

**ESCOLA POLITÉCNICA DA UNIVERSIDADE DE SÃO PAULO**

**GUSTAVO RYUJI TAIRA**

**ESTUDO PRÁTICO DE ALGORITMOS DE MACHINE LEARNING**

**SÃO PAULO**

**2019**

**GUSTAVO RYUJI TAIRA**

**ESTUDO PRÁTICO DE ALGORITMOS DE MACHINE LEARNING**

**Estudo prático de diferentes algoritmos de machine learning. Atividade preparatória para o projeto de pesquisa “Aplicação de Bayesian Deep Learning para Monitoramento de Smart City e Operação de Processos da Indústria Química”.**

**Orientador: Prof<sup>o</sup>. Dr. Song Won Park**

**SÃO PAULO**

**2019**

## **RESUMO**

Este trabalho, intitulado “Estudo Prático de Algoritmos de Machine Learning”, foi realizado como etapa preparatória para o desenvolvimento do projeto de pesquisa “Aplicação de Bayesian Deep Learning para Monitoramento de Smart City e Operação de Processos da Indústria Química”. O estudo foi realizado com objetivo de ganhar experiência na área de machine learning e compreender os conceitos teóricos e práticos dos principais algoritmos de aprendizado de máquina. Desta forma, diversos algoritmos de aprendizado de máquina abordados em (WU; KUMAR, 2009) foram selecionados para estudo (C4.5, K-Means, SVM, Apriori, EM, AdaBoost, kNN, Naïve Bayes e CART). Além desses algoritmos, outros quatro algoritmos considerados relevantes foram selecionados para o estudo: Rede Neural Feed-forward, Rede Neural Convolucional, Rede Neural Recorrente LSTM e Support Vector Clustering. Para cada algoritmo abordado no estudo, são apresentadas uma breve descrição teórica do algoritmo e aplicações práticas do algoritmo em problemas de análise dados. O trabalho conseguiu abordar uma boa variedade de métodos supervisionados e não supervisionados de aprendizado de máquina, além de demonstrar a aplicabilidade dos algoritmos para solução de problemas de classificação, predição, agrupamento de dados (clustering) e análise associativa. De forma geral, este trabalho apresenta uma boa introdução aos diferentes algoritmos de aprendizado de máquina e suas aplicações práticas.

## LISTA DE FIGURAS

Figura 1 - Pseudocódigo do algoritmo C4.5 .....	13
Figura 2 - Árvore de decisão gerada para a base de dados simples .....	15
Figura 3 - Código em Python da implementação do caso simples do C4.5 .....	16
Figura 4 - Código em Python da implementação do C4.5 para uma base de dados real .....	26
Figura 5 - Pseudocódigo do algoritmo k-Means (WU; KUMAR, 2009).....	28
Figura 6 - Base de dados para implementação do k-Means .....	29
Figura 7 - k-Means (k = 2) .....	29
Figura 8 - k-Means (k = 3) .....	30
Figura 9 - k-Means (k = 4) .....	30
Figura 10 - k-Means (k = 5) .....	31
Figura 11 - k-Means (k = 6) .....	31
Figura 12 - Código em Python para implementação do k-Means.....	36
Figura 13 - Ilustração do hiperplano ótimo em SVC para um caso linearmente separável (WU; KUMAR, 2009).....	37
Figura 14 - SVC com Kernel linear (WU; KUMAR, 2009).....	43
Figura 15 - SVC com Kernel de base radial (WU; KUMAR, 2009) .....	43
Figura 16 - Resultado da implementação do algoritmo SVM .....	44
Figura 17 - Código em Python para implementação do SVM .....	47
Figura 18 - Algoritmo do algoritmo Apriori (WU; KUMAR, 2009).....	49
Figura 19 - Função apriori-gen (WU; KUMAR, 2009).....	49
Figura 20 - Algoritmo para geração de regras de associação (WU; KUMAR, 2009).....	50
Figura 21 - Resultado da implementação do algoritmo apriori para o caso simples ..	51
Figura 22 - Resultado da implementação do algoritmo Apriori para o caso complexo .....	53
Figura 23 - Código em Python para implementação do algoritmo Apriori .....	56
Figura 24 - Implementação 1D - Iteração 1 .....	58
Figura 25 - Implementação 1D - Iteração 2 .....	59
Figura 26 - Implementação 1D - Iteração 3 .....	59
Figura 27 - Implementação 1D - Iteração 4 .....	60
Figura 28 - Implementação 1D - Iteração 5 .....	60
Figura 29 - Implementação 1D - Iteração 6 .....	60
Figura 30 - Implementação 1D - Iteração 7 .....	61
Figura 31 - Implementação 1D - Iteração 8 .....	61
Figura 32 - Implementação 1D - Iteração 9 .....	61
Figura 33 - Implementação 1D - Iteração 10 .....	62
Figura 34 - Código em Python para implementação do EM para o caso 1D.....	64
Figura 35 - Estado inicial.....	65
Figura 36 - Curva de log-verossimilhança .....	65
Figura 37 - Estado final .....	65
Figura 38 - Código em Python para implementação do EM para o caso 2D.....	68

Figura 39 - Procedimento genérico para implementação de algoritmos boosting (WU; KUMAR, 2009). .....	70
Figura 40 - Pseudocódigo do algoritmo AdaBoostT (WU; KUMAR, 2009).....	71
Figura 41 - Performance do classificador forte com 50 classificadores fracos .....	72
Figura 42 - Performance do classificador forte com 400 classificadores fracos .....	72
Figura 43 - Código em Python para implementação do AdaBoost.....	75
Figura 44 - Pseudocódigo do algoritmo k-Nearest Neighbors .....	77
Figura 45 - k-Nearest Neighbors com valor pequeno, médio e grande de k (WU; KUMAR, 2009). .....	78
Figura 46 - Localização das amostras em gráfico 2D .....	79
Figura 47 - Resultado da classificação da amostra analisada pelo algoritmo .....	80
Figura 48 - Código em Python da implementação do caso simples do kNN.....	81
Figura 49 - Código em Python da implementação do caso simples do kNN.....	83
Figura 50 – Código em Python para implementação do Naïve Bayes .....	89
Figura 51 - Algoritmo simplificado para a construção de árvores de decisão CART (WU; KUMAR, 2009). .....	91
Figura 52 - Algoritmo simplificado para a podagem de árvores de decisão CART (WU; KUMAR, 2009). .....	92
Figura 53 - Código em Python para a implementação do CART.....	99
Figura 54 - Unidade de processamento .....	101
Figura 55 - Rede Neural Feedfoward Multicamada.....	102
Figura 56 – Propagação forward (preto). Propagação backward(azul) .....	103
Figura 57 - Fluxo de sinais durante a aplicação do “backpropagation”. Parte superior: estágio “forward”. Parte inferior: estágio “backward” .....	106
Figura 58 - Amostras do conjunto de dados MNIST .....	107
Figura 59 – Exemplos de classificações realizadas pela rede neural feedforward..	108
Figura 60 – Código em Python para implementação da rede neural feedfoward....	110
Figura 61 - Rede convolucional comumente utilizada para reconhecimento de imagens .....	112
Figura 62 - Exemplo de convolução bidimensional .....	114
Figura 63 - Filtragem de componentes de uma imagem quando aplicada convolução .....	115
Figura 64 - Estágio de uma camada de convolução .....	116
Figura 65 - Exemplo da ação da função max pooling .....	116
Figura 66 – Exemplo de invariância gerada ao aplicar função pooling .....	117
Figura 67 - Exemplo de imagens de gato e cachorro analisadas pela rede neural convolucional.....	118
Figura 68 - Exemplos de classificações realizadas pela rede neural convolucional	118
Figura 69 - Código em Python para implementação da rede neural convolucional.	122
Figura 70 - Exemplo de rede neural recorrente com loop de retroalimentação.....	124
Figura 71 - Rede neural recorrente como uma sequência de redes neurais.....	124
Figura 72 – Gradient vanishing problem .....	125
Figura 73 - Módulo de repetição utilizado em redes recorrentes tradicionais .....	126
Figura 74 - Módulo de repetição nas redes LSTM .....	126

Figura 75 - Célula de estado .....	127
Figura 76 - Gate da célula de memória .....	127
Figura 77 - Forget gate.....	128
Figura 78 - Input gate .....	128
Figura 79 - Atualização da célula de estado .....	129
Figura 80 - Output gate .....	129
Figura 81 - Função erro ao longo do treinamento da rede LSTM.....	131
Figura 82 - Comparação: valores previstos x valores esperados.....	131
Figura 83 - Código em Python para implementação da rede neural LSTM.....	136
Figura 84 - Influência de q no número de clusters (a) q = 1 (b) q = 20 (c) q = 24 (d) q = 48 .....	141
Figura 85 - Influência de C no formato e número de clusters (a) C = 1 (b) C = 0.07 .....	141
Figura 86 - Dados SVC (caso simples) .....	142
Figura 87 - Clusters gerados para q = 6.5 e C = 10 .....	143
Figura 88 - Código em Python para implementação do SVC .....	147
Figura 89 - Código em Python para solução do caso simples.....	148
Figura 90 - Código em Python para solução do caso industrial real .....	149

## LISTA DE TABELAS

Tabela 1 – Base de dados utilizada para a implementação do caso simples do C4.5 .....	14
Tabela 2 - Taxa de precisão do C4.5 implementado para base de dados de qualidade de vinho.....	17
Tabela 3 - Base de dados de treinamento utilizada para implementação do SVM. ..	44
Tabela 4 – Base de dados de teste utilizada para avaliação de desempenho do algoritmo SVM.....	44
Tabela 5 - Base de dados utilizada para implementação do caso simples do Apriori .....	51
Tabela 6 – Base de dados utilizada para implementação do caso complexo do Apriori .....	52
Tabela 7 – Base de dados utilizada para implementação do kNN (caso simples) ....	79

## SUMÁRIO

1	C4.5.....	12
1.1	DESCRIÇÃO DO ALGORITMO C4.5.....	12
1.2	EXEMPLOS DE APLICAÇÃO DO ALGORITMO C4.5.....	14
1.2.1	APLICAÇÃO DO C4.5: CASO SIMPLES.....	14
1.2.2	CÓDIGO CASO SIMPLES .....	15
1.2.3	APLICAÇÃO DO C4.5: CASO REAL.....	16
1.2.4	CÓDIGO CASO REAL.....	17
1.3	BIBLOGRAFIA .....	26
2	K-MEANS .....	27
2.1	DESCRIÇÃO DO ALGORITMO .....	27
2.2	APLICAÇÃO DO ALGORITMO K-MEANS.....	28
2.2.1	CÓDIGO .....	32
2.3	BIBLIOGRAFIA .....	36
3	SVM: SUPPORT VECTOR MACHINES.....	37
3.1	SVC: SUPPORT VECTOR CLASSIFIER.....	37
3.2	SVC COM MARGEM SUAVE E OTIMIZAÇÃO.....	40
3.3	ARTIFÍCIO DO KERNEL.....	41
3.4	APLICAÇÃO DO SVM.....	44
3.4.1	CÓDIGO .....	45
3.5	BIBLIOGRAFIA .....	47
4	APRIORI.....	48
4.1	DESCRIÇÃO DO ALGORITMO .....	48
4.1.1	MINERANDO PADRÕES DE FREQUÊNCIA E REGRAS DE ASSOCIAÇÃO.....	48
4.1.2	APRIORI .....	48
4.2	APLICAÇÃO DO ALGORITMO APRIORI .....	51
4.2.1	APLICAÇÃO DO APRIORI: CASO SIMPLES.....	51
4.2.2	APLICAÇÃO DO APRIORI: CASO COMPLEXO.....	52
4.2.3	CÓDIGO CASO COMPLEXO.....	53
4.3	BIBLIOGRAFIA .....	56
5	EM: EXPECTATION-MAXIMIZATION.....	57



5.1	DESCRIÇÃO DO ALGORITMO EM.....	57
5.2	APLICAÇÃO DO ALGORITMO EM.....	58
5.2.1	APLICAÇÃO DO EM: CASO 1D.....	58
5.2.2	CÓDIGO CASO 1D .....	62
5.2.3	APLICAÇÃO DO EM: CASO 2D.....	64
5.2.4	CÓDIGO CASO 2D .....	66
5.3	BIBLIOGRAFIA .....	68
6	ADABOOST.....	69
6.1	DESCRIÇÃO DO ALGORITMO ADABOOST .....	69
6.2	APLICAÇÃO DO ALGORITMO ADABOOST .....	71
6.2.1	CÓDIGO .....	73
6.3	BIBLIOGRAFIA .....	75
7	KNN: K-NEAREST NEIGHBORS.....	77
7.1	DESCRIÇÃO DO ALGORITMO K-NEAREST NEIGHBORS .....	77
7.2	APLICAÇÃO DO ALGORITMO K-NEAREST NEIGHBORS .....	78
7.2.1	APLICAÇÃO DO KNN: CASO SIMPLES.....	79
7.2.2	CÓDIGO CASO SIMPLES .....	80
7.2.3	APLICAÇÃO DO KNN: CASO REAL.....	81
7.2.4	CÓDIGO CASO REAL.....	82
7.3	BIBLIOGRAFIA .....	83
8	NAÏVE BAYES.....	84
8.1	DESCRIÇÃO DO ALGORITMO NAÏVE BAYES .....	84
8.2	APLICAÇÃO DO ALGORITMO NAÏVE BAYES .....	85
8.2.1	CÓDIGO .....	86
8.3	BIBLIOGRAFIA .....	90
9	CART: CLASSIFICATION AND REGRESSION TREES .....	91
9.1	DESCRIÇÃO DO ALGORITMO CART .....	91
9.1.1	REGRA DE DIVISÃO .....	92
9.1.2	PODAGEM .....	93
9.2	APLICAÇÃO DO ALGORITMO CART .....	93
9.2.1	CÓDIGO .....	94
9.3	BIBLIOGRAFIA .....	99
10	REDE NEURAL FEEDFOWARD.....	100

10.1	UNIDADE DE PROCESSAMENTO .....	100
10.2	ARQUITETURAS DE REDES NEURAIS .....	102
10.2.1	ARQUITETURA FEEDFORWARD.....	102
10.3	PROCESSO DE TREINAMENTO .....	103
10.3.1	BACKPROPAGATION.....	103
10.4	APLICAÇÃO DA REDE NEURAL FEEDFORWARD .....	106
10.5	CÓDIGO.....	108
10.6	BIBLIOGRAFIA .....	110
11	REDE NEURAL CONVOLUCIONAL .....	112
11.1	OPERAÇÃO DE CONVOLUÇÃO .....	112
11.1.1	MOTIVAÇÃO PARA O USO DE CONVOLUÇÃO EM REDES NEURAIS 114	
11.2	POOLING (SUBSAMPLING).....	115
11.3	PROCESSO DE TREINAMENTO .....	117
11.4	APLICAÇÃO DA REDE NEURAL CONVOLUCIONAL .....	117
11.5	CÓDIGO.....	118
11.6	BIBLIOGRAFIA .....	123
12	REDE NEURAL RECORRENTE LSTM .....	124
12.1	LONG SHORT-TERM MEMORY .....	125
12.1.1	FORGET GATE .....	127
12.1.2	INPUT GATE .....	128
12.1.3	OUTPUT GATE .....	129
12.2	PROCESSO DE TREINAMENTO .....	129
12.3	APLICAÇÃO DE REDE NEURAL RECORRENTE LSTM.....	130
12.3.1	CÓDIGO .....	132
12.4	BIBLIOGRAFIA .....	136
13	SUPPORT VECTOR CLUSTERING (SVC) .....	137
13.1	ALGORITMO.....	137
13.1.1	DETERMINAÇÃO DA HIPERESFERA.....	137
13.1.2	DETERMINAÇÃO DOS CLUSTERS .....	140
13.1.3	AJUSTE DOS HIPERPARÂMETROS .....	141
13.2	APLICAÇÃO DO SVC .....	142
13.2.1	APLICAÇÃO SVC: CASO SIMPLES .....	142
13.2.2	APLICAÇÃO DO SVC: CASO REAL .....	143

13.2.3	CÓDIGOS.....	144
13.3	BIBLIOGRAFIA .....	149

## 1 C4.5

O C4.5 é um de algoritmo proposto por J. Ross Quinlan (QUINLAN, 2014) inspirado no sistema ID3 (QUINLAN, 1986) para indução de árvores de decisões. Este algoritmo pertence à classe de métodos de aprendizado supervisionado e é utilizado principalmente para problemas de classificação. Tal como no sistema ID3, a partir de uma base de dados rotulados / classificados  $D = \{x_i, y_i\}_{i=1}^N$ , o algoritmo C4.5 realiza a indução de uma árvore de decisão capaz de associar os diferentes valores de atributos  $x_i$  às diferentes classes  $y_i$ , gerando assim um modelo capaz de classificar novas instâncias de dados. Uma árvore de decisão pode ser interpretada como uma série de operações condicionais sistematicamente arranjada em formato de árvore utilizada para dividir os dados em subgrupos. Essas operações condicionais são chamadas de regras de decisão e permitem que os dados sejam divididos por meio de testes condicionais sobre seus valores de atributo. Quando sequenciadas de forma correta, essas operações acabam gerando subgrupos de dados de mesma classe, tornando-se assim possível identificar os valores condicionais dos atributos associados a cada uma das classes. Nesse contexto, o algoritmo C4.5 tem como principal virtude possibilitar a determinação do sequenciamento ótimo de operações condicionais capaz de induzir uma árvore de decisão apta para classificação de novos dados.

### 1.1 DESCRIÇÃO DO ALGORITMO C4.5

Seguindo o processo padrão dos métodos para indução de árvore de decisão, no algoritmo C4.5 a árvore é iniciada a partir de um nó raiz representado pela base de dados completo a ser analisada. A partir desse nó raiz, os dados são divididos em conjuntos menores de dados por meio de regras de decisão relacionadas a um atributo específico dos dados. Cada subconjunto gerado a partir dessa divisão denota um grupo de dados que satisfaz uma das regras definidas para o nó. Cada um desses subconjuntos é então definido como um nó filho da raiz, gerando assim ramos da árvore da decisão. Com a geração destes ramos, um processo semelhante de divisão de dados utilizado no nó raiz é aplicado nos nós filhos, resultando assim em novos nós e consequente em novos ramos. Este processo é recursivamente aplicado em cada novo ramo da árvore até que todos os nós gerados se tornem “puros”, ou seja, até que todos os dados presentes em cada subconjunto sejam da mesma classe. Quando um nó da árvore é considerado “puro”, este é definido com uma folha, pois determina o fim da ramificação de um ramo (WU; KUMAR, 2009). O algoritmo genérico para indução de uma árvore de decisão via C4.5 é descrito na Figura 1. Um componente chave nesse processo de indução de uma árvore de decisão é o método de escolha das regras de decisão a serem aplicadas em cada nó da árvore. No caso do C4.5 esse método se baseia em um método de minimização de entropia da informação (WU; KUMAR, 2009), no qual os atributos a serem utilizados nas regras de decisão são escolhidos por meio dos seguintes critérios da teoria de informação:

- **Ganho (*Gain*):** valor relacionado ao ganho de informação (redução de entropia), com relação aos dados, dada a escolha de um determinado atributo para regra de decisão.

$$Gain(a) = Entropy(a \text{ in } D) - \sum_v \frac{|D_v|}{|D|} Entropy(a \text{ in } D_v)$$

Onde  $a$  representa um atributo dos dados com  $v$  possíveis valores,  $D$  representa a base de dados analisada,  $D_v$  é um subconjunto dos dados com valor  $v$  para o atributo  $a$  e  $|\cdot|$  denota o tamanho dos conjuntos de dados (número de amostras). Já  $Entropy(a)$  representa a entropia do atributo  $a$  e é calculada por:

$$Entropy(a) = \sum_{i=1}^v -p_i \log_2 p_i$$

Tal que  $p_1, p_2, \dots, p_v$  denotam as probabilidades de cada um dos  $v$  possíveis valores para o atributo *Variable*.

- **Razão de Ganho (*GainRatio*):** valor também relacionado ao ganho de informação a partir da escolha de um determinado atributo para regra de decisão, porém corrige a tendência de o ganho favorecer atributos com muitos possíveis valores.

$$GainRatio(a) = \frac{Gain(a)}{Entropy(a)}$$

Figura 1 - Pseudocódigo do algoritmo C4.5

---

**Input:** an attribute-valued dataset  $D$

```

1: Tree = {}
2: if  $D$  is "pure" OR other stopping criteria met then
3:   terminate
4: end if
5: for all attribute  $a \in D$  do
6:   Compute information-theoretic criteria if we split on  $a$ 
7: end for
8:  $a_{best}$  = Best attribute according to above computed criteria
9: Tree = Create a decision node that tests  $a_{best}$  in the root
10:  $D_v$  = Induced sub-datasets from  $D$  based on  $a_{best}$ 
11: for all  $D_v$  do
12:   Tree $v$  = C4.5( $D_v$ )
13:   Attach Tree $v$  to the corresponding branch of Tree
14: end for
15: return Tree

```

---

Fonte: (WU; KUMAR, 2009)

Desta forma, durante o processo de indução da árvore, esses critérios são avaliados em cada novo nó gerado de forma a selecionar os atributos e regras de decisão que

geram maior ganho de informação para cada nó. Quando o atributo avaliado possui valores booleanos ou categóricos, regras de decisão para cada um dos valores possíveis do atributo são avaliados. Já quando o atributo que possui valores numéricos contínuos, opta-se pela avaliação de limites numéricos para as regras de decisão. Um problema comum no processo de indução de árvores de decisão é o sobre ajuste (*overfitting*) aos dados de treinamento, o qual pode afetar a capacidade de generalização da árvore e resultar em perdas de precisão do modelo ao classificar novos dados. Nesse contexto, a realização da poda da árvore é uma etapa crítica para a melhoria da precisão das classificações de novas amostras, pois mitigam a ocorrência de sobre ajuste (WU; KUMAR, 2009).

## 1.2 EXEMPLOS DE APLICAÇÃO DO ALGORITMO C4.5

Dois problemas de classificação de dados foram utilizados para demonstrar a aplicação prática do algoritmo C4.5. Para o primeiro caso implementado optou-se por utilizar uma simples base de dados gerada manualmente que permite gerar uma árvore de decisão com fácil interpretação da operação do algoritmo. Já para o segundo caso implementado optou-se pela análise de uma base de dados obtida em um banco de dados público com o objetivo de observar a atuação do algoritmo em um problema real de classificação.

### 1.2.1 APLICAÇÃO DO C4.5: CASO SIMPLES

Para ilustrar a aplicação do C4.5 em um problema simples optou-se pela análise de uma base de dados gerada manualmente. A base de dados gerada (Tabela 1) apresenta 6 amostras, cada uma com 2 atributos ( $x_1$  e  $x_2$ ) e uma classe binária ( $y$ ).

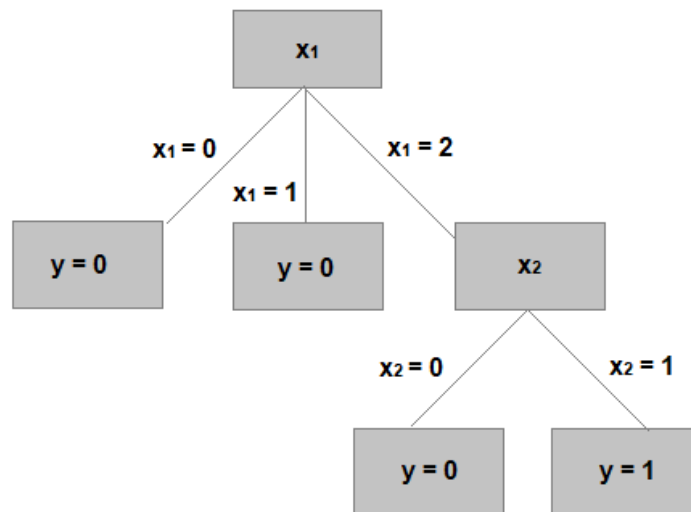
**Tabela 1 – Base de dados utilizada para a implementação do caso simples do C4.5**

Sample	Attributes		Class
	$x_1$	$x_2$	$y$
1	0	0	0
2	1	0	0
3	1	1	0
4	2	1	1
5	2	1	1
6	2	0	0

**Fonte: Autor**

Ao aplicar o algoritmo C4.5 para a base de dados da Tabela 1, uma árvore de decisão com 4 subconjuntos “puros” (folhas) é induzida (Figura 2). Analisando a árvore de decisão gerada, pode-se concluir que para todas as amostras em que  $x_1 = 0$  ou  $x_1 = 1$ , as amostras serão da classe 0. Já quando  $x_1 = 2$ , se  $x_2 = 0$  as amostras serão da classe 0, e se  $x_2 = 1$  as amostras serão da classe 1. Seguindo esta sequência lógica de testes, o algoritmo aplicado para a base de dados em questão se torna capaz de classificar novas amostras.

Figura 2 - Árvore de decisão gerada para a base de dados simples



Fonte: Autor

### 1.2.2 CÓDIGO CASO SIMPLES

```

import numpy as np

# Dados
x1 = [0, 1, 1, 2, 2, 2]
x2 = [0, 0, 1, 1, 1, 0]
y = np.array([0, 0, 0, 1, 1, 0])

# Função para split dos dados
def partition(a):
    return {c: (a==c).nonzero()[0] for c in np.unique(a)}

# Função para o cálculo da entropia
def entropy(s):
    res = 0
    val, counts = np.unique(s, return_counts = True)
    freqs = counts.astype('float')/len(s)
    for p in freqs:
        if p != 0.0:
            res -= p*np.log2(p)
    return res

# Função para o cálculo de ganho de informação (information gain)
def mutual_information(y, x):
    res = entropy(y)
    # Particionamos x de acordo com os valores dos atributos x_i
    val, counts = np.unique(x, return_counts = True)
    freqs = counts.astype('float')/len(x)
    # Calculamos uma média ponderada da entropia

```

```

for p, v in zip(freqs, val):
    res -= p*entropy(y[x == v])
return res

from pprint import pprint

def is_pure(s):
    return len(set(s)) == 1

def recursive_split(x, y):
    # Se não for possível realizar o split, retornar o set original
    if is_pure(y) or len(y) == 0:
        return y
    # Escolher o atributo que fornece o maior ganho de informação
    gain = np.array([mutual_information(y, x_attr) for x_attr in x.T])
    selected_attr = np.argmax(gain)
    # Se não houver ganho, nada deve ser feito e deve-
    se retornar o set original
    if np.all(gain<1e-6):
        return y
    # O split é realizado utilizando o atributo selecionado
    sets = partition(x[:, selected_attr])

    res = {}
    for k, v in sets.items():
        y_subset = y.take(v, axis = 0)
        x_subset = x.take(v, axis = 0)

        res["x_%d = %d" % (selected_attr + 1, k)] = recursive_split(x_subset
, y_subset)

    return res

X = np.array([x1,x2]).T
pprint(recursive_split(X, y))

```

Figura 3 - Código em Python da implementação do caso simples do C4.5

### 1.2.3 APLICAÇÃO DO C4.5: CASO REAL

Para ilustrar a aplicação do C4.5 para um problema de classificação real optou-se por utilizar uma base de dados do repositório público da Universidade da Califórnia Irvine (DUA; GRAFF, 2017) que contém diversas bases de dados para estudos práticos de aplicação de algoritmos de machine learning. A base de dados selecionada para esta aplicação consiste na Wine Quality Dataset (<https://archive.ics.uci.edu/ml/datasets/wine+quality>), a qual apresenta 4898 amostras de vinhos com 11 atributos (características físico-químicas e sensoriais) que descrevem a qualidade do vinho em uma escala de 0 a 10. O algoritmo C4.5 foi implementado para esta base de dados de forma a obter um modelo capaz de identificar se um vinho é de qualidade ou não. As



amostras de vinho de baixa qualidade (qualidade entre 0 e 6) são classificadas binariamente como 0 e as amostras de vinho de alta qualidade (qualidade > 7) são classificadas binariamente como 1. Com intuito de verificar a eficiência do algoritmo na classificação de novas amostras, a base de dados foi dividida em dois conjuntos, sendo um conjunto de dados para treinamento e outro conjunto de dados de teste. Ao testar o algoritmo para o conjunto de dados de teste verificou-se em todos os testes uma taxa de acerto acima de 80%, o qual é um valor satisfatório para o caso em questão. A realização da roda da árvore de decisão gerada pode ser utilizada para aumento da eficiência do algoritmo.

