tabela (qualquer tabela existente) usada para verificar a conexão com o banco
daemon.user	Nome do usuário do SO
daemon.comment	Comentário usado como id único para a tarefa cron
daemon.cron	Expressão CRON para a execução do comando
daemon.venv	Caminho para a ativação do virtual environment (opcional, usado caso os pacotes requeridos pelo script daemon não estejam disponíveis globalmente)
daemon.command	Comando a ser executado pelo daemon cron
daemon.log_file	Arquivo log de depuração - deixe em branco se não estiver debugando, o log será ignorado

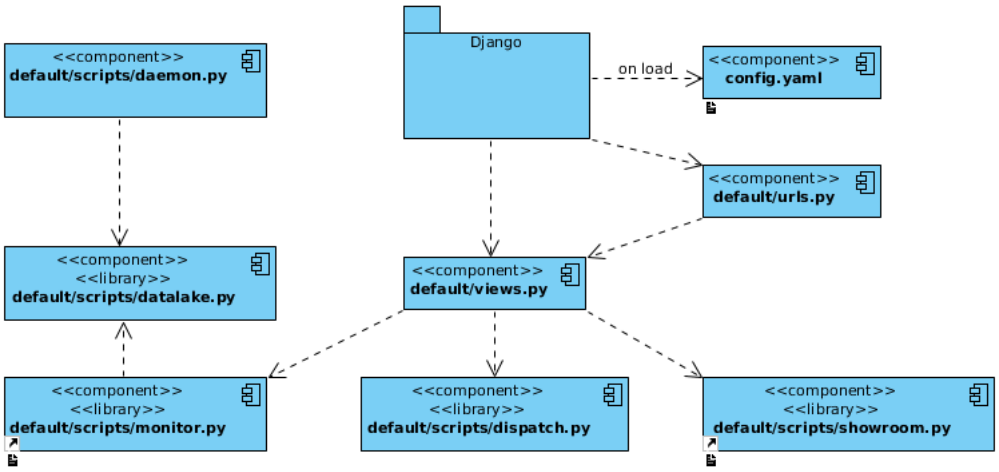


Figura 3.2: Diagrama de componentes da aplicação.

Tabela 3.2: Possíveis grupos de usuários

Grupo	Permissões
Admin	Acesso às views Monitor, Disparo e Showroom
DispatchUser	Acesso à view Disparo
ShowroomUser	Acesso à view Showroom

chamadas a depender do tipo de usuário que está acessando a aplicação. A tabela 3.2 mostra os possíveis grupos atribuídos a usuários, com suas respectivas permissões. A criação dos usuários e atribuição de grupos está detalhada na seção 3.3.

3.2.3 Datalake

A componente *datalake.py* é responsável pela conexão com o *datalake* que contém os dados sobre os quais serão feitas as inferências do modelo carregado na componente *daemon.py*. A tabela 3.3 contém a descrição de os métodos dessa componente. Os métodos refletem essencialmente o uso básico de um conector ODBC, requerendo uma cadeia de conexão adequada ao serviço empregado pelo *datalake*.

Tabela 3.3: Métodos do componente *datalake.py*.

Método	Descrição
connect	Inicializa a conexão com o banco de dados usando a cadeia de conexão provida
disconnect	Fecha a conexão com o banco de dados
execute	Executa uma query SQL, retornando seu resultado

Tabela 3.4: Métodos do componente *showroom.py*.

Método	Descrição
predict_all	Retorna o resultado das predições para todos os casos (linhas) presentes no arquivo indicado pelo caminho 'samples_path' contido no arquivo de configuração
predict_one	Faz o mesmo que predict_all, porém apenas para a linha 'id' do arquivo 'samples_path'

3.2.4 Showroom

A *view Showroom* é caracterizada pela existência de casos sobre os quais serão inferidas classificações a partir do modelo de aprendizado de máquina carregado na aplicação. Toda execução gerada a partir dessa *view* segue os passos vistos no diagrama de sequência apresentado na figura 3.4. Os dados para a predição são coletados, o evento é registrado na tabela de log de eventos, o resultado é computado e a *view* é atualizada com os resultados da predição. A tabela de eventos, por sua vez, é populada por eventos relacionados a cada execução, representados pelas colunas da entidade *ShowroomDispatch* na figura 3.3. Os dois métodos da componente *showroom.py* podem ser vistos na tabela 3.4.

