Centro Federal de Educação Tecnológica de Minas Gerais
Campus Divinópolis
Graduação em Engenharia Mecatrônica

Tarcísio Balbi Veloso Chaboudet

Técnicas inteligentes para manutenção de computações frequentes em ambientes com restrição de memória

Divinópolis
2022

Tarcísio Balbi Veloso Chaboudet

Técnicas inteligentes para manutenção de computações frequentes em ambientes com restrição de memória

Monografia de Trabalho de Conclusão de Curso apresentada ao Colegiado de Graduação em Engenharia Mecatrônica como parte dos requisitos exigidos para a obtenção do título de Engenhario Mecatrônico.
Eixo de Formação: Computação.

Orientador: Michel Pires da Silva

Divinópolis
2022

Centro Federal de Educação Tecnológica de Minas Gerais
CEFET-MG / Divinópolis
Curso de Engenharia Mecatrônica

Monografia entitulada “Técnicas inteligentes para manutenção de computações frequentes em ambientes com restrição de memória”, de autoria do(as) graduando(as) Tarcísio Balbi Veloso Chaboudet, aprovada pela banca examinadora constituída pelos seguintes professores:

Michel Pires da Silva

Eduardo Habib Bechelane Maia

Tiago Alves de Oliveira

Coordenador do Curso de Engenharia Mecatrônica

Marlon Antônio Pinheiro

Divinópolis

Janeiro de 2022

Resumo

Impulsionado pelos fenômenos da quarta revolução industrial e da sociedade 5.0, o *Big Data* trouxe consigo os desafios atrelados ao processamento de quantidades massivas de dados. Nesse cenário de alta demanda computacional, a utilização de técnicas inteligentes para tornar o processamento mais eficiente tem mostrado grande relevância. Uma dessas técnicas é a memoização, que consiste em armazenar computações parciais em *cache* para evitar retrabalho e tornar aplicações mais rápidas. Porém, como a capacidade das memórias é limitada, decidir o que será armazenado e o que será descartado não é uma tarefa trivial. Dessa forma, neste trabalho é proposto um sistema tomador de decisão baseado em políticas de gerenciamento de memória associadas a um modelo preditivo de aprendizado de máquina, cujo objetivo é aumentar a eficiência do armazenamento de computações comuns. A estratégia proposta busca transitar entre os modelos estático e dinâmico de *cache*, decidindo não só o que será descartado, mas se algo realmente deve ser descartado. Nas etapas iniciais, foram estudadas as políticas de gerenciamento mais comuns e testados, em três conjuntos de dados, dois modelos de aprendizado de máquina: *K-means* e redes neurais siamesas. Como resultado, foram escolhidas as políticas FIFO e LRU, com base nos estudos teóricos, e a redes neurais, que apresentaram desempenho superior ao algorítmo de agrupamento. Em seguida, foram realizados experimentos utilizando como aplicação geradora de computações o LAC (*Lazy Associative Classifier*), para o qual foram testadas as políticas de *cache* puras e em conjunto com decisores aleatório e neural. Observou-se que a política híbrida proposta gera ganhos expressivos em relação ao *baseline* em termos de acertos, porém o processo de decisão da rede neural é custoso em tempo.

Palavras-chave: Memoização, política de *cache*, aprendizado de máquina, redes neurais siamesas.

Abstract

Driven by the phenomena of the fourth industrial revolution and society 5.0, Big Data brought the challenges associated with processing massive amounts of data. In this scenario of high computational demand, intelligent techniques to make processing more efficient have shown great relevance. One of these techniques is memoization, which consists of storing partial computations in the cache to avoid rework and make applications faster. However, as the memory capacity is limited, deciding what will be stored and discarded is not a simple task. Thus, this work proposes a decision-making system based on memory management policies associated with a predictive machine learning model, whose objective is to increase the efficiency of the storage of standard computations. The proposed strategy seeks to move between the static and dynamic models of cache, deciding what will be discarded and if something really should be discarded. In the initial stages, the most common management policies were studied and, with three datasets, were tested two machine learning models: K-means and Siamese neural networks. As a result, FIFO and LRU policies were chosen based on theoretical studies and neural networks, presenting superior performance to the clustering algorithm. Then, experiments were carried out using the LAC (Lazy Associative Classifier) as a computational application, for which cache policies and sets with arbitrary and neural deciders were tested. It was observed that the proposed hybrid policy generates significant gains compared to baseline in terms of hits, but the neural network decision process is time-consuming.

Keywords: Memoization, cache politics, machine learning, siamese neural networks.

Sumário

Lista de figuras

Lista de tabelas

Lista de acrônimos e notações

FIFO - *First In First Out*

PLRU - *Pseudo - Last Recently Used*

MRU - *Most Recently Used*

IoT - *Internet of Things*

FNN - *Feedforward Neural Network*

LLF - *Least Fresh First*

LRU - *Least Recently Used*

LFU - *Least Frequently Used*

EBR - *Effectiveness Based Replacement*

ReLU - *Rectified Linear Unit*

VP - Verdadeiro Positivo

VN - Verdadeiro Negativo

FP - Falso Positivo

FN - Falso Negativo

LAC - *Lazy Associative Classification*

CAR - *Class Association Rules*

AC - *Associative Classifiers*

RAM - *Random Access Memory*

UCI - *University of California, Irvine*

MAE - *Mean Absolute Error*

MSE - *Mean Squared Error*

RMSE - *Root Mean Squared Error*

GPU - *Graphics Processing Unit*

Capítulo 1

Introdução e contextualização

Existem dois grandes paradigmas marcam os tempos modernos: i) a quarta revolução industrial, alavancada pelos processos de automação em larga escala, pela internet das coisas e pela computação em nuvem e; ii) a sociedade 5.0, responsável pelo rápido progresso na transformação da informação e das comunicações. (LASI et al., 2014; DALENOGARE et al., 2018; FUKUDA, 2020)

Em meio às evoluções dos paradigmas supracitados, evidencia-se um conceito conhecido como *Big Data*, comumente atrelado a conjuntos de informação compostos por diversas fontes, variados formatos e produzido massivamente em tempo real. Em uma era na qual dados tendem a ser equiparados ao novo petróleo, o eficiente tratamento da informação se faz de grande interesse e importância.(BARUH; POPESCU, 2017; NERSESSIAN, 2018)

Embora haja grande interesse em *Big Data*, a capacidade de processamento e quantidade de memória podem, muitas vezes, se apresentar como limitadores. Exemplos disso podem ser observados na robótica móvel, veículos autônomos, internet das coisas e sistemas embarcados de forma geral (CASTELLANO et al., 2020; CASTELLÓ et al., 2020). Nesses cenários, o reduzido poder computacional torna necessário o uso de estratégias mais inteligentes, principalmente, para aplicações de tempo real e/ou insensivas em dados cujas execuções produzem inúmeras computações caras, que demandam espaços de armazenamento considerável.

Há mais de três décadas, o emprego de técnicas como *cache* e memoização sob uma hierarquia de memória bem definida tem sido uma solução viável para impulsionar aplicações na busca por alto desempenho (EFFELSBERG; HAERDER, 1984). Nesse contexto,

os conjuntos de dados, embora extensos em número de entradas, são finitos em características e dimensionalidade, o que leva a execução de uma série de computações sobrepostas. Assim, aplicações que não utilizam *cache* re-computam todas as sobreposições, implicando no uso ineficiente de recursos computacionais e de energia, principalmente sob condições extremas de demanda de processamento (TANENBAUM, 2016).

O armazenamento de computações sobrepostas é uma ferramenta poderosa, entretanto, também tem seu custo. A quantidade de memória de alta velocidade disponível nos sistemas é baixa, principalmente em sistemas embarcados. Dessa forma, torna-se necessário decidir o que manter em *cache* e o que descartar (*dumped*). Para isso existem as chamadas políticas de *cache* (REINEKE; GRUND, 2013).

Em um cenário de aplicações cada vez mais exigentes, em termos de recursos e de volume de dados, orquestrar um *cache* sob suas políticas se mostra um processo não trivial. Isso ocorre porque o valor de uma computação depende de quão recorrente ela é e de quantos acertos (*hits*) em *cache* ela irá produzir. Logo, predizer sua necessidade com antecedência pode contribuir significativamente com a eficiência do sistema, embora essa não seja de uma tarefa simples e direta (JAVAID et al., 2017).

Visando contornar essas dificuldades, é possível introduzir, em conjunto às políticas de *cache*, métricas de similaridade. Nessas estratégias, as requisições que produzem as computações anteriormente citadas são agrupadas pela capacidade de sobreposição. Isso evita que a *cache* seja esvaziada prematuramente, o que induz ao aumento significativo da taxa de acertos *hits* em computações comuns ao passo que reduz o tempo de execução da aplicação (PIRES et al., 2019). Nesse sentido, técnicas em ciência de dados e aprendizado de máquina podem ser amplamente abordadas devido ao seu poder de generalização, variando desde algoritmos de agrupamento (*clustering*), até redes neurais (CARTWRIGHT, 2015; KUBAT, 2017).

Objetivando contribuir com o processo preditivo de computações comuns e com a eficiência de armazenamento em *cache*, este trabalho investiga esses mecanismos inteligentes de manutenção de computações comuns, a fim de criar um sistema de decisão que maximize os acertos em memória e diminua o tempo de execução das aplicações. Para tal, são investigadas políticas de *cache* existentes na literatura, bem como a utilização de redes neurais em conjunto com essas políticas.

1.1 Definição do problema

Dado um conjunto de parâmetros de configuração e/ou execução, T, e um conjunto de dados de entrada, D, temos, para cada sub-expressão $\{t_j \in T \| t_i \in T\}$, uma série de computações parciais de elevado custo em D. Em vista da natureza finita dessas computações, tais sub-expressões podem se sobrepor no espaço para diferentes entradas em T. Dessa forma, o armazenamento em *cache* dessas computações, consideradas comuns e/ou recorrentes, pode evitar reprocessamento indevido, diminuindo assim o custo computacional da aplicação.

Sob a ótica acima apresentada, é realizado, neste trabalho, o estudo de mecanismos eficientes que contribuam para a manutenção da *cache*. Para tanto, computações recorrentes são avaliadas a partir de seu grau de relevância. Se o valor futuro de tal computação for considerado alto, essa é então submetida à *cache*, a qual orquestra sua alocação segundo política adotada. Caso contrário, descarta-se a mesma objetivando manter sob a *cache* apenas grupos mais relevantes de computações comuns. A Figura 1.1 detalha, sob uma perspectiva de alto nível, como tal mecanismo atua sob o espaço de *cache*.

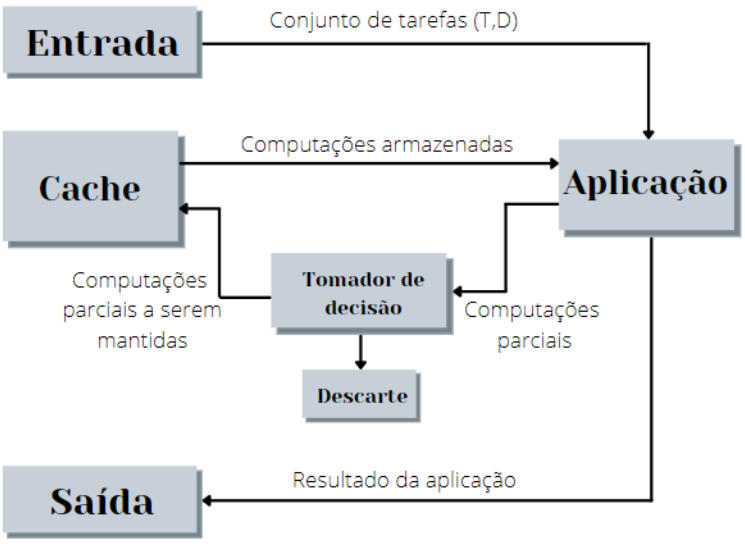

Figura 1.1 – Diagrama de blocos do sistema da aplicação.

1.2 Motivação

Muitas aplicações contemporâneas lidam com processamento intenso de dados, o que as conduz a cenários com múltiplas computações, muitas dessas caras em tempo de exe-

cução e consumo de recursos. Nesse contexto, é comum que tais computações sejam realizadas em conjuntos de dados finitos de característica investigatória, o que pode introduzir sobreposição a partir de entradas similares.

A medida que as sobreposições crescem, recursos como processador e memória são ineficientemente utilizados. Fato esse já discutido há mais de quatro décadas, em que a solução se faz pelo armazenamento das computações comuns por meio de estratégias de *cache* e/ou memoização (EFFELSBERG; HAERDER, 1984; PIRES et al., 2019).

Embora eficazes, essas estratégias têm se mostrado rudimentares sob cenários que envolvem grandes quantidades de dados e volatilidade de contexto. Dessa forma, muitas das computações ditas comuns podem perder a importância e/ou interesse ao decorrer do tempo, fazendo com que sua retenção, em espaços de memória limitadas, seja um problema. Desta limitação surge a necessidade de investigar técnicas mais inteligentes e eficazes de gerenciar a manutenção dessas computações.

1.3 Objetivos

Esta seção trata dos objetivos geral e específicos visados na confecção do trabalho.

1.3.1 Objetivo Geral

O objetivo geral deste trabalho é desenvolver, implementar e avaliar políticas de gerenciamento de memória no contexto de análise de similaridade de computações, utilizando técnicas inteligentes.

1.3.2 Objetivos Específicos

- Realizar um estudo acerca das principais políticas de *cache* no estado da arte;
- Realizar um estudo acerca de redes neurais artificiais e técnicas inteligentes adotadas na tomada de decisões futuras;
- Realizar um estudo acerca do fluxo de dados em ambientes com espaço de memória restrito (*cache*);
- Definir o modelo de avaliação e a metodologia de testes para as técnicas implementadas;

- Definir a estrutura da técnica inteligente adotada e avaliar seu desempenho e suas restrições em diferentes cenários;
- Implementar e avaliar as políticas de *cache* selecionadas;
- Definir uma política de *cache* híbrida juntando estratégias já consolidadas com nossa proposta de tomada de decisões sobre computações futuras;
- Implementar e avaliar a nova política desenvolvida.

1.4 Contribuições

Em resumo, as maiores contribuições deste trabalho são:

- Elaboração de uma estratégia inteligente de descarte de computações comuns em *cache*, cuja decisão se faz pela possibilidade da frequência de requisições.
- Uma revisão do estado da arte em estratégias de predição capazes de atuar no problema de detecção e agrupamento de computações comuns. Além disso, espera-se mapear estratégias capazes de tomar decisões a partir de características mineradas nas requisições de entrada.
- Instigar a comunidade científica para o estudo de políticas eficientes de gerenciamento de memórias de capacidade restrita.

1.5 Organização do trabalho

O presente trabalho é dividido em cinco capítulos. No primeiro deles, foram expostos a definição do problema, bem com a motivação, objetivos e organização do trabalho.

No segundo capítulo, é realizada a revisão de literatura, contando com o estado da arte e a fundamentação teórica a respeito de memórias, de politicas de *cache* e de redes neurais artificiais, sendo esses os conceitos-chave para a realização deste trabalho.

No terceiro capítulo, é apresentada a metologia empregada no desenvolvimento da pesquisa, desde o processo de seleção das políticas e da técnica inteligente até os métodos gerais de implementação e avaliação do sistema proposto.

No penúltimo capítulo, são expostos e discutidos os resultados dos processos supracitados.

Por fim, o capítulo cinco apresenta as considerações finais sobre os resultados obtidos, as limitações encontradas e as propostas de continuidade.

Capítulo 2

Revisão de literatura

A utilização de políticas de *cache* e de momoização tem sido estudada há décadas, mas a crescente demanda por poder computacional mantém o tópico no foco de muitos trabalhos ainda hoje.

Em Denning (1970), discute-se a utilização de memória virtual para modularizar programas dentro de uma máquina. Seus estudos mostraram que a combinação de *hardware* e políticas de gerenciamento de memória pode mitigar problemas de má utilização de recursos. Seguindo essa problemática, diversos trabalhos apresentaram diferentes políticas de gerenciamento de *cache*. Em REINEKE e GRUND (2013), a sensibilidade de uma ampla gama de algoritmos foi investigada e a influência do histórico de execução e seu impacto no número de acertos em *cache* observados e relatados. Dentre os resultados, há evidência do forte impacto do histórico em algoritmos como *first-in-first-out (FIFO)*, *pseudo-least-recently-used (PLRU)* e *most-recently-used (MRU)*.

Em JAVAID et al. (2017), um número ainda maior de algoritmos e técnicas foi investigado sob óticas como complexidade, custo, taxa de erros e acertos, entre outros. O resultado foi uma visão geral, tabular, da comparação entre múltiplos métodos, na qual o método de *Victim cache* se mostrou mais consistente que os demais.

Diferente dos trabalhos acima citados, no decorrer dos anos, foram desenvolvidos diversos estudos acerca do reconhecimento de padrão e análise de similaridade utilizando as mais diversas técnicas e para os mais variados contextos (LIN, 1998).

Em um primeiro momento, análises de similaridade baseadas em distâncias entre vetores e outras medidas estatísticas e matemáticas foram amplamente utilizadas. (LEE; KIM; LEE, 1993; LIN, 1998). Nos últimos anos, os avanços na área de inteligência artificial,

no campo de aprendizado de máquina (*Machine Learning*) e, mais especificamente, em aprendizado profundo (*Deep Learning*), possibilitaram o surgimento de novos e poderosos algoritmos para reconhecimento de padrões (LIU et al., 2017). Nesse contexto, a similaridade pode ser observada pelo agrupamento dos dados (*Clustering*) ou pela comparação das saídas provenientes das redes neurais (regressão ou classificação).

Considerando as abordagens emergentes em inteligência artificial, buscamos, neste trabalho, observar o comportamento das redes neurais siamesas para a classificação de computações. Esse modelo de rede neural foi proposta primeiramente por Bromley et al. (1993) para verificar a similaridade, e consequente autenticidade, de assinaturas. Seus resultados mostram a viabilidade desse método para identificar fraudes.

Sabendo-se que, em Xu e Wunsch (2005), diversos algoritmos de agrupamento, bem como métricas de similaridade, são estudados e têm suas aplicações ilustradas em alguns conjuntos de dados de teste. Utilizaremos desses algoritmos como *baseline* para determinar se a rede neural escolhida se adéqua a problemática apontada por este trabalho.

2.1 Estado da arte

Políticas de *cache* e análise de similaridade vêm sendo foco de trabalhos há anos. Os estudos mais recentes na área têm sido motivados pelos fenômenos do *Big Data*, da Internet das Coisas (*IoT*) e pela demanda crescente de poder computacional.

Em Fedchenko, Neglia e Ribeiro (2018), os autores propuseram uma política de *cache* baseada em métricas de popularidade, sendo essa a fração das requisições por determinado conteúdo em um determinado horizonte de tempo. O preditor da popularidade é baseado em uma rede neural do tipo *feedforward*. Seus resultados mostraram que a utilização de um *FNN* para essa finalidade não tem vantagens consideráveis sobre métodos mais simples e baratos, como estimadores lineares.

Em Meddeb et al. (2019), uma nova política de *cache* baseada em *data freshness* foi proposta para evitar a obsolescência de dados em redes de *IoT*. O método, denominado *Least Fresh First (LLF)*, mostrou-se eficaz em manter dados atualizados, ao mesmo tempo que foi capaz de mitigar a taxa de redução de acertos do servidor e melhorar a latência de resposta.

Em Pires et al. (2019), métricas de similaridade baseadas em *Page Rank* foram utiliza-

das para reordenar um conjunto de computações por afinidade, de maneira a suplementar estratégias típicas de *cache*. Como resultado, ganhos de até 60% em tempo de execução e de até 40% em taxa de acertos foram atingidos.

No trabalho de Lee, Lee et al. (2021) foi proposta uma rede neural convolucional siamesa para detecção de mudanças em áreas urbanas utilizando imagens aéreas. Segundo os autores, as técnicas existentes baseadas somente em redes convolucionais tendem a perder desempenho nesses cenários por conta de métricas de similaridade básicas. Os estudos revelaram que o emprego da nova técnica superou o então estado da arte, atingindo pontuações melhores nas três bases de dados testadas.

2.2 Fundamentação Teórica

A seguir, são apresentados conceitos importantes sobre algoritmos e técnicas relacionados às políticas de *cache* e ao aprendizado de máquina.

2.2.1 Políticas de *cache*

O propósito inicial do armazenamento de dados em memória de acesso rápido é diminuir a lacuna de velocidade entre o processador e as memórias principais, mantendo salvas as informações mais importantes para a execução das aplicações.(JAVAID et al., 2017; REINEKE; GRUND, 2013)

Em outros contextos, como é o caso deste trabalho, o conceito de *cache* pode ser extrapolado, sendo empregado no armazenamento de computações parciais. Nesses casos, o objetivo é diminuir o tempo de execução total da aplicação.

Devido ao tamanho limitado dessas memórias de alta velocidade, seu conteúdo deve ser bem gerenciado. A fim de maximizar os acertos, ou seja, o número de requisições atendidas com sucesso pelo conteúdo presente na memória, entram as políticas de *cache*, responsáveis por decidir o que deve ser mantido e o que deve ser descartado.(JAVAID et al., 2017)

A literatura apresenta um número extenso de algoritmos destinados à substituição e realocação de páginas em *cache*. Pelo escopo limitado do trabalho, são abordados apenas os mais tradicionais e alguns dos mais recentes.

2.2.1.1 Algoritmo ótimo de substituição

O primeiro algoritmo a ser tratado é o chamado ótimo. Esse método, apesar de simples, é impraticável, pois consiste em remover a página cuja referência está mais distante. Ou seja, esse método é irrealizável no mundo real por necessitar do conhecimento prévio de cada chamada do sistema. A única maneira de implementar tal método é executar o programa de antemão, salvando todas as referências nas páginas de memória e depois executá-lo uma segunda vez com a substituição ótima. (TANENBAUM, 2016)

De toda forma, esse procedimento não é inútil, sendo utilizado como base de comparação para demais métodos realizáveis.