**Tabela 2 - Taxa de precisão do C4.5 implementado para base de dados de qualidade de vinho**

Teste	1	2	3	4	5	6	7	8	9	10	Média
Taxa de Acerto	84,90%	83,27%	82,24%	82,65%	87,14%	83,67%	81,22%	83,27%	82,82%	84,05%	83,52%

### 1.2.4 CÓDIGO CASO REAL

```
#####
# Importação dos módulos de função #
#####

from __future__ import division # Permite utilizar funcionalidades futuras d
o python
import math # Funções matemáticas
import operator # Funções intrínsecas de operações matemáticas
import copy # Possibilita a realização de cópias de objetos
import csv # Possibilita a manipulação de dados no formato .csv
import time # Funções de tempo/data
import random # Funções de randomização
from collections import Counter # Cria um dicionário com a frequência dos el
ementos de um objeto

#####
# Classe csvdata para o armazenamento de dados .csv #
#####
class csvdata():
    def __init__(self, classifier):
        self.rows = []
        self.attributes = []
        self.attribute_types = []
        self.classifier = classifier
        self.class_col_index = None

#####
# Classe decisionTreeNode para construção da árvore de decisão #
#####
class decisionTreeNode():
    def __init__(self, is_leaf_node, classification, attribute_split_index,
attribute_split_value, parent, left_child, right_child, height):
        self.is_leaf_node = True
```

```

self.classification = None
self.attribute_split = None
self.attribute_split_index = None
self.attribute_split_value = None
self.parent = parent
self.left_child = None
self.right_child = None
self.height = None

#####
# Pré-processamento dos dados #
#####
def preprocessing(dataset):
    # Conversão dos atributos numéricos em float. 'True' = Numérico e 'False'
    # = Discreto
    for example in dataset.rows:
        for x in range(len(dataset.rows[0])):
            if dataset.attributes[x] == 'True':
                example[x] = float(example[x])

#####
# Construção da árvore de decisão de forma recursiva #
#####
def compute_decision_tree(dataset, parent_node, classifier):
    # Primeiro criar um nó da árvore
    node = decisionTreeNode(True, None, None, None, parent_node, None, None,
0)
    # Cálculo da altura da árvore
    if (parent_node == None):
        node.height = 0
    else:
        node.height = node.parent.height + 1
    # Checar se os dados do nó são puros
    ones = count_positives(dataset.rows, dataset.attributes, classifier) # c
ount_positives() irá contar o número de exemplos (rows) com classificação '1'

    if (len(dataset.rows) == ones):
        node.classification = 1
        node.is_leaf_node = True
        return node
    elif (ones == 0):
        node.classification = 0
        node.is_leaf_node = True
        return node
    else:
        node.is_leaf_node = False

    # Definir o melhor atributo para o split
    splitting_attribute = None

```

```

# O ganho de informação fornecido pelo melhor atributo
maximum_info_gain = 0
# Limite condicional
split_val = None
# O mínimo valor de ganho de informação permitido
minimum_info_gain = 0.01
# Cálculo da entropia do dataset
entropy = calculate_entropy(dataset, classifier)
for attr_index in range(len(dataset.rows[0])):
    if (dataset.attributes[attr_index] != classifier):
        local_max_gain = 0
        local_split_val = None
        attr_value_list = [example[attr_index] for example in dataset.ro
ws] # Dados que serão splitados
        attr_value_list = list(set(attr_value_list)) # Remove valores du
plicados de atributos
        # Caso o atributo for numérico, definir as condições limite para
o split
        if (len(attr_value_list) > 100):
            attr_value_list = sorted(attr_value_list)
            total = len(attr_value_list)
            ten_percentile = int(total/10)
            new_list = []
            for x in range(1, 10):
                new_list.append(attr_value_list[x*ten_percentile])
            attr_value_list = new_list
        # Definição do melhor valor de val
        for val in attr_value_list:
            # Calcular o valor de ganho utilizando este valor de limite
            # Se for maior que local_split_val, salvar este valor na mes
ma variável
            current_gain = calculate_information_gain(attr_index, datase
t, val, entropy)
            if (current_gain > local_max_gain):
                local_max_gain = current_gain
                local_split_val = val
        # Definição do melhor atributo
        if (local_max_gain > maximum_info_gain):
            maximum_info_gain = local_max_gain
            split_val = local_split_val
            splitting_attribute = attr_index

# Classificação do nó para casos quase puros
if (maximum_info_gain <= minimum_info_gain or node.height > 20):
    node.is_leaf_node = True
    node.classification = classify_leaf(dataset, classifier)
    return node

# Informações do nó (leaf) formado

```

```

node.attribute_split_index = splitting_attribute
node.attribute_split = dataset.attributes[splitting_attribute]
node.attribute_split_value = split_val

# Construção dos ramos após o split
left_dataset = csvdata(classifier)
right_dataset = csvdata(classifier)
left_dataset.attributes = dataset.attributes
right_dataset.attributes = dataset.attributes
left_dataset.attribute_types = dataset.attribute_types
right_dataset.attribute_types = dataset.attribute_types

# Alocação dos dados para cada ramo criado
for row in dataset.rows:
    if (splitting_attribute is not None and row[splitting_attribute] >=
split_val):
        left_dataset.rows.append(row)
    elif (splitting_attribute is not None and row[splitting_attribute] <
split_val):
        right_dataset.rows.append(row)

# Recursion
node.left_child = compute_decision_tree(left_dataset, node, classifier)
node.right_child = compute_decision_tree(right_dataset, node, classifier
)

return node

#####
# Função para classificação da folha (nó) #
#####
def classify_leaf(dataset, classifier):
    ones = count_positives(dataset.rows, dataset.attributes, classifier)
    total = len(dataset.rows)
    zeroes = total - ones
    if (ones >= zeroes):
        return 1
    else:
        return 0

#####
# Avaliação final dos dados #
#####
def get_classification(example, node, class_col_index):
    if (node.is_leaf_node == True):
        return node.classification
    else:
        if (example[node.attribute_split_index] >= node.attribute_split_valu
e):

```

```

        return get_classification(example, node.left_child, class_col_index)
    else:
        return get_classification(example, node.right_child, class_col_index)

#####
# Cálculo da entropia do dataset #
#####
def calculate_entropy(dataset, classifier):

    # Contagem de rows com classificação '1'
    ones = count_positives(dataset.rows, dataset.attributes, classifier)
    # Cálculo do número de rows
    total_rows = len(dataset.rows)
    # A entropia é calculada pela fórmula somatória de -
    # p*log2(p), onde p é a probabilidade de certa classificação
    entropy = 0
    # Probabilidade p de classificação '1' no dataset total
    p = ones/total_rows
    if (p != 0):
        entropy += p*math.log(p, 2)
    # Probabilidade p de classificação '0' no dataset total
    p = (total_rows - ones)/total_rows
    if (p != 0):
        entropy += p*math.log(p, 2)
    entropy = -entropy
    return entropy

#####
# Cálculo de ganho de informação #
#####
def calculate_information_gain(attr_index, dataset, val, entropy):
    classifier = dataset.attributes[attr_index]
    attr_entropy = 0
    total_rows = len(dataset.rows)
    #criando dois possíveis ramos da árvore
    gain_upper_dataset = csvdata(classifier)
    gain_lower_dataset = csvdata(classifier)
    gain_upper_dataset.attributes = dataset.attributes
    gain_lower_dataset.attributes = dataset.attributes
    gain_upper_dataset.attribute_types = dataset.attribute_types
    gain_lower_dataset.attribute_types = dataset.attribute_types
    # split de acordo com val
    for example in dataset.rows:
        if (example[attr_index] >= val):
            gain_upper_dataset.rows.append(example)
        elif (example[attr_index] < val):
            gain_lower_dataset.rows.append(example)

```

```

    if (len(gain_upper_dataset.rows) == 0 or len(gain_lower_dataset.rows) ==
0):
        return -1

    # Cálculo da entropia do atributo utilizado
    attr_entropy += calculate_entropy(gain_upper_dataset, classifier)*len(ga
in_upper_dataset.rows)/total_rows
    attr_entropy += calculate_entropy(gain_lower_dataset, classifier)*len(ga
in_lower_dataset.rows)/total_rows

    return entropy - attr_entropy

#####
# Contador de rows com classificação '1' #
#####
def count_positives(instances, attributes, classifier):
    count = 0
    class_col_index = None
    # Achar o índice do classificador
    for a in range(len(attributes)):
        if attributes[a] == classifier:
            class_col_index = a
        else:
            class_col_index = len(attributes) - 1
    # Contagem de '1's
    for i in instances:
        if i[class_col_index] == '1':
            count += 1
    return count

#####
# Validação da árvore #
#####
def validate_tree(node, dataset):
    total = len(dataset.rows)
    correct = 0
    for row in dataset.rows:
        #Validação de exemplo (row)
        correct += validate_row(node, row)
    return correct/total

#####
# Validação do exemplo (row) #
#####
# Para achar o melhor score antes de podar a árvore
def validate_row(node, row):
    if (node.is_leaf_node == True):
        projected = node.classification

```

```

        actual = int(row[-1])
        if (projected == actual):
            return 1
        else:
            return 0
    value = row[node.attribute_split_index]
    if (value >= node.attribute_split_value):
        return validate_row(node.left_child, row)
    else:
        return validate_row(node.right_child, row)

#####
# Poda (Prune) da árvore #
#####
def prune_tree(root, node, validate_set, best_score):
    # Se o nó for uma folha
    if (node.is_leaf_node == True):
        # classification = node.classification
        node.parent.is_leaf_node = True
        node.parent.classification = node.classification
        if (node.height < 20):
            new_score = validate_tree(root, validate_set)
        else:
            new_score = 0
        if (new_score >= best_score):
            return new_score
        else:
            node.parent.is_leaf_node = False
            node.parent.classification = None
            return best_score
    # Se o nó não for uma folha
    else:
        new_score = prune_tree(root, node.left_child, validate_set, best_score)
        if (node.is_leaf_node == True):
            return new_score
        new_score = prune_tree(root, node.right_child, validate_set, new_score)
        if (node.is_leaf_node == True):
            return new_score
        return new_score

#####
# Programa principal para rodar a árvore #
#####
def run_decision_tree():

    # Dados a serem utilizados
    dataset = csvdata('')

```

```

training_set = csvdata('')
test_set = csvdata('')

# Carregar o dados
f = open('wine_quality_dataset.csv')
original_file = f.read()
# Tratar os dados
rowsplit_data = original_file.splitlines()
dataset.rows = [rows.split(',') for rows in rowsplit_data]
dataset.attributes = dataset.rows.pop(0)
# Printar atributos
print("Attributes:")
print(dataset.attributes)

# Definição dos tipos de atributos (Numérico == 'true' e Nominal == 'false')
# Para cada caso deve ser alterado
dataset.attribute_types = ['true', 'true', 'true', 'true', 'true', 'true',
', 'true', 'true', 'true', 'true', 'true', 'false']

# Definir a classe
classifier = dataset.attributes[-1]
dataset.classifier = classifier

# Achar o índice da classe
for a in range(len(dataset.attributes)):
    if (dataset.attributes[a] == dataset.classifier):
        dataset.class_col_index = a
    else:
        dataset.class_col_index = len(dataset.attributes) - 1

# Printar qual é a classe
print(f'Classifier is {dataset.attributes[dataset.class_col_index]} (Index: {dataset.class_col_index})')

# Pré-processamento dos dados
preprocessing(dataset)

# Dados para treinamento, teste e validação
training_set = copy.deepcopy(dataset)
training_set.rows = []
test_set = copy.deepcopy(dataset)
test_set.rows = []
validate_set = copy.deepcopy(dataset)
validate_set.rows = []

# Caso realizar poda (pruning), ativar código abaixo
# Criar o dataset para validação para pós poda (post pruning)
# dataset.rows = [x for i, x in enumerate(dataset.rows) if i % 10 != 9]

```



```

#validate_set.rows = [x for i, x in enumerate(dataset.rows) if i % 10 ==
9]

# Número de runs a serem realizadas
K = 10
# Armazenar a precisão (accuracy) das 10 runs
accuracy = []
start = time.clock()

for k in range(K):
    print('Doing fold', k)
    # Parece estar criando novos datasets para treino e teste
    training_set.rows = [x for i, x in enumerate(dataset.rows) if i % K
!= k]
    test_set.rows = [x for i, x in enumerate(dataset.rows) if i % K == k
]

    # Printar quantos exemplos para treino e teste são criados
    print("Number of training records: %d" % len(training_set.rows))
    print("Number of test records: %d" % len(test_set.rows))

    # Construção da árvore
    root = compute_decision_tree(training_set, None, classifier)

    # Teste da árvore
    # Classificar os dados de teste usando a árvore construída
    results = []
    for instance in test_set.rows:
        result = get_classification(instance, root, test_set.class_col_i
ndex)
        results.append(str(result) == str(instance[-1]))

    # Cálculo da precisão (Accuracy)
    acc = float(results.count(True))/float(len(results))
    print("Accuracy: %.4f" % acc)

    # Se desejar, ativar o código de poda abaixo.
    # best_score = validate_tree(root, validate_set)
    # post_prune_accuracy = 100*prune_tree(root, root, validate_set, bes
t_score)
    # print ("Post-
pruning score on validation set: " + str(post_prune_accuracy) + "%")

    accuracy.append(acc)
    del root

mean_accuracy = math.fsum(accuracy)/K
print('Final results:')
print("Accuracy  %f " % (mean_accuracy))
print("Took %f secs" % (time.clock() - start))

```

```
# Cria um arquivo de resultados
f = open("result.txt", "w")
f.write("accuracy: %.4f" % mean_accuracy)
f.close()

#####
# Rodar algoritmo #
#####

if __name__ == "__main__":
    run_decision_tree()
```

Figura 4 - Código em Python da implementação do C4.5 para uma base de dados real

### 1.3 BIBLIOGRAFIA

DUA, Dheeru; GRAFF, Casey. UCI machine learning repository. 2017.

QUINLAN, J. R. C4.5: programs for machine learning. [s.l.] Elsevier, 2014

WU, Xindong; KUMAR, Vipin (Ed.). The top ten algorithms in data mining. CRC press, 2009

## 2 K-MEANS

O algoritmo k-Means faz parte da família de algoritmos de aprendizado não supervisionado utilizados para problemas de *clustering* (agrupamento) de dados. Dado um conjunto de dados não rotulados  $D = \{x_i\}_{i=1}^N$ , o objetivo em um problema de clustering é dividir os objetos  $x_i$  em subconjuntos (os chamados *clusters*) de acordo com valores de similaridade entre os objetos, ou seja, objetos similares são alocados em um mesmo *cluster* enquanto objetos poucos similares são alocados em *clusters* distintos. O algoritmo k-Means consiste em um simples algoritmo iterativo de *clustering* que particiona uma base de dados em um número especificado de  $k$  clusters baseado na determinação das localizações ótimas de  $k$  centroides. O k-Means é um algoritmo simples de ser implementado, relativamente rápido e de ampla utilização. É historicamente um dos mais importantes algoritmos de mineração de dados (WU; KUMAR, 2009).

### 2.1 DESCRIÇÃO DO ALGORITMO

Dada uma base de dados  $D = \{x_i\}_{i=1}^N$ , na qual cada objeto  $x_i$  consiste em um vetor de dimensão  $d$ , o algoritmo k-Means tem como função realizar o particionamento da base de dados em  $k$  clusters de tal forma que cada objeto  $x_i$  é alocado exclusivamente para um dos  $k$  clusters, não podendo ser aplicado para mais de um cluster. O número de clusters  $k$  é um parâmetro de entrada do algoritmo e normalmente é definido baseado em conhecimentos prévios sobre a base de dados ou em um número desejado de clusters. No k-Means, cada um dos  $k$  clusters é representado por um único vetor de  $d$  dimensões, chamado centroide, o qual consiste normalmente no ponto médio do cluster ao qual é representante. O conjunto desses pontos centroides é denotado por  $C = \{c_i\}_{i=1}^k$  e desempenha um importante papel para a avaliação de proximidade ou similaridade entre os objetos da base de dados. Utilizando a distância Euclidiana como medida de similaridade entre dois pontos, o agrupamento dos dados via algoritmo k-Means se baseia na minimização da seguinte função de custo:

$$Cost = \sum_{i=1}^N \left( \operatorname{argmin}_j \|x_i - c_j\|_2^2 \right)$$

Ou seja, k-Means realiza o agrupamento dos dados por meio da minimização da soma das distâncias Euclidianas entre os pontos  $x_i$  e o centroide  $c_j$  do cluster mais próximo a cada um desses pontos.

O pseudocódigo do algoritmo k-Means é apresentado na Figura 5. Como se pode notar, o primeiro passo do algoritmo é iniciar os representantes dos clusters selecionando  $k$  pontos de  $\mathbb{R}^d$ . Técnicas para selecionar estes pontos iniciais incluem escolhas randômicas de pontos da base de dados ou perturbação da média global dos dados  $k$  vezes. Após a escolha destes  $k$  pontos, a etapa seguinte é iterar o algoritmo entre dois passos:

**Passo 1: Alocação dos dados.** Cada ponto é alocado para o seu centroide mais próximo, resultando na partição dos dados. Em caso de pontos com distâncias iguais para dois centroides, um centroide é escolhido de forma arbitrária.

**Passo 2: Realocação dos centroides.** Cada centroide de cluster é realocado para o centro (média aritmética) de todos os dados alocados para o cluster em questão. A razão por trás deste passo é baseada na observação de que, dado um conjunto de dados, o melhor representante deste conjunto é nada mais que a média do conjunto destes dados.

---

```

Input: Dataset  $D$ , number clusters  $k$ 
Output: Set of cluster representatives  $C$ , cluster membership vector  $m$ 
  /* Initialize cluster representatives  $C$  */
  Randomly choose  $k$  data points from  $D$ 
5: Use these  $k$  points as initial set of cluster representatives  $C$ 
  repeat
    /* Data Assignment */
    Reassign points in  $D$  to closest cluster mean
    Update  $m$  such that  $m_i$  is cluster ID of  $i$ th point in  $D$ 
10: /* Relocation of means */
    Update  $C$  such that  $c_j$  is mean of points in  $j$ th cluster
  until convergence of objective function  $\sum_{i=1}^N (\argmin_j ||x_i - c_j||_2^2)$ 

```

---

Figura 5 - Pseudocódigo do algoritmo k-Means (WU; KUMAR, 2009).

O algoritmo converge quando os deslocamentos dos centroides não ocorrem mais. Devido a sua natureza de tentar minimizar um custo não convexo, o k-Means tende a convergir para um ótimo local, ou seja, iniciar o conjunto de representantes clusters  $C$  de diferentes formas pode resultar em clusters diferentes mesmo para uma mesma base de dados  $D$ .

## 2.2 APLICAÇÃO DO ALGORITMO K-MEANS

Para a implementação do algoritmo k-Means optou-se por utilizar uma base de dados reais que apresenta as coordenadas de localização de fontes de água potável presentes na cidade de Vancouver ([https://data.vancouver.ca/datacatalogue/drinking Fountains.htm](https://data.vancouver.ca/datacatalogue/drinking-Fountains.htm)). A ideia da utilização do algoritmo k-Means para esta base de dados é identificar diferentes regiões onde estão localizadas as fontes de água. Para implementar o algoritmo, os passos apresentados na Figura 5 foram seguidos. Os resultados da implementação do algoritmo são apresentados nas figuras a seguir.

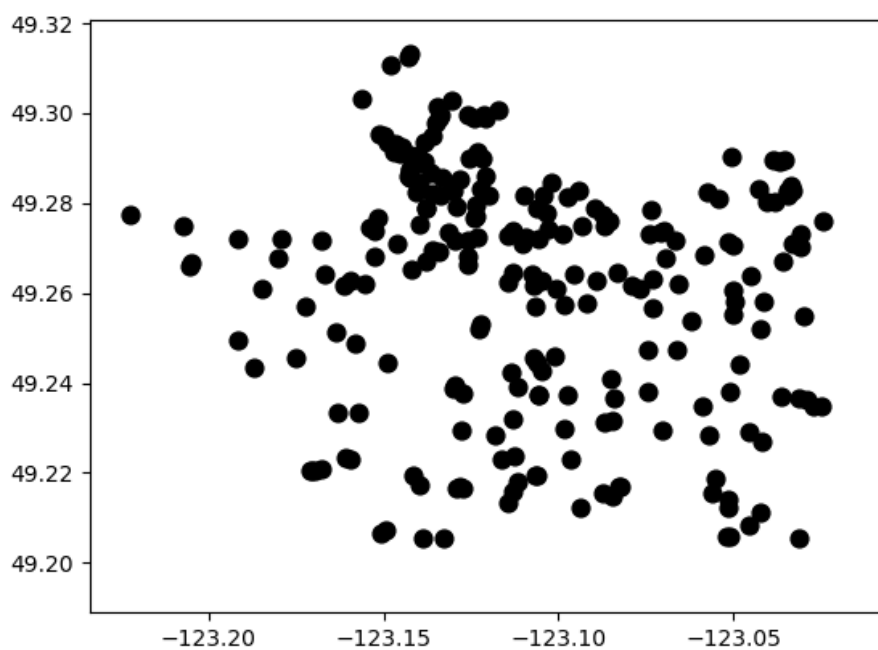


Figura 6 - Base de dados para implementação do k-Means

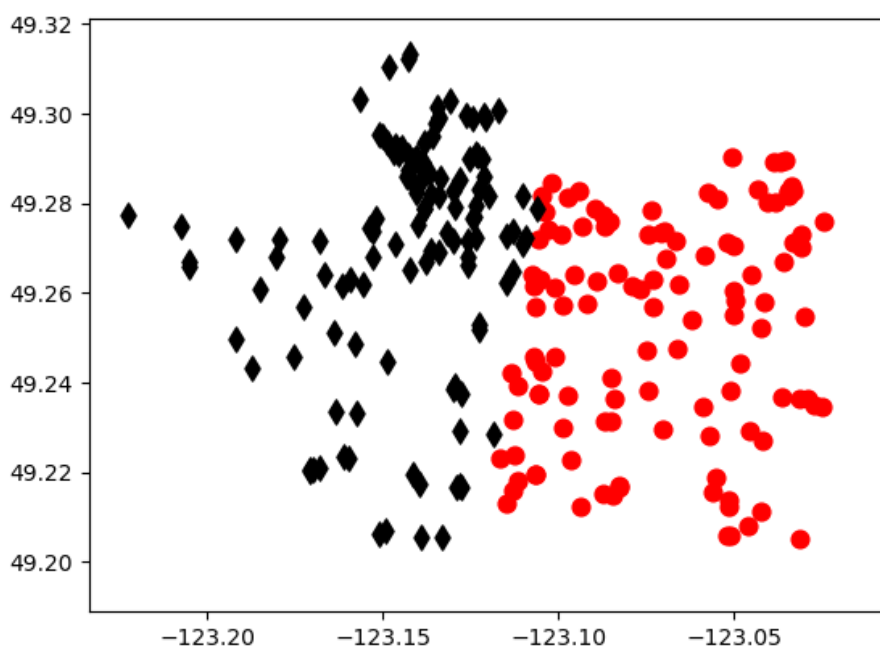


Figura 7 - k-Means ( $k = 2$ )

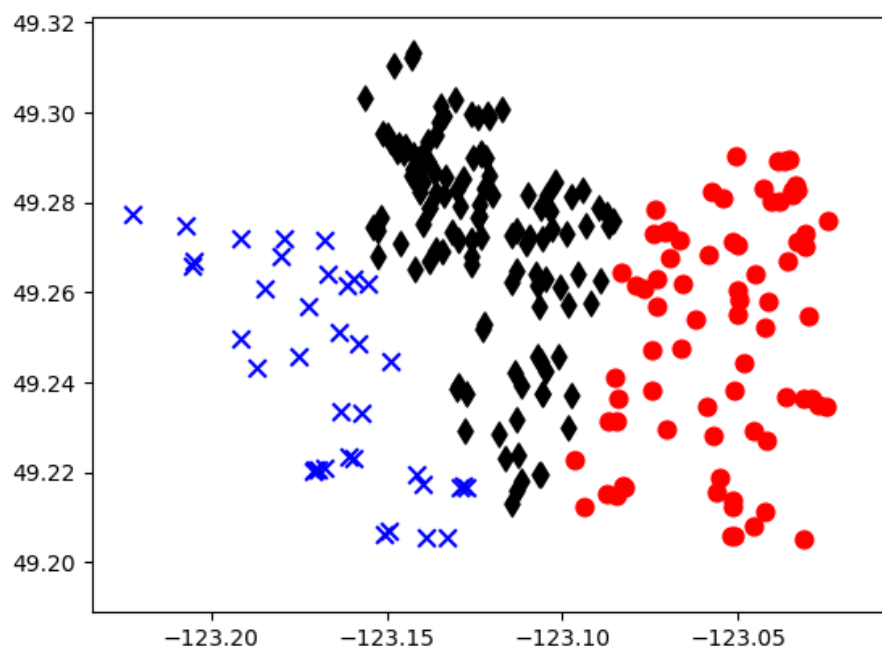


Figura 8 - k-Means ( $k = 3$ )

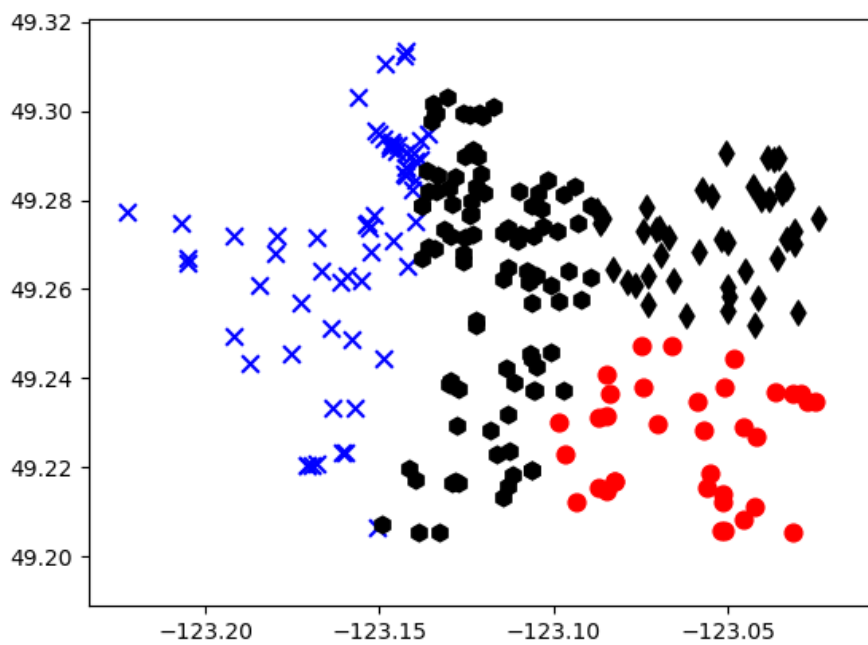


Figura 9 - k-Means ( $k = 4$ )

