

**MINISTÉRIO DA DEFESA
EXÉRCITO BRASILEIRO
DEPARTAMENTO DE CIÊNCIA E TECNOLOGIA
INSTITUTO MILITAR DE ENGENHARIA
CURSO DE GRADUAÇÃO EM ENGENHARIA DE COMPUTAÇÃO**

**1 Ten LUCAS BRAIDA NAZARETH
1 Ten RODRIGO GOMES LEMOS
LEONARDO LINHARES MUNIZ RIBEIRO**

**DESENVOLVIMENTO DE UMA BIBLIOTECA DE PROVENIÊNCIA DE
DADOS PARA EXPERIMENTOS DE APRENDIZADO DE MÁQUINA
EM C++**

**Rio de Janeiro
2018**

INSTITUTO MILITAR DE ENGENHARIA

**1 Ten LUCAS BRAIDA NAZARETH
1 Ten RODRIGO GOMES LEMOS
LEONARDO LINHARES MUNIZ RIBEIRO**

**DESENVOLVIMENTO DE UMA BIBLIOTECA DE
PROVENIÊNCIA DE DADOS PARA EXPERIMENTOS DE
APRENDIZADO DE MÁQUINA EM C++**

Projeto de Fim de Curso apresentado ao Curso de Graduação em Engenharia de Computação do Instituto Militar de Engenharia, como requisito parcial para a obtenção do título de Engenheiro de Computação.

Orientador: TC Julio Cesar Duarte - D.Sc.

Rio de Janeiro
2018

c2018

INSTITUTO MILITAR DE ENGENHARIA
Praça General Tibúrcio, 80 - Praia Vermelha
Rio de Janeiro - RJ CEP 22290-270

Este exemplar é de propriedade do Instituto Militar de Engenharia, que poderá incluí-lo em base de dados, armazenar em computador, microfilmar ou adotar qualquer forma de arquivamento.

É permitida a menção, reprodução parcial ou integral e a transmissão entre bibliotecas deste trabalho, sem modificação de seu texto, em qualquer meio que esteja ou venha a ser fixado, para pesquisa acadêmica, comentários e citações, desde que sem finalidade comercial e que seja feita a referência bibliográfica completa.

Os conceitos expressos neste trabalho são de responsabilidade do(s) autor(es) e do(s) orientador(es).

005.133 Nazareth, Lucas Braidá
N335.d Desenvolvimento de uma biblioteca de proveniência de dados para experimentos de aprendizado de máquina em C++ / Lucas Braidá Nazareth, Rodrigo Gomes Lemos, Leonardo Linhares Muniz Ribeiro, orientado por Julio Cesar Duarte - Rio de Janeiro: Instituto Militar de Engenharia, 2018.

67p.: il.

Projeto de Fim de Curso (graduação) - Instituto Militar de Engenharia, Rio de Janeiro, 2018.

1. Curso de Graduação em Engenharia de Computação - projeto de fim de curso. 1. Web4MEX. 2. FAMA. 3. JSON. I. Duarte, Julio Cesar. II. Título. III. Instituto Militar de Engenharia.

INSTITUTO MILITAR DE ENGENHARIA

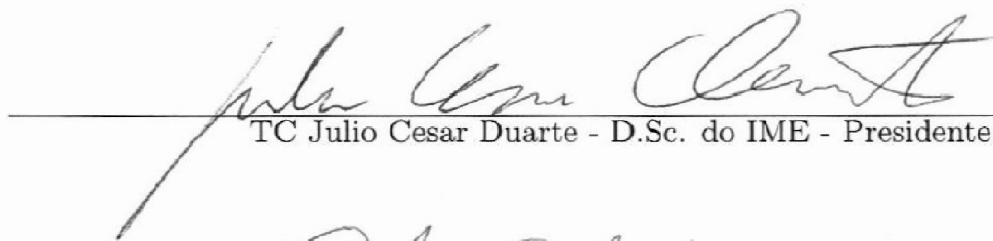
**1 Ten LUCAS BRAIDA NAZARETH
1 Ten RODRIGO GOMES LEMOS
LEONARDO LINHARES MUNIZ RIBEIRO**

**DESENVOLVIMENTO DE UMA BIBLIOTECA DE
PROVENIÊNCIA DE DADOS PARA EXPERIMENTOS DE
APRENDIZADO DE MÁQUINA EM C++**

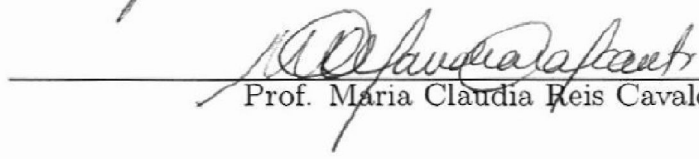
Projeto de Fim de Curso apresentado ao Curso de Graduação em Engenharia de Computação do Instituto Militar de Engenharia, como requisito parcial para a obtenção do título de Engenheiro de Computação.

Orientador: TC Julio Cesar Duarte - D.Sc.

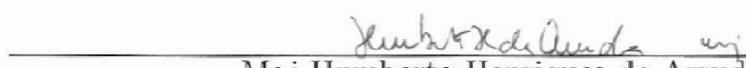
Aprovado em 10 de outubro de 2018 pela seguinte Banca Examinadora:



TC Julio Cesar Duarte - D.Sc. do IME - Presidente



Prof. Maria Claudia Reis Cavalcanti - D.Sc. do IME



Maj Humberto Henriques de Arruda - M.Sc. do IME

Rio de Janeiro
2018

SUMÁRIO

LISTA DE ILUSTRAÇÕES	5
LISTA DE TABELAS	6
LISTA DE SIGLAS	7
1 INTRODUÇÃO	10
1.1 Motivação	10
1.2 Objetivos	11
1.3 Justificativa	11
1.4 Organização da Dissertação	12
2 REFERÊNCIA TEÓRICA	13
2.1 Web Semântica	13
2.1.1 Ontologias	13
2.1.2 Vocabulários	13
2.1.3 Linked Data	14
2.2 Proveniência de Dados	14
2.2.1 Sistemas de Gerenciamento de <i>Workflows</i> Científicos (SWfMS)	15
2.2.2 <i>Open Provenance Model</i> (OPM)	15
2.2.3 <i>PROV Data Model</i> (PROV-DM)	16
2.3 Projeto MEX	16
2.3.1 Vocabulário MEX	16
2.3.2 API Log4MEX	17
2.3.3 MEX <i>Interfaces</i>	17
2.4 Web4MEX	18
2.4.1 Funcionamento do Web4Mex	19
2.5 Framework Cliente em Python	20
3 FRAMEWORK C++ PARA O MEX	22
4 RESULTADOS	25
4.1 Construção do Framework	25
4.2 Comparação com o Framework em Python	29
4.3 Teste em Experimento com o FAMa	32

5	CONCLUSÃO	34
6	REFERÊNCIAS BIBLIOGRÁFICAS	35
7	APÊNDICES	37
7.1	APÊNDICE 1: Protótipo do framework em C++	38
7.2	APÊNDICE 2: Teste com o FAMa	60
8	ANEXOS	63
8.1	ANEXO 1: Experimento em Python utilizando o Framework	64

LISTA DE ILUSTRAÇÕES

FIG.3.1	Diagrama do ciclo de dados do Projeto MEX com o Framework em C++	23
FIG.3.2	Diagrama de sequência do <i>Framework</i> em C++	24
FIG.4.1	Tela Inicial do Spring Boot	26
FIG.4.2	Chamada sobrecarregada do método <i>setAuthorName</i>	28

LISTA DE TABELAS

TAB.4.1	Métodos e Parâmetros para a captura de informação do Framework	27
---------	--	----

LISTA DE SIGLAS

FAMa	Framework de Aprendizizado de Máquina
REST	Representational State Transfer
URL	Unified Resource Location
URI	Uniform Resource Identifier
API	Application Programming Interface
JSON	JavaScript Object Notation
HTTP	Hyper Text Transfer Protocol
PFC	Projeto Final de Curso
AM	Aprendizado de Máquina
RDF	Resource Description Framework

RESUMO

Experimentos de aprendizado de máquina estão cada vez mais em alta no mundo da computação. O grande número de ferramentas e utilizações fez com que esse campo expandisse e o número de pesquisadores usando tais métodos aumentasse consideravelmente. O elevado número de ferramentas nas mais diversas linguagens de programação levou à geração de dados não padronizados que, por sua vez, aumentaram a dificuldade na troca de informações e na reprodução de experimentos. Além disso, a falta de notação da proveniência dos dados surge como mais um empecilho para a difusão do conhecimento.

Com fim de solucionar os problemas de proveniência dos dados foi criado o Projeto MEX. O projeto é composto por uma série de ferramentas que visa, de forma fácil, permitir ao usuário registrar seus dados e metadados. Além disso, o projeto propõe um vocabulário para os experimentos de aprendizado de máquina e uma forma automatizada de transcrever as informações para um padrão em RDF. O projeto possuía a limitação de só poder ser utilizado em experimento em Java, fato que foi resolvido pela criação do serviço web Web4MEX. O Web4MEX é um serviço web que visa possibilitar a utilização das ferramentas do projeto MEX por experimentos em linguagem de programação diferente de Java. Através da troca de pacotes JSON, o usuário pode enviar seus dados e receber um RDF formatado. Para fim de facilitar a utilização desse serviço foi desenvolvido um *framework* em Python que realizasse a comunicação para o usuário.

O objetivo desse projeto é desenvolver uma biblioteca em C++ que permita a comunicação com o serviço Web4MEX de forma que possa ser utilizado pelo FAMa. Dessa maneira, uma nova gama de dados de experimentos poderão ser utilizados, de forma fácil e prática, formatados e padronizados.

ABSTRACT

Machine Learning experiments are being more and more discussed by the day in the computer science world. The high variety of tools and utilizations made possible the growth of this field and the number of researchers utilizing its methods. The elevated number of tools on a high variety of programming languages led to the creation of non-standard data which increased the difficulties on the exchange of information and reproductions of experiments. The lack of notation on the data input came as another barrier for the knowledge spread.

With the intent to solve the data input problem the MEX project was created. This project is composed by a series of tools that tries to allow the user to register its data and metadata in an easy way. It also propose a new vocabulary to machine learning experiments and a automated way to transcript the information in the RDF pattern. The MEX project had a restriction that it could only be used within Java experiments, and for that, the Web4MEX was created. The Web4MEX is a web service that makes enables the use of the MEX project by experiments made with programming languages other than Java. By using exchange of JSON packages, the user can send his data and receive a formatted RDF. In order to help the use of this service, a Python framework was developed to do the communication with the user.

The objective of this project is to develop a framework in C++ to make the communication with the Web4MEX service in order to make possible its utilization by FAMA, thus enabling this for whole new level of experiment data that may be formatted in an easy and pratical way.

1 INTRODUÇÃO

Aprendizado de máquina (AM) ou aprendizado automático é um subcampo da ciência da computação que advém do estudo do reconhecimento de padrões e da teoria do aprendizado computacional em inteligência artificial. Têm como base a exploração do estudo e construção de algoritmos que, a partir de amostras previamente adquiridas, podem fazer previsões ou tomar ações otimizadas de certas situações. Possuindo, então, a capacidade de aprender com os próprios erros e, por isso, se tornam de grande utilidade em diversas áreas que se utilizam da análise de padrões para otimizar a decisão, tais como publicidade online, detecção de fraude virtual, jogos de estratégia e economia.

Outro assunto que tem tido grande utilidade dentro da computação é a Web Semântica. Surgiu com o grande crescimento da internet como um movimento colaborativo para categorizar e relacionar os dados presentes na rede com uma formatação específica. Têm como principal ideia que, apesar da imensa quantidade de informação, sempre seja possível criar a relação entre diferentes dados de tal forma que a leitura e a busca sejam facilitadas não só para os agentes humanos como para as máquinas.

Proveniência, na computação, refere-se, geralmente, a origem ou procedência dos dados. Ou seja, é um tópico que visa focar na derivação do dado com o objetivo de clarificar o seu “como”, “quando”, “onde” e “por que” foi obtido e “por quem” foi obtido. Facilitando, então, a compreensão e análise do dado na hora de sua manipulação, pois seu contexto se encontra mais claro.