2.2.1.2 Algoritmo *first-in, first-out* (FIFO)

O próximo algoritmo estudado é o "primeiro a entrar, primeiro a sair"(*first-in, first-out*, do inglês). Esse método tem foco na simplicidade e no baixo custo computacional. Seu funcionamento se baseia em manter uma lista das páginas da memória com a mais antiga no topo. Quando uma nova requisição acontece, a pagina mais antiga é removida e a nova entra na base. Por não avaliar o valor das páginas em termos do uso, esse método é muito suscitável ao esvaziamento prematuro do *cache*, sendo pouco utilizado na prática. (TANENBAUM, 2016; JAVAID et al., 2017)

2.2.1.3 Algoritmo *second chance*

Para sanar o problema do esvaziamento prematuro, surge o algoritmo da "segunda chance". Esse método é obtido através de uma modificação do FIFO, na qual se acrescenta um *bit* ou uma *flag* R, que indica a referenciação daquela página. Se a R da página mais antiga for 0, então ela pode ser descartada. Porém, se for 1, a página é colocada no início da pilha e seu *bit* R passa a ser 0.(TANENBAUM, 2016)

2.2.1.4 Algoritmo da página usada menos recentemente (LRU)

A política da página usada menos recentemente (*Least Recently Used*, do inglês) baseia-se na premissa de que páginas que foram bastante referenciadas até o momento, provavelmente serão muito úteis também no futuro. Dessa forma, as páginas menos referenciadas também devem permanecer, dessa forma, no futuro e podem ser descartadas.(TANENBAUM, 2016)

O algoritmo constrói uma fila com tamanho igual ao número de páginas ou blocos de memórias armazenados e os ordena de acordo com sua recência. A página mais recente está no início da lista, ao passo que a menos recente está no final, onde é realizado o descarte. Na prática, esse método pode ser realizado utilizando, associado a cada bloco, *bits* de envelhecimento, ou contadores, que serão responsáveis por determinar o tempo desde o último chamado daquele conteúdo. Esse aspecto torna o método eficiente e realizável, porém caro computacionalmente. (JAVAID et al., 2017; REINEKE; GRUND, 2013)

2.2.1.5 Algoritmo da página usada menos frequentemente (LFU)

Esse método consiste em manter uma contagem de quantas vezes uma página foi referenciada, substituindo a de menor frequência.(JAVAID et al., 2017). Sua implementação baseia-se em manter uma lista com as contagens associadas a cada bloco na memória, de forma análoga ao LRU.

2.2.1.6 Algoritmo de reposição baseada em efetividade (EBR)

Em Tian e Liebelt (2013) foi proposta uma nova política chamada *Effectiveness Based Replacement*, que consiste em combinar métricas de recência e frequência para gerenciar o conteúdo do *cache*.

Nesse método, cada bloco recebe um contador R, responsável por registrar a informação de frequência e que é zerado quando ocorre a substituição do mesmo. Um segundo contador (E) é responsável por representar o total de tempo t passado desde a última referência ao bloco.

Dessa maneira, a Efetividade ($E(t, t + \Delta t)$) para cada bloco pode ser calculada por:

$$E_i(t, t + \Delta t) = \frac{R_i(t, t + \Delta t)}{\Delta t} \tag{2.1}$$

Quando uma reposição é necessária, cada bloco terá os valores de frequência e recência, sendo possível estimar E para cada um deles. O bloco com menor valor de efetividade é então substituído.

2.2.2 Métricas de similaridade

Diversas técnicas podem ser empregadas para analisar a semelhança entre dois conjuntos de dados. A escolha do método está relacionada a natureza dos dados e ao contexto

no qual se encontra. Dentre elas, algumas das métricas mais típicas são mostradas na Tabela 2.1. (XU; WUNSCH, 2005)

Tabela 2.1 – Métricas de similaridade mais comuns.

Métrica	Equação
Distância Euclidiana	$D_{ij} = \left(\sum_{l=1}^{d} \lvert x_{il} - x_{jl} \rvert^{1/2} \right)^2$
Distância de Minkowski	$D_{ij} = \left(\sum_{l=1}^{d} \lvert x_{il} - x_{jl} \rvert^{1/n} \right)^n$
Similaridade do cosseno	$S_{ij} = \frac{\mathbf{x}_i^T \mathbf{x}_j}{\lVert \mathbf{x}_i \rVert \, \lVert \mathbf{x}_j \rVert}$

2.2.3 Aprendizado de máquina não supervisionado

Dentro do universo do aprendizado de máquina, os algoritmos podem ser classificados, majoritariamente, em dois paradigmas de aprendizagem: não supervisionada e supervisionada.(BRAGA; LUDERMIR; CARVALHO, 2000)

Em primeiro lugar, falaremos dos métodos não supervisionados. Nesses casos, nenhum tipo de rótulo é atribuído aos dados. A finalidade dos algoritmos é encontrar padrões nas entradas e agrupá-las, de acordo com algum critério e suas caracteristicas, em *clusters*. Esses são os chamados algoritmos de agrupamento, dos quais falamos a seguir. (KUBAT, 2017; BRAGA; LUDERMIR; CARVALHO, 2000; BRAMER, 2016)

2.2.3.1 Algoritmos de agrupamento

As métricas de similaridade apresentadas na Seção 2.2.2 podem ser incorporadas a algoritmos mais complexos, a fim de atribuir à cada entrada de um conjunto de dados, uma classe ou um valor em relação ao conjunto como um todo.

Um dos algoritmos de *clustering* mais clássicos, devido a sua simplicidade, é o *K-means*

2.2.3.1.1 Algoritmo *K-means*

O algoritmo *K-means* é um método de agrupamento (*clustering*), baseado na minimização do erro quadrático, sendo provavelmente o modelo mais simples para esse tipo de

tarefa.(XU; WUNSCH, 2005; BRAMER, 2016)

O objetivo desse método é separar os dados em K conjuntos, ou *clusters*, de maneira a reduzir o erro quadrático entre cada membro do grupo e seu centroide, ou seja, a distância Euclidiana no modelo clássico. O valor de K é ajustável, geralmente menor que 5, mas que pode variar bastante de acordo com a natureza do problema.(KUBAT, 2017; BRAMER, 2016)

Uma das técnicas existentes para uma estimativa do valor de k é o chamado método do cotovelo (*elbow method*). Ela consiste em retreinar o modelo com número de *clusters* variando de 1 a um valor arbitrário n, normalmente entre 5 e 10, e avaliá-lo a cada interação utilizando sua inércia, definida na seção 2.2.5.1. Assim, ao final é obtido um gráfico que relaciona o número de *clusters* à inercia, no qual é escolhido o ponto em que a queda se torna menos abrupta, ou seja, o "cotovelo".(GRUS, 2016)

O procedimento para o *K-means* se dá da seguinte forma: inicialmente cria-se *clusters* escolhendo K objetos de forma aleatória. Em seguida, calcula-se a distância de cada ponto para os centroides e atribuiu cada amostra àquele que apresenta a menor distância. Calcula-se, então, o novo centroide para cada um desses grupos formados. Esses passos são repetidos, reposicionando os centroides e re-atribuindo os objetos, até que os grupos não sofram alteração entre iterações. O Algoritmo 1 apresenta o pseudo-código.

Algorithm 1: Algoritmo K-means

$k \longleftarrow NumerodeClusters$

$D \longleftarrow Dados$

$C \longleftarrow Aleatorio(k, D)$

$Parar \longleftarrow Falso$

while $Parar = Falso$ **do**

  $d \longleftarrow AtribuiCentroides(D, C)$

  $New\ C \longleftarrow RecalculaCentroides(d)$

  if $New\ \ C = C$ **then**

    $Parar \longleftarrow Verdadeiro$

  end

end

Além da distância Euclidiana, outras métricas de similaridade, como as apresentadas na seção 2.2.2, podem ser utilizadas no treinamento do modelo. (BORA; GUPTA, 2014)

2.2.4 Aprendizado de máquina supervisionado

A segunda classe de algoritmos de aprendizado de máquina é a dos supervisionados. Nesses métodos cada entrada do conjunto de dados recebe um valor alvo, ou *target*, que pode ser um valor contínuo, como o preço de uma casa, ou uma classe, como ”benigno”ou ”maligno”para um câncer. Dessa forma, é possível comparar a saída do método com o valor real esperado e, assim, determinar o erro do algoritmo. (BRAGA; LUDERMIR; CARVALHO, 2000)

A gama de algoritmos dessa classe é ampla, abrangendo métodos de regressão linear e logística, árvores de decisão, K-vizinhos, redes neurais artificiais e *etc.* Assim, foge do escopo deste trabalho fundamentar cada um deles, em vez disso, focaremos no último citado, as redes neurais. Para aqueles interessados em se aprofundar no assunto, diversos desses métodos podem ser encontrados em Kubat (2017) e Grus (2016).

2.2.4.1 Redes neurais Artificiais

Uma sub-área importante do aprendizado de máquina são as redes neurais artificiais. Essas estruturas tem como elemento básico o neurônio, que realiza, geralmente, uma operação linear com suas entradas, como apresentado na Equação 2.2, onde w é chamado de peso e b de *bias*, ou polarizador.(GRUS, 2016)

$$y_i = w_i * x_i + b_i \tag{2.2}$$

Essas redes são construídas por camadas, sendo cada uma delas composta por n neurônios e por suas funções de ativação, responsáveis por adicionar não linearidades ao sistema. Algumas das funções de ativação mais comuns podem ser encontradas na Tabela 2.2.(CARTWRIGHT, 2015; ALCANTARA, 2017)

Essas estruturas, também conhecida como redes *Feed-forward* por cada neurônio levar informação ao seguinte, comumente apresentam a seguinte estrutura: uma camada de entradas, com o número de neurônios equivalente à dimensão das amostras, seguida por camadas escondidas, que variam em número e quantidade de neurônios, e, por fim, a camada de saída, cujo tamanho está ligado à resposta esperada. Além disso, como cada neurônio representa uma operação linear, tem associado a si um peso, w, e um valor de *bias*, b. (GRUS, 2016)

Tabela 2.2 – Funções de ativação mais comuns.

Função	Equação
Degrau	$y(x) = \begin{cases} 1, se\ x > 0 \\ 0, se\ x \leq 0 \end{cases}$
Sigmoid	$y(x) = \frac{1}{1 + e^{-x}}$
Tanh	$y(x) = \frac{e^{2x} - 1}{e^{2x} - 1}$
ReLU	$y(x) = \begin{cases} x, se\ x > 0 \\ 0, se\ x \leq 0 \end{cases}$
Leaky ReLU	$y(x) = \begin{cases} x, se\ x > 0 \\ \alpha x, se\ x \leq 0 \end{cases}$ Onde α é ajustável, geralmente menor que 1

A Figura 2.1 apresenta a estrutura de uma rede neural genérica. Vale ressaltar que devido ao fato de todos os neurônios da camada anterior se ligarem a todos da posterior, esse tipo de rede também é conhecida como *fully connected*.

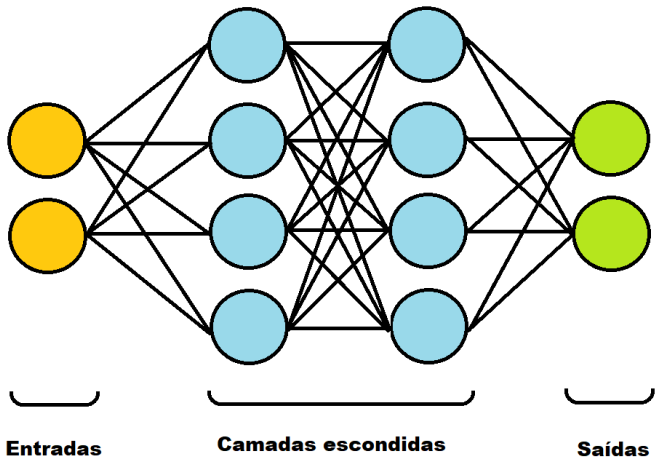

Figura 2.1 – Exemplo de rede neural com duas entradas, duas camadas escondidas e duas saídas. As indicações de pesos e *bias* foram omitidas para simplificação. Fonte: O autor.

O aprendizado dessas redes, ou seja, a atualização dos valores de pesos e *bias* de cada neurônio, se dá pela técnica conhecida como *back propagation*. Cada previsão gerada pela rede é comparada com o valor verdadeiro e o erro calculado é utilizado para modificar os valores de **w** e **b** através de um algoritmo de otimização, como gradiente descendente, por exemplo.(KUBAT, 2017)

2.2.4.1.1 Redes neurais siamesas

As métricas e os métodos apresentados até agora para análise de similaridade funcionam bem para vetores numéricos em um espaço qualquer N. Porém, para conjuntos de dados mais complexos, envolvendo diferentes dimensões e tipos, essas medidas já não são aplicáveis. Nesse contexto entram as redes neurais siamesas. (CARTWRIGHT, 2015)

Apresentadas primeiro em Bromley et al. (1993), essas redes possuem uma estrutura particular, consistindo de duas redes neurais idênticas. As duas metades da rede trabalham de forma paralela e suas saídas são combinadas e trabalhadas conforme o problema. Para o problema proposto por Bromley et al. (1993), por exemplo, as saídas de ambas as redes eram comparadas utilizando a distância de cossenos para dizer se uma assinatura era falsa ou não. A Figura 2.2 apresenta a estrutura genérica de uma rede neural siamesa.

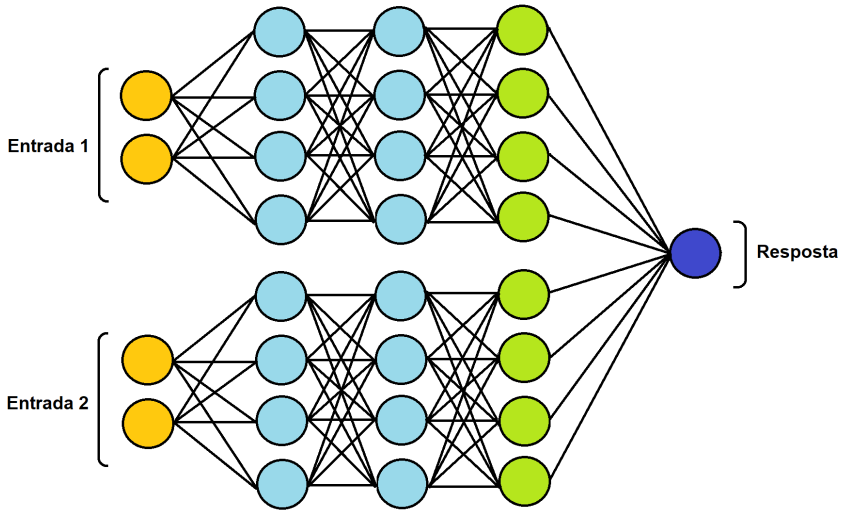

Figura 2.2 – Exemplo de rede neural siamesa com duas entradas, duas camadas escondidas, duas camadas de saída e uma resposta. As indicações de pesos e *bias* foram omitidas para simplificação.
Fonte: O autor.

2.2.5 Métricas de avaliação de modelos

Aqui são apresentadas as principais métricas para avaliação de modelos de aprendizado de máquina.

2.2.5.1 Inércia

Utilizada para avaliação de modelos de aprendizado não supervisionado, como o *K-means*, a inércia representa a soma quadrática das distâncias de cada amostra até o centroide do *cluster* mais próximo. Comumente é utilizada como parâmetro na definição do número ideal de *clusters* pela regra do cotovelo. (GRUS, 2016)

2.2.5.2 Matriz de confusão

A matriz de confusão é um método para avaliação para modelos de aprendizado supervisionado. Ela consiste em formar uma matriz cujas colunas são as categorias reais do conjunto de dados, ao passo que as linhas representam as categorias preditas pelo modelo, conforme o modelo da Tabela 2.3. (CARTWRIGHT, 2015)

Tabela 2.3 – Exemplo de matriz de confusão para duas categorias (Positivo e Negativo)

Valor Predito (↓) Valor real (→)	Positivo	Negativo
Positivo	Verdadeiro positivo (VP)	Falso Positivo (FP)
Negativo	Falso Negativo (FN)	Verdadeiro Negativo (VN)

Esse conceito pode ser expandido para n categorias, provendo uma visão geral dos acertos e erros do modelo.

2.2.5.3 Matriz de migração

A matriz de migração é construída da mesma forma que a de confusão, porém é utilizada quando não há um valor verdadeiro, ou não há relação direta entre o valor verdadeiro e o previsto, como é o caso dos algoritmos de grupamento.

2.2.5.4 Acurácia

A acurácia é a métrica mais simples para modelos de aprendizado supervisionado, representando o número de predições corretas em relação ao total. Em modelos com mais

de duas categorias, é comum encontrarmos a macro acurácia (*macro accuracy*), que representa a média das acurácias de cada categoria e também a acurácia ponderada (*wighted accuracy*) pelo número de amostras de cada categorias(*support*).(CARTWRIGHT, 2015)

$$Acuracia = \frac{(VP + VN)}{Total} \tag{2.3}$$

2.2.5.5 Precisão

A precisão também é utilizada para modelos de aprendizado supervisionado, representando o número de predições positivas corretos dentre todos os positivos preditos.(CARTWRIGHT, 2015)

$$Precisao = \frac{(VP)}{(VP + FP)} \tag{2.4}$$

2.2.5.6 *Recall*

O *Recall* é uma métrica para aprendizado supervisionado que representa o número de predições positivas corretas entre todos os positivos reais.(CARTWRIGHT, 2015)

$$Recall = \frac{(VP)}{(VP + FN)} \tag{2.5}$$

2.2.5.7 *f1-score*

O *f1-score* é obtido pela média harmônica do *recall* e da precisão.(CARTWRIGHT, 2015)

$$f1 = \frac{(TP)}{(TP + \frac{1}{2}(FP + FN))} \tag{2.6}$$

2.2.6 Validação cruzada

A validação cruzada é uma das técnicas mais importantes para o desenvolvimento de um modelo de aprendizado de máquina.

Durante o treino de um modelo, é essencial que sejam dividas porções de treino e de teste. O princípio da validação cruzada dita que esse processo seja feito k vezes de maneira aleatória, o chamado *k-fold cross validation*. Dessa forma, avaliamos o real desempenho

do modelo em todo o conjunto de dados e seu poder de generalização, evitando vieses que poderiam ser introduzidos por uma divisão única. (BRUNTON; KUTZ, 2019)

2.2.7 *Lazy Associative Classification - LAC*

A classificação associativa é uma área do aprendizado de máquina que explora o fato de podemos encontrar fortes associações entre as classes e as variáveis preditoras. Esses algoritmos (*associative classifiers*, ou AC), durante seu treinamento, descobrem essas regras de associação (*Class Association Rules, ou CARs*) e, a partir delas, criam funções que levam um conjunto de variáveis preditoras a uma classe. (VELOSO; MEIRA; GONÇALVES et al., 2011)

Obter as regras de associação é um ponto-chave desses modelos e, encontrar todas elas é frequentemente infactível. Dessa forma, uma classificação associativa preguiçosa (*Lazy Associative Classification, ou LAC*) reduz o número de CARs projetando, do conjunto de treino, apenas o subconjunto de instâncias que contêm as *features* presentes instância de teste. (VELOSO; MEIRA; ZAKI, 2006)

Capítulo 3

Metodologia

Este capítulo apresenta os materiais e métodos empregados para o cumprimento dos objetivos apresentados na seção 1.3. Em primeiro momento, serão expostos os materiais, explicitando linguagens de programação, ambientes e pacotes utilizados. Por último, serão apresentados os passos seguidos para configurar os ambientes, a metodologia de testes proposta e os critérios empregados para obtenção dos resultados.

3.1 Materiais

Esta seção tem por objetivo detalhar todos os materiais físicos e implementáveis necessários para o cumprimento dos objetivos propostos. Nesse sentido, é apresentado o ambiente computacional a ser utilizado para a realização dos experimentos e como esses são executados sob o conceito de *cache* proposto. Na sequência, é discutida a linguagem de programação adotada e os pacotes mais relevantes a serem considerados para a composição de todo material implementável associado ao tema.

3.1.1 Computador

O principal componente para realização deste projeto é um computador, sendo esse o único material físico / *hardware* necessário. Sabendo-se disso, é utilizado um equipamento em conformidade com as configurações detalhadas na Tabela 3.2.

Tabela 3.1 – Configuração do computador de testes

Componente	Especificação
Processador	I7 4770K (*stock*)
Memória RAM	16Gb DDR3 1600 Mhz
Armazenamento	240Gb SSD
Sistema operacional	Windows 10

3.1.2 Google Colab

Para a segunda etapa do projeto é utilizada uma máquina padrão na plataforma Google Colab, com as seguintes especificações disponíveis:

Tabela 3.2 – Configuração do computador de testes

Componente	Especificação
Processador	Intel Xeon @2.2 Ghz Dual Core) (*stock*)
Placa gráfica	-
Memória RAM	12GB
Armazenamento	120GB SSD

3.1.3 *Cache*

Uma memória *cache* se baseia no conceito de acesso rápido à informação. Assim, é muito comum que tal estrutura seja modelada por componentes físicos de alta velocidade ou estruturas de dados muito eficientes que, em sua maioria, apresentam custo computacional constante. Além disso, é usual observar sua organização sob dois conceitos: estático e dinâmico.

