CENTRO FEDERAL DE EDUCAÇÃO TECNOLÓGICA DE MINAS GERAIS
DEPARTAMENTO DE COMPUTAÇÃO
CURSO DE ENGENHARIA DE COMPUTAÇÃO

ALGORITMOS CONCORRENTES PARA RECONHECIMENTO DE PADRÕES DE RECEBIMENTO E ARMAZENAMENTO DE SUCATA METÁLICA EM UMA USINA SIDERÚRGICA

ABRAÃO GONÇALVES MAIA

Orientador: Lucas Pantuza Amorim
Coorientador: Douglas de Oliveira Nunes
Centro Federal de Educação Tecnológica de Minas Gerais - CEFET-MG

TIMÓTEO
01 DE SETEMBRO DE 2016

ABRAÃO GONÇALVES MAIA

ALGORITMOS CONCORRENTES PARA RECONHECIMENTO DE PADRÕES DE RECEBIMENTO E ARMAZENAMENTO DE SUCATA METÁLICA EM UMA USINA SIDERÚRGICA

CENTRO FEDERAL DE EDUCAÇÃO TECNOLÓGICA DE MINAS GERAIS
DEPARTAMENTO DE COMPUTAÇÃO
CURSO DE ENGENHARIA DE COMPUTAÇÃO
TIMÓTEO
01 DE SETEMBRO DE 2016

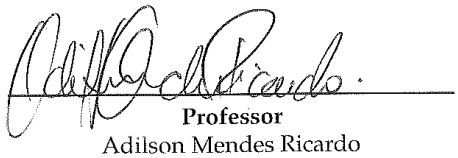
ABRAÃO GONÇALVES MAIA

ALGORITMOS CONCORRENTES PARA RECONHECIMENTO DE PADRÕES DE RECEBIMENTO E ARMAZENAMENTO DE SUCATA METÁLICA EM UMA USINA SIDERÚRGICA

Trabalho aprovado. Timóteo, 01 de Setembro de 2016

Lucas Pantuza Amorim
Orientador

Professor
Douglas de Oliveira Nunes



Professor
Adilson Mendes Ricardo

CENTRO FEDERAL DE EDUCAÇÃO TECNOLÓGICA DE MINAS GERAIS
DEPARTAMENTO DE COMPUTAÇÃO
CURSO DE ENGENHARIA DE COMPUTAÇÃO
TIMÓTEO
01 DE SETEMBRO DE 2016

Aos meus filhos Pedro e Emanuel...

Agradecimentos

A Deus por ter me permitido chegar até aqui quando nem mesmo eu acreditei, a minha esposa por todo o cuidado, carinho e paciência comigo, aos meus pais pelo apoio e aos familiares e amigos pelo incentivo.

"Você pode ser e fazer o que quiser na vida, desde que estude, trabalhe, persista e sempre faça mais do que esperam de você!" (Dona Zuleika)

Resumo

Este trabalho apresenta uma pesquisa que tem como objetivo analisar o desempenho de alguns algoritmos de classificação e reconhecimento de padrões dentro da área siderúrgica. A sucata metálica adquirida pelas empresas do ramo possui em sua composição química elementos metálicos de alto valor econômico que se não forem bem manipulados, poderão servir como fonte de prejuízo para seus negócios. A pesquisa é feita no intuito de buscar uma solução computacional que garanta a qualidade e a confiabilidade dos produtos adquiridos e fornecedores envolvidos no processo. Também tem como foco a otimização desta solução no sentido de garantir um bom desempenho no que diz respeito ao uso dos recursos computacionais. Primeiramente, foi realizado um estudo das técnicas de reconhecimento de padrões e inteligência artificial para tentar encontrar o método mais adequado para tratar o problema. Para isso, foram implementados três algoritmos conhecidos da área: o KMeans, o Knn e o Perceptron de Múltiplas Camadas com o objetivo de tentar encontrar a solução mais adequada. Por fim, esses algoritmos foram otimizados utilizando o paradigma de programação paralela. Essa prática tem como objetivo aumentar a performance dos mesmos e otimizar o uso dos recursos computacionais disponíveis.

Palavras-chave: inteligência artificial. reconhecimento de padrões. siderurgia. programação concorrente. elemento químico. espectômetro.

Abstract

This work represent a search that has as objective the performance of some algorithms of classification and recognition into of steel mill area. The metallic scrap acquired by the companies in the same business have in their chemical composition, metallic elements from high economic value that if they not be handled correctly, they may serve as a source of prejudice for their business. The search is done with intention to seek a computational solution that ensures the quality and reliability of the product purchased and suppliers involved in the process. It had also as focus the optimization this solution on the way to ensures a good performance as regards the use computational resources. First of all, it was done a study of the techniques of pattern recognition and artificial intelligence to try to find the most appropriate method to treat the problem. They have been implemented for this, three known algorithms of the area: the Kmeans, the Knn and the Perceptron Multiple Layers with abjective to try to find the most appropriate solution. Finally, these algorithms are optimized using the parallel programming paradigm. This practice aims to increase the performance itself and optimized the use of available computing resource.

keywords: Artificial Intelligence, Pattern Recognition, Steel Mill, Concurrent Programming, Chemical Element and Spectrometer.

Lista de Figuras

Lista de Tabelas

Sumário

1 Introdução

1.1 Contextualização e Motivação

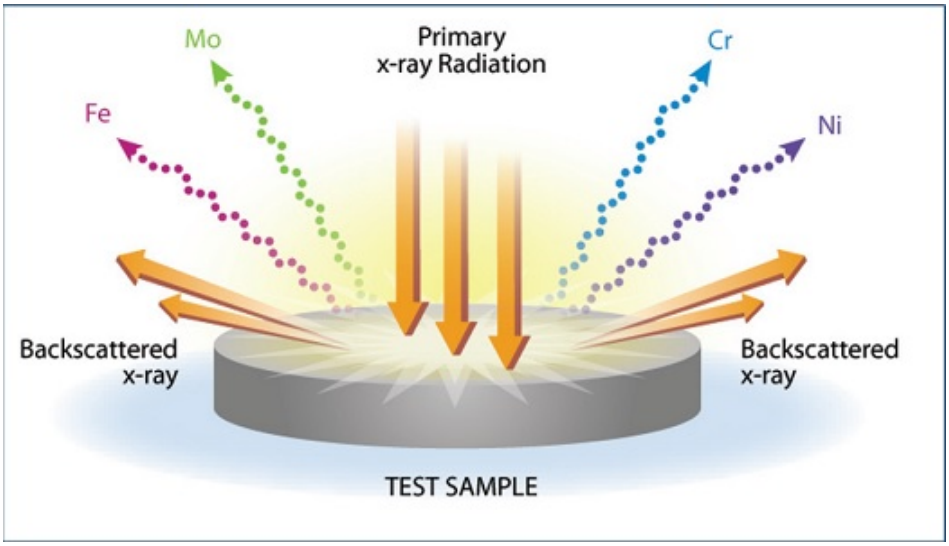
1.1.1 Processo de Fabricação dos Aços Inoxidáveis

O processo de fabricação dos aços inoxidáveis em uma usina siderúrgica passa por diversas etapas. Uma matéria-prima essencial para este processo é a sucata metálica que normalmente é gerada pela própria usina através de seus processos de produção ou adquirida de terceiros. De uma forma bem resumida, pode-se dizer que o processo do pré-metal (liga que dá origem as placas de aço) se dá através da combinação desta sucata com outros elementos químicos.

"Os aços inoxidáveis se dividem em três grupos: austeníticos, martensíticos e ferríticos [Souza (2006)]". Dentro de cada um desses grupos existem diversas categorias de aços diferentes, cada uma com sua própria composição química e aplicação. Por essa razão é que toda a sucata metálica utilizada no processo de fabricação desses aços deve ser analisada e separada de acordo com sua composição química, para que possa atender todas as especificações exigidas para o produto final.

A sucata metálica normalmente é armazenada em uma área dentro das siderúrgicas denominada Pátio de Sucatas. É nesse local que é feito todo o tratamento necessário na sucata como: inspeção, separação, beneficiamento entre outros, para que ela possa fazer parte do processo de produção. No caso da sucata adquirida, que é toda a matéria-prima recebida de fornecedores, é necessário que se tenha uma atenção especial na parte de inspeção, pois é nessa hora que se garante a qualidade do material. Esta etapa é feita utilizando uma técnica chamada de Espectometria por Fluorescência de Raios X (X Ray Fluorescence – XRF) que *"consiste em medir a intensidade dos raios x característicos que são emitidos pelos elementos que constituem a amostra, quando esta é excitada por partículas como elétrons, prótons ou íons que são produzidos na sua maioria por tubos de raios* x *[Anderson e Shein (2009)]."* A figura 1 ilustra este processo.

Figura 1 – Processo básico da tecnologia XRF.



Fonte: [Anderson e Shein (2009)]

Existem diversos equipamentos no mercado que são comercializados para este fim e são conhecidos como Espectômetro de Raio X. Eles utilizam algoritmos avançados para poder classificar com precisão a maioria dos materiais amostrados. Abaixo segue uma descrição de como o processo de inspeção da sucata adquirida é feito utilizando um espectômetro:

1. a carga (que normalmente é transportada por caminhões) chega e é descarregada em um local separado;
2. o inspetor checa diversas "partes"da carga com o espectômetro (normalmente o número de checagens é de uma checagem para cada tonelada de carga);
3. o espectômetro mostra através de uma tela a porcentagem presente de cada elemento químico encontrado na checagem;
4. através da média da porcentagem das amostras é que se calcula a qualidade do material.

A figura 2 exibe um modelo de espectômetro:

Figura 2 – Espectômetro modelo NITON XLT3 GOLDD.

Fonte: [Anderson e Shein (2009)]

1.1.2 Composição Química dos Aços Inoxidáveis

Os aços inoxidáveis são ligas de Ferro (Fe) e Cromo (Cr) que também podem conter Niquel (Ni), molibdênio (Mo), além de outros elementos metálicos. O Cr é um elemento que deve estar sempre presente (deve possuir teor mínimo de 10,5%) devido ao seu alto poder de resistência a corrosão, já o níquel (que não está presente em todos os aços) possui a a característica de melhorar as propriedades mecânicas do material [Araújo (2005)].

A princípio, os aços inoxidáveis podem ser divididos em dois grandes grupos:

- Série 300: aços inoxidáveis austenísticos, ligas de Fe-Cr-Ni;
- Serie 400: aços inoxidáveis ferríticos, basicamente ligas de Fe-Cr.

Em todos os aços inoxidáveis estão presentes também o Carbono (C) e outros elementos como o silício (Si), manganês (Mn), fósforo (P), Ferro (Fe) e enxofre (S).

A adição do níquel, em determinadas quantidades, transforma a estrutura ferrítica em austenística e a consequência disto é uma alteração em muitas propriedades do aço como excelente resistência a corrosão, excelente ductilidade e excelente soldabilidade. Os aços inoxidáveis austenísticos podem ser usados em temperatura ambiente,

altas temperaturas (até 1150 °C) e até em temperaturas criogênicas (baixíssimas temperaturas), uma série de alternativas que dificilmente podem ser conseguidas com outros materiais [Araújo (2005)].

O aço inoxidável 304 (18%Cr-8%Ni) é o aço da série 300 mais popular. A adição de 2% Molibdênio (Mb) o transforma no aço 316, material indicado para uso em meios agressivos como regiões litorâneas e até em equipamentos marítmos por seu alto poder de resistência a corrosão nestes meios.

Voltando a questão da separação da sucata, pode-se concluir que esta etapa é de suma importância no processo de fabricação dos aços pois é nela que são separadas as ligas mais importantes como o níquel e molibdênio. Estes elementos metálicos possuem altos valores econômicos e o uso incorreto deles pode gerar consequências desagradáveis ao processo de produção como: contaminação do aço, perdas econômicas, atraso de produção, entre outras.

1.1.3 Processo de *Data Mining*

O processo de *Data Mining* consiste em agregar e organizar dados encontrando entre eles padrões, mudanças, associações e anomalias relevantes. São utilizadas técnicas de recuperação da informação, inteligência artificial, reconhecimento de padrões e estatística para procurar correlações entre diferentes dados coletados que permitam gerar um conhecimento sobre um determinado processo, empresa ou indivíduo [Goldschmidt e Passos (2005)].

De uma forma geral pode-se dizer que o processo de *Data Mining* consiste em transformar dados brutos em conhecimento extraindo padrões e construindo modelos a partir do conjunto de dados disponíveis para descobrir conhecimento novo e útil.

As empresas possuem diversos dados de seus processos coletados ao longo dos anos e esses dados por muitas vezes ficam armazenados em seus computadores de forma obsoleta. Possuem muitos dados mas são carentes de informações relevantes que podem gerar conhecimento e entendimento sobre seu negócio.

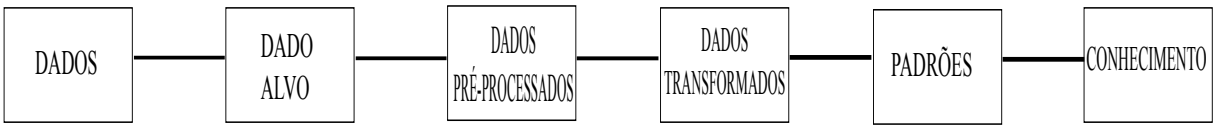
A maioria dos produtos desenvolvidos na indústria é feito através de hipóteses que são levantadas através dos dados coletados acerca de clientes e fornecedores. Partindo desse princípio pode-se concluir que o processo de *Data Mining* pode ser uma alternativa bastante eciente para descobrir padrões sobre a eciência de seus negócios e a qualidade dos produtos ofertados pela empresa.

Através do *Data Mining* cada vez mais empresas tem conseguido se destacar no mercado moldando seus produtos, prevendo tendências e maximizando seus lucros. Segundo [(DANTAS et al., 2008)] *"as técnicas de Data Mining podem proporcionar um inteligência competitiva a nível empresarial e isso pode levar a empresa a atingir suas metas mais*

rapidamente."

A figura 3 descreve as etapas da mineração de dados:

Figura 3 – Processo de descoberta da informação



Fonte: Elaborado pelo Autor

1.2 Objetivos do Trabalho

Considerando que o processo de aquisição de sucata mencionado anteriormente possui muitos dados coletados e que esses dados normalmente ficam armazenados como na forma descrita acima, este trabalho terá como principal objetivo utilizar esses dados e algumas técnicas de *Data Mining* como reconhecimento de padrões e inteligência artificial para tentar encontrar informações úteis para as empresas siderúrgicas gerirem melhor este processo.

Como o níquel e molibdênio são elementos metálicos de maior valor econômico sobre os demais e a quantidade presente deles na sucata adquirida deve ser na quantidade correta, o objetivo específico será tentar utilizar algoritmos de classificação e técnicas de inteligência artificial para encontrar informações relevantes sobre a qualidade dessa sucata e consequentemente sobre os fornecedores envolvidos no processo. Serão utilizadas também técnicas de programação concorrente para melhorar a performance computacional desses algoritmos.

Será utilizada uma base de dados já existente e novos dados poderão ser gerados de forma aleatória caso a base de dados não seja suficiente para extrair as informações necessárias. Serão estudados os algoritmos mais indicados para este tipo de pesquisa e quais as formas mais eficientes para implementá-los. Os algoritmos serão paralelizados utilizando técnicas de programação concorrente para poder melhorar sua performance computacional.

Espera-se que ao final do trabalho seja estabelecido um método que possa servir de base para novas implementações na área de gestão de processos de recebimento e armazenamento de sucata nas usinas siderúrgicas. Esta nova fórmula deve contemplar maneiras de se garantir a qualidade da sucata adquirida e aumentar o grau de confiabilidade dos fornecedores envolvidos no processo. O trabalho terá como foco principal apenas as porcentagens de níquel, cromo e molibdênio, mas nada impede que

futuramente possam ser feitas novas pesquisas abrangendo a composição química da sucata como um todo.

1.3 Estrutura do Trabalho

Este trabalho está dividido em 5 capítulos além da introdução, no primeiro são abordados os trabalhos pesquisados na área e que serviram de motivação para o estudo, o segundo descreve toda a fundamentação teórica relacionada aos temas que são objetos do estudo como reconhecimento de padrões, inteligência artificial, algoritmos de classificação, entre outros, o terceiro descreve a metodologia usada desde a coleta de informações até a fase de implementação dos algoritmos, o quarto descreve os resultados encontrados e o quinto descreve as conclusões que foram que foram obtidas após o trabalho e sugere novos trabalhos futuros que poderão ser realizados.

2 Trabalhos Relacionados

2.1 Reconhecimento de Padrões

Várias abordagens envolvendo redes neurais e algoritmos de classificação tem sido propostas na literatura para mineração de dados e detecção de padrões em ambientes industriais. Isto mostra o quanto essas técnicas são eficientes na análise de dados e resolução de problemas.

Alvaro et al. (2008) utiliza o algoritmo de classificação *K-Means* para otimizar a logística de transporte de uma empresa siderúrgica. O algoritmo tenta encontrar objetos de maior similaridade, ou seja, aqueles que possuem a menor distância. Este procedimento é recursivo e só acaba quando os grupos (ou clusters[1]) já estão todos formados. O critério de similaridade entre os clusters é feito ao contrário, ou seja, é calculado como sendo a maior distância entre eles. O número de centróides[2] é definido previamente. O estudo realizado propiciou uma diminuição na variedade de tarifas de frete pagas pela empresa em razão da similaridade geográfica e econômica encontrada pelo algoritmo.

Martins et al. (2009) faz uso do algoritmo de classificação Knn e da rede neural SOM em suas pesquisas para tentar automatizar a detecção de defeitos em aços laminados de uma determinada indústria. Foram selecionadas 300 imagens de 640×480 pixels de uma filmagem de uma tira de aço retirada de um processo de laminação com o objetivo de classificar o defeito de ondulação. Após a etapa de pré-processamento o objeto de estudo foi segmentado e particionado no formato de uma matriz 13×13 onde cada parte representou uma sub-imagem de 32×32 pixels. A partir dessas sub-imagens foi montada uma base de dados com 200 amostras das regiões do aço laminado sem defeito e 100 amostras das regiões com defeito. As amostras de treinamento foram convertidas em vetores linha e ficaram representadas como uma matriz 300×1024. Cada linha da matriz foi rotulada com um código dizendo se havia ou não defeito. Após o treinamento as amostras de teste foram submetidas ao classificador e o algoritmo obteve taxa de acerto de 98,72%. Os resultados foram satisfatórios e servirão de base para novas pesquisas.