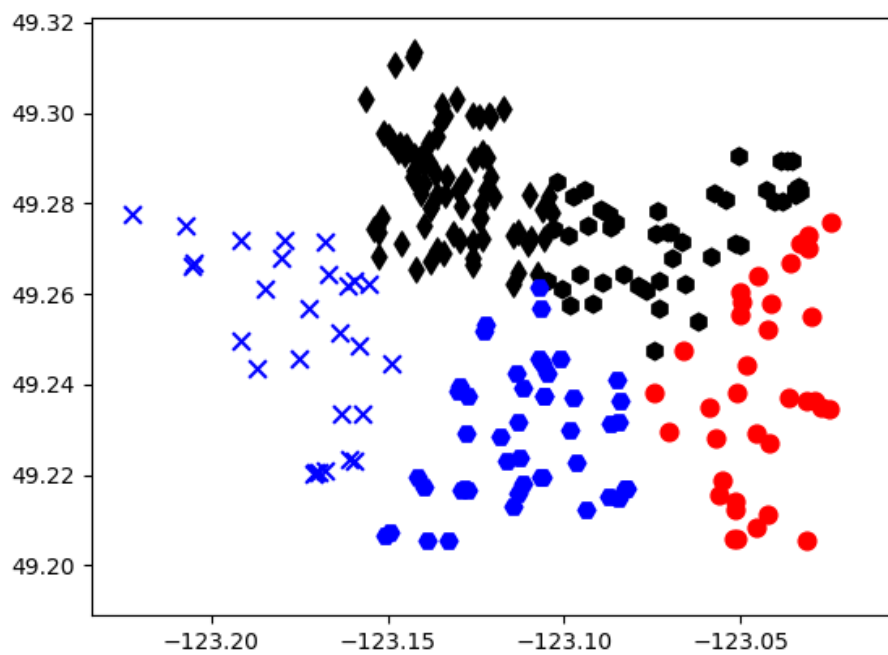


Figura 10 - k-Means ( $k = 5$ )

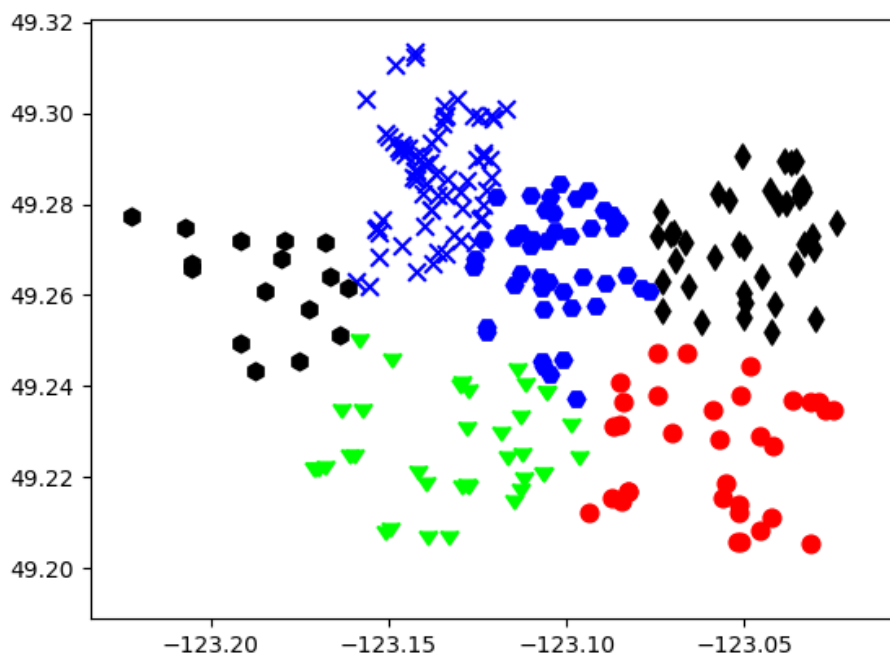


Figura 11 - k-Means ( $k = 6$ )

### 2.2.1 CÓDIGO

```

import random as rand
import math as math
import numpy as np
from mpl_toolkits.mplot3d import Axes3D
import matplotlib.pyplot as plt
import csv
from point import Point

class clustering:
    def __init__(self, geo_loc, k):
        self.geo_locations = geo_loc
        self.k = k
        self.clusters = [] #Clusters
        self.means = [] #Cluster's means
        self.debug = False #Debug flag

    # Função para a escolha dos 7 outros centróides após aleatoriamente escolher o primeiro
    def next_random(self, index, points, clusters):
        dist = {}
        for point_1 in points:
            if self.debug:
                print('point 1: %f%f' %(point_1.latit, point_1.longit))
            for cluster in clusters.values():
                point_2 = cluster[0]
                if self.debug:
                    print('point 2: %f%f' %(point_2.latit, point_2.longit))
                if point_1 not in dist:
                    dist[point_1] = math.sqrt(math.pow(point_1.latit - point_2.latit, 2.0) + math.pow(point_1.longit - point_2.longit, 2.0))
                else:
                    dist[point_1] += math.sqrt(math.pow(point_1.latit - point_2.latit, 2.0) + math.pow(point_1.longit - point_2.longit, 2.0))
            if self.debug:
                for key, value in dist.items():
                    print('( %f%f ) ==> %f' %(key.latit, key.longit, value))
        count_ = 0
        max_ = 0
        for key, value in dist.items():
            if count_ == 0:
                max_ = value
                max_point = key
                count_ += 1
            else:
                if value > max_:
                    max_ = value

```



```

        max_point = key
    return max_point

# Função para escolha de um centróide aleatório
def initial_means(self, points):
    point_ = rand.choice(points)
    if self.debug:
        print('point#0: %f %f' %(point_.latit, point_.longit))
    clusters = dict()
    clusters.setdefault(0, []).append(point_)
    points.remove(point_)
    for i in range(1, self.k):
        point_ = self.next_random(i, points, clusters)
        if self.debug:
            print('point#%d: %f %f' % (i, point_.latit, point_.longit))
        clusters.setdefault(i, []).append(point_)
        points.remove(point_)
    self.means = self.compute_mean(clusters)
    if self.debug:
        print("initial means:")
        self.print_means(self.means)

# Função para definição dos centróides
def compute_mean(self, clusters):
    means = []
    for cluster in clusters.values():
        mean_point = Point(0.0, 0.0)
        cnt = 0.0
        for point in cluster:
            mean_point.latit += point.latit
            mean_point.longit += point.longit
            cnt += 1.0
        mean_point.latit = mean_point.latit/cnt
        mean_point.longit = mean_point.longit/cnt
        means.append(mean_point)
    return means

# Função para alocação dos pontos para os centróides mais próximos
def assign_points(self, points):
    if self.debug:
        print('assign points')
    clusters = dict()
    for point in points:
        dist = []
        if self.debug:
            print("point(%f,%f)" % (point.latit, point.longit))
        for mean in self.means:
            dist.append(math.sqrt(math.pow(point.latit - mean.latit,2.0)
+ math.pow(point.longit - mean.longit,2.0)))

```

```

        if self.debug:
            print (dist)
        cnt_ = 0
        index = 0
        min_ = dist[0]
        for d in dist:
            if d < min_:
                min_ = d
                index = cnt_
            cnt_ += 1
        if self.debug:
            print ("index: %d" % index)
        clusters.setdefault(index, []).append(point)
    return clusters

# Função para checar convergência dos centróides
def update_means(self, means, threshold):
    for i in range(len(self.means)):
        mean_1 = self.means[i]
        mean_2 = means[i]
        if self.debug:
            print ("mean_1(%f,%f)" % (mean_1.latit, mean_1.longit))
            print ("mean_2(%f,%f)" % (mean_2.latit, mean_2.longit))
        if math.sqrt(math.pow(mean_1.latit - mean_2.latit,2.0) + math.pow(mean_1.longit - mean_2.longit,2.0)) > threshold:
            return False
    return True

# Função para o print dos clusters
def print_clusters(self, clusters):
    cluster_cnt = 1
    for cluster in clusters.values():
        print("nodes in cluster #%d" % cluster_cnt)
        cluster_cnt += 1
        for point in cluster:
            print ("point(%f,%f)" % (point.latit, point.longit))

# Função para o print dos valores de centróide
def print_means(self, means):
    for point in means:
        print("%f %f" % (point.latit, point.longit))

# Função do algoritmo de clustering k-means
def k_means(self, plot_flag):
    if len(self.geo_locations) < self.k:
        return -1 # error
    points_ = [point for point in self.geo_locations]
    # Computar os centróides iniciais
    self.initial_means(points_)

```

```

stop = False
while not stop:
    # Alocar cada ponto para o centróide mais próximo
    points_ = [point for point in self.geo_locations]
    clusters = self.assign_points(points_)
    if self.debug:
        self.print_clusters(clusters)
    means = self.compute_mean(clusters)
    if self.debug:
        print ("means:")
        print (self.print_means(means))
        print ("update mean:")
    stop = self.update_means(means, 0.01)
    if not stop:
        self.means = []
        self.means = means
self.clusters = clusters
# Plot dos clusters para avaliação
if plot_flag:
    fig = plt.figure()
    ax = fig.add_subplot(111)
    markers = ['o', 'd', 'x', 'h', 'H', 7, 4, 5, 6, '8', 'p', ',', ' ',
+',', '.', 's', '*', 3, 0, 1, 2]
    colors = ['r', 'k', 'b', [0,0,0], [0,0,1], [0,1,0], [0,1,1], [1,
0,0], [1,0,1], [1,1,0], [1,1,1]]
    cnt = 0
    for cluster in clusters.values():
        latits = []
        longits = []
        for point in cluster:
            latits.append(point.latit)
            longits.append(point.longit)
        ax.scatter(longits, latits, s=60, c=colors[cnt], marker=mark
ers[cnt])
        cnt += 1
    plt.show()
return 0

# Main()

geo_locs = []
# Leitura do arquivo csv constendo a localização das fontes
f = open('drinkingFountains.csv', 'r')
reader = csv.reader(f, delimiter=",")
# Salvar a localização de cada fonte como um objeto Point(latit, longit)
for line in reader:
    loc_ = Point(float(line[0]), float(line[1])) # tuples for location
    geo_locs.append(loc_)

```

```
# Comandos para debug
#print len(geo_locs)
#for p in geo_locs:
#    print "%f %f" % (p.latit, p.longit)

# Rodar o algoritmo k_means para realizar o clustering
cluster = clustering(geo_locs, 6 )
flag = cluster.k_means(True)
if flag == -1:
    print ("Error in arguments!")
else:
    #Printar o resultado do clustering
    print ("clustering results:")
    cluster.print_clusters(cluster.clusters
```

Figura 12 - Código em Python para implementação do k-Means

## 2.3 BIBLIOGRAFIA

JAIN, A. K.; DUBES, R. C. **Algorithms for clustering data**. [s.l.] Prentice-Hall, Inc., 1988.

WU, Xindong; KUMAR, Vipin (Ed.). The top ten algorithms in data mining. CRC press, 2009

### 3 SVM: SUPPORT VECTOR MACHINES

As máquinas de vetores suporte (support vector machines – SVMs) são compostas por classificadores de vetores suporte (support vector classifier – SVC) e regressores de vetores suporte (support vector regressor – SVR), e estão entre os mais robustos e precisos métodos aplicados para mineração de dados. SVMs foram originalmente desenvolvidos pelo Vapnik em meados de 1990 com base teórica fundada na teoria de aprendizado estatístico, exigindo apenas um número reduzido de amostras para treinamento e apresentando uma eficiência independentemente do número de dimensões das amostras. Este trabalho foca na apresentação do SVC (WU; KUMAR, 2009).

#### 3.1 SVC: SUPPORT VECTOR CLASSIFIER

Para uma base de dados com duas classes linearmente separáveis, o objetivo do SVC é achar um hiperplano capaz de separar as duas classes das amostras com uma margem máxima que possa oferecer a melhor capacidade de generalização. Capacidade de generalização se refere à característica de um classificador não apenas ter um bom desempenho de classificação dos dados de treinamento, mas também garantir uma elevada precisão preditiva para dados futuros (WU; KUMAR, 2009).

Intuitivamente, a margem pode ser definida como a quantidade de espaço, ou separação, entre as duas classes definida por um hiperplano. Geometricamente, a margem corresponde a menor distância entre os pontos mais próximos a qualquer ponto do hiperplano. A Figura 13 ilustra a construção geométrica de um hiperplano ótimo seguindo as condições para uma entrada de dados bidimensional.

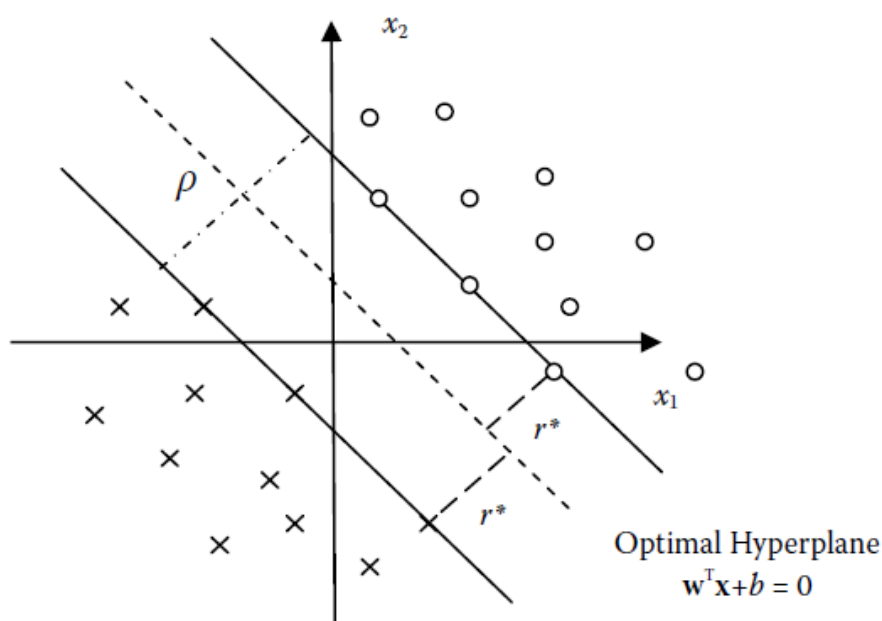


Figura 13 - Ilustração do hiperplano ótimo em SVC para um caso linearmente separável (WU; KUMAR, 2009)

Sendo  $w$  e  $b$  os pesos dos vetores e o bias de um hiperplano ótimo, respectivamente, o hiperplano correspondente pode ser definido como:

$$w^T x + b = 0$$

A distância geométrica direcional desejada da amostra  $x$  até o hiperplano ótimo é:

$$r = \frac{g(x)}{\|w\|}$$

Onde  $g(x) = w^T x + b$  é a função discriminante definida pelo hiperplano e também chamada de margem funcional de  $x$  dado  $w$  e  $b$ .

Consequentemente, SVC visa achar os parâmetros  $w$  e  $b$  para um hiperplano ótimo com o intuito de maximizar a margem de separação ( $\rho$ ) que é determinada pelas menores distâncias geométrica  $r^*$  das duas classes, respectivamente. Agora sem perda de generalidade, a margem funcional é definida como igual a 1, ou seja, dado uma base de dados de treinamento  $\{x_i, y_i\}_{i=1}^n \in \mathbb{R}^m \times \{\pm 1\}$ , obtém-se:

$$w^T x + b \geq 1 \text{ para } y_i = +1$$

$$w^T x + b \leq -1 \text{ para } y_i = -1$$

Os pontos  $(x_i, y_i)$  em que as igualdades entre a primeira e segunda parte nas equações acima são satisfeitas são chamados de vetores suporte (support vectors), os quais são exatamente os pontos mais próximos do hiperplano ótimo. Portanto, a distância geométrica correspondente ao vetor suporte  $x^*$  até o hiperplano ótimo é:

$$r^* = \frac{g(x^*)}{\|w\|} \begin{cases} \frac{1}{\|w\|} & \text{se } y^* = +1 \\ -\frac{1}{\|w\|} & \text{se } y^* = -1 \end{cases}$$

Pela Figura 13, claramente a margem de separação  $\rho$  é:

$$\rho = 2r^* = \frac{2}{\|w\|}$$

Para garantir que a máxima margem do hiperplano possa ser determinada, o SVC tenta maximizar  $\rho$  com respeito à  $w$  e  $b$ :

$$\begin{aligned} & \max_{w, b} \frac{2}{\|w\|} \\ & \text{s. t. } y_i(w^T x_i + b) \geq 1, \quad i = 1, \dots, n \end{aligned}$$

De forma equivalente:

$$\min_{w, b} \frac{1}{2} \|w\|^2$$

$$s. t. \ y_i(w^T x_i + b) \geq 1, \quad i = 1, \dots, n$$

Normalmente se utiliza  $\|w\|^2$  invés de  $\|w\|$  por conveniência de forma a facilitar os passos subsequentes de otimização. Geralmente, o problema de otimização limitado da equação anterior, conhecido como o problema primal, é resolvido por meio do método de multiplicadores de Lagrange. A função de Lagrange é descrita da seguinte forma:

$$L(w, b, \alpha) = \frac{1}{2} w^T w - \sum_{i=1}^n \alpha_i [y_i(w^T x_i + b) - 1]$$

Onde  $\alpha_i$  é o multiplicador de Lagrange com respeito à  $i$ ésima inequação. Diferenciando  $L(w, b, \alpha)$  com respeito ao  $w$  e  $b$ , e definindo os resultados igual à zero, as seguintes duas condições de otimização são obtidas:

$$\begin{cases} \frac{\partial L(w, b, \alpha)}{\partial w} = 0 \\ \frac{\partial L(w, b, \alpha)}{\partial b} = 0 \end{cases}$$

Ao resolver estas equações, os seguintes resultados são obtidos:

$$\begin{cases} w = \sum_{i=1}^n \alpha_i y_i x_i \\ \sum_{i=1}^n \alpha_i y_i = 0 \end{cases}$$

Substituindo estes resultados na função de Lagrange, o seguinte problema dual é obtido:

$$\begin{aligned} \max_{\alpha} W(\alpha) &= \sum_{i=1}^n \alpha_i - \frac{1}{2} \sum_{i=1}^n \sum_{j=1}^n \alpha_i \alpha_j y_i y_j x_i^T x_j \\ s. t. \quad &\sum_{i=1}^n \alpha_i y_i = 0 \\ &\alpha_i \geq 0, \quad i = 1, \dots, n \end{aligned}$$

Considerando também a condição complementar de Karush-Kuhn-Tucker tem-se:

$$\alpha_i [y_i(w^T x_i + b) - 1] = 0, \quad i = 1, \dots, n$$

Consequentemente, apenas os vetores suportes  $(x_i, y_i)$  que são os pontos mais próximos do hiperplano ótimo e determinam a margem máxima, correspondem aos  $\alpha_i$ s não nulos. Todos os outros  $\alpha_i$ s são iguais à zero.

O problema dual apresentado nas equações anteriores é um típico problema de otimização convexa que em muitos casos converge para o mínimo global adotando técnicas de otimização apropriadas.

Depois de determinar os multiplicadores ótimos de Lagrange  $\alpha_i^*$ , é possível computar os pesos dos vetores  $w^*$  pela equação:

$$w^* = \sum_{i=1}^n \alpha_i^* y_i x_i$$

Então, utilizando um vetor positivo  $x_s$ , o bias  $b^*$  correspondente pode ser escrito como:

$$b^* = 1 - w^{*T} x_s \quad \text{para } y_s = +1$$

### 3.2 SVC COM MARGEM SUAVE E OTIMIZAÇÃO

A máxima margem dos SVC representa o ponto inicial dos algoritmos SVM. Entretanto, em muitos casos de problemas reais, pode-se considerar muito rigoroso exigir que todos os pontos sejam linearmente separáveis, especialmente em casos complexos de classificação não linear. Para resolver estes problemas inseparáveis, normalmente duas abordagens são tomadas. O primeiro método é o alívio das rígidas inequações do problema primal e assim gerar uma margem de otimização suave. O outro método utilizado é aplicar o artifício do Kernel para linearizar problemas não lineares. Nesta seção será introduzida a otimização por margem suave.

Imagine os casos em que há alguns pontos de classes opostas misturados juntos nos dados. Esses pontos representam o erro de treino que existe mesmo para o hiperplano de máxima margem. A ideia da “margem suave” tem como objetivo estender o algoritmo SVC de tal forma que o hiperplano permita que alguns pontos estejam alocados em classes erradas. Em particular, a variável folga  $\xi_i$  é introduzida para levar em conta a quantidade de violações de classificação obtidas pelo classificador:

$$\begin{aligned} \min_{w,b} \quad & \frac{1}{2} \|w\|^2 + C \sum_{i=1}^n \xi_i \\ \text{s. t. } & y_i(w^T x_i + b) \geq 1 - \xi_i, \quad \xi_i \geq 0 \quad i = 1, \dots, n \end{aligned}$$

Onde o parâmetro  $C$  controla a relação entre a complexidade da máquina e o número de pontos inseparáveis. A variável folga  $\xi_i$  tem uma explicação geométrica direta sendo tratada como a distância entre o dado mal classificado e hiperplano. Esta distância mede o desvio de uma amostra em relação à condição ideal de separação. Utilizando o mesmo método de multiplicadores de Lagrange, é possível formular o problema dual de margem suave como:

$$\max_{\alpha} W(\alpha) = \sum_{i=1}^n \alpha_i - \frac{1}{2} \sum_{i=1}^n \sum_{j=1}^n \alpha_i \alpha_j y_i y_j x_i^T x_j$$



$$s. t. \sum_{i=1}^n \alpha_i y_i = 0$$

$$0 \leq \alpha_i \leq C, \quad i = 1, \dots, n$$

Comparando o problema dual original e este último, nota-se que a maior diferença é a restrição  $\alpha_i \geq 0$  substituída por  $0 \leq \alpha_i \leq C$ . Mesmo assim, ambos os casos são considerados similares, de tal forma que a obtenção dos valores ótimos de pesos de vetores  $w$  e bias  $b$  continua sendo efetuada utilizando o mesmo procedimento.

Para o novo problema dual a condição complementar de Karush-Kuhn-Tucker é definida como:

$$\alpha_i [y_i (w^T x_i + b) - 1 + \xi_i] = 0, \quad i = 1, \dots, n$$

Sendo:

$$\gamma_i \xi_i = 0, \quad i = 1, \dots, n$$

Onde  $\gamma_i$  são os multiplicadores de Lagrange correspondentes à  $\xi_i$ .

Os pesos ótimos  $w^*$  são obtidos por:

$$w^* = \sum_{i=1}^n \alpha_i^* y_i x_i$$

Já o bias  $b^*$  podem ser obtidos escolhendo qualquer ponto dos dados de treinamento em que temos  $0 < \alpha_i^* < C$  e seu correspondente  $\xi_i = 0$ , e o utilizando na equação:

$$\alpha_i [y_i (w^T x_i + b) - 1 + \xi_i] = 0, \quad i = 1, \dots, n$$

### 3.3 ARTIFÍCIO DO KERNEL

A artimanha do uso de Kernel é outra técnica também utilizada para resolver problemas não separáveis linearmente. O grande desafio desta técnica é definir uma função Kernel ( $K(x, x') = \Phi^T(x) \Phi(x')$ ) apropriada, que seja baseada no produto interno dos dados sendo analisados e que seja capaz de realizar uma transformação não-linear dos dados do espaço inicial de entrada para um espaço com maior dimensão com intuito de tornar o problema linearmente separável.

Considerando  $\Phi: X \rightarrow H$  como uma transformação não linear de um espaço inicial  $X \subset \mathbb{R}^m$  para um novo espaço  $H$  onde o problema é linearmente separável. Pode-se definir o hiperplano ótimo correspondente como:

$$w^{\Phi T} \Phi(x) + b = 0$$

Sem perder generalidade, definimos bias como  $b = 0$  e a equação é simplificada para:

$$w^{\Phi T} \Phi(x) = 0$$

Semelhante aos casos linearmente separáveis, o objetivo do modelo é obter o vetor de pesos ótimos  $w^{\Phi*}$  no novo espaço por meio do uso de multiplicadores de Lagrange:

$$w^{\Phi*} = \sum_{i=1}^n \alpha_i^* y_i \Phi(x_i)$$

E, portanto, o hiperplano ótimo no novo espaço é computado por:

$$\sum_{i=1}^n \alpha_i^* y_i \Phi^T(x_i) \Phi(x) = 0$$

O termo  $\Phi^T(x_i) \Phi(x)$  representa o produto interno de dois vetores  $\Phi^T(x_i)$  e  $\Phi(x)$ .

A combinação das técnicas de SVC como margem suave e o artifício do Kernel resultam no seguinte problema dual:

$$\begin{aligned} \max_{\alpha} W(\alpha) &= \sum_{i=1}^n \alpha_i - \frac{1}{2} \sum_{i=1}^n \sum_{j=1}^n \alpha_i \alpha_j y_i y_j K(x_i, x_j) \\ s. t. \quad &\sum_{i=1}^n \alpha_i y_i = 0 \\ &0 \leq \alpha_i \leq C, \quad i = 1, \dots, n \end{aligned}$$

Seguindo o método de multiplicadores de Lagrange, obtém-se o classificador ótimo:

$$f(x) = \sum_{i=1}^n \alpha_i^* y_i K(x_i, x) + b^*$$

Onde  $b^* = 1 - \sum_{i=1}^n \alpha_i^* y_i K(x_i, x_s)$ , para um vetor suporte positivo  $y_s = +1$ .

Exemplos do uso de Kernels para SVC são ilustrados nas Figura 14 e Figura 15.

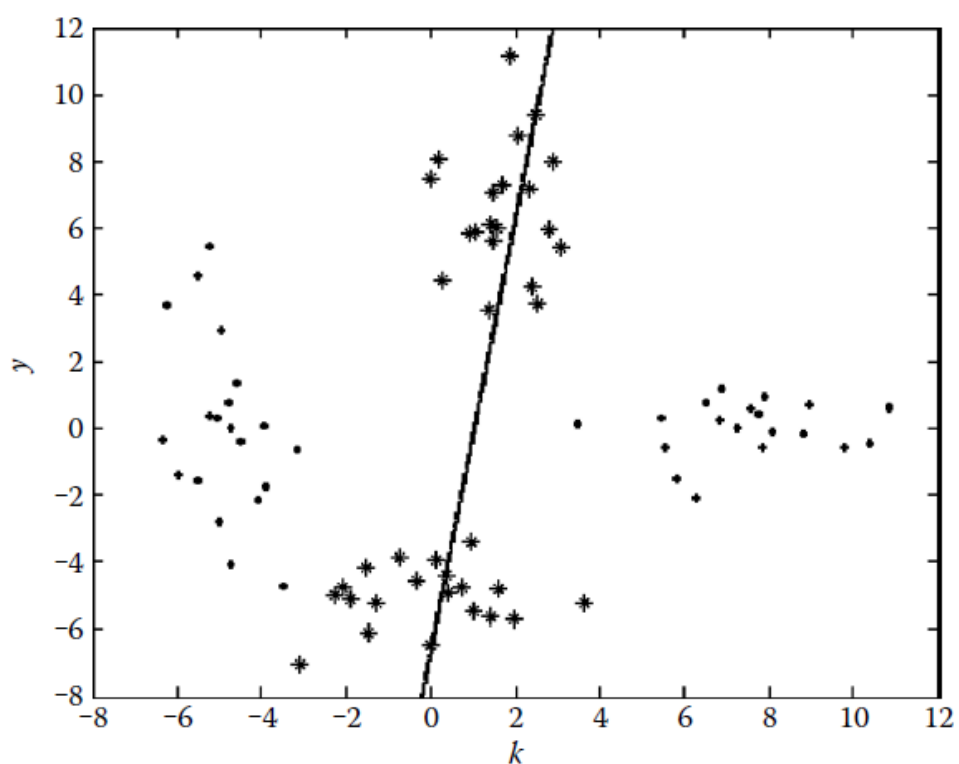


Figura 14 - SVC com Kernel linear (WU; KUMAR, 2009)

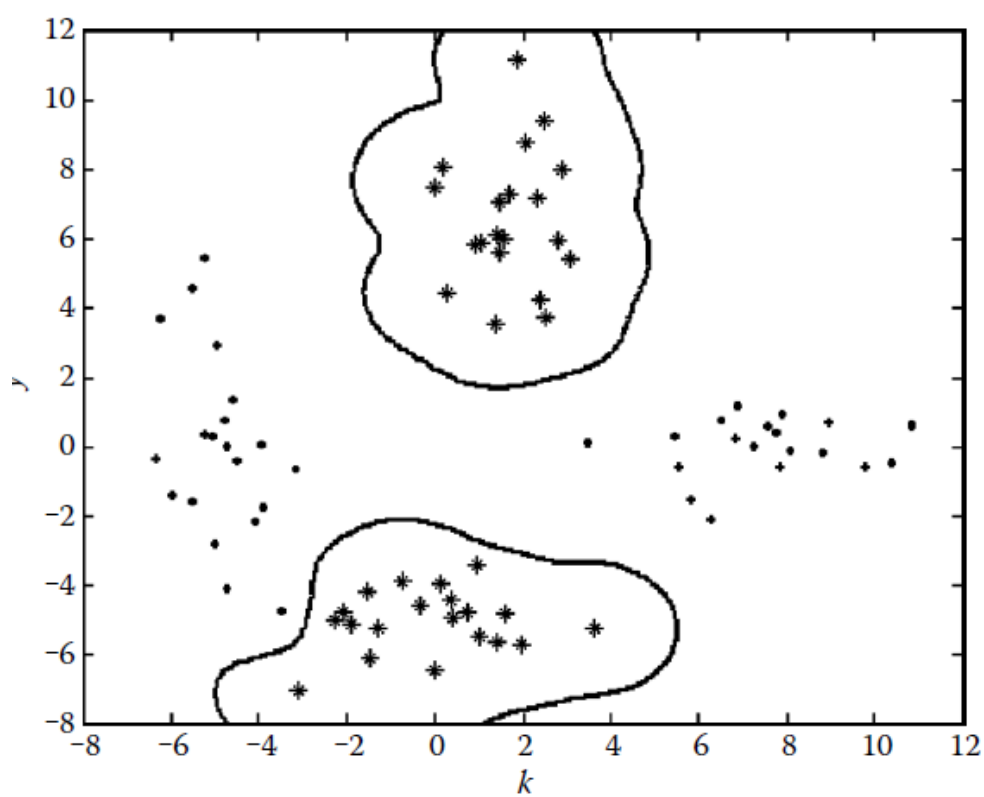


Figura 15 - SVC com Kernel de base radial (WU; KUMAR, 2009)

### 3.4 APLICAÇÃO DO SVM

Para implementação do SVM optou-se por utilizar uma base de dados simples gerada manualmente com o intuito de facilitar a visualização do desempenho do algoritmo. A base de dados utilizada no treinamento do SVM é apresentada na Tabela 3.