No modelo estático, ao carregar a *cache* com um conjunto de regras pertinentes ao problema observado, seu conteúdo não sofre quaisquer modificações ao decorrer do tempo. Em suma, tal conjunto é considerado aceitável para lidar com a maioria dos condicionan-

tes impostos pelo problema. Embora eficiente, muitos dos problemas emergentes não se adaptam a condicionantes lineares de execução, o que produz uma certa oscilação das necessidades ao decorrer do tempo. Tal oscilação induz os modelos de *cache* a se comportarem de forma dinâmica, o que implica no emprego de políticas de gestão que escolhem o que manter na *cache* e o que descartar para a liberação de espaço para novas computações. Independente do modelo adotado, a *cache* utiliza por definição um conceito de armazenamento baseado em chave e valor. Modelo esse apresentado em abstração na Figura 3.1.

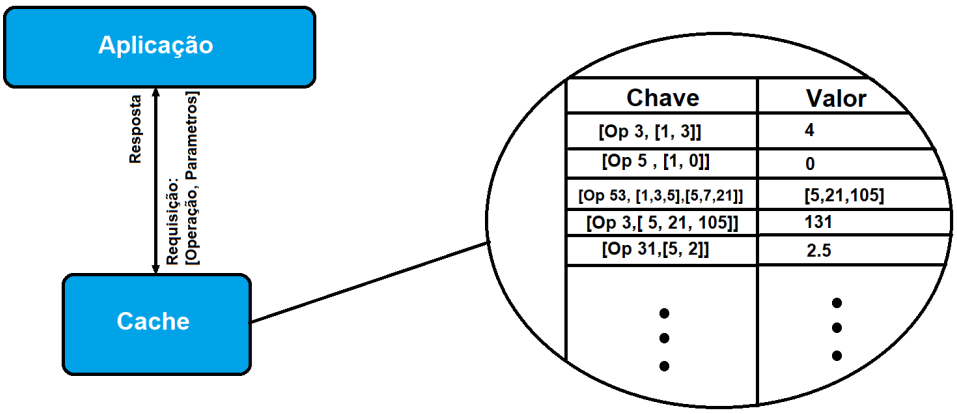

Figura 3.1 – Exemplo da estrutura de *cache* utilizada.
Fonte: O autor.

Neste trabalho, é proposto um modelo híbrido de *cache*, o qual permutará entre uma estrutura estática e dinâmica conforme análise prévia. O objetivo dessa estrutura híbrida é manter regras comuns por mais tempo na *cache* e assim maximizar sua utilização. Como muitas aplicações atuais induzem a computações caras e sobrepostas, armazenar tais computações em *cache*, mantendo-as por mais tempo nessa estrutura, resulta em redução significativa no tempo de execução da aplicação alvo, o que é algo desejável para um universo considerável de áreas da computação e robótica.

Para demonstrar a eficiência de nossa abordagem, é utilizada de uma estrutura *hash*, implementada em memória RAM. Tal estrutura é limitada propositalmente, tendo um algoritmo preditivo como base para coordenar o que deve ser mantido e o que deve ser substituído por instruções mais recentes. Para essa implementação / experimentação é utilizada a linguagem Python, discutida em maiores detalhes na sequência.

3.1.4 Linguagem de programação

Na literatura, há inúmeras linguagens que comportam facilmente a implementação de uma estrutura *hash* com as características acima apresentadas. Contudo, visando abordar tal estrutura sob um modelo híbrido, coordenado por um tomador de decisões preditivo, o *Python* é adotado como linguagem principal(PYTHON…, 2021).

Atualmente, essa é considerada a linguagem de maior utilização em aplicações de análise e tratamento de dados, sendo, provavelmente, a mais popular entre os cientistas de dados e engenheiros de *machine learning*. Esse fato se deve ao seu alto nível de abstração, o que torna mais simples a implementação de algorítimos complexos. Ademais, por ser o foco da comunidade, grande parte dos pacotes de ferramentas voltadas à inteligência artificial foram desenvolvidos para ela, além da fácil integração das ferramentas, possibilitando a criação de ecossistemas fluidos e com maior agilidade (RASCHKA; PATTERSON; NOLET, 2020). Mais informações sobre esses cenários e conceitos que apoiam essa decisão podem ser observados em Siddiqui, AlKadri e Khan (2017).

3.1.5 Principais pacotes

Durante a execução do trabalho, diversos pacotes do *Python* são utilizados. Nesta seção são apresentadas as principais.

3.1.5.1 *Pandas*

Pandas é uma biblioteca *open source* (código aberto) destinada à manipulação de dados estruturados, tanto no formato de séries, quanto de tabelas. Ela permite a leitura e a escrita de diversos tipos de arquivos como *csv* (*comma-separated values*), *Apache parquet* e até mesmo aqueles provenientes diretamente de outros *softwares* de manipulação de dados, como *Microsoft Excel*, por exemplo.

Além de leitura e escrita de arquivos, o pacote apresenta diversas funções para o tratamento de dados, como preenchimento de valores vazios, agrupamento de valores, remoção de duplicatas, junção de tabelas e operações matemáticas diversas.

A documentação completa do pacote pode ser encontradas em PANDAS… (2021).

3.1.5.2 *Numpy*

Numpy é um pacote cujo foco são operações lógicas e matemáticas utilizando matrizes e vetores. O grande diferencial desse pacote é a velocidade de execução das operações, mais rápido que as funções básicas do *Python*. Esse ganho se deve ao fato de suas principais funções serem pré-compiladas em linguagem C.

Entre as principais funções da biblioteca, destacam-se a manipulação de formato, as operações de álgebra linear e estatística e a busca e seleção de dados. Além disso, traz também suas próprias estruturas de dados, chamados de *ndarrays*.

A documentação completa pode ser encontrada em NUMPY… (2021)

3.1.5.3 *TensorFlow*

TensorFlow é uma biblioteca de código aberto desenvolvida pela *Google* cujo foco é o desenvolvimento de modelos de aprendizado de máquina.

Ela integra funções que simplificam a implementação de redes neurais profundas, desde a declaração das camadas, até o treinamento e monitoramento das métricas dos modelos. Além disso, sua implementação possibilita o uso de CPUs, GPUs e também TPUs, que são unidades de processamento especializada em tensores, os componentes base da biblioteca.

A documentação completa da biblioteca pode ser encontrada em TENSORFLOW… (2021)

3.1.5.4 *Keras*

Keras é uma biblioteca que tem em sua base o *TensorFlow*. Seu principal objetivo é encapsular algumas funções e proporcionar uma utilização mais amigável ao usuário, permitindo maior velocidade de experimentação.

A documentação completa do pacote pode ser encontrada em KERAS… (2021)

3.1.5.5 *Scikit-learn*

Scikit-learn é um pacote que traz uma ampla gama de métodos encapsulados de maneira simples para o usuário.

Dentre suas principais aplicações estão as implementações eficientes de algoritmos de aprendizado de máquina, as métricas de avaliação, as funções de validação cruzada e o ajuste de hiper-parâmetros.

Mais sobre essa biblioteca pode ser encontrado em Pedregosa et al. (2011)

3.1.5.6 *Matplotlib*

Matplotlib é uma biblioteca cujo foco é a visualização de dados, possuindo métodos que vão desde o simples *plot* no plano cartesiano, até histogramas, *boxplots* e outros gráficos estatísticos.

Mais sobre esse pacote pode ser encontrado em Hunter (2007)

3.2 Métodos

Para demonstrar que a realização de predições sob o conteúdo da *cache* pode contribuir significativamente com o tempo de execução de muitas das aplicações contemporâneas, é estabelecido um fluxo de trabalho generalista que se encaixa em inúmeras áreas do saber em computação e em mecatrônica. Nesse, há um elemento principal, gerador de computações (i.e., aplicação), referenciado como *master*. Assim, requisições são transformadas em computações e essas associadas a uma resposta final enviada para seu requerente. Para evitar que computações sobrepostas sejam diversas vezes executadas, uma *cache* armazenará essas informaçõs para serem reutilizadas assim que demandadas novamente. O Algoritmo 2 detalha em alto nível o fluxo de execução esperado no *master*.

Algorithm 2: Aplicação modelo designada como *master*

```
k ⟵ Requisicoes(p)
cache ⟵ init
hits ⟵ 0
for i de 1 a p do
    EmCache ⟵ VerificaHit(k[i], cache)
    if EmCache = Falso then
        if cache cheio then
            cache ⟵ Politica(cache)
            cache ⟵ GravaLinha(cache, k[i])
        else
            cache ⟵ GravaLinha(cache, k[i])
    else
        hits ⟵ hits + 1
```

Como pode ser observado no Algoritmo 2, a *cache* passa por momentos de substituição de conteúdo. Isso ocorre sempre que não há mais espaço para armazenar novas regras, processo esse realizado por uma política de substituição. Para computar a quantidade de *hits* em *cache* que cada uma das políticas alcança na sua gestão, são utilizados diferentes cenários de execução.

A composição da aplicação final será feita por uma política de *cache* e por um modelo de aprendizado de máquina. Assim, as seções 3.2.1 e 3.2.2 apresentam detalhadamente as etapas de seleção desses componentes, bem como a metodologia de testes a ser empregada em cada uma.

3.2.1 Etapa 1 - Seleção de componentes

Em vista da abundância de políticas de *cache* e de modelos de aprendizado de máquina, o principal objetivo desta etapa é selecionar, dentre os descritos, um subconjunto desses componentes com maior potencial inicial para integrar o sistema final.

A seleção das políticas é feita com base teórica, escolhendo algoritmos que apresentam relevância e também simplicidade, visto que o foco do trabalho é o sistema tomador de decisões.

Já para os modelos preditores, ambos os apresentados são avaliados em testes iniciais, não sendo descartado nenhum de antemão. Dessa forma, são desenvolvidos ambientes de teste utilizando *Python*. Os ambientes e os critérios de avaliação são descritos a seguir.

3.2.1.1 Testes de modelos preditores

Para testar o poder de generalização dos modelos de *machine learning* são utilizados três conjuntos de dados, com diferentes contextos e dimensionalidades de amostra. Os dados vêm do repositório da Universidade da Califórnia em *Irvine*, *UCI machine learning repository*, e da plataforma *Kaggle*.

O objetivo principal é verificar qual deles consegue agrupar, mais eficientemente, os conjuntos de tarefas de acordo com sua similaridade. Dessa forma, ainda não é aplicada a validação cruzada.

Os testes consistem em, para cada conjunto e para cada modelo, realizar os passos descritos a seguir.

1. Normalizar todas as variáveis preditoras para o intervalo [0,1]

2. Dividir os dados em treino e validação;
3. Treinar o modelo utilizando os dados de treino;
4. Realizar a previsão utilizando os dados de validação;
5. Aplicar as métricas de avaliação;
6. Guardar os resultados obtidos.

Para avaliar o algoritmo *K-means* é utilizada a matriz de migração e avaliação visual dos grupamentos obtidos, dado que outras métricas para modelos não-supervisionados conseguem apenas comparar os resultados no mesmo modelo e conjunto de dados, como a soma das distâncias de cada ponto até o centroide do grupo.

Já para a rede neural, os conjuntos têm um passo de tratamento prévio para adequar seu uso em uma estrutura siamesa. Para cada classe, são obtidas as médias das variáveis, e, então, construídos os pares (amostra, média) para o treinamento e para o teste do modelo. Dessa maneira, avalia-se o poder da rede a rede de prever a similaridade de um objeto com a média de um grupo.

Por fim, como há uma forma clara de associar as categorias preditas às originais, é possível aplicar, além da matriz de confusão, métricas de acurácia, precisão, *recall* e *f1-score*.

Ao fim dos testes, o modelo com melhores métricas de avaliação será o selecionado.

Em posse de uma política de *cache* e de um modelo preditor, é possível organizar um ambiente com um tomador de decisão completo, que recebe a nova computação e decide se ela tem valor futuro maior que o conteúdo atualmente guardado e, em caso positivo, qual fragmento da memória deve ser descartado. A seção seguinte explica em detalhes como tal ambiente é construído.

3.2.2 Etapa 2 - Aplicação com sistema completo

Para a aplicação completa é proposto o fluxo visto na Figura 3.2. É possível abstrair esse sistema como sendo composto por um mestre e um trabalhador, de modo que as tarefes são disparadas e executadas sequencialmente. Além disso, há um decisor que, a cada nova tarefa disparada, estima seu futuro valor e decide se as computações parciais

por ela geradas serão consideradas úteis ou inúteis, liberando ou não a possibilidade de armazenamento.

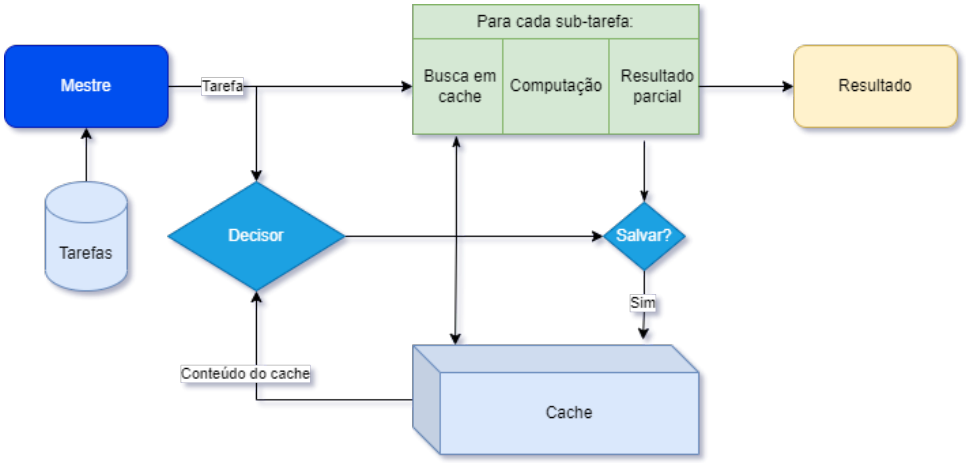

Figura 3.2 – Esquemático da aplicação final, na qual o *Master* dispara as tarefas.
Fonte: O autor.

A execução desse sistema acontece como mostra o Algoritmo 3. Em um primeiro momento, há o aquecimento da memória, ou seja, o sistema é livre para salvar quaisquer computações a partir da política utilizada, até que o *cache* atinja sua capacidade máxima.

A partir desse ponto o decisor, caso presente, começa a atuar alternando o paradigma de armazenamento entre dinâmico e estático. Cada tarefa disparada gera um conjunto de sub-tarefas que deverão ser computadas e, para cada uma delas, verifica-se a presença do seu resultado na memória. Em caso negativo, a sub-tarefa é calculada e sua resposta armazenada ou não, a depender da avaliação do decisor. Por fim, após o cálculo de todas as sub-tarefas, o resultado é gerado.

É utilizada uma classificação associativa preguiçosa (LAC), segundo a descrição da seção 2.2.7, como aplicação base nos testes realizados, que consideram combinações das políticas, das capacidades e dos decisores escolhidos, como apresentado a seguir.

Algorithm 3: Aplicação Completa

```
T ⟵ Requisicoes(t_1, ..., t_k)
D ⟵ DadosDeEntrada
hits ⟵ 0
resApp ⟵ init
cache ⟵ init
for t_i em T do
    resT ⟵ init
    if Cache Cheio then
        Salvar? ⟵ Decisor(cache, t_i, limiar)
    else
        Salvar? ⟵ Sim
    end
    T' ⟵ SubTarefas(t_i)
    for t'_j em T' do
        if t'_j em Cache then
            Comp ⟵ PegaCache(t'_j)
            resT ⟵ Combina(resT, Comp)
            hits ⟵ hits + 1
        else
            Comp ⟵ Processa(t'_j, D)
            resT ⟵ Combina(resT, Comp)
            ValorComp ⟵ Preditor(Comp)
            if Salvar = Sim then
                cache ⟵ Politica(cache, Comp)
            end
        end
    end
    resApp ⟵ Combina(resApp, resT)
end
```

3.2.2.1 Base de dados utilizada

Para a aplicação final é utilizada uma base de dados de preços de casas no estado da Califórnia, EUA. A decisão se deve ao fato de que ela conta apenas com variáveis de ponto flutuante e totalmente preenchidos.

Para utilização nos testes, são necessários tratamentos na base original para que se adequasse ao LAC. O primeiro deles é dividir as colunas cujos valores estavam na escala dos milhares por mil. A seguir, os valores são arredondados para apenas uma casa decimal.

Esses dois primeiros procedimentos visam reduzir a variedade de valores encontrados nas entradas, já que apenas valores vistos no treino são contabilizados para produção de sub-tarefas nas instâncias de teste.

Por último, a variável resposta do *dataset* é modificada, deixando de representar o

valor da casa para representar uma classe. As classes são decididas através de uma quebra volumétrica por quartis.

3.2.2.2 Testes *baseline*

De modo a observar o desempenho das políticas de *cache* de maneira isolada, ou seja, sem influência de nenhum decisor, são realizados testes envolvendo diferentes cenários.

Dessa forma, consideraram-se todas as combinações dos seguintes parâmetros:

- Políticas: [FIFO, LRU]
- Capacidades: [0,100,200,500,1000]

As políticas são as definidas na primeira etapa do projeto, já as capacidades testadas são selecionadas no intervalo de 0, ou seja, sem *cache*, até 1000 posições, quantidade que apresenta elevados usos de memória.

Para aplicação desses testes foi montada uma esteira conforme mostrada no Algoritmo 4, que essencialmente automatiza a troca de parâmetros de testes utilizando laços de repetição e salva os resultados de cada iteração.

3.2.2.3 Aquisição de dados

Após a realização dos testes iniciais, há a etapa de aquisição dos dados para treinamento da rede neural. O primeiro passo é a definição das variáveis de entrada e da variável resposta. Em virtude dos resultados obtidos na primeira etapa do projeto, que mostraram que essa estrutura de rede consegue extrair informações relevantes desse tipo de comparação, as entradas da rede neural são a tarefa geradora das computações e a média das variáveis presentes no *cache* no momento que a nova tarefa chega. Pois, esse é o ponto no qual o decisor deve atuar.

A variável resposta é escolhida em vista do objetivo principal do projeto proposto, determinar o valor futuro das computações. Dessa forma, escolheu-se medir a quantidade de *hits* hipoteticamente gerados por cada tarefa anteriormente salva. A construção das variáveis preditores e resposta é descrita em detalhes a seguir.

São feitas modificações na esteira de testes e na aplicação de maneira que, durante a execução, sejam armazenadas as informações de forma correta. A cada nova tarefa disparada, são salvos a tarefa e todo o conteúdo bruto do *cache*.

Algorithm 4: Esteira de testes

```
runs ⟵ Lista(0, ..., n)
politicas ⟵ Lista(politicas)
capacidades ⟵ Lista(min, ..., max)
resultados ⟵ init
for n em runs do
    for p em politicas do
        for c em capacidades do
            T ⟵ Requisicoes(t_1, ..., t_k)
            D ⟵ DadosDeEntrada
            hits ⟵ 0
            missings ⟵ 0
            resApp ⟵ init
            cache ⟵ init(p, c)
            for t_i em T do
                resT ⟵ init
                T' ⟵ SubTarefas(t_i, m)
                for t'_j em T' do
                    if t'_j em Cache then
                        Comp ⟵ PegaCache(t'_j)
                        resT ⟵ Combina(resT, Comp)
                        hits ⟵ hits + 1
                    else
                        Comp ⟵ Processa(t'_j, D)
                        resT ⟵ Combina(resT, Comp)
                    end
                end
                resApp ⟵ Combina(resApp, resT)
            end
            resultados ⟵ Combina(resultados, resApp)
        end
    end
end
```

Além disso, para o conjunto tarefa-*cache* é iniciado um contador em zero. Assim, a cada acerto procura-se, nas tarefas armazenadas até então, aquelas capazes de terem gerado aquele acerto, ou seja, as que possuem as combinações de parâmetros necessários, e essas têm seus contadores incrementados de um.

Ao final do processo de aquisição, os dados são tratados para calcular a média do *cache* em cada entrada e estruturar a base de dados em entradas gêmeas, sendo as linhas pares as tarefas e as linhas ímpares as médias.

Nesse ponto, são testadas também as combinações de políticas e capacidades apresentadas na etapa anterior, com exceção do *cache* desligado.

3.2.2.4 Estrutura e treinamento da rede neural

Dos dados coletados, é selecionado, segundo a eficiência observada entre consumo de memória e número de acertos, um para servir de base para o treinamento da rede neural.

O segundo passo é a análise exploratória desses dados, para entender a distribuição dos *hits* à medida que a execução avança e, então a base é tratada e dividida em treino e teste.

Com a base definida, o foco se volta para a estrutura da rede neural.

É proposta uma rede neural siamesa cuja estrutura e parâmetros de treinamento são selecionados através de um processo de busca *GridSearch* em duas etapas. Na primeira, são testados hiper-parâmetros de forma mais abrangente, no que diz respeito a faixa de valores. Dessa forma, os melhores parâmetros encontrados servem de ponto inicial para as faixas testadas na segunda etapa, de modo que o modelo finalista é selecionado com base no melhor obtido em qualquer uma das etapas.

A métrica utilizada como alvo na busca dos hiper-parâmetros é a correlação entre o número real de acertos e aqueles previstos pelo modelo.

Essa métrica se deve ao objetivo, propor um limiar que, servindo como ponto de corte, separe tarefas com maior potencial gerador de acerto. Assim, não é fundamental que o número previsto pelo modelo seja próximo ao real, mas sim que sejam fortemente correlacionados em alguma escala.

Outras métricas também são adotadas para avaliação do modelo final, sendo elas a média do erro absoluto (MAE), a média do erro quadrático (MSE) e a raiz quadrada do erro médio quadrático (RMSE).

3.2.2.5 Aplicação com decisores

Nos testes realizados, a decisão da mudança de paradigma é tomada no momento em que a tarefa chega, ou seja, determinará se qualquer nova sub-tarefa terá ou não seus resultados armazenados.

Dessa forma, são testados dois decisores, sendo o primeiro aleatório e o segundo baseado na rede neural treinada.

Para o primeiro, há uma chance de 50% do *cache* ser liberado para armazenamento. Isso é feito para verificar o impacto apenas da mudança de paradigma sobre o desempenho observado.