[1] Grupos de elementos que possuem alguma similaridade entre si
[2] Pontos centrais dos clusters

2.2 Paralelização de Algoritmos

Com o avanço da tecnologia na área de processadores *multicore* e placas gráficas (GPU), estes componentes tem sido cada vez mais alvos de pesquisa nas diversas áreas da computação. A necessidade cada vez maior de produzir aplicações que necessitam de alto desempenho computacional também tem contribuído para a ascensão de trabalhos na área. O que se vê são trabalhos que tentam combinar diversos conceitos para poder chegar a um só objetivo. No caso da paralelização, existem diversos temas combinando reconhecimento de padrões, inteligência artificial e outros assuntos ligados à área para gerar aplicações que sejam otimizadas de tal forma que aproveitem ao máximo todos os recursos computacionais de que dispoem.

Raizer et al. (2009) implementa uma rede neural multicamada com execução paralela utilizando CUDA, uma linguagem de programação baseada em C++ em uma GPU. A rede neural possui três camadas: camada de entrada, camada intermediária e camada de saída. Na fase de propagação para frente do treinamento são executadas multiplicações entre matrizes e vetores. O primeiro passo é efetuar um somatório dos pesos entre o vetor de entrada e a matriz de pesos que liga esta fase a camada intermediária. Depois, é efetuado o cálculo da função de ativação para cada neurônio da camada intermediária, estes passos se repetem para a segunda camada e então começa de fato a fase de retropropagação que é a fase em que o algoritmo faz o caminho inverso. Para calcular o valor de erro dos pesos para a camada de saída é feita uma subtração entre vetores para calcular a diferença entre os valores obtidos e os valores esperados. Enquanto isso, ocorre a multiplicação entre a matriz de pesos e o vetor de erros obtido anteriormente. Para avaliar o desempenho, a quantidade de neurônios da camada intermediária ficou entre 8, 16, 32, 64, 128, 256, 512 e 1024 respectivamente.

Sierra-Canto et al. (2010) utiliza CUDA para implementar uma rede neural paralela utilizando o algoritmo de retropropagação. A rede possui três camadas: camada de entrada, camada intermediária e camada de saída, sendo que esta última possui apenas um neurônio e a quantidade de neurônios das demais são variadas durante a execução dependendo do teste a ser realizado. Foram alocadas matrizes na GPU para armazenar os valores dos pesos da camada intermediária e da camada de saída. O valor das entradas foi alocado em um vetor. O algoritmo possui oito passos, sendo que os quatro primeiros são referentes a propagação para a frente e os quatro últimos a retropropagação. Para cada época de treinamento,é treinado sequencialmente cada caso da base de dados. O desempenho foi comparado variando-se a quantidade de neurônios da camada intermediária.

3 Fundamentação Teórica

3.1 Mineração de Dados

Atualmente, as organizações têm se mostrado muito eficientes na organização, captura e armazenamento de grandes quantidades de dados obtidos em suas operações diárias. Porém, ainda não usam essa grande quantidade de informações para transformá-las em conhecimento que possa ser utilizado em suas próprias atividades. Com os custos de aquisição de *hardware* cada vez mais baixos e o desenvolvimento de novas e complexas estruturas como banco de dados, *data warehouses*, bibliotecas virtuais e outras, novos estudos sobre o que fazer com essa grande quantidade de informações tem surgido.

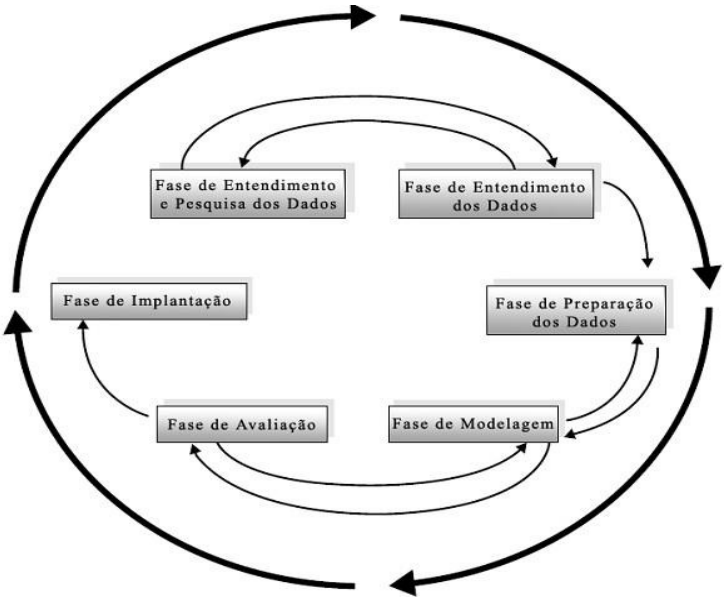
"A transformação de dados em informação (conhecimento) consiste em um processamento manual de todas essas operações por especialistas que, então, produzem relatórios que deverão ser analisados [Camilo e Silva (2009)]".

Mineração de Dados é uma técnica que surgiu no final dos anos 80. Trata-se de um processo analítico que consiste em explorar grande quantidade de dados na busca de padrões consistentes e/ou relacionamentos sistemáticos entre as variáveis. A partir daí, são detectados novos padrões e estes são aplicados a novos subconjuntos de dados [Camilo e Silva (2009)] .

3.1.1 Fases da Mineração de Dados

Existem diversos processos que definem e padronizam as fases da mineração de dados e todos possuem praticamente a mesma estrutura. O CRISP-DM (*Cross-Industry Standard Process of Data Mining*) é um dos modelos de maior aceitação. Ele é composto por seis fases organizadas conforme mostra a figura 4.

Figura 4 – Processo CRISP



Fonte: [Camilo e Silva (2009)]

As fases do processo podem ser definidas como:

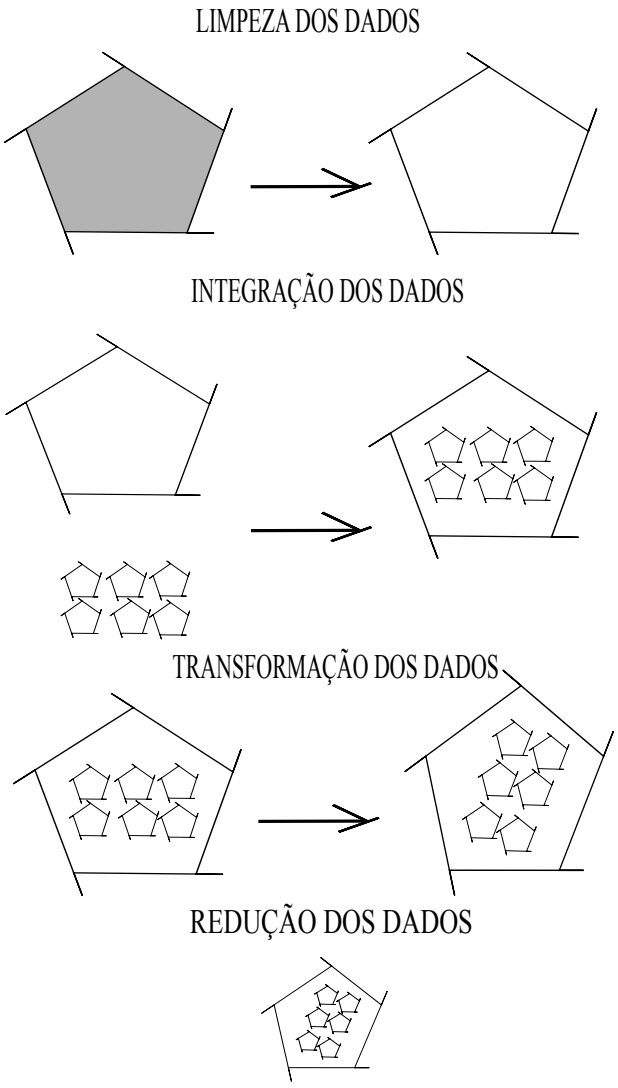
1. Entendimento dos Negócios - mantém o foco no objetivo que se deseja atingir com a mineração de dados;
2. Entendimento dos Dados - as fontes de dados podem originar-se de diversos locais e serem de diversos formatos;
3. Preparação dos Dados - os dados normalmente não estão preparados para os métodos de mineração e por isso, algumas ações podem ser necessárias para filtrar esses dados;
4. Modelagem - fase em que as técnicas de mineração são aplicadas;
5. Avaliação - nesta etapa participam os especialistas dos dados, conhecedores do negócio e os tomadores de decisão. É considerada a fase mais crítica do processo de mineração;
6. Distribuição - após as execuções do modelo com os dados reais os envolvidos são informados dos resultados.

3.1.2 Base de Dados

O conhecimento do tipo de dados que será manipulado é fundamental para a escolha dos métodos mais adequados. Para que sejam aplicados os algoritmos de mineração é necessário conhecer, explorar e preparar os dados. Após uma visão inicial

dos dados bem definida, é necessário uma exploração dos mesmos para que se adquira mais conhecimento, buscando encontrar valores que possam comprometer a qualidade do trabalho como: valores vazios, duplicados, variáveis duplicadas, entre outras. A figura 5 ilustra uma sequência de como os dados podem ser preparados:

Figura 5 – Pré-processamento dos dados.



Fonte: Elaborado pelo Autor

As fases do processo de preparação dos dados podem ser definidas como:

1. Limpeza dos Dados - etapa que visa eliminar os problemas de inconsistência como registros incompletos, valores faltantes, etc. As técnicas usadas nesta etapa variam desde a remoção do registro com problemas até a aplicação de técnicas de agrupamento;

2. Integração dos Dados - existe para padronizar a base de dados já que é comum que os dados obtidos venham de diversas fontes. É feito uma análise aprofundada observando redundâncias, valores conflitantes, etc;

3. Transformação dos Dados - muitas vezes torna-se necessário transformar valores numéricos em categórigos ou vice-versa dependendo do algoritmo a ser utilizado;

4. Redução dos Dados - quando o volume de dados é muito alto, pode tornar a mineração de dados impraticável. Torna-se necessário então, usar algumas técnicas de redução de dados para que a massa de dados original possa ser convertida em uma massa de dados menor, sem perder a representatividade dos dados originais.

3.1.3 Tarefas da Mineração de Dados

A mineração de dados pode ser classificada pela sua capacidade em realizar determinadas tarefas, as mais comumente utilizadas são: a descrição, a classificação, a estimação, a predição e o agrupamento. A forma como são utilizadas está descrita abaixo:

3.1.3.1 Descrição

É a fase que descreve os padrões e tendências que são obtidos através dos dados. É nessa fase que geralmente ocorre uma interpretação para os resultados obtidos. De acordo com Camilo e Silva (2009) essa fase "é muito utilizada em conjunto com as técnicas de análise exploratória de dados, para comprovar a influência de certas variáveis no resultado obtido."

3.1.3.2 Classificação

Fase que visa identificar a classe a que determinado registro pertence. O modelo analisa um conjunto de registros que são fornecidos, onde cada registro já apresenta a classe a qual pertence. A partir daí, o modelo tenta classificar os demais conjuntos de registros através de padrões que ele pré-estabeleceu ao analisar o primeiro conjunto de registros.

3.1.3.3 Estimação

Fase similar a classificação, a diferença é que o valor de uma determinada variável pode ser definido analisando os valores das demais. O registro nessa fase é identificado por um valor numérico e não categórico. Por exemplo, a estimação pode dizer qual o valor que um consumidor irá gastar após analisar um conjunto de registros que armazena valores mensais gastos por diferentes tipos de consumidores de acordo com o perfil de cada um.

3.1.3.4 Predição

É similar as tarefas de classificação e estimação porém mantém o foco no atributo. Dois exemplos comuns de sua aplicação são o cálculo do percentual de aumento do tráfego de uma rede se a velocidade da mesma aumentar e o valor de uma determinada ação em um certo tempo futuro.

3.1.3.5 Agrupamento ou *Clustering*

O objetivo desta tarefa é identificar e separar registros similares. Um agrupamento ou *cluster* é uma coleção de registros semelhantes, porém diferentes dos registros de outros agrupamentos. Esta tarefa não classifica, estima ou prediz o valor de uma variável, apenas a classifica. A figura 6 ilustra essa prática:

3.1.3.6 Associação

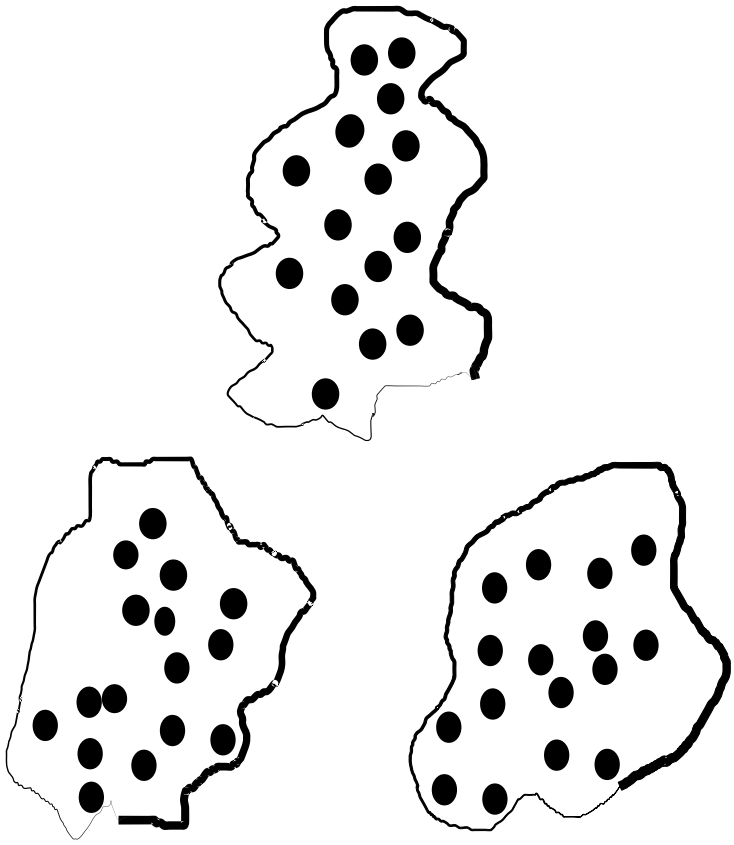
Esta tarefa consiste em analisar a relação entre os atributos. É uma das mais conhecidas devido aos bons resultados obtidos. Ideal para identificar os casos onde, por exemplo, um novo medicamento pode apresentar efeitos colaterais e também identificar usuários de algum plano que respondam bem a oferta de novos produtos.

3.2 Reconhecimento de Padrões

O reconhecimento de padrões é a área de pesquisa que tem por objetivo a classificação de objetos (padrões) em número de categorias ou classes [Theodoridis e Koutroumbas (1999)]. As pesquisas realizadas sobre o tema já apresentam resultados em diversas áreas como: bioinformática, área médica, classificação de imagens, inspeção visual e reconhecimento de voz.

Em sua tese de doutorado Morais (2010) afirma que o projeto de sistemas de reconhecimento de padrões essencialmente envolve três aspectos: aquisição de dados e pré-processamento, representação dos dados e tomada de decisão. O desafio maior está na escolha das técnicas para efetuar esses três aspectos. Alguns padrões podem ser

Figura 6 – Registros agrupados em três clusters.



Fonte: Elaborado pelo Autor

considerados bem estruturados já outros podem ser de difícil identificação e por isso o reconhecimento de padrões pode ser considerado como uma ciência não exata. Ainda segundo Morais (2010), um classificador de padrões é uma função que possui como entrada padrões desconhecidos e, como saída, rótulos que identificam a que classe tais padrões provavelmente pertencem. Para chegar ao seu objetivo um classificador utiliza um conjunto de amostras para treinamento onde são determinadas as fronteiras de decisão do espaço de características. Essas fronteiras de decisão são superfícies multidimensionais dentro de um espaço de características que particionam esse espaço em x regiões para um problema com x classes onde cada região corresponde a uma classe.

Existem vários métodos de classificação diferentes porém pode-se dizer que todos eles possuem os seguintes objetivos:

- minimizar o erro de classificação;
- fazer com que a classificação seja eficiente computacionalmente;

Obviamente, cada método atribui a cada um dos objetivos acima um grau de

importância diferente, porém o ideal é que o método utilizado seja rápido e apurado, mas, em problemas complexos essa situação quase não ocorre.

3.2.1 Algoritmos de Classificação

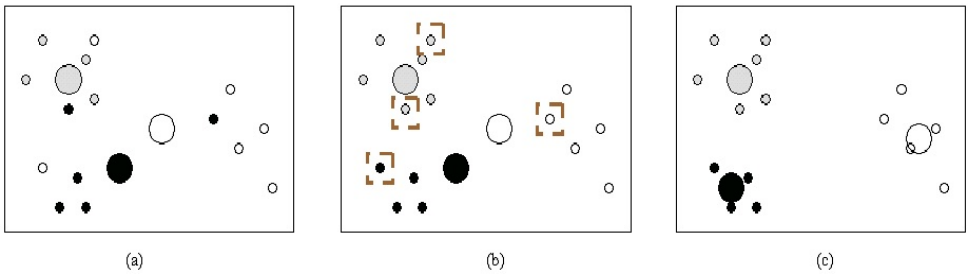
3.2.1.1 *K-Means*

O *K-Means* é um algoritmo de agrupamento que busca minimizar a distância dos elementos a um conjunto de k centros dado por x=$x1$,$x2$,...,xk de forma iterativa. A distância entre um ponto pi e um conjunto de clusters, dada por d(pi,x), é definida como sendo a distância do ponto ao centro mais próximo dele. As etapas do algoritmo estão descritas na sequência abaixo:

1. Escolher k valores distintos para o centro dos grupos
2. Associar cada ponto ao centro mais próximo
3. Recalcular o centro de cada grupo
4. Repetir os passos 2 e 3 até nenhum membro mudar de grupo

Trata-se de um algoritmo extremamente veloz, que geralmente converge com poucas iterações para uma configuração estável. Um exemplo de execução do algoritmo *K-Means* está descrito na figura 7.