Tabela 3 - Base de dados de treinamento utilizada para implementação do SVM.

$y_i$	$x_i$
-1	(1, 7)
	(2, 8)
	(3, 8)
1	(5, 1)
	(6, -1)
	(7, 3)

Além disso, de forma a testar o algoritmo, uma base de dados de teste também foi gerada manualmente.

Tabela 4 – Base de dados de teste utilizada para avaliação de desempenho do algoritmo SVM

$x_i$	(0, 10)	(1, 3)	(3, 4)	(3, 5)	(5, 5)	(5, 6)	(6, -5)	(5, 8)
-------	---------	--------	--------	--------	--------	--------	---------	--------

O resultado da implementação do algoritmo SVM é ilustrado na Figura 16.

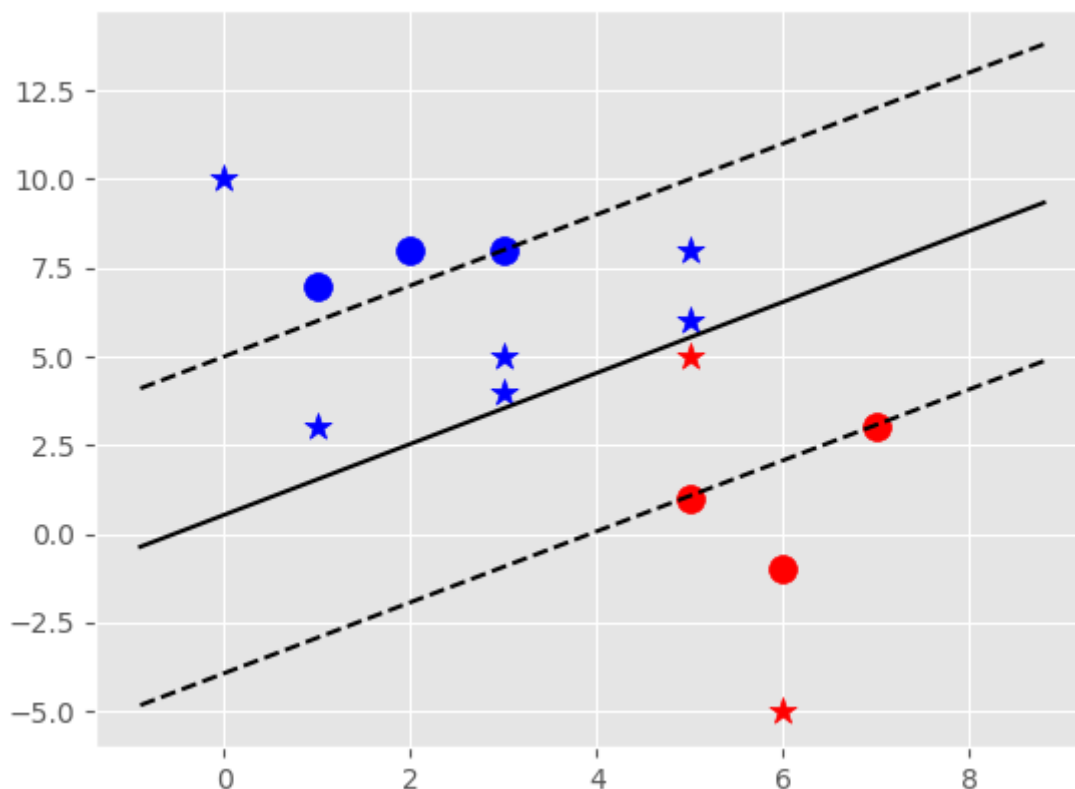


Figura 16 - Resultado da implementação do algoritmo SVM

Na Figura 16 os pontos com formato de bola são os dados de treino, os pontos que sobrepõem as linhas tracejadas são os vetores suporte e os pontos marcados por estrelas são amostras da base de dados de teste. Analisando os resultados nota-se que o algoritmo realizou corretamente a classificação de todos os pontos testes, dividindo de forma correta os pontos entre as classes.

### 3.4.1 CÓDIGO

```
import matplotlib.pyplot as plt
from matplotlib import style
import numpy as np
style.use('ggplot')

class Support_Vector_Machine:
    def __init__(self, visualization = True):
        self.visualization = visualization
        self.colors = {1:'r', -1:'b'}
        if self.visualization:
            self.fig = plt.figure()
            self.ax = self.fig.add_subplot(1,1,1)

    # Treinamento
    def fit(self, data):
        self.data = data
        # (||w||: [w,b])
        opt_dict = {} # Dicionário com valores otimizados
        # Resolve o problema do vetores com magitudes iguais mas produtos es
calares diferentes
        transforms = [[1,1], [-1,1], [-1,-1], [1,-1]]
        all_data = []
        for yi in self.data:
            for featureset in self.data[yi]:
                for feature in featureset:
                    all_data.append(feature)
        self.max_feature_value = max(all_data)
        self.min_feature_value = min(all_data)
        all_data = None
        step_sizes = [self.max_feature_value*0.1, self.max_feature_value*0.0
1, self.max_feature_value*0.001] # 0.001 é um ponto de custo
        # Extremamente caro
        b_range_multiple = 5
        # Para b não é necessário u step tão pequeno quanto o utilizado para
o w
        b_multiple = 5
        latest_optimum = self.max_feature_value*10
        for step in step_sizes:
            w = np.array([latest_optimum, latest_optimum])
            # Podemos fazer isso pois é um problema de otimização convexa
            optimized = False
```

```

        while not optimized:
            for b in np.arange(-
1*(self.max_feature_value*b_range_multiple), self.max_feature_value*b_range_
multiple, step*b_multiple):
                for transformation in transforms:
                    w_t = w*transformation
                    found_option = True
                    # Ponto do fraco do SVM
                    # Checar  $y_i \cdot (x_i \cdot w + b) \geq 1$ 
                    for i in self.data:
                        for xi in self.data[i]:
                            yi = i
                            if not yi*(np.dot(w_t,xi) + b) >= 1:
                                found_option = False
                    if found_option:
                        opt_dict[np.linalg.norm(w_t)] = [w_t, b]
            if w[0] < 0:
                optimized = True
                print('Optimized a step.')
            else:
                w = w - step # Matematicamente estranho
                norms = sorted([n for n in opt_dict])
                # ||w||: [w,b]
                opt_choice = opt_dict[norms[0]]
                self.w = opt_choice[0]
                self.b = opt_choice[1]
                latest_optimum = opt_choice[0][0] + step*2
            for i in self.data:
                for xi in self.data[i]:
                    yi=i
                    print(xi,':',yi*(np.dot(self.w,xi)+self.b))

def predict(self, features):
    # Sign( $x \cdot w + b$ )
    classification = np.sign(np.dot(np.array(features), self.w) + self.b
)

    if classification != 0 and self.visualization:
        self.ax.scatter(features[0], features[1], s=100, marker='*', c=s
elf.colors[classification])
        return classification

def visualize(self):
    [[self.ax.scatter(x[0], x[1], s=100, color=self.colors[i]) for x in
data_dict[i]] for i in data_dict]
    # hyperplane  $x \cdot w + b$ 
    #  $v = x \cdot w + b$ 
    # psv = +1
    # nsf = -1
    # db = 0

```

```

def hyperplane(x,w,b,v):
    return (-w[0]*x-b+v) / w[1]
datarange = (self.min_feature_value*0.9,self.max_feature_value*1.1)
hyp_x_min = datarange[0]
hyp_x_max = datarange[1]

# (w.x + b) = 1
# positive support vector hyperplane
psv1 = hyperplane(hyp_x_min, self.w, self.b, 1)
psv2 = hyperplane(hyp_x_max, self.w, self.b, 1)
self.ax.plot([hyp_x_min, hyp_x_max], [psv1, psv2], 'k--')

# (w.x + b) = -1
# negative support vector hyperplane
nsv1 = hyperplane(hyp_x_min, self.w, self.b, -1)
nsv2 = hyperplane(hyp_x_max, self.w, self.b, -1)
self.ax.plot([hyp_x_min, hyp_x_max], [nsv1, nsv2], 'k--')

# (w.x + b) = 0
# decision boundary hyperplane
db1 = hyperplane(hyp_x_min, self.w, self.b, 0)
db2 = hyperplane(hyp_x_max, self.w, self.b, 0)
self.ax.plot([hyp_x_min, hyp_x_max], [db1, db2], 'k')

plt.show()

data_dict = {
    -1: np.array([[1,7], [2,8], [3,8]]),
    1: np.array([[5,1], [6,-1], [7,3]])
}

svm = Support_Vector_Machine()
svm.fit(data=data_dict)
predict_us = [[0,10], [1,3], [3,4], [3,5], [5,5], [5,6], [6,-5], [5,8]]
for p in predict_us:
    svm.predict(p)
svm.visualize()

```

Figura 17 - Código em Python para implementação do SVM

### 3.5 BIBLIOGRAFIA

CORTES, C.; VAPNIK, V. Support-vector networks. Machine learning. Anais...1995

WU, Xindong; KUMAR, Vipin (Ed.). The top ten algorithms in data mining. CRC press, 2009

## 4 APRIORI

Nesta seção será abordado o mais básico e fundamental algoritmo de mineração de padrões de frequência e regras de associação conhecido como Apriori.

### 4.1 DESCRIÇÃO DO ALGORITMO

#### 4.1.1 MINERANDO PADRÕES DE FREQUÊNCIA E REGRAS DE ASSOCIAÇÃO

Uma das mais populares abordagens de mineração de dados é a procura de itens frequentes de uma base de dados de transações e derivar leis de associação. O problema é formalmente definido da seguinte forma. Seja  $I = \{i_1, i_2, \dots, i_m\}$  um conjunto de itens. Seja  $D$  um conjunto de transações onde cada transação  $t$  é composta por um conjunto de itens tal que  $t \subseteq I$ . Cada transação apresenta um identificador único chamado de *TID*. A transação  $t$  contém  $X$ , um conjunto de itens em  $I$  se  $X \subseteq t$ . Uma regra de associação é a implicação na forma  $X \Rightarrow Y$  onde  $X \subset I$ ,  $Y \subset I$  e  $X \cap Y = \emptyset$ . A regra  $X \Rightarrow Y$  é válida em  $D$  com confiança  $c$  ( $0 \leq c \leq 1$ ) se a fração de transações que também contém  $Y$  naquelas que contém  $X$  em  $D$  é igual a  $c$ . A regra  $X \Rightarrow Y$  tem suporte  $s$  ( $0 \leq s \leq 1$ ) em  $D$  se a fração de transações em  $D$  que contém  $X \cup Y$  é  $s$ . Dado um conjunto de transações  $D$ , o problema na mineração de regras de associação é gerar todas as regras de associação que tem suporte e confiança maior do que o mínimo suporte especificado (*minsup*) e a mínima confiança (*minconf*), respectivamente. Achar conjuntos de itens frequentes (conjuntos com suporte maior que *minsup*) não é trivial pelo fato de ser computacionalmente complexo devido à explosão combinatória do problema. Uma vez que os conjuntos de itens frequentes são obtidos, a geração de regras de associação com confiança maior que *minconf* é direta (WU; KUMAR, 2009).

#### 4.1.2 APRIORI

Apriori é um algoritmo utilizado para identificar conjunto de itens que possuem suporte maior que *minsup*. O suporte para um conjunto de itens é a razão do número de transações que possuem o conjunto de itens e o número total de transações. Os conjuntos de itens que satisfazem a restrição de mínimo suporte são chamados de conjunto de itens frequentes. O algoritmo realiza múltiplas análises sobre os dados. Na primeira passagem, o suporte individual de cada item é calculado e os itens frequentes são determinados. Em cada passagem subsequente, um item ou conjunto de itens determinado como frequente na passagem anterior é usado para gerar novos potenciais conjuntos de itens frequentes, chamados de candidatos de conjunto de itens frequentes, sendo os seus suportes calculados durante a passagem pelos dados. No final da passagem, os conjuntos que satisfizerem a restrição de suporte mínimo são coletados, ou seja, os conjuntos de itens frequentes são determinados e são utilizados para a passagem seguinte. Esse processo é repetido até que não haja novos conjuntos de itens frequentes.

O número de itens em um conjunto de itens é chamado de dimensão e um conjunto de itens com dimensão  $k$  é chamado  $k$ -conjunto de itens. Além disso, o conjunto de



grupo de itens frequentes com dimensão igual a  $k$  é denotado como  $F_k$  e seus candidatos como  $C_k$ .

O algoritmo Apriori é apresentado na Figura 18. O primeiro passo realiza a contagem de ocorrências que determinam os 1-conjunto de itens frequentes. Os passos subsequentes consistem em duas fases. Na primeira os conjuntos de itens frequentes  $F_{k-1}$  identificados no passo  $(k - 1)^{ésimo}$  passo são utilizados para gerar candidatos de conjunto de itens  $C_k$  usando a função *apriori-gen*. Em seguida, a base de dados é percorrida e o suporte dos candidatos em  $C_k$  é calculado.

---

```

 $F_1 = \{\text{frequent 1-itemsets}\};$ 
for ( $k = 2; F_{k-1} \neq \emptyset; k++$ ) do begin
   $C_k = \text{apriori-gen}(F_{k-1});$  //New candidates
  foreach transaction  $t \in \mathcal{D}$  do begin
     $C_t = \text{subset}(C_k, t);$  //Candidates contained in  $t$ 
    foreach candidate  $c \in C_t$  do
       $c.\text{count}++;$ 
    end
   $F_k = \{c \in C_k \mid c.\text{count} \geq \text{minsup}\};$ 
end
Answer =  $\cup_k F_k;$ 

```

---

Figura 18 - Algoritmo do algoritmo Apriori (WU; KUMAR, 2009)

A função *apriori-gen* recebe como argumento  $F_{k-1}$  e retorna um superconjunto com todos os  $k$ -conjuntos de itens frequentes. Primeiro, em um passo de junção,  $F_{k-1}$  é ligado com  $F_{k-1}$ .

```

insert into  $C_k$ 
select  $p.\text{fitemset}_1, p.\text{fitemset}_2, \dots, p.\text{fitemset}_{k-1}, q.\text{fitemset}_{k-1}$ 
from  $F_{k-1}p, F_{k-1}q$ 
where  $p.\text{fitemset}_1 = q.\text{fitemset}_1, \dots, p.\text{fitemset}_{k-2} = q.\text{fitemset}_{k-2},$ 
       $p.\text{fitemset}_{k-1} < q.\text{fitemset}_{k-1}$ 

```

Figura 19 - Função *apriori-gen* (WU; KUMAR, 2009)

Aqui,  $F_{k-1}p$  significa que o conjunto de itens  $p$  é um  $k$ -conjunto de itens frequente e  $p.\text{fitemset}_k$  é o  $k^{ésimo}$  item do conjunto de itens frequentes  $p$ . Então na etapa de poda, todos os conjuntos de itens  $c \in C_k$  em que algum  $(k - 1)$ -subconjunto não está em  $F_{k-1}$  são deletados.

A função *subset* recebe como argumento  $C_k$  e a transação  $t$ , e retorna todos os conjuntos de itens candidatos presentes na transação  $t$ . Para realizar uma contagem rápida, o Apriori adota uma árvore para armazenar os candidatos de conjunto de itens

$C_k$ . Os conjuntos são armazenados em folhas. Uma vez que a árvore é construída, a função *subset* identifica todos os candidatos presentes na transação  $t$ , começando pela raiz. A operação de junção é equivalente a estender  $F_{k-1}$  com cada item na base de dados e deletar os conjuntos de itens em que o  $(k-1)$ -subconjunto obtido deletando o  $(k-1)^{ésimo}$  item que não está em  $F_{k-1}$ .

A última etapa do processo é gerar as desejadas regras de associações dos conjuntos de itens frequentes. Um algoritmo direto que realiza esta tarefa é explicado a seguir. Para gerar regras, todos os subconjuntos não vazios de todo conjunto de itens frequente  $f$  são enumerados e para todo subconjunto  $a$ , a regra seguindo a forma  $a \Rightarrow (f - a)$  é gerada se a razão entre  $support(f)$  e  $support(a)$  é no mínimo  $minconf$ . Sendo assim, o algoritmo para gerar regras de associação é apresentado Figura 20.

O algoritmo Apriori apresenta bom desempenho ao reduzir a dimensão dos conjuntos de candidatos. Entretanto, em situações com muitos conjuntos de itens frequentes ou com baixos valores mínimos de suporte, o algoritmo ainda sofre com o custo de gerar um número grande de conjunto de candidatos e percorrer a base de dados repetidamente para checar um grande conjunto de candidatos de conjuntos de itens. (Xindong & Kumar, 2009)

---

```

 $H_1 = \emptyset$  //Initialize
foreach; frequent  $k$ -itemset  $f_k, k \geq 2$  do begin
   $A = (k-1)$ -itemsets  $a_{k-1}$  such that  $a_{k-1} \subset f_k$ ;
  foreach  $a_{k-1} \in A$  do begin
     $conf = support(f_k)/support(a_{k-1})$ ;
    if ( $conf \geq minconf$ ) then begin
      output the rule  $a_{k-1} \Rightarrow (f_k - a_{k-1})$ 
        with confidence =  $conf$  and support =  $support(f_k)$ ;
      add  $(f_k - a_{k-1})$  to  $H_1$ ;
    end
  end
  call ap-genrules( $f_k, H_1$ );
end

Procedure ap-genrules( $f_k$ : frequent  $k$ -itemset,  $H_m$ : set of  $m$ -item
  consequents)
if ( $k > m + 1$ ) then begin
   $H_{m+1} = apriori-gen(H_m)$ ;
  foreach  $h_{m+1} \in H_{m+1}$  do begin
     $conf = support(f_k)/support(f_k - h_{m+1})$ ;
    if ( $conf \geq minconf$ ) then
      output the rule  $f_k - h_{m+1} \Rightarrow h_{m+1}$ 
        with confidence =  $conf$  and support =  $support(f_k)$ ;
    else
      delete  $h_{m+1}$  from  $H_{m+1}$ ;
    end
  end
  call ap-genrules( $f_k, H_{m+1}$ );
end

```

---

Figura 20 - Algoritmo para geração de regras de associação (WU; KUMAR, 2009)

## 4.2 APLICAÇÃO DO ALGORITMO APRIORI

Para a implementação do algoritmo Apriori foram utilizadas duas bases de dados. A primeira base de dados utilizada é uma base simples gerada manualmente e que foi escolhida com intuito de aplicar o algoritmo para um caso simples. de fácil interpretação. Já a segunda base de dados é uma base mais complexa com várias transações realizadas em um mercado, que foi escolhida para observar o desempenho do algoritmo para um caso mais complexo.

### 4.2.1 APLICAÇÃO DO APRIORI: CASO SIMPLES

A base de dados de para o caso simples consiste em conjunto de dez transações que são utilizadas para identificar padrões e regras de associações. A base de dados é ilustrada na Tabela 5. Ao rodar o programa contendo o algoritmo para este caso, é solicitada a especificação do valor de *minsup* e *minconf*. Sendo assim, de forma a exemplificar a operação do algoritmo implementado, os resultados para operação do modelo com *minsup* = 0.6 e *minconf* = 0.7 são apresentados na Figura 21.

Tabela 5 - Base de dados utilizada para implementação do caso simples do Apriori

Nº	Transação
1	Bread,Milk
2	Bread,Diapers,Beer,Eggs
3	Milk,Diapers,Beer,Cola
4	Bread,Milk,Diapers,Beer
5	Bread,Milk,Diapers,Cola
6	Bread,Milk
7	Bread,Cola,Beer,Milk
8	Milk,Bread,Beer,Cola
9	Bread,Milk,Diapers,Beer
10	Bread,Beer,Diapers,Diapers

Minimum Support: 0.6  
Minimum Confidence: 0.7

Item Sets:

[frozenset({'Cola'}), frozenset({'Bread'}), frozenset({'Milk'}), frozenset({'Eggs'}), frozenset({'Beer'}), frozenset({'Diapers'})]

Transactions:

[frozenset({'Milk', 'Bread'}), frozenset({'Beer', 'Diapers', 'Bread', 'Eggs'}), frozenset({'Milk', 'Beer', 'Diapers', 'Cola'}), frozenset({'Milk', 'Beer', 'Diapers', 'Bread'}), frozenset({'Milk', 'Cola', 'Diapers', 'Bread'}), frozenset({'Milk', 'Bread'}), frozenset({'Milk', 'Beer', 'Cola', 'Bread'}), frozenset({'Milk', 'Beer', 'Diapers', 'Bread'}), frozenset({'Beer', 'Diapers', 'Bread'})]

---Frequent Itemsets---

[Itemset] | [Support]  
(Diapers,) : 0.6  
(Beer, Bread) : 0.6  
(Beer,) : 0.7  
(Milk, Bread) : 0.7  
(Milk,) : 0.8  
(Bread,) : 0.9

---Rules---

[Rule] | [Confidence]  
(Bread,) => (Milk,) : 0.7778  
(Beer,) => (Bread,) : 0.8571  
(Milk,) => (Bread,) : 0.875

Figura 21 - Resultado da implementação do algoritmo apriori para o caso simples

#### 4.2.2 APLICAÇÃO DO APRIORI: CASO COMPLEXO

A base de dados de para o caso complexo consiste em conjunto de 1001 transações obtidas em um banco de dados público (<https://github.com/luoyetx/Apriori/blob/master/data.csv>) que possui dados de transações de um mercado. O algoritmo é aplicado nesse caso para tentar obter padrões e regras de associações para consumo neste mercado. A base de dados é ilustrada na **Erro! Fonte de referência não encontrada..** Ao rodar o programa contendo o algoritmo para este caso, é solicitada a especificação do valor de *minsup* e *minconf*. Sendo assim, de forma a exemplificar a operação do algoritmo implementado, os resultados para operação do modelo com *minsup* = 0.3 e *minconf* = 0.5 são apresentados na Figura 22.