Para o segundo, quando uma nova tarefa chega, é comparada pela rede neural à média do conteúdo presente no cache naquele momento. Ou seja, o mesmo cenário no qual os dados foram coletados. Dessa forma, se o valor previsto for maior que um determinado limiar, definido previamente através de testes, o *cache* é liberado.

Assim, são desenvolvidos testes nos mesmos cenários anteriormente descritos, com exceção do *cache* com capacidade zero. Além disso, para melhor se aproximar de um cenário de execução real, a ordem das tarefas é aleatorizada e, para cada cenário, 10 testes foram realizados. Ao final da bateria de testes, são feitas as médias do número de acertos e do tempo de execução.

3.2.2.6 Tempos de decisão e execução

A inserção de um decisor, principalmente baseado em redes neurais, ocasionará impactos no tempo de execução.

Dessa forma, para melhor entender as implicações do método proposto, são realizados testes sobre o tempo de decisão da rede, além dos tempos de *get* e *set* do *cache* e, por último, o tempo necessário para computar a média dos valores no *cache* de acordo com sua capacidade.

Para realização desses cálculos, cada uma dessas partes é testada de forma separada.

Para os tempos de *get* e *set*, é criado um *cache* contendo o mesmo tipo de conteúdo que na aplicação real e calculados os tempos médios de dez mil operações. Além disso, é ajustada uma reta, a partir das medições, para verificar a relação linear entre os tempos e a capacidade.

Em vista das operações envolvidas no processo de decisão, são feitas medições separadas de cada uma das sub-etapas e calculadas as médias de cada tempo, desconsiderando as primeiras iterações, impactadas por inicializações e outros fatores.

Por fim, de modo a entender as condições nas quais a aplicação do decisor é benéfica, é possível aproximar o tempo de computação das tarefas nos dois cenários, sem decisor, Equação 3.1, e com decisor, Equação 3.2.

$$T_{total} = (In - H)t \tag{3.1}$$

$$T_{total} = (In - H.g)t + t_d I \tag{3.2}$$

Onde T_{total} representa o tempo total das computações na aplicação, t o tempo de execução de cada sub-tarefa, I o número total de tarefas, n o número de sub-tarefas geradas por tarefa, g é o ganho percentual esperado pela inclusão do decisor e t_d o tempo de decisão.

Igualando os tempos conclui-se que, para que o decisor gere ganhos, o custo mínimo de cada sub-tarefa é aproximado por:

$$t = \frac{t_d I}{(g-1)H} \tag{3.3}$$

Capítulo 4

Resultados e discussões

Neste capítulo, são apresentados os resultados obtidos rumo às contribuições e objetivos propostos. Em primeiro lugar, evidencia-se o melhor desempenho das redes neurais para a tarefa de identificar similaridades em conjuntos de dados multidimensionais. A partir daí, um conjunto de experimentos são propostos para observar: (*i*) o comportamento das políticas sob um modelo *baseline* e; (*ii*) quais os impactos de um decisor em um modelo de cache híbrido (*i.e estático/dinâmico*) sob o ponto de vista de tratamento de *dumps*.

Conforme será exposto nas seções seguintes, o modelo proposto baseado em uma rede neural siamesa apresenta ganhos expressivos se comparado aos testes de base, reiterando a importância da aplicação de técnicas inteligentes para o gerenciamento de recursos escassos.

4.1 Seleção das políticas de gerenciamento

Para compor os experimentos de avaliação das abordagens inteligentes, são selecionadas, dentre as políticas de manutenção de *cache*, duas amplamente adotadas em literatura: *FIFO* e *LRU*.

A FIFO, por considerar um processo natural de descarte de computações a partir das mais antigas produzidas, e a LRU, por ter bom desempenho e popularidade em diversos sistemas Berger et al. (2014) e ZHANG et al. (2013). A partir dessas, é introduzido um conceito híbrido, chaveando o *cache* entre estático e dinâmico através de um decisor.

4.2 Seleção do modelo preditivo

Com as políticas de *cache* definidas, busca-se escolher a estratégia inteligente a ser utilizada. Nesse processo, diferentes conjuntos de dados são selecionados, observando os diferentes graus de agrupamento e impacto das estratégias inteligentes sob tais características.

4.2.1 Conjuntos de Dados

Para compor os experimentos sob uma ótica o mais generalista possível, são selecionados e testados 3 conjuntos de dados com tamanhos diferentes. Assim, é possível observar o comportamento do modelo inteligente em relação não só ao número de amostras, mas seu comportamento sob entradas de diferentes dimensionalidades.

Assim, discutiremos na sequência esses conjuntos e detalhamos em cada sub-seção os diferentes comportamentos observados.

4.2.1.1 Conjunto de Dados *Star Type Dataset*

O conjunto *Star Type Dataset* contêm informações sobre classificação de estrelas, com seis variáveis para compor características e uma como alvo, a qual é representada sob um range de 6 classes. Nesta base, são observadas 240 amostras, modeladas conforme exemplo da Figura 4.1. O conjunto completo pode ser encontrado em https://www.kaggle.com/deepu1109/star-dataset

	Temperature (K)	Luminosity(L/Lo)	Radius(R/Ro)	Absolute magnitude(Mv)	Star type	Star color	Spectral Class
0	3068	0.002400	0.1700	16.12	0	Red	M
1	3042	0.000500	0.1542	16.60	0	Red	M
2	2600	0.000300	0.1020	18.70	0	Red	M
3	2800	0.000200	0.1600	16.65	0	Red	M
4	1939	0.000138	0.1030	20.06	0	Red	M

Figura 4.1 – Amostra do *Star type dataset*
Fonte: O autor.

Esse conjunto foi selecionado devido à baixa dimensionalidade e por conter grupamentos bem definidos espacialmente para cada uma das classes de estrelas. A análise bi-variada da Figura 4.2 detalha em alto nível essas observações.

Como pode-se perceber, são poucas as variáveis de entrada e é fácil identificar os grupos. Valeu lembrar que a diagonal representa as distribuições das variáveis.

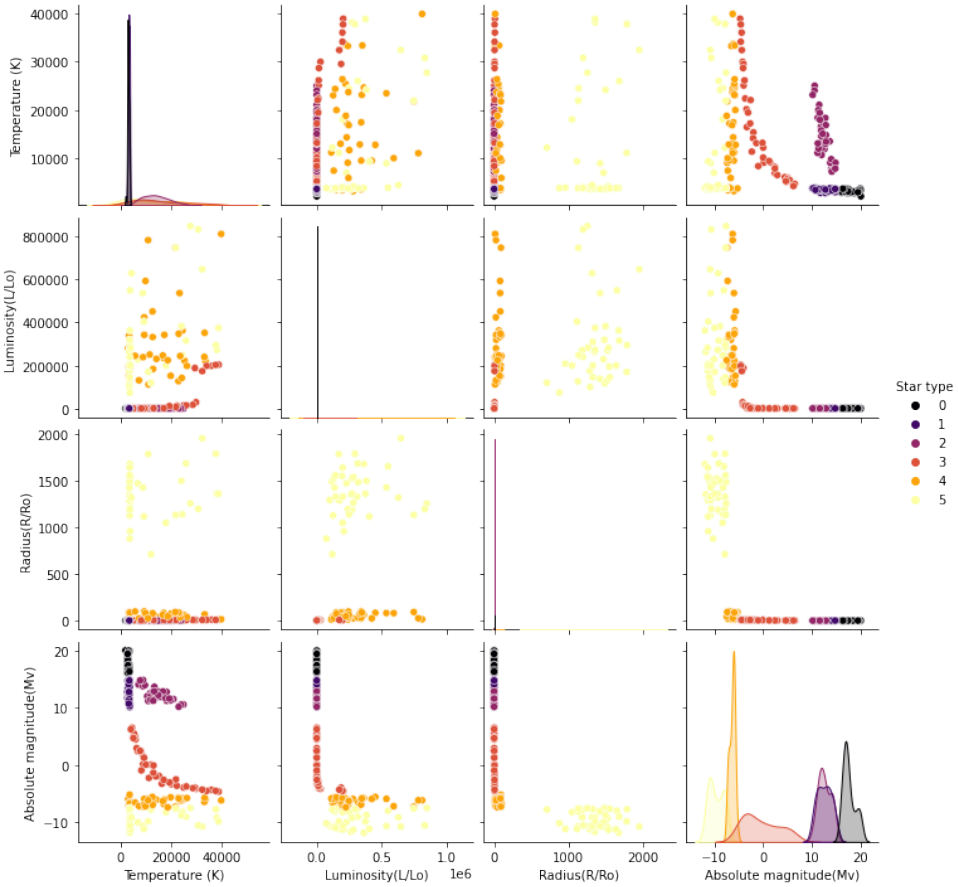

Figura 4.2 – Análise bi-variada do *Star Type Dataset*
Fonte: O autor.

4.2.1.2 Conjunto de Dados *Wine Quality Dataset*

O *Wine Quality Dataset* é um conjunto comum no meio da ciência de dados. Esse contem informações a respeito da qualidade de vinhos tintos, detalhando onze variáveis preditoras e/ou atributos e uma alvo, a qual define a qualidade do vinho e é separada em 7 classes. Nesta base há 4898 amostras sob uma dimensão de tamanho 12. O conjunto completo pode ser encontrado em https://archive.ics.uci.edu/ml/datasets/wine+quality

Este conjunto é de maior complexidade, visto que não há separações claras das classes para nenhuma combinação de duas variáveis.

Além disso, há aumento de dimensionalidade em relação ao primeiro, porém ainda é um número relativamente baixo. Porém, as classes-resposta não estão espacialmente bem definidas para nenhuma das variáveis, o que será um maior desafio para o modelo de grupamento.

	fixed acidity	volatile acidity	citric acid	residual sugar	chlorides	free sulfur dioxide	total sulfur dioxide	density	pH	sulphates	alcohol	quality
0	7.0	0.27	0.36	20.7	0.045	45.0	170.0	1.00100	3.00	0.45	8.8	6
1	6.3	0.30	0.34	1.6	0.049	14.0	132.0	0.99400	3.30	0.49	9.5	6
2	8.1	0.28	0.40	6.9	0.050	30.0	97.0	0.99510	3.26	0.44	10.1	6
3	7.2	0.23	0.32	8.5	0.058	47.0	186.0	0.99560	3.19	0.40	9.9	6
4	7.2	0.23	0.32	8.5	0.058	47.0	186.0	0.99560	3.19	0.40	9.9	6
...	...	...	...	...	...	...	...	...	...	...	...	...
4893	6.2	0.21	0.29	1.6	0.039	24.0	92.0	0.99114	3.27	0.50	11.2	6
4894	6.6	0.32	0.36	8.0	0.047	57.0	168.0	0.99490	3.15	0.46	9.6	5
4895	6.5	0.24	0.19	1.2	0.041	30.0	111.0	0.99254	2.99	0.46	9.4	6
4896	5.5	0.29	0.30	1.1	0.022	20.0	110.0	0.98869	3.34	0.38	12.8	7
4897	6.0	0.21	0.38	0.8	0.020	22.0	98.0	0.98941	3.26	0.32	11.8	6

Figura 4.3 – Amostra do *Wine quality dataset*
Fonte: O autor.

4.2.1.3 Conjunto de Dados *Power Quality Dataset*

O conjunto *Power Quality Dataset* é composto por 256 atributos que representam sinais relativos à qualidade da energia em uma rede elétrica. Nesse *dataset*, as classes são divididas em 6 categorias, havendo 6000 amostras disponíveis. O conjunto completo pode ser encontrado em https://www.kaggle.com/aswarthnarayanacv/power-quality-distribution-da

	0	1	2	3	4	5	6	7	8	9	...	247	
0	4885.767410	5266.642327	5383.134015	5769.524167	5930.508307	5953.194350	6169.537273	6010.471326	6131.493474	5857.529713	...	199.147167	8
1	573.652486	1003.343736	1588.404525	2317.576741	2804.364311	3225.322510	3662.821690	4174.627969	4656.244143	4939.070130	...	-4228.581226	-38
2	4757.365183	5264.598912	5428.642486	5650.413073	5939.710012	5911.948067	6147.642171	6076.921501	5958.797444	6053.817701	...	323.325836	8
3	4242.144824	4644.679402	5013.356532	5229.417051	5534.898007	5797.190678	5930.658682	5960.014599	6055.336310	6103.707793	...	-616.527428	-
4	2077.819247	2561.679246	3085.653813	3545.905160	4023.421592	4496.705157	4809.079868	5186.298840	5453.627533	5737.354699	...	-3017.951179	-25
...	...	...	...	...	...	...	...	...	...	...	...	...	
5995	4182.889029	4554.992727	4634.917713	5174.426105	4943.515768	5602.432539	5082.817543	5770.004732	5041.487028	5689.180989	...	-335.393727	8
5996	1075.044771	1333.557616	2035.652127	2407.618003	2836.652597	3421.979877	3579.298249	4390.184428	4137.708705	5037.241979	...	-3702.246047	-27
5997	1485.947799	2041.537942	2367.324807	3084.757587	3163.962005	4007.714969	3856.445609	4733.341684	4447.140464	5233.763694	...	-3344.522688	-23
5998	4025.194751	4447.713315	4611.030182	4973.135519	4963.756898	5468.895454	5219.171752	5658.045982	5347.294691	5640.257744	...	-520.593818	3
5999	2560.181511	2821.355683	3345.099297	3772.180014	4001.695102	4561.185945	4523.807031	5165.113773	4882.463857	5551.112260	...	-2408.852701	-14

Figura 4.4 – Amostra do *Power Quality Dataset*
Fonte: O autor.

Esse *dataset* apresenta um número elevado de variáveis preditoras e não há separação clara das classes, tendo maior complexidade que os demais. Entretanto, isso não quer dizer necessariamente que seja o mais difícil em termos de classificação, já que isso depende do poder preditor das variáveis em relação às classes.

4.3 Métodos Inteligentes para agrupamento

Definida as bases a serem observadas, tem-se como próximo passo a busca por estratégias inteligentes que consigam, com certa eficiência, agrupar esses dados levando em conta as diferentes caracterizações apresentadas. Nessa busca são avaliados dois dos métodos bastante citados em literatura, (i) o *K-Means* e (ii) as *Redes Neurais*.

4.3.1 *K-means*

A implementação do *K-means* é feita utilizando a biblioteca *Scikit Learn*, que segue a teoria apresentada na seção 2.2.3.1.1.

4.3.1.1 *K-means*: Análise do Conjunto *Star Type Dataset*

O primeiro passo é a aplicação da ”regra do cotovelo” para determinar o melhor número de *clusters* para o algoritmo, o resultado pode ser visto na Figura 4.5. Embora seja conhecido o correto número de grupamentos pelas categorias existentes no conjunto, a análise da curva mostra que o melhor valor para o número de grupamentos é 4.

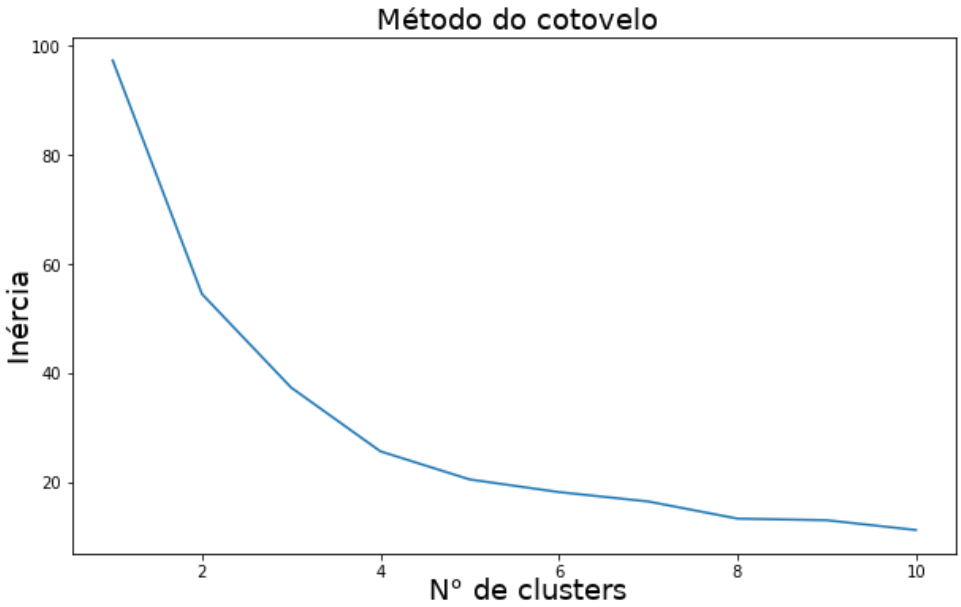

Figura 4.5 – *Score* de inércia por número de *Clusters* para o *Star type dataset*
Fonte: O autor.

Retreinando o modelo para n = 4 são obtidas a matriz de migração da Figura 4.6 para os dados de teste. Nela, o valor dentro de cada célula representa a quantidade percentual de amostras classificadas na categoria predita pela linha. Ou seja, todas as amostras de estrelas do tipo 0 foi classificada pelo modelo na categoria 2. A soma dos valores de cada coluna será sempre igual a 100%.

Como podemos perceber, o algoritmo consegue agrupar as categorias de maneira eficiente, criando um *clusters* para as categorias (0,1), (2,3), (4) e (5), que são de fato próximas no espaço. Foram escolhidas duas variáveis onde a separação das categorias se torna mais evidente a título de comparação, mostrada na Figura 4.7

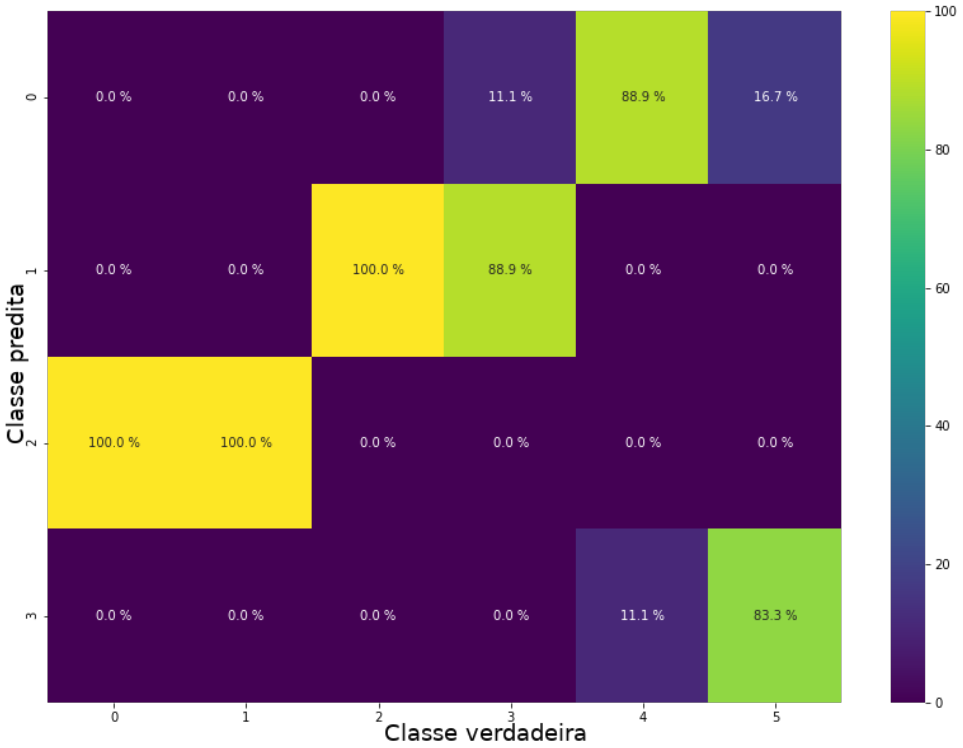

Figura 4.6 – Matriz de migração (teste) para o *Star type dataset*
Fonte: O autor.

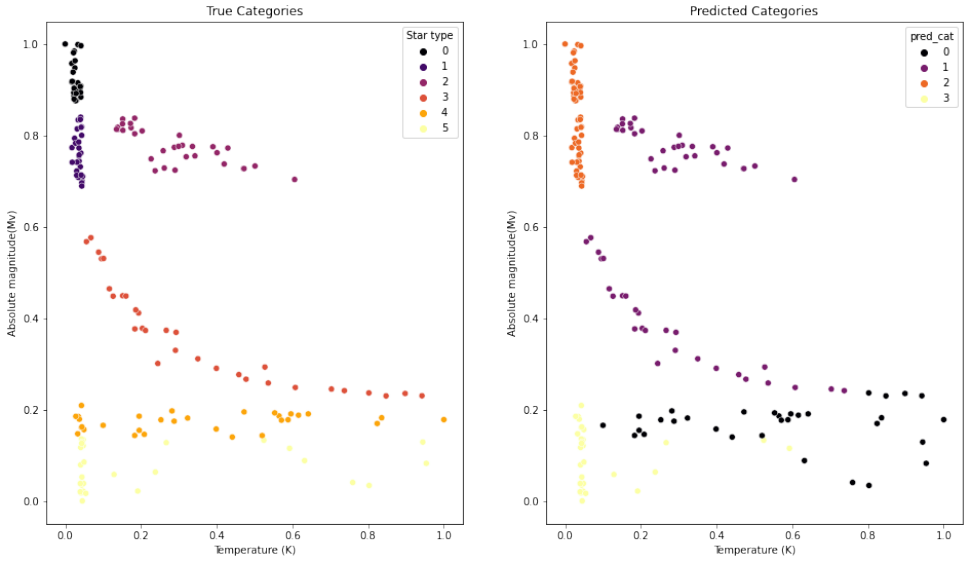

Figura 4.7 – Comparação entre os clusters verdadeiros e os preditos no *Star type dataset*
Fonte: O autor.

