Universidade Federal de Juiz de Fora
Instituto de Ciências Exatas
Bacharelado em Ciência da Computação

UFJF *Machine Learning Toolkit*
Projeto, desenvolvimento e utilização de um *framework* para algoritmos de aprendizado de máquina

Mateus Coutinho Marim

JUIZ DE FORA
DEZEMBRO, 2018

UFJF *Machine Learning Toolkit*
Projeto, desenvolvimento e utilização de um *framework* para algoritmos de aprendizado de máquina

Mateus Coutinho Marim

Universidade Federal de Juiz de Fora

Instituto de Ciências Exatas

Departamento de Ciência da Computação

Bacharelado em Ciência da Computação

Orientador: Saulo Moraes Villela

Coorientador: Alessandreia Marta de Oliveira Julio

JUIZ DE FORA

DEZEMBRO, 2018

Resumo

A utilização de técnicas de aprendizado de máquina tem se tornado cada vez mais comum devido à extensão dos seus domínios de aplicação e por poderem melhorar o seu desempenho quando expostos a novos dados. Com essa popularidade, diversos métodos foram propostos para abordar problemas da área, trazendo o desafio de comparar diferentes métodos para encontrar o que melhor resolve um problema. *Frameworks* e bibliotecas voltados para algoritmos de aprendizado podem reduzir esse esforço. Este trabalho descreve o UFJF-MLTK, um *framework* orientado a objetos que ajuda na escolha dentre diferentes métodos de aprendizado de máquina e no desenvolvimento de novos algoritmos através da instanciação de classes abstratas pertencentes a uma arquitetura de classes em C++ que abrange vários tipos de algoritmos de aprendizado e também auxilia no ensino do assunto. São discutidos os problemas enfrentados no projeto da arquitetura, os componentes que fazem parte do *framework*, os algoritmos que o compõem atualmente, a forma como foi documentado e exemplos de instanciação do mesmo.

Palavras-chave: Arquitetura de software, Orientação a objetos, Aprendizado de máquina.

Abstract

The use of machine learning techniques has become increasingly common due the extension of their application domains and because they can improve their performance when exposed to new data. With this popularity, several methods have been proposed to address problems of the area, bringing the challenge of comparing different methods to find the one that best solves a problem. Frameworks and libraries focused on learning algorithms can reduce this effort. This work describes the UFJF-MLTK, an object-oriented framework that helps choosing between different machine learning methods, in the development of new algorithms through the instantiation of abstract classes belonging to a C++ class architecture that covers various types of learning algorithms and also helps in teaching the subject. We discuss the problems faced in the architecture project, the components of the framework, the algorithms that currently compose it, how it was documented and examples of its instantiation.

Keywords: Software architecture, Object-oriented programming, Machine learning.

Agradecimentos

Agradeço à minha família, à minha mãe, Cristina Coutinho Marim, e ao meu pai ,Mauro Antônio Marim, por me apoiarem incondicionalmente, independente da distância que nos separa e das nossas divergências, para que eu me tornasse quem sou hoje.

Aos meus amigos, principalmente ao Luiz Gustavo de Paula, que esteve ao meu lado me ajudando nas minhas fases mais difíceis em Juiz de Fora.

Agradeço aos professores Saulo Moraes Villela e Alessandreia Marta de Oliveira Julio pela confiança, paciência e orientação. Sem eles esse trabalho não se realizaria.

Aos professores do Departamento de Ciência da Computação pelos seus ensinamentos, que foram essenciais para minha formação. E aos funcionários do curso, que durante esses anos, contribuíram de algum modo para o nosso enriquecimento pessoal e profissional.

Compreendo essa liberdade de cada um que, longe de deter-se como um limite diante da liberdade alheia, encontra ali, ao contrário, sua confirmação e sua extensão ao infinito; a liberdade ilimitada de cada um pela liberdade de todos, a liberdade pela solidariedade, a liberdade na igualdade; a liberdade, que após derrubar todos ídolos celestes e terrestres, fundará e organizará um novo mundo, aquele da humanidade solidária, sobre as ruínas de todas as igrejas e de todos os estados.

Mikhail Bakunin, *O princípio do estado e outros ensaios*

Conteúdo

Lista de Figuras

Lista de Abreviações

AM	Aprendizado de Máquina
API	*Application Programming Interface*
DCC	Departamento de Ciência da Computação
MLTK	*Machine Learning Toolkit*
UFJF	Universidade Federal de Juiz de Fora

1 Introdução

1.1 Apresentação do tema

Para resolver um problema em um computador é necessário um algoritmo, um conjunto de instruções que diz ao computador como gerar uma saída a partir de uma entrada. Mas para várias tarefas não existe um algoritmo como, por exemplo, fazer a distinção se um e-mail é um *spam* ou um e-mail legítimo (ALPAYDIN, 2010). Este problema poderia ser resolvido através da criação de regras para classificar tais e-mails, mas, além de o custo computacional para esse tipo de solução ser alto, as regras teriam de ser atualizadas de forma periódica. Problemas como este são muito frequentes e, com isso, algoritmos capazes de induzir as regras a partir dos dados de entrada são de grande importância.

Um dos motivos do grande desenvolvimento do aprendizado de máquina é o rápido crescimento de suas áreas de aplicação, como o uso em tecnologias relacionadas à Internet, por exemplo, filtros de spam, sistemas de recomendação e sistemas de detecção de intrusos, que estão sendo utilizados rotineiramente. Está sendo vista hoje a mudança cada vez mais rápida de sistemas especializados para métodos de aprendizado de máquina que estão ficando mais eficientes e dando resultados melhores conforme as pesquisas na área avançam (ALPAYDIN, 2010).

O uso de *frameworks*[1] reunindo ferramentas para o desenvolvimento de novos algoritmos e contendo métodos de aprendizado de máquina já implementados é de grande importância para pesquisadores e desenvolvedores da área, pois pode reduzir o custo de tempo de seus projetos de forma eficiente.

1.2 Problema

Devido à essa grande demanda na área de aprendizado de máquina, muitos algoritmos foram desenvolvidos e aperfeiçoados e, junto a isso, WOLPERT (1996), em seus teoremas

[1] o conceito de *framework* diz respeito a uma arquitetura definida para uma família de aplicações. Tal conceito encontra-se desenvolvido no Capítulo 2 deste trabalho.

apelidados de “Não existe almoço grátis”, chegou a resultados teóricos que concluem que mesmo que um algoritmo consiga um bom resultado em uma classe de problemas, não necessariamente ele vai obter resultados bons no restante do conjunto de problemas. Portanto, não existe um único melhor algoritmo para uma tarefa.

Com o surgimento de novos métodos de aprendizado de máquina e dado os resultados do teorema supracitado, vem a necessidade de se criar ferramentas que permitam pesquisadores e desenvolvedores padronizar suas implementações para execução e reprodução de experimentos. Além disso, tais ferramentas devem conter implementações de algoritmos considerados como estado da arte, dando a possibilidade de fazer comparações para se encontrar os métodos que fornecem os melhores resultados nas bases de dados de interesse.

1.3 Motivação

Ao se pesquisar métodos de aprendizado de máquina, muitas vezes o autor não fornece a implementação dos mesmos ou não há uma padronização do código. Junto a isso também tem o tempo que leva para que o algoritmo esteja disponível em bibliotecas voltadas ao aprendizado de máquina, que pode ser longo, tornando difícil a reprodução de experimentos e atrasando o uso do método em outros projetos.

Nos últimos anos, alunos e professores do Departamento de Ciência da Computação (DCC) da Universidade Federal de Juiz de Fora (UFJF) vêm desenvolvendo vários trabalhos na área de Aprendizado de Máquina. Com isso surgiu a ideia de desenvolver um *framework* que permita a padronização dos algoritmos desenvolvidos em pesquisas acadêmicas, a execução de testes, assim como comparações com os já implementados. Tal artefato também mostra-se relevante em seu potencial pedagógico.

1.4 Objetivos

Este trabalho apresenta o UFJF *Machine Learning Toolkit* (UFJF-MLTK), um *framework* para algoritmos de aprendizado de máquina com os seguintes objetivos:

- Possuir componentes genéricos o suficiente para que possam ser reutilizados em

outros projetos;

- Criar um padrão para os algoritmos de aprendizado de máquina desenvolvidos por professores e alunos do DCC da UFJF;
- Prover pesquisadores e desenvolvedores com ferramentas básicas para a implementação e testes de métodos de aprendizado de máquina;
- Fornecer a implementação de algoritmos de aprendizado de máquina para a utilização em outros projetos;
- Auxiliar o ensino e a aprendizagem do assunto para alunos de graduação e de pós-graduação.

O código do UFJF-MLTK está disponível para quem estiver interessado no repositório do projeto, que está hospedado na plataforma *GitHub*[2][3].

1.5 Organização do texto

O presente trabalho encontra-se dividido em seis capítulos, incluindo este. O Capítulo 2 apresenta descrições, conceitos e resultados sobre *frameworks* e Aprendizado de Máquina. O Capítulo 3 apresenta trabalhos relacionados a *frameworks* para a área de aprendizado de máquina. No Capítulo 4 são dados detalhes sobre o UFJF-MLTK, como as decisões tomadas na implementação, suas características gerais, métodos e algoritmos implementados e como foi feita a documentação do *framework*. No Capítulo 5 são dados exemplos de instanciação por um especialista e por um usuário, a fim de mostrar como é feita a instanciação do *framework* e apresenta a interface de utilização do UFJF-MLTK pela linha de comando. Finalizando, o Capítulo 6 discute as conclusões para este trabalho e faz sugestões de trabalhos futuros.

[2] www.github.com
[3] Acesso ao repositório: https://github.com/mateus558/Machine-Learning-Toolkit

2 Fundamentação Teórica

Neste capítulo são abordados conceitos fundamentais para o entendimento do presente trabalho. Na Seção 2.1 são discutidos conceitos básicos sobre *frameworks* e os benefícios e desafios em sua utilização. Na Seção 2.2 são apresentadas definições de Aprendizado de Máquina, que é o domínio de aplicação do *framework*, de algoritmos de aprendizado e os tipos de algoritmos de aprendizado de máquina que são abordados pelo *framework*.

2.1 *Framework*

Os autores (FAYAD et al., 1999, apud OLIVEIRA (2011)) definem um *framework* como uma arquitetura para uma família de subsistemas oferecendo construtores básicos para criá-los. Os pontos de extensão são adaptações de código que dizem como o funcionamento específico de certos módulos devem ser feitos.

Um *framework* fornece ao seu utilizador implementações reusáveis de classes abstratas e concretas. As classes abstratas implementam parte da abstração do *framework*, deixando as decisões cruciais de implementação para a criação de subclasses. As classes concretas fazem a implementação de suas operações para que sejam instanciadas sem a necessidade da criação de sub-classes (RIEHLE, 2000).

As classes abstratas, que são os pontos de flexibilidade, são chamados de *hot spots*. Para a geração de um executável, o *framework* deve ser instanciado com a implementação de código específico da aplicação em cada um dos *hot spots*. As classes concretas são chamadas de *frozen spots*, que são os pontos fixos de imutabilidade, e constituem o núcleo do *framework* (MARKIEWICZ; LUCENA, 2001).

O funcionamento de um *framework* pode ser pensado como análogo ao de um motor, assim como mostrado na Figura 2.1. Ao contrário de um motor convencional, um *framework* tem várias entradas de energia. Cada uma dessas entradas é um *hot spot* que deve ser implementado para o motor (*framework*) funcionar. Os geradores de energia são os códigos específicos da aplicação que devem ser ligados nos *hot spots*. O código

adicionado na aplicação vai ser usado pelo código do núcleo do *framework*. O motor não vai funcionar até que todas tomadas sejam ligadas (MARKIEWICZ; LUCENA, 2001).

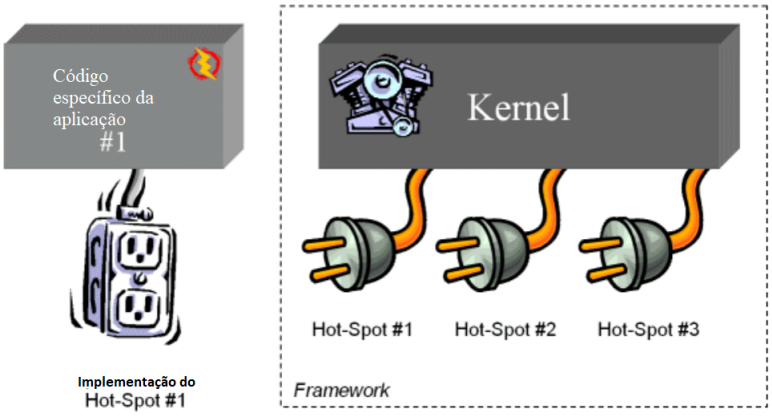

Figura 2.1: Analogia de um *framework* com um motor. Adaptado de (MARKIEWICZ; LUCENA, 2001).

2.1.1 Tipos de *framework*

Segundo (FAYAD et al., 1999, apud MARKIEWICZ; LUCENA (2001)), um *framework* pode ser classificado por sua extensibilidade. Em *frameworks* caixa-branca, a instanciação somente é possível através da criação de novas classes introduzidas no *framework* através de herança ou composição. O utilizador desse tipo de *framework* deve entender muito bem a arquitetura do mesmo para produzir uma instância. Já os *frameworks* caixa-preta, esses são instanciados através de *scripts* de configuração seguido de uma ferramenta de instanciação automatizada que cria as classes e código fonte. Geralmente esse tipo de *framework* é o mais fácil de usar. Os *frameworks* com as características de ambos os tipos são chamados de *frameworks* caixa-cinza. A Figura 2.2 mostra uma representação desses tipos de *framework*.

2.1.2 Desafios da utilização

Frameworks são de difícil desenvolvimento, pois necessitam alto grau de reuso e flexibilidade, que são atributos de difícil alcance em um *software* e, portanto, elevam o custo de um projeto em relação a um sem reuso. Apesar disso, os benefícios da construção deste tipo de artefato justificam o esforço necessário para seu desenvolvimento (FAYAD et al.,

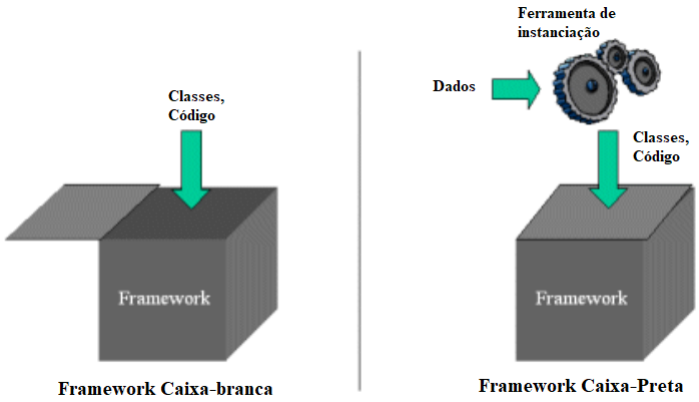

Figura 2.2: *Framework* caixa-branca e caixa-preta. Adaptado de (MARKIEWICZ; LUCENA, 2001).

1999, apud OLIVEIRA (2011)).

Um problema comum na utilização de *frameworks* é a sua curva de aprendizado íngreme (RIEHLE, 2000). Alguns *frameworks* são tão complexos que um programador pode levar cerca de seis a doze meses, dependendo de sua experiência, para serem bem compreendidos e utilizados de maneira produtiva (FAYAD et al., 1999, apud OLIVEIRA (2011)). Também não existem diretrizes bem definidas para a documentação de um *framework*. *Frameworks* mal documentados vão ter poucos utilizadores e mantenedores ao longo de sua existência, portanto, a documentação deve ter ao menos um guia de como estendê-la e/ou ferramentas de instanciação (MARKIEWICZ; LUCENA, 2001).