Tabela 6 – Base de dados utilizada para implementação do caso complexo do Apriori

Nº	Transação
1	corned_b,peppers,bourbon,cracker,chicken,apples,coke
2	olives,bourbon,coke,turkey,ice_crea,ham,baguette
3	hering,corned_b,olives,ham,turkey,bourbon,peppers
4	baguette,sardines,apples,peppers,avocado,ice_crea,bourbon
5	baguette,soda,hering,cracker,heineken,peppers,apples
6	baguette,soda,hering,cracker,heineken,corned_b,ham
7	avocado,cracker,artichok,heineken,ham,ice_crea,olives
8	hering,corned_b,apples,olives,steak,cracker,chicken
9	corned_b,peppers,bourbon,cracker,chicken,avocado,soda
10	baguette,sardines,apples,peppers,avocado,ice_crea,ice_crea
11	soda,olives,bourbon,cracker,heineken,steak,corned_b
12	soda,olives,bourbon,cracker,heineken,steak,steak
13	soda,olives,bourbon,cracker,heineken,ham,hering
14	corned_b,peppers,bourbon,cracker,chicken,sardines,olives
15	sardines,heineken,chicken,coke,ice_crea,hering,cracker
16	sardines,heineken,chicken,coke,ice_crea,artichok,steak
17	olives,bourbon,coke,turkey,ice_crea,heineken,apples
18	sardines,heineken,chicken,coke,ice_crea,soda,ham
19	hering,corned_b,olives,ham,turkey,cracker,avocado
20	hering,corned_b,apples,olives,steak,bordeaux,avocado
21	sardines,heineken,chicken,coke,ice_crea,steak,cracker
22	baguette,sardines,apples,peppers,avocado,olives,cracker
23	corned_b,peppers,bourbon,cracker,chicken,steak,baguette
24	baguette,sardines,apples,peppers,avocado,cracker,bourbon
25	baguette,hering,avocado,artichok,heineken,olives,cracker
26	baguette,hering,avocado,artichok,heineken,chicken,apples
27	olives,bourbon,coke,turkey,ice_crea,apples,baguette
28	avocado,cracker,artichok,heineken,ham,coke,ice_crea
29	sardines,heineken,chicken,coke,ice_crea,olives,ham
30	soda,olives,bourbon,cracker,heineken,bordeaux,artichok
31	baguette,soda,hering,cracker,heineken,peppers,chicken
32	avocado,cracker,artichok,heineken,ham,ice_crea,bourbon
33	baguette,hering,avocado,artichok,heineken,soda,olives
34	soda,olives,bourbon,cracker,heineken,ham,peppers
35	sardines,heineken,chicken,coke,ice_crea,olives,cracker
36	baguette,soda,hering,cracker,heineken,steak,sardines
37	sardines,heineken,chicken,coke,ice_crea,artichok,turkey
38	soda,olives,bourbon,cracker,heineken,artichok,hering
39	baguette,hering,avocado,artichok,heineken,sardines,peppers
40	sardines,heineken,chicken,coke,ice_crea,bordeaux,olives
41	sardines,heineken,chicken,coke,ice_crea,bourbon,peppers
42	olives,bourbon,coke,turkey,ice_crea,baguette,steak
43	baguette,hering,avocado,artichok,heineken,peppers,ham
...	...
1001	olives,bourbon,coke,turkey,ice_crea,peppers,peppers

```

Minimum Support: 0.3
Minimum Confidence: 0.5

---Frequent Itemsets---
[Itemset] | [Support]
('ham',) : 0.3047
('artichok',) : 0.3047
('ice_crea',) : 0.3127
('apples',) : 0.3137
('chicken',) : 0.3147
('soda',) : 0.3177
('avocado',) : 0.3626
('heineken', 'cracker') : 0.3656
('corned_b',) : 0.3906
('baguette',) : 0.3916
('bourbon',) : 0.4026
('olives',) : 0.4725
('hering',) : 0.4855
('cracker',) : 0.4875
('heineken',) : 0.5994

---Rules---
[Rule] | [Confidence]
('heineken',) => ('cracker',) : 0.61
('cracker',) => ('heineken',) : 0.75

```

Figura 22 - Resultado da implementação do algoritmo Apriori para o caso complexo

#### 4.2.3 CÓDIGO CASO COMPLEXO

```

# Importação dos pacotes utilizados no algoritmo
from itertools import chain, combinations
import operator

# Função para obtenção de todas as combinações de items
def subsets(itemset):
    return chain(*[combinations(itemset, i + 1) for i, a in enumerate(itemse
t)])

# Função do algoritmo Apriori
def apriori(data, min_support, min_confidence):
    # Lista de item sets e transações
    itemset, transaction_list = itemset_from_data(data)
    print('\n')
    print(f'Item Sets: \n \n{list(itemset)}')
    print('\n')
    print(f'Transactions: \n \n{list(transaction_list)}')

    # Gerar candidatos
    candidates = get_candidates(transaction_list, itemset, min_support)

```

```

rules = list()
for sets in candidates.keys():
    if len(sets) > 1:
        for subset in subsets(sets):
            item = sets.difference(subset)
            if item: # If not None
                subset = frozenset(subset)
                subset_item = subset | item # União de sets
                confidence = float(candidates[subset_item]) / candidates
[subset]

                if confidence >= min_confidence:
                    rules.append((subset, item, confidence))
return rules, candidates

# Função para obtenção de combinações de k-itens
def joinset(itemset, k):
    joint_set = set()
    for i in itemset:
        for j in itemset:
            if len(i.union(j)) == k:
                joint_set.add(i.union(j))
    return joint_set

# Função para determinar os candidatos à itemsets frequentes
def get_candidates(transaction_list, itemset, min_support):
    candidates = dict()
    k = 1
    k_itemset = get_freq_itemset(transaction_list, itemset, min_support)
    candidates.update(k_itemset)
    k += 1
    while True:
        itemset = joinset(k_itemset, k)
        k_itemset = get_freq_itemset(transaction_list, itemset, min_support)
        if not k_itemset: # If None
            break
        candidates.update(k_itemset)
        k += 1
    return candidates

# Função para determinar os itens mais frequentes de acordo com o valor de s
# uporte
def get_freq_itemset(transaction_list, itemset, min_support):
    len_transaction_list = len(transaction_list)
    freq_itemsets = dict()
    for item in itemset:
        freq_itemsets[item] = 0
        for row in transaction_list:
            if item.issubset(row):

```

```

        freq_itemsets[item] += 1
    freq_itemsets[item] = freq_itemsets[item] / len_transaction_list
    relevant_itemsets = dict()
    for item, support in freq_itemsets.items():
        if support >= min_support:
            relevant_itemsets[item] = support
    return relevant_itemsets

# Constrói lista de itemsets e transações
def itemset_from_data(data):
    itemset = set()
    transaction_list = list()
    for row in data:
        transaction_list.append(frozenset(row))
        for item in row:
            if item not in itemset:
                itemset.add(frozenset([item]))
    return itemset, transaction_list

# Função para imprimir resultados
def print_report(rules, candidates):
    print('\n')
    print('---Frequent Itemsets---')
    print('[Itemset] | [Support]')
    sorted_candidates = sorted(candidates.items(), key=operator.itemgetter(1))
    for candidate in sorted_candidates:
        print(f'tuple(candidate[0]) : {round(candidate[1], 4)}')

    print('\n')
    print('---Rules---')
    sorted_rules = sorted(rules, key=lambda s : s[2])
    print('[Rule] | [Confidence]')
    for rule in sorted_rules:
        print(f'tuple(rule[0]) => {tuple(rule[1])} : {round(rule[2], 4)}')

# Função para leitura de dados csv
def get_csv_data(filename):
    data = []
    f = open(filename, 'r')
    csv_data = f.read()
    rows = csv_data.strip().split('\n')
    for row in rows:
        split_row = row.strip().split(',')
        data.append(split_row)
    return data

# Main()

```

```
data = get_csv_data('supermarket_data.csv')
print('\n')
#print('Leitura dos dados:')
#print(data)

print('\n')
min_support = float(input('Minimum Support: '))
min_confidence = float(input('Minimum Confidence: '))

rules, candidates = apriori(data, min_support, min_confidence)
print_report(rules, candidates)
```

Figura 23 - Código em Python para implementação do algoritmo Apriori

### 4.3 BIBLIOGRAFIA

AGRAWAL, R.; SRIKANT, R. Fast algorithms for mining association rules. In Proc. of the 20th International Conference on Very Large Data Bases (VLDB 1994). Anais.1994

WU, Xindong; KUMAR, Vipin (Ed.). The top ten algorithms in data mining. CRC press, 2009



## 5 EM: EXPECTATION-MAXIMIZATION

O algoritmo expectativa maximização (EM - Expectation-Maximization) é uma abordagem amplamente aplicável para o cálculo iterativo de estimativas de máxima verossimilhança (ML) e útil em uma grande variedade de problemas de dados incompletos. Em particular, o algoritmo EM simplifica consideravelmente o problema de ajuste de modelos de mistura finita por ML, onde os modelos de mistura são usados para modelar a heterogeneidade no contexto de análise de clusters e reconhecimento de padrões. O algoritmo EM possui várias propriedades interessantes, incluindo estabilidade numérica, simplicidade de implementação e convergência global confiável (WU; KUMAR, 2009).

### 5.1 DESCRIÇÃO DO ALGORITMO EM

O algoritmo EM é um algoritmo iterativo no qual em cada iteração dois passos são realizados, o passo Expectativa (E-step) e o passo Maximização (M-Step). O passo a passo do algoritmo será detalhado a seguir.

O primeiro passo do algoritmo é decidir quantos clusters/fontes ( $c$ ) são desejados para ajustar os dados e, em seguida, se deve iniciar os parâmetros média  $\mu_c$ , covariância  $\Sigma_c$  e fração  $\pi_c$ .

#### E-Step:

Calcular para cada ponto  $x_i$  a probabilidade  $r_{ic}$  do ponto  $x_i$  pertencer ao cluster  $c$ .

$$r_{ic} = \frac{\pi_c N(x_i | \mu_c, \Sigma_c)}{\sum_{k=1}^K \pi_k N(x_i | \mu_k, \Sigma_k)}$$

Onde  $N(x|c, \Sigma)$  descreve uma Gaussiana multivariável:

$$N(x_i, \mu_c, \Sigma_c) = \frac{1}{(2\pi)^{\frac{n}{2}} |\Sigma_c|^{\frac{1}{2}}} \exp \left( -\frac{1}{2} (x_i - \mu_c)^T \Sigma_c^{-1} (x_i - \mu_c) \right)$$

O  $r_{ic}$  fornece, para cada ponto na base de dados  $x_i$ , a razão entre a probabilidade de  $x_i$  pertencer ao cluster  $c$  e a soma da probabilidade de  $x_i$  pertencer aos clusters. Portanto, se  $x_i$  estiver bem próximo de uma gaussiana  $c$ , este apresentará um valor elevado de  $r_{ic}$  para esta Gaussiana e relativamente um valor baixo para as outras Gaussianas.

#### M-Step:

Para cada cluster  $c$ : calcular o peso total  $m_c$  (fração de pontos alocados para o cluster  $c$ ) e atualizar os valores de  $\pi_c$ ,  $\mu_c$  e  $\Sigma_c$  utilizando  $r_{ic}$ .

$$m_c = \sum_i r_{ic}$$

$$\pi_c = \frac{m_c}{m}$$

$$\mu_c = \frac{1}{m_c} \sum_i r_{ic} x_i$$

$$\Sigma_c = \frac{1}{m_c} \sum_i r_{ic} (x_i - \mu_c)^T (x_i - \mu_c)$$

Os passos E e M devem ser iterativamente repetidos até a função log-verossimilhança do modelo convergir. A função log-verossimilhança (*log-likelihood*) é calculada por:

$$\ln p(X|\pi, \mu, \Sigma) = \sum_{i=1}^N \ln \left( \sum_{k=1}^K \pi_k N(x_i | \mu_k, \Sigma_k) \right)$$

## 5.2 APLICAÇÃO DO ALGORITMO EM

Para aplicação prática do algoritmo dois casos foram implementados. No primeiro caso o algoritmo foi aplicado para uma base unidimensional gerada utilizando uma função randômica. No segundo caso o algoritmo foi aplicado para uma base de dados bidimensional também gerada por meio de uma função randômica.

### 5.2.1 APLICAÇÃO DO EM: CASO 1D

Para este caso de implementação, a base de dados foi gerada de tal forma que três clusters podem ser verificados. Sendo assim, para este caso, ao se implementar o algoritmo o objetivo dividir os dados em três clusters modelados cada um por uma Gaussiana. O resultado da implementação do algoritmo para este caso pode ser observado nas figuras a seguir. O algoritmo realiza 10 iterações e em cada uma das iterações é possível observar as Gaussianas se ajustando aos seus respectivos clusters. Sendo assim, no final das iterações é evidente a divisão da base de dados em três grupos.