Os dois métodos representam dois casos de uso possíveis. O primeiro se trata de quando a *view* é inicializada e todos os casos são inferidos, fazendo uso do método *predict_all*, o segundo se trata da inferência de apenas um dos casos assincronamente, fazendo uso do método *predict_one*.

3.2.5 Daemon

A componente *daemon.py* se trata do *script Python* responsável pelos disparos cuja execução é gerenciada pela aplicação por meio do arquivo *crontab*. A componente contém apenas um método, o *execute*, que é responsável por executar a predição de todos os casos disponíveis no *datalake*. Sua execução segue os passos do

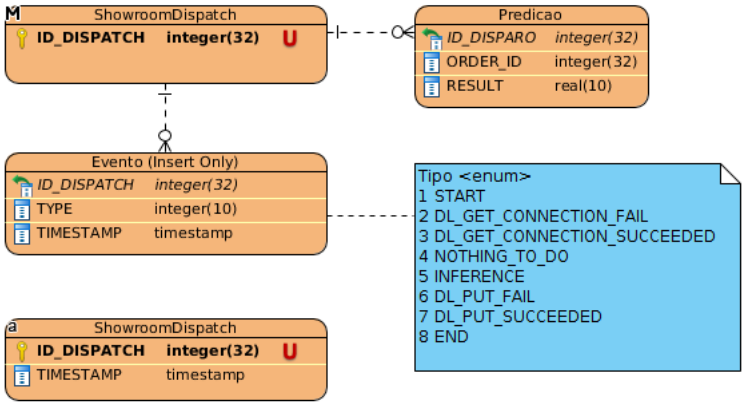


Figura 3.3: Diagrama Entidade-Relacionamento do banco.

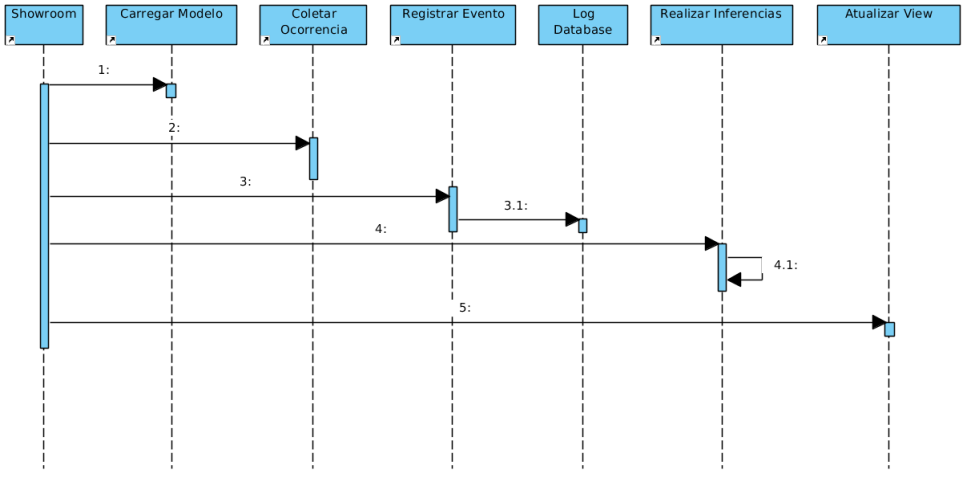


Figura 3.4: Diagrama de sequência da *view Showroom*.