Outro grande desafio é a manutenção: como o *framework* tem seu código presente em vários subsistemas, uma modificação na sua estrutura pode causar o mal funcionamento de seus sistemas dependentes ou até criando aplicações órfãs que utilizam uma versão antiga do *framework* para poder funcionar (OLIVEIRA, 2011).

2.1.3 Benefícios da utilização

O uso de *frameworks* permite maior produtividade e menor custo de tempo para criar aplicações prontas para o mercado através do projeto de código reusável. Utilizando conceitos de Orientação a Objetos um desenvolvedor pode modelar um domínio, definir uma arquitetura de software para representá-lo no computador e implementar a arquitetura para que este execute o modelo (RIEHLE, 2000).

Projetos que apresentam um tipo de problema em comum podem ter seu tempo de desenvolvimento reduzido, uma vez que desenvolvedores são providos de ferramentas que lhes fornecem flexibilidade para adequarem o *framework* às suas necessidades. A quantidade de código desenvolvido é reduzida quando fragmentos de código que são comuns a uma família de problemas são incluídos no *framework*, fazendo com que a implementação seja feita apenas uma vez, reduzindo a quantidade de código em relação a um projeto sem reuso (OLIVEIRA, 2011).

A tarefa da criação de um *framework* não é viável quando o desenvolvedor precisa fazer apenas uma instância do mesmo, mas o esforço vale quando é necessário fazer várias instâncias para aplicações pertencentes a mesma classe de problemas, principalmente por não ser necessário repetir o código toda vez que for criar soluções para problemas relacionados. A utilização de um *framework* em um projeto facilita a manutenção do sistema sendo desenvolvido, pois é possível testar os componentes que pertencem ao projeto separadamente e a maior parte das decisões de projeto e implementação se encontram no núcleo do *framework*. Normalmente os *frameworks* são criados por especialistas no domínio e por vários desenvolvedores em conjunto, portanto, os sistemas derivados tendem a ser mais estáveis (OLIVEIRA, 2011).

2.1.4 Construção de um *framework*

Embora não exista uma metodologia específica para a construção de *frameworks*, podem-se classificar as abordagens utilizadas em duas categorias: *top-down* e *bottom-up*. A abordagem *top-down* parte da etapa de análise de domínio, onde são esclarecidos os requisitos comuns e não comuns de toda uma classe de aplicações. Pode-se utilizar, para essa abordagem, qualquer metodologia orientada a objetos baseada em domínios (MATHIAS FILHO; LUCENA, 2002). A Figura 2.3 demonstra essa abordagem.

A abordagem *bottom-up* parte do princípio, largamente difundido e aceito na comunidade de orientação a objetos, de que o desenvolvimento de *frameworks* precisa de muita experiência e experimentação do domínio. Por isso, *frameworks* construídos com esta abordagem utilizam várias aplicações pré-existentes do domínio que abordam, para produzir depois de várias iterações, um design genérico a partir dos pontos comuns e não

comuns das aplicações (MATHIAS FILHO; LUCENA, 2002).

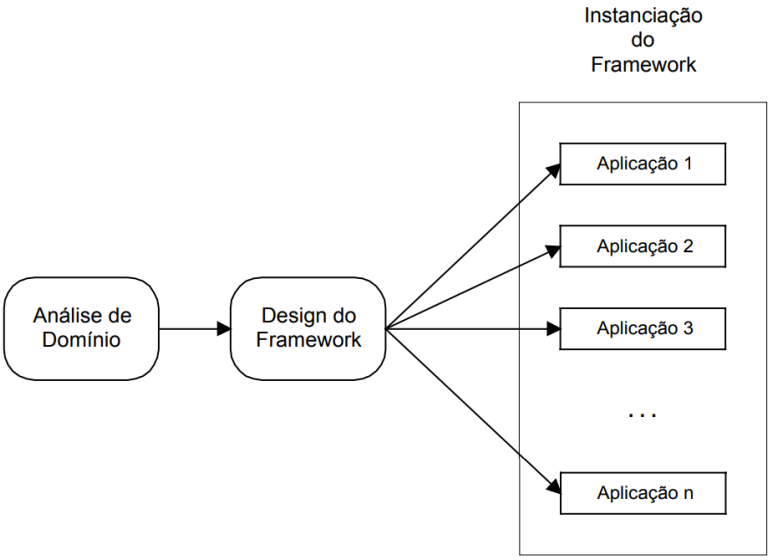

Figura 2.3: Desenvolvimento de *frameworks* (MATHIAS FILHO; LUCENA, 2002).

2.1.5 Documentação de um *framework*

Como um *framework* cobre muito da complexidade do domínio de aplicação, que deve ser descrito e entendido, surge a necessidade de uma descrição clara do *framework*. A documentação do *framework* deve ser descrita em diferentes níveis de abstração por ter que satisfazer a necessidade de desenvolvedores com diferentes níveis de experiência (MATTSON, 1996). A documentação de um *framework* pode ser categorizada em dois tipos, cada um deles voltado a um grupo de desenvolvedores específicos. O primeiro é voltado para os projetistas do *framework* e tem como objetivo documentar todo seu processo de *design*. A segunda categoria é voltada para os desenvolvedores de aplicação. A documentação para a segunda categoria deve ser projetada de tal forma que dê suporte na seleção do *framework* adequado para abordar o problema em questão e a sua posterior reutilização para instanciação de uma aplicação (MATHIAS FILHO; LUCENA, 2002).

Segundo JOHNSON (1992) a documentação de um *framework* deve descrever:

- O propósito: deve ser o primeiro item a ser descrito. Se o *framework* não for apropriado, o leitor não precisa continuar lendo;
- Como usar o *framework*: a documentação deve mostrar ao usuário como construir

aplicações. A maioria dos usuários não estão interessados na descrição do *design* do *framework*, mas sim em um tipo de *cookbook* dando instruções detalhadas de como utilizar o *framework*;

- O *design* detalhado: isso não inclui apenas uma descrição das diferentes classes que o compõem, mas também como as suas instâncias colaboram. Apesar de programadores poderem interconectar objetos sem compreenderem completamente o seu funcionamento e até criar subclasses seguindo um *cookbook*, um *framework* é melhor aproveitado por alguém que o conhece em detalhes.

2.1.6 Interface de Programação de Aplicações

Uma Interface de Programação de Aplicações (*Application Programming Interface* – API) fornece um padrão de como um cliente deve interagir com os componentes de um *software* que fornecem a implementação da solução de um problema. Uma API tem como propósito fornecer uma interface lógica aos componentes enquanto esconde os detalhes de implementação (REDDY, 2011). Em relação a um *framework*, uma API é a forma como são acessados os componentes para a criação de uma aplicação por um desenvolvedor, ou seja, a parte pública do *framework* que permite sua instanciação. Segundo REDDY (2011) uma API está especificada de forma incompleta, a não ser que venha com uma documentação, pois os cabeçalhos vindos com a API definem apenas as convenções de chamada para os métodos e funções e não o comportamento da mesma, logo, uma boa documentação é um elemento crítico para uma API.

2.2 Aprendizado de máquina

Desde a invenção dos computadores, sempre nos perguntamos se eles poderiam aprender. Se pudéssemos entender como programar computadores que aprendem, o impacto na vida humana seria enorme. O campo de Aprendizado de Máquina é o que está interessado em responder tal questão (MITCHELL, 1997).

2.2.1 Algoritmos de aprendizado

Programas de computador aprendendo quais são os melhores tratamentos para novas doenças, casas otimizando o consumo de energia pelo padrão de uso de seus moradores e até softwares se adaptando a utilização do usuário. O entendimento de algoritmos de aprendizado de máquina elevaria a um novo nível a utilização de computadores e, além disso, um entendimento detalhado do seu funcionamento nos daria uma melhor compreensão das capacidades e deficiências do aprendizado humano (MITCHELL, 1997).

Um algoritmo é dito ser de aprendizado quando, dado um conjunto de dados, é capaz de criar um modelo representando o conhecimento extraído desses dados e, a partir desse modelo, consegue com um certo grau de confiabilidade ou acurácia prever informações de novos dados. Uma definição mais formal é dada por MITCHELL (1997) a seguir:

> Um programa de computador é dito estar aprendendo a partir da experiência E com respeito a alguma classe de tarefas T e medida de performance P, se sua performance nas tarefas em T, como mensuradas em P, melhoram com experiência E (MITCHELL, 1997).

Existem diversos tipos de algoritmos de aprendizado. Nesta Seção são definidas apenas os paradigmas de aprendizado que são atualmente abordados pelo *framework* proposto, entre eles tem-se o aprendizado supervisionado, o não supervisionado e o semissupervisionado.

2.2.2 Aprendizado supervisionado

No aprendizado supervisionado o algoritmo recebe pares de entrada-saída e, a partir dessa entrada, aprende uma função que mapeia a entrada para a saída. Pode-se definir a tarefa do aprendizado supervisionado matematicamente como: Dado um conjunto de treinamento com m pares de entrada-saída definidos pelo conjunto $Z = \{(x_1, y_1), \ldots, (x_m, y_m)\}$, sendo que x e y podem ser de quaisquer valores, em que cada y foi gerado por uma função desconhecida $y = f(x_i)$, se quer encontrar uma função h, também chamada de hipótese, que se aproxime da função real f. O aprendizado consiste na busca de uma hipótese que tenha um bom desempenho mesmo em um conjunto de exemplos fora do conjunto de treinamento. Para a avaliação da precisão da hipótese, normalmente é dado um conjunto

de testes de exemplos que são diferentes do conjunto de treinamento. Uma hipótese é considerada com uma boa generalização, quando ela prevê de forma correta o valor de y de novos exemplos (RUSSEL; NORVIG, 2009).

Problema de classificação

Em instituições financeiras, um problema muito comum é o cálculo do risco da aplicação de um cliente para um empréstimo, com dados sobre o cliente, seu histórico de empréstimos passados e a informação de se foram aceitos ou não, um sistema de aprendizado de máquina ajusta um modelo nos dados passados capaz de prever o risco e decidir se a aplicação deve ser aceita ou não (ALPAYDIN, 2010).

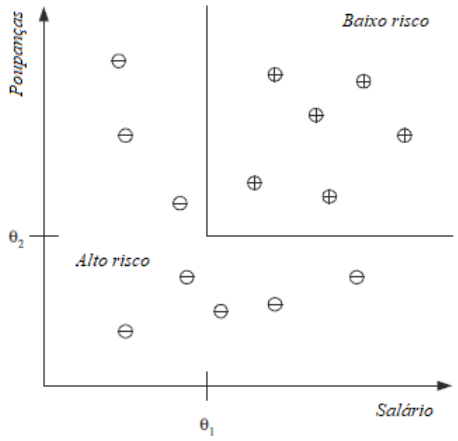

Figura 2.4: Exemplo do problema do cálculo de risco da aplicação de um cliente para um empréstimo. Adaptado de ALPAYDIN (2010).

Esse é um problema de classificação onde existem duas classes: clientes de baixo e alto risco. A informação dos dados passados do cliente é a entrada do algoritmo, ao qual, é dada a tarefa de associar a entrada a um dos dois rótulos. Na figura 2.4 podemos ver um exemplo desse problema. Por simplicidade, existem apenas dois atributos para os clientes dados como entrada, salário e poupança, representados por θ_1 e θ_2 respectivamente, e as duas classes são baixo risco ('-') e alto risco ('+') Também é mostrado um exemplo de discriminante separando os dois tipos de exemplos (ALPAYDIN, 2010).

Classificação binária

O exemplo dado anterior é chamado de problema de classificação binária, que pode ser definido como se segue. Seja um conjunto de exemplos $Z = (x_i, y_i)$ de tamanho m, onde $x_i \in R^d$, chamado de espaço de entrada do problema, y_i é um escalar representando a classe de cada vetor x_i e para classificação binária $y_i \in \{+1, -1\}$, para $i = \{1, \ldots, m\}$. Um classificador linear, em um espaço de entrada linearmente separável, é representado por um hiperplano com a seguinte equação (VILLELA; XAVIER; NETO, 2011):

$$h(x) = \langle w, x \rangle + b. \tag{2.1}$$

Pode-se obter uma forma mais geral desse classificador passando cada ponto do espaço de entrada para um espaço chamado de espaço de características ou $\phi - space$. Pode-se considerar nesse sentido, a inserção do bias no vetor w, e também o acréscimo de um componente adicional $+1$ no vetor representativo de cada ponto. Com a equação geral tomando a forma (VILLELA; XAVIER; NETO, 2011):

$$h(x) = \langle w, \phi(x) \rangle . \tag{2.2}$$

O resultado da classificação pode ser obtida através de uma função sinal φ aplicada ao valor do discriminante associado à equação do hiperplano, ou seja (VILLELA; XAVIER; NETO, 2011):

$$\varphi(h(x)) = +1 \quad \text{se} \quad h(x) \geq 0 \quad \text{ou} \quad \varphi(h(x)) = -1 \quad \text{se} \quad h(x) < 0. \tag{2.3}$$

Na Figura 2.5 pode-se ver um exemplo de classificação binária com discriminante linear.

Problema de regressão

Seja o problema de prever o preço de um carro usado, cujos atributos são o modelo, ano, motor, a quilometragem e outras informações que acredita-se que afetam o valor do carro. O valor desejado é o preço do carro. A esses problemas em que o valor a ser previsto é um

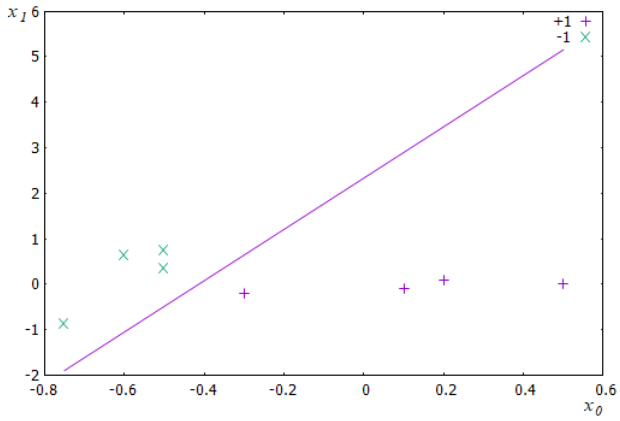

Figura 2.5: Exemplo do problema de classificação binária.

número real, é dado o nome de problema de regressão. Seja x os atributos e y os preços do carro, analisando novamente as transações passadas, deseja-se obter um conjunto de treinamento $Z = (x_i, y_i)$ para que o algoritmo de aprendizado de máquina possa ajustar uma função para aprender y em função de x. Na Figura 2.6 tem-se um exemplo em que a função ajustada é da seguinte forma (ALPAYDIN, 2010):

$$y = w \cdot x + b \tag{2.4}$$

Em alguns casos o modelo linear é muito restritivo e pode-se, por exemplo, usar a equação quadrática ou qualquer outra função não linear que se ajuste aos dados (ALPAYDIN, 2010).

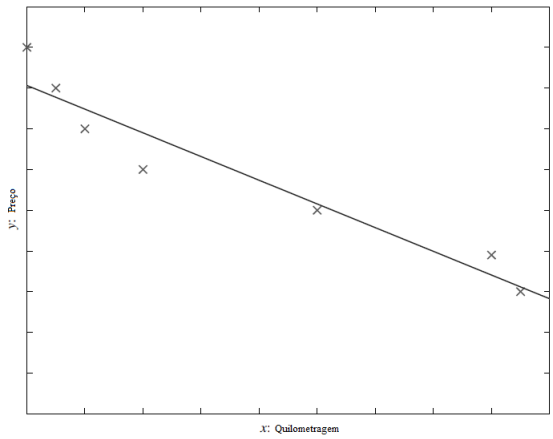

Figura 2.6: Exemplo do problema de regressão. Adaptado de ALPAYDIN (2010).