Figura 7 – Exemplo de execução do algoritmo *K-Means*.



Fonte: [Linden (2009)]

Na parte (a) da figura 7, cada elemento foi designado para um dos três grupos aleatoriamente e os centróides (círculos maiores) de cada grupo foram calculados. Na parte (b) os elementos foram designados agora para os grupos cujos centróides lhe estão mais próximos e em (c) os centróides foram recalculados e os grupos já estão em sua forma final. Caso não estivessem, repetiríamos os passos (b) e (c) até que estivessem. Um ponto que pode afetar a qualidade dos resultados é a escolha do número de conjuntos realizado pelo usuário. Um número pequeno pode causar a união de dois clusters

parecidos, já um número grande demais pode fazer com um cluster seja dividido em dois.

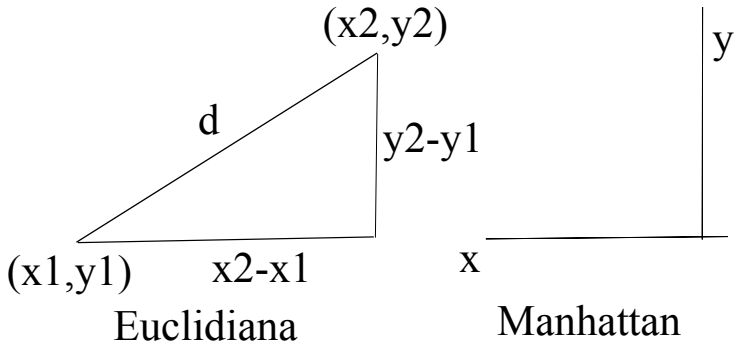
3.2.1.2 Algoritmo K-NN

O K-NN (*K-Nearest Neighbors*) é um dos métodos de classificação existentes mais simples de ser implementado. Para utilizá-lo é necessário um conjunto de treinamento que contenha algumas entradas com suas respectivas classificações. Ele utiliza o método da distância mínima, que atribui cada elemento desconhecido à classe cuja média é a mais próxima, ou seja, todas as distâncias devem ser verificadas até que se ache a menor delas [Hodgson (1988)]. Basicamente, para utilizar o K-NN é necessário:

1. um conjunto de exemplos de treinamento
2. definir uma métrica que será usada para calcular a distância entre os exemplos de treinamento
3. definir o valor de K (número de vizinhos mais próximos que serão considerados para o cálculo da distância)

O método K-NN pode utilizar diferentes métricas para fazer o cálculo da classificação porém as mais utilizadas são a Distância Euclidiana e a Distância Manhattan. A primeira pode ser definida como a distância direta entre os pontos ($x1$,$y1$) e ($x2$,$y2$) já a segunda é o cálculo da soma das diferenças dos atributos x e y. Ambas estão descritas na figura 8 abaixo.

Figura 8 – Distâncias Euclidiana e Manhattan

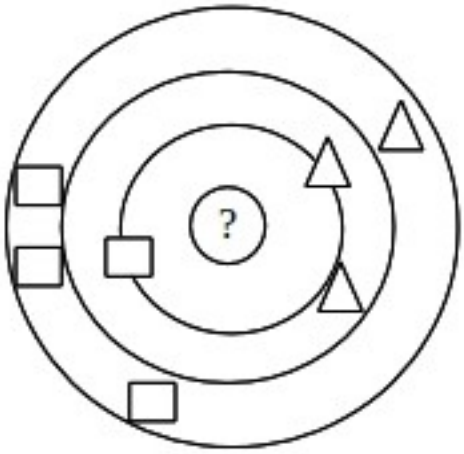


Fonte: Elaborado pelo autor

A figura 9 exemplifica o funcionamento do algoritmo, suponha que deseja-se classificar o círculo entre os registros existentes, para isso deve-se calcular a distância entre esse círculo e cada um dos registros existentes e considerar as k menores distâncias da entrada e a classe das k menores distâncias, depois verifica-se qual a classe com

maior frequência entre os k exemplos mais próximos e assumir a entrada com essa verificação.

Figura 9 – Exemplo de execução do algoritmo K-NN



Fonte: Elaborado pelo autor

A grande vantagem do algoritmo K-NN é que ele cria uma superfície de decisão que se adapta à forma de distribuição dos dados de treinamento de maneira detalhada, possibilitando a obtenção de boas taxas de acerto [Morais (2010)]. Para que isso ocorra é necessário que haja uma grande número de padrões de treinamento cuja classe é conhecida previamente e isso em alguns casos pode se tornar uma enorme desvantagem.

3.3 Redes Neurais

O ressurgimento da área de Redes Neurais Artificiais (RNAs) se deu no final da década de 80 como uma alternativa a computação convencional. Essas redes possuem esse nome porque seu funcionamento relembra a estrutura do cérebro humano e elas não se baseiam em regras ou programas.

RNAs podem ser definidas como "sistemas paralelos distribuídos compostos por unidades de processamento simples (nodos) que calculam determinadas funções matemáticas" [Braga (2005)]. Estas unidades de processamento estão dispostas em camadas e são interligadas por um grande número de conexões que normalmente estão associadas a pesos que armazenam informações e servem para ponderar a entrada recebida por cada neurônio na rede.

O procedimento adotado por uma RNA para solucionar um problema pode possibilitar um desempenho superior a dos modelos convencionais. O processo de resolução de problemas começa por uma fase denominada "aprendizagem" em que um conjunto de exemplos é apresentado a rede e ela adquire a informação fornecida

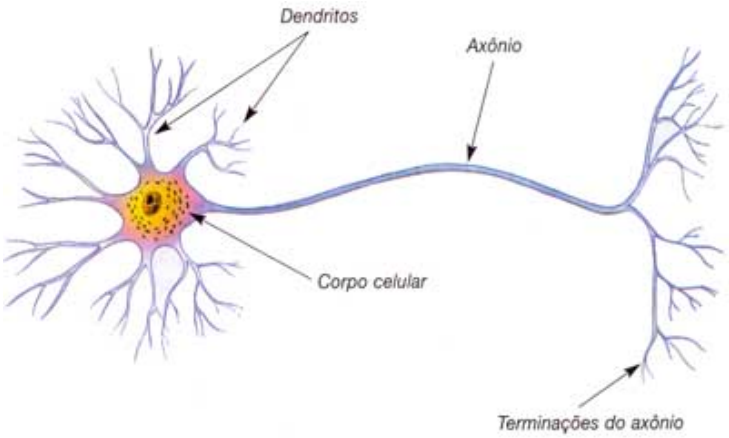
extraindo as características necessárias desse conjunto. Estas informações serão usadas posteriormente para solucionar o problema.

3.3.1 Neurônios Biológicos

O cérebro humano possui cerca de 10^{11} neurônios que processam e trocam informações com milhares de outros continuamente e em paralelo. A estrutura individual destes nodos, sua topologia e comportamento formam o que se chama de base para o estudo das RNAs. Apesar de vários estudos, o cérebro humano ainda não foi totalmente decifrado pelo homem e não se sabe ao certo a forma como as funções cerebrais são realizadas. As emoções, pensamentos, percepções assim como as funções sensoriomotoras e autônomas que são originadas no cérebro ainda não são totalmente conhecidas pelo homem. O que se conhece até o momento são modelos que são atualizados toda vez que novas descobertas são realizadas. Apenas a estrutura fisiológica básica destas redes é conhecida e é nessa estrutura que se baseiam as RNAs.

"Os neurônios são divididos em três seções: o corpo da célula, os dentritos e o axiônio, cada um com funções específicas porém complementares" [Braga (2005)]. Os dentritos são os responsáveis por receber as informações recebidas de outros neurônios e levá-las até o corpo celular onde essas informações são processadas e novas são geradas. A figura 10 ilustra os componentes de um neurônio.

Figura 10 – Modelo de neurônio. As setas indicam o sentido da propagação do impulso nervoso.



Fonte: [Smith (2015)]

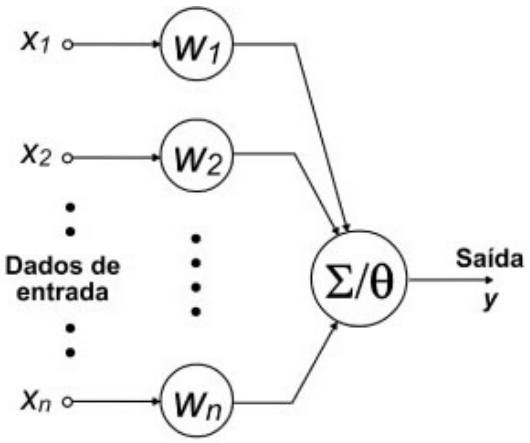
3.3.2 Neurônios Artificiais

Em 1943 o psiquiatra e neuroanatomista Warren McCulloch juntamente com o matemático Walter Pitts criaram um modelo de neurônio artificial obtido através do que se sabia sobre o neurônio biológico na época. Este trabalho ficou conhecido

como modelo MCP[1] [Braga (2005)]. Esse modelo pode ser descrito como um modelo que possui n terminais de entrada (representando os dentritos) e um terminal de saída (que representa o axônio). Para emular o comportamento do neurônio, os terminais de entrada possuem pesos acoplados cujos valores podem ser positivos ou negativos e determinam qual grau o neurônio deve considerar quando receber um sinal de entrada.

"Um neurônio biológico dispara quando a soma dos impulsos que ele recebe ultrapassa o seu limiar de excitação (*threshold*)" [Braga (2005)]. No modelo MCP esta ativação se dá através da aplicação de uma função de ativação que ativa ou não a saída dependendo da soma ponderada dos valores recebidos como entrada. A figura 11 ilustra o esquema de funcionamento de um neurônio MCP de acordo com a proposta apresentada por McCulloch e Pitts.

Figura 11 – Representação de um neurônio artificial.



Fonte: [Fiorin et al. (2011)]

O modelo MCP foi simplificado considerando que os nodos em cada camada da rede são disparados simultaneamente e que são avaliados todos ao mesmo tempo e que isso não acontece em sistemas biológicos pois não existe um mecanismo que sincronize as ações desses nodos.

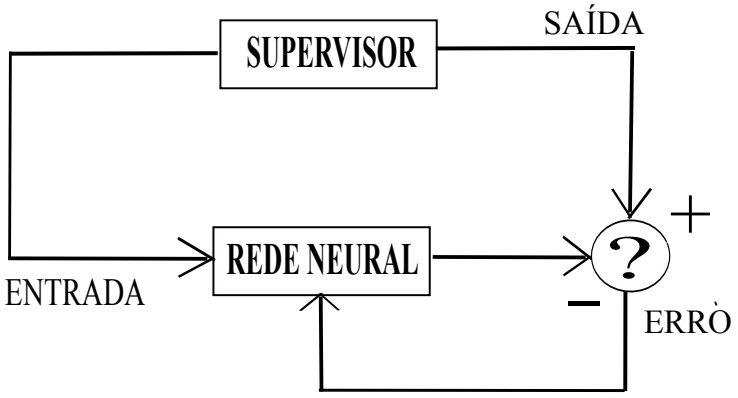
3.3.3 Processo de Aprendizagem

Durante este processo é que são extraídas informações relevantes de padrões de informação apresentados a rede criando assim uma representação própria para o problema. Esses padrões de informação são um conjunto de procedimentos bem definidos que servem para adaptar os padrões da RNA e normalmente são conhecidos como Algoritmos de Aprendizado.

[1] Recebeu esse nome para fazer referência ao nome dos autores

Os tipos de aprendizado de uma rede neural são basicamente divididos em dois grupos: aprendizado supervisionado e aprendizado não supervisionado. O primeiro é o mais comum e tem esse nome porque a entrada e a saída desejadas da rede são fornecidas por um supervisor externo. A cada saída da rede o supervisor compara a saída corrente com a saída desejada e caso não sejam iguais, é feito um ajuste nos pesos para que essa diferença seja minimizada. A figura 12 ilustra o mecanismo de aprendizado supervisionado.

Figura 12 – Mecanismo de aprendizado supervisionado de uma rede neural.

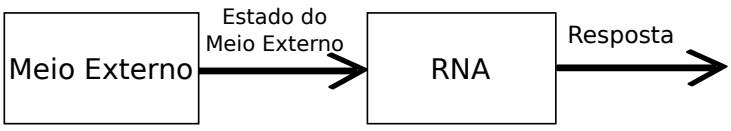


Fonte: Elaborado pelo Autor

O aprendizado não supervisionado é o contrário do anterior, ou seja, não existe um supervisor para fazer a comparação das saídas obtidas pela rede. Este tipo de aprendizado se assemelha muito a alguns tipos de aprendizado desenvolvidos pelos seres humanos como a visão e a audição.

A rede neural é capaz de obter novos padrões, classes e grupos automaticamente a partir do momento em que ela estabelece uma harmonia com as regularidades estatísticas da entrada de dados. A figura 13 ilustra o mecanismo de aprendizado não supervisionado.

Figura 13 – Mecanismo de aprendizado não supervisionado de uma rede neural.



Fonte: Elaborado pelo Autor

3.3.4 Redes Perceptron

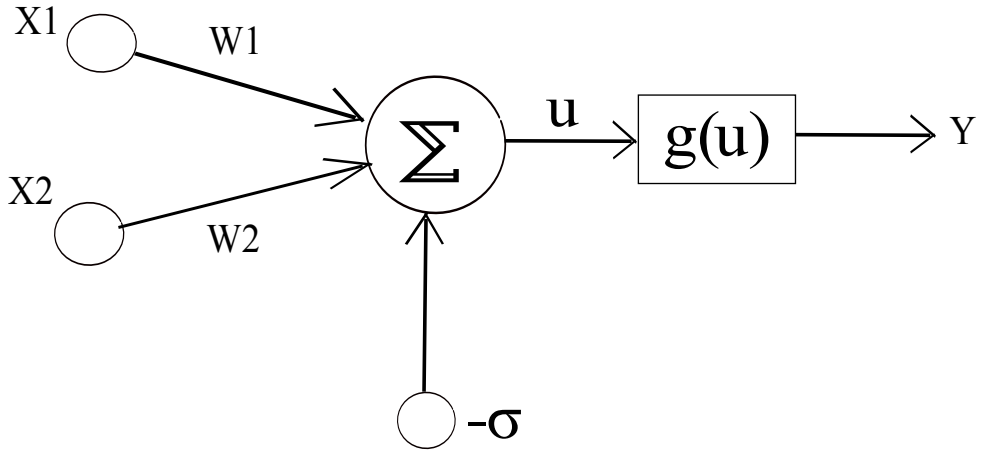
Entre 1950 e 1960 vários pesquisadores propuseram redes neurais contendo modificações do trabalho feito por McCulloch e Pitts. Talvez a modificação que tenha

causado mais impacto na época foi o modelo de aprendizado denominado *perceptron*. Este modelo foi desenvolvido pelo cientista da computação Frank Rosenblatt e teve como base neurônios MCP e uma regra de aprendizado.

"A topologia original do perceptron era composta por unidades de entrada (retina), por um nível intermediário formado pelas unidades de associação e por um nível de saída formado pelas unidades de resposta"[Braga (2005)]. Apesar dessa topologia possuir três níveis ela ficou conhecida como Perceptron de uma única camada pois somente o nível de saída possuía unidades adaptativas.

O perceptron foi desenvolvido para lidar com problemas envolvendo reconhecimento de padrões, tarefas que os seres humanos fazem sem nenhum esforço aparente e de forma quase instantânea, mas que é um dos problemas mais difíceis de serem resolvidos por uma máquina. A figura 14 mostra um esboço da tecnologia do perceptron que funciona da seguinte forma: as entradas Xi representam as informações que são dadas ao algoritmo para que o mesmo possa mapeá-las e cada uma dessas entradas está associada a um peso sináptico Wi que representa a importância de cada entrada em relação ao valor de saída Y desejado. O somatório das entradas ponderadas será somado ao limiar de ativação σ e então repassado como argumento da função de ativação $g(u)$ que terá como resultado a saída desejada. Essa função de ativação normalmente é do tipo degrau ou bipolar.

Figura 14 – Perceptron simples com uma única saída.



Fonte: Elaborado pelo Autor

3.3.5 Perceptron de Múltiplas Camadas

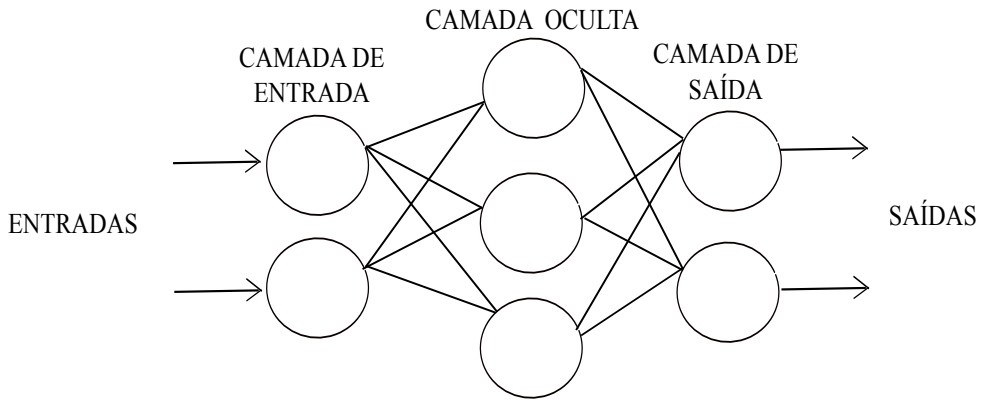
As redes MLP (Multilayer Perceptron) são redes que possuem uma ou mais camadas escondidas. Na década de 70 o desconhecimento sobre como treinar este tipo de rede foi a causa da redução das pesquisas envolvendo RNAs naquela época [Braga (2005)]. Essas redes possuem poder computacional muito maior do que aquelas que

possuem apenas uma camada, pois são capazes de tratar dados que não são linearmente separáveis.