1.1 MOTIVAÇÃO

De fato, a manipulação eficiente de dados importantes têm se mostrado uma tarefa complicada devida a grande quantidade de informação presente na web.

Por isso, é necessário que exista uma forma eficiente de implementar um padrão entre experimentos de AM tal que seja possível a interoperabilidade entre eles. Sendo, então, uma das melhores soluções até agora a utilização de Web Semântica para a construção de vocabulários dentro de um padrão semântico com o intuito de otimizar as buscas de dados e que os formatos sejam legíveis por todos. Além destes problemas, há em muitos casos também a falta de proveniência dos dados.

Utilizando aprendizado de máquina como exemplo, há, hoje, além de uma quanti-

dade absurda de informação disponível, diversas formas de implementar-se experimentos disponíveis publicamente, mas há, efetivamente, poucas formas de interoperabilidade entre ferramentas ou linguagens diferentes. Afinal elas tendem a possuir interpretações e estruturas diferenciadas

1.2 OBJETIVOS

Este trabalho possui como objetivo principal ampliar o Projeto MEX, tornando-o mais acessível para experimentos escritos em C++ como, por exemplo, aqueles utilizando o *Framework* de Aprendizado de Máquina(FAMA) (DUARTE et al., 2017) e a forma encontrada para alcançar tal objetivo foi através do desenvolvimento de um *framework* em C++ para gerir o fluxo de dados entre experimentos de aprendizagem de máquina e o serviço *Web4MEX*.

Como objetivo secundário, será proposta uma maneira de se descrever o experimento da forma menos intrusa possível no código, inspirada em anotações e decorações.

1.3 JUSTIFICATIVA

A falta de padronização existente nas diversas ferramentas de AM disponíveis e no grande volume de dados se torna um sério problema a partir do momento que essas diversas ferramentas precisam interagir para atuar sobre uma mesma amostra. Afinal, a falta de um padrão entre experimentos torna a leitura e a comparação dos dados complexa e, em AM, que possui uma grande riqueza de detalhes técnicos em suas amostragens, faz com que o tempo necessário seja em grande parte das vezes inviável. Inclusive, tais efeitos negativos afetam diretamente a capacidade do experimento de ser reproduzido e, por consequência, a execução do método científico que exige a reprodutibilidade, pois eles limitam a pesquisa ao uso de apenas uma ferramenta de AM. O conteúdo estudado e desenvolvido neste trabalho pode ser útil para melhorar ainda mais a abrangência dos relatórios de AM gerados, tornando cada vez mais próximo de uma forma universal para se descrever experimentos, independente da linguagem escolhida pelo usuário.

O FAMa, *framework* de experimentos de AM desenvolvido pelo IME, será o maior beneficiado da criação do *framework* em C++. Tal projeto irá permitir ao FAMa a integração com o Web4MEX e a seus usuários a facilidade de salvar os dados em formato padrão.

1.4 ORGANIZAÇÃO DA DISSERTAÇÃO

A dissertação está estruturada da seguinte forma: no capítulo 2 é realizada a referência teórica, onde são apresentados todos os conceitos teóricos estudado e necessários para a realização deste trabalho. No capítulo 3, é feita uma descrição da modelagem do projeto com o diagrama utilizado para representar o processo realizado e a apresentação da arquitetura geral na qual a biblioteca a ser desenvolvida C++ está inserida. No capítulo 4 são apresentados os resultados obtidos pelo projeto: o desenvolvimento de um protótipo, a comparação do *framework* desenvolvido com o já existente em *Python*, a apresentação dos métodos presentes no programa e a análise se um teste da utilização do *framework* em um experimento do FAMa. Finalmente o capítulo 6 apresenta a conclusão do trabalho realizado.

2 REFERÊNCIA TEÓRICA

2.1 WEB SEMÂNTICA

A Web Semântica é uma extensão da web tradicional, onde computadores e pessoas podem trabalhar em cooperação, com o intuito de fazer a informação disponível ser compreensível tanto por agentes humanos, quanto por agentes de softwares (BERNERS-LEE et al., 2001)

Para alcançar tal objetivo, é necessário estruturar a informação contida, através de modelos e estruturas que podem ser deduzidos através de um processo da dedução lógica pelas aplicações. Propósito este que pode ser alcançado através do uso de ontologias.

As ontologias, por sua vez, são utilizadas como ferramenta de abstração de conhecimento específico em diversas áreas, servindo para a construção de vocabulários formalizados e para a transmissão de informação entre agentes.

2.1.1 ONTOLOGIAS

Uma Ontologia é um campo da metafísica que trata dos entes e de sua realidade, ou seja, ela trata o ser enquanto ser. Dito isso, utilizar ontologias é deveras interessante para a definir e formalizar o conhecimento. (HITZLER; JANOWICZ, 2013)

Em ciência da computação, ontologias são utilizadas para descrever o conhecimento de uma área de interesse, no qual a informação contida pode ser processada, possuindo significados formalmente definidos. Por meio de um conjunto de regras e taxonomias para representar entidades de um modelo, ontologias podem ser utilizadas para a construção de vocabulários. Para isto é necessário que as representações sejam bem definidas, de tal forma que não haja ambiguidade na descrição de uma entidade de modo que toda aplicação que utilize essas entidades esteja ciente de seu significado.

2.1.2 VOCABULÁRIOS

Vocabulários podem ser definidos como uma lista de termos enumerados que compõem uma linguagem para descrever formalmente uma área do conhecimento. Os termos utilizados em vocabulários representam conceitos, através de palavras-chave finitas e padroniza-

das. Assim, o objetivo principal de um vocabulário é organizar e formalizar a informação contida nos dados para que posteriormente possa ser recuperada.

Ao construir vocabulários, deve-se tomar cuidado para evitar coisas como a ambiguidade e redundância na definição dos conceitos utilizados, afinal, caso ocorra de existirem múltiplos termos para um mesmo conceito, deve-se escolher um termo preferido para a descrição e os termos remanescentes são listados como sinônimos.

2.1.3 LINKED DATA

Uma quantidade considerável de dados já está disponível na web. No entanto, para que esses dados sejam relevantes e se tornem uma web de dados, é preciso que eles estejam acessíveis, padronizados e conectados. Para isso, existe uma iniciativa chama Linked Data que se propõe a determinar as melhores práticas para a publicação e padronização de dados na Web Semântica. A iniciativa propõe, basicamente, a criação de normas específicas para a acessibilidade e o relacionamento entre vocabulários diferentes.

2.2 PROVENIÊNCIA DE DADOS

A proveniência de dados é o entendimento do contexto de dados, ou seja, um conjunto de informações que descrevem o histórico de um dado, sendo assim é composta por dados sobre o histórico de um determinado dado (metadados) gerados desde de sua origem e incluindo todas as suas mudanças e transformações, e que disponibilizam informações sobre o estado atual do dado (de Souza Costa (2017)).

A proveniência de dados é um assunto pertinente na comunidade científica. Isso ocorre pois informações como “Por que o dado se encontra dessa forma?”, “Como foi processado?”, “Qual foi o equipamento utilizado?”, “Qual sistema operacional foi utilizado?”, “Quem realizou o processo?”, “Quando foi realizado?”, entre outras são importantes para realizar-se medidas e interpretações do contexto do dado conforme ele é manipulado por pessoas diferentes.

Devido ao aumento do volume de dados utilizados nos projetos computacionais atuais com o aumento da capacidade de processamento, a popularização do *big data*, projetos de grande escala abertos à colaboração e processamento distribuído de dados, surgiu a necessidade da coleta e manutenção ainda mais robusta e detalhada dos dados sendo processados. Isso ocorre, pois tais projetos demandam uma descrição rica de seu histórico e metadados disponibilizados para que possam ser validados e compreendidos e possibilitar a manipulação e o compartilhamento de dados de maneira menos custosa ao longo do

desenvolvimento do projeto.

Nesse contexto de crescimento da relevância do tema entre os membros da comunidade científica foram desenvolvidas algumas iniciativas para tratar da proveniência de dados. Em especial foram desenvolvidas iniciativas para tratar a captura de proveniência de dados através de SWfMS (*Scientific Workflow Management Systems*).

2.2.1 SISTEMAS DE GERENCIAMENTO DE *WORKFLOWS* CIENTÍFICOS (SWFMS)

Os sistemas de gerenciamento de *workflows* científicos são responsáveis por gerenciar todo o fluxo de dados e automatizar processos repetitivos. O seu uso é motivado por possuir funcionalidades que facilitam a realização de experimentos como ferramentas gráficas e intuitivas para a construção de fluxos, geração de relatórios sobre todas as etapas do processo, entre outras. O funcionamento desses sistemas se dá através de um pipeline, onde a saída de dados de uma tarefa se conecta a entrada de dados da tarefa seguinte e assim por diante, e tem seu ciclo de vida definido por todas as etapas do fluxo.

Os SWfMS possuem a capacidade de extrair dados de proveniência de experimentos devido a capacidade de consultar os dados em qualquer estágio do ciclo de vida.

2.2.2 *OPEN PROVENANCE MODEL* (OPM)

O OPM é uma iniciativa que visa a padronização da representação de proveniência de dados em sistemas diferentes através da utilização de grafos. Para tal, o modelo define alguns requisitos para a proveniência de dados que seguir o modelo. Entre tais requisitos destacam-se:

- Definição de um modelo preciso de maneira agnóstica a tecnologia;
- Sustentar uma representação digital de proveniência para qualquer "coisa", sendo produzida por computador ou não;
- Definição de um conjunto de regras que identificam inferências válidas que podem ser feitas em grafos de proveniência; etc

Além desses requisitos existe uma preocupação do modelo com a definição de entidades, dependências, papéis, etc, de forma a tornar o modelo robusto para estruturar proveniências de dados diversas.

2.2.3 PROV DATA MODEL (PROV-DM)

O PROV-DM é um conjunto de especificações que visam expressar proveniência em arquivos de dados na web, proposto pela W3C. Tem como proposta ser capaz de capturar proveniências em fontes de dados variadas, sendo assim um modelo genérico.

O PROV-DM utiliza a representação em grafos e segue o *Open Provenance Model*, tendo como principais elementos entidades, atividades e agentes. O projeto MEX utilizou como base para seu desenvolvimento a ontologia PROV, que se trata de uma instância do modelo PROV em forma de ontologia através do uso da linguagem OWL com características adequadas para a representação de um experimento de aprendizagem de máquina genérico.

2.3 PROJETO MEX

A evolução da utilização de AM e as diversas ferramentas existentes acarretam em uma falta de padronização e compatibilidade entre as ferramentas. Cada ferramenta apresenta uma forma específica para representar dados e metadados de tal forma que não existe um modelo padrão para os experimentos. As diferenças presentes fazem com que seja necessário uma grande quantidade de tempo para entender e reproduzir experimentos feitos utilizando tecnologias distintas.

Em face desse problema, a comunidade científica tem trabalhado em formas de padronizar os termos e entidades utilizados nos experimentos de AM através de vocabulários e ontologias.

O Projeto MEX tem como objetivo solucionar o problema de interoperabilidade entre ferramentas e facilitar o compartilhamento de dados gerados por experimentos de AM. O projeto era composto inicialmente por: Vocabulário MEX, a API Log4MEX e o MEX *Framework*. O projeto foi depois expandido incluindo o serviço web Web4MEX(de Souza Costa (2017)) e um *framework* cliente do serviço em Python que serviram de ponto de partida deste trabalho. Essa seção apresenta uma visão resumida do projeto MEX.

2.3.1 VOCABULÁRIO MEX

O vocabulário do MEX(Esteves et al. (2015)) é baseado na ontologia PROV-O e tem como objetivo fornecer um vocabulário para representar os algoritmos e a informação gerada pelos experimentos de aprendizado de máquina, de forma padronizar termos e entidades relevantes ao tema. O MEX é composto de três subvocabulários: MEX *Core*, MEX *Algorithm* e MEX *Performance*.