Métodos *Kernel*

É frequente não ser possível fazer uma separação linear na prática. Nesses casos é necessário a utilização de funções mais complexas para separar as classes, como mostrado na Figura 2.7. Uma maneira de se definir um separador não linear é através de uma função de mapeamento do espaço de entrada X para um espaço de maior dimensão onde a separação é possível (MEHRYAR et al., 2012). Em modelos que são baseados em um mapeamento do espaço de características não linear fixo $\phi(x)$, a função *kernel* é definida a seguir (BISHOP, 2006):

$$k(x, x') = \phi(x)^T \phi(x'). \tag{2.5}$$

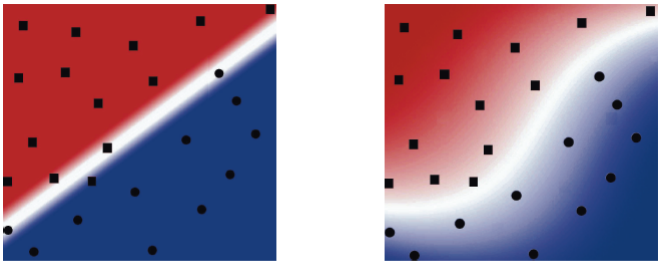

Figura 2.7: Exemplo de separação não linear. A tarefa de classificação consiste em discriminar entre círculos e quadrados. (a) Um hiperplano não consegue separar as duas classes. (b) Um mapeamento não linear pode ser usado no lugar (MEHRYAR et al., 2012).

O caso de método *kernel* mais simples considerando o mapeamento na equação 2.5 é o *kernel* linear em que $\phi(x) = x$ e $k(x, x') = x^T x$. O conceito de *kernel* formulado como um produto interno no espaço de entrada permite fazer a generalização de muitos algoritmos conhecidos. Um muito popular proposto por VAPNIK at al. (1995) é o SVM Dual. A ideia principal é que se um algoritmo é formulado de tal maneira que o vetor de entrada x se apresenta na forma de produto escalar, pode-se substituir o produto interno por outro *kernel*. Esse tipo de extensão é conhecido como truque *kernel* ou substituição de *kernel* (BISHOP, 2006).

2.2.3 Aprendizado não supervisionado

No aprendizado não supervisionado o algoritmo recebe um conjunto de treinamento $Z = \{x_1, \ldots, x_n\}$, mas não recebe nenhum rótulo relacionado aos dados. O objetivo é encontrar

uma descrição plausível para os dados. De um ponto de vista probabilístico, deseja-se modelar a distribuição $p(x)$. Uma medida de avaliação da descrição é a probabilidade do modelo gerar os dados (BARBER, 2012).

2.2.4 Aprendizado semissupervisionado

O aprendizado semissupervisionado está entre o supervisionado e o não supervisionado. Nele pode-se encontrar dados não rotulados junto com informações de supervisão, mas não necessariamente para todos exemplos. No caso em que tem-se a função objetivo associada com parte dos exemplos, pode-se separar o conjunto de treinamento em $Z_l = \{(x_1, y_1), \ldots, (x_l, y_l)\}$, para o qual os rótulos foram fornecidos, e $Z_u = \{x_{l+1}, \ldots, x_{l+u}\}$, onde os rótulos não são conhecidos (CHAPELLE; SCHLKOPF, 2010).

2.2.5 Seleção de características

O processo de seleção de características se baseia na seleção de um subconjunto de atributos dos dados do conjunto original com a menor perda na geração de resultados (VILLELA; XAVIER; NETO, 2011). A presença de características sem correlação com as classes servem apenas como ruído, introduzindo um bias no preditor e reduzindo a performance do algoritmo de classificação. Logo, a utilização de um método de seleção de características pode reduzir o custo computacional e aumentar a acurácia do método de aprendizado de máquina (CHANDRASHEKAR; SAHIN, 2011).

A seguir são apresentados três tipos de métodos de seleção de características: em filtro, embutidos e *wrapper*.

Métodos de seleção em filtro

Os métodos de seleção em filtro são processos aplicados antes dos algoritmos de aprendizado e tem como objetivo remover características com pouca relevância através de algum critério de ranqueamento (VILLELA; XAVIER; NETO, 2011). Como exemplo, tem-se os método de seleção de Golub (GOLUB et al., 1999).

Métodos de Seleção Embutidos

Métodos de seleção embutido tem como sub-rotina um algoritmo de classificação e utilizam a sua performance como função objetivo para avaliar um subconjunto de características. Um exemplo de método embutido é a utilização do IMA_p na sua formulação L_∞. Nessa formulação o algoritmo faz a minimização da norma L_1 do vetor w, selecionando os componentes do vetor de maior valor como características e eliminando as de menor valor, ou seja, as de menor relevância (VILLELA; XAVIER; NETO, 2011).

Métodos de seleção *Wrapper*

Nessa abordagem são gerados vários subconjuntos de atributos por um algoritmo de busca que utiliza um classificador como função de avaliação do subconjunto. A principal vantagem desses métodos é a dependência entre o algoritmo de seleção e aprendizado. Mas, apesar disso, o custo computacional é muito caro pois a busca deve executar o algoritmo de aprendizado para cada subconjunto de atributos gerado. Podem ser citados como exemplo desses métodos o RFE e o AOS. A utilização dessa técnica é justificada pelas suas vantagens, como a diminuição da dimensionalidade do problema original pela remoção de atributos irrelevantes, que não influenciam no resultado final e redundantes, que são altamente correlacionados não agregam informação (VILLELA; XAVIER; NETO, 2011).

2.3 Considerações finais

Este capítulo apresentou os principais conceitos necessários para a compreensão desse trabalho. *Frameworks* são arquiteturas definidas para uma família de aplicações oferecendo construtores básicos para criá-lo. As classes de um *framework* são divididas em dois tipos: as concretas e abstratas, onde as classes abstratas são chamadas de pontos de extensão e permitem a instanciação do *framework*. Um *framework* é dito ser do tipo caixa-branca quando o utilizador necessita conhecer a arquitetura da mesma para realizar sua insanciação através da herança dos pontos de extenção. Enquanto que os do tipo caixa-preta executam *scripts* de configuração seguido de uma ferramenta de instanciação

automátizada para criação das classes e do código fonte. Devido ao seu alto grau de reuso, o desenvolvimento de um *framework* é um processo difícil, mas devido aos seus benefícios, o esforço em sua construção é justificado.

Os benefícios da construção desse artefato incluem a maior produtividade no desenvolvimento de aplicações, flexibilidade na adequação do *framework* as suas necessidades e redução da quantidade de código quando fragmentos de código de uma família de problemas são incluídos no mesmo. A construção de um *framework* pode seguir a abordagem *bottom-up*, em que sua construção é baseada em aplicações pré-existentes do domínio atacado pelo *framework*, e a *top-down*, em que é feita uma análise do domínio do problema e são esclarecidos os requisitos comuns e não comuns de toda uma classe de aplicações. A documentação de um *framework* deve descrever o seu propósito, como é feita a sua utilização e deve conter uma descrição detalhada do seu *design*.

O campo de Aprendizado de Máquina está interessado em responder a questão de se os computadores são capazes de aprender. Um algoritmo é dito de aprendizado de máquina se ele é capaz de melhorar o seu desempenho a partir de uma experiência, sendo a experiência, uma informação disponível previamente ao algoritmo. Os tipos de algoritmo de aprendizado abordados pelo *framework* são os de aprendizado supervisionado, não supervisionado e o semissupervisionado. No aprendizado supervisionado o algoritmo recebe pares de entrada-saída e aprende uma função que mapeia a entrada para a saída. Pode-se encontrar no aprendizado supervisionado dois tipos de problema, os problemas de classificação, onde dado um conjunto de treinamento com seus devidos rótulos discretos (classes), deseja-se ajustar um modelo capaz de prever os rótulos de dados desconhecidos, e os problemas de regressão, que são similares aos de classificação com exceção de que os rótulos são números reais. No aprendizado não supervisionado, não são fornecidos os rótulos aos dados e se deseja encontrar uma descrição plausível para os dados. O aprendizado semissupervisionado se situa entre o supervisionado e o não supervisionado, nele pode-se encontrar dados não rotulados junto com informações de supervisão, mas não necessariamente para todos exemplos.

O processo de seleção de características se baseia na seleção de um subconjunto de atributos que melhor representa o conjunto original com a menor perda na generalização

dos resultados. Os métodos de seleção de características são de três tipos: em filtro, embutidos e *wrapper*. Os métodos de seleção em filtro utilizam um critério de ranqueamento para remover os atributos vom pouca relevância. Os embutidos utilizam a performance de um algoritmo de classificação como função objetivo para avaliar um subconjunto de características. E nos métodos *wrapper* são gerados vários subconjuntos de atributos por um algoritmo de busca que utiliza um classificador como função de avaliação do subconjunto. O próximo capítulo descreve trabalhos relacionados a *frameworks* de aprendizado de máquina.

3 Trabalhos Relacionados

Nas últimas décadas, houve vários esforços para se reunir algoritmos e ferramentas de aprendizado de máquina para auxiliar no desenvolvimento de aplicações que abordam problemas da área. Este capítulo descreve de forma sucinta quatro sistemas que fornecem ferramentas de aprendizado de máquina ou têm objetivos similares aos do UFJF-MLTK.

3.1 MLC++

O MLC++ (KOHAVI et al., 1994) que é um sistema criado pela Universidade de Stanford para problemas de classificação, foi um dos pioneiros na tentativa de reunir ferramentas e algoritmos de aprendizado de máquina em um lugar só. Seu objetivo é facilitar a tarefa de comparação de diversos algoritmos de classificação para uma base de dados e, também, fornecer uma vasta coleção de classes em C++ para ajudar no desenvolvimento de novos algoritmos. Inicialmente o seu código se encontrava em domínio público, mas não foi possível encontrar a sua implementação para uso na fonte dada pelo artigo. Uma das limitações do MLC++ é a sua abrangência, por trabalhar apenas com algoritmos de classificação ele não é adequado para grande parte dos problemas de aprendizado de máquina.

3.2 *Java Intelligent Optimisation* (JIOP)

O JIOP (HATLEDAL; SANFILIPPO, 2014) é um *framework* orientado a objetos desenvolvido em Java. Inicialmente, o seu desenvolvimento se deu para que fosse uma coleção de métodos de otimização computacional já existentes, mas seu principal objetivo é de ser usado para propósitos pedagógicos por alunos de graduação e pós-graduação permitindo que os alunos usem algoritmos de otimização, os combinem, os estendam e até mesmo criem novos algoritmos. O JIOP contém uma classe abstrata do qual todos os algoritmos implementados são derivados. O *framework* tem os seguintes algoritmos já implementa-

dos: algoritmo genético, *simulated annealing*, evolução diferencial, otimização por enxame de partículas e colônia artificial de abelhas. Apesar disso, o mesmo não contém implementações de algoritmos considerados "clássicos" de aprendizado de máquina.

3.3 Dlib-ml

O Dlib-ml (KING, 2009) é uma biblioteca desenvolvida em C++ de algoritmos de aprendizado de máquina e de código aberto. A biblioteca é composta por dois componentes principais, um de ferramentas de aprendizado de máquina e outro de álgebra linear. Um dos objetivos principais do Dlib-ml é fornecer uma arquitetura altamente modular e simples para lidar com algoritmos que utilizam funções *kernel*, permitindo a solução de problemas não linearmente separáveis, cada algoritmo é parametrizado de forma que permita ao usuário escolher uma das funções *kernel* pré-definidas ou um *kernel* customizado. A biblioteca provê implementações de algoritmos populares de classificação e regressão. O Dlib-ml oferece uma interface genérica o suficiente para torná-la facilmente extensível por outros desenvolvedores. Apesar de ser uma boa biblioteca de aprendizado de máquina, ainda carece de uma ferramenta para visualização, que é importante para o pré-processamento dos dados. Pelo Dlib-ml apenas implementar os algoritmos mais populares, ele não permite a comparação com algoritmos de aprendizado de máquina recentemente pesquisados, fazendo com que pesquisadores tenham de esperar muito tempo até que estes algoritmos estejam disponíveis ou que tenham de fazer as suas próprias implementações.

3.4 Tensorflow

O Tensorflow (ABADI et al., 2016) é um sistema para aprendizado de máquina em larga escala desenvolvido pelo time da *Google Brain* e com projeto de código aberto. O Tensorflow está disponível nas linguagens Python, Java, C++ e Go. Uma de suas principais características é a utilização de um grafo de fluxo de dados para as computações. O grafo de fluxo de dados expressa a comunicação entre subcomputações de forma explícita, tornando assim fácil executar computações independentes em paralelo. O Tensorflow permite

que os vértices representem as computações que possuem ou atualizam estados mutáveis. As arestas transportam *tensors* (vetores multidimensionais) entre os nós, e o Tensorflow insere de forma transparente as subcomputações entre subcomputações distribuídas, permitindo assim, que programadores experimentem diferentes esquemas de paralelização. Por ter sido desenvolvido para ser usado por desenvolvedores profissionais e por utilizar de vários conceitos complexos de programação como o paralelismo, o Tensorflow carece de uma linguagem simples, não sendo recomendado para quem está apenas começando na área de aprendizado de máquina.

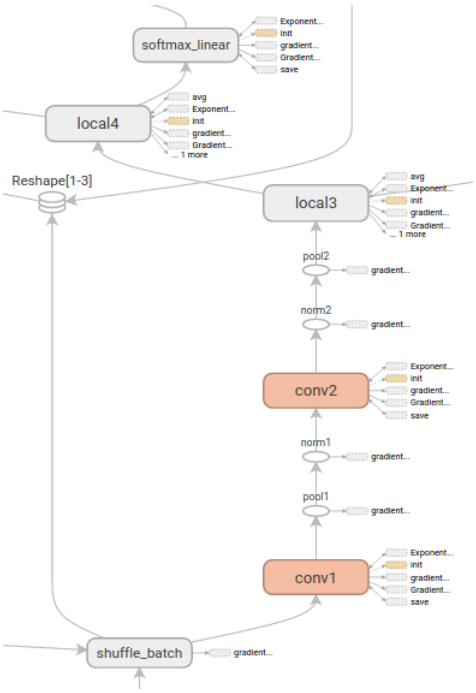

Figura 3.1: Visualização de um grafo de computação do Tensorflow.

3.5 Weka

O Weka (HALL et al, 2009) é um *workbench* com o objetivo de prover pesquisadores e praticantes de aprendizado de máquina com algoritmos de aprendizado de máquina e ferramentas de pré-processamento de dados. Permite que os usuários rapidamente testem e comparem diferentes métodos de aprendizado de máquina em novas base de dados. O principal diferencial do Weka é a sua interface gráfica, que permite o acesso fácil as suas funcionalidades internas. O Weka tem uma API simples, mecanismos de *plugins* e facilidades que automatizam a integração de novos algoritmos de aprendizado a sua

interface gráfica. O Weka foi desenvolvido na linguagem Java e inclui algoritmos para regressão, classificação, agrupamento, mineração de regras de associação e seleção de características. Também é possível fazer exploração dos dados com a sua ferramenta de visualização de dados e com as suas ferramentas de pré-processamento de dados. Assim como o Scikit-learn, a sua principal limitação é estar disponível apenas na linguagem em que foi implementado.