Em uma rede MLP o processamento realizado por cada nodo é definido através da combinação de processamentos feitos pelos nodos da camada anterior que estão conectados a ele. As funções implementadas vão se tornando mais complexas a medida que os dados vão "passando" pelas camadas, pois este tipo de rede possui camadas de neurônios ocultos que não fazem parte da entrada e nem da saída da rede. São estes neurônios que capacitam a rede a aprender tarefas complexas, extraindo as características mais significativas dos padrões de entrada.

Em resumo, pode-se dizer que a camada de neurônios interna funciona como um detector de características de tal forma que se o número dessas camadas for muito grande pode-se formar representações internas para qualquer conjunto de padrões de entrada. A figura 15 ilustra um perceptron com camada oculta:

Figura 15 – Perceptron com uma camada oculta.



Fonte: Elaborado pelo Autor

3.3.6 Treinamento de Redes MLP

Existem diversos algoritmos usados para treinar redes MLP que normalmente usam o tipo de aprendizado supervisionado. Pode-se dizer que o mais conhecido é o algoritmo *back-propagation*. A maioria dos métodos de aprendizagem utiliza variações deste algoritmo que foi um dos principais responsáveis pela volta do interesse das pesquisas envolvendo RNAs na década de 80. Este algoritmo consiste em calcular o erro na saída da rede para depois retropopagá-lo, modificando seus pesos e minimizando o erro da próxima saída.

Um dos problemas enfrentados no treinamento de redes MLP diz respeito a definição de seus parâmetros. Pequenas diferenças nesses parâmetros podem ocasionar grandes diferenças nos tempos de treinamento e generalizações obtidas e por isso

podem ser encontradas na literatura implementações que utilizam o mesmo método de treinamento e possuem tempos de treinamento que diferem muito um do outro.

3.4 Computação Paralela

3.4.1 Introdução

Quando se deseja que uma determinada série de tarefas ou uma tarefa mais complexa seja realizada com maior grau de desempenho, surgem algumas opções básicas como: aumentar o ritmo de trabalho, aumentar a eficiência do trabalho ou pedir ajuda.

Analisando pelo lado da computação, pode-se imaginar essa tarefa complexa como a melhoria de desempenho de um sistema computacional e que para isso poderiam ser utilizadas as seguintes opções: troca do processador por outro mais veloz, o uso de melhores algoritmos e por fim o uso da computação paralela.

A utilização de computadores mais velozes foi prevista por Moore (1965) quando afirmou que a cada dezoito meses os sistemas dobrariam seu desempenho. Todavia, algumas limitações como a grande geração de calor e o consumo elevado de energia dos novos processadores jogou por terra essa idéia e a progressão de desempenho que vinha acontecendo.

A idéia de utilizar algoritmos mais otimizados foi bem aceita durante uma primeira análise mas a dificuldade de implementação desses algoritmos fez com que ela perdesse um pouco de força. O custo elevado de otimização também faz com que algumas empresas deixem de se preocupar um pouco com o desempenho. O desafio maior dos fabricantes tem sido o de aumentar o desempenho dos processadores e ao mesmo tempo fazer com que esses problemas sejam sanados.

O uso da computação paralela ou distribuída pode ser o diferencial na melhoria do desempenho das aplicações se for bem planejada. A utilização de computadores locais ou geograficamente distribuídos pode ser a solução mais adequada para a melhoria de desempenho das aplicações.

3.4.2 Arquiteturas Computacionais

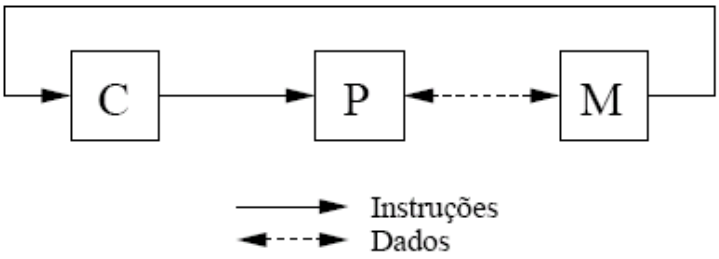
Devido a diversidade de arquiteturas de computadores existentes já foram formuladas inúmeras taxonomias com o intuito de tratar de maneira mais coerente as características dos diferentes sistemas computacionais. A taxonomia de flynn é a classificação mais utilizada e leva em consideração o número de instruções executadas em paralelo em uma máquina *versus* o conjunto de dados sob os quais essas instruções

são submetidas [Flynn e Rudd (1996)]. De acordo com ela as seguintes classificações são estabelecidas:

3.4.2.1 Arquitetura SISD - *Single Instruction Single Data*

Arquiteturas com esta característica executam somente uma instrução por programa por vez, normalmente os computadores pessoais e estações de trabalhos estão inseridos nesta arquitetura. Apesar da forma sequencial que estão organizados os programas, suas instruções podem ser executadas de forma sobreposta em diferentes estágios. A figura 16 ilustra este processo.

Figura 16 – Classe SISD da Arquitetura de Flynn.

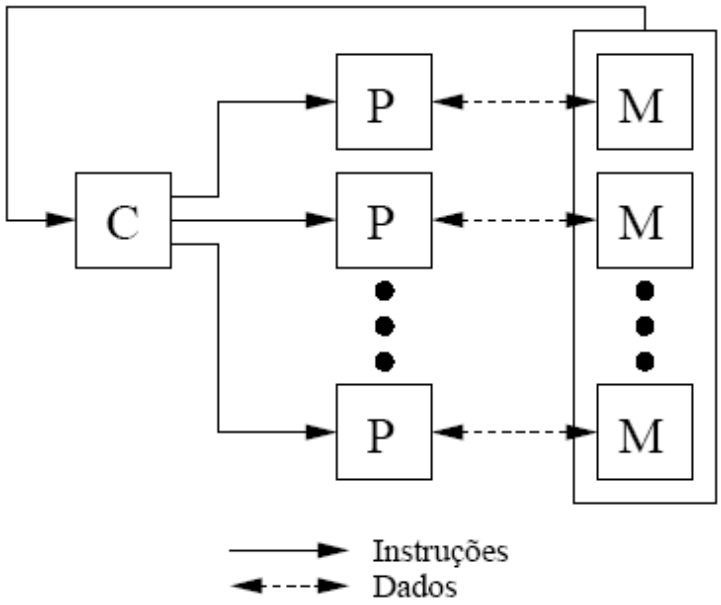


Fonte: [Ramos (2003)]

3.4.2.2 Arquitetura SIMD - *Single Instruction Multiple Data*

Nesta arquitetura vários dados são processados sob o comando de apenas uma instrução. O programa segue uma organização sequencial e para possibilitar o acesso a múltiplos dados é preciso uma organização de memória em diversos módulos. A figura 17 descreve essa classificação.

Figura 17 – Classe SIMD da Arquitetura de Flynn.

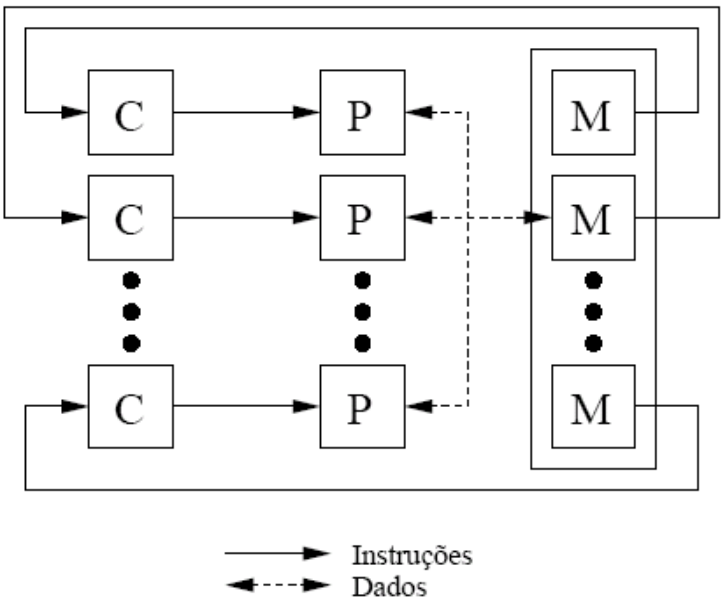


Fonte: [Ramos (2003)]

3.4.2.3 Arquitetura MISD - *Multiple Instruction Single Data*

Neste tipo de arquitetura existem múltiplas unidades de controle executando instruções diferentes operando sob o mesmo dado. Como essa prática não faz parte de nenhum paradigma de programação esta técnica é impraticável tecnologicamente. A figura 18 ilustra este processo.

Figura 18 – Classe MISD da Arquitetura de Flynn.

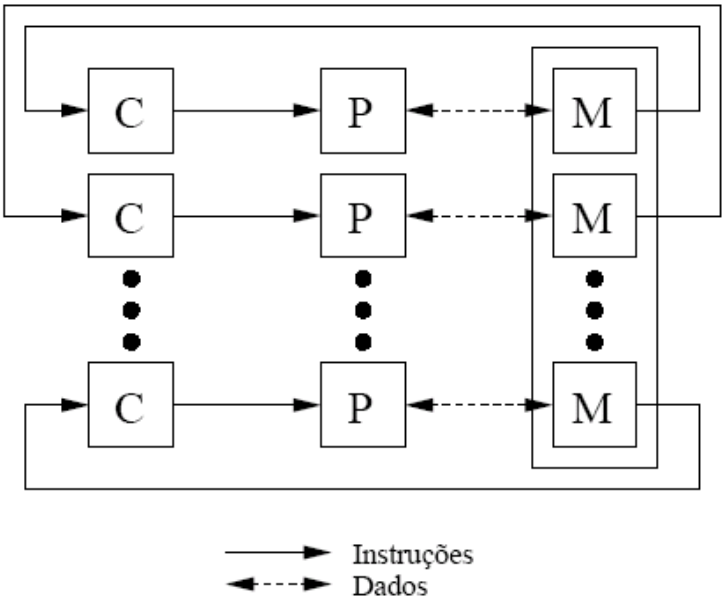


Fonte: [Ramos (2003)]

3.4.2.4 Arquitetura MIMD - *Multiple Instruction Multiple Data*

Esta arquitetura envolve o processamento de múltiplos dados por parte de múltiplas instruções. Várias unidades de controle comandam suas unidades funcionais e estas tem acesso a vários módulos de memória. São parte desta arquitetura os servidores multiprocessados, as redes de estações e as arquiteturas massivamente paralelas. A figura 19 ilustra este processo.

Figura 19 – Classe MIMD da Arquitetura de Flynn.



Fonte: [Ramos (2003)]

3.4.3 Programação Concorrente

3.4.3.1 Processos

"O processamento de alto desempenho (PAD) é um paradigma computacional que tem como um de seus principais objetivos a execução de milhares de aplicações ao mesmo instante e ainda o processamento de tarefas paralelas complexas com um elevado grau de sucesso" [Souza (2012)] .

Pode-se dizer que todos os computadores modernos são capazes de executar várias tarefas ao mesmo tempo e isso não é totalmente percebido por todos os usuários. Um exemplo que deixa bem claro esta característica é o do servidor web que recebe várias solicitações de páginas de vários lugares diferentes durante todo o tempo. Se a página estiver no *cache* o servidor a envia de volta para o usuário se não é enviada uma solicitação de acesso ao disco para buscá-la.

Analisando pelo ponto de vista de uma CPU, pode-se dizer que as solicitações de acesso ao disco duram muito tempo porque quando uma solicitação está em processo

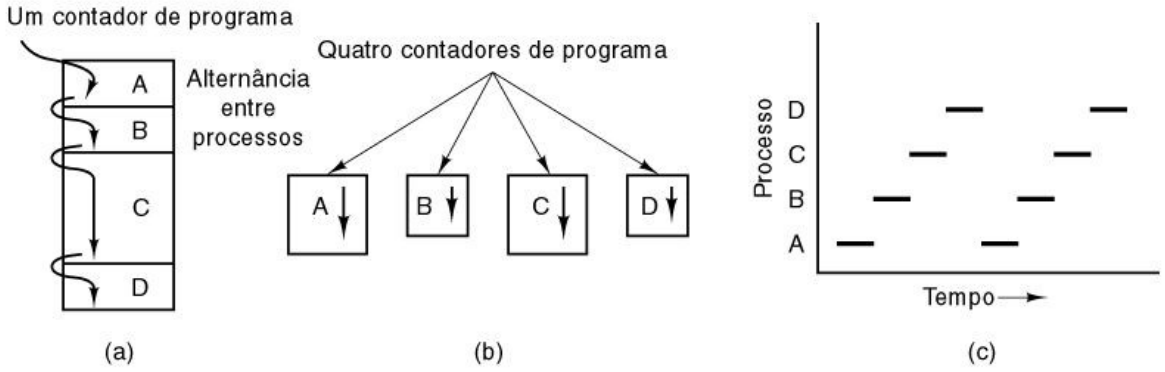
muitas outras podem chegar e ficar aguardando na fila. Se existissem muitos discos, muitas solicitações poderiam ser feitas ao mesmo tempo e isso adiantaria e muito o processo como um todo.

Quando um sistema é iniciado muitos processos desconhecidos do usuário são iniciados em segundo plano e toda esta atividade deve ser administrada e um sistema multiprogramado se torna essencial nesse caso.

Em um sistema multiprogramado a CPU pode, no decorrer de um segundo, trabalhar com diversos programas dando aos usuários a ilusão de paralelismo [TANENBAUM (1999)]. O controle sobre estas atividades é algo extremamente difícil para a maioria dos usuários e por isso projetistas de sistemas operacionais estão se concentrando na construção de um modelo que facilite o paralelismo.

Nesse modelo, todos os programas que podem ser executados pelo computador são organizados em vários processos sequenciais que podem ser chamados apenas de processos. Segundo TANENBAUM (1999) "um processo é apenas um programa em execução, acompanhado dos valores atuais do contador de programa, dos registradores e das variáveis". A figura 20 exemplifica como o computador lida com alguns processos.

Figura 20 – Exemplo de atuação de processos em um computador.



Fonte: [TANENBAUM (1999)]

A parte (a) da figura 20 exemplifica a multiprogramação de quatro programas, a parte (b) mostra um modelo sequencial de quatro processos sequenciais independentes e a parte (c) apresenta um gráfico onde somente um programa está ativo a cada momento.

3.4.3.2 *Threads*

3.4.3.3 Uso de *Threads*

Thread é uma forma que um processo tem de dividir-se a sim mesmo em duas ou mais tarefas que podem ser executadas concorrecialmente. O sistema operacional for-

nece suporte para este método ou o implementa através de alguma biblioteca oferecida por alguma linguagem de programação [Silberschatz et al. (2000)].

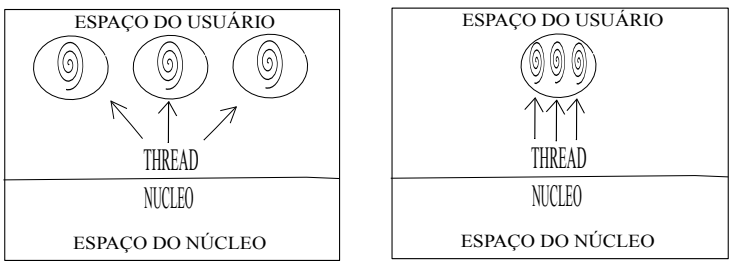
Nos sistemas operacionais tradicionais cada processo tem um espaço de endereçamento e um único *thread* de controle [TANEMBAUM (1999)]. Por muitas vezes torna-se necessário ter muitos *threads* de controle no mesmo espaço de endereçamento executando quase que em paralelo como se fossem processos separados.

Pode-se dizer que os *threads* são miniprocessos que ocorrem dentro de processos e que a principal razão de sua existência é que em muitas aplicações ocorrem múltiplas atividades ao mesmo tempo. A partir deste conceito, pode-se chegar a conclusão de que os *threads* são entidades paralelas que podem compartilhar o mesmo espaço de endereçamento e todos os seus dados entre si.

Uma das maiores vantagens do uso de *threads* é que eles tornam possível o uso de paralelismo sem deixar que os processos sejam sequenciais. Isso mantém a programação teoricamente mais fácil e melhora o desempenho da aplicação. Segundo TANEMBAUM (1999) "Os *threads* também permitem que muitas execuções ocorram no mesmo ambiente do processo, com um grande grau de independência uma da outra".

A figura 21 mostra um pacote de *threads* administrado pelo usuário onde existem três processos com um *thread* cada e um pacote de *threads* administrado pelo núcleo onde existe um processo com três *threads*.

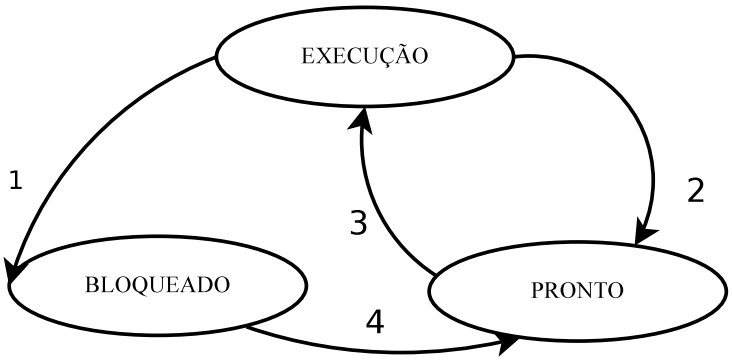
Figura 21 – Interação entre threads e processos.



Fonte: Elaborado pelo Autor

Como nos processos tradicionais (processos com apenas um *thread*), o *thread* pode estar em um de quatro estados: bloqueado, em execução, pronto ou finalizado. Um *thread* bloqueado permanece bloqueado enquanto não houver uma ação que o desbloqueie, um *thread* em execução mantém o controle da CPU e está ativo. Um *thread* pronto está aguardando para executar e se tornará ativo quando chegar a sua vez. A figura 22 ilustra a transição entre os processos de um *thread*.

Figura 22 – Estados de um processo.



Fonte: Elaborado pelo Autor

1. ocorre quando o processo solicita alguma coisa ao sistema operacional como por exemplo: entrada e saída, mais memória, etc.
2. ocorre como consequência de uma interrupção de hardware.
3. ocorre quando um processo que está na fila é escolhido para ocupar a CPU.
4. ocorre quando a solicitação do processo é atendida pela CPU.