4.3.1.2 *K-means*: Análise do Conjunto *Wine quality Dataset*

Novamente é aplicada a ”regra do cotovelo”e o modelo é treinado 10 vezes com diferentes números de *clusters*, como mostrado na Figura 4.8.

Desta vez, o número escolhido é n = 6, sendo o modelo retreinado com ele. Os resulta-

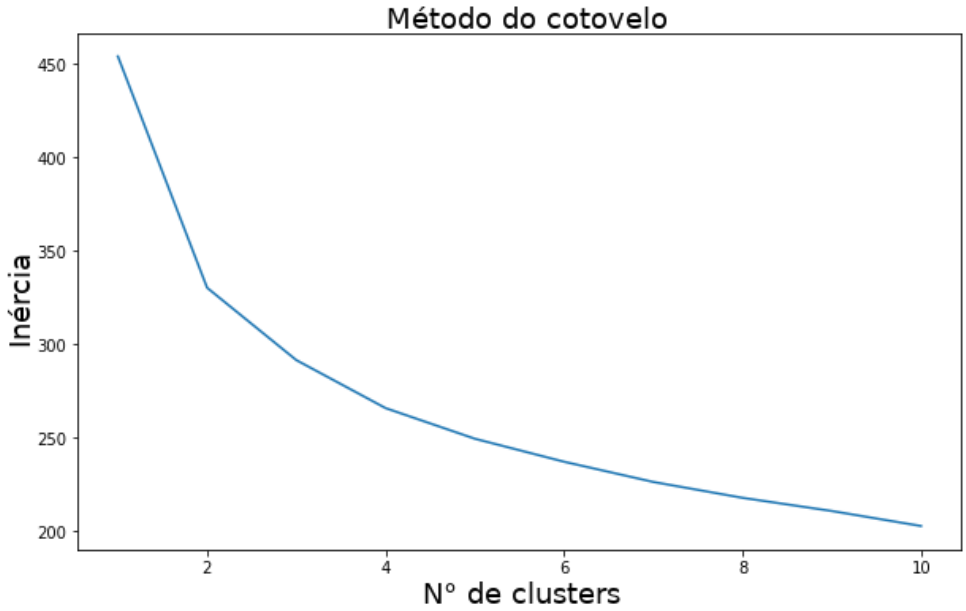

Figura 4.8 – *Score* de inércia por número de *Clusters* para o *Wine quality dataset*
Fonte: O autor.

dos para os dados de teste podem ser vistos na Figura 4.9. Como para o conjunto anterior, os valores das células representam o percentual da categoria real (colunas) classificados em cada classe prevista (linhas).

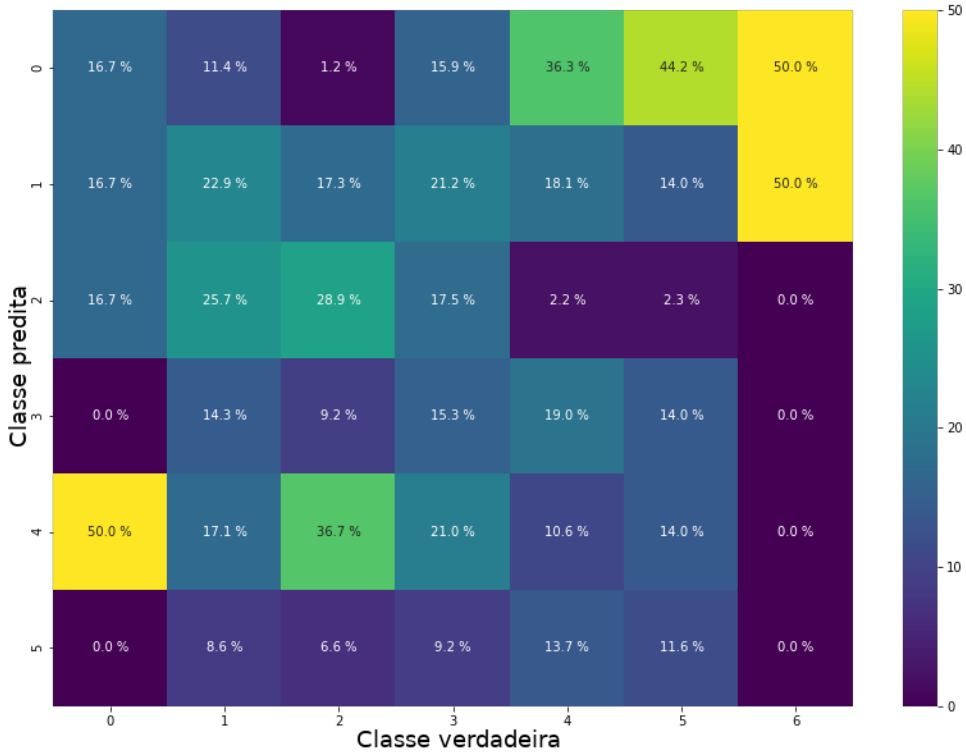

Figura 4.9 – Matriz de migração (teste) para o *Wine quality dataset*
Fonte: O autor.

Observa-se, analisando cada coluna da matriz de migração, que o algoritmo não foi capaz de corretamente categorizar as amostras de treino, havendo amostras de vinhos com mesma qualidade em diversas categorias previstas diferentes, não há concentração de valores em nenhuma categoria prevista.

4.3.1.3 *K-means*: Análise do Conjunto *Power quality Dataset*

Assim como nos demais conjuntos, em primeiro lugar é aplicada a ”regra do cotovelo”, como visto na Figura 4.10.

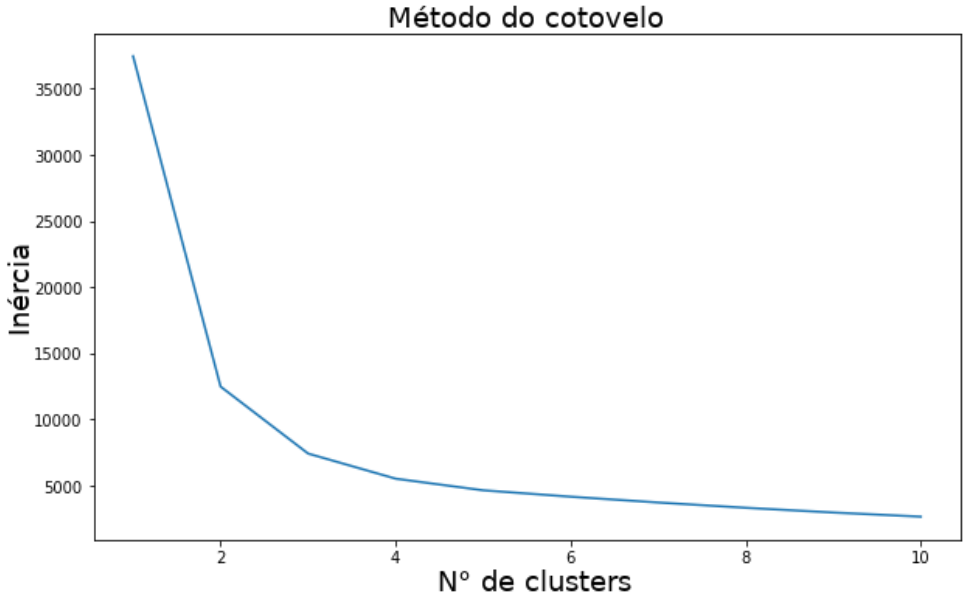

Figura 4.10 – *Score* de inércia por número de *Clusters* para o *Power quality dataset*
Fonte: O autor.

A partir daí, escolhe-se o valor n = 4 para o treino do modelo final, cuja matriz de migração para os dados de teste pode ser vista na Figura 4.11.

Aqui nota-se que as amostras foram espalhadas dentro de todas as categorias previstas, não havendo separação eficiente destas. Dessa forma, o método *K-Means* se mostra, de novo, insuficiente para a classificação correta dos dados.

4.3.1.4 *K-means*: Discussões

Ao observar os três resultados, fica claro que o método *K-means* apresenta limitações quando o conjunto tratado apresenta elevada dimensionalidade e maior homogeneidade na distribuição espacial da variável alvo. Apesar de ter conseguido ordenar com certo grau de eficiência os tipos estelares no primeiro conjunto de dados, quando apresentado a conjuntos

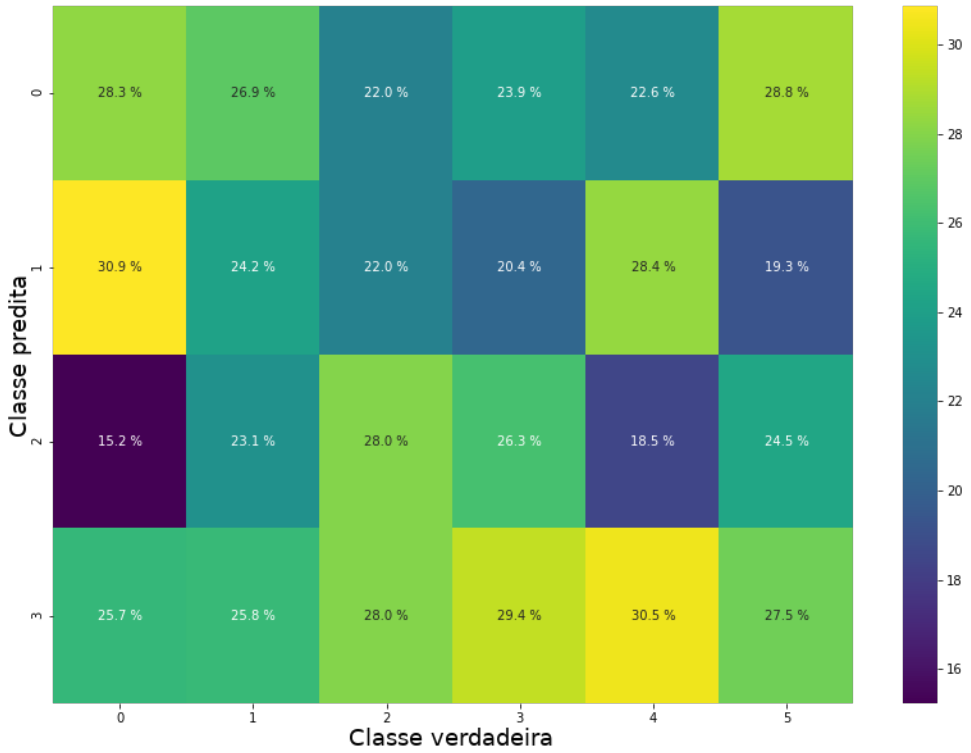

Figura 4.11 – Matriz de migração (teste) para o *Power quality dataset*
Fonte: O autor.

mais complexos, sua resposta não foi satisfatória, não havendo correta ordenação em nenhum grau. Dessa forma, não o método não é adequado para a continuidade deste trabalho.

4.3.2 Rede neural siamesa

A implementação da rede neural siamesa foi feita com base na fundamentação da seção 2.2.4.1.1 e utilizando o *framework Keras*. O *link* para o repositório de códigos completo no *GitHub* podem ser encontrados no Apêndice A.

4.3.2.1 Rede neural: Análise do Conjunto *Star Type Dataset*

Para aplicação do conjunto na rede neural, são montados os pares (amostra,média), com o rótulo *'Similarity'* verdadeiro quando a amostra é da mesma classe da média. Uma amostra do *dataset* de treino pode ser vista na Figura 4.12. Vale ressaltar que os valores das médias utilizadas foram os de treino, garantindo que não houvesse *data leakege*.

A partir daí, é feito o ajuste paramétrico manualmente, obtendo-se seguintes parâmetros:

	Temperature (K)	Luminosity(L/Lo)	Radius(R/Ro)	Absolute magnitude(Mv)	Star type	Similarity
0	0.244634	7.911280e-04	0.003578	0.300813	3.0	0
1	0.027931	6.189714e-10	0.000052	0.922024	0.0	0
2	0.244634	7.911280e-04	0.003578	0.300813	3.0	0
3	0.035881	5.457679e-09	0.000187	0.761289	1.0	0
4	0.244634	7.911280e-04	0.003578	0.300813	3.0	0
5	0.288702	2.789759e-09	0.000001	0.775848	2.0	0
6	0.244634	7.911280e-04	0.003578	0.300813	3.0	1
7	0.365443	3.318370e-02	0.002291	0.362881	3.0	1
8	0.244634	7.911280e-04	0.003578	0.300813	3.0	0
9	0.353814	3.457293e-01	0.027064	0.173326	4.0	0
10	0.244634	7.911280e-04	0.003578	0.300813	3.0	0
11	0.242801	3.528931e-01	0.695074	0.073461	5.0	0

Figura 4.12 – Primeira amostra do *Star type dataset* de treino.
Fonte: O autor.

- Entradas: [4,4]
- Camadas escondidas: [[512,256,128],[512,256,128]]
- Camada de junção: Distância absoluta(Tensor1,Tensor2)
- Saídas: [2] (classes 0 e 1)
- Número de épocas: 200
- Taxa de aprendizado: 1e-4
- Peso das classes: 0:1, 1:6
- *Dropout*: 5%

Como há uma relação direta entre as classes preditas e as categorias reais, utiliza-se a matriz de confusão, vista na Figura 4.13. Assim como antes, os valores em cada células representam a distribuição percentual de cada classe verdadeira dentre as previstas. Neste caso, quando mais próxima a uma matriz diagonal, mais preciso é o resultado.

Como observado, a rede é capaz de classificar os tipos estelares com precisão perfeita, separando até mesmo as classes espacialmente indistintas, o que fica ainda mais claro pelas as métricas da Figura 4.14.

Por fim, a Figura 4.15 traz uma análise bi-variada onde a separação das classes é mais clara, de modo a comparar os resultados da rede com as classes originais nos dados de teste.

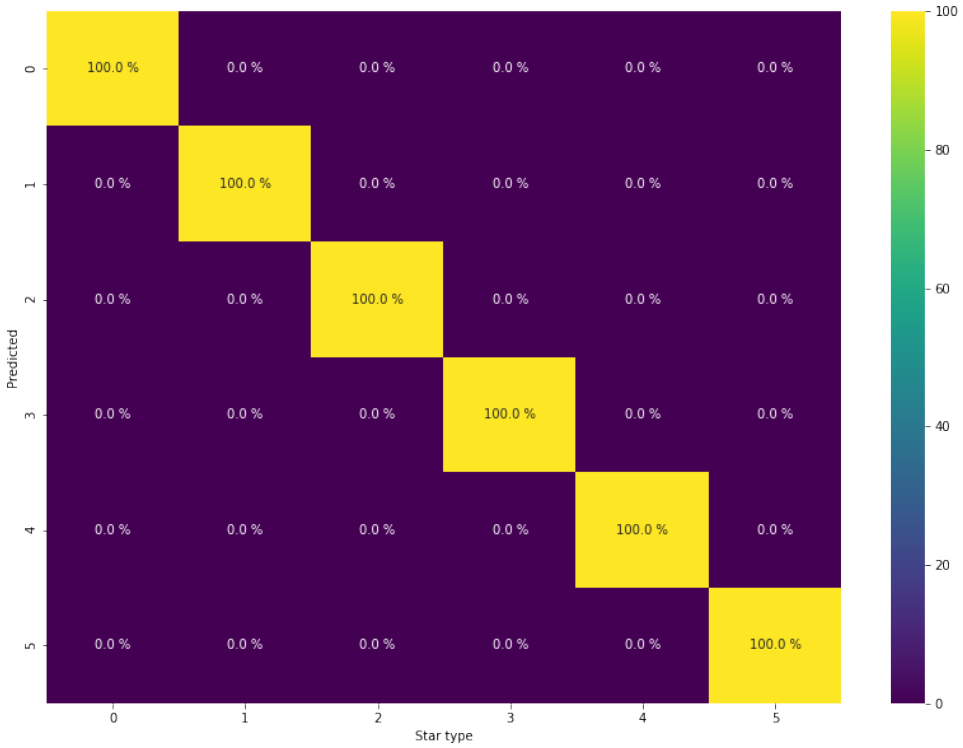

Figura 4.13 – Matriz de migração (teste) para o *Star type dataset*
Fonte: O autor.

	precision	recall	f1-score	support
0.0	1.00	1.00	1.00	10
1.0	1.00	1.00	1.00	11
2.0	1.00	1.00	1.00	9
3.0	1.00	1.00	1.00	9
4.0	1.00	1.00	1.00	9
5.0	1.00	1.00	1.00	12
accuracy			1.00	60
macro avg	1.00	1.00	1.00	60
weighted avg	1.00	1.00	1.00	60

Figura 4.14 – Métricas de teste para o *Star type dataset*
Fonte: O autor.

Durante o teste, o modelo foi capaz de predizer corretamente todas as classes das amostras, mostrando-se bastante superior ao *K-Means*

4.3.2.2 Rede neural: Análise do conjunto *wine quality Dataset*

De forma análoga ao feito no primeiro conjunto, são montados os pares (amostra, média), com o rótulo *'Similarity'* na *target*.

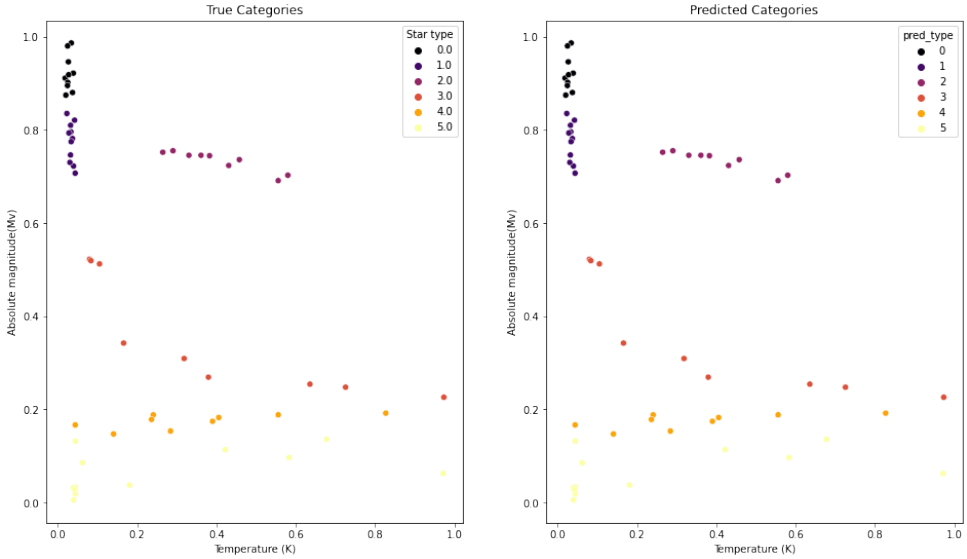

Figura 4.15 – Comparação das classes reais e preditas (teste) para o *Star type dataset*
Fonte: O autor.

A partir daí, é feito o ajuste paramétrico manualmente, obtendo-se os seguintes parâmetros:

- Entradas: [11,11]
- Camadas escondidas: [[1024,512,256],[1024,512,256]]
- Camada de junção: Distância absoluta(Tensor1,Tensor2)
- Número de épocas: 35
- Taxa de aprendizado: 1e-4
- Peso das classes: 0:1, 1:7
- *Dropout*: 20%

A matriz de confusão para os dados de teste pode ser vista na Figura 4.16. Vale ressaltar que, há apenas 6 classes, ao invés das 7 do conjunto, pois a classe de número 6 possui um número muito baixo de amostras, apenas 5 no total, não havendo representação dela nos dados de teste.

Nota-se que a rede foi capaz de classificar de maneira mais eficiente dos vinhos se comparada ao *K-means*, havendo a diagonalização esperada na matriz, o que mostra que parte das predições foi correta. Além disso, as métricas da Figura 4.17 mostram uma taxa de acerto próxima a 60%.

in

Figura 4.16 – Matriz de migração (teste) para o *Wine quality dataset*
Fonte: O autor.

	precision	recall	f1-score	support
0.0	1.00	0.14	0.25	7
1.0	0.67	0.12	0.21	32
2.0	0.63	0.66	0.64	358
3.0	0.58	0.72	0.64	544
4.0	0.61	0.39	0.48	241
5.0	0.00	0.00	0.00	43
accuracy			0.60	1225
macro avg	0.58	0.34	0.37	1225
weighted avg	0.58	0.60	0.57	1225

Figura 4.17 – Métricas de teste para o *Wine quality dataset*
Fonte: O autor.

É provável que com maior refinamento dos hiper-parâmetros um resultado ainda melhor possa ser alcançado. Entretanto, o intuito aqui é, inicialmente, mostrar a diferença performática entre os modelos, a fim de selecionar o mais adequado. Dessa forma, não é necessária a melhor resposta possível, apenas uma que seja superior ao modelo anterior para a prova de conceito.

4.3.2.3 Rede neural: Análise do conjunto *power quality Dataset*

Para o terceiro conjunto novamente são montados os pares (amostra, média), com o rótulo *'Similarity'* par a variável resposta e o ajuste paramétrico e feito manualmente. Os parâmetros obtidos são:

- Entradas: [256,256]
- Camadas escondidas: [[1024,512,256],[1024,512,256]]
- Camada de junção: Distância absoluta(Tensor1,Tensor2)
- Número de épocas: 35
- Taxa de aprendizado: 1e-4
- Peso das classes: 0:1, 1:7
- *Dropout*: 20%

A matriz de migração para os dados de teste pode ser vista na Figura 4.18.