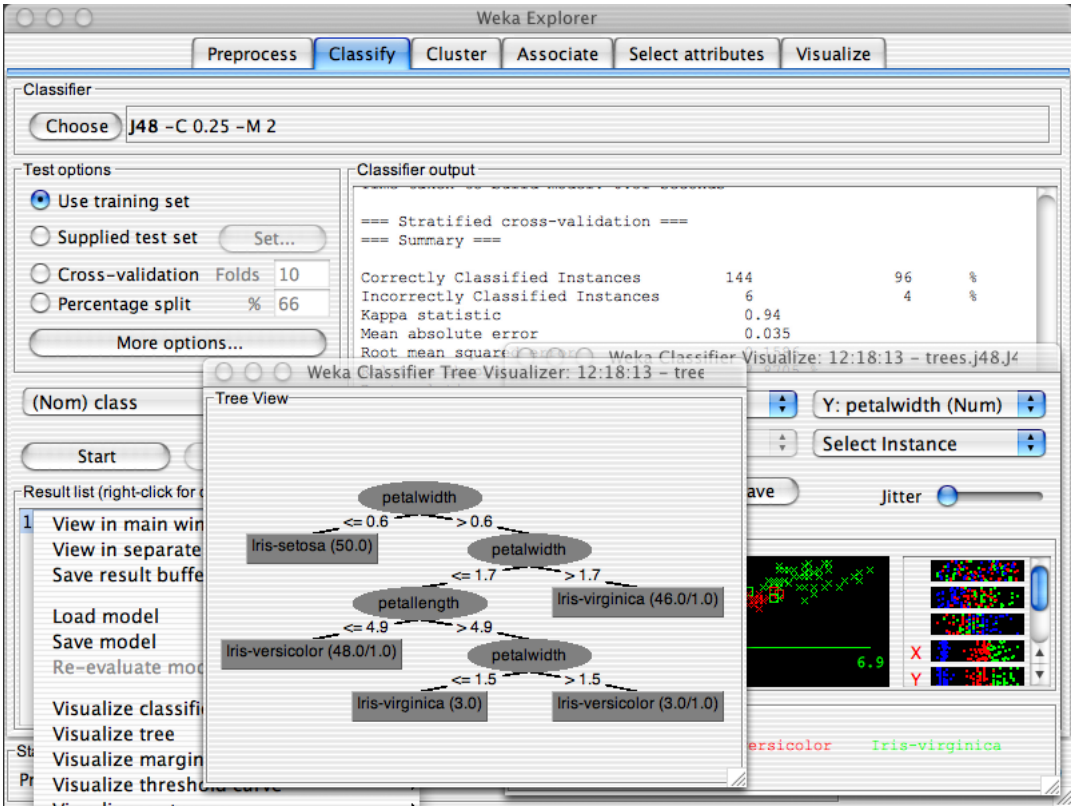

Figura 3.2: Painel para problemas de classificação do Weka.

3.6 LIBSVM

O LIBSVM (CHANG; LIN, 2011) é uma biblioteca para a utilização de máquinas de vetor suporte (*Support Vector Machines* - SVM). O uso típico do LIBSVM envolve dois passos, primeiro, o treinamento de uma base de dados para obter um modelo e segundo, a utilização do modelo para prever informações de uma base de dados de teste. No início de sua criação no ano de 2000 o LIBSVM comportava apenas o C-SVM para duas classes, com o passar do tempo foram incluídas outras variações do SVM e atualmente se tornou um pacote SVM completo. Assim como o JIOP, o LIBSVM é capaz apenas com um

subconjunto dos problemas de aprendizado de máquina, podendo resolver apenas uma parte dos problemas de classificação e regressão.

3.7 Scikit-learn

O Scikit-learn (PEDREGOSA et al., 2011) é um módulo desenvolvido para a linguagem Python, uma das linguagens mais populares para computação científica (PEDREGOSA et al., 2011), com a implementação de muitos algoritmos conhecidos e considerados estado da arte e mantendo uma interface fácil de se usar. O Scikit-learn entrega o que é prometido tendo dependência com poucos módulos do Python e abrange uma grande parte dos tipos de algoritmos de aprendizado. Apesar disso, o módulo só pode ser utilizado na linguagem Python, tornando o seu público alvo limitado.

3.8 Considerações finais

Todos esses trabalhos compartilham objetivos com o UFJF-MLTK, tendo suas vantagens e desvantagens. As bibliotecas que focam no uso profissional, geralmente tem uma linguagem de difícil aprendizado, tornando sua utilização por estudantes penosa. Enquanto que os que seguem um objetivo mais pedagógico acabam se limitando a apenas um subconjunto pequeno dos problemas possíveis em aprendizado de máquina. O objetivo do UFJF-MLTK é um meio termo entre esses apresentados por, apesar de não ser recomendado para aplicações em larga escala, abranger boa parte dos problemas de aprendizado de máquina sem dificultar o entendimento de sua utilização, sendo muito útil no auxílio do ensino de aprendizado de máquina. A maioria dos trabalhos apresentados foram criados em linguagens como o Python e o Java, sendo que os implementados em C++ apresentam o problema de se limitarem a apenas um subconjunto dos tipos de algoritmos de aprendizado ou carecem de alguma ferramenta importante no processo da aplicação de técnicas de aprendizado de máquina. Muitos deles implementam apenas os algoritmos mais populares, não dando oportunidade para os algoritmos recentemente pesquisados. O UFJF-MLTK é um *framework* desenvolvido em C++ que tenta cobrir essas necessidades e é detalhado no próximo capítulo. A Tabela 3.8 sintetiza os principais pontos discutidos

Tabela 3.1: Tabela com os principais pontos discutidos sobre os trabalhos relacionados.

	Linguagem	**Tipos de aprendizado suportados**	**Fornece uma ferramenta de visualização**	**Propósito**
MLC++	C++	Supervisionado	Sim	Acadêmico, indústrial
JIOP	Java	Nenhum (Inteligência computacional)	Sim	Acadêmico
Dlib-ml	C++	Todos	Não	Acadêmico, indústrial
Tensorflow	Python	Todos	Sim	Acadêmico, indústrial
Weka	Java	Todos	Sim	Acadêmico, industrial
LIBSVM	C++	Supervisionado	Não	Acadêmico, indústrial
Scikit-learn	Python	Todos	É possível através de pacotes de terceiros.	Acadêmico, indústrial

sobre os trabalhos deste capítulo.

4 UFJF *Machine Learning Toolkit*

Este capítulo apresenta o UFJF *Machine Learning Toolkit*, um *framework* orientado a objetos desenvolvido em C++ para algoritmos de aprendizado de máquina, com o objetivo de prover ferramentas e métodos de aprendizado de máquina para desenvolvedores e pesquisadores, junto também com um padrão definido para a implementação de novos métodos. O UFJF-MLTK alcança esses objetivos mantendo uma linguagem simples e objetiva, de modo que a curva de aprendizado do *framework* não seja muito íngreme e seja útil no auxílio ao ensino do assunto.

4.1 Visão geral

A Figura 4.1 apresenta o diagrama com as principais atividades executadas com o UFJF-MLTK. No diagrama existem dois papéis que interagem com o UFJF-MLTK: o especialista, que faz extensão do *framework* através dos *hot-spots* (classes base ou abstratas), também chamados de pontos de extensão, para implementação de um algoritmo de aprendizado de máquina, e o usuário, que usa os métodos e ferramentas disponibilizados pelo especialista em sua aplicação.

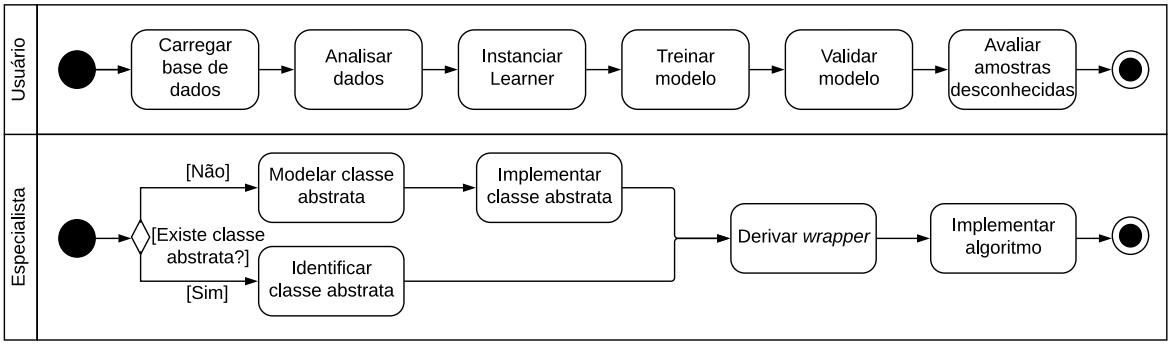

Figura 4.1: Diagrama de atividades com as principais atividades com o UFJF-MLTK.

A primeira pergunta que o especialista deve fazer antes de fazer uma extensão do UFJF-MLTK é “Existe uma classe abstrata para o algoritmo que vai ser implementado?”. Caso a resposta seja positiva, deve-se identificar qual a classe abstrata que melhor repre-

senta a extensão que será feita. Caso contrário, o especialista deve modelar e implementar uma classe abstrata para o tipo de algoritmo de aprendizado desejado seguindo o padrão do UFJF-MLTK. Após a criação da classe abstrata é feita a derivação da mesma para se criar um *wrapper* para encapsular o algoritmo sendo implementado em uma classe do C++, que devido a forma que foi modelada a arquitetura do UFJF-MLTK, vai seguir o padrão proposto pelo *framework*. Após a criação do *wrapper*, vem a fase de implementação do algoritmo, em que o especialista pode se aproveitar das ferramentas fornecidas pela arquitetura e pelo *wrapper* para tornar o desenvolvimento mais rápido e fácil por não ter que se preocupar com algumas decisões de implementação tomadas pelo padrão do UFJF-MLTK. Com isso, o papel do especialista chega ao fim e o algoritmo se torna disponível para uso.

O papel do usuário se dá pela utilização dos métodos e ferramentas do UFJF-MLTK como se fossem uma caixa-preta, com o objetivo de testar algoritmos ou utilizá-los sem se preocupar com uma implementação. A primeira atividade executada pelo usuário é o carregamento da base de dados na memória para manipulação dos dados. Após isso, podem ser feitas análises dos dados através de gráficos, técnicas de visualização ou com a utilização de métodos estatísticos para se escolher o método de aprendizado de máquina que melhor represente os dados no seu domínio do problema. Tendo ideia de qual algoritmo se utilizar, o usuário instância um dos *wrappers* derivados da classe abstrata *Learner* que envolve o método de aprendizado de máquina adequado ao problema a ser resolvido. Após a instanciação, é iniciada a fase de treinamento do método para gerar o modelo, em que o usuário fornece os parâmetros do método e, após o treinamento, obtém o modelo gerado pelo método. Após a fase de treinamento, são executados métodos de validação para verificar a acurácia do modelo nos dados, possibilitando ao usuário saber o desempenho do modelo quando lhe forem fornecidos novos dados do problema. E para finalizar, o modelo é utilizado para avaliar amostras desconhecidas que fazem parte do domínio do problema ao qual a base de dados representa.

O UFJF-MLTK também pode ser utilizado no auxílio do ensino de aprendizado de máquina. Para este fim, três classes são importantes: a `Data`, `Statistics` e a `Visualization`. A classe `Data` permite fazer manipulação dos dados e, com ela, o usuário

pode ver o impacto da retirada de exemplos, redução da dimensionalidade do problema e muitas outras operações que impactam no resultado de um método de aprendizado de máquina. Com a utilização da classe `Statistics` é possível aplicar métodos estatísticos nos dados representados pela classe `Data` para se extrair informações sobre a base de dados. Assim, o usuário aprende a fazer análises para conseguir um melhor desempenho durante o treinamento de um modelo. Finalmente, a classe `Visualization` permite visualizar os dados graficamente, podendo visualizá-los em 2 ou 3 dimensões e com ou sem o hiperplano resultante dos métodos disponíveis. A visualização dos dados é importante para se ter uma noção melhor da relação entre as classes e para o usuário conseguir ver o resultado de um método em bases de dados com poucas características, algo que é difícil de perceber apenas olhando para os valores numéricos da base de dados.

4.2 Arquitetura do *framework*

A arquitetura do UFJF-MLTK é composta por um conjunto de classes concretas e abstratas que interagem entre si para a instanciação do *framework*, com a instanciação podendo ser a criação de uma aplicação para solução de um problema, o teste e comparação de métodos, como também a inclusão de novos métodos para utilização por outros usuários. A Figura 4.2 mostra um diagrama de classes simplificado do UFJF-MLTK, que não representa toda a estrutura do *framework*. Como exemplo, a representação não deixa claro que cada uma das classes `Classifier`, `Regressor`, `Ensemble` e `Clusterer` tem suas versões primal e dual que permitem, respectivamente, a solução de problemas lineares e não lineares. As classes representam uma abstração para o domínio dos problemas de aprendizado de máquina construída a partir da implementação de diversos algoritmos já existentes e com a ajuda de professores com anos de experiência na área. Com isso, a arquitetura consegue abranger os problemas de aprendizado de máquina que aborda de forma satisfatória, reduzindo muito o trabalho do utilizador.

Muitas decisões tiveram de ser feitas no desenvolvimento do UFJF-MLTK. O *framework* foi desenvolvido com a utilização da linguagem C++14 por possuir uma biblioteca padrão ampla e madura, facilitando o desenvolvimento pela utilização de ferramentas da linguagem que são bem testadas e com um desempenho satisfatório. Por isso, os algorit-

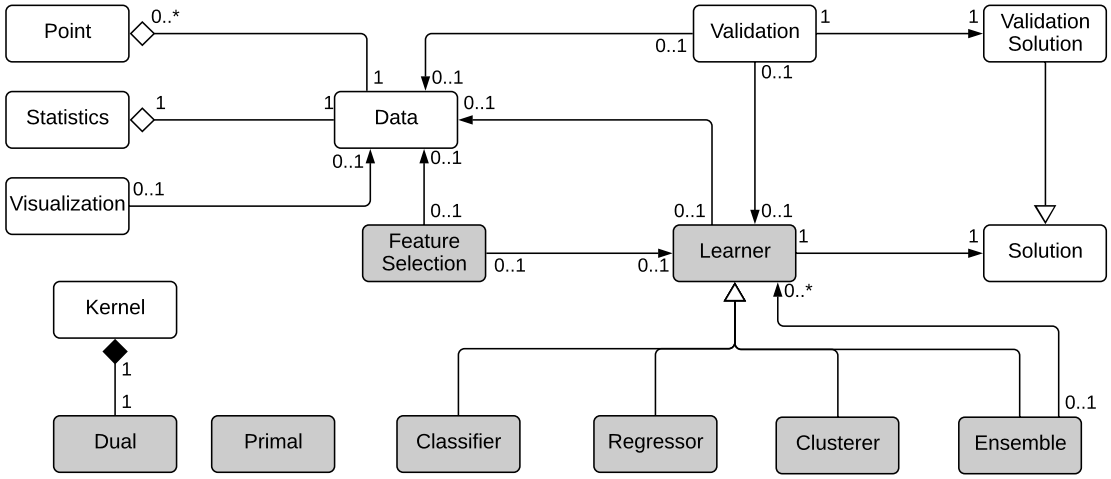

Figura 4.2: Diagrama de classes simplificado do UFJF-MLTK.

mos e estruturas de dados da *Standard Template Library* (STL) disponíveis pela linguagem foram amplamente usados. Devido às implementações sólidas de suas funcionalidades e por seguirem um padrão em todas implementações, o aprendizado do UFJF-MLTK se torna mais suave.