O MEX *Core* representa as entidades básicas para a execução dos algoritmos. Nesse vocabulário estão inclusas entidades como: informações básicas do projeto(nome do autor, da instituição e outros), objetivo do projeto(Predição de Câncer), informações do experimento(data de criação), entre outros. O MEX *Algorithm* representa os algoritmos de aprendizado de máquina, assim como seus parâmetros de entrada e classes do algoritmo. Entre as entidades desse vocabulário estão: o método de aprendizado(Supervisionado, Não-Supervisionado), a ferramenta utilizada(FAMa, Scikit) entre outros.

O MEX *Performance* representa as saídas da execução de um experimento e as medidas de desempenho obtidas. Entre as suas entidades estão: medidas estatísticas(Correlação Pearson), medidas comuns em problemas de classificação(taxa de falsos positivos e negativos) e outros.

2.3.2 API LOG4MEX

Log4Mex é uma biblioteca em Java cujo objetivo é transcrever os dados para o formato do Vocabulário MEX. Ao realizar a captura de dados e metadados do experimentos e passá-los para o formato descrito no formulário, a biblioteca permite a padronização e homogeneização dos resultados a partir de fontes e ferramentas distintas. Com isso, o tempo necessário para análise e comparação de dados é significativamente reduzido quando comparado ao trato com dados não padronizados.

A biblioteca consiste na instância de objetos de ontologias por intermédio do pacote **org.aksu.mex.util.ontology** e opera utilizando duas classes principais MyMex e MEX-Serializer. A classe MyMEX realiza a captura dos dados e metadados do experimento, enquanto a classe MEXSerializer grava as informações e as trata para o vocabulário.

2.3.3 MEX INTERFACES

O MEX *Interfaces* é um *framework* em Java proposto para a geração de metadados de forma independente da ferramenta utilizada para gerar o experimento. O *framework* automatiza a geração dos metadados de forma que o usuário não precise se preocupar com a utilização da API Log4MEX, funcionando como uma camada intermediária entre a API e o experimento. O *framework* é composto por uma única classe denominada MetaGeneration.

O funcionamento é baseado na utilização das técnicas de Reflexão e Anotação de código. Reflexão é uma técnica que permite a inspeção de variáveis em tempo de execução da aplicação alvo de forma que, ao ser utilizada em Java, permite a inspeção de classes,

métodos, atributos e interfaces em tempo de execução.

Anotação é uma técnica para prover metadados de uma aplicação. Em Java, através do símbolo '@' é possível indicar informações auxiliares que poderão ser capturadas em tempo de compilação ou execução. Dessa forma, utilizando a Reflexão e a Anotação é possível capturar dados e metadados do experimento com pouco esforço de produção de código para o pesquisador.

2.4 WEB4MEX

O projeto MEX apresentou uma solução para problemas de interoperabilidade no compartilhamento de dados entre ferramentas, mas sua utilização está limitada ao uso da API Log4MEX e do *framework* MEX *Interfaces*, ambos desenvolvidos para Java. Diferenças estruturais entre as linguagens tornam-as incompatíveis e motivaram o estudo de formas para usar o vocabulário MEX em ferramentas desenvolvidas para outras linguagens de programação.

Devido às diferenças entre linguagens e ferramentas, as saídas de dados dos experimentos desenvolvidos em cada tecnologia apresentam formatos diferentes. Se faz necessário uma camada intermediária entre o vocabulário e o experimento de forma a permitir a ação do vocabulário MEX e a padronização dos dados e metadados obtidos. A solução escolhida foi a criação de um serviço web que realiza a recepção de dados de experimentos de AM de clientes e realiza a integração com o vocabulário MEX.

O *Web4MEX*, através de chamadas JSON em URIs específicas, converte dados encapsulados extraídos dos experimentos para o vocabulário MEX. De forma que, em qualquer linguagem que se tenha a capacidade de realizar chamadas HTTP e interpretar pacotes JSON, é possível implementar uma camada intermediária que faça o acesso a esse vocabulário.

Seu funcionamento é inspirado nos conceitos de *REST*, apesar de não constituir um serviço *RESTful*, e baseia-se nas especificações do protocolo HTTP, utilizando, principalmente, os verbos GET e POST. Cada entidade do vocabulário é endereçada por uma URI de forma única e os dados são transferidos em pacotes de formato JSON restritos pelo escopo dos dados que tratam. Por exemplo, um pacote com as informações referentes ao hardware, outro referente às informações do autor entre outros.

A divisão em pacotes, ao invés de um arquivo JSON único, permite a diminuição do consumo de banda e processamento, uma vez que somente os pacotes alterados necessitam ser enviados para o servidor à cada atualização do experimento.

Como o projeto MEX foi inicialmente desenvolvido em Java, o *Web4MEX* também foi desenvolvido em Java. O módulo era composto inicialmente por 3 classes: *MexApplication*, *MexController* e *MexServlet*.

- *MexApplication*: A classe é uma extensão da classe *Application* do pacote **javax.ws.rs.core**, não sobrescreve nenhum método desta classe e não implementa nenhum método novo. A classe funciona como *main* do projeto.
- *MexController*: A principal classe do *Web4MEX* é responsável por, ao receber os dados dos clientes, fazer as chamadas à *API Log4MEX*, assim transcrevendo-os para o formato do vocabulário MEX. Cada método é responsável por receber um pacote JSON específico e possui uma URI de endereço única.
- *MexServlet*: Essa classe cria uma instancia de uma página web através de um *servlet* e possui um método para receber requisições e responder mensagens do servidor. A classe tinha como objetivo, também, exibir logs de experimentos e criar uma interface onde o usuário poderia recuperar os dados em formato RDF. Na versão atual do projeto a classe foi substituída pelo *framework Spring Boot*.

O *Spring Boot* é um *framework* que pode ser utilizado para agilizar a criação de serviços web RESTful¹. Com a possibilidade de ser utilizado com servidores com o *Tomcat* ou *Jetty* já completamente integrados ao *framework*, o trabalho que era feito através de um servidor *GlassFish* e a classe *MexServlet* foi substituído pelo uso dessa ferramenta.

2.4.1 FUNCIONAMENTO DO WEB4MEX

Após a geração dos dados do experimento de AM, o *framework*, seja o MEX *Interfaces* ou qualquer outro *framework* desenvolvido para outra linguagem de programação, empacota tanto os dados gerados quanto os metadados e envia para o Web4Mex.

Os pacotes JSON são direcionados para os respectivos métodos da *MexController* responsáveis pelos seus tratamentos e é verificado a falta de dados essenciais ao experimento. Caso algum erro seja detectado, somente o método responsável por esse pacote é interrompido, sendo retornado para o usuário uma mensagem de erro para esse pacote e de confirmação para os demais. Na próxima etapa, os dados são armazenados em cache na forma de arquivos .txt e separados de forma a ter um arquivo para cada método.

¹Guia disponível em <https://spring.io/guides/gs/rest-service/> (PivotalSoftware (2018))

Os dados são então transcritos para o vocabulário MEX. Cada arquivo é aberto e seus dados são tratados pelas classes da API Log4Mex e depois transcritos para .ttl(Turtle) através do *framework Apache Jena* gerando um arquivo único. Por fim o arquivo é armazenado no servidor onde poderá ser enviado a um repositório ou disponibilizado para o usuário. Na requisição do cliente ao serviço é gerado um *token* que é utilizado pra recuperar o RDF gerado na interface web criada pelo *Spring Boot*.

2.5 FRAMEWORK CLIENTE EM PYTHON

O *framework* em Python foi desenvolvido de forma a validar o funcionamento do Web4MEX. O objetivo do trabalho foi desenvolver um *framework* em C++ para gerir o fluxo de dados entre experimentos de aprendizagem de máquina e o serviço *Web4MEX*. Dito isto, é importante notar que tal *framework* é similar ao feito em Python, logo, é de vital importância o entendimento do funcionamento deste. O *framework* tem como objetivo prover um funcionamento similar à versão em Java, mas realizando chamadas ao serviço web ao invés da API.

Para a utilização do *framework* é necessário, somente, a importação de um arquivo. Seu funcionamento dá-se em 3 etapas: encapsulamento dos dados referentes ao experimento, a captura das informações dos experimentos e, por último, o envio ao Web4MEX.

O código fonte do experimento deve ser escrito de forma modular usando funções. Cada função irá tratar uma parte específica dos dados do experimento e gerar um dicionário para cada conjunto de dados. O uso de dicionários ao invés de JSON ocorre devido à semelhança entre os dois e o fato do dicionário ser uma estrutura nativa do *Python*. Os dados tratados por cada função são divididos da mesma forma que a divisão referente ao envio de pacotes JSON ao Web4MEX.

A captura de informações do experimento, que no *framework* em Java era feito utilizando anotação e reflexão, é feito utilizando *decorators*. Tal técnica foi escolhida de forma a gerar uma sintaxe próxima à presente no *Mex Interfaces*. *Decorators* estão presentes em Python desde a versão 2.4, fazendo seu uso viável para quase a totalidade dos experimentos.

Decorators funcionam de uma forma onde é possível que uma função seja enviada como parâmetro para outra. O usuário deve definir funções que retornem os dados necessários pelo *framework* e, depois, passá-las como parâmetro para o método responsável por tratar daquele conjunto de dados. A chamada do decorator no código fonte é dada pelo símbolo “@” seguido do nome da função exatamente antes de definir a função responsável

por passar os dados. Dessa forma, ao chamar a função definida no experimento, a função do *decorator* também será chamada e tomada a primeira como parâmetro.

A utilização desse mecanismo permite que o usuário não seja obrigado a instanciar um objeto do *framework* e possa utilizar a ferramenta com uma sintaxe bem similar à usada em Java.

As funções do *framework* são responsáveis por converter os dados para um formato JSON e passá-los para o serviço web. Todas as funções possuem uma estrutura bem similar e utilizam o método *POST* para realizar chamadas HTTP para o Web4MEX e passar seus respectivos pacotes.

O *framework* também faz a solicitação de um *token* para que os dados possam ser recuperados já no formato padronizado. Tanto o *token*, quanto as respostas dos envios e os pacotes JSON enviados são apresentados para o usuário.

Abaixo temos uma parte de um experimento desenvolvido que utiliza o framework em Python. A utilização do '@' define a chamada do *decorator* e a função definida embaixo dele é a função que será passada como parâmetro. O código completo do experimento encontra-se no apêndice 8.1.

Código 2.1: Python example

```
@mex_framework.author_name
def set_author_name():
    author = 'Igor_Costa'
    return author

@mex_framework.author_email
def set_author_email():
    author_email = 'igor@igor.com'
    return author_email

@mex_framework.organization
def set_organization():
    organization = 'Instituto_Militar_de_Engenharia'
    return organization
```

3 FRAMEWORK C++ PARA O MEX

O *framework* em C++ trabalha como um cliente do Web4MEX e se encarrega de todas as chamadas ao mesmo. Dessa forma o mesmo possibilita que o usuário apenas tenha que instanciar uma classe e chamar seus métodos para utilizar o serviço, garantindo assim a facilidade de utilização do mesmo.

O uso da linguagem C++ é bem popular dentre os pesquisadores que utilizam aprendizado de máquina. Isso ocorre pois o C++ possui uma performance melhor do que as demais linguagens, fornecendo tempos de execução e resposta mais rápidos, sendo assim ideal para projetos que sejam sensíveis ao tempo. Além disso, a linguagem permite reutilização eficiente de código devido à utilização de encapsulamento e herança, não sendo assim muito onerosa na fase de desenvolvimento. Dessa forma, é justificada a escolha da linguagem para ser implementado o MEX *framework* em sua versão cliente do Web4MEX.

O objetivo dessa implementação é que o *framework* em C++ possua um funcionamento similar ao *framework* em *Python* realizando chamadas ao *Web4MEX*, diferentemente do *framework* em *Java* que realiza chamadas diretamente a API Log4MEX. Esse objetivo em questão está representado na Figura 3.1 que demonstra o ciclo de dados do projeto MEX nas 3 linguagens citadas, onde temos no tracejado verde o que já estava pronto e no pontilhado roxo o que foi adicionado pelo *framework*.