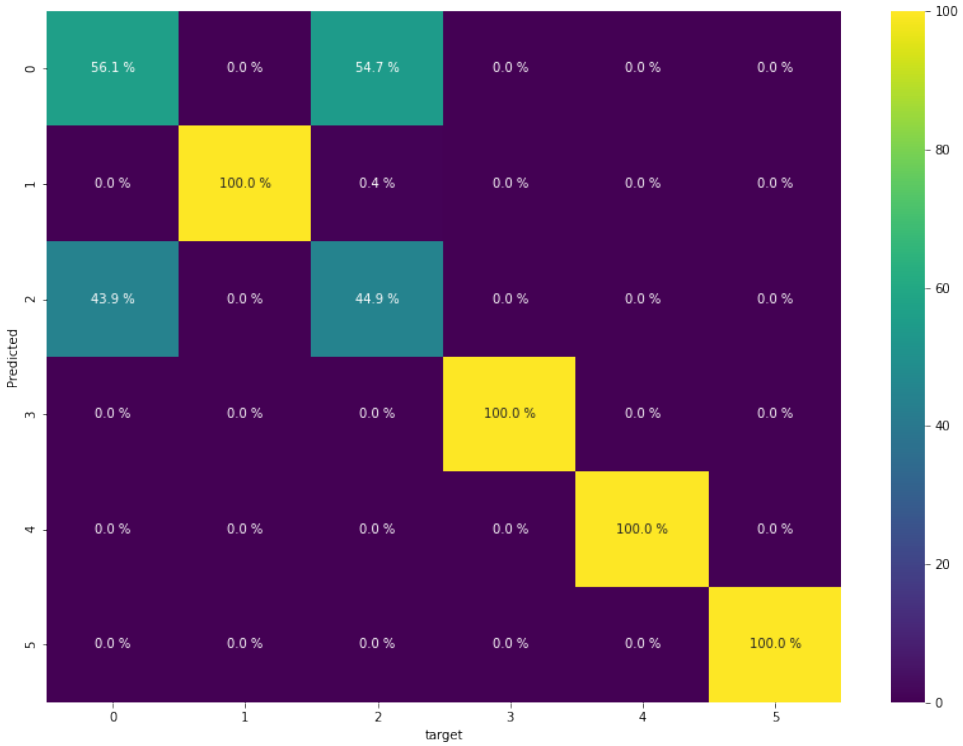

Figura 4.18 – Matriz de migração (teste) para o *Power quality dataset*
Fonte: O autor.

Percebe-se que a rede é capaz de classificar a qualidade da rede elétrica de maneira bastante eficiente. Apesar de algumas amostras classificadas erroneamente entre as classes 0 e 2, há a diagonalização esperada na matriz. Além disso, as métricas da Figura 4.19 mostram uma taxa de acerto de 83%

	precision	recall	f1-score	support
0.0	0.51	0.44	0.48	269
1.0	1.00	1.00	1.00	264
2.0	0.45	0.52	0.48	236
3.0	1.00	1.00	1.00	255
4.0	1.00	1.00	1.00	243
5.0	1.00	1.00	1.00	233
accuracy			0.82	1500
macro avg	0.83	0.83	0.83	1500
weighted avg	0.83	0.82	0.82	1500

Figura 4.19 – Métricas de teste para o *Power quality dataset*
Fonte: O autor.

4.3.3 Rede neural: discussões

Ao observar os testes nos três conjuntos, fica claro o poder da rede proposta em categorizar uma amostra com base na semelhança da mesma em relação a uma média da classe. No conjunto de tipos estelares, ela conseguiu prever as classes com precisão perfeita e nos conjuntos de maior complexidade, também teve resultados melhores que os alcançados pelo *K-means*, apresentando maiores taxas de acerto.

4.3.4 Discussões: Modelos preditivos

A partir dos resultados dos experimentos, conclui-se que o algoritmo *K-Means* não lida bem com dados com elevada dimensionalidade e homogeneidade das classes no espaço, sendo eficiente apenas para conjuntos que possuem distinções claras das localizações categorias da variável alvo, ou seja, quando há *clusters* espacialmente bem definidos. Assim, esse método não é capaz de fornecer o poder preditor necessário para a aplicação deste trabalho e será descontinuado.

Além disso, constatou-se que o modelo de rede neural siamesa é capaz de extrair informações relevantes a partir de uma variedade de contextos, com diferentes quantidades

amostrais e número de variáveis, o que mostra seu poder de generalização. Dessa maneira, este será o modelo escolhido para dar continuidade ao trabalho proposto.

4.4 Aplicação completa

A seguir, são dispostos e analisados os resultados relativos à aplicação final, desde os testes de base até a inserção do modelo decisor.

4.4.1 Bases de dados

Em primeiro momento, são apresentados os dados utilizados como entrada na aplicação proposta, bem como os utilizados para treinamento da rede neural.

4.4.1.1 Dados de entrada da aplicação

Os dados utilizados na aplicação foram retirados do conjunto *California Housing Prices*, disponível na plataforma *Kaggle*. Dessa forma, a base possui 20 mil amostras compostas por 8 variáveis preditoras e uma variável resposta, como ilustra a Figura 4.20.

	var_0	var_1	var_2	var_3	var_4	var_5	var_6	var_7	median_house_value
0	-122.05	37.37	27.0	3885.0	661.0	1537.0	606.0	6.6085	344700.0
1	-118.30	34.26	43.0	1510.0	310.0	809.0	277.0	3.5990	176500.0
2	-117.81	33.78	27.0	3589.0	507.0	1484.0	495.0	5.7934	270500.0
3	-118.36	33.82	28.0	67.0	15.0	49.0	11.0	6.1359	330000.0
4	-119.67	36.33	19.0	1241.0	244.0	850.0	237.0	2.9375	81700.0
5	-119.56	36.51	37.0	1018.0	213.0	663.0	204.0	1.6635	67000.0

Figura 4.20 – Amostra das entradas da aplicação antes do tratamento
Fonte: O autor.

Em vista de diminuir a faixa de valores encontrados nas colunas para aplicação LAC, são realizados três tratamentos. O primeiro consiste em dividir os valores das colunas "*var_3*" a "*var_7*" por 1000. A seguir, todas as colunas têm seus valores arredondados para uma casa decimal. Por último, a variável resposta, que traz o preço médio da casa, é quebrada em 5 quantis e transformada em categorias, em consideração ao fato da aplicação ser um classificador.

Dessa forma, as chances de encontrar o mesmo conjunto de valores para as variáveis preditoras no treino e no teste aumenta, adequando os dados ao algoritmo escolhido. A Figura 4.21 traz as mesmas amostras anteriores após o tratamento.

	var_0	var_1	var_2	var_3	var_4	var_5	var_6	var_7	house_class
0	-122.0	37.4	27.0	3.9	0.7	1.5	0.6	6.6	4
1	-118.3	34.3	43.0	1.5	0.3	0.8	0.3	3.6	2
2	-117.8	33.8	27.0	3.6	0.5	1.5	0.5	5.8	3
3	-118.4	33.8	28.0	0.1	0.0	0.0	0.0	6.1	4
4	-119.7	36.3	19.0	1.2	0.2	0.8	0.2	2.9	0
5	-119.6	36.5	37.0	1.0	0.2	0.7	0.2	1.7	0

Figura 4.21 – Amostra das entradas da aplicação após o tratamento
Fonte: O autor.

4.4.1.2 Dados de treinamento

Em seguida, são realizados os testes para aquisição dos dados de treino. Dentre os possíveis cenários, é utilizado o conjunto com as seguintes características:

- Capacidade de cache: 200
- Política: FIFO
- Pares Tarefa-Média: 3000

A Figura 4.22 traz uma amostra da base utilizada. Nela é possível ver as variáveis preditoras, indicas por "*var_x*", e a variável resposta "*hit_counts*", as demais variáveis não são utilizadas para o treinamento, sendo que "*entrada*" é o índice do par e "*tipo*" indica se é uma tarefa ou uma média.

A capacidade escolhida é de 200 por ser um valor intermediário e eficiente, considerando a troca entre menor consumo de memória e maior número de acertos. Além disso, a política FIFO é utilizada como base por não aplicar nenhuma inteligência na decisão de qual computação manter e, dessa forma, gerar dados de treino menos enviesados.

	entrada	var_0	var_1	var_2	var_3	var_4	var_5	var_6	var_7	tipo	hit_counts
2036	1118	-124.200000	40.800000	52.000000	2.200000	0.500000	0.900000	0.500000	1.600000	0_task	146
2037	1118	-118.109756	33.900000	30.800000	1.741791	0.431068	1.177465	0.400000	2.763636	1_avg	146
1090	645	-121.800000	36.500000	18.000000	3.200000	0.500000	1.300000	0.400000	6.500000	0_task	175
1091	645	-118.557303	34.350667	34.166667	2.017105	0.406122	1.045745	0.386598	4.600000	1_avg	175
1492	846	-122.100000	37.700000	30.000000	5.400000	1.200000	2.700000	1.000000	3.200000	0_task	16
1493	846	-118.870000	34.922222	35.428571	2.589286	0.482000	1.390323	0.441414	3.695000	1_avg	16

Figura 4.22 – Amostra dos dados de treino
Fonte: O autor.

A seguir, observa-se que a quantidade média de acertos cai com o tempo, como mostra na Figura 4.23. Esse resultado já era esperado pela maneira com que os acertos foram atribuídos, de forma retroativa. Assim sendo, computações mais novas têm menos iterações para acumular acertos.

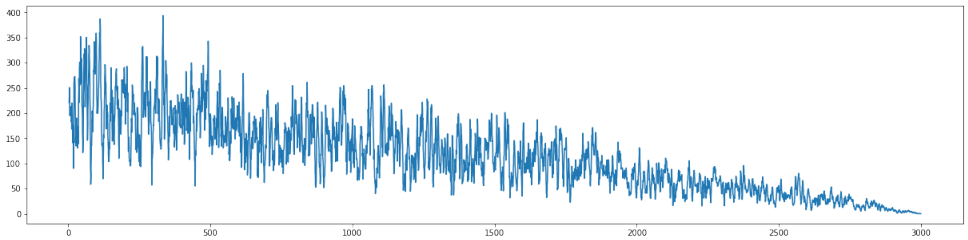

Figura 4.23 – Média móvel de acertos em 10 execuções.
Fonte: O autor.

De modo a diminuir esse efeito, são selecionados para o treinamento apenas amostras que estivessem entre as posições 100 e 1500 de execução. A seguir, é feita a separação aleatória de 80% da base para treino e 20% para teste.

4.4.2 Testes *baseline*

Os primeiros testes realizados com a aplicação são para as políticas escolhidas anteriormente na ausência do decisor, para servirem como base de comparação. Assim, a Figura 4.24 mostra os resultados obtidos.

Percebe-se que a inclusão do *cache* de baixa capacidade aumenta o tempo de execução total da aplicação, uma vez que o baixo número de acertos não é capaz de compensar o tempo gasto com a busca e armazenagem das computações. Porém, para maiores capacidades há uma redução considerável no tempo, chegando a mais de 15% no caso da LRU.

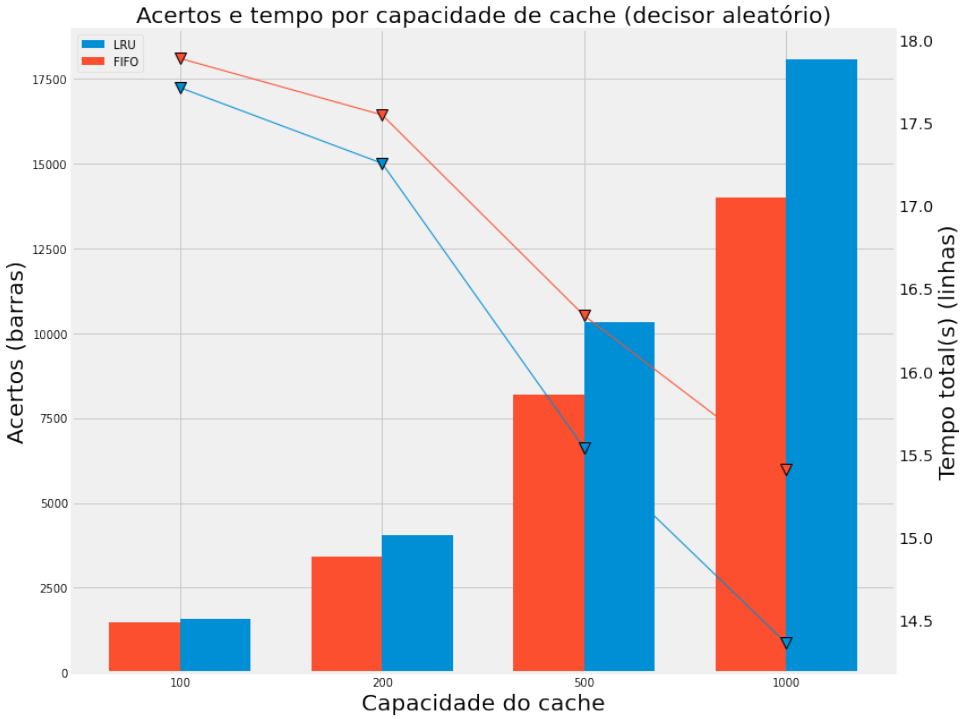

Figura 4.24 – Resultado dos testes sem a presença do decisor (*baseline*)
Fonte: O autor.

Além disso, a Figura 4.25 mostra a quantidade total de *sets*, computações feitas e armazenadas, durante a execução. Conforme o número de *hits* aumenta, nas maiores capacidades, esse valor naturalmente diminui.

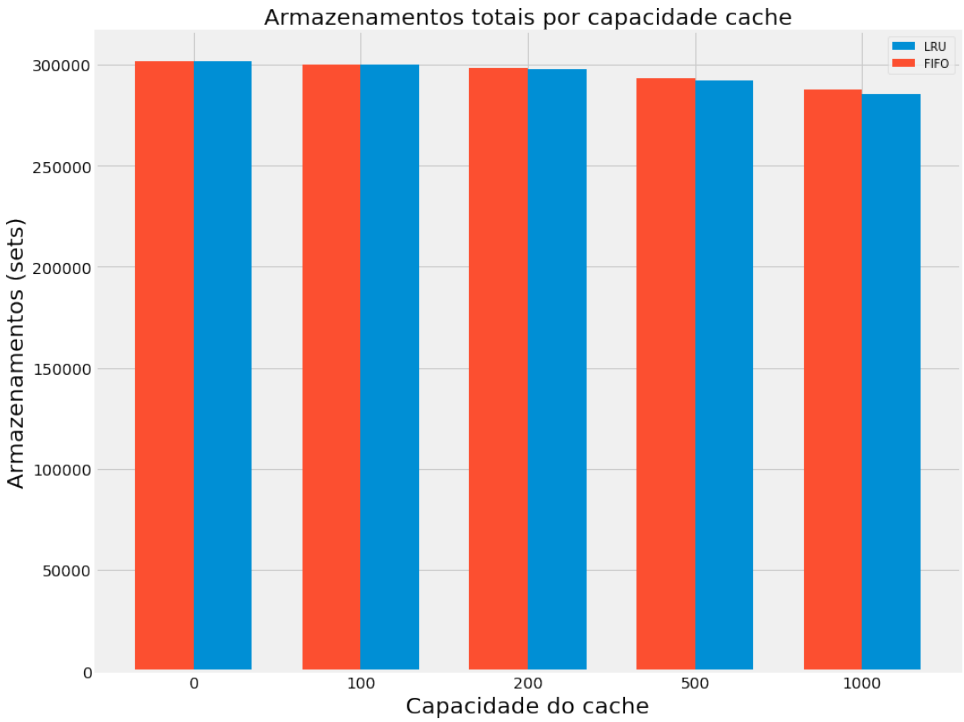

Figura 4.25 – Armazenamentos feitos
Fonte: O autor.

4.4.3 Treinamento da rede neural

A partir da base de dados escolhida, é feito o processo de treinamento e refinamento de hiper-parâmetros, resultando nas seguintes configurações:

- Estrutura: [200 100 50]
- Épocas: 1500
- taxa de aprendizado: 0.001
- *dropout*: 20%

O treinamento do modelo final pode ser visto na Figura 4.26. Nela percebe-se que, apesar do número elevado de épocas, não há grande distanciamento das curvas de treino e teste, o que indica pouco de sobre-ajuste do modelo aos dados de treino.

A correlação das predições com os valores reais no teste foi alta, e pode ser vista junto às outras métricas na Tabela 4.1, indicando que o modelo está aderente aos dados e consegue generalizar o comportamento, como observado na Figura 4.27.

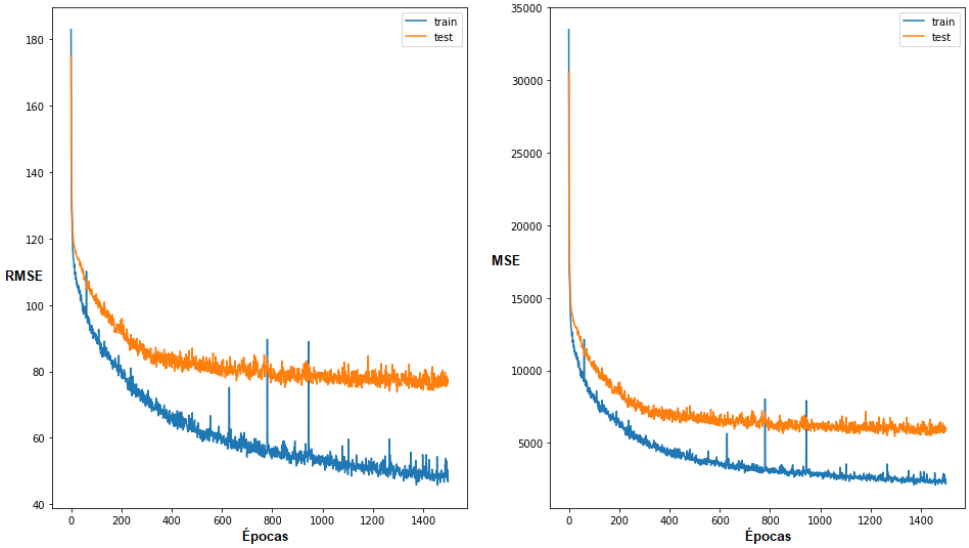

Figura 4.26 – Treinamento da rede neural
Fonte: O autor.

Tabela 4.1 – Métricas do modelo final

	MAE	RMSE	Corr.
Treino	33.23	47.91	0.92
Teste	56.36	77.67	0.77

A diferença de métricas observada é provavelmente devida a presença de mínimos locais na função de perda, entretanto, esse fato não foi prejudicial ao desempenho do modelo, que apresentou as melhores métricas nos dados de teste.

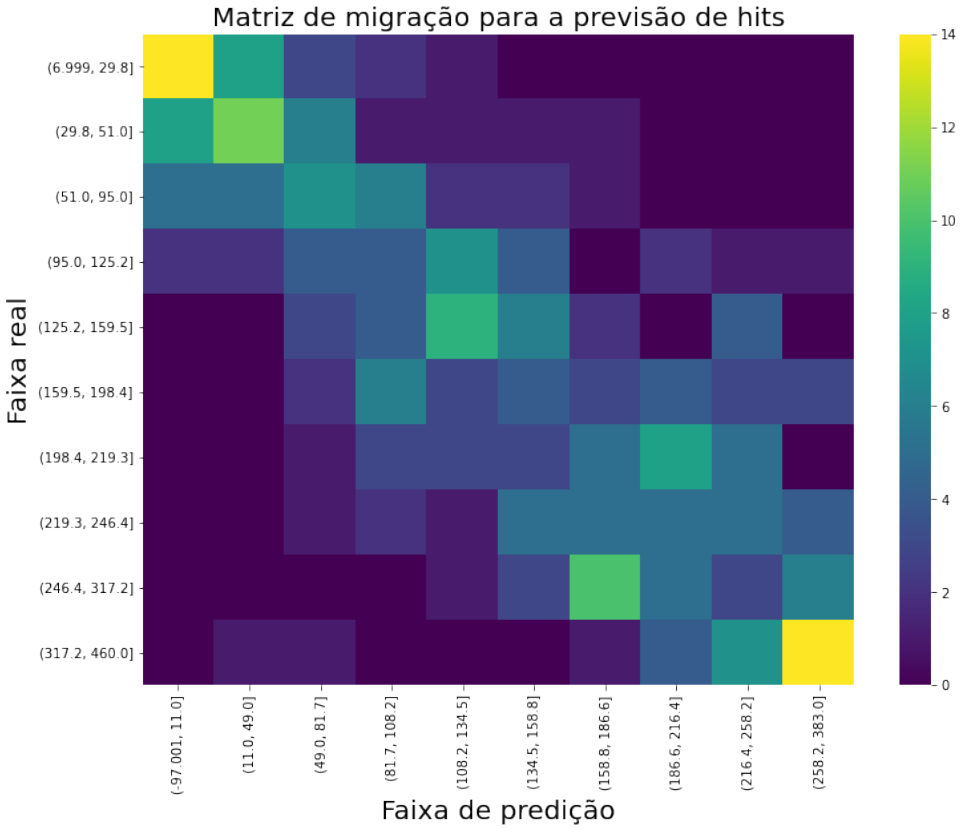

Figura 4.27 – Matriz de migração (teste): valor real x predição da rede
Fonte: O autor.

4.4.4 Resultados: Testes finais

A seção a seguir traz os resultados obtidos pela introdução dos decisores aleatório e baseado na rede neural siamesa na aplicação proposta.

Serão mostrados, para cada política, os limiares de decisão testados, a quantidade de armazenamentos realizados e os ganhos no número de acertos em relação aos testes base.

4.4.4.1 FIFO

Em primeiro lugar, são realizados os testes para determinar o melhor valor de limiar a ser utilizado, conforme mostra a Figura 4.28. Nela é visto o comportamento da média de acertos para cada capacidade e diferentes valores de limiares.

É selecionado, para análise de cada capacidade, o valor que resultou no maior número de acertos.

A seguir, é feita a comparação entre o número de acertos do teste base, do decisor aleatório e do decisor neural para essa política, cujos resultados são mostrados nas Figuras 4.32 e 4.33.