O uso do recurso de *templates*[4], que é um conceito da programação genérica para a implementação de classes e funções que utilizam tipos genéricos, permitiu que os componentes fossem mais genéricos e reusáveis. Assim, o utilizador tem total controle sobre o tipo de dados que melhor representa a base de dados, tornando possível a otimização do consumo de memória dos algoritmos. Também é feita a utilização extensiva de ponteiros inteligentes[5], uma estrutura abstrata que simula os ponteiros crus evitando os problemas que comumente acontecem com o gerenciamento de memória manual, permitindo que a implementação dos algoritmos tenha menos erros como, por exemplo, vazamento de memória. Devido aos ponteiros inteligentes, também é possível compartilhar memória entre objetos com mais facilidade, evitando que algoritmos que antes fariam muitas cópias da base de dados, e inevitavelmente precisariam de computadores com mais capacidade, gastem uma quantidade de memória constante em sua execução, desde que os dados iniciais não necessitem ser modificados.

O desenvolvimento do UFJF-MLTK se deu feito através de um ciclo de cinco etapas, sendo elas as etapas de pesquisa, implementação, teste, documentação do código

[4]Mais informações sobre esses recursos podem ser encontradas em MEYERS (2015).
[5]Veja a nota de rodapé 4.

e lançamento, representadas na Figura 4.3. Na etapa de pesquisa foi realizado o estudo dos métodos, domínio do problema e ferramentas necessárias para implementação do algoritmo ou componente. A pesquisa se deu através de artigos e livros clássicos da literatura, sendo às vezes necessário recorrer a pesquisas na *web* quando essas fontes não foram suficientes. Após a etapa de pesquisa, ocorreu a de implementação, em que foram implementados os componentes e algoritmos através de consultas aos materiais pesquisados.

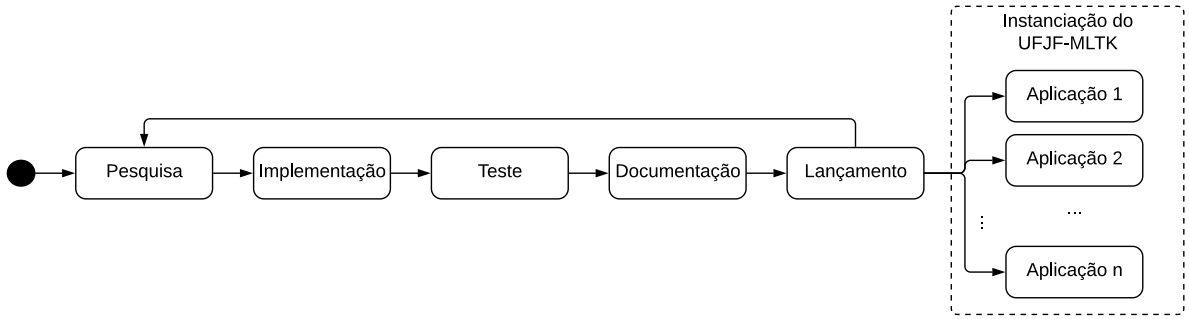

Figura 4.3: Desenvolvimento do UFJF-MLTK.

Tendo uma versão da implementação, começa a etapa de testes, em que o componente ou algoritmo é executado diversas vezes para se identificar defeitos. Adicionalmente são utilizadas as ferramentas *valgrind*[6], usada para a correção de vazamentos de memória, e *GNU Project Debugger* (GDB[7]), para depuração. A fase de testes pode ser executada junto com a de implementação para se ter uma primeira versão funcional. Com os testes concluídos, a funcionalidade precisa ser documentada. A documentação segue o padrão descrito na Seção 4.6 e foi feita com a adição de exemplos da *wiki* do repositório do projeto e nos comentários para execução da ferramenta *doxygen*[8], que está descrita na Seção 4.6 supracitada. Finalizada a documentação, a atualização é enviada para o repositório hospedado na plataforma *GitHub*[9] que também permite que se tenha um controle de versão. Esse processo é repetido a cada vez que se deseja incrementar o UFJF-MLTK de alguma forma.

As classes do UFJF-MLTK podem ser vistas como sendo de três tipos. Todas são criadas através de ferramentas do próprio C++ para evitar a dependência de muitas

[6]http://www.valgrind.org
[7]https://www.gnu.org/software/gdb/
[8]http://www.doxygen.org/
[9]https://github.com/

bibliotecas externas:

- Classes de utilidade: classes em que geralmente são implementados algoritmos que não são necessariamente associados ao aprendizado de máquina, mas oferecem operações de utilidade aos algoritmos, como classes para geração de números pseudo-aleatórios, contadores de tempo e manipulação de cadeias de caracteres;

- Classes de suporte geral: classes geralmente compartilhadas entre algoritmos de aprendizado de máquina que oferecem operações de suporte, como classes para visualização dos dados, validação de um método em uma base de dados e que fornecem operações estatísticas para aplicação nos dados;

- *Wrappers*: essas classes são envólucros para a implementação dos métodos de aprendizado de máquina e permitem que eles sejam utilizados por outros algoritmos de forma mais fácil, como por exemplo, durante a execução de um algoritmo de validação.

Dentro do UFJF-MLTK existem duas classes chaves, que estão representadas na Figura 4.4 com os seus principais métodos e atributos. A primeira é a classe `Data`, por ser a classe que representa os dados a serem processados. Um objeto de `Data` é composto por um conjunto de objetos da classe `Point`, que é a representação de um exemplo do conjunto de entrada, ao qual podem ser aplicadas as operações implementadas em `Data`. A classe `Data` é utilizada principalmente pelos métodos de aprendizado de máquina para criação de modelos de predição. A segunda é a classe `Learner`, que é uma abstração para os tipos de aprendizado suportados pelo *framework*, permitindo se fazer algoritmos mais genéricos como algoritmos de validação e do tipo *ensemble* (comitê), por poderem ser executados com algoritmos de diferentes tipos.

Através da classe abstrata `Learner` é possível criar *wrappers* para algoritmos de classificação, regressão, agrupamento, *ensemble* e suas respectivas variações primal e dual. Todos algoritmos no UFJF-MLTK estão dentro de um *wrapper* para que o desenvolvedor possa aproveitar das vantagens dadas pela arquitetura do *framework*. Para se utilizar do padrão dado é necessário que se faça instanciação de um dos pontos de extensão disponíveis no UFJF-MLTK através da criação de classes que herdem as características de uma class

e abstrata representando um tipo de algoritmo de aprendizado de máquina. Um ponto importante a se observar, é que a criação de algoritmos dentro de *wrappers* facilita a criação de métodos que tratam outros algoritmos como subrotinas como, por exemplo, a validação cruzada. Um exemplo disso são os métodos implementados pela classe abstrata `FeatureSelection`, que é uma abstração do problema de seleção de características, que frequentemente utilizam resultados de outros métodos de aprendizado de máquina como funções objetivo internas. As classes do UFJF-MLTK são reusáveis, ou seja, podem ser exportadas para outros projetos de *software* sem a necessidade de copiar todo código do *framework*, desde que, quando existentes, também sejam exportadas as classes que são dependentes.

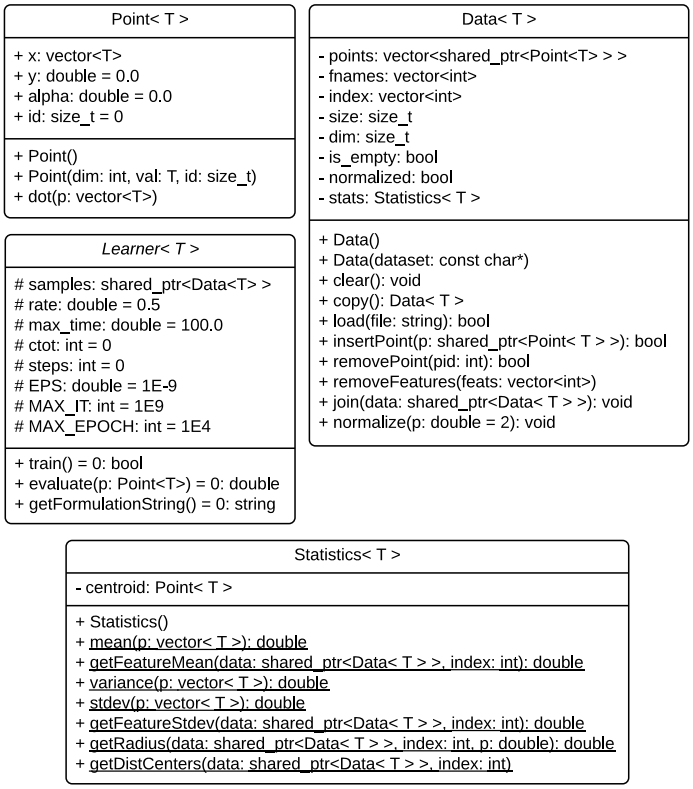

Figura 4.4: Representação das principais classes do UFJF-MLTK.

4.3 Pontos de extensão

Os pontos de extensão foram feitos através da análise de domínio dos problemas de aprendizado de máquina. Através da análise de domínio foi possível identificar os principais tipos de algoritmos de aprendizado de máquina atualmente estudados, permitindo a iden-

tificação dos pontos de extensão do *framework*. Em seguida foi feita uma análise da implementação de vários algoritmos dos mesmos tipos para identificação das similaridades entre eles. Essas similaridades foram incluídas nos pontos de extensão para criar uma abstração dos tipos de algoritmos de aprendizado, facilitando o processo de desenvolvimento de algoritmos através da redução do esforço tido pelo utilizador na implementação.

Existem vários pontos de extensão no UFJF-MLTK, permitindo que o desenvolvedor tenha liberdade para incrementar o *framework* aproveitando das funcionalidades fornecidas pelas classes concretas. O desenvolvedor pode fazer instanciação do UFJF-MLTK através da herança das classes abstratas derivadas da classe `Learner`, onde pode escolher fazer derivação diretamente das classes abstratas que representam os tipos de aprendizado, do qual tem-se as classes `Regressor`, `Classifier`, `Clusterer` e `Ensemble`, ou pode derivar suas variações, em que cada uma dessas classes tem a sua formulação primal e dual. Por último, o desenvolvedor também pode instanciar o UFJF-MLTK através da classe abstrata `FeatureSelection` caso deseje implementar um método de seleção de características. O processo de derivação de um *wrapper* a partir de um ponto de extensão (classe abstrata) é representado na Figura 4.5.

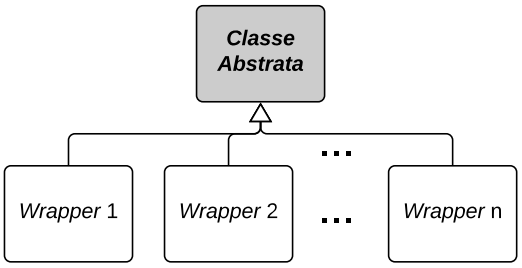

Figura 4.5: Derivação de um *wrapper* a partir de um ponto de extensão.

O padrão dado pelo UFJF-MLTK para implementação de algoritmos de aprendizado de máquina, existe devido as hierarquias de classes abstratas projetadas na arquitetura do *framework*. Essas hierarquias obrigam o utilizador a manter um padrão em sua implementação, através da obrigatoriedade da definição dos métodos abstratos advindos das classes superiores, para que possa aproveitar dos benefícios propostos pelo *framework*.

4.4 As classes do UFJF-MLTK

Esta Seção descreve as classes presentes no UFJF-MLTK, que podem ser vistas no diagrama de classes da Figura 4.6. Não são descritas as classes que aparecem sem detalhes no diagrama supracitado, por ainda estarem em desenvolvimento no momento da apresentação deste trabalho.

4.4.1 Point

A classe `Point` é uma abstração para os exemplos da base de dados. Ela fornece operações e atributos pertinentes para a manipulação do exemplo, como, o vetor x que representa as características, o atributo y que representa o valor desejado e o *alpha* usado em métodos na formulação dual. As características dos exemplos representados pela classe `Point` podem ser acessadas da mesma forma que se faz o acesso dos elementos de um vetor, isso é possível através da sobrescrita do operador [] que permite o acesso aos elementos por índices. O acesso as características é demonstrado na Listagem 4.1.

```
1 Point<double> p(10);
2 size_t i, dim = p.getDim();
3
4 for(i = 0; i < dim; i++){
5     std::cout << p[i] << " ";
6 }
```

Listagem 4.1: Exemplo de acesso as características de um exemplo. O objeto p é inicializado com dimensão 10 e todas posições são preenchidas com o valor padrão do tipo *double* de 0.0.

4.4.2 Data

A classe `Data` é usada para manipulação dos dados contidos na base de dados. Ela contém um conjunto de pontos representando cada exemplo da base de dados e também operações que podem ser aplicadas sobre os mesmos. Por os pontos da classe estarem sendo utilizados através do ponteiro inteligente *shared_ptr*, os objetos de `Data` podem ser atribuídos através da operação de atribuição fazendo compartilhamento da memória, isso é muito útil quando se deseja manipular os dados sem alterá-los, assim otimizando o uso da memória. Portanto, a cópia dos dados de um objeto de `Data` para outro deve ser feita

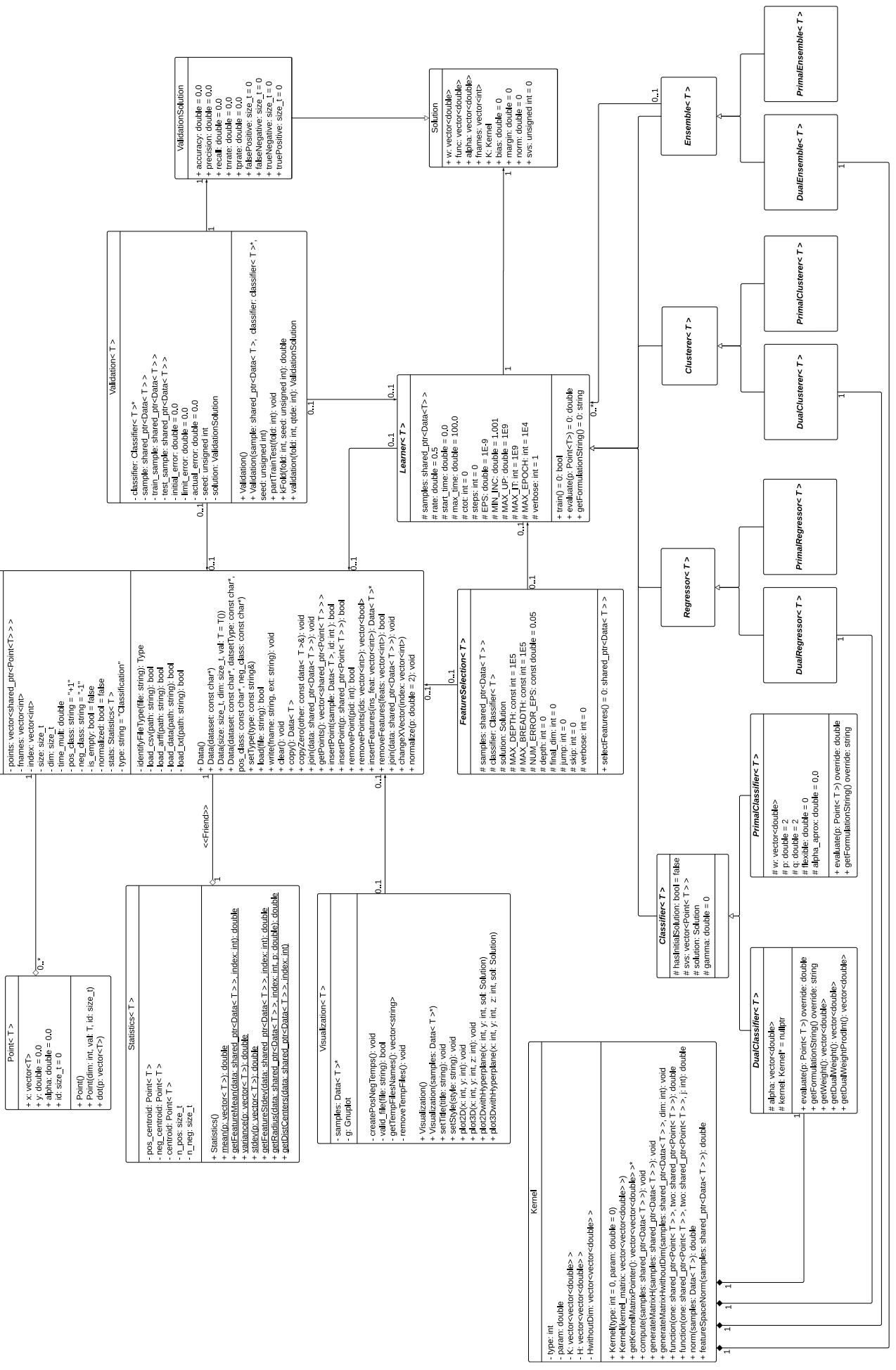

Figura 4.6: Diagrama de classes completo do UFJF-MLTK.

de forma explícita através da chamada do método *copy*(). A Listagem 4.2 mostra como os exemplos da base de dados são representados na estrutura da classe.

```
1 std::vector<std::shared_ptr<Point< T > > > points;
```

Listagem 4.2: Representação dos exemplos na classe `Data`.