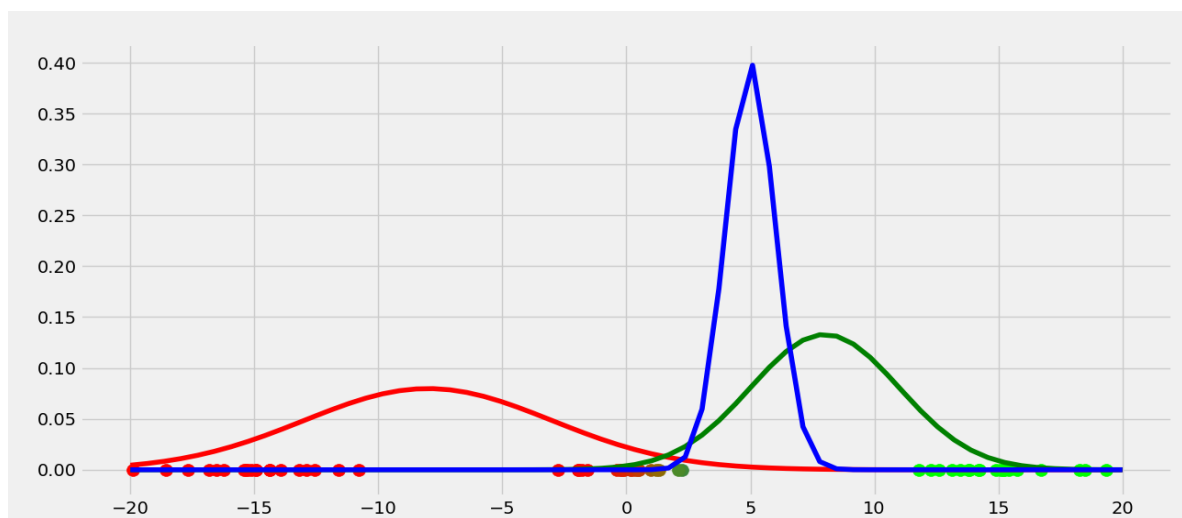


Figura 24 - Implementação 1D - Iteração 1

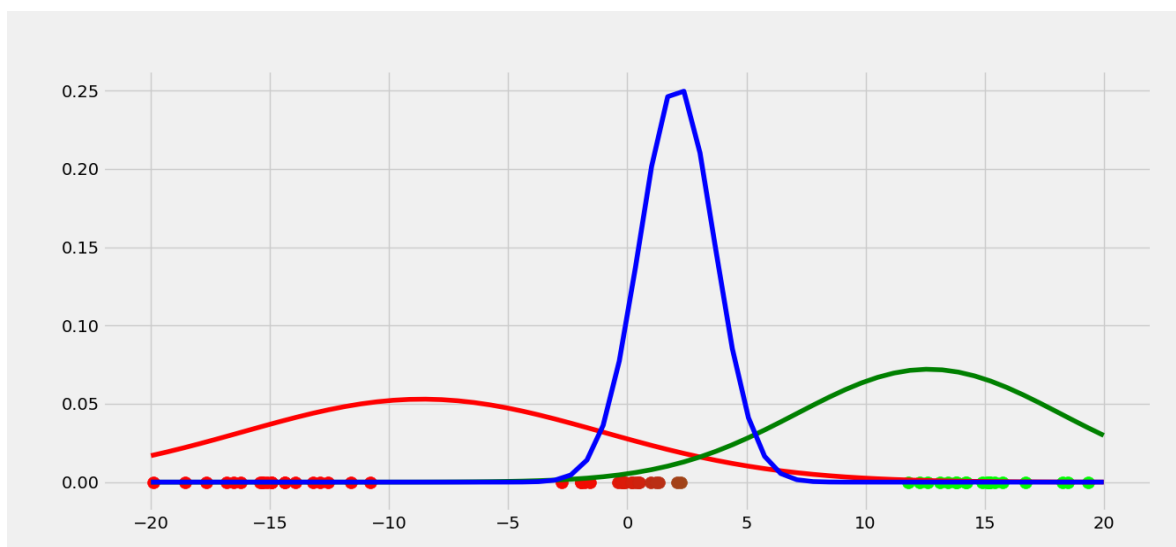


Figura 25 - Implementação 1D - Iteração 2

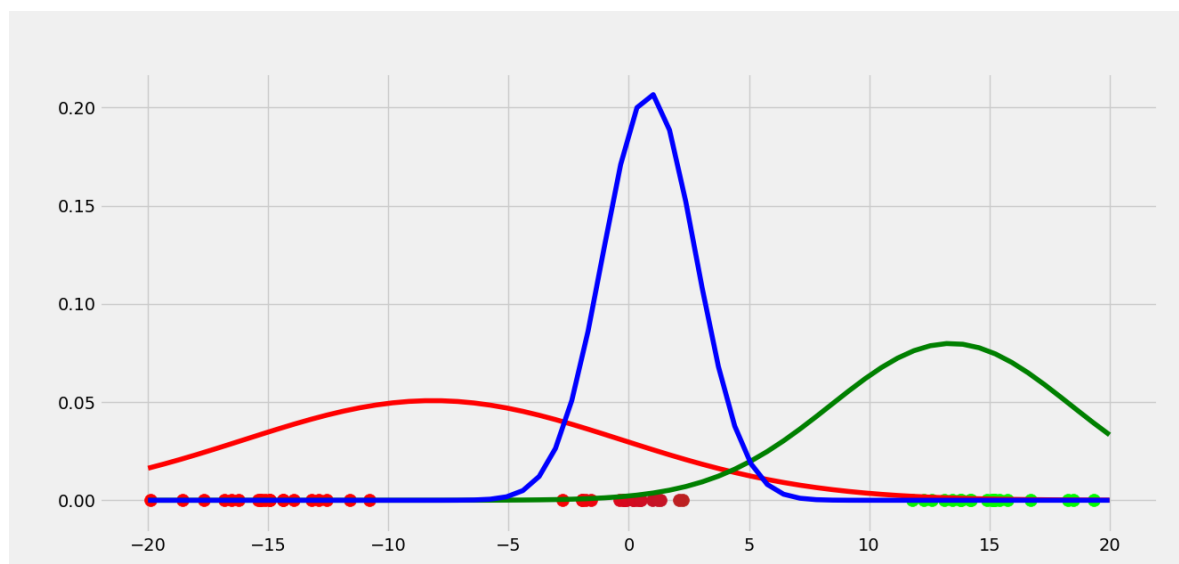


Figura 26 - Implementação 1D - Iteração 3

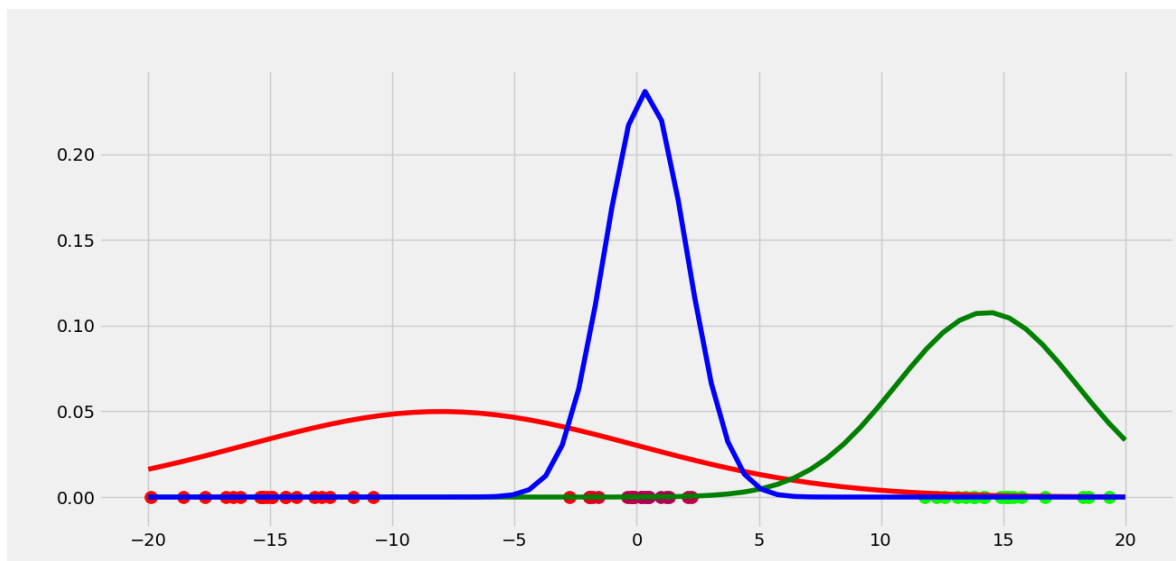


Figura 27 - Implementação 1D - Iteração 4

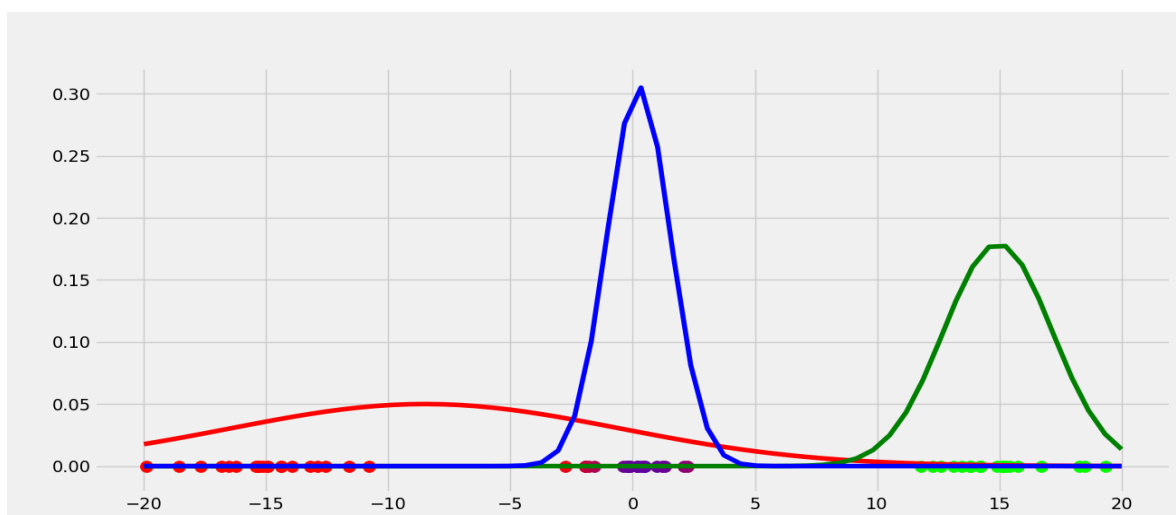


Figura 28 - Implementação 1D - Iteração 5

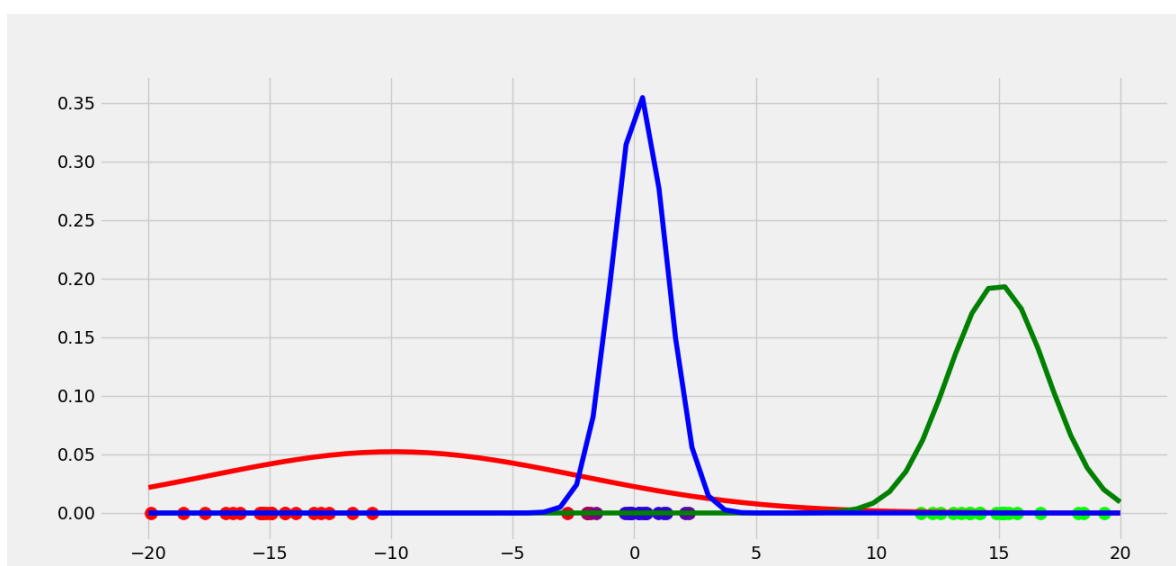


Figura 29 - Implementação 1D - Iteração 6

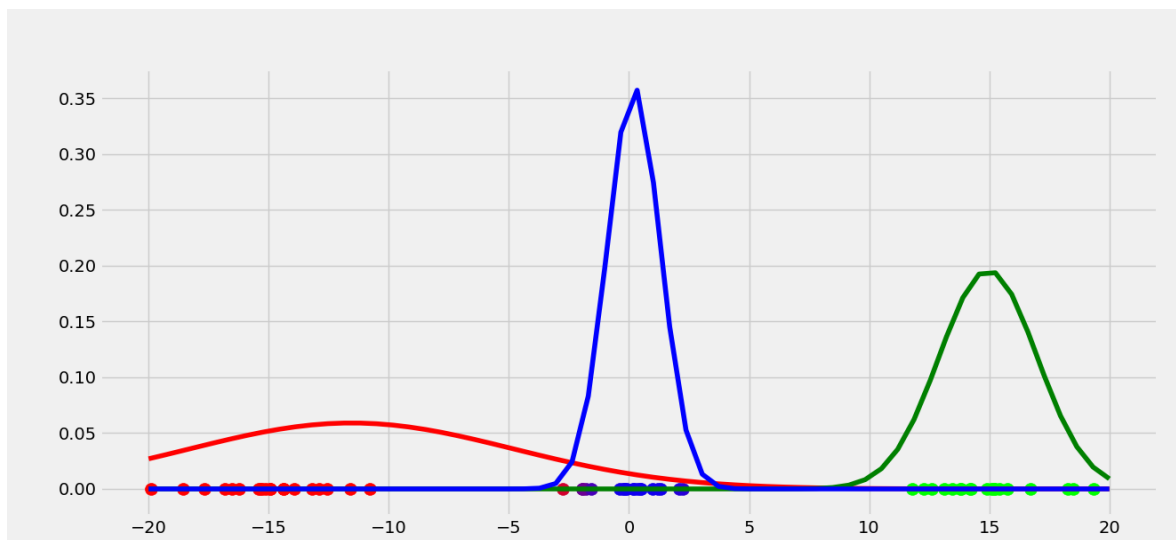


Figura 30 – Implementação 1D – Iteração 7

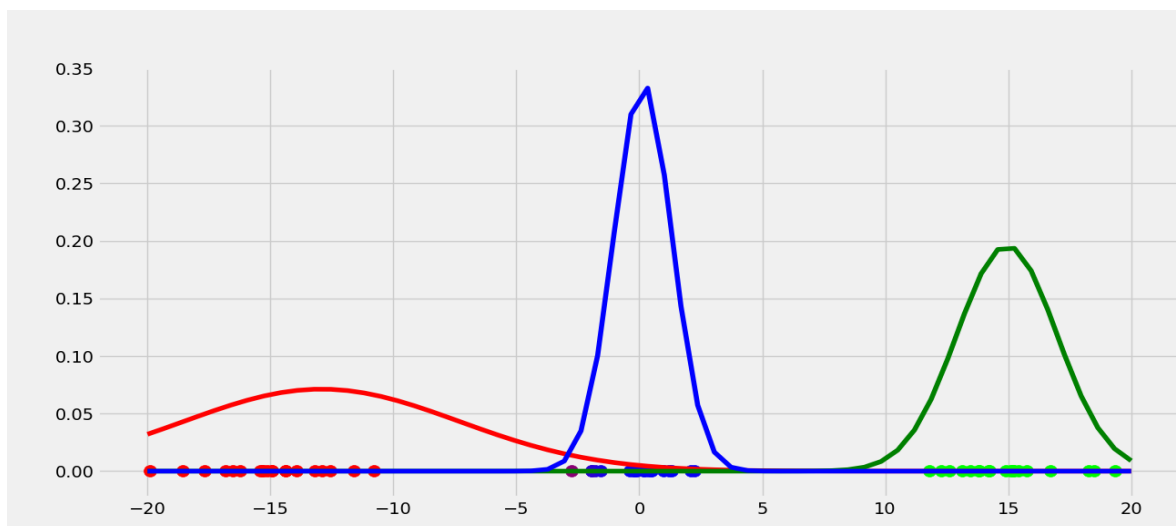


Figura 31 - Implementação 1D – Iteração 8

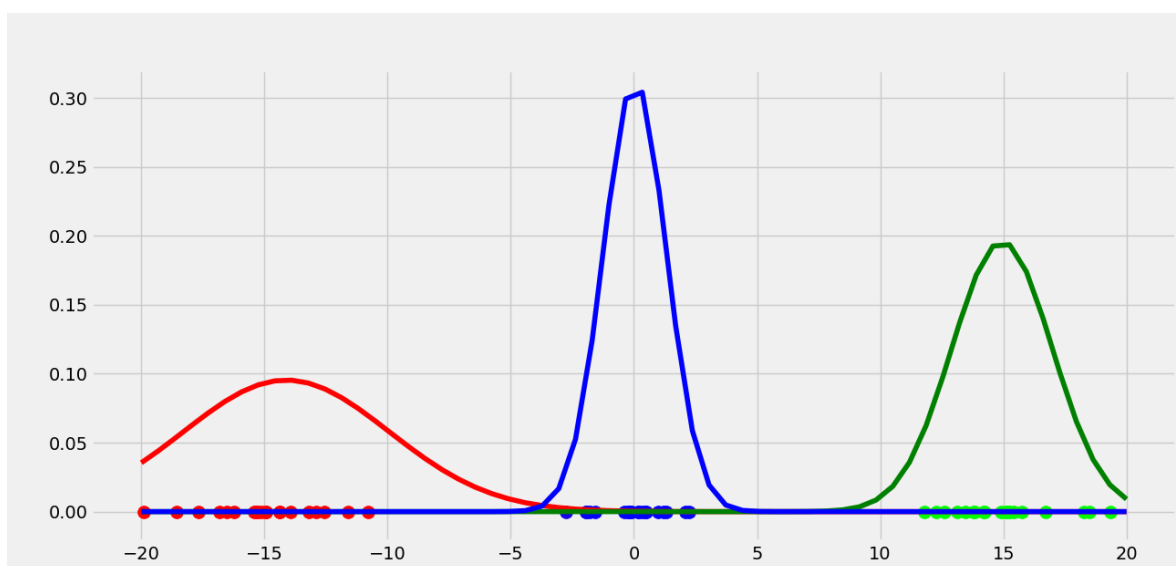


Figura 32 - Implementação 1D – Iteração 9

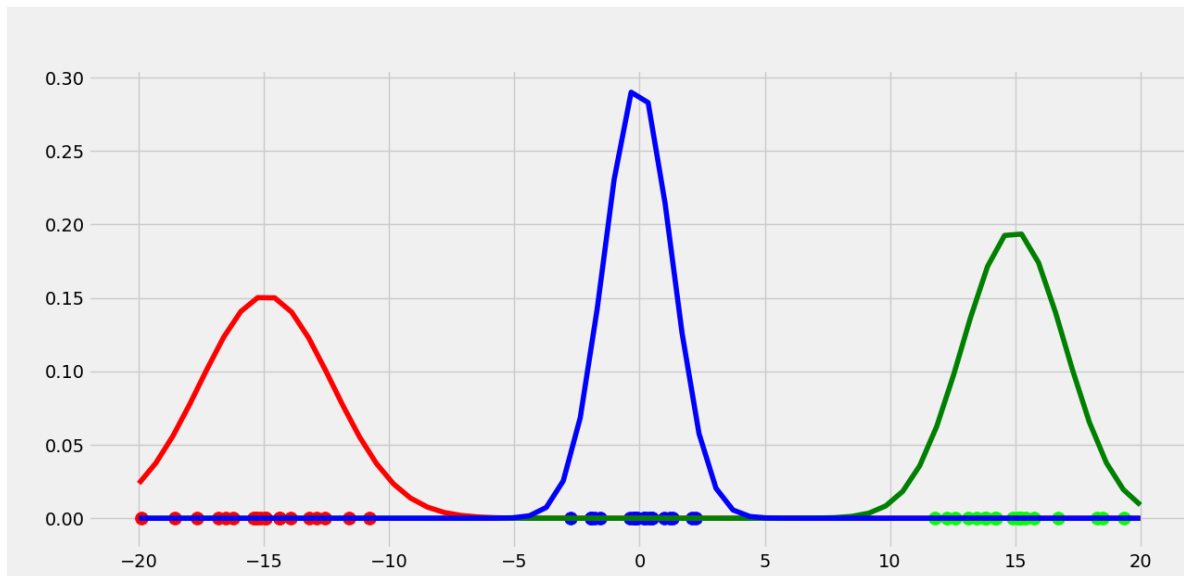


Figura 33 - Implementação 1D – Iteração 10

### 5.2.2 CÓDIGO CASO 1D

```
# Módulos
import matplotlib.pyplot as plt
from matplotlib import style
style.use('fivethirtyeight')
import numpy as np
from scipy.stats import norm
from math import sqrt
np.random.seed(0)

#####
# Gerando dados #
#####
X = np.linspace(-5,5,num=20)
X_1 = X*np.random.rand(len(X)) + 15 # Cluster 1
X_2 = X*np.random.rand(len(X)) - 15 # Cluster 2
X_3 = X*np.random.rand(len(X)) # Cluster 3
X_tot = np.stack((X_1,X_2,X_3)).flatten() # Combina os clusters em uma lista

#####
# Gaussian Mixtrure 1D #
#####

class GM1D:
    def __init__(self,X,iterations):
        self.iterations = iterations
        self.X = X
        self.mu = None # Médias
        self.pi = None # Pi's
        self.var = None # Variâncias
```

```

def run(self):
    # Instacia randomicamente (eu escolhi) os valores de mu, pi e var
    self.mu = [-8, 8, 5]
    self.pi = [1/3, 1/3, 1/3]
    self.var = [25, 9, 1]

    for iter in range(self.iterations):

        #####
        # E-Step #
        #####

        r = np.zeros((len(self.X), 3)) # Matriz r para armazenar probabi
lidades
        # Cálculo da probabilidade de um dado x_i pertencer a uma gaussi
ana g
        for c, g, p in zip(range(3), [norm(loc=self.mu[0], scale=sqrt(se
lf.var[0])),
                                     norm(loc=self.mu[1], scale=sqrt(se
lf.var[1])),
                                     norm(loc=self.mu[2], scale=sqrt(se
lf.var[2]))], self.pi):
            r[:,c] = p*g.pdf(self.X) # Probabilidade dos dados X_tot per
tencerem à Gaussiana c
            # Normalizar as probabilidades de tal forma que cada linha de r
some igual a 1 e adicionar pesos (pi) para cada probabilidade
            for i in range(len(r)):
                r[i] = r[i] / (np.sum(r,axis=1)[i]) # Precisa do np.sum(pi)
            ??????

        # Plot dos dados
        fig = plt.figure(figsize=(10,10))
        ax0 = fig.add_subplot(111)
        for i in range(len(r)):
            ax0.scatter(self.X[i], 0, c=np.array([r[i][0],r[i][1],r[i][2
]]), s=100)

        # Plot das Gaussianas
        for g, c in zip([norm(loc=self.mu[0], scale=sqrt(self.var[0])).p
df(np.linspace(-20,20,num=60).reshape(60,1)),
                        norm(loc=self.mu[1], scale=sqrt(self.var[1])).p
df(np.linspace(-20,20,num=60).reshape(60,1)),
                        norm(loc=self.mu[2], scale=sqrt(self.var[2])).p
df(np.linspace(-20,20,num=60).reshape(60,1))],
                        ['r','g','b']):
            ax0.plot(np.linspace(-20,20,num=60),g,c=c)

        #####

```

```

# M-Step #
#####

# Cálculo do m_c
m_c = [0]*len(r[0])
for c in range(len(r[0])):
    m = np.sum(r[:,c])
    m_c[c] = m # Cálculo de m_c para cada cluster

# Cálculo do pi_c
for k in range(len(m_c)):
    self.pi[k] = (m_c[k]/np.sum(m_c)) # Para cada cluster, calcular a fração de pontos que fazem parte do cluster

# Cálculo de mu_c
self.mu = np.sum(self.X.reshape(len(self.X),1)*r, axis=0)/m_c

# Cálculo de var_c
#var_c = [0]*len(r[0])
for c in range(len(r[0])):
    #var_c[c] = (1/m_c[c])*np.dot((np.array(r[:,c]).reshape(60,1))*(self.X.reshape(len(self.X),1)-self.mu[c])).T,(self.X.reshape(len(self.X),1)-self.mu[c]))

    for i in range(len(self.X)):
        self.var[c] += r[i][c]*((self.X[i] - self.mu[c])**2)
    self.var[c] = self.var[c] / m_c[c]

plt.show()

GM1D = GM1D(X_tot, 10)
GM1D.run()

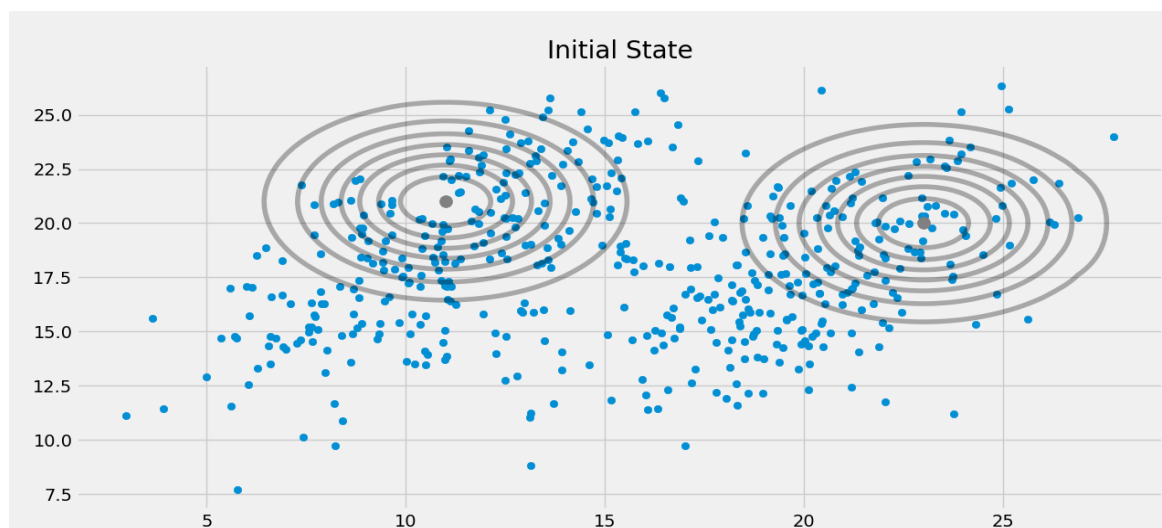
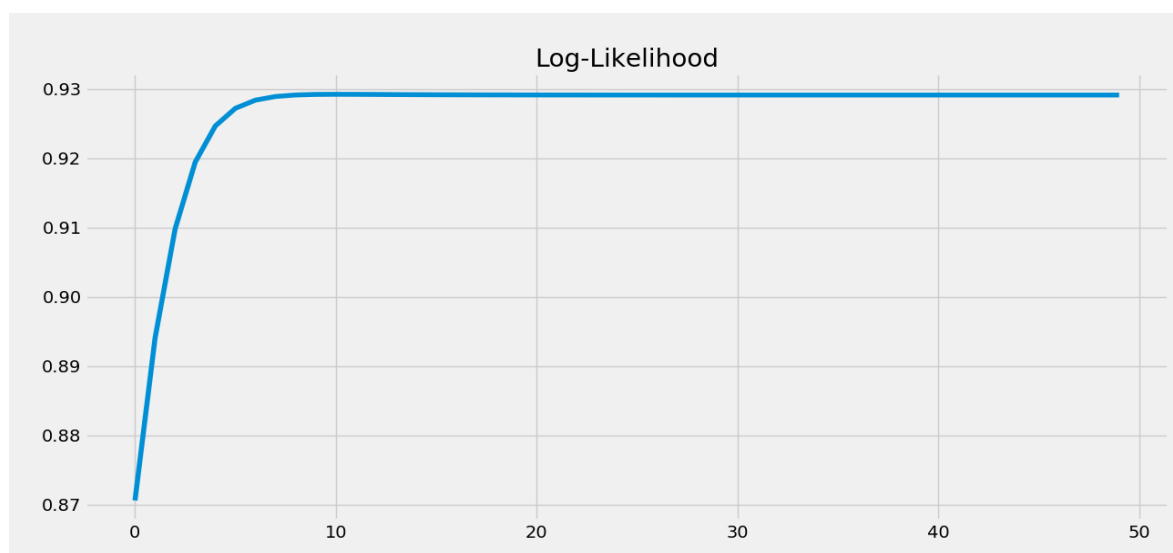
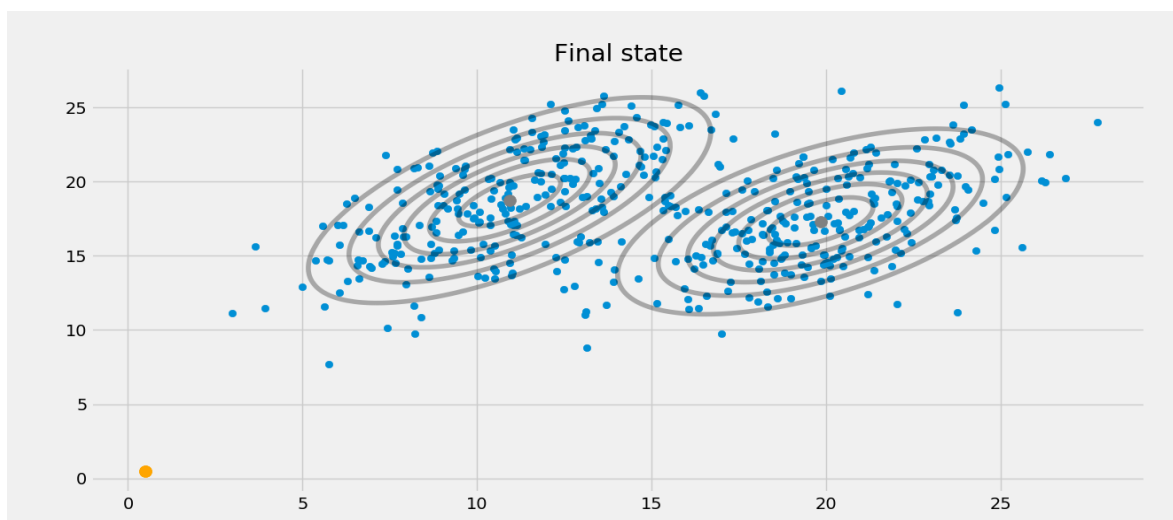
```

Figura 34 - Código em Python para implementação do EM para o caso 1D

### 5.2.3 APLICAÇÃO DO EM: CASO 2D

Para este caso de implementação, a base de dados foi gerada de tal forma que dois clusters podem ser verificados. Sendo assim, para este caso, ao se implementar o algoritmo, o objetivo é dividir os dados em dois clusters modelados cada um por uma Gaussiana, porém agora em um plano bidimensional. O algoritmo realiza 50 iterações. O resultado da implementação do algoritmo para este caso pode ser observado nas figuras a seguir. São apresentados o estado inicial antes da aplicação do algoritmo e o estado final após aplicação. Também é apresentado o gráfico de log-verossimilhança ao longo das iterações. Após as 50 iterações nota-se que as Gaussianas se ajustaram ao seus respectivos clusters e que o gráfico de verossimilhança atinge convergência.



**Figura 35 - Estado inicial****Figura 36 - Curva de log-verossimilhança****Figura 37 - Estado final**

### 5.2.4 CÓDIGO CASO 2D

```
# Módulos

import matplotlib.pyplot as plt
from matplotlib import style
style.use('fivethirtyeight')
from sklearn.datasets.samples_generator import make_blobs
import numpy as np
from scipy.stats import multivariate_normal
np.random.seed(0)

# Criar dataset
X,Y = make_blobs(cluster_std=1.5, random_state=20, n_samples=500, centers=2)
# Esticar datasets de forma a gerar dados elipsóides
X = np.dot(X, np.random.randn(2,2)) # RandomState(0) ?????

# Classe Guassian Mixture Models
class GMM:
    def __init__(self, X, number_of_sources, iterations):
        self.iterations = iterations
        self.number_of_sources = number_of_sources
        self.X = X
        self.mu = None
        self.pi = None
        self.cov = None
        self.XY = None

    # Função que roda o algoritmo
    def run(self):
        self.reg_cov = 1e-6*np.identity(len(self.X[0]))
        x,y = np.meshgrid(np.sort(self.X[:,0]), np.sort(self.X[:,1]))
        self.XY = np.array([x.flatten(), y.flatten()]).T

    # Definir valores iniciais para mu, covariância e valores de pi
        self.mu = np.random.randint(min(self.X[:,0]), max(self.X[:,0]), size
=(self.number_of_sources,len(self.X[0]))) # retorna matriz 3x2
        self.cov = np.zeros((self.number_of_sources, len(X[0]), len(X[0])))
        for dim in range(len(self.cov)):
            np.fill_diagonal(self.cov[dim], 5)
        self.pi = np.ones(self.number_of_sources)/self.number_of_sources
        log_likelihoods = [] # Lista para armazenar valores de verossimilhan
ça em cada iteração

    # Plot dos estados iniciais
    fig = plt.figure(figsize=(10,10))
    ax0 = fig.add_subplot(111)
    ax0.scatter(self.X[:,0], self.X[:,1])
    ax0.set_title('Initial State')
```

```

for m, c in zip(self.mu, self.cov):
    c += self.reg_cov
    multi_normal = multivariate_normal(mean=m, cov=c)
    ax0.contour(np.sort(self.X[:,0]),np.sort(self.X[:,1]),multi_norm
al.pdf(self.XY).reshape(len(self.X),len(self.X)),colors='black',alpha=0.3)
    ax0.scatter(m[0],m[1],c='grey',zorder=10,s=100)
plt.show()

for i in range(self.iterations):

    # E-Step
    # Contabilizar a probabilidade de cada ponto pertencer a cada cl
uster
    r_ic = np.zeros((len(self.X),len(self.cov)))
    for m, co, p, r in zip(self.mu, self.cov, self.pi, range(len(r_i
c[0]))):
        co += self.reg_cov
        mn = multivariate_normal(mean=m, cov=co)
        r_ic[:, r] = p*mn.pdf(self.X) / np.sum([pi_c*multivariate_no
rmal(mean=mu_c, cov=cov_c).pdf(self.X)
                                for pi_c, mu_c, cov_c in zip
(self.pi, self.mu, self.cov+self.reg_cov)], axis=0)
        # M-Step
        # Atualizar os valores de mu, cov e pi de acordo o cálculo de r_
ic
        self.mu = []
        self.cov = []
        self.pi = []
        for c in range(len(r_ic[0])):
            # mu
            m_c = np.sum(r_ic[:,c], axis=0)
            mu_c = (1/m_c)*np.sum(self.X*r_ic[:,c].reshape(len(self.X),1
), axis=0)

            self.mu.append(mu_c)
            # cov
            self.cov.append(((1/m_c)*np.dot((np.array(r_ic[:,c])).reshape
(len(self.X),1)*(self.X-mu_c)).T,
            (self.X-mu_c)))+self.reg_cov)
            # pi
            self.pi.append(m_c/np.sum(r_ic))

        # Log Likelihood
        log_likelihoods.append(np.log(np.sum([k*multivariate_normal(self
.mu[i],self.cov[j]).pdf(self.X) for k,i,j in
                                                    zip(
self.pi,range(len(self.mu)),range(len(self.cov))]))))

```

```

# Plot do Likelihood (Verossimilhança)
fig2 = plt.figure(figsize=(10,10))
ax1 = fig2.add_subplot(111)
ax1.set_title('Log-Likelihood')
ax1.plot(range(0,self.iterations,1),log_likelihoods)
plt.show()

# Função para prever a qual cluster um ponto pertence
def predict(self,Y):

    # Plot do estado final após todas as iterações
    fig3 = plt.figure(figsize=(10,10))
    ax2 = fig3.add_subplot(111)
    ax2.scatter(self.X[:,0],self.X[:,1])
    for m,c in zip(self.mu,self.cov):
        multi_normal = multivariate_normal(mean=m,cov=c)
        ax2.contour(np.sort(self.X[:,0]),np.sort(self.X[:,1]),multi_normal.pdf(self.XY).reshape(len(self.X),len(self.X)),colors='black',alpha=0.3)
        ax2.scatter(m[0],m[1],c='grey',zorder=10,s=100)
        ax2.set_title('Final state')
        for y in Y:
            ax2.scatter(y[0],y[1],c='orange',zorder=10,s=100)

    # Prediction
    prediction = []
    for m,c in zip(self.mu,self.cov):
        prediction.append(multivariate_normal(mean=m,cov=c).pdf(Y)/np.sum([multivariate_normal(mean=mean,cov=cov).pdf(Y) for mean,cov in zip(self.mu,self.cov)]))
    plt.show()
    return prediction

# Run
GMM = GMM(X,2,50)
GMM.run()
print(GMM.predict([[0.5,0.5]]))

```

Figura 38 - Código em Python para implementação do EM para o caso 2D

### 5.3 BIBLIOGRAFIA

DEMPSTER, A. P.; LAIRD, N. M.; RUBIN, D. B. Maximum likelihood from incomplete data via the EM algorithm. *Journal of the Royal Statistical Society: Series B (Methodological)*, v. 39, n. 1, p. 1–22, 1977

WU, Xindong; KUMAR, Vipin (Ed.). *The top ten algorithms in data mining*. CRC press, 2009.

## 6 ADABOOST

O AdaBoost é um algoritmo de aprendizado de máquina que faz parte da categoria de modelos de aprendizado em conjunto (*ensemble learning*). Comparado às abordagens mais comuns de aprendizado de máquina, que tentam gerar um único aprendiz dos dados de treinamento, os métodos de aprendizado em conjunto tentam construir um conjunto de aprendizes bases que serão utilizados em conjunto para assim alcançar resultados mais precisos. Os aprendizes bases são normalmente gerados a partir de algoritmos de aprendizado comuns, os quais podem ser uma árvore de decisão, uma rede neural, ou outros tipos de algoritmos de aprendizado de máquina. A característica que mais chama atenção para algoritmos deste tipo é a sua capacidade de realizar o *boost* de aprendizes fracos (*weak learners*), ou seja, impulsionar estes aprendizes a se tornarem de forma conjunta em aprendizes fortes (*strong learners*) capazes de realizar previsões precisas (WU; KUMAR, 2009).

O algoritmo Adaboost, abreviação de *Adaptive Boosting*, foi o primeiro algoritmo prático proposto por Freund e Schapire em 1996 durante suas pesquisas de desenvolvimento de algoritmos de boosting e é considerado um dos mais importantes métodos de aprendizado em conjunto. Ele tem como foco problemas de classificação e tem como ideia base a conversão de um grupo de classificadores fracos em um classificador forte.

### 6.1 DESCRIÇÃO DO ALGORITMO ADABOOST

Suponha um problema de classificação binária que tem como objetivo a classificação de amostras como positivas ou negativas:

$$x_i \in \mathbb{R}^d, y_i \in \{-1, 1\}$$

Onde -1 denota a classe negativa e 1 denota a classe positiva.

Suponha agora que, por falta de sorte, só um classificador fraco  $h_1$  está disponível e que este possui uma precisão ligeiramente maior que palpites aleatórios sobre uma distribuição de amostras  $D$ . É óbvio que este não é o classificador com eficiência desejada e que é necessário melhorá-lo. Uma ideia natural é tentar corrigir os erros realizados por  $h_1$ . Para isso, uma nova distribuição de amostras  $D'$  é derivada de  $D$  de forma a tornar os erros de  $h_1$  mais evidentes. Com isso, um novo classificador  $h_2$  é treinado a partir de  $D'$ . Suponha que novamente, por falta de sorte,  $h_2$  também é um classificador fraco. Dado que  $D'$  foi derivado de  $D$ ,  $h_2$  poderá atingir uma performance melhor que a de  $h_1$  nos pontos em  $D$  onde  $h_1$  cometeu erros, sem afetar os pontos em que  $h_1$  performou corretamente. Portanto, combinando  $h_1$  e  $h_2$  de modo apropriado, o classificador combinado realizará menos erros que  $h_1$ . Sendo assim, ao repetir este processo, um classificador eficiente resultante da combinação de outros classificadores fracos podem ser obtido em relação a  $D$  (WU; KUMAR, 2009).

Este processo é o conceito base aplicado para algoritmos de *boosting*. O procedimento genérico para implementação de algoritmos de *boosting* é apresentado na Figura 39.

---

```

Input: Instance distribution  $\mathcal{D}$ ;
        Base learning algorithm  $L$ ;
        Number of learning rounds  $T$ .

Process:
1.  $\mathcal{D}_1 = \mathcal{D}$ .           % Initialize distribution
2. for  $t = 1, \dots, T$ :
3.    $h_t = L(\mathcal{D}_t)$ ;       % Train a weak learner from distribution  $\mathcal{D}_t$ 
4.    $\epsilon_t = \Pr_{x \sim \mathcal{D}_t} I[h_t(x) \neq y]$ ; % Measure the error of  $h_t$ 
5.    $\mathcal{D}_{t+1} = \text{AdjustDistribution}(\mathcal{D}_t, \epsilon_t)$ 
6. end
Output:  $H(x) = \text{CombineOutputs}(\{h_t(x)\})$ 

```

---

Figura 39 - Procedimento genérico para implementação de algoritmos boosting (WU; KUMAR, 2009).

O algoritmo Adaboost é um algoritmo de aprendizado em conjunto via boosting que segue este procedimento, porém com alguns detalhes que o diferenciam dos outros algoritmos do mesmo tipo. O AdaBoost atua de forma a gerar uma sequência de hipóteses (classificadores) que posteriormente são combinadas por meio de pesos, ou seja, uma combinação ponderada aditiva na forma de:

$$H(x) = \sum_{t=1}^T \alpha_t h_t(x)$$

Por meio deste conceito, o AdaBoost resolve dois problemas, que são, como gerar hipóteses (classificadores)  $h_t$ 's e como determinar seus correspondentes pesos  $\alpha_t$ 's. Sendo assim, o processo para a aplicação do AdaBoost para uma base de dados  $D = \{(x_i, y_i)\} (i \in \{1, \dots, m\})$  e um algoritmo de aprendizado  $L$  segue o seguinte passo a passo:

- Inicializar a distribuição dos valores pesos para cada amostra de forma uniforme:

$$D_1(i) = \frac{1}{m}, i = 1, \dots, m$$

- Para cada rodada de aprendizado  $t = 1, \dots, T$ :
  1. Treinar um classificador fraco  $h_t$  para a base de dados utilizando o algoritmo de aprendizado  $L$  e a distribuição de pesos  $D$

$$h_t = L(D, D_t)$$

2. Medir os erros realizados pelo classificador treinado  $h_t$ :

$$\epsilon_t = \Pr_{x \sim D_t} I[h_t(x) \neq y]$$

Onde  $I[\cdot]$  é uma função de indicação que retorna o valor 1 se sua expressão é verdadeira e 0 caso contrário.

3. Calcular o peso para o classificador fraco  $h_t$ :

$$\alpha_t = \frac{1}{2} \ln \left( \frac{1 - \epsilon_t}{\epsilon_t} \right)$$

Para qualquer classificador com precisão acima de 50%, o seu peso correspondente é positivo. Quanto mais preciso o classificador, maior

será o seu peso. Enquanto que para o classificador com precisão menor que 50%, o seu peso correspondente é negativo.

4. Atualizar a distribuição de valores de pesos para cada amostra:

$$D_{t+1}(i) = \frac{D_t(i) \exp(-\alpha_y y_i h_t(x_i))}{Z_t}$$

Onde  $Z_t$  é um fator de normalização que garante que a soma dos pesos das amostras seja igual a 1. Se um erro de classificação é realizado por um classificador de peso positivo, o termo exponencial no numerador será sempre maior que 1. Portanto, casos de erro de classificação são atualizados com pesos maiores depois de uma iteração. A mesma lógica se aplica para classificadores com peso negativo.

- Depois das  $T$  rodadas de aprendizado é possível obter a predição final pelo sinal da soma ponderada da predição de cada classificador treinado durante as  $T$  rodadas de treinamento:

$$H(x) = \text{sign} \left( \sum_{t=1}^T \alpha_t h_t(x) \right)$$

O pseudocódigo do algoritmo AdaBoost que resume este processo é ilustrado na Figura 40.

---

**Input:** Data set  $D = \{(x_1, y_1), (x_2, y_2), \dots, (x_m, y_m)\}$ ;  
 Base learning algorithm  $L$ ;  
 Number of learning rounds  $T$ .

**Process:**

1.  $\mathcal{D}_1(i) = 1/m$ .      % Initialize the weight distribution
2. **for**  $t = 1, \dots, T$ :
3.     $h_t = L(D, \mathcal{D}_t)$ ;      % Train a learner  $h_t$  from  $D$  using distribution  $\mathcal{D}_t$
4.     $\epsilon_t = \Pr_{x \sim \mathcal{D}_t} [h_t(x) \neq y]$ ;      % Measure the error of  $h_t$
5.    **if**  $\epsilon_t > 0.5$  **then break**
6.     $\alpha_t = \frac{1}{2} \ln \left( \frac{1 - \epsilon_t}{\epsilon_t} \right)$ ;      % Determine the weight of  $h_t$
7.     $\mathcal{D}_{t+1}(i) = \frac{\mathcal{D}_t(i)}{Z_t} \times \begin{cases} \exp(-\alpha_t) & \text{if } h_t(x_i) = y_i \\ \exp(\alpha_t) & \text{if } h_t(x_i) \neq y_i \end{cases}$   
     $\frac{\mathcal{D}_t(i) \exp(-\alpha_t y_i h_t(x_i))}{Z_t}$       % Update the distribution, where  
         %  $Z_t$  is a normalization factor which  
         % enables  $\mathcal{D}_{t+1}$  to be distribution
8.    **end**

**Output:**  $H(x) = \text{sign} \left( \sum_{t=1}^T \alpha_t h_t(x) \right)$

---

Figura 40 - Pseudocódigo do algoritmo AdaBoost (WU; KUMAR, 2009).