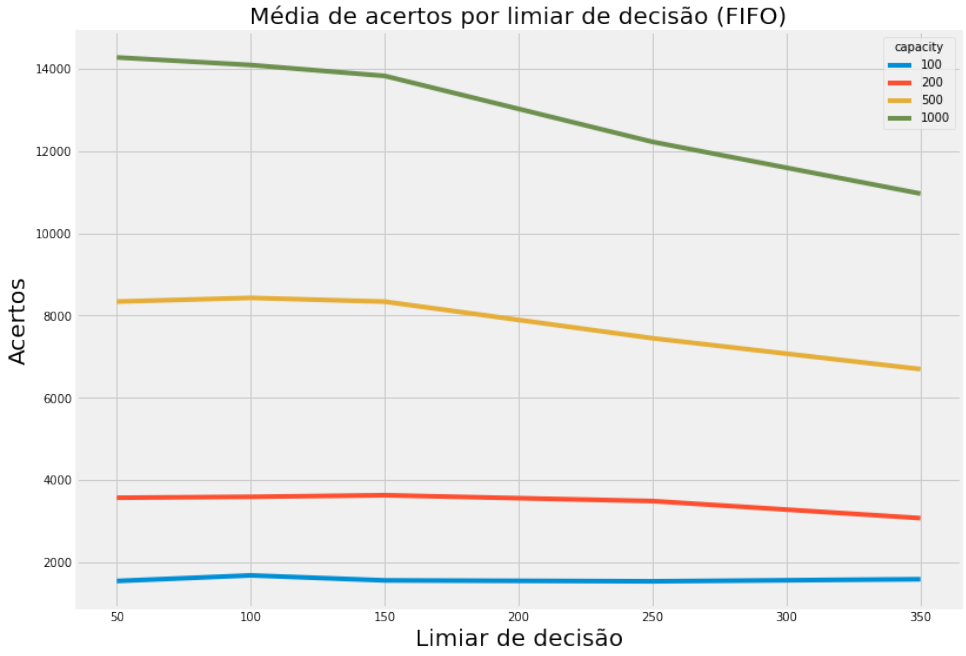

Figura 4.28 – Média de acertos por valor de limiar para a FIFO
Fonte: O autor.

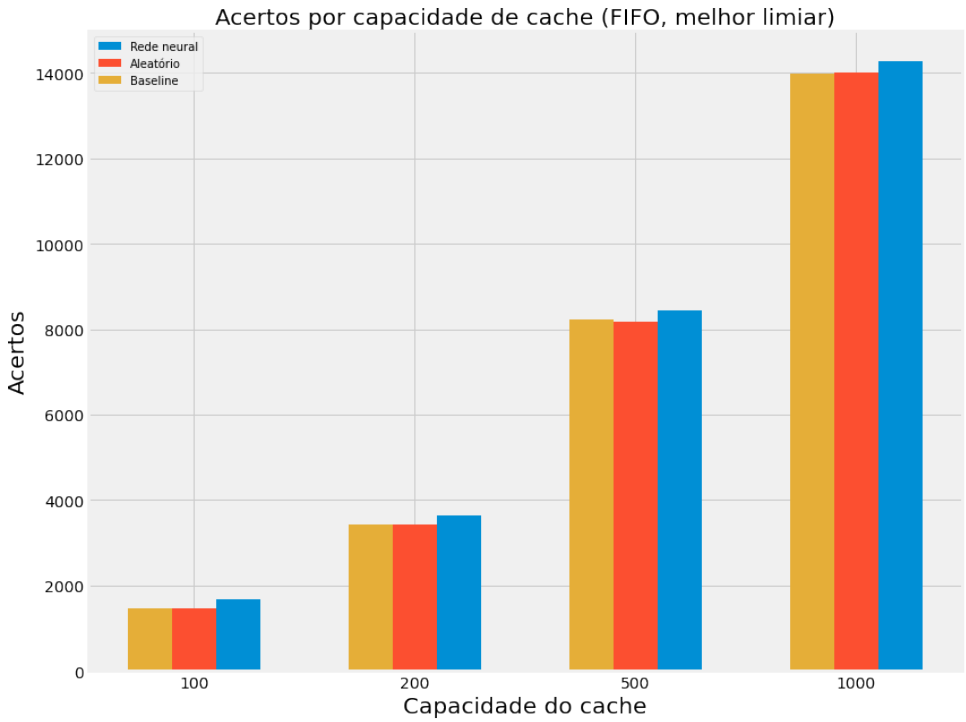

Figura 4.29 – Comparação das médias de acertos para a política FIFO
Fonte: O autor.

Como podemos ver, a inclusão do decisor neural gerou ganhos de até 16% em relação ao teste base, com ganhos menos expressivos a medida que a capacidade cresce.

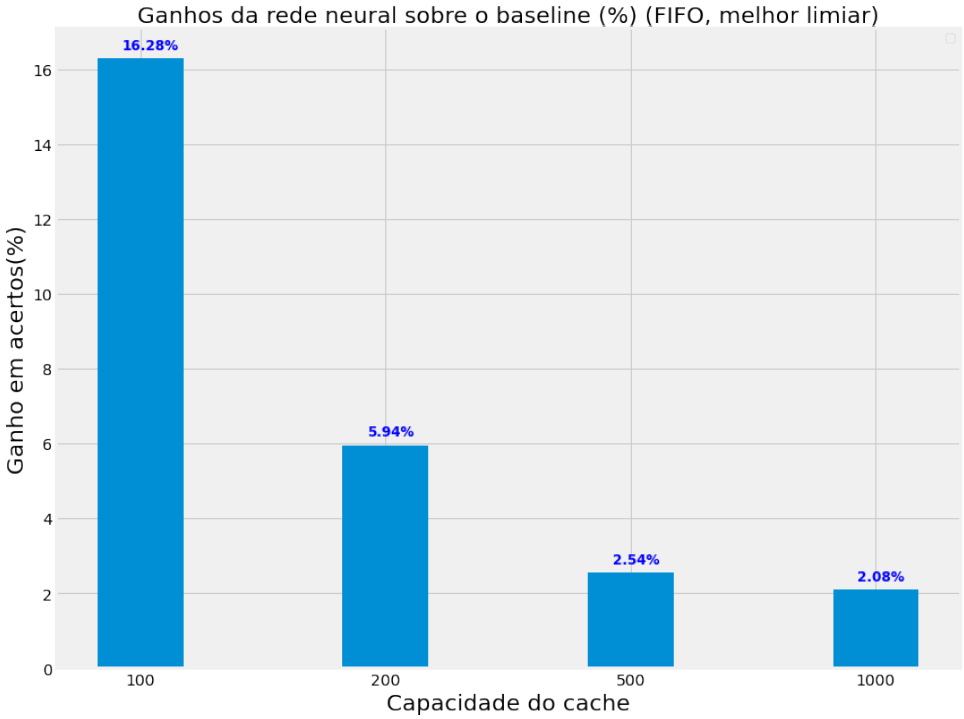

Figura 4.30 – Ganho percentual da rede neural sobre o teste base para a política FIFO
Fonte: O autor.

4.4.4.2 LRU

A seguir, são realizados também os testes para determinar o melhor valor de limiar a ser utilizado com a LRU, conforme mostra a Figura 4.31. Nela pode-se ver o comportamento da média de acertos para cada capacidade e diferentes valores de limiares. Fica evidente que para capacidades de 100 e 200, um limiar de 300 trouxe maiores ganhos, enquanto para os demais valores, um limiar de 350 resultou em mais acertos.

Novamente, é selecionado, para cada capacidade, o valor que resultou no maior número de acertos.

De forma análoga ao feito para a FIFO, são mostrados nas Figuras 4.32 e 4.33 os comparativos e ganhos da rede neural em relação às demais técnicas com a LRU.

A inclusão do decisor neural gerou ganhos de até 60% em relação ao *baseline*, potencializando a estratégia empregada pela política. Os ganhos mais altos ocorrem em baixas capacidades de *cache*, o que já era esperado visto que, a medida que o número de posições armazenadas aumenta, a média se torna mais homogênea, o que pode dificultar o processo de avaliação. Ainda assim, para esses valores os ganhos ficaram acima dos 40%.

Além disso, percebe-se que o decisor aleatório também gera ganhos em relação ao *baseline*, indicando que a mudança de paradigma é benéfica no contexto estudado.

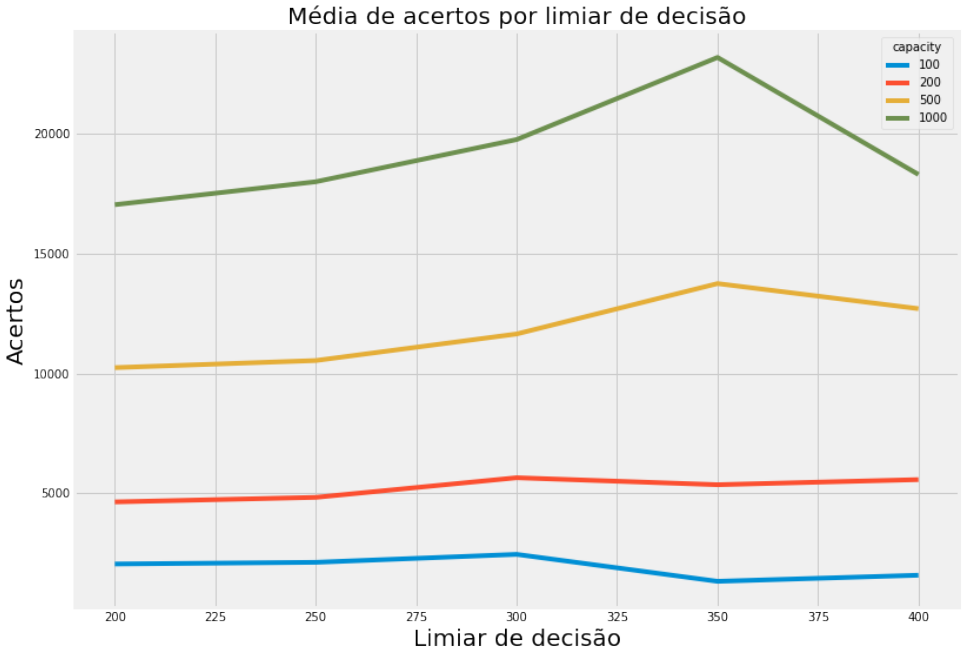

Figura 4.31 – Média de acertos por valor de limiar para a LRU
Fonte: O autor.

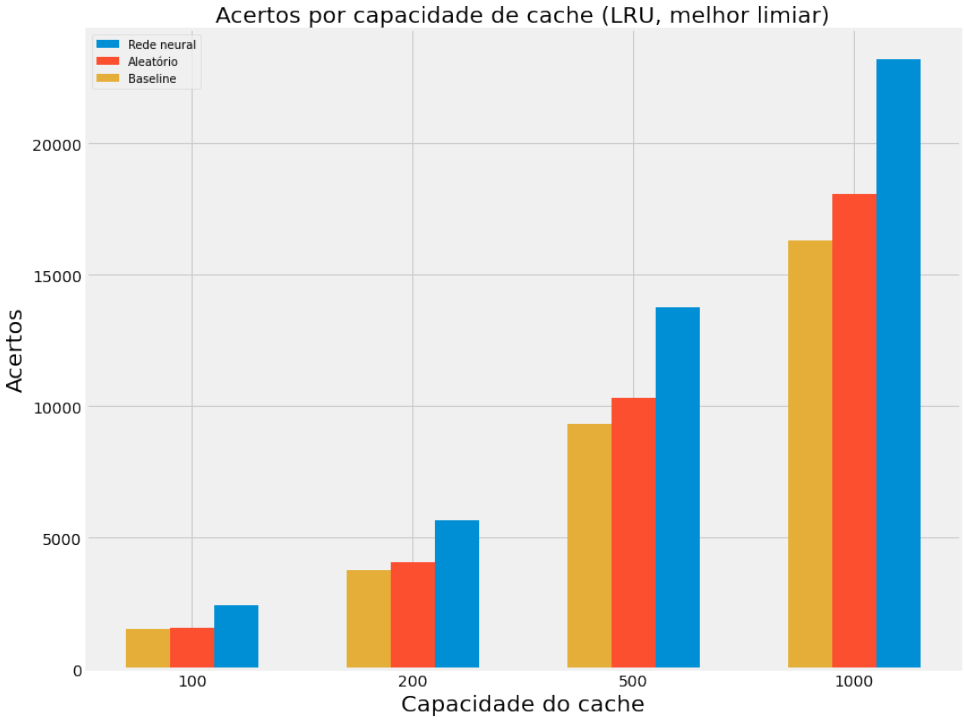

Figura 4.32 – Comparação das médias de acertos para a política LRU
Fonte: O autor.

4.4.4.3 Armazenamentos realizados

Outro ponto a ser analisado é o número de armazenamentos realizados durante a execução. Para a política pura, esse número será a totalidade de sub-tarefas computadas,

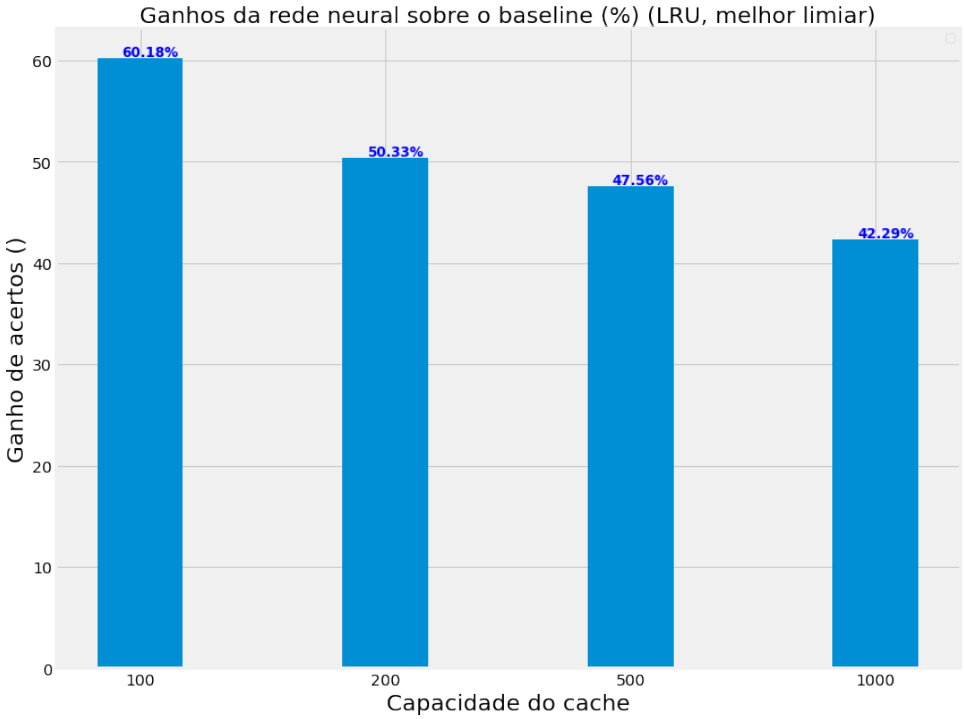

Figura 4.33 – Ganho percentual da rede neural sobre o teste base para a política LRU
Fonte: O autor.

menos o número de acertos em *cache*, como mostrado anteriormente na Figura 4.25.

Entretanto, quando o *cache* se torna estático, as computações deixam de ser guardadas, reduzindo o tempo gasto nesse processo. A Figura 4.34 mostra a quantidade de *sets* realizados com a política LRU nos três cenários.

Fica evidente que a utilização do decisor aleatório reduz o número de armazenamentos em cerca de 50%, o que era o esperado.

Além disso, o decisor baseado em redes neurais consegue reduzir esse número para menos de 10% em relação ao *baseline* em todos os casos. Para a capacidade de 100, por exemplo, para o qual houve um número de acertos até 60% maior, é vista uma redução na quantidade de salvamentos para até menos de 5% da original.

4.4.4.4 Limitações

O modelo baseado na rede neural é bastante poderoso e apresenta ganhos significativos em número de acertos, não obstante, a técnica apresenta limitações quanto ao tempo de execução, como mostra a Tabela 4.2. Nela é possível perceber que a introdução do decisor baseado em redes neurais foi mais custosa que o uso sistema sem decisor. Assim, é necessário entender em que contexto haverão ganhos também em termos do tempo.

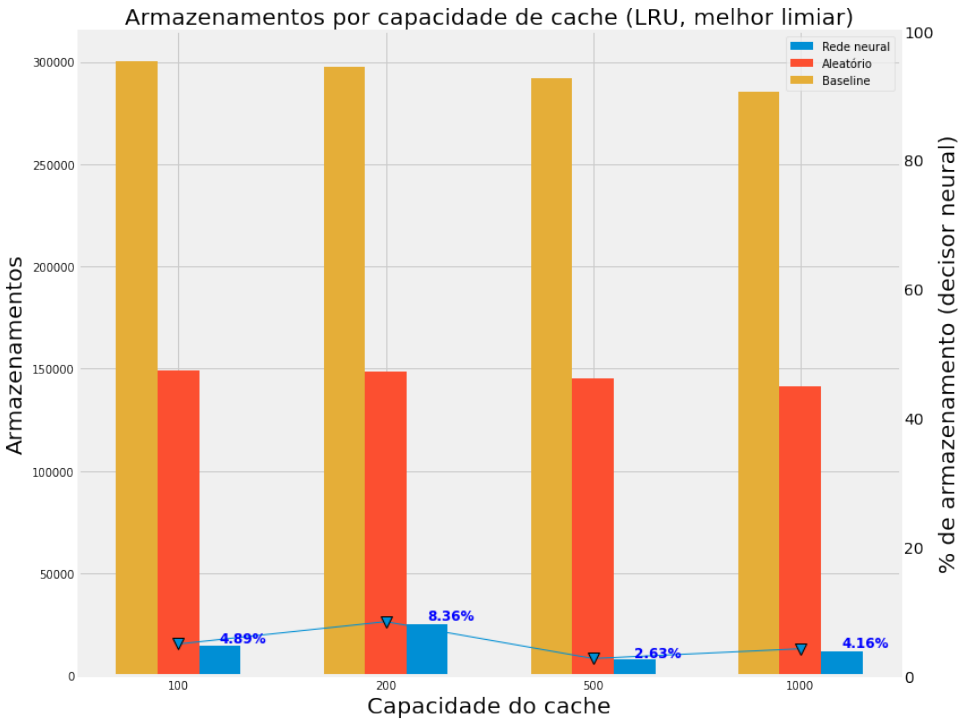

Figura 4.34 – Quantidade de *sets* realizados para a política LRU
Fonte: O autor.

Tabela 4.2 – Tempos médios da aplicação com decisor neural (LRU)

	Capacidade	0	100	200	500	1000
Tempo médio (s)	Com decisor	-	187.3	188.27	172.4	181.49
Tempo médio (s)	Sem decisor	17.1	17.7	17.2	16.0	14.7

Por se tratar de um modelo complexo, o tempo gasto na predição pode ser alto, tornando-o ideal para aplicações mais pesadas, cujas computações são mais custosas, como tratado na próxima seção.

4.4.5 Testes de tempo de execução

A fim de melhor entender as relações de tempo na aplicação, são apresentados os testes relacionados ao *cache* e ao processo de tomada de decisão.

4.4.5.1 Consulta e armazenamento em *cache*

Primeiro, a Figura 4.35 traz o tempo médio de consulta aos dados armazenados(*get*) conforme a capacidade utilizada. São realizadas dez mil operações a partir de um *cache* cheio para cada valor de capacidade máxima, variando de 10 a 2000 em intervalos de 10, totalizando 200 pontos.

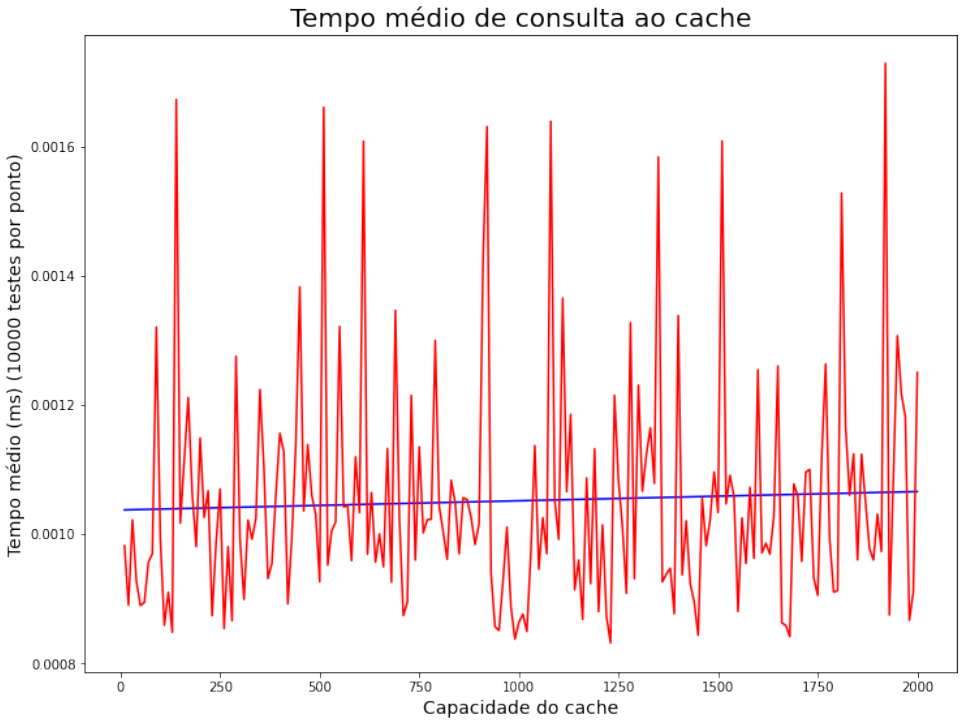

Figura 4.35 – Tempo médio de consulta ao *cache*
Fonte: O autor.