Cada *thread* possui sua própria pilha e cada pilha possui uma estrutura para cada rotina chamada que ainda não retornou. Por sua vez, cada estrutura possui as variáveis locais da rotina e o endereço do retorno para usá-lo quando a rotina chamada terminar [TANEMBAUM (1999)]

Quando muitas *threads* são executadas, os processos se iniciam normalmente com um único *thread* que tem a capacidade de criar novos *threads* chamando uma rotina de biblioteca. A rotina *thread_create* por exemplo, cria uma nova rotina para um novo *thread* executar. Existem alguns casos em que os *threads* são hierárquicos, mas normalmente esse processo não acontece e os *threads* são criados todos iguais. Um *thread* é finalizado através da chamada de rotina *thread_exit* que faz ele desaparecer e se torna impossível executá-lo novamente.

Outra rotina também muito utilizada é a *thread_join*, que faz com que um determinado *thread* fique bloqueado até que um outro *thread* específico tenha terminado. Existem várias outras rotinas de biblioteca e algumas das mais importantes estão descritas na tabela 1.

Tabela 1 – Chamadas da função de pthreads

Chamada de Thread	**Descrição**
pthread_create	Cria um novo thread
pthread_exit	Conclui a chamada de thread
pthread_join	Espera que um thread específico seja abandonado
pthread_yeld	Libera a CPU para que outro thread seja executado
pthread_attr_init	Cria e inicializa uma estrutura de atributos do thread
pthread_attr_destroy	Remove uma estrutura de atributos do thread

4 Metodologia Utilizada

Este capítulo descreve os procedimentos metodológicos e os instrumentos utilizados para a realização deste trabalho. A primeira tarefa a ser realizada foi a coleta de amostras do espectômetro. Estas amostras foram coletadas em cargas de sucata descarregadas em uma usina siderúrgica.

O formato de saída dos dados colhidos pelo espectômetro varia conforme a escolha do usuário. Para esta pesquisa foi escolhido o formato de dados xls[1] que é visualizado através de uma planilha eletrônica. Este formato permite uma melhor manipulação dos dados pois os mesmos são divididos em forma tabular.

Após esta etapa, a planilha foi formatada de forma a permitir uma melhor manipulação dos dados. Foram retiradas todas as colunas que exibiam conteúdo que não fosse relevante para a pesquisa e as demais foram renomeadas e rearranjadas conforme a necessidade. Os aços inoxidáveis basicamente são divididos em três classes:

1. 3XX - aços onde predomina uma maior quantidade de níquel;
2. 4XX - aços que contém no mínimo 18% de cromo e não pode haver nenhuma quantidade de níquel ou molibdênio;
3. 316 - aços que possuem no mínimo 2% de molibdênio.

A tabela 2 apresenta a porcentagem desejada de cada elemento químico presente em cada aço pesquisado.

Tabela 2 – Composição química dos aços pesquisados

Tipo de Aço	Ni (%)	Cr (%)	Mo (%)
3xx	8	17	0
4xx	0	18	0
316	10	17	2

Foram retiradas todas as amostras de valor nulo, ou seja, aquelas que não possuíam resultado definido. Também foram geradas aleatoriamente novas amostras a partir das amostras existentes para que os testes envolvendo a performance do algoritmo pudessem ter uma confiança maior. Isso foi feito de acordo com a necessidade de cada problema tratado. A base de dados original continha 880 amostras conforme a tabela 3.

[1] formato de arquivos do Microsoft Excel

Tabela 3 – Composição da base de dados

Qtde 3XX	Qtde 4XX	Qtde 316	Outros	Total
450	107	51	272	880

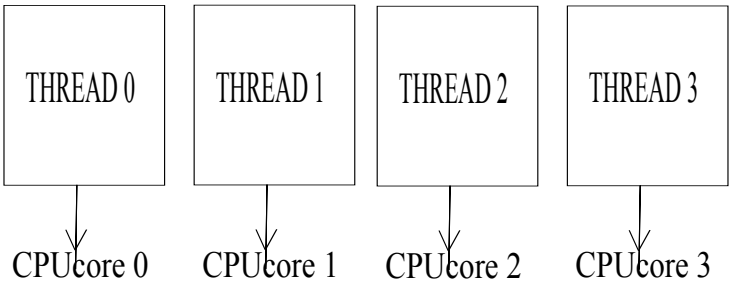
Todos os códigos foram implementados usando a linguagem de programação C++, a IDE[2]Netbeans 8.0.1 e o sistema operacional Ubuntu 14.04 LTS que faz parte da plataforma Linux. Para a implementação da concorrência nos algoritmos foi utilizada a biblioteca de programação Pthread[3]. A tabela 4 descreve as especificações técnicas do processador usado para os testes.

Tabela 4 – Especificações técnicas do processador

Marca	Intel
Número	i5-3337U
Número de núcleos	2
Número de threads	4
Frequência	1,80 Hz
Cache	3 MB SmartCache

Os algoritmos possuem em comum uma fase de cálculos muito extensa, onde normalmente são feitas multiplicações entre matrizes muito grandes que demandam um tempo muito alto de processamento. É justamente nessa fase que foi implementado o paralelismo que funciona da seguinte forma: ao invés dos cálculos serem feitos sequencialmente (linha por linha) eles são feitos paralelamente de acordo com o número de *threads* disponíveis. Cada *thread* fica responsável por uma determinada quantidade de linhas da matriz e a partir daí os cálculos são feitos simultaneamente. Ao término dessa fase os dados são reagrupados novamente formando um único conjunto. A figura 23 ilustra como esse método funciona.

Figura 23 – Método de programação concorrente



Fonte: Elaborado pelo autor

[2] ambiente de desenvolvimento integrado
[3] biblioteca usada para criar e manipular threads

Para cada algoritmo foi dada a opção de escolha do método de processamento desejado: sequencial ou concorrente e no caso do segundo método foi dada a opção também do número de *threads* desejadas para a execução.

4.1 Implementação dos Algoritmos

4.1.0.4 Implementação do Algoritmo K-Means

Este algoritmo possui a fase mais crítica, ou seja, que demanda maior tempo de processamento quando são calculadas as distâncias entre cada amostra e os centróides estabelecidos. Isto é feito após a definição dos centróides iniciais que são definidos aleatoriamente e ocorre da seguinte forma: as amostras da base de treinamento foram alocadas em uma matriz onde cada coluna representa um elemento químico e seu valor representa a posição em cada ponto no plano tridimensional. A partir daí são feitos os cálculos das distâncias onde todas as linhas da matriz são percorridas e são calculadas as distâncias entre os pontos de cada amostra e os pontos dos centróides estabelecidos. A amostra então passa a pertencer ao cluster onde está o centróide com a menor distância encontrada. Após isso é feita a atualização dos centróides calculando-se a distância média de todos os pontos que se ligaram a cada centróide, o que pode fazer com que os centróides mudem de posição. O processo recomeça e o algoritmo opera em *loop* até que nenhum centróide mude mais de posição indicando que já estão na posição central da distância entre os pontos.

A concorrência no algoritmo foi feita utilizando uma função que verifica o tamanho da matriz e divide a quantidade de linhas de acordo com o número de *threads* existentes e a partir daí os cálculos são feitos paralelamente, ou seja, de forma simultânea.

O pseudo-código abaixo exemplifica o funcionamento básico do algoritmo:

Algoritmo 1: Pseudo código do algoritmo K-Means

```
Entrada: base de treinamento
Saída: base de treinamento classificada
inicio
    definição do número de clusters;
    definição do número de threads;
    definição do modo de execução;
    criação dos k centróides;
    atribuição dos centróides aos clusters;
    se modo sequencial então
        repita
            cálculo da distância entre cada amostra da matriz de treinamento
            e os centróides;
            atualização dos centróides;
        até enquanto não houver mais mudanças na posição dos centróides;
    fim
    se modo paralelo então
        verificação do número de threads;
        definição do número de amostras da base de treinamento por thread;
        repita
            cálculo da distância entre cada amostra da matriz de treinamento
            e os centróides;
            atualização dos centróides;
        até enquanto não houver mais mudanças na posição dos centróides;
    fim
fin
```

4.1.0.5 Implementação do Algoritmo K-NN

O algoritmo K-NN é um algoritmo mais simples de ser implementado, porém também possui uma fase muito crítica e que demanda muito do processador que ocorre quando o algoritmo calcula as distâncias entre cada amostra da base de treino com as amostras da base de teste.

As amostras da base de treino também foram alocadas em uma matriz onde cada coluna representa um elemento químico e seu valor representa a posição em cada ponto no eixo tridimensional. A amostras da base de treino também foram manipuladas seguindo este modelo.

O início do algoritmo se dá com a escolha do número de vizinhos desejados

e a partir daí as linhas da matriz da base de treinamento são percorridas e é feito o cálculo da distância entre cada amostra pertencente a base de treino e cada amostra pertencente a base de teste da seguinte forma: calcula-se a distância entre a amostra 1 da matriz de treinamento com a amostra 1 da matriz de teste, em seguida com a amostra 2 e assim sucessivamente até atingir a última amostra da matriz de teste. Logo após reinicia-se o processo tendo como base a amostra 2 da matriz de treinamento e assim sucessivamente até atingir a última amostra da matriz de treinamento. Após todos estes passos o algoritmo verifica quais são as menores distâncias encontradas de acordo com o número de vizinhos escolhidos e classifica a amostra verificando quais tipos de amostras apareceram mais dentro do conjunto encontrado.

A concorrência no algoritmo K-NN foi implementada da mesma forma que foi implementada no algoritmo K-Means utilizando uma função que verifica o tamanho da matriz de treinamento e faz os cálculos de forma simultânea após dividir a quantidade de linhas de acordo com o número de *threads* existentes. O pseudo-código abaixo exemplifica o funcionamento básico do algoritmo:

Algoritmo 2: Pseudo-código do algoritmo K-NN

```
Entrada: base de treinamento, base de teste
Saída: base de treinamento classificada
inicio
    definição do número de classes;
    definição do número de threads;
    definição do modo de execução;
    definição da quantidade k de vizinhos;
    se modo sequencial então
        cálculo da distância entre cada amostra da matriz de treinamento e
        cada amostra da matriz de teste;
        definição dos k vizinhos mais próximos;
    fim
    se modo paralelo então
        verificação do número de threads;
        definição do número de amostras da base de treinamento por thread;
        cálculo da distância entre cada amostra da matriz de treinamento e
        cada amostra da matriz de teste;
        definição dos k vizinhos mais próximos;
    fim
    classificação das amostras da matriz de treinamento de acordo com a
    classe dos vizinhos encontrados;
fin
```

4.1.0.6 Implementação do Algoritmo Perceptron de Múltiplas Camadas

Como o próprio nome já diz o Perceptron de Múltiplas Camadas é um algoritmo que trabalha através de camadas: a camada de entrada, a camada oculta e a camada de saída. A camada de entrada recebe as informações que serão tratadas e a transmite para a camada oculta que faz o seu trabalho e retransmite a informação já tratada para a camada de saída que se encarrega de retornar a resposta encontrada pelo algoritmo.

Para a execução do Perceptron é necessário que sejam pré-definidos alguns parâmetros como a quantidade de atributos, o número de neurônios na camada oculta e o número de saídas do algoritmo. Todos esses valores devem ser informados antes da execução do algoritmo.

A primeira etapa da execução do algoritmo é a geração dos pesos que é feita de forma aleatória. Logo após são criadas as camadas que são implementadas através de matrizes. A fase mais crítica do algoritmo ocorre durante a fase de treinamento que é quando os dados são transferidos entre as camadas. Isso acontece da seguinte forma: a matriz que contém as amostras para o treinamento é percorrida linha por linha e cada amostra é apresentada para a camada oculta para que sejam feitos os cálculos e o ajuste dos pesos. Toda a etapa de treinamento foi implementada através de uma função. A eficiência do algoritmo é medida através de épocas, cada época corresponde a uma iteração completa do código e o número de épocas também deve ser definido antes do início da execução.

A implementação da concorrência no Perceptron foi feita da mesma forma que foi feita nos algoritmos anteriores, a matriz de treinamento foi dividida tendo como base o número de *threads* disponíveis e a função de cada *thread* foi de executar o treinamento fazendo com que ele fosse executado simultaneamente com diversas linhas da matriz ao mesmo tempo. Abaixo segue um pseudo código que exemplifica o funcionamento

básico do algoritmo:

Algoritmo 3: Pseudo-código do algoritmo Perceptron de Múltiplas Camadas

```
Entrada: base de treinamento, base de teste
Saída: base de treinamento classificada
inicio
    definição do número de neurônios na camada oculta;
    definição da quantidade de épocas de treinamento;
    definição do número de threads;
    definição do modo de execução;
    inicialização dos pesos;
    se modo sequencial então
        repita
            propagação das amostras pelas camadas intermediárias;
            ajuste dos pesos;
            classificação das amostras;
        até até ser alcançada a quantidade de épocas;
    fim
    se modo paralelo então
        definição do número de amostras da base de treinamento por thread;
        repita
            propagação das amostras pelas camadas intermediárias;
            ajuste dos pesos;
            classificação das amostras;
        até até ser alcançada a quantidade de épocas;
    fim
fin
```

5 Análise de Resultados

5.1 Execução e Testes

Este trabalho teve como objetivo fazer uma análise do uso de três algoritmos da área de inteligência artificial e reconhecimento de padrões para classificar determinados padrões de sucata adquirida de terceiros dentro de uma usina siderúrgica e tentar fazer com que tenham a melhor performance possível no sentido de otimizar os recursos computacionais através do uso da programação concorrente.

Como um dos objetivos do trabalho foi medir a performance dos algoritmos implementados, a base de dados teve que ser aumentada para permitir uma melhor análise dos resultados. A geração das novas amostras foi feita da seguinte forma:

1. foi calculado a quantidade de amostras pertencente a cada tipo de aço;
2. a quantidade gerada de cada tipo de amostra foi proporcional a quantidade de cada tipo de aço;
3. o valor das novas amostras foi gerado utilizando uma função na planilha eletrônica que somava o número -0,01 ou 0,01 a porcentagem de algum elemento químico pertencente a amostra de acordo com o seu tipo para que não mudasse a sua classificação;
4. a quantidade total de amostras passou de 880 para 823640.

Para cada método de processamento foram realizadas dez execuções e calculadas as médias do tempo de processamento de cada execução. Os tópicos abaixo descrevem os resultados obtidos para cada algoritmo implementado com relação a sua eficiência (capacidade de resolução do problema) e também em relação a sua performance (otimização do uso dos recursos computacionais).

5.1.1 Algoritmo K-Means

A primeira etapa para a realização dos testes com esse algoritmo foi a escolha do número de *clusters* desejados que nesse caso foi igual a três, um para cada tipo de aço procurado. A base de dados utilizada está descrita na tabela 5 e os valores encontrados para os *clusters* após os testes estão descritos na tabela 6.

Tabela 5 – Base de treinamento do algoritmo K-Means

Tipo de Aço	Quantidade de Amostras
3xx	609493
4xx	148255
316	65442
Total	823640

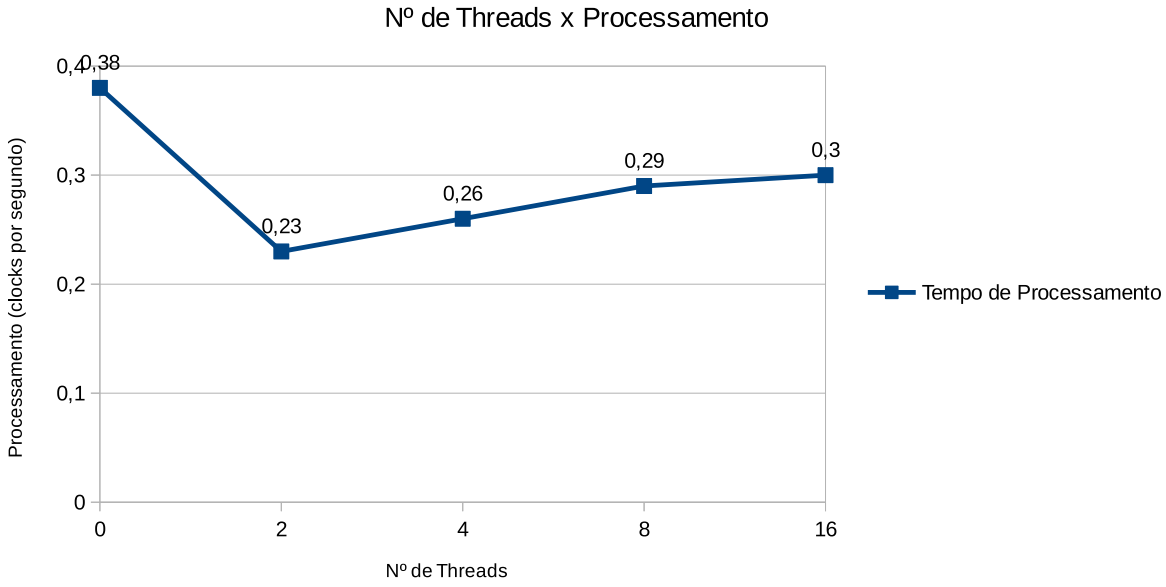
Tabela 6 – Posição central dos clusters encontrados pelo algoritmo

Cluster	Posição x (Ni)	Posição y (Cr)	Posição z (Mo)
1	0	14.04	0
2	0	0.99	0.35
3	0	0.52	0

Analisando a tabela percebe-se que os *clusters* foram divididos levando em consideração o atributo de maior quantidade presente, o que não é o ideal para o problema tratado. O ideal é que fossem divididos tomando como base a distribuição uniforme dos atributos.

A figura 24 faz a comparação da performance do algoritmo utilizando o modelo sequencial e também concorrente:

Figura 24 – Gráfico da performance do algoritmo K-Means



Fonte: Elaborado pelo autor

Através do gráfico percebe-se que quanto mais *threads* são utilizadas maior é o tempo gasto com processamento, uma possível explicação para isso pode ser o tempo de criação de cada *thread* que supera o ganho na redução do tempo de processamento. Como esse algoritmo não possui muitas iterações entre matrizes, mesmo com a grande quantidade de amostras presente na base de dados não foi possível realizar um teste que pudesse certificar essa informação. De qualquer forma, o modelo de programação concorrente oferece um ganho considerável de desempenho em relação ao modelo sequencial.