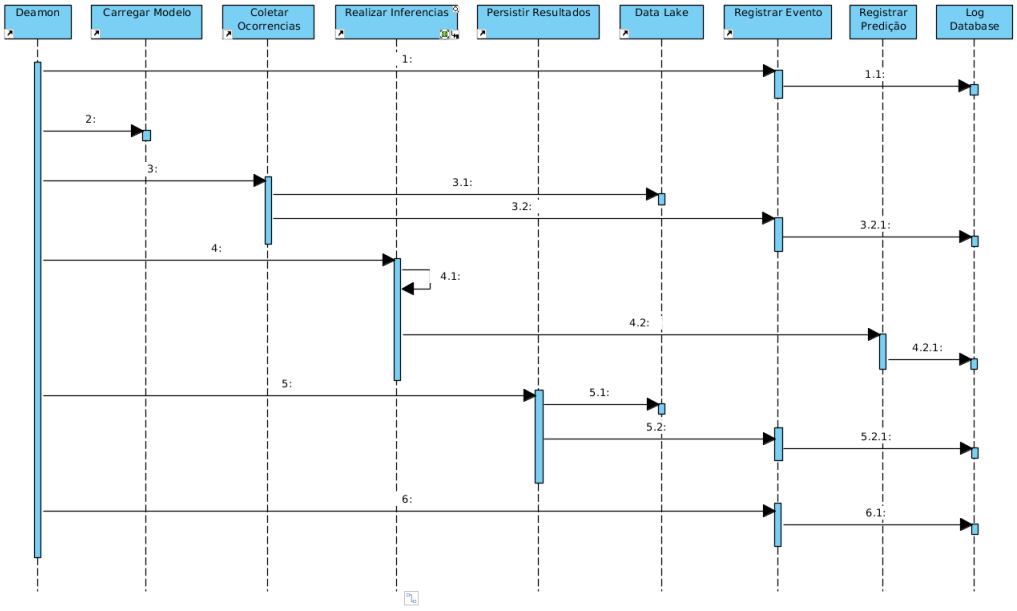


Figura 3.5: Diagrama de sequência da *view Showroom*.

diagrama de sequência visto na figura 3.5. Inicialmente um novo disparo é inicializado e o evento de inicialização é armazenado no banco da aplicação, o modelo de inferência é carregado para memória da aplicação, as ocorrências a sofrerem o processo preditivo são coletadas do *datalake* e o evento de coleta é registrado. Após isso as predições são realizadas e seus resultados são registrados no banco da aplicação. Posteriormente, os resultados são salvos no *datalake* e o evento de inserção é salvo. Por fim, o evento de finalização é armazenado.

O disparo, os eventos e os resultados das predições das ocorrências são armazenados no banco da aplicação fazendo uso das entidades observadas na figura 3.3.

3.2.6 Disparo

A *view Disparo* é responsável por permitir a visualização dos disparos executados, onde cada disparo é dado pela execução do script de inferência. Nela é possível visualizar todos os disparos executados pela componente *daemon.py* a partir de um filtro temporal. Todos os resultados de todas as predições dentro do intervalo temporal selecionado são visualizados em tabelas, uma tabela para cada disparo. Os

Tabela 3.5: Métodos do componente *dispatch.py*.

Método	Descrição
get_dispatch_events	Retorna todos os eventos relacionados ao disparo com id igual ao fornecido
get_dispatch_predictions	Retorna todos os resultados de predições relacionados ao disparo com id igual ao fornecido
get_dispatches_by_interval	Retorna todos os ids dos disparos que ocorreram dentro de um intervalo temporal
to_utc	Converte uma cadeia de caracteres no formato de hora e data yyyy-MM-ddThh:mm:ss-hh:mm para um objeto datetime na zona UTC
get_last_dispatches	Retorna todos os eventos e resultados de predições para todos os disparos que ocorreram entre as datas start e end, agrupados pelos ids dos disparos

métodos dessa componente podem ser vistos na tabela 3.5, referente à componente dispatch.py.