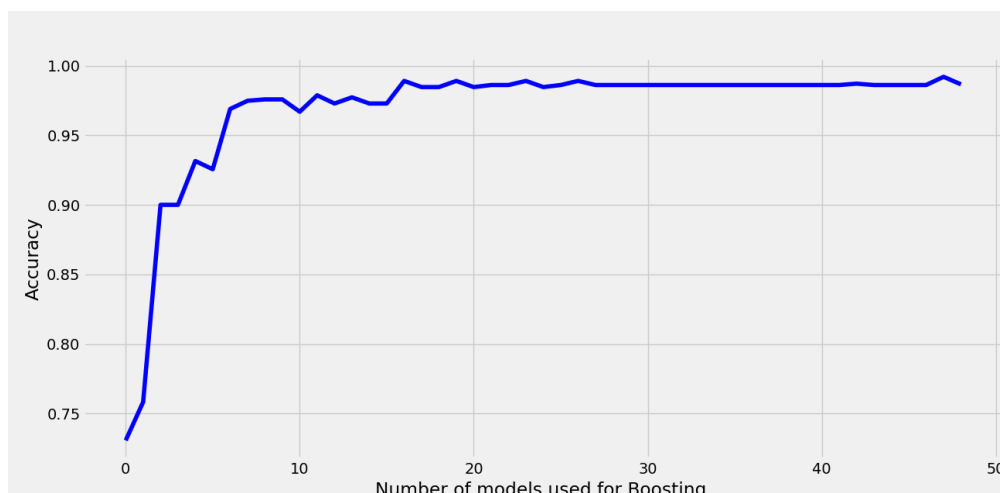
## 6.2 APLICAÇÃO DO ALGORITMO ADABOOST

Para a implementação do algoritmo AdaBoost optou-se por utilizar uma base de dados reais que apresenta atributos que permitem caracterizar se um cogumelo é venenoso ou não (<https://archive.ics.uci.edu/ml/machine-learning-databases/mushroom/>). A base de dados apresenta 22 atributos e 8124 amostras. A implementação do algoritmo

AdaBoost para esse conjunto de dados tem como o objetivo construir um classificador binário forte que consiga classificar de forma precisa um cogumelo como venenoso ou não venenoso. Para implementar o algoritmo, os passos apresentados na Figura 40 foram seguidos. O modelo de aprendizado escolhido para treinar os classificadores fracos foi o de árvore de decisão de monocamada. O algoritmo AdaBoost foi testado para o caso de um classificador forte treinado com 50 classificadores fracos ( $T = 50$ ) e para o caso de um classificador forte treinado com 400 classificadores fracos ( $T = 400$ ). Os resultados dos testes são apresentados a seguir:

- 50 classificadores

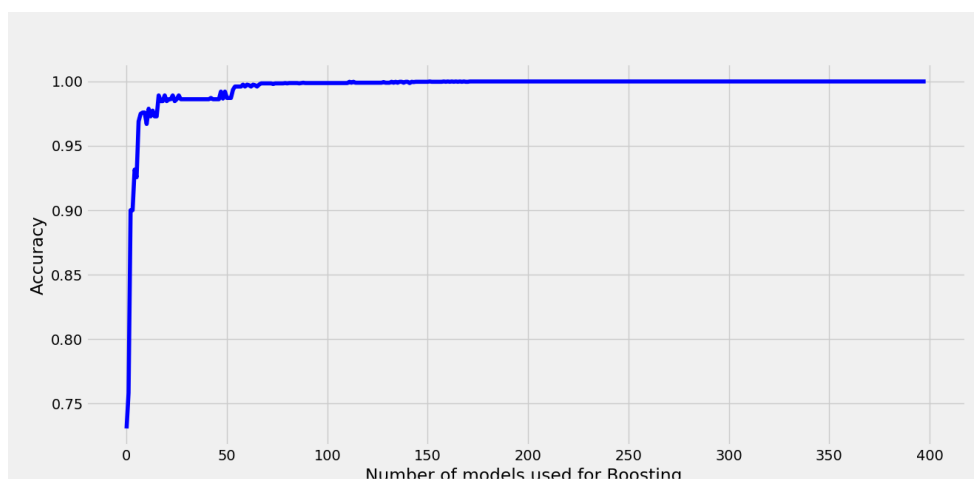
Com um classificador forte treinado com 50 classificadores fracos, o modelo AdaBoost conseguiu classificar de forma correta 98,67% das amostras testes de cogumelo.



**Figura 41 - Performance do classificador forte com 50 classificadores fracos**

- 400 classificadores

Com um classificador forte treinado com 400 classificadores fracos, o modelo AdaBoost conseguiu classificar de forma correta 100,00% das amostras testes de cogumelo.



**Figura 42 - Performance do classificador forte com 400 classificadores fracos**



## 6.2.1 CÓDIGO

```
# Importação do pacotes que serão utilizados
import numpy as np
import matplotlib.pyplot as plt
import pandas as pd
from matplotlib import style
style.use('fivethirtyeight')
from sklearn.tree import DecisionTreeClassifier
from sklearn.preprocessing import LabelEncoder
from sklearn.model_selection import cross_validate
#Dataset: https://archive.ics.uci.edu/ml/machine-learning-
databases/mushroom/agaricus-lepiota.data

# Importação dos dados
dataset = pd.read_csv('mushroom_dataset.csv', header=None)
dataset = dataset.sample(frac=1) # Randomiza a ordem dos dados
dataset.columns = ['target', 'cap-shape', 'cap-surface', 'cap-
color', 'bruises', 'odor', 'grill-attachment', 'grill-spacing',
                  'grill-size', 'grill-color', 'stalk-shape', 'stalk-
root', 'stalk-surface-above-ring', 'stalk-surface-below-ring',
                  'stalk-color-above-ring', 'stalk-color-below-
ring', 'veil-type', 'veil-color', 'ring-number', 'ring-type',
                  'spore-print-
color', 'population', 'habitat'] # Nomeia as colunas

# Tratamento dos dados
# Conversão dos dados String para Integer dado que o sklearn DecisionTreeCl
assifier só aceita valores numéricos
for label in dataset.columns:
    dataset[label] = LabelEncoder().fit(dataset[label]).transform(dataset[la
bel])
    # Todos os dados são convertidos para valores numéricos

# Classe que implementa o algoritmo AdaBoost
class AdaBoost:
    def __init__(self, dataset, T, test_dataset):
        self.dataset = dataset
        self.T = T
        self.test_dataset = test_dataset
        self.alphas = None
        self.models = None

    # Função para o treinamento dos classificadores fracos
    def fit(self):
        # Valores dos atributos
        X = self.dataset.drop('target', axis=1) # Remove a coluna 'target'
        # Valores de classe
```

```

        Y = self.dataset['target'].where(dataset['target']==1, other=-1) # Valores em que a condição é falso são substituídos pelo valor em other
        # Inicializar os pesos para cada amostra dos dados como  $w_i = 1/N$  e criar uma tabela que compute esta Evaluation de pesos
        Evaluation = pd.DataFrame(Y.copy()) # Cria uma cópia de Y
        Evaluation['weights'] = 1 / len(self.dataset) # Define  $w_i$  (peso de cada amostra) inicial como  $1/N$ 
        models = [] # Lista para armazenar os modelos de classificadores gerados
        alphas = [] # Lista para armazenar os pesos correspondentes aos classificadores treinados

        for t in range(self.T):
            # Treinamento de um Decision Stump (Toco de Decisão)
            # Árvore de Decisão de 1 camada será utilizada como modelo de classificador
            Tree_model = DecisionTreeClassifier(criterion="entropy", max_depth=1) # Modelo de árvore de decisão com 1 camada
            # Construção de um classificador árvore de decisão para os dados de treinamento
            model = Tree_model.fit(X, Y, sample_weight=np.array(Evaluation['weights']))
            # Armazenamento do modelo em uma lista de modelos
            models.append(model)
            # Teste do classificador
            predictions = model.predict(X) # Predição das classes dos dados de X
            # Adicionar valores a tabela Evaluation
            Evaluation['predictions'] = predictions
            Evaluation['evaluation'] = np.where(Evaluation['predictions'] == Evaluation['target'], 1, 0) # 1 para True e 0 para False
            Evaluation['misclassified'] = np.where(Evaluation['predictions'] != Evaluation['target'], 1, 0) # 1 para True e 0 para False

            # Cálculo do erro
            err = sum(Evaluation['weights']*Evaluation['misclassified'])
            # Cálculo dos valores de alpha (pesos dos classificadores)
            alpha = (0.5*np.log((1 - err)/ err))
            alphas.append(alpha)

            # Atualização os valores dos pesos das amostras
            Evaluation['weights'] *= np.exp(-alpha*Y*Evaluation['predictions'])
            Evaluation['weights'] /= sum(Evaluation['weights'])

        self.alphas = alphas
        self.models = models

```

```

    # Função para teste do modelo de classificador AdaBoost treinado (Conjun
to de classificadores fracos)
    def predict(self):
        # Geração de dados de teste (Apenas um rearranjo dos dados treina
mento)
        X_test = self.test_dataset.drop(['target'],axis=1).reindex(range(len
(self.test_dataset)))
        Y_test = self.test_dataset['target'].reindex(range(len(self.test_dat
aset))).where(self.dataset['target']==1,-1)

        accuracy = None
        predictions = [] # Lista para armazenamento de valores de predição

        for alpha,model in zip(self.alphas,self.models):
            # Predição das classes dos dados de teste
            prediction = alpha*model.predict(X_test)
            predictions.append(prediction)
            # Para gerar lista de valores de precisão para plotar precisão em fu
nção do número de classificadores utilizados
            accuracy = (np.sum(np.sign(np.sum(np.array(predictions),axis=0)) ==
Y_test.values)/len(predictions[0]))
            return accuracy

# ____Main____()

number_of_base_learners = 400 # Número de classificadores fracos utilizados
fig = plt.figure(figsize=(10,10))
ax0 = fig.add_subplot(111)
accuracy = [] # Lista para armazenar precisão de acordo com o número de clas
sificadores utilizados
# Aplicação do algoritmo para modelos com diferentes números de classificad
ores
for i in range(1,number_of_base_learners):
    model = AdaBoost(dataset,i,dataset)
    model.fit()
    accuracy.append(model.predict())
ax0.plot(range(len(accuracy)),accuracy,'-b')
ax0.set_xlabel('Number of models used for Boosting ')
ax0.set_ylabel('Accuracy')
print('With a number of ',number_of_base_learners,'base models we receive an
accuracy of ',accuracy[-1]*100,'%')
plt.show()

```

Figura 43 - Código em Python para implementação do AdaBoost

### 6.3 BIBLIOGRAFIA

SCHAPIRE, R. E. A brief introduction to boosting. IJCAI International Joint Conference on Artificial Intelligence, v. 2, n. 5, p. 1401–1406, 1999.

WU, Xindong; KUMAR, Vipin (Ed.). The top ten algorithms in data mining. CRC press, 2009.

## 7 KNN: K-NEAREST NEIGHBORS

k-Nearest Neighbors (kNN) é um algoritmo de classificação gerado por meio de aprendizado supervisionado. O algoritmo tem como objetivo determinar um grupo de  $k$  objetos presentes em um conjunto de dados de treinamento que são considerados os objetos mais próximos de um objeto teste a ser classificado. A classificação deste objeto teste é baseada na predominância de uma classe particular no grupo de  $k$  objetos determinados. De forma simplificada, o kNN classifica um objeto com a classe de seus objetos vizinhos mais próximo ou com a classe majoritária entre os seus objetos vizinhos mais próximos (WU; KUMAR, 2009).

A classificação por kNN é uma técnica de fácil compreensão e implementação, que pode ser aplicado com eficiência em diversas áreas de estudo.

### 7.1 DESCRIÇÃO DO ALGORITMO K-NEAREST NEIGHBORS

Dado um conjunto de dados de treinamento  $D$  e um objeto teste  $z$ , o qual é um vetor de valores de atributos que não possui classificação ainda, o algoritmo calcula a distância (ou similaridade) entre  $z$  e todos os objetos de treinamento para determinar sua lista de k-Nearest Neighbors ( $k$  objetos vizinhos mais próximos). Desta forma, a classificação de  $z$  é baseada na classe majoritariamente presente na lista dos  $k$  objetos mais próximos. Em caso de empate, a classe mais frequente é escolhida. O pseudocódigo para implementação do algoritmo é apresentado na Figura 44.

---

#### Algorithm 8.1 Basic kNN Algorithm

---

**Input :**  $D$ , the set of training objects, the test object,  $z$ , which is a vector of attribute values, and  $L$ , the set of classes used to label the objects

**Output :**  $c_z \in L$ , the class of  $z$

**foreach** object  $y \in D$  **do**

    | Compute  $d(z, y)$ , the distance between  $z$  and  $y$ ;

**end**

Select  $N \subseteq D$ , the set (neighborhood) of  $k$  closest training objects for  $z$ ;

$c_z = \operatorname{argmax}_{v \in L} \sum_{y \in N} I(v = \text{class}(c_y))$ ;

where  $I(\cdot)$  is an indicator function that returns the value 1 if its argument is true and 0 otherwise.

---

Figura 44 - Pseudocódigo do algoritmo k-Nearest Neighbors

Existem diversas peculiaridades do algoritmo que devem ser destacadas. A primeira é sobre a escolha do valor de  $k$ . Na Figura 45 pode-se observar o comportamento do algoritmo de acordo com o valor de  $k$ . Se  $k$  for muito pequeno, o resultado pode ser sensível a pontos isolados. Por outro lado, se  $k$  for muito grande a vizinhança definida como mais próxima pode conter muitos pontos de classes diferentes. Uma estimativa do melhor valor de  $k$  pode ser obtida por validação cruzada. Entretanto é importante destacar que  $k = 1$  pode ser capaz de ter uma performance parecida com o de outros valores de  $k$ , particularmente para pequenas bases de dados. Já para conjunto de dados maiores, valores maiores de  $k$  são mais resistentes a distúrbios.

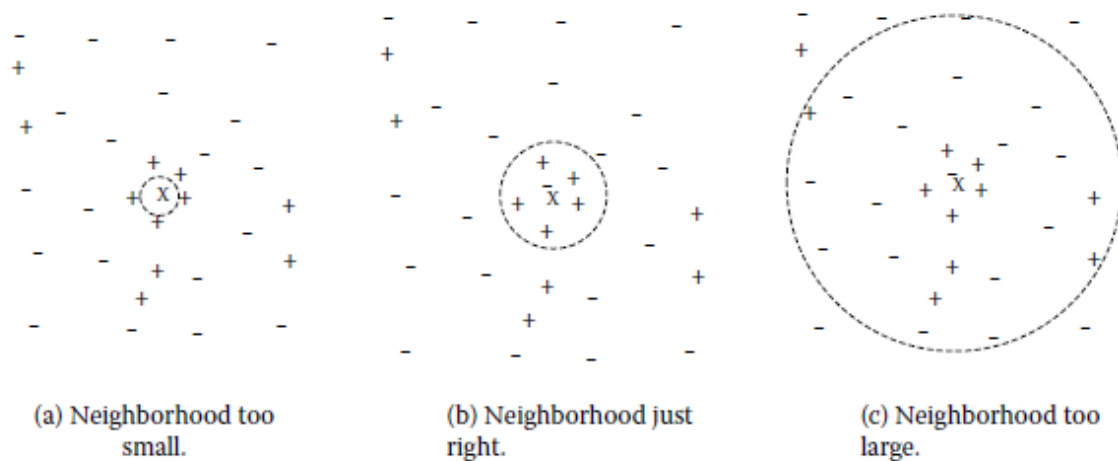


Figura 45 - k-Nearest Neighbors com valor pequeno, médio e grande de  $k$  (WU; KUMAR, 2009).

Outra peculiaridade que deve ser destacada é o caso em que os  $k$  objetos mais próximos do objeto teste são de classes diferentes. O método mais simples neste caso é realizar a classificação de acordo com a classe mais frequente, porém esta técnica pode gerar problemas quando as distâncias dos objetos vizinhos mais próximos variarem muito e o objeto mais próximo apresentar a classe mais confiável de ser adotada para o objeto analisado. Uma abordagem mais sofisticada é ponderar o voto de cada objeto de acordo com o valor de sua distância.

A escolha da forma a ser medida a distância é uma outra importante consideração a ser feita. Normalmente, a distância Euclidiana e distância de Manhattan são utilizadas. Para dois pontos,  $x$  e  $y$ , com  $n$  atributos, estas distâncias são obtidas pelas fórmulas:

$$d(x, y) = \sqrt{\sum_{k=1}^n (x_k - y_k)^2} \quad \text{Distância Euclidiana}$$

$$d(x, y) = \sqrt{\sum_{k=1}^n |x_k - y_k|} \quad \text{Distância de Manhattan}$$

Onde  $x_k$  e  $y_k$  são os  $k^{\text{ésimos}}$  atributos (componentes) de  $x$  e  $y$ , respectivamente.

Apesar dessas e várias outras medidas poderem ser utilizadas para computar a distância entre dois pontos, conceitualmente, a medida de distância mais desejável é aquela em que a menor distância entre dois objetos implica em uma verossimilhança alta de pertencer a mesma classe.

## 7.2 APLICAÇÃO DO ALGORITMO K-NEAREST NEIGHBORS

Dois problemas de classificação de dados foram utilizados para demonstrar a aplicação prática do algoritmo kNN. Para o primeiro caso implementado utilizou-se

uma simples base de dados gerada manualmente que permite observar a operação do algoritmo de forma didática. Já para o segundo caso implementado optou-se por utilizar uma base de dados obtida em um banco de dados público com o objetivo de observar e avaliar a performance do algoritmo em um caso real mais complexo.

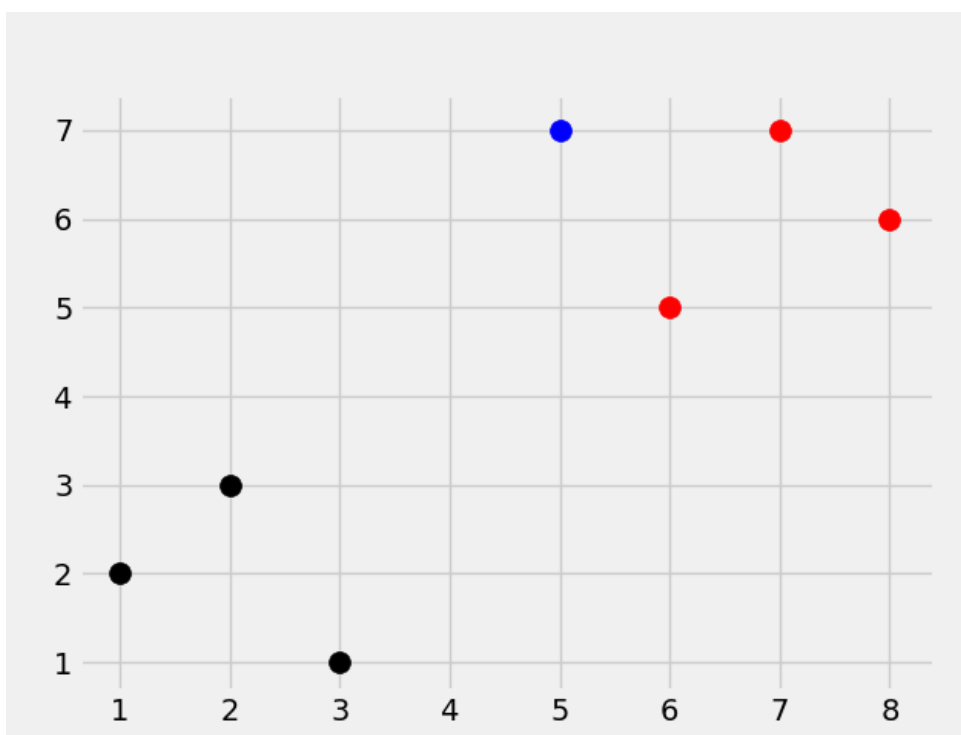
### 7.2.1 APLICAÇÃO DO KNN: CASO SIMPLES

Tal como foi dito anteriormente, para este caso de implementação optou-se por utilizar uma base de dados gerada manualmente para o treinamento do modelo de classificação. A base de dados gerada (Tabela 7) apresenta 6 amostras, 2 atributos ( $x_1$  e  $x_2$ ) e duas classes (k - preto ou r - vermelho). Além disso, uma amostra sem classificação (amostra 7) é gerada para o teste do modelo.

**Tabela 7 – Base de dados utilizada para implementação do kNN (caso simples)**

Sample	$x_1$	$x_2$	Class
1	1	2	k
2	2	3	k
3	3	1	k
4	6	5	r
5	7	7	r
6	8	6	r
7	5	7	?

A localização em gráfico de cada amostra pode ser observada na Figura 46.



**Figura 46 - Localização das amostras em gráfico 2D**

Ao aplicar o algoritmo kNN para a base de dados da Tabela 7, se torna possível classificar a amostra sem classe de acordo com a classe das  $k$  amostras mais próximas.

Tal como era esperado ao analisar a Figura 46, utilizando  $k = 3$  para o algoritmo kNN, a amostra (5, 7) é classificada como r – vermelho.

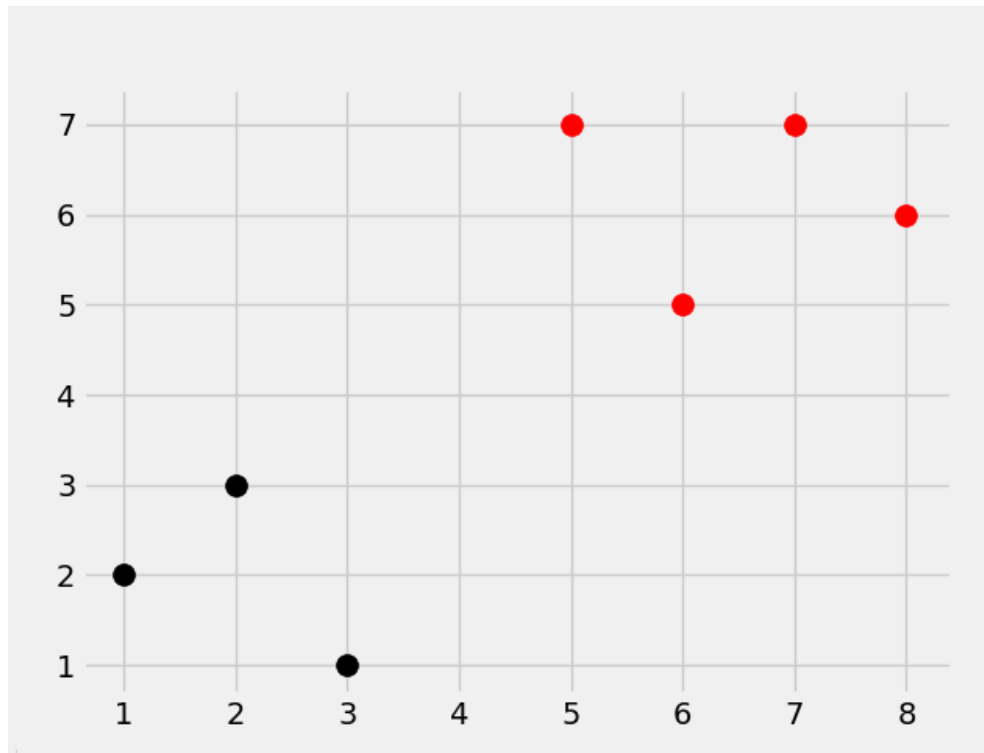


Figura 47 - Resultado da classificação da amostra analisada pelo algoritmo

### 7.2.2 CÓDIGO CASO SIMPLES

```
# Importação dos pacotes a serem utilizados
import numpy as np
import matplotlib.pyplot as plt
from matplotlib import style
import warnings
from math import sqrt
from collections import Counter
style.use('fivethirtyeight')

# Função para implementação do algoritmo
def k_nearest_neighbors(data, predict, k=3):
    # Warning caso o valor de k não seja coerente
    if len(data) >= k:
        warnings.warn('k is set to a value less than total voting objects!')
        distances = [] # Lista para armazenar valores de distância entre o dado
        # de predição e todos os dados de treino
        # Cálculo das distâncias Euclidianas entre o dado de predição e todos os
        # dados de treino
```



```

    for group in data:
        for feature in data[group]:
            euclidean_distance = np.linalg.norm(np.array(feature) - np.array
(predict)) # Cálculo de distância Euclideana
            distances.append([euclidean_distance, group])
        # Computação dos votos dos k dados mais próximos do dado de predição
        votes = [object[1] for object in sorted(distances)[:k]]
        # Classe resultante da votação
        vote_result = Counter(votes).most_common(1)[0][0]
        return vote_result

# __Main__():

# Geração dos dados a serem utilizados
dataset = {'k': [[1, 2],[2, 3],[3, 1]], 'r':[[6, 5], [7, 7], [8, 6]]} # Base
de dados com duas classes ('k' e 'r') e 6 pontos
new_features = [5, 7] # Dado a ser classificado pelo algoritmo

# Plotando os dados
for classe in dataset:
    for point in dataset[classe]:
        plt.scatter(point[0], point[1], s=100, c=classe)
# Em uma linha: [[plt.scatter(point[0], point[1], s=100, c=classe) for point
in dataset[classe]] for classe in dataset]

# Plotando dado a ser classificado
plt.scatter(new_features[0], new_features[1], s=100, c='b')
plt.show()

# Aplicação do algoritmo para dado a ser classificado
result = k_nearest_neighbors(dataset, new_features)

# Plotando o resultado
print(f'The class of the new featrure is {result}.')
for classe in dataset:
    for point in dataset[classe]:
        plt.scatter(point[0], point[1], s=100, c=classe)

# Plotando o dado classificado
plt.scatter(new_features[0], new_features[1], s=100, c=result)
plt.show()

```

Figura 48 - Código em Python da implementação do caso simples do kNN

### 7.2.3 APLICAÇÃO DO KNN: CASO REAL

Para este caso de implementação optou-se por utilizar uma base de dados obtida em um banco de dados público ([https://archive.ics.uci.edu/ml/datasets/breast+cancer+wisconsin+\(original\)](https://archive.ics.uci.edu/ml/datasets/breast+cancer+wisconsin+(original))). A base de dados apresenta 699 amostras de análises clínicas relacionadas ao diagnóstico de câncer de mama, com 10 atributos (características citológicas) e uma classificação que diz se a amostra caracteriza um tumor como

maligno ou benigno. O kNN foi implementado para esta base de dados de forma a obter um modelo capaz de identificar se uma amostra clínica caracteriza um tumor na mama como maligno ou benigno.