O *framework* é composto apenas por uma classe, chamada `mex_framework.cpp`, que é instanciada pelo usuário no experimento para o seu funcionamento. A classe é composta por diversas funções, sendo cada uma responsável por tratar uma porção específica dos dados do experimento.

O seu funcionamento, assim como o do *framework* em *Python*, consiste em três etapas: encapsulamento dos dados gerados, captura de informações dos experimentos e envio das informações ao Web4MEX.

No caso do *framework* em C++, o encapsulamento dos dados gerados é realizado em uma estrutura padrão que no caso é o JSON. A captura de informações é realizada por meio de métodos. E o envio das informações ao Web4MEX é feito através de requisições HTTP utilizando o método POST.

Com base na arquitetura definida acima, foi montado o diagrama de sequência de um dos casos de uso do *framework* de forma a registrar o comportamento do mesmo e exibir as interações e mensagens passadas entre as entidades envolvidas no mesmo.

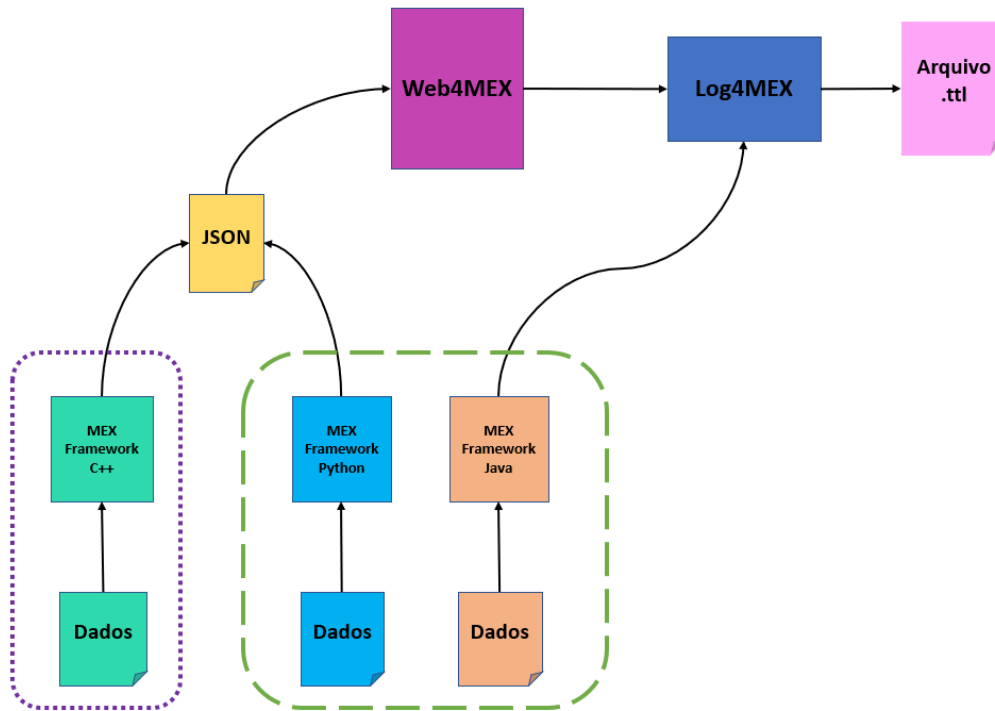


FIG. 3.1: Diagrama do ciclo de dados do Projeto MEX com o Framework em C++

No diagrama da Figura 3.2, é possível notar a ordenação temporal das ações para que um experimento utilize o Web4MEX como repositório de sua proveniência de dados. Assim que o usuário instancia o *framework* é gerado, automaticamente, um token que será usado para identificar o experimento e os seus pacotes no *Web4MEX*. Após isso, temos o usuário fazendo a chamada do método *setAuthorName* que recebe a *string authorName* como parâmetro. O *framework* gera o respectivo JSON e o envia para o serviço web, recebe a resposta e repassa para o usuário.

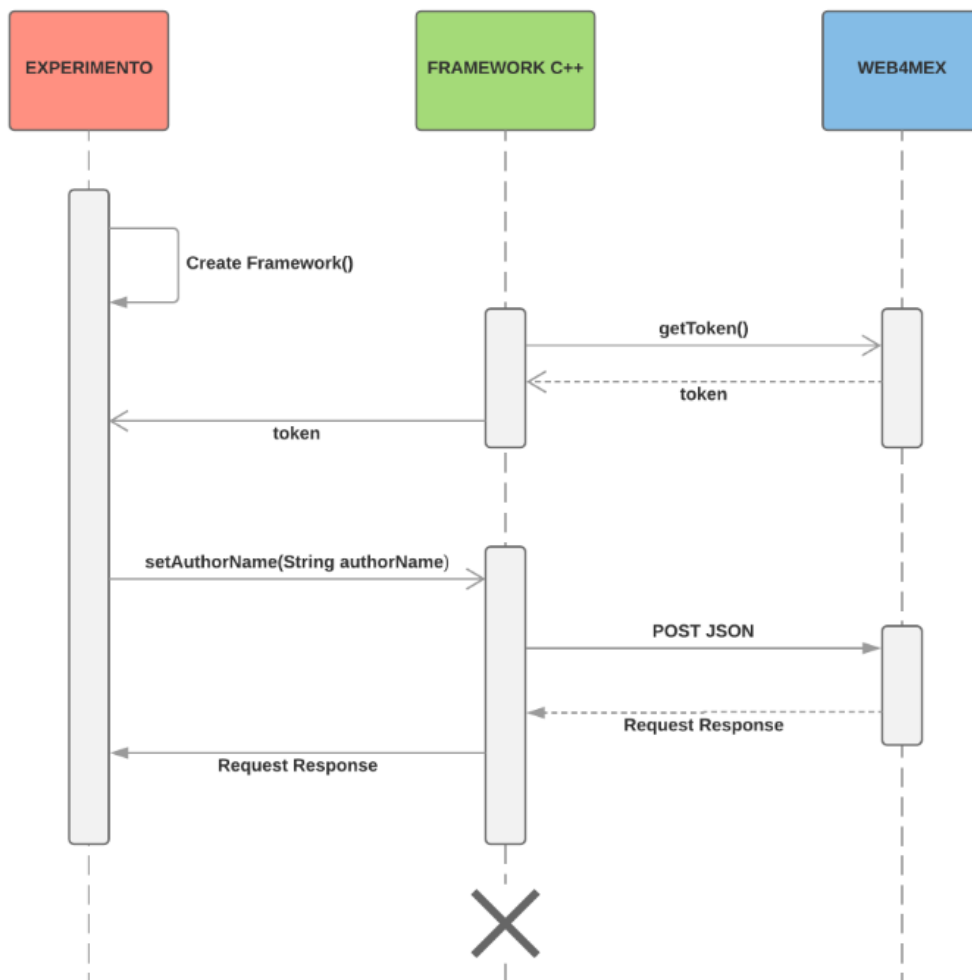


FIG. 3.2: Diagrama de sequência do *Framework* em C++

4 RESULTADOS

Esse capítulo tem por objetivo reportar os resultados obtidos neste trabalho. Após o estudo do sistema *Web4MEX*, definição da forma pela qual seria desenvolvido o *framework* e demonstração da capacidade de realizar chamadas ao servidor em C++ conforme requisitado pelo projeto, o objetivo tornou-se desenvolver o programa em si de forma a consolidar o funcionamento da aplicação e realizar um teste completo com um experimento do FAMa.

Serão abordados pontos relevantes da construção do *framework*, assim como diferenças e semelhanças notáveis em relação ao *framework* em *Python* que serve como modelo para esse projeto, a apresentação de uma tabela contendo todos os métodos do *framework* e os resultados obtidos com o teste em um experimento do FAMa.

4.1 CONSTRUÇÃO DO FRAMEWORK

Ao criar a classe é gerado automaticamente, no construtor, um token que será utilizado pelo framework para passar as informações ao *Web4MEX*, identificando o experimento e permitindo a recuperação do arquivo RDF. O token é disponibilizado para o usuário para ser utilizado no serviço *web*, de forma a gerar um arquivo RDF na extensão desejada e permitida pelo sistema, através do método **getToken()**.

De forma a facilitar o trabalho do pesquisador foi criado o *namespace* MEX. Nesse *namespace* é instanciado um objeto global da classe *mexframework* que corresponde ao *framework* em si. Dessa forma, ao utilizar o *namespace* o usuário já ganha acesso ao objeto inicial sem ser necessário instanciá-lo e, ao mesmo tempo, mantém a liberdade para instanciar novos objetos de forma a passar informação de experimentos distintos simultaneamente.

As chamadas ao *Web4MEX* são feitas através de métodos da classe *mexframework*. O usuário passa as informações como parâmetros e o *framework* se encarrega de passá-las ao serviço web na forma necessária.

O usuário, ao acessar o serviço do *Web4MEX*, tem acesso a uma interface como a da figura 4.1. O recebimento das informações e geração do RDF funciona em uma máquina remota, o servidor do *Web4MEX*, de forma transparente ao usuário. Nessa interface estão presentes todos os métodos que são responsáveis por receber os pacotes JSON e também os métodos para geração do *token* e da obtenção do RDF gerado separados por

controladores.

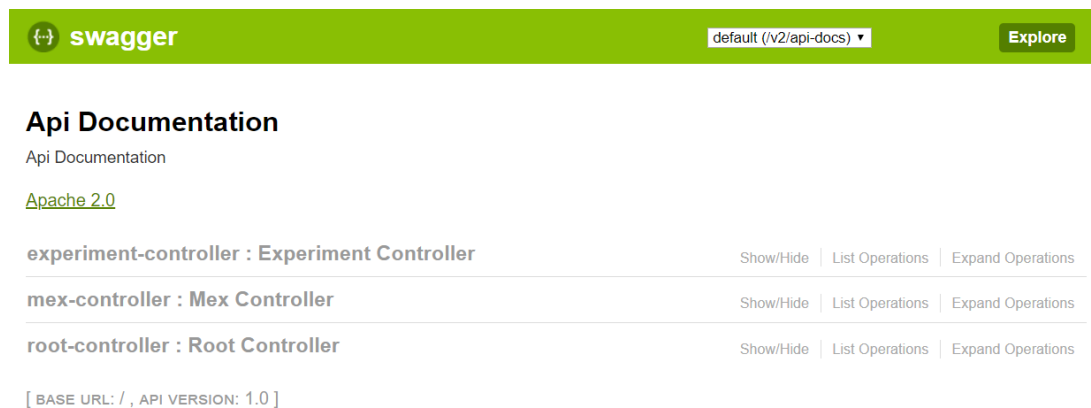


FIG. 4.1: Tela Inicial do Spring Boot

O *Web4MEX* recebe os parâmetros de duas formas: *string* e *JSON*. Alguns métodos como *setAuthorName* e *setAuthorEmail* só passam como informação uma única *string*, enquanto métodos como *setExperimentAlgorithm* devem passar um combinado de *strings* e valores numéricos na forma de um *JSON*.

Devido ao fato de o *JSON* não ser uma estrutura nativa do *C++*, o *framework* foi projetado de forma a facilitar a vida do usuário nesse quesito. O pesquisador somente passa as informações como parâmetros do método desejado, ou as anota ao longo do experimento para depois executar o método desejado, e o *framework* encarrega-se de gerar o *JSON* adequado. Os métodos desenvolvidos e seus parâmetros se encontram na Tabela 4.1.

A biblioteca *cpprestsdk* (Microsoft (2018)) foi desenvolvida de forma a utilizar *wstring* ao invés de somente *string*. A *wstring* é um conjunto de *wchar* ao invés de *char* como a *string* normal. Os *wchar* são mais abrangentes do que o *char*, ocupando 32 bits, diferente dos tradicionais 8 bits do *char*, em sistemas que suportam Unicode (Cplusplusreference (2018)). Uma exceção interessante é o *Windows* onde eles só ocupam 16 bits, suportando UTF-16.