3.2.7 Monitor

A *view Monitor* é responsável por monitorar: o estado de um *host*, o estado do banco de dados, as datas dos último e próximo disparos e o estado do escalonamento da componente *daemon.py* a partir do arquivo *crontab*. Em intervalos periódicos de tempo a *view* é atualizada trazendo informações recentes para a interface, o que é feito por meio de uma requisição assíncrona chamada pela interface. A cada execução da atualização, os métodos *get_datalake_status*, *get_process_status*, *get_dispatch_shedule*, *get_server_status* e *get_db_status* são executados pela componente *view.py* e os resultados são enviados como resposta à requisição assíncrona, coforme visto no trecho de código a seguir. Os métodos dessa componente podem ser vistos na tabela 3.6.

```
def get_monitor_status(request):
    monitor = modules.Monitor()
    data = {
        'server': monitor.get_server_status(
                        settings.FRAMEWORK_CONFIG \
                        ['monitor']['controller']['server']),
        'datalake': monitor.get_datalake_status(
                settings.FRAMEWORK_CONFIG['datalake']),
        'process': monitor.get_process_status(
                        settings.FRAMEWORK_CONFIG),
        'visits': monitor.get_visits_datetimes(
                        settings.FRAMEWORK_CONFIG),
        'database': monitor.get_db_status(
                        settings.FRAMEWORK_CONFIG["mysql_events_db"])
    }
    return JsonResponse(data)
```

Para a verificação do estado de um servidor, uma conexão socket é criada em uma porta aberta qualquer do host. A conexão é tida como *offline* se ocorrer um timeout, sendo 10 segundos o intervalo de tempo para que isso ocorra. O host informado pode ser tanto como um endereço IPV4 quanto um endereço IPV6. Ainda é possível monitorar servidores a partir de seu nome de domínio, mudando o mecanismo configurado de *plain* ou *ssl* para *dns*, conforme visto no parâmetro *monitor.controller.server.mechanism* da tabela 3.1. O método que realiza essa verificação pode ser visto a seguir.

```
def server_is_up(self,server):
    srv, mechanism, port = server
    try:
        if mechanism == 'plain':
            socket.create_connection(
                    (srv, port), timeout=10)
        elif mechanism == 'ssl':
            ssl.wrap_socket(
                    socket.create_connection(
                            ("%s" % srv, port), timeout=10))
        elif mechanism == 'dns':
            host = socket.gethostbyname(srv)
            socket.create_connection(
                    (host, port), timeout=10)
        else:
            raise ValueError("Invalid mechanism")
        return ("online","")
    except socket.timeout:
        return ("offline","connection timeout")
    except Exception as err:
        return ("error", err.__str__())
```

A verificação do estado do *datalake* é feita por meio de uma conexão, usando a cadeia de conexão adequada ao banco a ser conectado. Após a conexão ser estabelecida, é realizada uma consulta simples em uma tabela qualquer pertencente ao banco, sem que nada seja retornado, com o intuito de que todos os passos da conexão sejam testados. Caso ocorra algum problema em algum momento durante a tentativa de conexão, esse problema é reportado. Essa lógica pode ser vista no trecho de código a seguir.

```
def get_datalake_status(self, datalake_settings):
    try:
        if self.datalake is None:
            self.datalake =
                dl.Datalake(
                        datalake_settings["connection_string"])
        table = datalake_settings["table"]

        self.datalake.execute(
                'SELECT * FROM {} LIMIT 1'.format(table))
    except pyodbc.Error as err:
        print(err)
            return { 'status': "error", 'error': str(err)}
        return { 'status': "online", 'error': ''}
```

O último disparo e o estado do próximo disparo são monitorados fazendo uso da biblioteca *crontab* para *Python 3*. Tanto a agenda de execuções quanto o comando a ser executado pelo *daemon cron* são obtidos por meio do uso dessa biblioteca. A verificação do estado do comando é feita como mostrado no trecho a seguir.

```
def get_process_status(self, config):
    try:
        expr = config['daemon']['cron']
        sheduler = CronTab(user=config['daemon']['user'])
        for job in sheduler:
            if job.comment == config['daemon']['comment']:
                if not job.is_valid():
                    return {"status": "invalid", "cron": expr, "error": ""}
                if not job.is_enabled():
                    return {"status": "disabled", "cron": expr, "error": ""}
                return {"status": "active", "cron": expr, "error": ""}
        return {"status": "inactive", "cron": expr, "error": ""}
    except Exception as e:
        return {"status": "error", "cron": expr, "error":str(e)}
```