Atualmente, a classe `Data` pode ser utilizada por métodos de regressão e de classificação, sendo necessário apenas dizer que tipo de base de dados que está sendo carregado através do atributo *type*. Os formatos dos arquivos das bases de dados aceitos pela classe são: `csv`, `arff`, `data` e `txt`.

Os objetos de `Data` podem ser tratados de forma análoga a vetores ou matrizes, em que é possível acessar, respectivamente as referências para os exemplos armazenados ou as características individuais de cada ponto. Isso facilita o acesso dos dados armazenados e torna a utilização dos objetos mais familiar aos utilzadores. O acesso aos pontos armazenados no objeto de `Data` podem ser vistos na Listagem 4.3. Observe que o $data[i]$ retorna um *shared_ptr* para um exemplo da base de dados.

```
1 size_t i, j, size = data.getSize(), dim = data.getDim();
2
3 for(i = 0; i < size; i++){
4     for(j = 0; j < dim; j++)
5         std::cout << (*data[i])[j] << std::endl;
6 }
```

Listagem 4.3: Exemplo de acesso aos dados armazenados no objeto de `Data` para imprimir as características dos exemplos armazenados.

Através da utilização do construtor `Data(size_t size, size_t dim, T val = T())` é possível instanciar a classe `Data` com um tamanho, dimensão e valor padrão inicial. Isso é útil para, por exemplo, a criação de bases de dados artificiais sem a necessidade da criação e inserção de um ponto, através dos métodos correspondentes, para representar um exemplo, tornando o custo computacional mais baixo.

4.4.3 Statistics

A classe `Statistics` permite aplicar operações estatísticas sobre os objetos de *Data*. Essas operações são úteis quando se deseja obter informações úteis sobre os dados, como por exemplo, a variância em uma característica ou na base de dados, a média e também

informações em conjunto com os rótulos dos pontos, como a distância entre os centros e o raio dos pontos. A classe *Statistics* tem os atributos *pos_centroid*, *neg_centroid* e *centroid*, representando os centróides dos pontos com rótulos positivos e negativos, e o centróide geral dos pontos, respectivamente. Os atributos *n_pos* e *n_neg* são contadores com o número de pontos com classes positivas e negativas. A classe *Data* tem um objeto da *Statistics* associada para facilitar o uso das operações estatísticas e dos atributos.

4.4.4 Visualization

A visualização dos dados é uma ferramenta útil para a análise dos dados antes do processamento para extração de conhecimento. A classe `Visualization` permite a visualização dos dados em duas ou três dimensões, dando também a opção de visualizar os dados com o hiperplano gerado por um classificador. A visualização dos dados é possível com a ajuda do *gnuplot* através da passagem das dimensões que se deseja visualizar como parâmetro dos métodos da classe. É possível utilizar essa ferramenta de visualização de dados tanto no sistema operacional linux quanto no windows.

4.4.5 Solution

A classe `Solution` é um *wrapper* para a solução dos métodos de aprendizado. Sempre que um algoritmo de aprendizado de máquina é executado sobre uma base de dados, o modelo resultante é salvo dentro de um objeto da *Solution*. Os atributos dessa classe são o vetor de pesos w, o vetor $func$ que representa os resultados da aplicação do modelo em cada ponto, o vetor *alpha* é salvo por métodos kernel, o atributo K que representa a matriz kernel, etc.

4.4.6 ValidationSolution

A classe `ValidationSolution` solution herda os atributos da classe `Solution` e armazena os resultados da validação de um método executada pela classe `Validation`. Os atributos $accuracy$, $precision$, $recall$, $tnrate$, $tprate$, $falsePositive$, $falseNegative$, $trueNegative$ e $truePositive$ cada um representando respectivamente a acurácia, a precisão, o *recall*, a taxa de positivos verdadeiros, a taxa de falsos positivos, o número de falsos positivos, o

número de falsos negativos, o número de negativos verdadeiros e o número de positivos verdadeiros. A relação entre a classe `Solution` e `ValidationSolution` está representada na Figura 4.7.

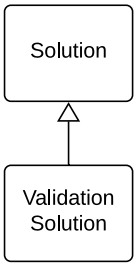

Figura 4.7: Relação entre as classes `Solution` e `ValidationSolution`.

4.4.7 Learner

A classe `Learner` é a principal abstração para os algoritmos de aprendizado de máquina disponíveis dentro do UFJF-MLTK, todas classes abstratas usadas para implementação de métodos de aprendizado de máquina, com exceção da `FeatureSelection`, derivam dela. A classe `Learner` contém os métodos e atibutos que são comuns as classes que representam os tipos de aprendizado. Portanto, qualquer classe abstrata que for incluída no UFJF-MLTK para representar um tipo de aprendizado deve ter uma relação de herança com a `Learner`. O padrão de desenvolvimento dos *wrappers* de algoritmos é resultado da hierarquia de classes a partir da `Learner`, qualquer algoritmo implementado deve fazer a sobrescrita dos métodos abstratos $train()$ e $evaluate()$.

4.4.8 Validation

Frequentemente é necessário medir a acurácia da predição de um modelo para escolher o que melhor resolve um problema. A classe `Validation` é usada para executar algoritmos de validação, até o momento podendo ser em qualquer classificador binário, sendo essa uma das vantagens de se abstrair os tipos de algoritmos de aprendizado de máquina em classes abstratas. A classe `Validation` pode particionar a base de dados em treino e teste e fazer a validação utilizando o método de validação cruzada estratificada. Os principais atributos são um ponteiro para o classificador sendo validado, um ponteiro para a base

de dados e para as partições de treino e teste e a semente usada pelo gerador de números pseudoaleatórios.

4.4.9 Classifier

A classe `Classifier` têm os atributos que são compartilhados pelos classificadores e faz parte da hierarquia de classes do padrão do UFJF-MLTK. Os *wrappers* dos classificadores, apesar de possível, não devem fazer herança diretamente dessa classe, mas sim de suas derivações que vão ser aludidas posteriormente. A Figura 4.8 mostra a relação entre as classes `Learner` e `Classifier`.

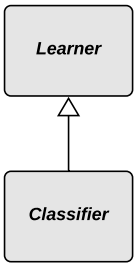

Figura 4.8: Relação entre as classes `Learner` e `Classifier`.

4.4.10 FeatureSelection

A classe `FeatureSelection` faz abstração dos métodos de seleção de características. Com ela é possível derivar *wrappers* apenas fazendo sobrescrita do método abstrato $selectFeatures()$. A Figura 4.9 mostra o processo de criação de *wrappers* para métodos de seleção de características.

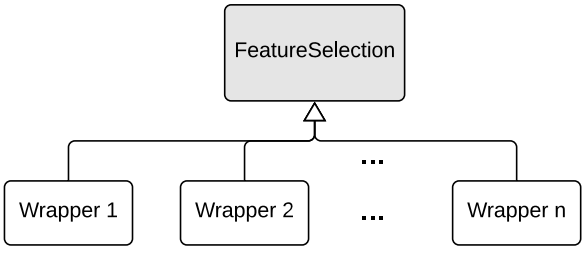

Figura 4.9: Derivação de *wrappers* a partir da classe `FeatureSelection`.

4.4.11 As variações primal e dual

Cada tipo de aprendizado abordado pelo UFJF-MLTK têm suas formulações primal e dual. A formulação primal implementa métodos que resolvem problemas lineares, enquanto que a dual permite a solução de problemas não lineares. São dessas classes que os *wrappers* dos algoritmos de aprendizado de máquina devem ser derivados. Quando for derivada uma classe representando a formulação primal ou dual de um problema de aprendizado, deve-se também implementar o método abstrato $getFormulationString()$, retornando a *string* "Primal" ou "Dual", vindo da classe *Learner*, esse método informa aos *wrappers* mais genéricos, como os de validação, qual formulação de um algoritmo está sendo usada. Como exemplo, a Figura 4.10 mostra como fica a hierarquia de classes abstratas que permite a criação de *wrappers* para algoritmos de classificação.

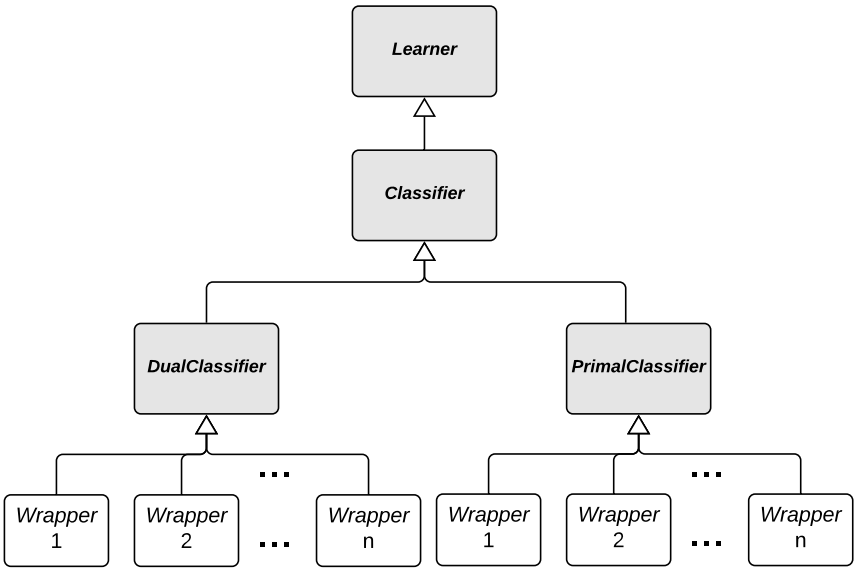

Figura 4.10: Hierarquia de classes para implementação de *wrappers* para classificadores.

4.4.12 Kernel

A classe `Kernel` é utilizada pelos métodos de aprendizado de máquina em suas formulações duais para executar predições não lineares. Ela é responsável por executar as computações necessárias para criar uma matriz kernel baseada na base de dados passada como parâmetro e também por manipular essa matriz resultante. Até o momento é possível computar matrizes kernel com as seguintes funções: linear, polinomial e gaussiana.

4.5 Algoritmos e métodos implementados

Com foco inicial no problema de classificação binária, os seguintes algoritmos foram implementados no UFJF-MLTK até o momento e podem ser executados através da API do *framework*:

- *Perceptron* (ROSENBLATT, 1958): algoritmo para aprendizagem de um classificador binário que ajusta um modelo linear nos dados no caso de sua formulação primal e um modelo não linear em sua formulação dual;

- *Fixed Margin Perceptron* (FMP) (DUDA; HART; STORK, 2012): Algoritmo para aprendizagem de um classificador binário que tenta ajustar um modelo linear em sua formulação primal ou não linear em sua formulação dual com a margem passada como parâmetro;

- *Incremental p-Margin Algorithm* (IMA_p) (VILLELA; LEITE; NETO, 2016): Algoritmo para classificadores de larga margem que faz uma aproximação da margem máxima de um modelo linear através de várias execuções do Perceptron de margem fixa passando uma margem incremental como parâmetro. O IMA_p pode utilizar uma norma p arbitrária;

- *Incremental Margin Algorithm* Dual (IMA Dual) (LEITE et al, 2018): Algoritmo IMA em sua formulação dual, pode fazer o ajuste de modelos não lineares;

- *Sequential Minimal Optimization* (SMO) (PLATT, 1998): algoritmo para o treinamento de máquinas de vetores suporte que resolve um problema de programação quadrática. O SMO quebra o problema nos menores subproblemas possíveis e os resolve analiticamente;

- Golub (GOLUB et al., 1999) e Fisher (BISHOP et al., 1994): métodos de seleção de características em filtro baseados na diferença do nível de expressão de cada atributo. Os métodos de Golub e Fisher se diferenciam apenas no cálculo do ranqueamento das características;

- *Recursive Feature Elimination* (RFE) (GUYON et al, 1994): método de seleção de

características *wrapper* baseado na eliminação dos componentes menos expressivos do vetor de pesos e no reajuste do modelo em cada passo do processo;

- *Admissible Ordered Search* (AOS) (VILLELA; XAVIER; NETO, 2011): Método de seleção de características que se baseia na realização de uma busca ordenada admissível. Este algoritmo tem a capacidade de encontrar o classificador de maior margem em cada dimensão do problema;

- *K-Fold Cross-Validation* (KOHAVI et al., 1995): método para estimativa da acurácia e seleção de modelos baseado na divisão da base de dados em k partições mutuamente exclusivas. O modelo é treinado e testado k vezes. Em cada vez é treinado em $(k-1)$ partições e testado na restante. A acurácia é estimada pelo total do número de acertos dividido pelo número de exemplos.

4.6 Documentação

A documentação de um *framework* deve abranger o seu propósito, como deve ser utilizado, os propósitos das suas aplicações de exemplo e o seu projeto (JOHNSON, 1992).

O UFJF-MLTK utiliza de duas abordagens para a documentação. A primeira é através de um *cookbook*. Essa abordagem é motivada pela ideia de que a maioria dos usuários não estão interessados nos detalhes de como usar o *framework*, mas sim em uma descrição de como utilizá-lo (MATTSON, 1996). A documentação do UFJF-MLTK aproveita dessa abordagem através da criação de exemplos de utilização, no qual cada exemplo é descrito detalhadamente. Esse *cookbook* pode ser encontrado no *wiki*[10] do projeto e está dividido em seções com instruções de como compilar o código, uma breve descrição do projeto e várias outras seções exemplificando o uso do UFJF-MLTK.

A segunda abordagem é através da utilização de uma ferramenta de documentação automática chamada *doxygen*. Essa ferramenta gera a documentação no formato HTML que pode ser hospedado em um site. O site é hospedado na plataforma *GitHub Pages*[11][12] e nele se encontram as descrições, diagramas de dependências, diagramas de herança e

[10] Acesso à *wiki*: https://github.com/mateus558/Machine-Learning-Toolkit/wiki
[11] https://pages.github.com/
[12] Acesso ao site: https://mateus558.github.io/Machine-Learning-Toolkit/

diagramas de colaboração das classes e funções que fazem parte do UFJF-MLTK, além da documentação da sua API. Essa documentação gerada também permite ao usuário ter uma visão sucinta da arquitetura do *framework* através dos tipos de listagem de classes disponíveis. Por fim, a documentação do UFJF-MLTK consegue abranger todos itens citados anteriormente, evitando que os usuários tenham muitas dificuldades no aprendizado da utilização.