No intuito de não impor essa necessidade de trabalhar com *wstring* para o usuário e, simultaneamente, não impedir a utilização de cadeias de *wchar*, todos os métodos foram sobrecarregados. Dessa forma, o usuário tem liberdade de usar *string* ou *wstring* e o *framework* converte as informações necessárias e faz a chamada ao método conforme mostrado no exemplo 4.1.

```
1 void setAuthorName(string authorNameStr)
2 {
3     wstring authorName = convertString(authorNameStr);
```

```

4   setAuthorName ( authorName );
5 }

```

Código 4.1: Método *setAuthorName* em *C++*

TAB. 4.1: Métodos e Parâmetros para a captura de informação do Framework

Métodos	Parâmetros
setAuthorEmail	string <i>authorEmailStr</i>
setAuthorName	string <i>authorNameStr</i>
setOrganization	string <i>organizationStr</i>
setContext	string <i>contextStr</i>
setExperimentDate	string <i>experimentDateStr</i>
setExperimentDescription	string <i>experimentDescriptionStr</i>
setExperimentId	string <i>experimentIdStr</i>
setExperimentNoiseRemovalDescription	string <i>experimentNoiseRemovalDescriptionStr</i>
setExperimentOutlierDetectionDescription	string <i>experimentOutlierDetectionDescriptionStr</i>
setExperimentTitle	string <i>experimentTitleStr</i>
setExperimentAttributeSelectionDescription	string <i>experimentAttributeSelectionDescriptionStr</i>
setExperimentDataNormalizationDescription	string <i>experimentDataNormalizationDescriptionStr</i>
setExperimentDataSet	string <i>descriptionStr</i> , string <i>nameStr</i> , string <i>uriStr</i>
setExperimentAlgorithm	string <i>algorithmClassStr</i> , string <i>algorithmIdStr</i> , string <i>executionTypeStr</i> , string <i>measureStr</i> , double <i>measureValue</i> , string <i>phaseStr</i>
setSamplingMethod	string <i>aSMStr</i> , int <i>trainSize</i> , int <i>testSize</i>
setExperimentHardware	string <i>cacheStr</i> , string <i>cpuStr</i> , string <i>hdStr</i> , string <i>memoryStr</i> , string <i>osStr</i> , string <i>videoStr</i>

Vale ressaltar que o sistema não suporta que o usuário mescle o uso de *string* e *wstring* na passagem de parâmetros. Os dois métodos disponíveis têm a propriedade de receberem somente *strings* ou somente *wstrings*, devendo o pesquisador atentar para esse uso.

O *Web4MEX* foi desenvolvido de forma a receber como resposta em alguns parâmetros *strings*, ou *wstrings*, definidas e em outros dar a liberdade ao usuário de escrever o texto que melhor o convém. Tomando como exemplo o método *setExperimentAlgorithm* da Figura 4.2, o campo *algorithmID* aceita como parâmetro qualquer *string*, enquanto o campo *measure* só aceita palavras chaves, diferenciando caracteres em caixa alta ou não.

```

void setName(string authorNameStr)
{
    wstring authorName = convertString(authorNameStr);
    setName(authorName);
}

```

FIG. 4.2: Chamada sobrecarregada do método *setName*

O *framework* disponibiliza para o usuário um método, **serialize()**, que captura a resposta do *Web4MEX* e gera o arquivo em *Turtle*, ou outro formato de RDF suportado, poupando o usuário de recuperar manualmente o arquivo no *Web4MEX*.

De forma a facilitar a passagem de informação em métodos que recebem diversos parâmetros, o método *Annotate* foi desenvolvido e uma nova sobrecarga aos métodos da Tabela 4.1 foi criada. Com o auxílio do método o usuário tem a liberdade de passar informações de forma descentralizada conforme for mais interessante no desenvolver do experimento e, por fim, realizar o envio centralizado ao chamar o método referente aos dados sem enviar parâmetros. Tomemos como exemplo a sua utilização para o método *setExperimentAlgorithm*.

Como visto na tabela 4.1, o método *setExperimentAlgorithm* recebe diversos parâmetros que, sem a criação do *Annotate* deveriam ser passados de uma só vez pelo pesquisador. Com a utilização dessa nova funcionalidade, o usuário pode passar, inicialmente, informações que ele já possui, como classe do algoritmo e tipo de execução, e, após o experimento ser realizado, passar individualmente o valores da medida obtida no experimento referente ao parâmetro *measureValue*.

O método exposto no Código 4.2 funciona através do par descritor e informação. O usuário informa o descritor da informação que deseja enviar e seu valor como parâmetros para o método e ele combina essas informações no JSON que será enviado para o *Web4MEX* ao ser chamado. A ressalva que deve ser feita é que o descritor deve ser igual ao valor esperado pela função. Se ao invés de *algorithmClass* o pesquisador decidisse utilizar o descritor *classeDoAlgoritmo*, ele causaria um erro. Tal restrição é condizente com o intuito de padronização dos dados de experimento de AM do projeto MEX. Para projetos futuros, um leque de variações de cada descritor pode ser desenvolvido de forma a facilitar seu uso.

1	framework.Annotate("algorithmClass", "AdaptativeBoost");
2	framework.Annotate("algorithmId", "Adaptative_Boost");
3	framework.Annotate("executionType", "SINGLE");
4	framework.Annotate("measure", "ACCURACY");

```

5     framework.Annotate("measureValue", acuraciaMedia);
6     framework.Annotate("phase", "TRAIN");
7     framework.setExperimentAlgorithm();

```

Código 4.2: Utilização do método *Annotate*

4.2 COMPARAÇÃO COM O FRAMEWORK EM PYTHON

Uma das diretrizes iniciais desse projeto é replicar no *framework* em C++ o trabalho já desenvolvido em *Python*. Dessa forma, a comparação entre os dois programas é de grande valia.

Inicialmente notamos uma diminuição da necessidade de código que o usuário deve adicionar ao seu projeto para a utilização do *framework* em C++, como visto no apêndice 7.2, quando em comparação com o desenvolvido em *Python*, presente no anexo 8.1. Para utilizar cada método no programa em *Python* o usuário deve escrever uma função que será utilizada pelo *decorator* na recuperação das informações. No protótipo desenvolvido em C++, o cliente só precisa chamar um método e passar seus parâmetros, ou utilizar o método *Annotate*, diferença relevante tendo em vista que o objetivo fim do programa é facilitar o trabalho do pesquisador. No Código 4.3 podemos ver como é utilizado o método *setExperimentAlgorithm* em *Python* e compará-lo com a forma mais reduzida do protótipo em C++ presente no Código 4.4.

```

1  @mex_framework.algorithm
2  def set_algorithm():
3      algorithmID = 'Support_Vector_Machine'
4      algorithmClass = 'SupportVectorMachines'
5      executionType = 'SINGLE'
6      measure = 'ACCURACY'
7      measureValue = accuracies
8      phase = 'TRAIN'
9      return { "algorithmClass": algorithmClass,
10              "algorithmID": algorithmID,
11              "executionType": executionType,
12              "measure": measure,
13              "measureValue": measureValue,
14              "phase": phase}

```

Código 4.3: Chamada do método *setExperimentAlgorithm* em *Python*

```

1  framework.setExperimentAlgorithm("SupportVectorMachines",

```

```

2         "Support_Vector_Machine" ,
3         "LINEAR" , "ACCURACY" , 0.98 , "TRAIN" );

```

Código 4.4: Chamada do método *setExperimentAlgorithm* em C++

Em contrapartida, o *framework* em *Python* apresenta pro cliente a possibilidade de nomear livremente os parâmetros que deseja no momento da criação do JSON e, conseqüentemente, o nome deles no RDF. No programa desenvolvido em C++, os nomes dos parâmetros já são definidos pelo *framework* e o usuário só é responsável por passar os respectivos valores.

Na atual implementação do *Web4MEX*, o serviço confere o valor de alguns parâmetros em comparação com um conjunto de valores possíveis, mas não o faz para os nomes. Dessa forma, a liberdade que a versão em *Python* oferece pode ser prejudicial para o objetivo fim do *Web4MEX*: padronização da informação. O usuário ganha uma liberdade vantajosa para a autodocumentação, mas corre o risco de perder o objetivo fim do serviço. O *framework* em C++, ao limitar o usuário, garante a geração de documentos devidamente padronizados.

Diferente do programa em *Python*, pelo fato do *framework* em C++ ser uma classe é possível que o usuário instancie e envie mais de um experimento em um único código. No projeto em *Python*, ao adicionar o *framework* é criado um único token associado ao experimento naquele código, enquanto na versão em C++ o cliente já possui um objeto global para utilizar na comunicação com o *Web4MEX*, mas também tem liberdade de instanciar outros conforme lhe for conveniente.

Vale ressaltar que a liberdade fornecida por poder instanciar diversos objetos do *framework* para enviar múltiplos experimentos em um único código deve ser acompanhada da prudência do pesquisador. O usuário deve atentar para não particionar o mesmo experimento em diversos RDFs e nem confundir o envio de informações entre experimentos.

Alguns métodos do *Web4MEX* requisitam entradas específicas, como já foi mencionado acima, e para isso seria possível realizar uma verificação no *framework* dos parâmetros que o usuário está tentando enviar. Tomando como base o projeto em *Python* e tendo em mente que a verificação já é realizada no servidor, foi optado que o *framework* não fizesse esse controle. Na situação atual do sistema, o pesquisador tem que ter conhecimento, através da documentação, das palavras chave que pode enviar em determinados campos de forma que realizar a verificação delas no servidor e no cliente parece, a princípio, redundante e de não muito ajuda para o usuário.

No *framework* foi inserido um método que não está presente no *framework* em *Python*:

framework.serialize. Esse método recupera o RDF gerado no servidor e gera o documento na máquina do usuário. Por *default* o arquivo é salvo em **ttl**, mas o cliente pode escolher outro formato contanto que o *Web4MEX* tenha a capacidade de gerar o documento no formato desejado.

O funcionamento desse método consiste em gerar um *buffer* associado ao arquivo em que será salvo o RDF e então configurar o servidor proxy cliente para possibilitar o download do RDF através de uma requisição HTTP e escrita do corpo da resposta no buffer que está associado ao arquivo em questão, conforme mostrado no código 4.5 abaixo.

```

1 void serialize(wstring format = L"ttl") {
2     auto fileBuffer = std::make_shared<concurrency::streams::
      streambuf<uint8_t>>();
3     file_buffer<uint8_t>::open(_T("response_-" + token + L".
      ttl"), std::ios::out).then([=](concurrency::streams::
      streambuf<uint8_t> outFile) -> pplx::task<http_response>
4     {
5         *fileBuffer = outFile;
6
7         // Create an HTTP request.
8         // Encode the URI query since it could contain
          special characters like spaces.
9         http_client client(U("http://localhost:3011"),
          client_config_for_proxy());
10        return client.request(methods::GET, uri_builder(
          token).append(L"/serialize").append_query(U("
          format"), format).to_string());
11    })
12
13    // Write the response body into the file buffer.
14    .then([=](http_response response) -> pplx::task<
      size_t>
15    {
16        cout << "Post_Status:_" << response.status_code() <<
          endl;
17
18        return response.body().read_to_end(*fileBuffer);
19    })
20
21    // Close the file buffer.
22    .then([=](size_t)

```

```

23         {
24             return fileBuffer ->close();
25         })
26
27         // Wait for the entire response body to be written
28         // into the file.
29         .wait();
30     }

```

Código 4.5: Método *serialize* em C++

4.3 TESTE EM EXPERIMENTO COM O FAMA

O último passo do projeto foi a realização de um teste da utilização do *framework* em um experimento desenvolvido para o FAMA. Como cliente inicial do produto deste trabalho, conferir o correto funcionamento do produto desenvolvido com esse experimento de AM é de vital importância para validar o que foi feito.

No teste presente no apêndice 7.2, vemos que inicialmente o usuário inclui o arquivo *mex_framework* e, ao utilizar o *namespace* MEX nele presente, inclui automaticamente o objeto **framework** da classe *mex_framework* para utilizar no experimento. Retirando, então, a necessidade da instanciação inicial do objeto **framework** enquanto que, ao mesmo tempo, mantém a liberdade do pesquisador de criar outros objetos caso lhe seja conveniente.