Além de monitorar os serviços associados ao *framework*, a *view* Monitor também é responsável pelo controle do estado do escalonamento da componente *daemon.py*. Em sua interface, acessível apenas por administradores, é possível adicionar e remover o comando de execução do *script daemon.py* do arquivo *crontab*, efetivamente controlando se a execução ocorrerá ou não conforme previsto pela expressão CRON presente no arquivo de configuração no parâmetro *daemon.cron*, visto na tabela 3.1. Esse controle é feito por meio dos métodos *kill_process* e *start_process*, vistos a seguir.

```
def kill_process(self,config):
    sheduler = CronTab(user=config['daemon']['user'])
    for job in sheduler:
        if job.comment == config['daemon']['comment']:
            sheduler.remove(job)
            sheduler.write()

def start_process(self, config):
    self.kill_process(config)
    cmd = '''{} "{}" '''.format(
        config['daemon']['command'], config['path'])
    venv = config['daemon']['venv']
    if venv is not None and venv != "":
        cmd = "source {}; {}; deactivate".format(venv, cmd)
        cmd = '''"$(command -v bash)" -c '{}' '''.format(cmd)

    sheduler = CronTab(user=config['daemon']['user'])
    job = sheduler.new(command= cmd,
                       comment=config['daemon']['comment'])
    job.parse(config['daemon']['cron'])
```

3.2.8 Urls

A componente *urls.py* é responsável por descrever as rotas da aplicação. Nela, os caminhos que levam às *views* e às chamadas assíncronas estão detalhadas em uma lista, fazendo a associação direta entre as funções da componete *views.py* e as rotas usadas pelo usuário e pela interface da aplicação. As rotas existentes podem ser

Tabela 3.6: Métodos do componente *monitor.py*.

Método	Descrição
get_datalake_status	Retorna o estado da conexão com o datalake e o erro de conexão, se houver
get_process_status	Retorna o estado do comando gerenciado no arquivo crontab
get_dispatch_schedule	Retorna a expressão CRON usada para agendar as execuções do comando gerenciado no arquivo crontab
kill_process	Remove o comando de execução do daemon do arquivo crontab
start_process	Inicia o agendamento do comando gerenciado no arquivo crontab, removendo quaisquer comandos que possuam o mesmo id
get_server_status	Retorna o estado de um host, obtido a partir de conexões criadas via sockets
get_db_status	Se conecta com o banco de dados da aplicação gerenciado pelo Django e retorna o estado da conexão e o erro de conexão, se houver
get_visits_datetimes	Retorna a última e próxima data de execução para o comando gerenciado no arquivo crontab, se ativo. Retorna 'N/A' para ambas caso contrário
server_is_up	Verifica se um host está online usando sockets e retorna seu estado e o erro de conexão, se houver

vistas na tabela 3.7. As figuras 3.6, 3.7 e 3.8 mostram as interfaces acessadas pelas rotas *monitor/*, *dispatch/* e *showroom/*, respectivamente.

3.3 Implantação

3.3.1 Configuração do ambiente

Para configuração do ambiente, o primeiro passo é sua obtenção, que, para o caso da implantação desse projeto, foi obtido por meio do aluguel de um servidor da

Tabela 3.7: Rotas definidas na componente *urls.py*.

Rota	Descrição
redirect/	Rota de redirecionamento após o \textit{login}
monitor/	Rota para acesso à *view* Monitor
dispatch/	Rota para acesso à *view* Disparo
showroom/	Rota para acesso à *view* Showroom
dispatch/update_dispatch/	Rota para requisição assíncrona de atualização da *view* Disparo
showroom/update_showroom/	Rota para requisição assíncrona de atualização da *view* Showroom
showroom/update_showroom_item/	Rota para requisição assíncrona de atualização de um dos items da *view* Showroom
monitor/monitor_status/	Rota para requisição assíncrona de atualização da *view* Monitor
monitor/process_kill/	Rota para requisição assíncrona de remoção do comando no arquivo *crontab*
monitor/process_start/	Rota para requisição assíncrona de inserção do comando no arquivo *crontab*

Figura 3.6: Interface da *view Monitor*.