5.1.2 Algoritmo K-NN

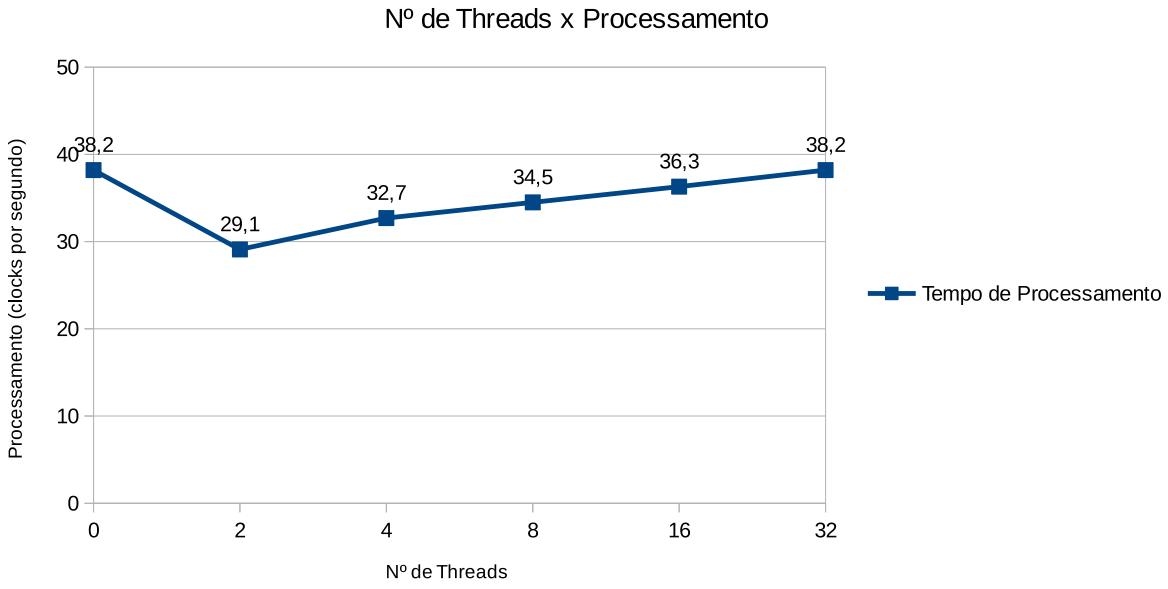
Para os testes com este algoritmo o valor de K foi definido como 3, ou seja, foram considerados os três vizinhos mais próximos de cada amostra para definição do resultado. A base de dados do algoritmo foi criada conforme a tabela 7 abaixo:

Tabela 7 – Base de dados do algoritmo K-NN

Treinamento	Teste	Total
822588	1052	823640

Após os testes o algoritmo chegou a um resultado de 494184 acertos, ou seja 60% do total da base de treinamento. A figura 25 faz a comparação entre o modelo sequencial e concorrente implementados com o algoritmo:

Figura 25 – Gráfico da performance do algoritmo K-NN



Fonte: Elaborado pelo autor

A quantidade de amostras usadas com o algoritmo K-NN foi a mesma quantidade usada com algoritmo K-Means e como o K-NN possui mais iterações entre matrizes seu teste de performance pôde ser melhor avaliado. O que se percebe através do gráfico é que até o uso de 32 *threads* o custo de processamento no modelo concorrente é melhor que o custo de processamento no modelo sequencial.

Novamente o desempenho piora quando se aumenta o número de *threads* e a razão para isso pode ser a mesma descrita anteriormente.

5.1.3 Algoritmo Perceptron de Múltiplas Camadas

A base de dados para o algoritmo Perceptron teve que ser preparada adicionando-se mais três colunas em cada linha, ou seja, cada amostra passou a contar com três colunas a mais. Isto foi feito da seguinte forma: se a amostra é da classe 3XX as três colunas finais passaram a ser 1 0 0 e assim por diante conforme a tabela 8 abaixo:

Tabela 8 – Tabela de respostas para a base de dados do algoritmo perceptron de múltiplas camadas

Tipo de Aço	Coluna 1	Coluna 2	Coluna 3
3XX	1	0	0
4XX	0	1	0
316	0	0	1

Normalmente se usam duas bases de dados para os testes com o algoritmo Perceptron, uma base de treino e uma base de teste. Após o treinamento do algoritmo com a base de treino o mesmo executa as mesmas funções com a base de teste para poder consolidar os resultados.

As tabelas 9 e 10 descrevem as bases de dados usadas para os cálculos do algoritmo:

Tabela 9 – Base de treinamento do algoritmo perceptron de múltiplas camadas

Tipo de Aço	Quantidade de Amostras
3xx	4951
4xx	1177
316	560
Total	6688

Tabela 10 – Base de teste do algoritmo perceptron de múltiplas camadas

Tipo de Aço	Quantidade de Amostras
3xx	1485
4xx	353
316	168
Total	2006

O resultado final deste algoritmo é obtido através de épocas, ou seja, a cada vez que o algoritmo faz um treinamento com a base de dados conta-se uma época, e é através do número de épocas que pode-se obter um resultado pretendido. As tabelas 11 e 12 ilustram os resultados de eficiência obtidos com o algoritmo:

Tabela 11 – Tabela de eficiência do algoritmo perceptron de múltiplas camadas - base de treino

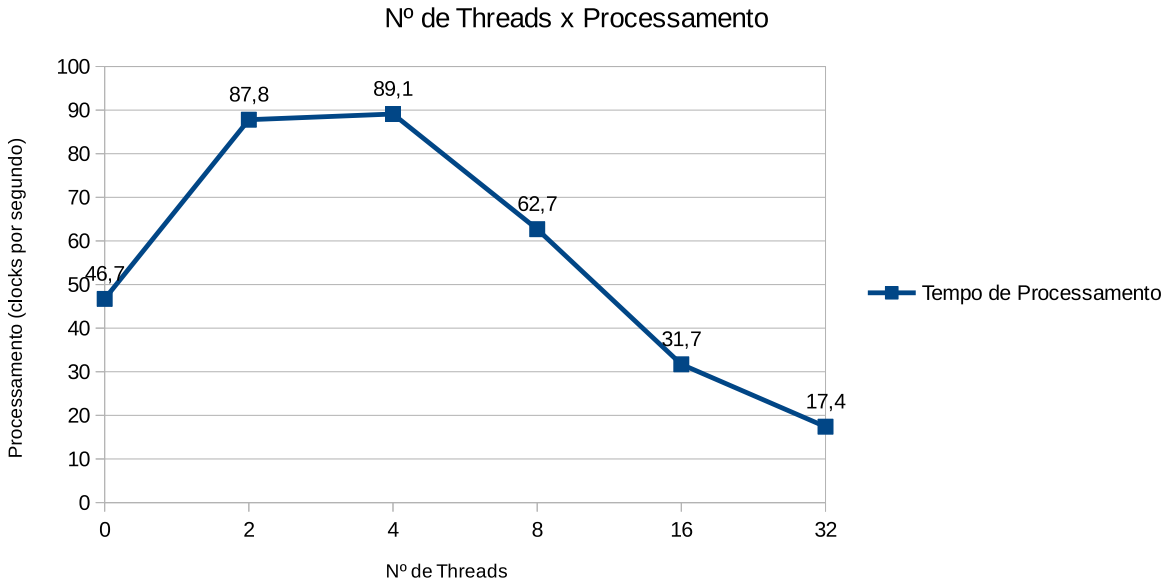
Base de Treino			
Nº de Épocas	Acertos	Erros	% Acertos
1000	5952	736	% 89,4
5000	6208	480	% 92,9
10000	6332	356	% 94,7

Tabela 12 – Tabela de eficiência do algoritmo perceptron de múltiplas camadas - base de teste

Base de Teste			
Nº de Épocas	Acertos	Erros	% Acertos
1000	1843	163	% 91,8
5000	1844	162	% 91,9
10000	1845	161	% 92,0

Através da tabela percebe-se que com 10000 épocas a taxa de acertos do algoritmo fica em torno de 95%, o que pode ser considerado um ótimo resultado em função da quantidade de amostras utilizadas tanto na base de treino como na base de teste. O resultado da performance do algoritmo está ilustrado na figura 26:

Figura 26 – Gráfico da performance do algoritmo perceptron



Fonte: Elaborado pelo autor

Analisando a figura, percebe-se que o rendimento do algoritmo no modelo concorrente começa a evoluir quando o mesmo é executado com 8 *threads* e supera o do modelo sequencial a partir de 16 *treads*. Uma possível explicação para esse fato pode ser que devido ao tamanho da base de treino ser muito grande e que quando o algoritmo trabalha com menos de 16 *treads* elas são obrigadas a trabalhar com uma quantidade de linhas que faz com que seu tempo de criação e execução se torne superior ao tempo no modelo sequencial.

6 Conclusão

Este trabalho teve como objetivo a comparação entre soluções diferentes e melhoria em suas performances através da implementação de alguns algoritmos para tentar detectar padrões de recebimento e armazenamento de sucata metálica em uma usina siderúrgica. Para isso, foram implementados três algoritmos clássicos das áreas de reconhecimento de padrões e inteligência artificial para verificar se os mesmos seriam capazes de resolver o problema. Após a análise foram usadas técnicas de programação concorrente para tentar melhorar a performance desses algoritmos.

Os resultados obtidos em relação a performance dos algoritmos mostram que os métodos de programação concorrente podem ser bem vantajosos se empregados em aplicações que utilizam grandes quantidades de dados e necessitam aproveitar ao máximo os recursos computacionais, porém deve ser explorado o fato de que o desempenho dos algoritmos piora quando se aumenta o número de *threads*.

Com relação a eficiência, o algoritmo K-Means demonstrou não ser aplicável para o tipo de problema tratado, mas podem ser feitos novos estudos para avaliar sua aplicação em outros problemas relacionados a indústria siderúrgica. O algoritmo K-NN também não demonstrou uma eficiência satisfatória pois conseguiu uma taxa de acertos de apenas 60%. Por fim, pode-se considerar que o Perceptron de Múltiplas Camadas seja o mais indicado dos três pois conseguiu acertar mais de 90% dos dados pesquisados.

O trabalho proposto é de fundamental importância não só no que diz respeito a aprendizagem dos fundamentos de computação concorrente como também gerou muito conhecimento nas áreas de reconhecimento de padrões e inteligência artificial. Ele também pode servir como base de estudos que a indústria possa tentar melhorar sua atuação nessa área.

6.1 Trabalhos futuros

Para fins de aprimoramento dos resultados, pode ser feito um estudo para tentar avaliar o real motivo pelo qual os algoritmos não apresentaram melhora de rendimento quando se aumenta o número de *threads* utilizadas e esses algoritmos podem ser implementados usando a linguagem de programação paralela Cuda, que permite que o processamento possa ser melhorado utilizando os recursos disponíveis para processamento gráfico (GPU).

O algoritmo Perceptron foi implementado para detectar padrões de apenas três tipos de aços, isso também pode ser melhorado fazendo com que o algoritmo possa

tentar detectar os padrões de todos os aços comercializados pela indústria para garantir um melhor gerenciamento da sucata adquirida.

Por fim, esse trabalho pode servir de base para a implementação de um *software* de gerenciamento industrial que poderá garantir a qualidade de toda a sucata adquirida pela empresa desde o momento de sua compra até sua utilização.

Referências

ALVARO; FERREIRA, J. J.; LUISA; SILVA, M. V. da. Análise de Cluster Aplicada a Logística: Definição de Zonas de Transporte Para Uma Empresa do Setor Siderúrgico. 2008. Citado na página 7.

ANDERSON, T.; SHEIN, J. **Hand-Held XRF Analysers Advance PMI Programs**. [S.l.], 2009. Citado 3 vezes nas páginas 1, 2 e 3.

ARAÚJO, L. A. de. **Manual de siderurgia**. [S.l.]: Arte & Ciência, 2005. Citado 2 vezes nas páginas 3 e 4.

BRAGA, A. d. P. **Redes neurais artificiais: teoria e aplicações**. [S.l.: s.n.], 2005. Citado 4 vezes nas páginas 17, 18, 19 e 21.

CAMILO, C. O.; SILVA, J. a. C. da. Mineração de Dados: Conceitos, Tarefas, Métodos e Ferramentas. 2009. Citado 3 vezes nas páginas 9, 10 e 12.

DANTAS, E. R. G.; ALMEIDA, J. C.; JÚNIOR, P.; LIMA, D. S. de; PESSOA-UNIPÊ, J. O uso da descoberta de conhecimento em base de dados para apoiar a tomada de decisões. **V Simpósio de Excelência em Gestão e Tecnologia-SEGeT**, v. 1, p. 50–60, 2008. Citado na página 4.

FIORIN, D. V.; MARTINS, F. R.; SCHUCH, N. J.; PEREIRA, E. B. Aplicaçoes de redes neurais e previsoes de disponibilidade de recursos energéticos solares. **Revista Brasileira de Ensino de Fısica**, SciELO Brasil, v. 33, n. 1, p. 1309, 2011. Citado na página 19.

FLYNN, M. J.; RUDD, K. W. Parallel architectures. **ACM Computing Surveys (CSUR)**, ACM, v. 28, n. 1, p. 67–70, 1996. Citado na página 24.

GOLDSCHMIDT, R.; PASSOS, E. **Data Mining: um guia prático**. [S.l.]: Gulf Professional Publishing, 2005. Citado na página 4.

HODGSON, M. E. Reducing the computational requirements of the minimum-distance classifier. **Remote Sensing of Environment**, Elsevier, v. 25, n. 1, p. 117–128, 1988. Citado na página 16.

LINDEN, R. Técnicas de Agrupamento. In: ____. 4. ed. [S.l.: s.n.], 2009. p. 18–36. Citado na página 15.

MARTINS, L. A. O.; PáDUA, F. L. C.; ALMEIDA, P. E. M. de. Aplicação de Redes Som e Técnicas Não Paramétricas para Inspeção Visual Automática de Defeitos em Aços Laminados. 2009. Citado na página 7.

MOORE, G. Moore's law. **Electronics Magazine**, v. 38, n. 8, 1965. Citado na página 23.

MORAIS, E. C. **Reconhecimento de padrões e redes neurais artificiais em predição de estruturas secundárias de proteínas**. Tese (Doutorado) — Universidade Federal do Rio de Janeiro, 2010. Citado 3 vezes nas páginas 13, 14 e 17.

RAIZER, K.; IDAGAWA, H. S.; NóBREGA, E. G. de O.; FERREIRA, L. O. S. Training and applying a feedforward multilayer neural network in gpu. 2009. Citado na página 8.

RAMOS, J. P. S. Redes neurais artificiais na classificação de frutos: Ce-nário bidimensional. SciELO Brasil, 2003. Citado 3 vezes nas páginas 24, 25 e 26.

SIERRA-CANTO, X.; MADERA-RAMIREZ, F.; UC-CETINA, V. Parallel training of a back-propagation neural network using cuda. In: IEEE. **Machine Learning and Applications (ICMLA), 2010 Ninth International Conference on**. [S.l.], 2010. p. 307–312. Citado na página 8.

SILBERSCHATZ, A.; GALVIN, P. B.; GAGNE, G. **Fundamentos de Sistemas: Operacionais. Princípios Básicos**. [S.l.]: Grupo Gen-LTC, 2000. Citado na página 28.

SMITH, J. A. **Sistema Nervoso: Origem e Divisão do SN**. 2015. Http://www.sogab.com.br/anatomia/sistemanervosojonas.htm. Citado na página 18.

SOUZA, C. L. de. **Recuperação de Vídeos Baseada em Conteúdo em um Sistema de Informação para Apoio à Análise do Discurso Televisivo**. Dissertação (Mestrado) — Centro Federal de Educação tecnológica de Minas Gerais, 2012. Citado na página 26.

SOUZA, S. A. de. **Composição química dos aços**. [S.l.]: Edgard Blücher, 2006. Citado na página 1.

TANEMBAUM, A. S. **Sistemas Operacionais Modernos. São Paulo, 2003**. 1999. Citado 3 vezes nas páginas 27, 28 e 29.

THEODORIDIS, S.; KOUTROUMBAS, K. Pattern recognition: Elsevier academic press. 1999. Citado na página 13.