Neste teste são utilizados 5 dos métodos de capturas disponíveis no *framework* e expostos na tabela 4.1: *setAuthorEmail*, *setAuthorName*, *setExperimentAlgorithm*, *setSamplingMethod* e *setExperimentDataSet*. Também é exposta a utilização do método *Annotate* para a passagem de diversas informações referentes ao mesmo pacote de JSON, como ocorre no *setExperimentAlgorithm*.

No recorte do teste exposto em 4.6, vemos como a *Annotate* é utilizado em combinação com o *setExperimentAlgorithm*. As informações que o usuário possuía no início do experimento são enviadas para o *framework* inicialmente e o valor da acurácia é enviado após seu cálculo. O JSON gerado é então enviado quando o método sobrecarregado *setExperimentAlgorithm* é chamado sem envio de parâmetros.

A conclusão com sucesso do teste validou o produto desenvolvido e demonstrou seu potencial na busca da formalização dos dados de experimento de AM.

```

1 framework . Annotate ( "phase" , "TRAIN" );

```

```

2      framework.Annotate("algorithmClass", "AdaptativeBoost");
3      framework.Annotate("algorithmId", "Adaptative_Boost");
4
5      novoatributo = objCorpus.criarAtributo("me");
6
7      framework.Annotate("executionType", "SINGLE");
8      framework.Annotate("measure", "ACCURACY");
9
10     acuraciaMedia = 0;
11     for (c=0;c<ndobras;c++){
12         cout << c << "__" << 100.*v[c][0] << "%\n";
13         acuraciaMedia += v[c][0];
14     }
15     acuraciaMedia /= ndobras;
16     framework.Annotate("measureValue", acuraciaMedia);
17     cout << "*" << acuraciaMedia << "\n";
18
19     framework.setExperimentAlgorithm();

```

Código 4.6: Recorte do Teste com o FAMa

5 CONCLUSÃO

Com o aumento da capacidade de processamento dos computadores e o uso de processamento distribuído, a utilização de aplicações de aprendizagem de máquina, tanto no meio científico quanto comercial, tem sido cada vez mais aproveitada. Com esse avanço cresce a necessidade do desenvolvimento de ferramentas eficientes para a gestão de proveniência de dados, ratificando assim a importância do projeto.

O objetivo do trabalho foi desenvolver um *framework* em C++ para gerir o fluxo de dados entre experimentos de aprendizagem de máquina e o serviço *Web4MEX*. O *framework* visa facilitar aos pesquisadores de AM a utilização do *Web4MEX* e, assim, contribuir para a padronização dos dados dos experimentos nessa área.

O produto final garante ao usuário, através de um modelo simples e intuitivo, a comunicação com o servidor do MEX. Através da chamada de métodos, o pesquisador em C++ tem uma forma simples de fazer a captura dos dados e passá-los no formato JSON sem preocupações adicionais ao experimento. Por fim, a possibilidade de recuperar o RDF através do *framework* foi adicionada, provendo um auxílio que não estava presente no projeto em Python.

Como proposta de trabalhos futuros o *framework* pode continuar a ser otimizado de forma a cada vez mais facilitar a vida do pesquisador ao apresentar maior flexibilidade para as necessidades específicas do projeto e podem ser implementados *frameworks* para outras linguagens usuais em experimentos de aprendizagem de máquina como R, MatLab, etc.

6 REFERÊNCIAS BIBLIOGRÁFICAS

- BERNERS-LEE, T.; HENDLER, J. ; LASSILA, O. The semantic web. **Scientific American**, v. 284, n. 5, p. 34–43, 2001. Disponível em: <<http://www.sciam.com/article.cfm?articleID=00048144-10D2-1C70-84A9809EC588EF21>>. Acesso em: 2018-05-04.
- CPPREFERENCE. Fundamental types. Disponível em: <<https://en.cppreference.com/w/cpp/language/types>>. Acesso em: 25 julho de 2018.
- DE SOUZA COSTA, I. **UM SERVIÇO INTEROPERÁVEL PARA A PROVENIÊNCIA DE DADOS EM EXPERIMENTOS DE APRENDIZADO DE MÁQUINA**. 2017. 100 f. Dissertação (Mestrado em Sistemas e Computação) – Instituto Militar de Engenharia, Rio de Janeiro, 2017.
- DUARTE, J. C.; CAVALCANTI, M. C. R.; DE SOUZA COSTA, I. ; ESTEVES, D. An interoperable service for the provenance of machine learning experiments. In: PROCEEDINGS OF THE INTERNATIONAL CONFERENCE ON WEB INTELLIGENCE, 1., 2017. **Anais eletrônicos...** New York, NY, USA: ACM, 2017, p. 132–138. Disponível em: <<http://doi.acm.org/10.1145/3106426.3106496>>. Acesso em: 7 de maio de 2018.
- ESTEVES, D.; MOUSSALLEM, D.; NETO, C. B.; SORU, T.; USBECK, R.; ACKERMANN, M. ; LEHMANN, J. Mex vocabulary: A lightweight interchange format for machine learning experiments. In: PROCEEDINGS OF THE 11TH INTERNATIONAL CONFERENCE ON SEMANTIC SYSTEMS, 11., 2015. **Anais eletrônicos...** New York, NY, USA: ACM, 2015, p. 169–176. Disponível em: <<http://doi.acm.org/10.1145/2814864.2814883>>. Acesso em: 7 de maio de 2018.
- HITZLER, P.; JANOWICZ, K. Linked data, big data, and the 4th paradigm. **Semant. web**, v. 4, n. 3, p. 233–235, 2013. Disponível em: <<http://dl.acm.org/citation.cfm?id=2786071.2786072>>. Acesso em: 2018-05-04.
- MICROSOFT. C++ REST SDK Documentation. Disponível em: <<https://github.com/Microsoft/cpprestsdk/wiki>>. Acesso em: 3 março de 2018.

PIVOTALSOFTWARE. Building a RESTful Web Service. Disponível em:
<<https://spring.io/guides/gs/rest-service/>>. Acesso em: 7 maio de 2018.

7 APÊNDICES

APÊNDICE 1: PROTÓTIPO DO FRAMEWORK EM C++

```
1 #ifndef MEX_FRAMEWORK_H
2 #define MEX_FRAMEWORK_H
3 #include <cpprest/http_client.h>
4 #include <cpprest/filestream.h>
5 #include <cpprest/json.h>
6 #include <unordered_map>
7 using namespace utility;
8 using namespace web;
9 using namespace web::http;
10 using namespace web::http::client;
11 using namespace concurrency::streams;
12 typedef std::unordered_map<wstring, wstring> dictionary;
13 class mex_framework
14 {
15 private:
16     wstring token;
17     dictionary Annotations;
18     wstring makeJsonLine(wstring parameterName, wstring parameter
19         , bool last = false)
20     {
21         wstring jsonLine = U("\"" + parameterName + L"\":_\"
22             + L\"_\" + parameter + L\"_\"");
23         if (last)
24             return jsonLine;
25         return jsonLine + L",";
26     }
27     wstring makeJsonLine(wstring parameterName, double parameter,
28         bool last = false)
29     {
30         wstring jsonLine = U("\"" + parameterName + L"\":_\"
31             + to_wstring(parameter));
32         if (last)
```



```

29         return jsonLine;
30     return jsonLine + L",";
31 }
32 wstring convertString(string str)
33 {
34     wstring wstr(str.begin(), str.end());
35     return wstr;
36 }
37 web::http::client::http_client_config client_config_for_proxy
38     ()
39 {
40     web::http::client::http_client_config client_config;
41 #ifdef _WIN32
42     wchar_t* pValue = nullptr;
43     std::unique_ptr<wchar_t, void(*) (wchar_t*)> holder(
44         nullptr, [](wchar_t* p) { free(p); });
45     size_t len = 0;
46     auto err = _wdupenv_s(&pValue, &len, L"http_proxy");
47     if (pValue)
48     {
49         holder.reset(pValue);
50         if (!err && pValue && len) {
51             std::wstring env_http_proxy_string(pValue,
52                 len - 1);
53 #else
54             if (const char* env_http_proxy = std::getenv("
55                 http_proxy")) {
56                 std::string env_http_proxy_string(
57                     env_http_proxy);
58 #endif
59                 if (env_http_proxy_string == U("auto"))
60                     client_config.set_proxy(web::
61                         web_proxy::use_auto_discovery);
62                 else
63                     client_config.set_proxy(web::
64                         web_proxy(env_http_proxy_string));
65             }
66 }

```

```

59         return client_config;
60     }
61
62 public:
63     mex_framework()
64     {
65         createToken();
66         wcout << L"Token_:_" << getToken() << endl;
67     };
68     void Annotate(string descriptionStr, string valueStr)
69     {
70         wstring description = convertString(descriptionStr);
71         wstring value = convertString(valueStr);
72         Annotate(description, value);
73     }
74     void Annotate(string descriptionStr, int valueInt)
75     {
76         wstring description = convertString(descriptionStr);
77         wstring value = std::to_wstring(valueInt);
78         Annotate(description, value);
79     }
80     void Annotate(wstring description, int valueInt)
81     {
82         wstring value = std::to_wstring(valueInt);
83         Annotate(description, value);
84     }
85     void Annotate(string descriptionStr, double valueDouble)
86     {
87         wstring description = convertString(descriptionStr);
88         wstring value = std::to_wstring(valueDouble);
89         Annotate(description, value);
90     }
91     void Annotate(wstring description, double valueDouble)
92     {
93         wstring value = std::to_wstring(valueDouble);
94         Annotate(description, value);
95     }

```

```

96     void Annotate(wstring description , wstring value)
97     {
98         Annotations.insert({description , value});
99     }
100    void createToken()
101    {
102        wstring tokenTemp;
103        http_client client(U("http://localhost:3011"));
104
105        client
106            .request(methods::GET, uri_builder(L"token").
107                to_string())
108            .then([&tokenTemp](http_response response) ->
109                pplx::task<wstring>
110            {
111                if (response.status_code() == status_codes::
112                    OK)
113                {
114                    return response.extract_string();
115                }
116                return pplx::task_from_result(wstring());
117            })
118            .then([&tokenTemp](pplx::task<wstring> result
119                )
120            {
121                try
122                {
123                    tokenTemp = result.get();
124                }
125                catch (http_exception const & e)
126                {
127                    wcout << e.what() << endl;
128                }
129            })
130            .wait();
131        token = tokenTemp;
132    }

```

```

129     void setAuthorEmail(string authorEmailStr)
130     {
131         wstring authorEmail = convertString(authorEmailStr);
132         setAuthorEmail(authorEmail);
133     }
134     void setAuthorEmail(wstring authorEmail)
135     {
136         http_client client(U("http://localhost:3011"));
137         client
138             .request(methods::POST, uri_builder(token).
139                 append(L"authorEmail").to_string(),
140                 authorEmail, L"application/json")
141             .then([](http_response response) -> void
142                 {
143                     try
144                     {
145                         cout << "Post_Status:_ " << response.
146                             status_code() << endl;
147                     }
148                     catch (http_exception const & e)
149                     {
150                         wcout << e.what() << endl;
151                     }
152                     return;
153                 })
154             .wait();
155     }
156     void setAuthorName(string authorNameStr)
157     {
158         wstring authorName = convertString(authorNameStr);
159         setAuthorName(authorName);
160     }
161     void setAuthorName(wstring authorName)
162     {
163         http_client client(U("http://localhost:3011"));
164         client
165             .request(methods::POST, uri_builder(token).