4.7 Considerações finais

Este capítulo descreveu o *framework* UFJF-MLTK. Foram detalhadas as principais atividades executadas pelos papéis que o utilizador pode ter, sendo esses papéis, os do especialista e do usuário. O especialista utiliza o UFJF-MLTK para inclusão de novos componentes e algoritmos, enquanto que o usuário cria programas utilizando das implementações do especialista. O *framework* foi desenvolvido através de um projeto orientado a objetos, onde as classes concretas representam ferramentas e métodos frequentemente utilizados pela comunidade de aprendizado de máquina e as classes abstratas criam um padrão de implementação para os algoritmos de aprendizado de máquina através de suas hierarquias de classes. O UFJF-MLTK foi documentado seguindo a abordagem de criação de um *cookbook* e por um *site* com as descrições das funções e classes pertencentes ao *framework*. Atualmente, o UFJF-MLTK conta com *wrappers* de alguns algoritmos de cassificação clássicos e de algoritmos desenvolvidos dentro do Departamento de Ciência da Computação da UFJF. O próximo capítulo exemplifica a instanciação do *framework* pelo ponto de vista do usuário e do especialista.

5 Instanciação e utilização do UFJF-MLTK

As seções a seguir mostram um exemplo de instanciação por um especialista e por um usuário. Como exemplo, a instanciação pelo especialista do algoritmo *Perceptron* em sua formulação primal utilizando o padrão dado pelo UFJF-MLTK, e logo em seguida, a utilização da implementação do mesmo para instanciação pelo usuário. Para demonstrar apenas a estrutura da instanciação do UFJF-MLTK, alguns detalhes de implementação dos métodos foram ocultados.

5.1 Instanciação por um especialista

Para o exemplo do *Perceptron* primal, é necessário que primeiro seja incluída a biblioteca "`PrimalClassifier.hpp`", que contém a o ponto de extensão necessário para a instanciação. Depois é feita a herança da classe abstrata `PrimalClassifier`, contida na biblioteca incluída, que tem a estrutura básica para implementação de classificadores primais. Inicialmente é necessário criar o construtor para inicialização dos campos da classe e os métodos abstratos que têm implementação obrigatória pela hierarquia de classes ao qual a classe `PrimalClassifier` pertence. Na Listagem 5.1 é mostrado como fica o *header* do *wrapper*. Duas observações importantes devem ser feitas nessa listagem: primeiro que se deve dar um nome ao tipo genérico que vai ser usado, nesse exemplo o nome dado é T. Isso ocorre porque as classes são implementadas como *templates* e o algoritmo deve ser compatível com o tipo usado para representar a base de dados. Outra observação é que o parâmetro do construtor, nesse caso a base de dados representada por `Data`, está sendo referenciado através do ponteiro inteligente `shared_ptr`. Isso permite que a memória não seja utilizada além do necessário e também que o mesmo local de memória possa ser utilizado por vários métodos e algoritmos sem fazer cópias.

```
1 #include "../includes/PrimalClassifier.hpp"
2
3 template < typename T >
4 class Perceptron: public PrimalClassifier< T > {
```

```
5  public:
6      Perceptron(std::shared_ptr<Data< T > > samples):
7      this->samples(samples) {}
8      bool train() override;
9      double evaluate(Point< T > p) override;
10 };
```

Listagem 5.1: *Wrapper* do algoritmo Perceptron Primal.

Após a criação do *header* é necessário fazer a implementação dos métodos do *wrapper*. No caso do *Perceptron* primal é necessária a implementação apenas do construtor e dos métodos abstratos `train` e `evaluate` que são comuns a todos algoritmos de aprendizado e têm sua implementação obrigatória pela classe abstrata `Learner`. O método abstrato `train` mostrado na Listagem 5.2 é usado para executar a fase de treinamento do algoritmo e gerar um modelo para os dados.

```
1  template< typename T >
2  Perceptron< T >::train(){
3      size_t i = 0, j = 0, k = 0, size = this->samples->getSize();
4      size_t dim = this->samples->getDim(), error = 0;
5      double bias = 0;
6      std::vector<double> w(dim, 0.0), func(size, 0.0);
7
8      for(i = 0; i < this->MAX_IT; i++){
9          error = 0;
10         for(j = 0; j < size; j++){
11             auto input = *this->samples[j];
12             for(func[j] = bias, k = 0; k < dim; k++)
13                 func[j] += (*input)[k] * w[k];
14
15             if(input->y*func[j] <= 0.0){
16                 for(k = 0; k < dim; k++){
17                     w[k] += this->rate * input->y * (*input)[k];
18                 }
19                 bias += this->rate * input->y;
20                 error++;
21                 this->ctot++;
22             }
23         }
24         this->steps++;
25
26         if(error == 0)
27             break;
28     }
29     this->solution.w = w;
30     this->solution.bias = bias;
31 }
```

Listagem 5.2: Função *train* do Perceptron.

A implementação do método `evaluate`, encontrado na Listagem 5.3, tem a função de atribuir uma classe, no caso de ser implementado em um classificador, ao exemplo dado como parâmetro.

```
1 template< typename T >
2 double Perceptron< T >::evaluate(Point< T > p){
3     double func = 0.0;
4     size_t i, dim = this->solution.w.size();
5
6     for(func = this->solution.bias, i = 0; i < dim; i++)
7         func += this->solution.w[i] * p[i];
8     return (func >= 0)?1:-1;
9 }
```

Listagem 5.3: Função *evaluate* do Perceptron.

5.2 Instanciação por um usuário

Programar com o UFJF-MLTK geralmente requer pouca quantidade de código por já existirem muitas utilidades implementadas, então o foco do usuário é de como usar o código já existente, através da API disponibilizada pelo *framework*, para desenvolver programas. A Listagem 5.4 demonstra a instanciação do UFJF-MLTK por um usuário, em que os métodos e ferramentas são utilizados para a criação de uma aplicação para resolver um problema de aprendizado de máquina. Este exemplo demonstra a utilização da classe concreta `Data` para a manipulação dos dados e incluída através da biblioteca "`Data.hpp`", da classe concreta `Visualization` para a visualização dos dados, incluída através da biblioteca "`Visualization.hpp`", e do *wrapper* do algoritmo *Perceptron* em sua formulação primal, incluído através da biblioteca "`Perceptron.hpp`", criado previamente por um especialista, para a solução de um problema de classificação binária.

```
1  #include "includes/Data.hpp"
2  #include "includes/Visualization.hpp"
3  #include "includes/Perceptron.hpp"
4
5  int main(){
6      Data<double> data("exemp.data");
7      Perceptron<double> perc(std::make_shared<Data<double> >
8      (data));
9      Visualization<double> plot(&data);
10     Solution sol;
```

```
11
12      perc.setMaxIterations(100);
13      perc.setLearningRate(1.0);
14      perc.train();
15      sol = perc.getSolution();
16      plot.plot3DwithHyperplane(1,2,3,sol);
17
18      for(size_t i = 0; i < data.getDim(); i++)
19          std::cout << sol.w[i] << " ";
20      std::cout << "bias: " << sol.bias << std::endl;
21
22      Validation<double> validate(std::make_shared<Data<double> >
23      (data), &perc);
24      ValidationSolution valSol;
25
26      valSol = validate.validation(10, 10);
27
28      std::cout << "\nAccuracy: " << valSol.accuracy;
29      std::cout << "\nPrecision: " << valSol.precision;
30      std::cout << "\nRecall: " << valSol.recall;
31  }
```

Listagem 5.4: Instanciação do UFJF-MLTK por um usuário.

Primeiramente, o usuário precisa incluir as bibliotecas do UFJF-MLTK necessárias para resolver o problema atacado pela aplicação. Em seguida, o usuário carrega a base de dados "exemp.data", que consiste em um conjunto de amostras sintéticas com três dimensões e duas classes. Em seguida o objeto com a base de dados é passado para o construtor do *Perceptron* e do objeto da classe `Visualization`, também é instanciado um objeto da classe `Solution` para armazenar os resultados do treinamento do *Perceptron*. Após esse passo, os parâmetros do *Perceptron* são passados através das funções herdadas da classe abstrata `Learner`. No exemplo são passados o número máximo de iterações pelo método `setMaxIterations()` e a taxa de aprendizagem pelo método `setLearningRate()`. Os parâmetros não passados são colocados como os padrões do *wrapper*. Com os parâmetros desejados atribuídos, é chamada a função de treinamento que ajusta um modelo nos dados e gera uma solução.

A solução é passada para o objeto da classe `Solution`. Com o objeto `sol` é possível visualizar o modelo gerado pelo *Perceptron* com os métodos de visualização da classe `Visualization`. Para isso, é preciso informar as dimensões que vão compor os eixos para o objeto `plot` através do método $plot3DwithHyperplane()$, que logo após a sua chamada, cria uma janela com o gráfico 3D dos pontos com o hiperplano gerado pelo

Perceptron. No exemplo foram usadas as três dimensões da base de dados e o gráfico gerado com o hiperplano é mostrado na Figura 5.1. Então, o usuário imprime os pesos do modelo gerado pelo *Perceptron* na tela.

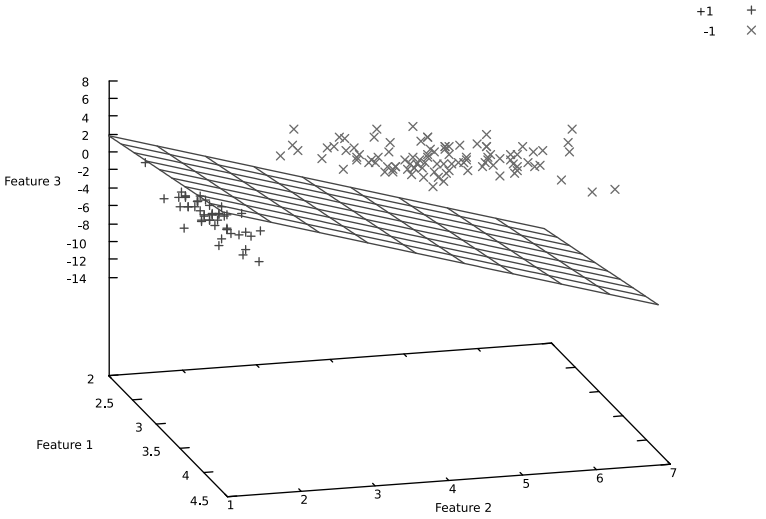

Figura 5.1: Visualização do hiperplano separador gerado pelo Perceptron.

Após a visualização do modelo em um gráfico 3D com os exemplos da base de dados, o usuário cria um objeto da classe `Validation` passando como parâmetros do construtor um *shared_ptr* para o objeto de `Data` que representa a base de dados e um ponteiro para o objeto do *wrapper* do algoritmo *Perceptron*, também é criado um objeto da classe `ValidationSolution` onde os resultados da validação são armazenados. Após esse passo, é chamado o método *validation*() do objeto `validate` que executa internamente o algoritmo *K-Fold Cross-Validation*, passando como parâmetros o número k de partições e quantidade de execuções que são feitas do algoritmo. O resultado da chamada do método *validation*() é retornado e salvado no objeto `valSol`, permitindo que o usuário possa acessar as informações computadas sobre o modelo gerado pelo *Perceptron* na base de dados. Em seguida, o usuário imprime na tela a acurácia, a precisão e a revocação do modelo. A Figura 5.2 mostra que o *Perceptron* conseguiu uma acurácia de 99.84%, precisão de 99.53% e revocação de 100.00% na validação com a base de dados dada.

```
Accuracy: 0.998429
Precision: 0.995274
Recall: 1
```

Figura 5.2: Resultado da validação do *Perceptron*.

5.3 Interface pela linha de comando

O UFJF-MLTK também oferece uma interface com as suas funcionalidades internas através da linha de comando. Isso permite que o usuário possa utilizar as ferramentas e algoritmos implementados no *framework* tendo conhecimento apenas dos dados que vai processar e dos métodos e técnicas de aprendizado de máquina. A Figura 5.3 mostra o menu principal com as opções disponíveis na interface. Essas opções representam o que já é possível fazer com o UFJF-MLTK. A seguir estão descritas as funcionalidades que cada opção permite acessar:

- *Dataset*: permite carregar a base de dados em vários formatos e tem opções para mostrar informações sobre a mesma;
- *Data*: contém as operações possíveis sobre os dados, tanto estatísticas, quanto operações para manipulação dos dados;
- *Data Visualization*: acessa as funcionalidades da classe `Visualization`, permitindo visualizar os dados com ou sem o modelo gerado por um classificar;
- *Classifiers*: acessa outro submenu permitindo a escolha entre métodos de classificação em suas formulações primal ou dual para aplicação na base de dados caregada;
- *Feature Selection*: permite escolher entre os métodos de seleção de características para aplicação na base de dados;
- *Validation*: faz a validação de um método de classificação com a base de dados carregada executando o *k-Fold Cross Validation*;
- *Set Verbose* e *Set Max Time*: permitem atribuir respectivamente, o nível de detalhes mostrados da execução dos algoritmos e o máximo de tempo que um método de aprendizado de máquina pode executar.

Cada *wrapper* implementado no UFJF-MLTK permite a impressão de detalhes da execução dos seus algoritmos correspondentes com um nível de detalhes escolhido através do atributo *verbose*. Por padrão, a interface do UFJF-MLTK coloca o valor do atributo

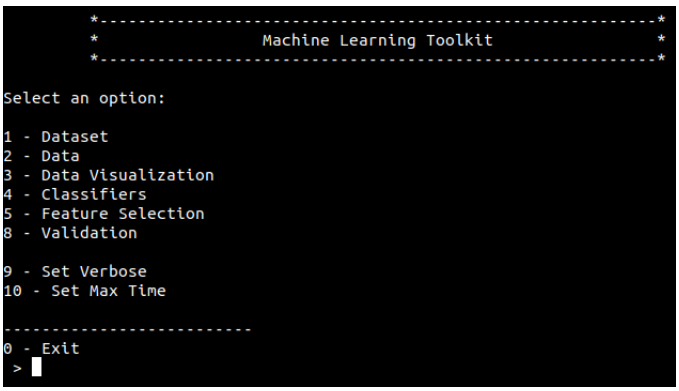

Figura 5.3: Menu principal da interface do UFJF-MLTK.

verbose de tal forma que todos detalhes da execução são mostrados. A Figura 5.4 mostra os detalhes da execução do algoritmo IMAp a cada execução do algoritmo *Fixed Margin Perceptron*, executado como subrotina, em uma base de dados carregada. Esse tipo de informação sobre a execução do algoritmo é importante para que o usuário não precise esperar até o fim da execução para saber como o método está se comportando na base de dados utilizada. Caso não houvessem esses detalhes durante a execução, o usuário poderia perder muito do seu tempo até esperar o fim da execução do método de aprendizado de máquina para saber se o resultado conseguido era o esperado.

Esses detalhes também são úteis durante o ensino de aprendizado de máquina, por ser possível mostrar para o aluno como os algoritmos se comportam logo após terem sido apresentados durante a aula, também por ser possível mostrar com facilidade o modelo resultante pela opção *Data Visualization* logo após a execução do algoritmo.