Apêndices

APÊNDICE A – Código Fonte do Algoritmo K-Means

```
1  //declaracao das bibliotecas
2  #include <cstdlib>
3  #include "stdio.h"
4  #include <stdlib.h>
5  #include <iostream>
6  #include <sys/types.h>
7  #include <assert.h>
8  #include "kMeans.h"
9  #include <pthread.h>
10
11 //parametros utilizados
12 #define numThreads 4
13 #define numClusters 3
14 #define PAR_1_SEQ_0 1
15
16 using namespace std;
17
18 //variaveis globais
19 float **dados; //matriz que armazena os dados de entrada[numDados][
       numAtr]
20 float **clusters; //matriz que armazena os centroides de cada dado[
       numClusters][numAtr]
21 int numAtr, numDados;
22 int *resultado; //vetor que armazena o cluster a que pertence a
       amostra
23 float threshold = 0.001; //valor usado como criterio de parada
24 static int aux[10000000];
25
26 //struct que contem os dados das threads
27
28 typedef struct {
29     int idThread, size, indice, index;
30 } t_Args;
31
32 //funcao de normalizacao
33
34 void normalizaDados(float **base, int tamMatriz) {
35
```

```
36      float maior[numAtr];
37      float menor[numAtr];
38
39      // Encontra maior e menor valor de cada coluna
40      for (int i = 0; i < numAtr; i++) {
41          for (int j = 0; j < tamMatriz; j++) {
42              if (j == 0) {
43                  maior[i] = base[j][i];
44                  menor[i] = base[j][i];
45              } else {
46                  if (base[j][i] > maior[i]) {
47                      maior[i] = base[j][i];
48                  }
49                  if (base[j][i] < menor[i]) {
50                      menor[i] = base[j][i];
51                  }
52              }
53          }
54      }
55
56      for (int i = 0; i < tamMatriz; i++) {
57          for (int j = 0; j < numAtr; j++) {
58              base[i][j] = (base[i][j] - menor[j])
59                      / (maior[j] - menor[j]);
60
61          }
62
63      }
64
65  }
66
67  //funcao que calcula a distancia euclidiana
68
69  float euclid_dist_2(int numdims, /* no. dimensoes */
70          float *coord1, /* [numdims] */
71          float *coord2) /* [numdims] */ {
72      int i;
73      float ans = 0.0;
74
75      for (i = 0; i < numdims; i++)
76          ans += (coord1[i] - coord2[i]) * (coord1[i] - coord2[i]);
77
```

```
78      return (ans);
79  }
80
81  //procura o cluster mais proximo
82
83  int find_nearest_cluster(
84          int numCoords, /* no. coordinates */
85          float *object, /* [numCoords] */
86          float **clusters) /* [numClusters][numCoords] */ {
87      int index, i;
88      float dist, min_dist;
89
90      //procura o cluster que cont m a dist ncia  m nima do objeto
91      index = 0;
92      min_dist = euclid_dist_2(numCoords, object, clusters[0]);
93
94      for (i = 1; i < numClusters; i++) {
95          dist = euclid_dist_2(numCoords, object, clusters[i]);
96
97          if (dist < min_dist) {
98              min_dist = dist;
99              index = i;
100         }
101     }
102     return (index);
103 }
104
105 //funcao que dispara a thread
106
107 void *criaThread(void *params) {
108
109     t_Args *arg = (t_Args *) params; //receber  os argumentos para a
            thread
110     int elemPorThread = arg->size / numThreads;
111     int pri = arg->idThread * elemPorThread;
112     int ult = pri + elemPorThread;
113
114     for (int k = pri; k < ult; k++) {
115         arg->indice = k;
116         arg->index = find_nearest_cluster(numAtr, dados[k], clusters)
               ;
117         aux[arg->indice] = arg->index;
```

```
118
119     }
120     return 0;
121 }
122
123 //funcao que faz processa o algoritmo de acordo com a  o p   o
       escolhida
124
125 void processa(int opcao) {
126
127     int i, j, index, loop = 0;
128     int *newClusterSize;
129     float delta;
130     float **newClusters;
131     pthread_t threads[numThreads];
132     t_Args *arg; //receber  os argumentos para a thread
133
134     clusters = (float**) malloc(numClusters * sizeof (float*));
135     assert(clusters != NULL);
136     clusters[0] = (float*) malloc(numClusters * numAtr * sizeof (
           float));
137     assert(clusters[0] != NULL);
138     for (i = 1; i < numClusters; i++)
139         clusters[i] = clusters[i - 1] + numAtr;
140
141     for (i = 0; i < numClusters; i++)
142         for (j = 0; j < numAtr; j++)
143             clusters[i][j] = dados[i][j];
144
145     for (i = 0; i < numDados; i++) resultado[i] = -1;
146
147     newClusterSize = (int*) calloc(numClusters, sizeof (int));
148     assert(newClusterSize != NULL);
149
150     newClusters = (float**) malloc(numClusters * sizeof (float*));
151     assert(newClusters != NULL);
152     newClusters[0] = (float*) calloc(numClusters * numAtr, sizeof (
           float));
153     assert(newClusters[0] != NULL);
154     for (i = 1; i < numClusters; i++)
155         newClusters[i] = newClusters[i - 1] + numAtr;
156
```

```
157     if (opcao)cout << "Executando algoritmo paralelo" << endl;
158     else cout << "Executando algoritmo sequencial." << endl;
159 
160     do {
161         delta = 0.0;
162 
163         if (opcao) {
164 
165             //inicia o contador para o tempo de processamento
166             //tStart = clock();
167             for (int t = 0; t < numThreads; t++) {
168 
169                 arg = (t_Args*) malloc(sizeof (t_Args));
170                 assert(arg != NULL);
171                 arg->idThread = t;
172                 arg->size = numDados;
173 
174                 if (pthread_create(&threads[t], NULL, criaThread, (
                        void*) arg)) {
175                     printf("--ERRO: pthread_create()\n");
176                     exit(1);
177                 }
178 
179             }
180 
181             ///--espera todas as threads terminarem
182             for (int t = 0; t < numThreads; t++) {
183                 if (pthread_join(threads[t], NULL)) {
184                     cout << "--ERRO: pthread_join() \n";
185                     exit(-1);
186                 }
187 
188             }
189 
190             for (int k = 0; k < numDados; k++) {
191                 if (resultado[k] != aux[k]) {
192                     delta += 1.0;
193 
194                 }
195 
196                 resultado[k] = aux[k];
197 
```

```
198
199                 newClusterSize[aux[k]]++;
200                 for (j = 0; j < numAtr; j++)
201                     newClusters[aux[k]][j] += dados[k][j];
202             }
203
204         } else {
205
206             for (int i = 0; i < numDados; i++) {
207
208                 aux[i] = find_nearest_cluster(numAtr, dados[i],
                       clusters);
209             }
210
211             for (int k = 0; k < numDados; k++) {
212                 if (resultado[k] != aux[k]) {
213                     delta += 1.0;
214
215                 }
216
217                 resultado[k] = aux[k];
218
219
220                 newClusterSize[aux[k]]++;
221                 for (j = 0; j < numAtr; j++)
222                     newClusters[aux[k]][j] += dados[k][j];
223
224             }
225
226
227             for (int i = 0; i < numClusters; i++) {
228                 for (int j = 0; j < numAtr; j++) {
229                     if (newClusterSize[i] > 0)
230                         clusters[i][j] = newClusters[i][j] /
                               newClusterSize[i];
231                     newClusters[i][j] = 0.0; /* set back to 0 */
232                 }
233                 newClusterSize[i] = 0; /* set back to 0 */
234             }
235
236             delta /= numDados;
237         }
```

```
238     } while (delta > threshold && loop++ < 500);
239
240
241
242     free(newClusters[0]);
243     free(newClusters);
244     free(newClusterSize);
245
246
247 }
248
249 int main(int argc, char** argv) {
250
251     FILE *fentrada, *fsaida; //arquivos para leitura e escrita
252     kMeans *c = new kMeans();
253     clock_t tStart; // vari veis para calcular o tempo
254
255     //inicia o contador para o tempo de entrada de dados
256     tStart = clock();
257
258     // abertura dos arquivos de entrada e de saida
259     fentrada = fopen("BaseKMeans_Cr_Ni_Mo.csv", "r");
260     if (fentrada == NULL) perror("Error opening file fentrada");
261
262
263     //pega o tamanho da matriz de entrada
264     fscanf(fentrada, "%d %d", &numDados, &numAtr);
265     cout << numDados << endl;
266     cout << numAtr << endl;
267
268
269     //aloca e le a matriz de entrada
270     if ((dados = (float**) malloc(sizeof (float*)*numDados)) == NULL)
            return -1;
271     for (int i = 0; i < numDados; i++) {
272         if ((dados[i] = (float*) malloc(sizeof (float)*numAtr)) ==
               NULL) return -1;
273         for (int j = 0; j < numAtr; j++) {
274             fscanf(fentrada, "%f", &dados[i][j]);
275         }
276     }
277     fclose(fentrada);
```

```
278
279     //encerra o contador para tempo de entrada de dados
280     cout << "Tempo de Entrada de Dados: " << (double) (clock() -
            tStart) / CLOCKS_PER_SEC << endl;
281
282     //aloca espa o para o vetor de resultados
283     resultado = (int*) malloc(numDados * sizeof (int));
284     assert(resultado != NULL);
285
286     normalizaDados(dados, numDados);
287
288     //inicia o contador para o tempo de processamento
289     tStart = clock();
290
291     //executa o algoritmo sequencial
292     processa(PAR_1_SEQ_0);
293
294     //encerra o contador para tempo de processamento
295     cout << "Tempo de Processamento: " << (double) (clock() - tStart)
            / CLOCKS_PER_SEC << endl;
296
297     //inicia o contador para o tempo de saida de dados
298     tStart = clock();
299
300     //escreve o resultado final no arquivo
301     c->file_write("teste_result.txt", numClusters, numDados, numAtr,
            clusters, resultado);
302
303     //encerra o contador para tempo para saida de dados
304     cout << "Tempo de Sa da de dados: " << (double) (clock() -
            tStart) / CLOCKS_PER_SEC << endl;
305
306     //libera a memoria
307     free(dados);
308
309     return 0;
310 }
311  }
```

APÊNDICE B – Código Fonte do Algoritmo K-NN

```
1  //declaracao das bibliotecas
2  #include <cstdlib>
3  #include <iostream>
4  #include "stdio.h"
5  #include <assert.h>
6  #include <pthread.h>
7  #include <sys/types.h>
8
9
10 #define numClasses 3
11 #define K 3
12 #define numThreads 4
13 #define PAR_1_SEQ_0 0
14
15 clock_t tStart; // vari veis para calcular o tempo
16 double tempo = 0;
17
18 using namespace std;
19
20 //struct que cria a thread
21
22 typedef struct {
23     int idThread, indice;
24     float dist;
25 } t_Args;
26
27 //struct que armazena as distancias encontradas
28
29 struct Dist {
30     int id, classe;
31     float dist;
32 };
33
34
35 //variaveis globais
36 float **baseTeste, **baseTreino; //matrizes de entrada
37 int linhaTeste, colTeste, linhaTreino, colTreino; //dimensoes das
      matrizes de entrada
```

```
38 static Dist distancias[10000]; //vetor que armazena as distancias
39 pthread_mutex_t mutex; // lock para exclusao mutua entre as threads
40
41
42 //funcao que escreve os dados finais em um arquivo
43
44 void file_write(char *filename, int *resultado) {
45
46     FILE *fptr;
47     int i, j;
48     char outFileName[1024];
49     sprintf(outFileName, "%s.Resultado", filename);
50     printf("Gravando dados no arquivo \"%s\"\n", outFileName);
51     fptr = fopen(outFileName, "w");
52     fprintf(fptr, "N mero de vizinhos escolhidos:\n");
53     fprintf(fptr, "%d\n", K);
54     fprintf(fptr, "Resultado final:\n");
55     fprintf(fptr, "C digo:\t");
56     fprintf(fptr, "Classe:\n");
57     for (i = 0; i < linhaTreino; i++) {
58         fprintf(fptr, "%d\t", i);
59         fprintf(fptr, "%d\n", resultado[i]);
60     }
61     fclose(fptr);
62
63     //return 1;
64 }
65
66 //funcao que calcula a distancia euclidiana
67
68 float distEucl(int numCoord, float *linha1, float *linha2) {
69
70     float ans = 0.0;
71     for (int i = 0; i < numCoord; i++)
72         ans += (linha1[i] - linha2[i]) * (linha1[i] - linha2[i]);
73     return (ans);
74 }
75
76 // f u n   o que gerencia as threads
77
78 void *criaThread(void *params) {
79
```

```
80      t_Args *arg = (t_Args *) params; //receber  os argumentos para a
            thread
81      int elemPorThread = linhaTreino / numThreads;
82      int pri = arg->idThread * elemPorThread;
83      int ult = pri + elemPorThread;
84      // tStart = clock();
85
86      for (int k = pri; k < ult; k++) {
87          // printf("In cio da Thread: %d %d\n", arg->idThread, arg->
                indice);
88          // pthread_mutex_lock(&mutex);
89
90          for (int i = 0; i < linhaTeste; i++) {
91              distancias[i].dist = distEucl(colTeste - 1, baseTreino[k
                    ], baseTeste[i]);
92          }
93          //pthread_mutex_unlock(&mutex); //saida da secao critica
94          // printf("Fim da Thread: %d\n", arg->idThread);
95          //tempo += (double) (clock() - tStart) / CLOCKS_PER_SEC;
96      }
97      //tempo += (double) (clock() - tStart) / CLOCKS_PER_SEC;
98      //cout << "Tempo de Processamento Paralelo: " << (double) (clock
            () - tStart) / CLOCKS_PER_SEC<< endl;
99  }
100
101 int calcKnn(/*int indice,*/ int opcao) {
102
103
104     int classes[numClasses]; //vetor que guarda a qtde de distancias
            de cada classe
105     //struct Dist distancias[linhaTeste]; //struct que guarda os
            dados recebidos
106     pthread_t threads[numThreads];
107     t_Args *arg; //receber  os argumentos para a thread
108
109
110     //inicializando vetor classes
111     for (int i = 0; i < numClasses; i++) {
112         classes[i] = 0;
113     }
114
115     //colocando as classes em cada linha da matriz de treino
```

```
116     for (int i = 0; i < linhaTreino; i++) {
117         distancias[i].classe = baseTreino[i][colTreino - 1];
118         distancias[i].id = i;
119     }
120 
121     if (opcao) {
122 
123         tStart = clock();
124         //cout << "Executando algoritmo paralelo" << endl;
125 
126         for (int t = 0; t < numThreads; t++) {
127 
128             arg = (t_Args*) malloc(sizeof (t_Args));
129             assert(arg != NULL);
130             arg->idThread = t;
131             //arg->indice = indice;
132 
133             if (pthread_create(&threads[t], NULL, criaThread, (void*)
                    arg)) {
134                 printf("--ERRO: pthread_create()\n");
135                 exit(1);
136             }
137         }
138         //encerra o contador para tempo de processamento
139         tempo += (double) (clock() - tStart) / CLOCKS_PER_SEC;
140 
141 
142 
143         ///espera todas as threads terminarem
144         for (int t = 0; t < numThreads; t++) {
145             if (pthread_join(threads[t], NULL)) {
146                 cout << "--ERRO: pthread_join() \n";
147                 exit(-1);
148             }
149 
150         }
151 
152 cout << "Tempo de Processamento Paralelo: " << (double) (clock() -
    tStart) / CLOCKS_PER_SEC << endl;
153 
154     } else {
155         tStart = clock();
```

```
156         // cout << "Executando algoritmo sequencial." << endl;
157         //calculando as distancias
158 
159         for (int i = 0; i < linhaTreino; i++) {
160             for (int k = 0; k < linhaTeste; k++) {
161             distancias[k].dist = distEucl(colTeste - 1, baseTreino[i
                ], baseTeste[k]);
162 
163 
164 
165             }}
166         //tempo += (double) (clock() - tStart) / CLOCKS_PER_SEC;
167         cout << "Tempo de Processamento Sequencial: " << (double) (
            clock() - tStart) / CLOCKS_PER_SEC << endl;
168     }
169 
170     //ordenando o vetor de distancias
171     float auxdist = 0;
172     int auxclasse, auxid;
173     for (int i = 0; i < linhaTeste - 1; i++) {
174         for (int j = i + 1; j < linhaTeste; j++) {
175             if (distancias[i].dist > distancias[j].dist) {
176                 auxdist = distancias[j].dist;
177                 auxclasse = distancias[j].classe;
178                 auxid = distancias[j].id;
179                 distancias[j].dist = distancias[i].dist;
180                 distancias[j].classe = distancias[i].classe;
181                 distancias[j].id = distancias[i].id;
182                 distancias[i].dist = auxdist;
183                 distancias[i].classe = auxclasse;
184                 distancias[i].id = auxid;
185             }
186         }
187     }
188 
189 
190     //calculando a classe a qual pertence a amostra
191     for (int i = 0; i < numClasses; i++) {
192         for (int j = 0; j < K; j++) {
193             if (distancias[j].classe == i + 1) {
194                 classes[i]++;
195             }
```

```
196
197             }
198         }
199
200
201         //pegando o maior valor de ocorrencias de uma classe
202         int aux = 0;
203         for (int i = 0; i < numClasses; i++) {
204             if (classes[i] > aux) {
205                 aux = i;
206             }
207         }
208
209
210
211         return aux + 1;
212     }
213
214     int main(int argc, char** argv) {
215
216         FILE *fbTeste, *fbTreino;
217
218         int *resultad;
219
220
221         //inicia o contador para o tempo de entrada de dados
222         tStart = clock();
223
224         // abertura dos arquivos de treinamento e teste
225         fbTeste = fopen("BaseKnn_TesteFinal2_Contagem.csv", "r"); //
                matriz que contem as amostras definidas
226         if (fbTeste == NULL) perror("Error opening file fentrada");
227         fbTreino = fopen("BaseKnn_TreinoFinal2_Contagem.csv", "r"); //
                matriz que contem as amostras sem defini ao
228         if (fbTreino == NULL) perror("Error opening file fsaida");
229
230         /* fbTeste = fopen("BaseKnn_TreinoFinal2.csv", "r"); //matriz que
                contem as amostras definidas
231          if (fbTeste == NULL) perror("Error opening file fentrada");
232          fbTreino = fopen("BaseKnn_TesteFinal2.csv", "r"); //matriz que
                contem as amostras sem defini ao
233          if (fbTreino == NULL) perror("Error opening file fsaida");*/
```

```
234
235
236     //atribuindo tamanho as colunas
237     fscanf(fbTeste, "%d %d", &linhaTeste, &colTeste);
238     fscanf(fbTreino, "%d %d", &linhaTreino, &colTreino);
239
240     ///aloca e le a matriz de teste
241     if ((baseTeste = (float**) malloc(sizeof (float*)*linhaTeste)) ==
            NULL) return -1;
242     for (int i = 0; i < linhaTeste; ++i) {
243         if ((baseTeste[i] = (float*) malloc(sizeof (float)*colTeste))
                == NULL) return -1;
244         for (int j = 0; j < colTeste; ++j) {
245             fscanf(fbTeste, "%f", &baseTeste[i][j]);
246         }
247     }
248     fclose(fbTeste);
249
250     ///aloca e le a matriz de treino
251     if ((baseTreino = (float**) malloc(sizeof (float*)*linhaTreino))
           == NULL) return -1;
252     for (int i = 0; i < linhaTreino; ++i) {
253         if ((baseTreino[i] = (float*) malloc(sizeof (float)*colTreino
               )) == NULL) return -1;
254         for (int j = 0; j < colTreino; ++j) {
255             fscanf(fbTreino, "%f", &baseTreino[i][j]);
256         }
257     }
258     fclose(fbTreino);
259
260     int resultado[linhaTreino]; //vetor que guarda os resultados das
           distancias
261
262     //encerra o contador para tempo de entrada de dados
263     cout << "Tempo de Entrada de Dados: " << (double) (clock() -
           tStart) / CLOCKS_PER_SEC << endl;
264
265     //aloca espa o para o vetor de resultados
266     resultad = (int*) malloc(linhaTreino * sizeof (int));
267     assert(resultad != NULL);
268
269
```

```
270     //atribuindo o valor -1 a todas as posi oes do vetor resultado
271     for (int k = 0; k < linhaTreino; ++k) {
272         resultado[k] = -1;
273         //cout<<k<<"t";
274     }
275 
276     //inicia o contador para o tempo de processamento
277     //tStart = clock();
278 
279     //executa o algoritmo
280     cout << "Aguarde..." << endl;
281     /* for (int i = 0; i < linhaTreino; i++) {
282          resultado[i] = calcKnn(i, PAR_1_SEQ_0);
283 
284      }*/
285     calcKnn(PAR_1_SEQ_0);
286     //encerra o contador para tempo de processamento
287   //  cout << "Tempo de Processamento: " << (double) (clock() -
        tStart) / CLOCKS_PER_SEC << endl;
288 
289     //inicia o contador para o tempo de saida de dados
290     tStart = clock();
291 
292     //escreve o resultado no arquivo
293     file_write("Knn_result.txt", resultado);
294 
295 
296 
297     //encerra o contador para tempo para sa da de dados
298     cout << "Tempo de Sa da de dados: " << (double) (clock() -
          tStart) / CLOCKS_PER_SEC << endl;
299 
300 
301 
302 
303     //parte do codigo que verifica a qtde de acertos
304     int acertos = 0;
305     int compara[linhaTreino];
306 
307 
308     for (int i = 0; i < linhaTreino; i++) {
309         if (resultado[i] == baseTreino[i][colTreino - 1]) {
```

```
310                 acertos++;
311
312             }
313         }
314         cout << "Qtde de acertos do algoritmo: " << acertos << endl;
315
316
317         return 0;
318 }
319
320
321  }
```