Inclinação da reta: 1.42726628e-08 R_2 : 0.00220

A análise visual mostra que há pouca influência da capacidade máxima no tempo de consulta, o que fica comprovado quando uma reta é ajustada sobre esses dados, vista em azul. O coeficiente de inclinação obtido foi de $1,4273.10^{-8}$, com a média geral próxima a $0,00105$ segundos, indicando pouco ou nenhuma proporcionalidade entre o tempo da operação e a capacidade.

Já nos tempos de *set*, como visto nas Figuras 4.36 (LRU) e 4.37 (FIFO), nota-se que existe um aumento considerável no tempo gasto na operação, com as maiores diferenças observadas para capacidades inferiores a 500. Para essa operação, há não só maior inclinação, mas também maior valor de R_2, o que indica que a reta captura melhor o comportamento desses dados.

Dessa forma, para ambos os casos, manter um *cache* de menor capacidade se mostrou ser mais eficiente do ponto de vista das operações individuais, principalmente devida à operação de *set*.

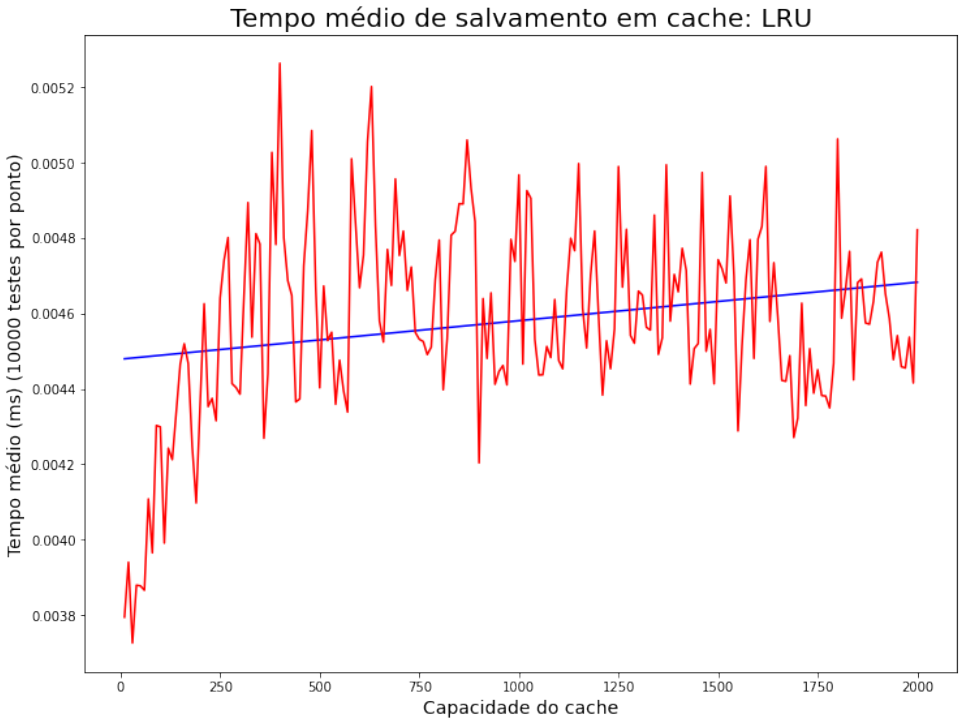

Figura 4.36 – Tempo médio de armazenamento no *cache* para a política LRU
Fonte: O autor.

Inclinação da reta: 1.01996049e-07 R_2 : 0.0537

4.4.5.2 Tempo de decisão

A seguir buca-se analisar o processo de decisão da rede neural, composto de 4 etapas: normalização da tarefa, cálculo da média do *cache*, normalização da média obtida e predição. Assim, é necessário investigar o impacto de cada uma dessas ações no tempo de execução da aplicação.

A Tabela 4.3 traz os dados obtidos através dos testes de cada etapa de forma isolada. Nela fica evidente que os processos de aplicação do normalizador e de cálculo da média do *cache* são mais rápidos que o da decisão da rede neural por uma ampla margem. Para o cálculo da média, foram realizados 100 testes, sendo os 20 primeiros descartados, para evitar o impacto de quaisquer inicializações do código.

Nesse sentido, a adição do decisor pode ter significativo impacto no tempo de execução da aplicação e, com isso em mente, é preciso entender em que condições o emprego dessa técnica começa a gerar ganhos efetivos.

Assim, para um teste feito utilizando a Equação 3.3 com $I = 300$ tarefas e um *cache* LRU de tamanho 200, temos $H = 372$, e, com os resultados anteriores, $g = 1.6$, $t_d = 0.05$.

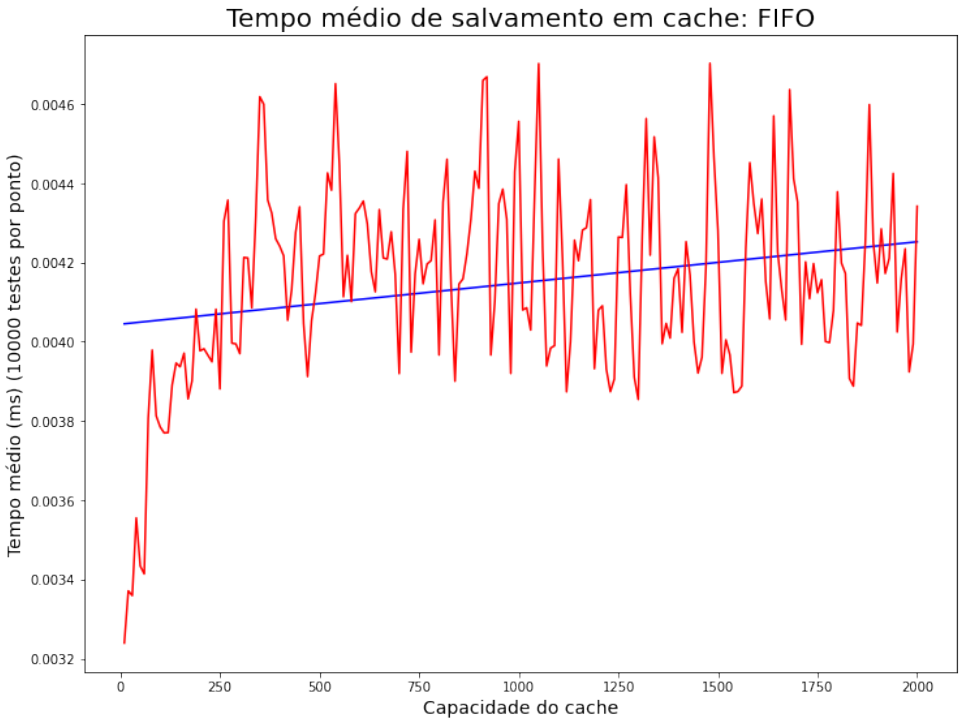

Figura 4.37 – Tempo médio de armazenamento no *cache* para a política FIFO
Fonte: O autor.

Inclinação da reta: 1.0401036e-07 R_2 : 0.0603

Tabela 4.3 – Tempo de decisão (s)

	normalizador	cálculo da média*	tempo de decisão
Média	0,00024	0,00081	0,04965
Desvio padrão	$2,27.10^{-5}$	$9,615.10^{-5}$	0,004

* Tempos médios para capacidade de 100

Dessa forma, é possível estimar o valor mínimo de $t \approx 0.07$.

Com isso, é realizado um teste aplicando um atraso sintético de 0.1 segundos em cada sub-tarefa, e pode-se constatar que, para tarefas mais pesadas, o decisor de fato gera ganhos em tempo de execução, como mostra a Tabela 4.4. Nesse caso, por ser uma tarefa longa, apenas um teste foi feito para fins de prova de conceito.

Tabela 4.4 – Tempo total da aplicação com atraso com e sem o decisor

Aplicação	Acertos	Tempo total(s)
Sem decisor	372	7599.57
Com decisor	542	7559.85

Evidencia-se, então, que quanto mais custosas forem as tarefas em questão, menor o impacto do tempo de decisão da rede neural e maiores serão os ganhos.

4.4.6 Discussões: Aplicação completa

Sob a ótica de aumentar o número de *hits*, os resultados obtidos até aqui mostram que a introdução do modelo híbrido, resultante da adição do decisor, pode potencializar a estratégia aplicada pela política de *cache*, gerando ganhos de até 60% sobre os valores base.

Foi evidenciado também que, para um decisor baseado em redes neurais como o proposto, os ganhos mais expressivos ocorrem em situações de maior restrição de memória, já que a média do conteúdo do *cache* é menos homogênea, embora ganhos acima de 40% tenham sido observados até para os maiores valores de capacidade.

Além disso, o modelo híbrido consegue evitar a maior parte dos armazenamentos desnecessários, impedindo grande parte dos *sets* de baixo valor futuro.

Por fim, foi constatado que a inclusão de uma etapa de predição com redes neurais pode apresentar um impacto significativo na aplicação, a depender do tempo de execução das sub-tarefas da mesma. Por isso, a técnica é ideal para aplicações mais custosas em processamento.

Capítulo 5

Considerações

Com o advento da quarta revolução industrial e da sociedade 5.0, quantidades cada vez maiores precisam ser processadas e a demanda por capacidade computacional cresce como nunca. Em vista desse cenário, técnicas inteligentes que otimizem o uso de computações desempenham um papel fundamental.

Neste trabalho, iniciamos a investigação do uso de mecanismos de gerenciamento de *cache* aliados a algoritmos de aprendizado de máquina para maximizar os acertos em *cache*, visando aumentar a eficiência de aplicações que lidam com computações custosas.

O objetivo final é a implementação de um tomador de decisão que, baseado na similaridade de uma computação com o conjunto já em *cache*, seja capaz selecionar quando uma nova entrada deve ser colocada na memória e quando não, alternando entre os paradigmas estático e dinâmico. Este decisor é formado de uma política de gerenciamento de memória e um modelo preditor baseado em aprendizado de máquina.

Foram testadas as políticas *FIFO* e *LRU* aliadas a um tomador de decisão baseado em redes neurais siamesas que, a partir da requisição e conteúdo atual do *cache*, consegue predizer o valor futuro das computações parciais geradas.

O sistema proposto obteve ganhos significativos quando aplicado em um cenário de tarefas independentes, cujos parâmetros de entrada pertencem a um contexto definido, como é o caso da LAC.

5.0.0.1 Propostas de continuidade

Os resultados alcançados pelo método proposto superaram as expectativas iniciais, atingindo ganhos percentuais elevados. Apesar disso, a há espaço para melhorias e evo-

lução da linha de pesquisa, como as citadas a seguir:

- Utilização de GPUs no processo de treinamento e predição da rede neural, visando diminuir o tempo gasto no processo de decisão.
- Utilização do método em diferentes aplicações, fora do contexto de classificação, de modo a entender mais a fundo as limitações do mesmo.
- Introdução de maior número de *workers*, com *caches* compartilhados e individuais.

Referências

ALCANTARA, Giovanni. Empirical analysis of non-linear activation functions for Deep Neural Networks in classification tasks, 2017. arXiv: 1710.11272 [cs.LG].

BARUH, Lemi; POPESCU, Mihaela. Big data analytics and the limits of privacy self-management. **New Media & Society**, v. 19, n. 4, p. 579–596, 2017. DOI: 10.1177/1461444815614001. Disponível em: <https://doi.org/10.1177/1461444815614001>.

BERGER, Daniel S. et al. Exact analysis of TTL cache networks. **Performance Evaluation**, v. 79, p. 2–23, 2014. Special Issue: Performance 2014. ISSN 0166-5316. DOI: https://doi.org/10.1016/j.peva.2014.07.001. Disponível em: <https://www.sciencedirect.com/science/article/pii/S0166531614000649>.

BORA, Dibya; GUPTA, Dr. Effect of Different Distance Measures on the Performance of K-Means Algorithm: An Experimental Study in Matlab. v. 5, mai. 2014.

BRAGA, Antonio de Pádua; LUDERMIR, Teresa Bernarda; CARVALHO, André Carlos Ponce de Leon Ferreira. **Redes neurais artificiais: teoria e aplicações**. [S.l.]: LTC, 2000.

BRAMER, M. **Principles of Data Mining**. 3. ed. [S.l.]: Springer-Verlag London, 2016. DOI: 10.1007/978-1-4471-7307-6.

BROMLEY, Jane et al. Signature Verification using a "Siamese"Time Delay Neural Network. **International Journal of Pattern Recognition and Artificial Intelligence**, v. 7, p. 25, ago. 1993. DOI: 10.1142/S0218001493000339.

BRUNTON, Steven L.; KUTZ, J. Nathan. **Data-Driven Science and Engineering: Machine Learning, Dynamical Systems, and Control**. [S.l.]: Cambridge University Press, 2019. DOI: 10.1017/9781108380690.

CARTWRIGHT, H. **Artificial Neural Networks**. 3. ed. Nova York: Humana Press, 2015. (Methods in Molecular Biology).

CASTELLANO, Giovanna et al. Crowd Detection in Aerial Images Using Spatial Graphs and Fully-Convolutional Neural Networks. **IEEE Access**, v. 8, p. 64534–64544, 2020. DOI: 10.1109/ACCESS.2020.2984768.

CASTELLÓ, Vicent Ortiz et al. High-Profile VRU Detection on Resource-Constrained Hardware Using YOLOv3/v4 on BDD100K. **Journal of Imaging**, MDPI AG, v. 6, n. 12, p. 142, dez. 2020. ISSN 2313-433X. DOI: 10.3390/jimaging6120142. Disponível em: <http://dx.doi.org/10.3390/jimaging6120142>.

DALENOGARE, Lucas Santos et al. The expected contribution of Industry 4.0 technologies for industrial performance. **International Journal of Production Economics**, v. 204, p. 383–394, 2018. ISSN 0925-5273. DOI: https://doi.org/10.1016/j.ijpe.2018.08.019. Disponível em: <https://www.sciencedirect.com/science/article/pii/S0925527318303372>.

DENNING, Peter J. Virtual Memory. **ACM Comput. Surv.**, Association for Computing Machinery, New York, NY, USA, v. 2, n. 3, p. 153–189, set. 1970. ISSN 0360-0300. DOI: 10.1145/356571.356573. Disponível em: <https://doi-org.ez107.periodicos.capes.gov.br/10.1145/356571.356573>.

EFFELSBERG, Wolfgang; HAERDER, Theo. Principles of Database Buffer Management. **ACM Trans. Database Syst.**, Association for Computing Machinery, New York, NY, USA, v. 9, n. 4, p. 560–595, dez. 1984. ISSN 0362-5915. DOI: 10.1145/1994.2022. Disponível em: <https://doi.org/10.1145/1994.2022>.

FEDCHENKO, Vladyslav; NEGLIA, Giovanni; RIBEIRO, Bruno. Feedforward Neural Networks for Caching: Enough or Too Much?, 2018. arXiv: 1810.06930 [cs.NI].

FUKUDA, Kayano. Science, technology and innovation ecosystem transformation toward society 5.0. **International Journal of Production Economics**, v. 220, 2020. ISSN 0925-5273. DOI: https://doi.org/10.1016/j.ijpe.2019.07.033. Disponível em: <https://www.sciencedirect.com/science/article/pii/S0925527319302701>.

GRUS, Joel. **Data science do zero**: primeiras regras com o Python. 1. ed. [S.l.]: Alta Books, 2016.

HUNTER, J. D. Matplotlib: A 2D graphics environment. **Computing in Science & Engineering**, IEEE COMPUTER SOC, v. 9, n. 3, p. 90–95, 2007. DOI: 10.1109/MCSE.2007.55.

JAVAID, QAISAR et al. Cache Memory: An Analysis on Replacement Algorithms and Optimization Techniques. **Mehran University Research Journal of Engineering and Technology**, v. 36, n. 4, p. 831–840, 2017. ISSN 0254-7821. Disponível em: <https://doaj.org/article/68346833056148bb9212e346aed2b96c>.

KERAS documentation. [S.l.: s.n.], 2021. Disponível em: <https://keras.io/>.

KUBAT, M. **An Introduction to Machine Learning**. 2. ed. Cham: Springer, 2017.

LASI, H. et al. Industry 4.0. **Bus Inf Syst Eng**, v. 6, p. 239–242, 2014. DOI: 10.1177/10.1007/s12599-014-0334-4. Disponível em: <https://doi-org.ez107.periodicos.capes.gov.br/10.1007/s12599-014-0334-4>.

LEE, Haeyun; LEE, Kyungsu et al. Local Similarity Siamese Network for Urban Land Change Detection on Remote Sensing Images. **IEEE Journal of Selected Topics in Applied Earth Observations and Remote Sensing**, v. 14, p. 4139–4149, 2021. DOI: 10.1109/JSTARS.2021.3069242.

LEE, Joon HO; KIM, Myoung Ho; LEE, Yoon Joon. Information retrieval based on conceptual distance in IS-A hierarchies. **Journal of Documentation**, v. 49, n. 2, p. 188–207, 1993.

LIN, Dekang. An Information-Theoretic Definition of Similarity. Morgan Kaufmann, p. 296–304, 1998.

LIU, Weibo et al. A survey of deep neural network architectures and their applications. **Neurocomputing**, v. 234, p. 11–26, 2017. ISSN 0925-2312. DOI: https://doi.org/10.1016/j.neucom.2016.12.038. Disponível em: <https://www.sciencedirect.com/science/article/pii/S0925231216315533>.

MEDDEB, Maroua et al. Least fresh first cache replacement policy for NDN-based IoT networks. **Pervasive and Mobile Computing**, v. 52, p. 60–70, 2019. ISSN 1574-1192. DOI: https://doi.org/10.1016/j.pmcj.2018.12.002. Disponível em: <https://www.sciencedirect.com/science/article/pii/S1574119218300889>.

NERSESSIAN, David. The law and ethics of big data analytics: A new role for international human rights in the search for global standards. **Business Horizons**, v. 61, n. 6, p. 845–854, 2018. ETHICS, CULTURE, AND PEDAGOGICAL PRACTICES IN THE GLOBAL CONTEXT. ISSN 0007-6813. DOI: https://doi.org/10.1016/j.bushor.2018.07.006. Disponível em: <https://www.sciencedirect.com/science/article/pii/S0007681318301095>.

NUMPY documentation. [S.l.: s.n.], 2021. Disponível em: <https://numpy.org/doc/stable/>.

PANDAS documentation. [S.l.: s.n.], 2021. Disponível em: <https://pandas.pydata.org/docs/index.html>.

PEDREGOSA, F. et al. Scikit-learn: Machine Learning in Python. **Journal of Machine Learning Research**, v. 12, p. 2825–2830, 2011.

PIRES, Michel et al. Efficient Parallel Associative Classification Based on Rules Memoization. In: RODRIGUES, João M. F. et al. (Ed.). **Computational Science – ICCS 2019**. Cham: Springer International Publishing, 2019. P. 31–44.

PYTHON 3.7 documentation. [S.l.: s.n.], 2021. Disponível em: <https://docs.python.org/3.7/>.

RASCHKA, Sebastian; PATTERSON, Joshua; NOLET, Corey. Machine Learning in Python: Main Developments and Technology Trends in Data Science, Machine Learning, and Artificial Intelligence. **Information**, v. 11, n. 4, 2020. ISSN 2078-2489. DOI: 10.3390/info11040193. Disponível em: <https://www.mdpi.com/2078-2489/11/4/193>.

REINEKE, JAN; GRUND, DANIEL. Sensitivity of Cache Replacement Policies. **ACM Transactions on Embedded Computing Systems**, 42–42:18, 2013. ISSN 15399087. Disponível em: <http://search-ebscohost-com.ez107.periodicos.capes.gov.br/login.aspx?direct=true&db=iih&AN=96122087&lang=pt-br&site=ehost-live>.

SIDDIQUI, Tamanna; ALKADRI, Mohammad; KHAN, Najeeb Ahmad. Review of Programming Languages and Tools for Big Data Analytics. **International Journal of Advanced Research in Computer Science**, v. 8, n. 5, p. 1112–1118, 2017. ISSN 0976-5697. DOI: 10.26483/ijarcs.v8i5.3578. Disponível em: <http://www.ijarcs.info/index.php/Ijarcs/article/view/3578>.

TANENBAUM, Andrew S. **Sistemas Operacionais Modernos**. 4. ed. São Paulo: Pearson, 2016.

TENSORFLOW documentation. [S.l.: s.n.], 2021. Disponível em: <https://www.tensorflow.org/>.

TIAN, Geng; LIEBELT, Michael. An effectiveness-based adaptive cache replacement policy. **Microprocessors and Microsystems**, v. 38, n. 1, p. 98–111, 2013. ISSN 0141-9331. DOI: https://doi.org/10.1016/j.micpro.2013.11.011.

VELOSO, Adriano; MEIRA, Wagner; GONÇALVES, Marcos et al. Calibrated lazy associative classification. eng. **Information sciences**, Elsevier Inc, v. 181, n. 13, p. 2656–2670, 2011. ISSN 0020-0255.

VELOSO, Adriano; MEIRA, Wagner; ZAKI, Mohammed J. Lazy Associative Classification, p. 645–654, 2006. DOI: 10.1109/ICDM.2006.96.

XU, Rui; WUNSCH, D. Survey of clustering algorithms. **IEEE Transactions on Neural Networks**, v. 16, n. 3, p. 645–678, 2005. DOI: 10.1109/TNN.2005.845141.

ZHANG, Xi et al. Improving Cache Partitioning Algorithms for Pseudo-LRU Policies. **IEICE Transactions on Information and Systems**, E96.D, n. 12, p. 2514–2523, 2013. DOI: 10.1587/transinf.E96.D.2514.

Apêndice A

Algoritmos em Python

Os algorítimos e scripts utilizados na confecção do trabalho podem ser encontrados na íntegra no seguinte repositório do *GitHub*: https://github.com/TarcisioBalbi/MachineLearning/tree/main/Codigos_TCC