```

```

        append(L"authorName").to_string(),
        authorName, L"application/json")
163     .then([](http_response response) -> void
164     {
165         try
166         {
167             cout << "Post_Status:_ " << response.
                status_code()<<endl;
168         }
169         catch (http_exception const & e)
170         {
171             wcout << e.what() << endl;
172         }
173         return;
174     })
175     .wait();
176 }
177 void setOrganization(string organizationStr)
178 {
179     wstring organization = convertString(organizationStr)
        ;
180     setOrganization(organization);
181 }
182 void setOrganization(wstring organization)
183 {
184     http_client client(U("http://localhost:3011"));
185     client
186         .request(methods::POST, uri_builder(token).
            append(L"organization").to_string(),
            organization, L"application/json")
187     .then([](http_response response) -> void
188     {
189         try
190         {
191             cout << "Post_Status:_ " << response.
                status_code()<<endl;
192         }

```

```

193         catch (http_exception const & e)
194         {
195             wcout << e.what() << endl;
196         }
197         return;
198     })
199     .wait();
200 }
201 void setContext(string contextStr)
202 {
203     wstring context = convertString(contextStr);
204     setContext(context);
205 }
206 void setContext(wstring context)
207 {
208     http_client client(U("http://localhost:3011"));
209     client
210         .request(methods::POST, uri_builder(token).
211             append(L"context").to_string(), context, L
212             "application/json")
213         .then([](http_response response) -> void
214             {
215                 try
216                 {
217                     cout << "Post_Status:_ " << response.
218                         status_code() << endl;
219                 }
220                 catch (http_exception const & e)
221                 {
222                     wcout << e.what() << endl;
223                 }
224                 return;
225             })
226     .wait();
227 }
228 void setExperimentDate(string experimentDateStr)
229 {

```

```

227         wstring experimentDate = convertString(
                experimentDateStr);
228         setExperimentDate(experimentDate);
229     }
230     void setExperimentDate(wstring experimentDate)
231     {
232         http_client client(U("http://localhost:3011"));
233         client
234             .request(methods::POST, uri_builder(token).
                append(L"experimentDate").to_string(),
                experimentDate, L"application/json")
235             .then([](http_response response) -> void
236                 {
237                     try
238                     {
239                         cout << "Post_Status:_ " << response.
                            status_code() << endl;
240                     }
241                     catch (http_exception const & e)
242                     {
243                         wcout << e.what() << endl;
244                     }
245                     return;
246                 })
247             .wait();
248     }
249     void setExperimentDescription(string experimentDescriptionStr
        )
250     {
251         wstring experimentDescription = convertString(
                experimentDescriptionStr);
252         setExperimentDescription(experimentDescription);
253     }
254     void setExperimentDescription(wstring experimentDescription)
255     {
256         http_client client(U("http://localhost:3011"));
257         client

```

```

258         .request(methods::POST, uri_builder(token).
                append(L"experimentDescription").to_string
                (), experimentDescription, L"application/
                json")
259         .then([](http_response response) -> void
260             {
261                 try
262                 {
263                     cout << "Post_Status:_ " << response.
                        status_code() << endl;
264                 }
265                 catch (http_exception const & e)
266                 {
267                     wcout << e.what() << endl;
268                 }
269                 return;
270             })
271         .wait();
272     }
273     void setIdExperiment(string experimentIdStr)
274     {
275         wstring experimentId = convertString(experimentIdStr)
                ;
276         setIdExperiment(experimentId);
277     }
278     void setIdExperiment(wstring experimentId)
279     {
280         http_client client(U("http://localhost:3011"));
281         client
282             .request(methods::POST, uri_builder(token).
                append(L"experimentId").to_string(),
                experimentId, L"application/json")
283             .then([](http_response response) -> void
284                 {
285                     try
286                     {
287                         cout << "Post_Status:_ " << response.

```



```

288         status_code() << endl;
289     }
290     catch (http_exception const & e)
291     {
292         wcout << e.what() << endl;
293     }
294     return;
295 })
296     .wait();
297 }
298 void setExperimentNoiseRemovalDescription(string
299     experimentNoiseRemovalDescriptionStr)
300 {
301     wstring experimentNoiseRemovalDescription =
302         convertString(experimentNoiseRemovalDescriptionStr
303             );
304     setExperimentNoiseRemovalDescription(
305         experimentNoiseRemovalDescription);
306 }
307 void setExperimentNoiseRemovalDescription(wstring
308     experimentNoiseRemovalDescription)
309 {
310     http_client client(U("http://localhost:3011"));
311     client
312         .request(methods::POST, uri_builder(token).
313             append(L"experimentNoiseRemovalDescription
314                 ").to_string(),
315             experimentNoiseRemovalDescription, L"
316                 application/json")
317         .then([](http_response response) -> void
318             {
319                 try
320                 {
321                     cout << "Post_Status:_" << response.
322                         status_code() << endl;
323                 }
324                 catch (http_exception const & e)

```

```

314         {
315             wcout << e.what() << endl;
316         }
317         return;
318     })
319     .wait();
320 }
321 void setExperimentOutlierDetectionDescription(string
    experimentOutlierDetectionDescriptionStr)
322 {
323     wstring experimentOutlierDetectionDescription =
        convertString(
            experimentOutlierDetectionDescriptionStr);
324     setExperimentOutlierDetectionDescription(
        experimentOutlierDetectionDescription);
325 }
326 void setExperimentOutlierDetectionDescription(wstring
    experimentOutlierDetectionDescription)
327 {
328     http_client client(U("http://localhost:3011"));
329     client
330         .request(methods::POST, uri_builder(token).
            append(L"
                experimentOutlierDetectionDescription").
            to_string(),
                experimentOutlierDetectionDescription, L"
                application/json")
331         .then([](http_response response) -> void
332             {
333                 try
334                 {
335                     cout << "Post_Status:_ " << response.
                        status_code() << endl;
336                 }
337                 catch (http_exception const & e)
338                 {
339                     wcout << e.what() << endl;

```

```

340         }
341         return;
342     })
343     .wait();
344 }
345 void setExperimentTitle(string experimentTitleStr)
346 {
347     wstring experimentTitle = convertString(
348         experimentTitleStr);
349     setExperimentTitle(experimentTitle);
350 }
351 void setExperimentTitle(wstring experimentTitle)
352 {
353     http_client client(U("http://localhost:3011"));
354     client
355         .request(methods::POST, uri_builder(token).
356             append(L"experimentTitle").to_string(),
357             experimentTitle, L"application/json")
358         .then([](http_response response) -> void
359             {
360                 try
361                 {
362                     cout << "Post_Status:_" << response.
363                         status_code() << endl;
364                 }
365                 catch (http_exception const & e)
366                 {
367                     wcout << e.what() << endl;
368                 }
369                 return;
370             })
371     .wait();
372 }
373 void setExperimentAttributeSelectionDescription(string
374     experimentAttributeSelectionDescriptionStr)
375 {
376     wstring experimentAttributeSelectionDescription =

```

```

372         convertString(
            experimentAttributeSelectionDescriptionStr);
373     setExperimentAttributeSelectionDescription(
        experimentAttributeSelectionDescription);
374 }
375 void setExperimentAttributeSelectionDescription(wstring
    experimentAttributeSelectionDescription)
376 {
377     http_client client(U("http://localhost:3011"));
378     client
        .request(methods::POST, uri_builder(token).
            append(L"
                experimentAttributeSelectionDescription").
                to_string(),
                experimentAttributeSelectionDescription, L
                "application/json")
379     .then([](http_response response) -> void
380     {
381         try
382         {
383             cout << "Post_Status:_ " << response.
                status_code() << endl;
384         }
385         catch (http_exception const & e)
386         {
387             wcout << e.what() << endl;
388         }
389         return;
390     })
391     .wait();
392 }
393 void setExperimentDataNormalizationDescription(string
    experimentDataNormalizationDescriptionStr)
394 {
395     wstring experimentDataNormalizationDescription =
        convertString(
            experimentDataNormalizationDescriptionStr);

```

```

396         setExperimentDataNormalizationDescription(
397             experimentDataNormalizationDescription);
398     }
399     void setExperimentDataNormalizationDescription(wstring
400         experimentDataNormalizationDescription)
401     {
402         http_client client(U("http://localhost:3011"));
403         client
404             .request(methods::POST, uri_builder(token).
405                 append(L"
406                     experimentDataNormalizationDescription").
407                     to_string(),
408                     experimentDataNormalizationDescription, L"
409                         application/json")
410             .then([](http_response response) -> void
411                 {
412                     try
413                     {
414                         cout << "Post_Status:_ " << response.
415                             status_code() << endl;
416                     }
417                     catch (http_exception const & e)
418                     {
419                         wcout << e.what() << endl;
420                     }
421                     return;
422                 })
423             .wait();
424     }
425     void setExperimentDataSet(string descriptionStr, string
426         nameStr, string uriStr)
427     {
428         wstring description = convertString(descriptionStr);
429         wstring name = convertString(nameStr);
430         wstring uri = convertString(uriStr);
431         setExperimentDataSet(description, name, uri);
432     }

```

```

424 void setExperimentDataSet(wstring description, wstring name,
425                             wstring uri)
426 {
427     http_client client(U("http://localhost:3011"));
428     web::json::value obj = json::value::parse(U("{_" +
429         makeJsonLine(convertString("description"),
430             description) + makeJsonLine(convertString("name"),
431             name) + makeJsonLine(convertString("uri"), uri,
432             true) + L"_}"));
433     client
434         .request(methods::POST, uri_builder(token).
435             append(L"experimentDataset").to_string(),
436             obj)
437         .then([](http_response response) -> void
438 {
439     try
440     {
441         cout << "Post_Status:_" << response.
442             status_code() << endl;
443     }
444     catch (http_exception const & e)
445     {
446         wcout << e.what() << endl;
447     }
448     return;
449 })
450     .wait();
451 }
452 void setExperimentDataSet()
453 {
454     wstring description, name, uri;
455     if ((description = Annotations[wstring(L"description"
456         )]).empty() || (name = Annotations[wstring(L"name"
457         )]).empty() || (uri = Annotations[wstring(L"uri"
458         )]).empty())
459     {
460         cout << "Post_Status:_Error_(Not_enough_

```

```

450         annotations" << endl;
451     }
452     setExperimentDataSet(description, name, uri);
453 }
454 void setExperimentAlgorithm(string algorithmClassStr, string
    algorithmIdStr, string executionTypeStr, string measureStr
    , double measureValue, string phaseStr)
455 {
456     wstring algorithmClass = convertString(
        algorithmClassStr);
457     wstring algorithmId = convertString(algorithmIdStr);
458     wstring executionType = convertString(
        executionTypeStr);
459     wstring phase = convertString(phaseStr);
460     wstring measure = convertString(measureStr);
461     setExperimentAlgorithm(algorithmClass, algorithmId,
        executionType, measure, measureValue, phase);
462 }
463 void setExperimentAlgorithm(wstring algorithmClass, wstring
    algorithmId, wstring executionType, wstring measure,
    double measureValue, wstring phase)
464 {
465     http_client client(U("http://localhost:3011"));
466     web::json::value obj = json::value::parse(U("{_" +
        makeJsonLine(convertString("algorithmClass"),
        algorithmClass) + makeJsonLine(convertString("
        algorithmID"), algorithmId) + makeJsonLine(
        convertString("executionType"), executionType) +
        makeJsonLine(convertString("measure"), measure) +
        makeJsonLine(convertString("measureValue"),
        measureValue) + makeJsonLine(convertString("phase"
        ), phase, true) + L"_}"));
467     client
468         .request(methods::POST, uri_builder(token).
            append(L"experimentAlgorithm").to_string()
            , obj)

```

```

469         .then([](http_response response) -> void
470     {
471         try
472         {
473             cout << "Post_Status:_ " << response.
474                 status_code() << endl;
475         }
476         catch (http_exception const & e)
477         {
478             wcout << e.what() << endl;
479         }
480         return;
481     })
482     .wait();
483 }
484 void setExperimentAlgorithm()
485 {
486     wstring algorithmClass, algorithmId, executionType,
487         measure, measureValue, phase;
488     if ((algorithmClass = Annotations[wstring(L"
489         algorithmClass")]).empty() || (algorithmId =
490         Annotations[wstring(L"algorithmId")]).empty() || (
491         executionType = Annotations[wstring(L"
492         executionType")]).empty() || (measure =
493         Annotations[wstring(L"measure")]).empty() || (
494         measureValue = Annotations[wstring(L"measureValue"
495         )]).empty() || (phase = Annotations[wstring(L"
496         phase")]).empty())
497     {
498         cout << "Post_Status:_Error_(Not_enough_
499             annotations" << endl;
500         return;
501     }
502     setExperimentAlgorithm(algorithmClass, algorithmId,
503         executionType, measure, (double)_wtof(measureValue
504         .c_str()), phase);
505 }

```



```

493 void setSamplingMethod(string aSMStr, int trainSize, int
      testSize) {
494     wstring aSM = convertString(aSMStr);
495     setSamplingMethod(aSM, trainSize, testSize);
496 }
497 void setSamplingMethod(wstring aSM, int trainSize, int
      testSize) {
498     http_client client(U("http://localhost:3011"));
499     web::json::value obj = json::value::parse(U("{_ " +
      makeJsonLine(convertString("aSM"), aSM) +
      makeJsonLine(convertString("trainSize"), trainSize
      ) + makeJsonLine(convertString("testSize"),
      testSize, true) + L"}"));
500     client
501         .request(methods::POST, uri_builder(token).
      append(L"experimentSamplingMethod").
      to_string(), obj)
502         .then([](http_response response) -> void
503 {
504     try
505     {
506         cout << "Post_Status:_ " << response.
      status_code() << endl;
507     }
508     catch (http_exception const & e)
509     {
510         wcout << e.what() << endl;
511     }
512     return;
513 })
514     .wait();
515 }
516 void setSamplingMethod()
517 {
518     wstring aSM, trainSize, testSize;
519     if ((aSM = Annotations[wstring(L"aSM")]).empty() || (
      trainSize = Annotations[wstring(L"trainSize")]).