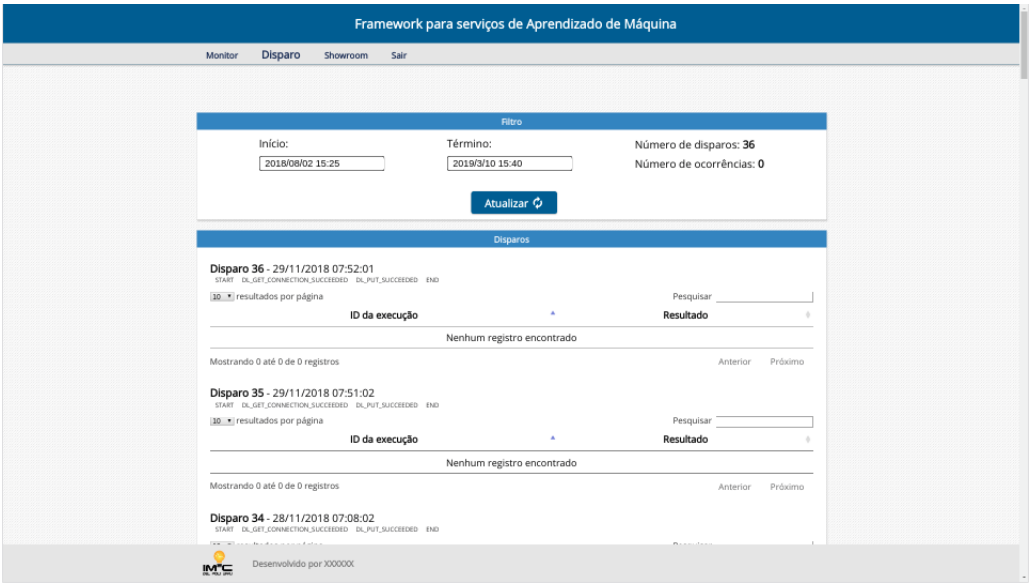


Figura 3.7: Interface da *view Disparo*.

Figura 3.8: Interface da *view Showroom*.

Figura 3.9: Diagrama de implantação

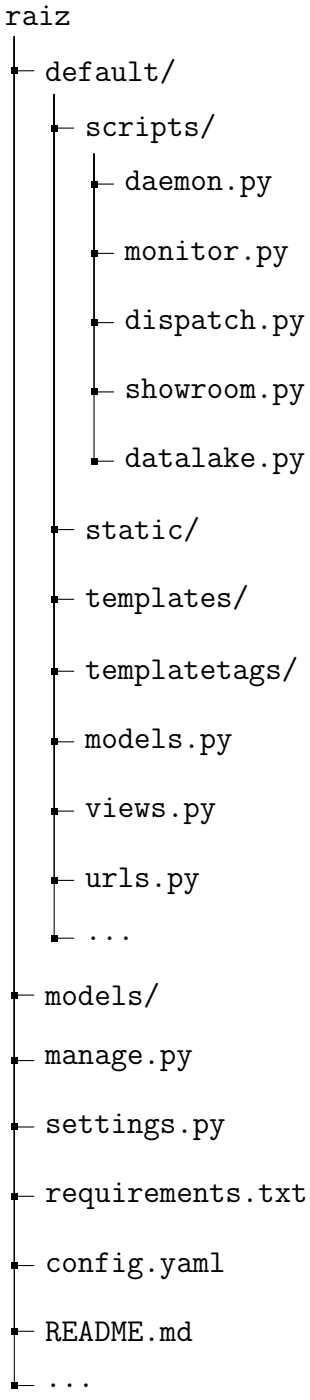
Amazon AWS. O segundo é a obtenção do código fonte, que se encontra hospedado em um repositório do serviço *Github*. Com o uso de uma conta autorizada a ter acesso ao repositório, é preciso realizar o download do projeto. A estrutura de arquivos e diretórios pode ser vista na figura 3.10, onde se encontram destacados todos os diretórios e componentes abordados nesse documento. Posteriormente, é necessário seguir as instruções presentes no arquivo *README.md*, que contém detalhados todos os passos necessários para a configuração do ambiente. Estas instruções são detalhadas a seguir, para uso em um sistema operacional *Linux*.