APÊNDICE C – Código Fonte do Algoritmo Perceptron de Múltiplas Camadas

```
1  #include <cstdlib>
2  #include <iostream>
3  #include "stdio.h"
4  #include "cmath"
5  #include <time.h>
6  #include <stdlib.h>
7  #include <pthread.h>
8  #include <sys/types.h>
9  #include <assert.h>
10
11 using namespace std;
12
13
14 #define colHidd 4   //num de neuronios na camada oculta + bias
15 #define atributos 4
16 #define linhaW1 atributos +1       //atributos + bias
17 #define colW1 colHidd    //colunas Hidd
18 #define linhaW2 colHidd      //colunas Hidd
19 #define colW2 atributos         //num atributos
20 #define colOut 3       //num respostas
21 #define colD1 atributos         //num atributos
22 #define colD2 3        //num saidas
23 #define colInput atributos +1     //num de atributos + bias
24 #define numThreads 16
25 #define PAR_1_SEQ_0 1
26 #define numEpocas 10000
27
28 //struct que cria a thread
29
30 typedef struct {
31     int idThread, size;
32     float in[atributos];
33     float out[colOut];
34 } t_Args;
35
36
```

```
37
38 //typedef unsigned char byte;
39
40 using namespace std;
41
42
43 //variaveis globais
44 float **baseTeste, **baseTreino, **BaseSaida; //matrizes de entrada
45 int linhaTeste, colTeste, linhaTreino, colTreino; //dimensoes das
      matrizes de entrada
46 int epoca; //contador de  pocas
47 float input[colInput]; //camada de entrada (n  atributos + bias)
48 float hidden[colHidd]; //camada escondida;
49 double w2[linhaW2][colW2]; //matriz de pesos de saida
50 float output[colOut]; //matriz de resultado
51 float delta1[colD1]; //diferen a de pesos da entrada
52 float delta2[colD2]; //diferen a de pesos da camada oculta
53 double w1[linhaW1][colW1]; //new double[data.length][atributos]; //
      matriz de pesos de entrada
54 clock_t tStart; // vari veis para calcular o tempo
55 pthread_mutex_t mutex; // lock para exclusao mutua entre as threads
56 double tempo = 0;
57
58
59
60 //declara  o das fun oes
61 void iniciaPesos();
62 void normalizaDados(float *data, int tamMatriz);
63 void Treinamento(float **base, int opcao);
64 void Aprendizado(float **base, int indice); //float entrada[], float
      saida[]);
65 void Teste(float entrada[], float saida[]);
66 void iniciaMatrizes();
67
68
69
70 //inicializa as matrizes auxiliares
71
72 void iniciaMatrizes() {
73
74     //inicia a matriz de entrada
75     for (int i = 0; i < colInput; i++) {
```

```
76          if (i < colInput - 1) {
77              input[i] = 0;
78          } else {
79              input[i] = 1;
80          }
81
82      }
83
84
85
86      //inicia a camada oculta
87      for (int i = 0; i < colHidd; i++) {
88          if (i < colHidd - 1) {
89              hidden[i] = 0;
90          } else {
91              hidden[i] = 1;
92          }
93
94
95      }
96
97
98
99      //inicia a matriz de saida
100     for (int i = 0; i < colOut; i++) {
101
102         output[i] = 0;
103
104     }
105
106
107     //inicia as matrizes de ajustes dos pesos
108     for (int i = 0; i < colD1; i++) {
109
110         delta1[i] = 0;
111
112     }
113
114
115
116
117     for (int i = 0; i < colD2; i++) {
```

```
118
119         delta2[i] = 0;
120
121     }
122
123     //inicia as matrizes de pesos
124     for (int i = 0; i < linhaW1; i++) {
125         for (int j = 0; j < colW1; j++) {
126             w1[i][j] = 0;
127         }
128     }
129
130     for (int i = 0; i < linhaW2; i++) {
131         for (int j = 0; j < colW2; j++) {
132             w2[i][j] = 0;
133         }
134     }
135
136 }
137
138 //inicializa a matriz de pesos
139
140 void iniciaPesos() {
141     //byte i, j;
142     srand(time(NULL));
143
144     // Inicializa os pesos randomicos entre 0.1 and 0.9
145     for (int i = 0; i < linhaW1; i++) {
146         for (int j = 0; j < colW1; j++) {
147             w1[i][j] = ((rand() % 8 + 1)*0.1);
148             //  cout << w1[i][j] << endl;
149
150         }
151     }
152
153
154     for (int i = 0; i < linhaW2; i++) {
155         for (int j = 0; j < colW2; j++) {
156             w2[i][j] = ((rand() % 8 + 1)*0.1);
157
158         }
159     }
```

```
160
161
162 }
163
164
165
166 //fun ao de normaliza  o
167
168 void normalizaDados(float **base, int tamMatriz) {
169
170     float maior[atributos];
171     float menor[atributos];
172
173     //int x = 0;
174     // Encontra maior e menor valor de cada coluna
175     for (int i = 0; i < atributos; i++) {
176         for (int j = 0; j < tamMatriz; j++) {
177             if (j == 0) {
178                 maior[i] = base[j][i];
179                 menor[i] = base[j][i];
180             } else {
181                 if (base[j][i] > maior[i]) {
182                     maior[i] = base[j][i];
183                 }
184                 if (base[j][i] < menor[i]) {
185                     menor[i] = base[j][i];
186                 }
187             }
188         }
189     }
190
191
192     //Fun   o de normaliza  o
193     for (int i = 0; i < tamMatriz; i++) {
194         for (int j = 0; j < atributos; j++) {
195             base[i][j] = (base[i][j] - menor[j])
196                     / (maior[j] - menor[j]);
197             // System.out.print("    "+data[i][0][j]);
198         }
199         // System.out.println("");
200     }
201
```

```
202
203
204 }
205
206 void *criaThread(void *params) {
207
208     t_Args *arg = (t_Args *) params; //receber  os argumentos para a
        thread
209     int elemPorThread = arg->size / numThreads;
210     int pri = arg->idThread * elemPorThread;
211     int ult = pri + elemPorThread;
212     tStart = clock();
213     for (int k = pri; k < ult; k++) {
214
215         Aprendizado(baseTreino, k);
216
217
218
219     }
220
221 }
222
223 //fun ao que faz o treinamento
224
225 void Treinamento(float **base, int opcao) {
226
227     float entrada[atributos];
228     float saida[atributos];
229     pthread_t threads[numThreads];
230     t_Args *arg; //receber  os argumentos para a thread*/
231
232
233     if (opcao) {
234
235         //inicia o contador de tempo de processamento
236         tStart = clock();
237
238         //  cout << "Executando algoritmo paralelo" << endl;
239         for (int t = 0; t < numThreads; t++) {
240             arg = (t_Args*) malloc(sizeof (t_Args));
241             assert(arg != NULL);
242             arg->idThread = t;
```

```
243             arg->size = linhaTreino;
244
245
246             // cout << "Tempo de Processamento loop: " << (double) (
                    clock() - tStart) / CLOCKS_PER_SEC << endl;
247
248
249             if (pthread_create(&threads[t], NULL, criaThread, (void*)
                    arg)) {
250                 printf("--ERRO: pthread_create()\n");
251                 exit(1);
252             }
253
254
255
256         }
257
258         ///espera todas as threads terminarem
259         for (int t = 0; t < numThreads; t++) {
260             if (pthread_join(threads[t], NULL)) {
261                 cout << "--ERRO: pthread_join() \n";
262                 exit(-1);
263             }
264
265             //-- desaloca o lock de exclusao mutua
266             pthread_mutex_destroy(&mutex);
267
268         }
269
270         //encerra o contador para tempo de processamento
271         tempo += (double) (clock() - tStart) / CLOCKS_PER_SEC;
272         //if(epoca==4999)
273         cout << "Tempo de Processamento Paralelo: " << tempo/*(double
                ) (clock() - tStart) / CLOCKS_PER_SEC*/ << endl;
274
275
276     } else {
277
278         //inicia o contador para tempo de processamento
279         tStart = clock();
280         //cout << "Executando algoritmo sequencial" << endl;
281         // chamando o metodo de aprendizado
```

```
282         for (int i = 0; i < linhaTreino; i++) {
283 
284             Aprendizado(baseTreino, i);
285 
286 
287         }
288         tempo += (double) (clock() - tStart) / CLOCKS_PER_SEC;
289         //encerra o contador para tempo de processamento
290         cout << "Tempo de Processamento Sequencial: " << tempo <<
               endl;
291 
292 
293     }
294     epoca++;
295 
296 }
297 
298 //fun ao de aprendizado
299 
300 void Aprendizado(float **base, int indice) {//(float entrada[], float
        saida[]) {
301 
302 
303     float soma, saida_j;
304 
305 
306 
307     // Initialize input units
308     for (int i = 0; i < atributos; i++) {
309         input[i] = base[indice][i];
310 
311     }
312 
313 
314     // Calculate hidden units
315     for (int j = 0; j < colHidd - 1; j++) {
316         soma = 0;
317         for (int i = 0; i < colInput; i++) {
318             soma = soma + w1[i][j] * input[i]; //base[indice][i]; //
                   input[i];
319         }
320         hidden[j] = 1 / (1 + exp(-soma));
```

```
321     }
322
323
324     // Calculate output units
325     for (int j = 0; j < colOut; j++) {
326         soma = 0;
327         for (int i = 0; i < colHidd; i++) {
328             soma = soma + w2[i][j] * hidden[i];
329         }
330         output[j] = 1 / (1 + exp(-soma));
331
332     }
333
334
335
336     // Calculate delta2 errors
337     for (int j = 0; j < colOut; j++) {
338         //for (int j = atributos; j < atributos + colOut; j++) {
339
340         //saida_j = base[indice][j+atributos] - output[j]; //saida[j]
                - output[j];
341
342         if (base[indice][j + atributos] == 0) {
343             saida_j = 0.1;
344         } else if (base[indice][j + atributos] == 1) {
345             saida_j = 0.9;
346         } else {
347             saida_j = base[indice][j + atributos];
348         }
349         delta2[j] = output[j] * (1 - output[j]) * (saida_j - output[j
                ]);
350
351     }
352
353
354     // Calculate delta1 errors
355     for (int j = 0; j < colHidd; j++) {
356         soma = 0;
357         for (int i = 0; i < colOut; i++) {
358             soma = soma + delta2[i] * w2[j][i];
359         }
360
```

```
361             delta1[j] = hidden[j] * (1 - hidden[j]) * soma; //derivada da
                    fun ao sigmoide
362         }
363 
364 
365 
366         //entrada na secao critica
367         // pthread_mutex_lock(&mutex);
368 
369         for (int i = 0; i < linhaW2; i++) {
370             for (int j = 0; j < colW2; j++) {
371                 w2[i][j] = w2[i][j] + 0.5 * delta2[j] * hidden[i];
372             }
373         }
374 
375 
376         for (int i = 0; i < linhaW1; i++) {
377             for (int j = 0; j < colW1; j++) {
378                 w1[i][j] = w1[i][j] + 0.5 * delta1[j] * input[i];
379             }
380         }
381 
382         // pthread_mutex_unlock(&mutex); //saida da secao critica
383 
384 }
385 
386 void Teste(float entrada[], float saida[]) {
387 
388 
389         int i, j;
390         double soma;
391 
392 
393         // Initialize input units
394         for (i = 0; i < colInput - 1; i++) {
395             input[i] = entrada[i];
396 
397         }
398 
399         // Calculate hidden units
400         for (j = 0; j < colHidd - 1; j++) {
401             soma = 0;
```

```
402             for (i = 0; i < colInput; i++) {
403                 soma = soma + w1[i][j] * input[i];
404             }
405             hidden[j] = 1 / (1 + exp(-soma));
406         }
407 
408         // Calculate output units
409         for (j = 0; j < colOut; j++) {
410             soma = 0;
411             for (i = 0; i < colHidd; i++) {
412                 soma = soma + w2[i][j] * hidden[i];
413             }
414             output[j] = 1 / (1 + exp(-soma));
415 
416         }
417 
418 
419         // Assign output to param out[]
420         for (i = 0; i < colOut; i++) {
421 
422 
423             if (output[i] >= 0.5) {
424                 saida[i] = 1;
425             } else {
426                 saida[i] = 0;
427             }
428 
429 
430 
431         }
432 
433 }
434 
435 int main(int argc, char** argv) {
436 
437     FILE *fbTeste, *fbTreino;
438     clock_t tStart; // vari veis para calcular o tempo
439 
440 
441     //inicia o contador para o tempo de entrada de dados
442     tStart = clock();
443 
```

```
444
445     // abertura dos arquivos de treinamento e teste
446     fbTeste = fopen("Teste.csv", "r"); //matriz de testes
447     if (fbTeste == NULL) perror("Error opening file fentrada");
448     fbTreino = fopen("Treino3.csv", "r"); //matriz de treinamento
449     if (fbTreino == NULL) perror("Error opening file fsaida");
450
451     //atribuindo tamanho as colunas
452     fscanf(fbTeste, "%d %d", &linhaTeste, &colTeste);
453     fscanf(fbTreino, "%d %d", &linhaTreino, &colTreino);
454
455     ///aloca e le a matriz de teste
456     if ((baseTeste = (float**) malloc(sizeof (float*)*linhaTeste)) ==
            NULL) return -1;
457     for (int i = 0; i < linhaTeste; ++i) {
458         if ((baseTeste[i] = (float*) malloc(sizeof (float)*colTeste))
                == NULL) return -1;
459         for (int j = 0; j < colTeste; ++j) {
460             fscanf(fbTeste, "%f", &baseTeste[i][j]);
461         }
462     }
463     fclose(fbTeste);
464
465     ///aloca e le a matriz de treino
466     if ((baseTreino = (float**) malloc(sizeof (float*)*linhaTreino))
           == NULL) return -1;
467     for (int i = 0; i < linhaTreino; ++i) {
468         if ((baseTreino[i] = (float*) malloc(sizeof (float)*colTreino
               )) == NULL) return -1;
469         for (int j = 0; j < colTreino; ++j) {
470             fscanf(fbTreino, "%f", &baseTreino[i][j]);
471         }
472     }
473     fclose(fbTreino);
474
475     //encerra o contador para tempo de entrada de dados
476     cout << "Tempo de Entrada de Dados: " << (double) (clock() -
           tStart) / CLOCKS_PER_SEC << endl;
477
478
479     //normaliza as matrizes
480     normalizaDados(baseTreino, linhaTreino);
```

```
481
482     //inicia as matrizes auxiliares
483     iniciaMatrizes();
484     iniciaPesos();
485
486     //inicia o contador para o tempo de processamento
487     // tStart = clock();
488     //executa o algoritmo
489     cout << "Aguarde..." << endl;
490
491
492
493
494     //for (int i = 0; (i < 1) && ((TEerros > 0) || (TRerros > 0));
495     //        i++) {
496
497
498     for (int k = 0; k < numEpocas; k++) {
499
500
501         Treinamento(baseTreino, PAR_1_SEQ_0);
502
503
504         //encerra o contador para tempo de processamento
505         // cout << "Tempo de Processamento: " << (double) (clock() -
                tStart) / CLOCKS_PER_SEC << endl;
506
507         int TRacertos = 0;
508         int TRerros = 0;
509
510         int TEacertos = 0;
511         int TEerros = 0;
512
513
514         float out[colOut];
515         float in[atributos];
516
517         //verificando o aprendizado da massa de treino
518         for (int i = 0; i < linhaTreino; i++) {
519             for (int j = 0; j < atributos; j++) {
520
521                 in[j] = baseTreino[i][j];
```

```
522                 }
523
524                 Teste(in, out);
525
526                 if ((out[0] == baseTreino[i][4])&&(out[1] == baseTreino[i
                        ][5])&&(out[2] == baseTreino[i][6])) {
527                     TRacertos++;
528
529                 } else {
530                     TRerros++;
531                 }
532
533
534
535             }
536
537
538
539             cout << "Epoca: " << +epoca << endl;
540             cout << "TRacertos: " << +TRacertos << endl;
541             cout << "TRerros: " << +TRerros << endl;
542
543
544         }
545
546     }
```