```

```

        empty() || (testSize = Annotations[wstring(L"
        testSize")]).empty())
520     {
521         cout << "Post_Status:_Error_(Not_enough_
        annotations" << endl;
522         return;
523     }
524     setSamplingMethod(aSM, std::stoi(trainSize.c_str()),
        std::stoi(testSize.c_str()));
525 }
526 void setExperimentHardware(string cacheStr, string cpuStr,
    string hdStr, string memoryStr, string osStr, string
    videoStr)
527 {
528     wstring cache = convertString(cacheStr);
529     wstring cpu = convertString(cpuStr);
530     wstring hd = convertString(hdStr);
531     wstring memory = convertString(memoryStr);
532     wstring os = convertString(osStr);
533     wstring video = convertString(videoStr);
534     setExperimentHardware(cache, cpu, hd, memory, os,
        video);
535 }
536 void setExperimentHardware(wstring cache, wstring cpu,
    wstring hd, wstring memory, wstring os, wstring video)
537 {
538     http_client client(U("http://localhost:3011"));
539     web::json::value obj = json::value::parse(U("{_ " +
        makeJsonLine(convertString("cache"), cache) +
        makeJsonLine(convertString("cpu"), cpu) +
        makeJsonLine(convertString("hd"), hd) +
        makeJsonLine(convertString("memory"), memory) +
        makeJsonLine(convertString("os"), os) +
        makeJsonLine(convertString("video"), video, true)
        + L"_}"));
540     client
541         .request(methods::POST, uri_builder(token).

```

```

        append(L"experimentHardware").to_string(),
        obj)
542     .then([](http_response response) -> void
543     {
544         try
545         {
546             cout << "Post_Status:_ " << response.
                    status_code() << endl;
547         }
548         catch (http_exception const & e)
549         {
550             wcout << e.what() << endl;
551         }
552         return;
553     })
554     .wait();
555 }
556 void setExperimentHardware()
557 {
558     wstring cache, cpu, hd, memory, os, video;
559     if((cache = Annotations[wstring(L"cache")]).empty()
        || (cpu = Annotations[wstring(L"cpu")]).empty() || (
            hd = Annotations[wstring(L"hd")]).empty() || (
            memory = Annotations[wstring(L"memory")]).empty()
        || (os = Annotations[wstring(L"os")]).empty() || (
            video = Annotations[wstring(L"video")]).empty())
560     {
561         cout << "Post_Status:_Error_(Not_enough_
                    annotations"<<endl;
562         return;
563     }
564     setExperimentHardware(cache, cpu, hd, memory, os,
                    video);
565 }
566 void serialize(wstring format = L"ttl") {
567     auto fileBuffer = std::make_shared<concurrency::
                    streams::streambuf<uint8_t>>();

```

```

568     file_buffer<uint8_t>::open(_T("response_-" + token +
569         L".ttl"), std::ios::out).then([=](concurrency::
570         streams::streambuf<uint8_t> outFile) -> pplx::task
571         <http_response>
572     {
573         *fileBuffer = outFile;
574
575         // Create an HTTP request.
576         // Encode the URI query since it could
577         // contain special characters like spaces.
578         http_client client(U("http://localhost:3011")
579             , client_config_for_proxy());
580         return client.request(methods::GET,
581             uri_builder(token).append(L"/serialize").
582             append_query(U("format"), format).
583             to_string());
584     })
585
586     // Write the response body into the file
587     // buffer.
588     .then([=](http_response response) -> pplx::
589     task<size_t>
590     {
591         cout << "Post_Status:_" << response.
592             status_code()<<endl;
593
594         return response.body().read_to_end(*
595             fileBuffer);
596     })
597
598     // Close the file buffer.
599     .then([=](size_t)
600     {
601         return fileBuffer->close();
602     })
603
604     // Wait for the entire response body to be

```

```

593         written into the file.
594         .wait();
595     }
596     wstring getToken() { return token; }
597 };
598
599 namespace MEX {
600     mex_framework framework;
601 }
602 #endif

```

APÊNDICE 2: TESTE COM O FAMA

```
1 #include <iostream>
2 #include <fstream>
3 #include "../..../treinador/treinador.h"
4 #include "../..../treinador/adaboost.h"
5 #include "../..../stump/stump.h"
6 #include "../..../classificador/classificador.h"
7 #include "../..../corpus/corpusmatriz.h"
8 #include "../..../avaliador/avaliador_acuracia.h"
9 #include "../..../validador/validadorkdobras.h"
10
11 using namespace std;
12 #include "mex_framework.h"
13 using namespace MEX;
14
15 int main()
16 {
17
18     framework.setAuthorEmail("autor@gmail.com");
19     framework.setAuthorName("Autor");
20
21     vector<string> atributos, classes;
22     int atributo, novoatributo, c;
23     AvaliadorAcuracia objAvalAcur;
24
25     float acuraciaMedia;
26
27     srand(time(NULL));
28
29     //carrega conjunto de dados
30     CorpusMatriz objCorpus(atributos, ',', true, true);
31     objCorpus.carregarArquivo("../..../inputs/adult.data");
32
```

```

33 framework.Annotate("name", "adult");
34 framework.Annotate("uri", "../..../inputs/adult.data");
35 framework.Annotate("description", "Base_de_dados_sobre_adultos");
36
37     atributo = objCorpus.pegarPosAtributo("resposta");
38
39     classes.push_back("<=50K");
40     classes.push_back(">50K");
41
42     atributos = objCorpus.pegarAtributos();
43     atributos.pop_back();
44
45     //treina
46
47     framework.Annotate("phase", "TRAIN");
48
49     DecisionStump stump(atributos, classes);
50     TreinadorAdaboost adab(stump, classes, 30, 0.01, false);
51 framework.Annotate("trainSize", 30);
52 framework.Annotate("iterations", 30);
53 framework.Annotate("minAlpha", 30);
54 framework.Annotate("baseAlgorithm", "DecisionStump");
55 framework.Annotate("algorithmClass", "AdaptativeBoost");
56 framework.Annotate("algorithmId", "Adaptative_Boost");
57
58     novoatributo = objCorpus.criarAtributo("me");
59
60 framework.Annotate("executionType", "SINGLE");
61
62     //faz experimento
63     int ndobras = 10;
64     ValidadorKDobras objValidador(objAvalAcur, ndobras, false);
65 framework.Annotate("aSM", "N_FOLDS_CROSS_VALIDATION");
66     vector< vector< float > > v;
67     v = objValidador.executarExperimento(adab, objCorpus, atributo,
        novoatributo);
68

```

```

69 framework.Annotate("measure", "ACCURACY");
70 framework.Annotate("folds", ndobras);
71
72     acuraciaMedia = 0;
73     for (c=0;c<ndobras;c++){
74         cout << c << "_-" << 100.*v[c][0] << "%\n";
75         acuraciaMedia += v[c][0];
76     }
77     acuraciaMedia /= ndobras;
78 framework.Annotate("measureValue", acuraciaMedia);
79     cout << "*" << acuraciaMedia << "\n";
80
81 framework.setExperimentAlgorithm();
82 framework.setSamplingMethod();
83 framework.setExperimentDataSet();
84
85     return 0;
86 }

```


8 ANEXOS

ANEXO 1: EXPERIMENTO EM PYTHON UTILIZANDO O FRAMEWORK

```
1 import numpy as np
2 from sklearn.datasets import load_iris
3 from sklearn.svm import SVC
4 from sklearn.model_selection import train_test_split
5 from sklearn.model_selection import cross_val_score
6 from sklearn.model_selection import cross_val_predict
7 from sklearn import metrics
8 import mex_framework
9
10
11 @mex_framework.author_name
12 def set_author_name():
13     author = 'Igor_Costa'
14     return author
15
16 @mex_framework.author_email
17 def set_author_email():
18     author_email = 'igor@igor.com'
19     return author_email
20
21 @mex_framework.organization
22 def set_organization():
23     organization = 'Instituto_Militar_de_Engenharia'
24     return organization
25
26 @mex_framework.hardware
27 def set_hardware_info():
28     cache = "CACHE_3MB"
29     cpu = "INTEL_COREI7"
30     hd = "SSD"
31     memory = "SIZE_16GB"
32     os = "MacOS"
```

```

33     return {'cpu': cpu, 'memory': memory, 'hd': hd, 'cache': cache,
34            'os': os}
35
36 iris = load_iris()
37
38 X = iris.data
39 y = iris.target
40
41 @mex_framework.dataset_name
42 def set_dataset_name():
43     name = 'Iris'
44     URI = 'http://archive.ics.uci.edu/ml/datasets/Iris'
45     description = 'This is perhaps the best known database to be
46                  found in the pattern recognition literature.' \
47                  'Fisher\'s paper is a classic in the field and is
48                  referenced frequently to this day. (See Duda &
49                  Hart, for example.)' \
50                  'The data set contains 3 classes of 50 instances
51                  each, where each class refers to a type of iris
52                  plant. One class is linearly
53                  separable from the other 2; the latter are NOT
54                  linearly separable from each other.'
55
56     return {'name': name, 'uri': URI, 'description': description }
57
58
59 #train/test split
60 X_train, X_test, y_train, y_test = train_test_split(X, y, test_size
61                                                    =0.3)
62
63
64 clf = SVC(kernel='linear', C=1).fit(X_train, y_train)
65
66
67 y_pred = clf.predict(X_test)
68 #accuracies = metrics.accuracy_score(y_test, y_pred)
69

```

```

62 clf.fit(X,y)
63
64 new_data = [[1,2,3,4],
65             [4,3,2,1],
66             [4,2,1,3],
67             [3,1,4,2],
68             [2,2,4,4],
69             ]
70
71 predictions = clf.predict(new_data) # 0 - Setosa, 1 - Versicolor, 2
    - Virginica
72 print(predictions)
73
74 predicted = cross_val_predict(clf,X,y)
75 #accuracies = cross_val_score(clf, X, y)
76
77
78 accuracies = metrics.accuracy_score(y, predicted)
79 print(accuracies)
80
81
82 @mex_framework.algorithm
83 def set_algorithm():
84     algorithmID = 'Support_Vector_Machine'
85     algorithmClass = 'SupportVectorMachines'
86     executionType = 'SINGLE'
87     measure = 'ACCURACY'
88     measureValue = accuracies
89     phase = 'TRAIN'
90     return { "algorithmClass": algorithmClass,
91             "algorithmID": algorithmID,
92             "executionType": executionType,
93             "measure": measure,
94             "measureValue": accuracies,
95             "phase": phase}
96
97 @mex_framework.sampling_method

```

```
98 def set_sampling_method():
99     aSM = 'CROSS_VALIDATION'
100     return {'aSM': aSM, 'trainSize':70, 'testSize':30}
101
102
103
104 set_author_name()
105 set_author_email()
106 set_organization()
107 set_dataset_name()
108 set_hardware_info()
109 set_algorithm()
110 set_sampling_method()
```