Inicia-se a configuração a preparação do ambiente *Python 3*, certificando-se de que a linguagem está instalada no servidor junto com o programa utilitário para criação de ambientes virtuais, o *virtualenv*. Com esses passos concluídos, os próximos são inicializar o ambiente virtual, obter o código fonte e instalar as dependências. Esses passos são concluídos com os comandos abaixo.

```
$ # criar o diretório para o ambiente virtual
$ mkdir myenv
$ # obter código fonte
$ git clone https://github.com/pboueke/ml-services-framework.git
$ # criar e inicializar o ambiente virtual
$ python3 -m venv myvenv
$ source myvenv/bin/activate
(myvenv) $ # instalar dependências
(myvenv) $ cd ml-services-framework
(myvenv) $ pip install -r requirements.txt
```

Em seguida, é necessária a instalação do banco de dados MySQL em um servidor acessível ao servidor da aplicação, podendo ser o mesmo servidor. Para isso, siga as instruções de instalação do banco de dados para o seu sistema operacional e, em seguida, instale no servidor da aplicação a biblioteca para conexão ao banco para clientes, a *libmysqlclient-dev*. Uma vez instalado, o próximo passo é configurar o banco de dados para uso da aplicação. Se conecte localmente ao banco com o comando a seguir.

Figura 3.10: Estrutura do sistema de arquivos do projeto.



```
$ mysql -u USERNAME -p
```

Quando conectado, execute os comandos abaixo, substituindo o termo *localhost* pelo endereço apropriado ao servidor da aplicação, *USERNAME* pelo nome de usuário cadastrado no serviço de banco de dados e *PASSWORD* pela senha de acesso do usuário. É necessário que os dados preenchidos aqui sejam os mesmos que os preenchidos nos parâmetros *mysql_events_db.** do arquivo de configuração, conforme visto na tabela 3.1.

```
> CREATE DATABASE MLServicesFramework;
> CREATE USER 'django'@'localhost' IDENTIFIED BY 'PASSWORD';
> GRANT ALL ON MLServicesFramework.* TO 'django'@'localhost';
> FLUSH PRIVILEGES;
```

Com o banco configurado, é necessário inicializar as tabelas e modelos a partir dos arquivos do *framework Django*. Para alcançar esse objetivo, basta a execução dos comandos a seguir, que transformam os modelos em *Python* em tabelas do banco.

```
$ python3 manage.py makemigrations
$ python3 manage.py migrate --database=default
$ python3 manage.py migrate --database=app_db
```

Posteriormente, é preciso criar os usuários do sistema. Para isso, executa-se o comando a seguir para criar um usuário administrador e inicia-se a aplicação para criação dos demais usuários e grupos. Com a aplicação rodando localmente é feito o acesso à interface de administração de grupos do *Django*, *http://127.0.0.1:8000/admin/auth/group/add/*, e são criados os dois grupos, *DispatchUser* e *ShowroomUser*, vistos na tabela 3.2. Usuários podem, então, ser criados pela interface de admi-

nistração de usuários do *Django*, *http://127.0.0.1:8000/admin/auth/user/add/*, e atribuídos aos grupos existentes conforme a necessidade de cada um.

```
$ # criar um usuário administrador
$ python3 manage.py createsuperuser
$ # inicia a aplicação na porta padrão
$ python3 manage.py runserver
```

Para a conexão com o *datalake*, é necessária a instalação no servidor da aplicação do conector ODBC relativo ao tipo de banco de dados usado por ele. Primeiramente instale os pacotes *unixodbc-dev*, *unixodbc-bin* e *unixodbc* e prossiga para a instalação do *driver* relativo ao banco de dados específico. Note que o *datalake* precisa ter uma interface ODBC, ou ele não será compatível com o projeto.