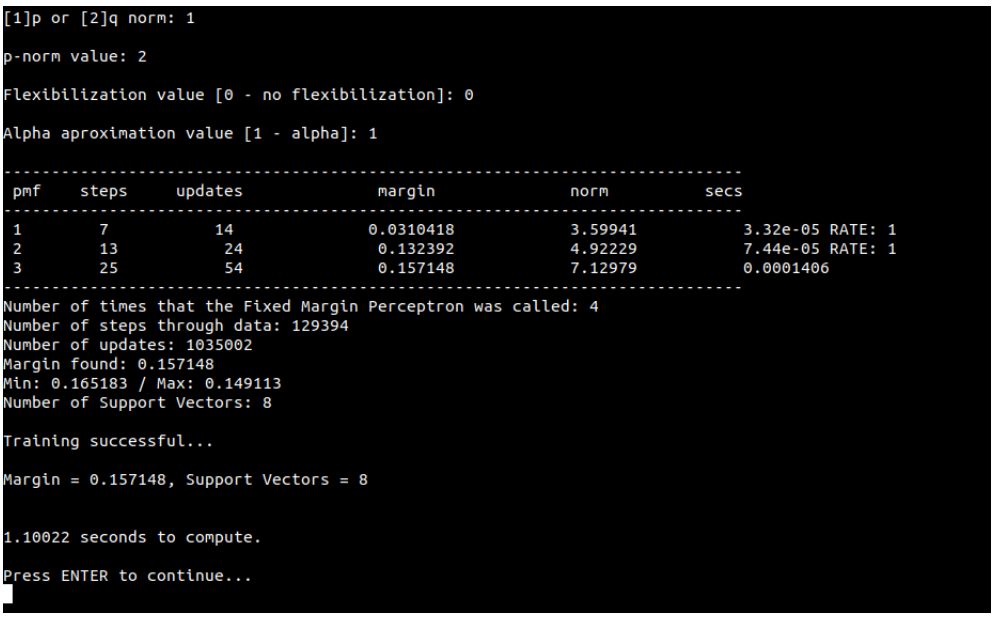

Figura 5.4: Detalhes da execução do IMA$_p$.

5.4 Considerações finais

Esse capítulo apresentou dois exemplos de instanciação do UFJF-MLTK, um do ponto de vista do papel de um especialista e outro do usuário. A instanciação feita pelo especialista é feita com o objetivo de extender o *framework*, para inclusão de novos componentes ou para implementação de métodos de aprendizado de máquina. No exemplo do especialista, foi implementado o algoritmo *Perceptron* através da criação de um *wrapper* seguindo o padrão definido pela hierarquia de classes definida para implementação de classificadores. O usuário faz instanciação do UFJF-MLTK para criação de programas utilizando as funcionalidades disponibilizadas pelo especialista. O exemplo de instanciação pelo usuário mostra a criação de um programa em que é criado um modelo para resolver um problema de classificação binária dado pela base de dados carregada através da sua API. Além disso, o exemplo demonstra como é feita a visualização do modelo com base nos dados carregados e executa a validação do mesmo para estimar o desempenho do modelo no problema. O UFJF-MLTK também oferece uma interface para utilização de suas ferramentas sem a necessidade de ter conhecimento da estrutura do *framework*, fazendo com que possa ser utilizado como uma caixa de ferramentas. O próximo capítulo descreve as conclusões e trabalhos futuros relacionados ao UFJF-MLTK.

6 Conclusão

Este trabalho apresentou o UFJF-MLTK, um *framework* orientado a objetos desenvolvido em C++, de código aberto e disponível para a comunidade de aprendizado de máquina. O UFJF-MLTK inclui abordagens para diversos problemas de aprendizado de máquina e pode ser facilmente extendido para inclusão de novos métodos de aprendizado de máquina e componentes para adicionar utilidades ao *framework*. O presente trabalho também foi escrito em formato de artigo e está em processo de avaliação para o Simpósio Brasileiro de Sistemas de Informação (SBSI) de 2019.

Devido a utilização de ferramentas e algoritmos da *Standard Template Library* e técnicas de programação do C++ moderno, a linguagem do *framework* é simples e bem documentada o suficiente, através de vários exemplos de utilização e descrições de suas funcionalidades, para que a sua utilização seja de fácil aprendizado. Isso junto com a variedade de métodos implementados e com os componentes de manipulação e visualização de dados, faz com que o UFJF-MLTK seja uma boa ferramenta no auxílio ao ensino de aprendizado de máquina. Além disso, essas características reduz consideravelmente a quantidade de código necessário para o desenvolvimento de programas.

As classes desenvolvidas no UFJF-MLTK são reusáveis devido ao seu desacoplamento, portanto, elas podem ser utilizadas em outros projetos de forma independente. O UFJF-MLTK também fornece um padrão de desenvolvimento para os utilizadores com a intenção de criar *wrappers* para algoritmos de aprendizado de máquina, esse padrão é possível através das hierarquias de classes abstratas existentes que abstraem as similaridades entre os algoritmos de mesmo tipo. O UFJF-MLTK pode ser facilmente instanciado a partir de seus pontos de extensão e pode ser utilizado por desenvolvedores com diferentes níveis de conhecimento de sua arquitetura. Sendo esses desenvolvedores, aqueles que a conhecem com profundidade, que adicionam componentes e algoritmos ao UFJF-MLTK, os que tem conhecimento apenas de sua parte pública, ou seja, a sua API, e pelos usuários que não tem contato direto com a sua estrutura, que utilizam o UFJF-MLTK através de sua interface pela linha de comando do sistema operacional utilizado e fazem uso dela

como uma caixa de ferramentas.

Essas características fazem com que o UFJF-MLTK seja um esforço para atacar as dificuldades descritas que muitos pesquisadores e desenvolvedores de aprendizado de máquina enfrentam ao tentar achar o melhor algoritmo para os problemas que estão trabalhando. O padrão proposto pelo mesmo também diminui a dificuldade de compreender o código criado por diferentes desenvolvedores que utilizam o padrão do UFJF-MLTK.

6.1 Trabalhos futuros

Para que se escolha o melhor algoritmo para um dado problema, é necessário que o *framework* tenha a implementação de diversos algoritmos para se realizar comparações de resultados. Portanto, é importante que se faça a expansão do UFJF-MLTK através da inclusão de algoritmos já em desenvolvimento, como algoritmos de multiclassificação, de aprendizado semissupervisionado e o IMA Regressor, assim como a inclusão de outros algoritmos "clássicos", como o k-NN (ALTMAN, 1992), *k-means* (MACQUEEN et al., 1994), LMS (HAYKIN; WIDROW, 2003), *AdaBoost* (FREUND; SCHAPIRE, 1997) etc. Também é importante aumentar a abrangência dos problemas de aprendizado de máquina suportados pela inclusão de pontos de extensão que adicionem o suporte para outros tipos de algoritmos de aprendizado, entre eles, podemos citar os pontos de extensão para algoritmos de regressão, que encontra-se em desenvolvimento, de agrupamento e algoritmos do tipo *ensemble*.

Durante a implementação dos algoritmos de seleção de características, foi possível observar que existiam muitos pedaços de código em comum entre eles, como por exemplo, na criação dos *wrappers* para os métodos de seleção em características em filtro, bastava alterar a função de ranqueamento das características de um método para que se tivesse outro método. Isso mostra que é possível a criação de um ponto de extensão para esse tipo de método de seleção de característica, e possivelmente, também seja viável a criação para os outros tipos.

Como foi visto, a utilização de funções *kernel* permite a solução de problemas não lineares sem a necessidade de mapear o espaço de características para uma dimensão superior manualmente. Até o momento o UFJF-MLTK implementa apenas as funções

kernel linear, polinomial e gaussiana. A implementação de novas funções *kernel* pode aumentar as possibilidades de solução para o usuário. E também, nem sempre as funções implementadas são as que o usuário realmente quer, portanto é fundamental implementar um meio de dar a possibilidade de usar funções *kernel* definidas pelo usuário.

Nem sempre a aplicação de uma técnica de aprendizado de máquina, mesmo em bases de dados simples, resulta em modelos lineares e atualmente a ferramenta de visualização do UFJF-MLTK é capaz de mostrar apenas esse tipo de modelo. Consequentemente, o componente de visualização precisa ser modificado para mostrar modelos não lineares. Com o suporte para modelos não lineares, vai ser possível visualizar melhor o resultado dos algoritmos em suas formulações dual. Também é importante a adição de métodos na classe de visualização para alterar as informações dos gráficos, como por exemplo, métodos para alterar o título, o nome dos eixos, o nome e a cor dos rótulos dos exemplos etc. Também é interessante incluir a possibilidade de gerar gráficos com as curvas de aprendizado dos algoritmos em uma base de dados, isso seria útil para a utilização do UFJF-MLTK no auxílio do ensino de aprendizado de máquina.

O foco inicial dos algoritmos implementados no UFJF-MLTK é o problema de classificação binária. Com a inclusão de suporte para mais tipos de algoritmo de aprendizado, também vai ser necessário implementar novos meios de se observar o comportamento dos algoritmos e de validá-los. Por exemplo, com a inclusão do suporte para algoritmos de multiclassificação é importante a implementação de um algoritmo para gerar a matriz de confusão dos modelos gerados pelos algoritmos desse tipo.

Bibliografia

Abadi, M.; Barham, P.; Chen, J.; Chen, Z.; Davis, A.; Dean, J.; Devin, M.; Ghemawat, S.; Irving, G.; Isard, M. ; others. **Tensorflow: a system for large-scale machine learning.** In: OSDI, volume 16, p. 265–283, 2016.

Alpaydin, E. **Introduction to machine learning**. The MIT Press, 2010.

Altman, N. S. An introduction to kernel and nearest-neighbor nonparametric regression. **The American Statistician**, v.46, n.3, p. 175–185, 1992.

Barber, D. **Bayesian reasoning and machine learning**. Cambridge University Press, 2012.

Bishop, C. M.; others. **Neural networks for pattern recognition**. Oxford university press, 1995.

Bishop, C. M. **Pattern recognition and machine learning (information science and statistics)**. Springer-Verlag, Berlin, Heidelberg, 2006.

Chapelle, O.; Schlkopf, B. ; Zien, A. **Semi-supervised learning**. The MIT Press, 1st. ed., 2010.

Chang, C.-C.; Lin, C.-J. Libsvm: a library for support vector machines. **ACM transactions on intelligent systems and technology (TIST)**, v.2, n.3, p. 27, 2011.

Chandrashekar, G.; Sahin, F. **A survey on feature selection methods**. volume 40, p. 16 – 28, 2014. 40th-year commemorative issue.

Duda, R. O.; Hart, P. E. ; Stork, D. G. **Pattern classification**. John Wiley & Sons, 2012.

Fayad, M. E.; Schmidt, D. C. ; Johnson, R. E. **Building application frameworks: Object-oriented foundations of framework design**. John Wiley & Sons, Inc., New York, NY, USA, 1999.

Freund, Y.; Schapire, R. E. A decision-theoretic generalization of on-line learning and an application to boosting. **Journal of computer and system sciences**, v.55, n.1, p. 119–139, 1997.

Golub, T. R.; Slonim, D. K.; Tamayo, P.; Huard, C.; Gaasenbeek, M.; Mesirov, J. P.; Coller, H.; Loh, M. L.; Downing, J. R.; Caligiuri, M. A. ; others. Molecular classification of cancer: class discovery and class prediction by gene expression monitoring. **science**, v.286, n.5439, p. 531–537, 1999.

Guyon, I.; Weston, J.; Barnhill, S. ; Vapnik, V. Gene selection for cancer classification using support vector machines. **Machine learning**, v.46, n.1-3, p. 389–422, 2002.

Hall, M.; Frank, E.; Holmes, G.; Pfahringer, B.; Reutemann, P. ; Witten, I. H. The weka data mining software: an update. **ACM SIGKDD explorations newsletter**, v.11, n.1, p. 10–18, 2009.

Hatledal, L. I.; Sanfilippo, F. ; Zhang, H. **Jiop: A java intelligent optimisation and machine learning framework.** In: ECMS, p. 101–107, 2014.

Haykin, S.; Widrow, B. **Least-mean-square adaptive filters**, volume 31. John Wiley & Sons, 2003.

Johnson, R. E. **Documenting frameworks using patterns**. In: ACM Sigplan Notices, volume 27, p. 63–76. ACM, 1992.

King, D. E. Dlib-ml: A machine learning toolkit. **Journal of Machine Learning Research**, v.10, n.Jul, p. 1755–1758, 2009.

Kohavi, R.; John, G.; Long, R.; Manley, D. ; Pfleger, K. **Mlc++: A machine learning library in c++**. In: Proceedings Sixth International Conference on Tools with Artificial Intelligence. TAI 94, p. 740–743. IEEE, 1994.

Kohavi, R.; others. **A study of cross-validation and bootstrap for accuracy estimation and model selection**. In: Ijcai, volume 14, p. 1137–1145. Montreal, Canada, 1995.

Leite, S. C.; Neto, R. F. Incremental margin algorithm for large margin classifiers. **Neurocomputing**, v.71, n.7-9, p. 1550–1560, 2008.

MacQueen, J.; others. **Some methods for classification and analysis of multivariate observations**. In: Proceedings of the fifth Berkeley symposium on mathematical statistics and probability, volume 1, p. 281–297. Oakland, CA, USA, 1967.

Eduardo Markiewicz, M.; Lucena, C. **Object oriented framework development**. In: Crossroads, volume 7, p. 3–9. 07 2001.

Mattsson, M. Object-oriented frameworks. **Licentiate thesis**, 1996.

Mathias Filho, I.; de Lucena, P. **A Documentação e a Instanciação de Frameworks Orientados a Objetos**. 2002. Tese de Doutorado - Tese de Doutorado, PUC–Rio.

Mohri, M.; Rostamizadeh, A. ; Talwalkar, A. **Foundations of machine learning**. MIT press, 2012.

Meyers, S. **Effective modern C++: 42 specific ways to improve your use of C++ 11 and C++ 14**. "O'Reilly Media, Inc.", 2015.

Mitchell, T. M. **Machine learning**. McGraw-Hill, Inc., New York, NY, USA, 1. ed., 1997.

da Silva Oliveira, L. L.; Loiola, E. M. ; da Silveira, D. S. **Um framework para instanciação de blogs acessíveis visando os usuários que necessitam usar leitores de tela**. Universidade de Pernambuco, 2011.

Pedregosa, F.; Varoquaux, G.; Gramfort, A.; Michel, V.; Thirion, B.; Grisel, O.; Blondel, M.; Prettenhofer, P.; Weiss, R.; Dubourg, V. ; others. Scikit-learn: Machine learning in python. **Journal of machine learning research**, v.12, n.Oct, p. 2825–2830, 2011.

Platt, J. Sequential minimal optimization: A fast algorithm for training support vector machines. 1998.

Reddy, M. **Api design for c++**. Elsevier, 2011.

Riehle, D. **Framework design: A role modeling approach**. Swiss Federal Institute of Technology, 2000.

Rosenblatt, F. The perceptron: a probabilistic model for information storage and organization in the brain. **Psychological review**, v.65, n.6, p. 386, 1958.

Russell, S.; Norvig, P. **Artificial intelligence: A modern approach**. Prentice Hall Press, Upper Saddle River, NJ, USA, 3rd. ed., 2009.

Cortes, C.; Vapnik, V. **Support-vector networks**. volume 20, p. 273–297. Springer, 1995.

Villela, S. M.; Xavier, A. E. ; Neto, R. F. **Seleção de características com busca ordenada e classificadores de larga margem**. Universidade Federal do Rio de Janeiro, COPPE, Programa de Engenharia de Sistemas e Computação, 2011.

Villela, S. M.; de Castro Leite, S. ; Neto, R. F. Incremental p-margin algorithm for classification with arbitrary norm. **Pattern Recognition**, v.55, p. 261–272, 2016.

Wolpert, D. H. **The lack of a priori distinctions between learning algorithms**. volume 8, p. 1341–1390. Oct 1996.