Por fim, é necessária a configuração do arquivo *config.yaml*, como visto na tabela 3.1. É necessária a revisão de todos os caminhos apontados pelo arquivo, assim como dados de conexão aos bancos de dados e servidores monitorados.

3.3.2 Inicialização da aplicação

Para a inicialização do projeto, basta a ativaçãodo ambiente virtual e execução do comando de inicialização, com a especificação da porta desejada, coforme visto abaixo.

```
$ # inicializar o ambiente virtual
$ source myvenv/bin/activate
$ # inicializar a aplicação
(myenv) $ python3 manage.py runserver <seu_ip>:<porta>
```

Capítulo 4

Conclusões

4.1 Conclusão

Conforme observado na proposta de plataforma, a arquitetura e implementação do projeto se revelam suficientes em atender os objetivos propostos. O escalonamento de execuções de modelos de aprendizado de máquina é possível por meio do uso do *crontab*, gerenciado pela aplicação *web*. A visualização dos resultados e eventos é possível por meio da mesma aplicação, que, por sua vez, é configurável de diversas formas, como visto anteriormente.

O uso e manutenção de uma base de dados relacional para persistência dos resultados e eventos do sistema também foi coberto pela implementação e configuração do ambiente, bem como o uso de uma componente responsável pela execução dos modelos de aprendizado de máquina.

Com todos os pontos objetivados alcançados, tem-se o sucesso da implementação da proposta.

4.2 Trabalhos Futuros

No curto prazo, observa-se a possibilidade de ampliar as funcionalidades da aplicação de forma que parte de suas configurações estejam disponíveis aos usuários administradores da aplicação, não dependendo mais apenas do arquivo *config.yaml*. Isso se

daria na forma de uma nova *view* de configurações do sistema, onde parâmetros e os modelos carregados possam ser alterados.

Em uma escala temporal maior, existe a possibilidade da expansão do escopo da aplicação, permitindo que a mesma seja capaz de gerenciar e executar vários modelos de aprendizado de máquina, e não apenas um modelo, fazendo uso da *view* descrita acima para sua configuração. Essa proposta ainda faria uso da ferramenta *crontab* e permitiria que várias aplicações de modelos fossem gerenciadas em uma plataforma única, apresentando as mesmas características da plataforma desenvolvida nesse projeto.

No longo prazo, atingidas as propostas anteriores, tornar-se-ia interessante a criação de uma interface programática para a execução dos modelos de aprendizado de máquina disponíveis na aplicação, por meio de uma *web API*. Essa proposta permitiria que inferências sobre os modelos carregados fossem executadas sob demanda dos usuários, e não apenas por meio do escalonamento de execuções gerenciado a partir do arquivo *crontab*. Essa funcionalidade abriria a possibilidade de que a plataforma fosse usada em mais contextos operacionais.

Bibliografia

[1] THE STACK OVERFLOW NETWORK, "Stack Overflow Annual Developer Survey", [Arquivo de dados], Acessado em 6 de março de 2019, https://insights.stackoverflow.com/survey.

[2] BISHOP, C. M., "Pattern Recognition and Machine Learning", *Springer*, v. ISBN 978-0-387-31073-2, 2006.

[3] "Slurm Workload Manager", [Software], Acessado em 6 de março de 2019, https://slurm.schedmd.com/.

[4] "crontab - schedule periodic background work - Commands and Utilities Reference", [Software], Acessado em 6 de março de 2019, https://pubs.opengroup.org/onlinepubs/9699919799/utilities/crontab.html.

[5] "Django", [Software], Acessado em 6 de março de 2019, https://www.djangoproject.com/, 2018.

[6] "Amazon Web Services", Acessado em 6 de março de 2019, https://aws.amazon.com/pt/products/.