Com intuito de verificar a eficiência do algoritmo na classificação de novas amostras, a base de dados foi dividida em dois conjuntos, sendo um conjunto de dados para treinamento (80% dos dados originais) e outro conjunto de dados de teste (20% dos dados originais). Ao testar o algoritmo (com valor de  $k = 3$ ) para o conjunto de dados de teste verificou-se em uma série de testes uma taxa de acerto em torno de 95%

#### 7.2.4 CÓDIGO CASO REAL

```
# Importação dos pacotes a serem utilizados
import numpy as np
from math import sqrt
import warnings
from collections import Counter
import pandas as pd
import random

#Dataset: https://archive.ics.uci.edu/ml/datasets/breast+cancer+ wisconsin+(original)

# Função para implementação do algoritmo
def k_nearest_neighbors(data, predict, k=3):
    # Wraning caso o valor de k não seja coerente
    if len(data) >= k:
        warnings.warn('k is set to a value less than total voting objects!')
    distances = [] # Lista para armazenar valores de distância entre o dado
de predição e todos os dados de treino
    # Cálculo das distâncias Euclidianas entre o dado de predição e todos os
dados de treino
    for group in data:
        for feature in data[group]:
            euclidean_distance = np.linalg.norm(np.array(feature) - np.array
(predict)) # Cálculo de distância Euclidiana
            distances.append([euclidean_distance, group])
    # Computação dos votos dos k dados mais próximos do dado de predição
    votes = [object[1] for object in sorted(distances)[:k]]
    # Classe resultante da votação
    vote_result = Counter(votes).most_common(1)[0][0]
    return vote_result

# Importação dos dados
df = pd.read_csv('breast_cancer_wisconsin_dataset.csv')
df.replace('?', -
99999, inplace=True) # Torna os dados com informações desconhecida em outlie
rs
df.drop(['id'], axis=1, inplace=True) # Remove a coluna de id
```

```

full_data = df.astype(float).values.tolist() # Converte a tabela panda para
uma lista de listas
# .astype converte os dados para float
# .values converte os valores para uma numpy array
# .tolist() retorna um lista com os valores

# Embaralha os dados
random.shuffle(full_data)

# Split de dados de treino e dados de teste
test_size = 0.2 # 20% dos dados será utilizado para teste
train_set = {2:[], 4:[]} # Dados de treino
test_set = {2:[], 4:[]} # Dados de teste
train_data = full_data[:-
int(test_size*len(full_data))] # Todos os dados menos os últimos 20%
test_data = full_data[-
int(test_size*len(full_data)):] # Os últimos 20% dos dados

# Alocando os valores de atributos e classes na base de dados de treino
for object in train_data:
    train_set[object[-1]].append(object[:-1])
# Alocando os valores de atributos e classes na base de dados de teste
for object in test_data:
    test_set[object[-1]].append(object[:-1])

# Variáveis de contagem para cálculo da precisão
correct = 0
total = 0
# Cálculo da precisão
for group in test_set:
    for data in test_set[group]:
        vote = k_nearest_neighbors(train_set, data, k=5)
        if vote == group:
            correct += 1
        total += 1
print('Accuracy:', correct/total*100, '%')

```

Figura 49 - Código em Python da implementação do caso simples do kNN

### 7.3 BIBLIOGRAFIA

WU, Xindong; KUMAR, Vipin (Ed.). The top ten algorithms in data mining. CRC press, 2009.

## 8 NAÏVE BAYES

Também chamado de *idiot's Bayes*, *simple Bayes* e *Independence Bayes*, o algoritmo Naïve Bayes (Bayes Ingênuo) é um dos métodos mais importantes dentre os algoritmos de classificação supervisionada. É um algoritmo conhecido por ser fácil de construir, não necessitar de esquemas complicados de estimativa de parâmetros iterativos, ser aplicável para bases de dados de grande extensão, ser fácil de se interpretar e, ainda mais importante, apresentar excelente performance. Ele pode não ser o melhor classificador para todas as aplicações, mas pode ser normalmente considerado um modelo confiável, robusto e com boa precisão (WU; KUMAR, 2009).

Tal como qualquer algoritmo de classificação binária, com os valores de classe definidos como  $i = 0, 1$ , o objetivo do modelo de Naïve Bayes é utilizar um conjunto de objetos que já possuem classificação (conhecidos como dados de treinamento) para gerar uma regra classificatória que seja capaz de classificar um novo objeto sem classe. Por uma questão de conveniência, neste trabalho o modelo Naïve Bayes será tratado apenas como um modelo para realizar classificações binárias, entretanto deve-se destacar a capacidade do algoritmo de ser generalizado para casos com mais de duas classes.

### 8.1 DESCRIÇÃO DO ALGORITMO NAÏVE BAYES

Começando com um paradigma de amostragem onde  $P(i|x)$  é a probabilidade de um objeto com vetor de atributos  $x = (x_1, \dots, x_p)$  pertencer à classe  $i$  ( $i = 0, 1$ ),  $f(x|i)$  é a distribuição condicional de  $x$  para objetos de classe  $i$ ,  $P(i)$  é a probabilidade de um objeto pertencer a uma classe  $i$  no caso em que nenhuma outra informação é conhecida (probabilidade 'prior' da classe  $i$ ), e  $f(x)$  é a distribuição de mistura geral das duas classes, tal que:

$$f(x) = f(x|0)P(0) + f(x|1)P(1)$$

A ideia base do algoritmo Naïve Bayes é considerar que uma estimativa de  $P(i|x)$ , por si própria, seria um valor adequado para ser utilizado como regra de classificação. Uma aplicação simples do teorema de Bayes diz que  $P(i|x) = f(x|i)P(i)/f(x)$ , e que para obter uma estimativa de  $P(i|x)$  a partir dela, precisamos estimar cada valor de  $P(i)$  e cada um dos  $f(x|i)$ . Se o conjunto de treinamento é uma amostra aleatória simples extraída da distribuição da população total  $f(x)$ , o  $P(i)$  pode ser estimado diretamente a partir da proporção de objetos de classe no conjunto de treinamento.

O núcleo do método Naïve Bayes está no método de estimar o  $f(x|i)$ . O método de Naïve Bayes assume que os componentes de  $x$  são independentes dentro de cada classe, de modo que  $f(x|i) = \prod_{j=1}^p f(x_j|i)$ , sendo daí a origem do nome alternativo de "*independence Bayes*". Cada uma das distribuições marginais univariadas,  $f(x_j|i)$ ,  $j = 1, \dots, p$ ;  $i = 0, 1$ , é então estimada separadamente. Por este meio, o problema multivariado  $p$  dimensional é reduzido a um problema de estimativa univariada. A estimativa univariada é simples, e requer tamanhos menores de conjuntos de

treinamento para obter estimativas precisas comparada à estimativa de distribuições multivariadas (WU; KUMAR, 2009).

Se as distribuições marginais  $f(x_j|i)$  são discretas e  $x_j$  possui uma quantidade limitada de valores que pode receber, pode-se estimar cada um dos  $f(x_j|i)$  por meio de estimadores do tipo histograma multinominal simples. Por ser tão simples, essa é uma abordagem muito comum ao classificador Naïve Bayes, e muitas implementações adotam essa abordagem. Esta abordagem tem a capacidade de fornecer uma estimativa não-paramétrica muito geral da distribuição univariada, evitando assim quaisquer suposições de distribuição. Em particular, é uma técnica de transformação não-linear, de modo que, por exemplo, a relação entre estimativas de  $f(x_j|i)$  não precisa ser monotônica em  $x_j$ . A um custo computacional maior, pode-se ajustar também modelos mais elaborados às margens univariadas. Por exemplo, pode-se supor formas paramétricas particulares para as distribuições (por exemplo, normal, lognormal, etc) e estimar seus parâmetros por estimadores padrões, ou pode-se adotar estimadores não-paramétricos mais sofisticados, como a estimação da densidade de kernel (WU; KUMAR, 2009).

Sendo assim, seguindo este processo de cálculo de  $f(x|i) = \prod_{j=1}^p f(x_j|i)$ , para se definir qual classe ( $i = 0$  ou  $1$ ) deve ser escolhida como a classificação de um novo objeto teste  $x$ , o algoritmo Naïve Bayes realiza o cálculo das probabilidades  $P(0|x)$  e  $P(1|x)$ , e, de acordo com estes valores, a classificação é realizada. Se  $P(0|x) > P(1|x)$ , o objeto  $x$  será classificado como 0. Já, caso  $P(1|x) > P(0|x)$ , o objeto  $x$  será classificado como 1.

Uma questão que chama muita a atenção do algoritmo Naïve Bayes é o fato de se assumir independência dos atributos dentro de uma classe e esta condição não ocorrer na maioria dos casos reais. A priori, seria esperado que, devido a esta suposição desta hipótese improvável, o método tivesse uma performance fraca. Entretanto, de fato, o que normalmente ocorre é que o modelo consegue realizar, na maioria dos casos reais em que é aplicado, classificações de forma bem precisa. A razão pela qual essa suposição não é tão prejudicial quanto parece é o fato de que, fatores relacionados à natureza do modelo e ao estado dos dados que são utilizados tornarem esta suposição pouco relevante nos resultados do algoritmo.

## 8.2 APLICAÇÃO DO ALGORITMO NAÏVE BAYES

Para a implementação do algoritmo Naïve Bayes optou-se por utilizar uma base de dados obtida em um banco de dados público (<http://archive.ics.uci.edu/ml/datasets/Pima+Indians+Diabetes>). A base de dados apresenta 768 amostras de análises clínicas relacionadas ao diagnóstico de diabetes para mulheres de uma tribo de nativos americanos no Arizona, com 8 atributos (exames clínicos) e uma classificação que diz se o paciente é diabético ou não. O algoritmo foi implementado para esta base de dados de forma a obter um modelo capaz de identificar se uma amostra clínica caracteriza o diagnóstico de uma mulher com diabetes ou de uma mulher sem diabetes.

Com intuito de verificar a eficiência do algoritmo na classificação de novas amostras, a base de dados foi dividida em dois conjuntos, sendo um conjunto de dados para treinamento (67% dos dados originais) e outro conjunto de dados de teste (33% dos dados originais). Ao testar o algoritmo para o conjunto de dados de teste verificou-se uma taxa de acerto em torno de 75%

### 8.2.1 CÓDIGO

```
import csv
import random
import math
# dataset: https://raw.githubusercontent.com/jbrownlee/Datasets/master/pima-indians-diabetes.data.csv
# Informação sobre o dataset: https://www.andreagrandi.it/2018/04/14/machine-learning-pima-indians-diabetes/

## Preparação dos dados
# Função para importação dos dados
def loadCSV(filename):
    lines = csv.reader(open(filename, 'r'))
    dataset = list(lines)
    for i in range(len(dataset)):
        dataset[i] = [float(x) for x in dataset[i]]
    return dataset

# Função para os split dos dados em dados de treinamento e dados de teste
def splitDataset(dataset, splitRatio):
    trainSize = int(len(dataset)*splitRatio)
    trainSet = [] # Lista com dados de treinamento
    testSet = list(dataset) # Lista com dados de teste
    while len(trainSet) < trainSize:
        index = random.randrange(len(testSet)) # Gera um valor de índice randomicamente
        trainSet.append(testSet.pop(index)) # Remove um dado da lista de teste e o adiciona para a lista de treinamento
    return [trainSet, testSet]

## Sumarizar as informações dos dados para realizar predições utilizando o algoritmo Naive Bayes
''' A sumarização das informações dos dados consiste no cálculo, por classe, da média e do desvio padrão para cada atributo. Portanto, como a base de dados analisada possui 2 valores de classe e 7 atributos, será necessário o cálculo da média e do desvio padrão dos 7 atributos para cada uma das 2 classes, ou seja, serão necessários 14 sumários dos atributos (2 para cada atributo).'''

# Função para determinar os valores de classe presente na base de dados e separar os dados de acordo com as classes
```

```

def separateByClass(dataset):
    classes = {}
    for data in dataset:
        if data[-1] not in classes:
            classes[data[-1]] = []
        classes[data[-1]].append(data)
    return classes

# Função para cálculo da média dos valores dos atributos
def mean(attribute_values):
    return sum(attribute_values) / float(len(attribute_values))

# Função para cálculo do desvio padrão
def stdev(attribute_values):
    avg = mean(attribute_values)
    variance = sum([pow(x - avg, 2) for x in attribute_values]) / float(len(
attribute_values)-1)
    return math.sqrt(variance)

# Função de sumarização dos valores dos atributos
def summarize(dataset):
    summaries = [(mean(attribute_values), stdev(attribute_values)) for attri
bute_values in zip(*dataset)]
    # zip(*dataset) agrupa os valores das colunas em tuplas
    del summaries[-1] # Não é necessário sumarizar os valores de classe
    return summaries # Retorna um lista de tuplas com valores de média e des
vio padrão para cada atributo

# Função para sumarização por classe
def summarizeByClass(dataset):
    classes = separateByClass(dataset)
    summariesByClass = {}
    for classValue, instances in classes.items():
        summariesByClass[classValue] = summarize(instances)
    return summariesByClass # Retorna os valores de sumarização para cada cl
asse

# Função para sumarização dos dados em geral (média e desvio padrão dos dado
s originais sem levar em conta a classe)
def generalSummarize(dataset):
    generalSummaries = summarize(dataset)
    return generalSummaries

## Realização de predições
''' Realizar predições consiste no cálculo da probabilidade de um certo dado
pertencer a cada classe. A classe que
    apresentar maior probabilidade será escolhida para a predição. Podemos u
sar a função Gaussiana para estimar a

```

```

    probabilidade de um dado valor de atributo, dado que são conhecidos sua
    média e desvio padrão para o atributo.
    for the attribute estimated from the training data. Dado que os sumários
    dos atributos foram gerados para cada
    atributo e classe, o resultado é uma probabilidade condicional de um dad
    o atributo dado um valor de classe.'''

# Função para cálculo de probabilidade
def calculateProbability(x, mean, stdev):
    exponent = math.exp(-(math.pow(x-mean, 2) / (2*math.pow(stdev, 2))))
    return (1 / (math.sqrt(2*math.pi)*stdev))*exponent

# Função para cálculo de probabilidade de uma classe em relação à todos os d
ados (Prior Probability)
def classesProbabilities(dataset):
    classesProb = {}
    for data in dataset:
        if data[-1] not in classesProb:
            classesProb[data[-1]] = 1 / float(len(dataset))
        else:
            classesProb[data[-1]] += 1 / float(len(dataset))
    return classesProb

# Função para o cálculo da probabilidade de um dado pertencer a uma classe.
# Basta multiplicar as probabilidades de cada atributo pertencer à classe.
def calculateClassProbabilities(summariesByClass, generalSummaries, inputVec
tor, classesProb):
    probabilities = {} #  $P(\text{Class}|\text{X}) = P(\text{X}|\text{Class}) * P(\text{Class}) / P(\text{X})$ 
    # Cálculo de  $P(\text{X}|\text{Class}) * P(\text{Class})$ 
    for classValue, classSummaries in summariesByClass.items():
        probabilities[classValue] = 1
        for i in range(len(classSummaries)):
            mean, stdev = classSummaries[i]
            x = inputVector[i]
            probabilities[classValue] *= calculateProbability(x, mean, stdev
)
        probabilities[classValue] *= classesProb[classValue] #  $P(\text{Class})$ 
    # Cálculo de  $P(\text{X})$ 
    normalizingConstantProb = 1
    for j in range(len(generalSummaries)):
        g_mean, g_stdev = generalSummaries[j]
        x = inputVector[j]
        normalizingConstantProb *= calculateProbability(x, g_mean, g_stdev)
    # Cálculo de  $P(\text{X}|\text{Class}) * P(\text{Class}) / P(\text{X})$ 
    for classValue, value in probabilities.items():
        probabilities[classValue] = value / normalizingConstantProb
    return probabilities

# Função para realizar a predição. Escolhe a classe com maior probabilidade.

```



```

def predict(summariesByClass, generalSummaries, inputVector, classesProb):
    probabilities = calculateClassProbabilities(summariesByClass, generalSummaries, inputVector, classesProb)
    bestLabel, bestProb = None, -1
    for classValue, probability in probabilities.items():
        if bestLabel is None or probability > bestProb:
            bestProb = probability
            bestLabel = classValue
    return bestLabel

## Predições para os dados de teste
# Função que retorna uma lista de predições para os dados de teste
def getPredictions(summariesByClass, generalSummaries, testSet, classesProb):
    :
    predictions = []
    for i in range(len(testSet)):
        result = predict(summariesByClass, generalSummaries, testSet[i], classesProb)
        predictions.append(result)
    return predictions

## Cálculo da precisão do modelo. Compara as predições com as classes dos dados de teste
# Função para cálculo de precisão
def getAccuracy(testSet, predictions):
    correct = 0
    for x in range(len(testSet)):
        if testSet[x][-1] == predictions[x]:
            correct += 1
    return (correct / float(len(testSet))) * 100.0

## Main()
def main():
    splitRatio = 0.67
    dataset = loadCSV('pima_indians_diabetes_dataset.csv')
    trainingSet, testSet = splitDataset(dataset, splitRatio)
    print(f'Split {len(dataset)} rows into train={len(trainingSet)} and test={len(testSet)} rows')
    # Preparação do modelo
    summariesByClass = summarizeByClass(trainingSet)
    generalSummaries = generalSummarize(trainingSet)
    classesProb = classesProbabilities(trainingSet)
    # Teste do modelo
    predictions = getPredictions(summariesByClass, generalSummaries, testSet, classesProb)
    accuracy = getAccuracy(testSet, predictions)
    print(f'Accuracy: {accuracy}%')
main()

```

Figura 50 – Código em Python para implementação do Naïve Bayes

### **8.3 BIBLIOGRAFIA**

ZHANG, H. The optimality of Naive Bayes. Proceedings of the Seventeenth International Florida Artificial Intelligence Research Society Conference, FLAIRS 2004, v. 2, p. 562–567, 2004

WU, Xindong; KUMAR, Vipin (Ed.). The top ten algorithms in data mining. CRC press, 2009.

## 9 CART: CLASSIFICATION AND REGRESSION TREES

A monografia de 1984 intitulada, “CART: Classification and Regression Trees”, de coautoria de Leo Breiman, Jerome Friedman, Richard Olshen e Charles Stone (BFOS), representa um marco importante na evolução da inteligência artificial, aprendizado de máquina, estatística não paramétrica e mineração de dados. O trabalho é de grande relevância para o estudo de árvores de decisão devido às suas inovações técnicas, aos seus exemplos sofisticados de análise de dados estruturados em árvores e à sua abordagem da teoria de grandes amostras para árvores de decisão (WU; KUMAR, 2009).

### 9.1 DESCRIÇÃO DO ALGORITMO CART

A árvore de decisão CART é um procedimento de particionamento recursivo binário capaz de processar atributos contínuos e nominais como foco de análises e previsões. Os dados são manipulados em sua forma bruta; nenhum pré-tratamento é necessário ou recomendado. O algoritmo segue um procedimento no qual, a partir do nó raiz, os dados são divididos em dois nós filhos e cada um dos filhos é dividido em nós netos. As árvores são cultivadas em um tamanho máximo sem o uso de uma regra de parada. Essencialmente, o processo de crescimento da árvore é interrompido quando não há mais divisões possíveis devido à falta de dados. A árvore de tamanho máximo é então podada em direção à raiz (essencialmente divisão por divisão – *split by split*) por meio de um novo método de podagem de complexidade de custo. A divisão a ser podada é aquela que menos contribui para o desempenho geral da árvore em relação aos dados de treinamento (e mais de uma divisão pode ser removida por podagem).

O mecanismo CART é destinado a produzir não uma árvore, mas uma sequência de árvores podadas aninhadas, cada uma das quais é candidata a ser a árvore considerada como a ótima. A árvore do “tamanho certo” é identificada pela avaliação do desempenho preditivo de cada árvore na sequência de poda em dados de teste independentes (WU; KUMAR, 2009).

Uma declaração completa do algoritmo CART, incluindo todos os detalhes técnicos relevantes é longa e complexa. Um algoritmo simplificado para a construção de árvores de decisão CART é esboçado na Figura 51.

```
BEGIN:  Assign all training data to the root node
        Define the root node as a terminal node

SPLIT:
New_splits=0
FOR every terminal node in the tree:
    If the terminal node sample size is too small or all instances in the
    node belong to the same target class goto GETNEXT
    Find the attribute that best separates the node into two child nodes
    using an allowable splitting rule
    New_splits+1
GETNEXT:
NEXT
```

Figura 51 - Algoritmo simplificado para a construção de árvores de decisão CART (WU; KUMAR, 2009).

Com a árvore gerada a partir do algoritmo da Figura 51, o algoritmo CART tem como passo seguinte gerar a sequência aninhada de sub-árvores podadas. Um esboço de algoritmo simplificado para poda é ilustrado na Figura 52. Este modelo de poda é diferente do algoritmo que o CART utiliza e é incluído aqui para simplificar e facilitar a entender a ideia base da etapa de podagem. O procedimento começa pela maior árvore cultivada ( $T_{max}$ ) e removendo todas suas divisões, gerando dois nós terminais que não melhoram a precisão da árvore em relação aos dados de treinamento. Este é o ponto de partida para a poda do CART. A poda prossegue com uma ação natural de remover iterativamente os elos mais fracos da árvore, aquelas divisões que contribuem menos para o desempenho da árvore nos dados de teste. No algoritmo apresentado na Figura 52, a ação de remoção é restrita aos pais de dois nós terminais.

```

DEFINE:  r(t)= training data misclassification rate in node t
         p(t)= fraction of the training data in node t
         R(t)= r(t)*p(t)
         t_left=left child of node t
         t_right=right child of node t
         |T| = number of terminal nodes in tree T

BEGIN:   Tmax=largest tree grown
         Current_Tree=Tmax
         For all parents t of two terminal nodes
           Remove all splits for which  $R(t) = R(t\_left) + R(t\_right)$ 
         Current_tree=Tmax after pruning

PRUNE:   If |Current_tree|=1 then goto DONE
         For all parents t of two terminal nodes
           Remove node(s) t for which  $R(t) - R(t\_left) - R(t\_right)$ 
             is minimum
         Current_tree=Current_Tree after pruning

```

Figura 52 - Algoritmo simplificado para a podagem de árvores de decisão CART (WU; KUMAR, 2009).

O algoritmo de poda do CART difere do descrito acima ao empregar uma penalidade no mecanismo de nós que pode remover uma sub-árvore inteira em uma única podagem.

### 9.1.1 REGRA DE DIVISÃO

As regras de divisão CART são sempre apresentadas na seguinte forma:

*Uma instância vai para a esquerda se CONDIÇÃO, e vai para a direita, caso contrário.*

onde a CONDIÇÃO é expressa como “atributo  $X_i \leq C$ ” para atributos contínuos. Para atributos categóricos ou nominais, a CONDIÇÃO é expressa como uma associação em uma lista de valores. Por exemplo, uma divisão em uma variável como CIDADE pode ser expressa como:

*Uma instância vai para a esquerda se CIDADE estiver em {Chicago, Detroit, Nashville} e vai para a direita, caso contrário.*

O divisor e o ponto de divisão são ambos encontrados automaticamente pelo CART com a divisão ideal selecionada por meio de uma regra de divisão. Os autores do CART argumentam que os splits binários devem ser preferidos a splits múltiplos porque (1) fragmentam os dados mais lentamente que splits múltiplos e (2) splits repetidos no mesmo atributo são permitidos e, se selecionados, irão eventualmente gerar tantas partições para um atributo conforme o necessário.

Os autores do CART discutem exemplos usando quatro regras de separação para árvores de classificação (*Gini*, *twoing*, *ordered twoing*, *gini simétrico*), mas a monografia original concentra a maior parte de sua discussão sobre a utilização do coeficiente *Gini*, que é semelhante ao critério de entropia (ganho de informação) utilizado pelo algoritmo C4.5. Para um alvo binário (0/1), a “medida de impureza *Gini*” de um nó  $t$  é:

$$G(t) = 1 - p(t)^2 - (1 - p(t))^2$$

Onde  $p(t)$  é a frequência relativa (possivelmente ponderada) da classe 1 no nó. A melhoria (ganho) gerada por uma divisão do nó pai  $P$  em nó filho esquerdo  $L$  e em um nó direito  $R$  é calculada por:

$$I(P) = G(P) - qG(L) - (1 - q)G(R)$$

Sendo  $q$  a fração (possivelmente ponderada) das instâncias que vão para o nó filho esquerdo. Os autores da CART preferem o coeficiente *Gini* sobre a entropia, pois ele pode ser computado mais rapidamente, pode ser facilmente estendido para incluir custos simetrizados, e é menos provável de gerar divisões “final split” (split com um nó filho muito pequeno (e relativamente puro) e outro nó filho muito maior).

### 9.1.2 PODAGEM

O mecanismo de podagem utilizado pelo CART baseia-se estritamente nos dados de treinamento e começa com uma medida de complexidade de custo definida como:

$$Ra(T) = R(T) + a|T|$$

onde  $R(T)$  é o custo da amostra de treinamento da árvore,  $|T|$  é o número de nós terminais na árvore e  $a$  é uma penalidade imposta a cada nó. Se  $a = 0$ , a árvore de complexidade de custo mínima é claramente a maior possível. Se  $a$  poder ser aumentado progressivamente, a árvore de complexidade de custo mínimo ficará menor, pois os splits na parte inferior da árvore que reduzem  $R(T)$  serão cortadas. O parâmetro  $a$  é progressivamente aumentado em pequenas etapas, de 0 a um valor suficiente para eliminar todas as divisões. A árvore ótima é definida como aquela árvore na sequência de poda que atinge o custo mínimo dos dados de teste.

## 9.2 APLICAÇÃO DO ALGORITMO CART

Para a implementação do algoritmo CART optou-se por utilizar uma base de dados obtida em um banco de dados público (<http://archive.ics.uci.edu/ml/datasets/banknote+authentication>). A base de dados apresenta 1372 amostras contendo

informações de fotos de cédulas de banco em forma de 5 atributos que permitem dizer se uma cédula de banco é genuína ou forjada. O algoritmo CART foi implementado para esta base de dados de forma a obter um modelo capaz de identificar se uma cédula é verdadeira ou falsa.

Com intuito de verificar a eficiência do algoritmo na classificação de novas amostras, a base de dados foi dividida em dois conjuntos, sendo um conjunto de dados para treinamento e outro conjunto de dados de teste. Ao realizar uma série de testes do algoritmo aplicado ao conjunto de dados e teste verificou-se em todos os testes uma taxa de acerto médio de 97,3%. Ou seja, em média 97,3% das cédulas foi classificada de forma correta, o qual é um valor bem satisfatório para o caso em questão.

Deve-se ressaltar que, para esta implementação, optou-se por utilizar critérios de interrupção de crescimento da árvore ao invés da podagem pelo fato de ser uma abordagem menos complexa comparada a um algoritmo de poda. Entretanto, é esperado que a realização da poda da árvore de decisão gerada consiga aumentar ainda mais a eficiência do algoritmo.

### 9.2.1 CÓDIGO

```
# Dataset: http://archive.ics.uci.edu/ml/datasets/banknote+authentication

# Pacotes a serem utilizados no código
from random import seed
from random import randrange
from csv import reader

# Função para importação dos dados
def load_csv(filename):
    file = open(filename)
    lines = reader(file)
    dataset = list(lines)
    return dataset

# Conversão de coluna string para float
def str_column_to_float(dataset, column):
    for row in dataset:
        row[column] = float(row[column].strip())

# Divisão dos dados em k partições para cross validation
def cross_validation_split(dataset, n_folds):
    dataset_split = list()
    dataset_copy = list(dataset)
    fold_size = int(len(dataset) / n_folds)
    for i in range(n_folds):
        fold = list()
        while len(fold) < fold_size:
            index = randrange(len(dataset_copy))
            fold.append(dataset_copy.pop(index))
```

```

        dataset_split.append(fold)
    return dataset_split

# Função para cálculo de porcentagem de precisão
def accuracy_metric(actual, predicted):
    correct = 0
    for i in range(len(actual)):
        if actual[i] == predicted[i]:
            correct += 1
    return correct / float(len(actual)) * 100

# Função para avaliar o algoritmo utilizando cross validation
def evaluate_algorithm(dataset, algorithm, n_folds, *args): # *args é usado
    # caso possa adicionar mais parâmetros na função
    folds = cross_validation_split(dataset, n_folds)
    scores = list()
    for fold in folds:
        train_set = list(folds)
        train_set.remove(fold)
        train_set = sum(train_set, [])
        test_set = list()
        for row in fold:
            row_copy = list(row)
            test_set.append(row_copy)
            row_copy[-1] = None
        predicted = algorithm(train_set, test_set, *args)
        actual = [row[-1] for row in fold]
        accuracy = accuracy_metric(actual, predicted)
        scores.append(accuracy)
    return scores

## Cálculo do coeficiente Gini
# Função para cálculo de coeficiente Gini para um conjunto de sub-
# nós (grupos) e valores de classe conhecidos
def gini_index(groups, classes):
    # Contagem do total de amostras no ponto de divisão (split point)
    n_instances = float(sum([len(group) for group in groups]))
    # Soma dos coeficientes Gini ponderados para cada grupo
    gini = 0.0
    for group in groups:
        size = float(len(group))
        # Condicional de segurança contra divisão por zero
        if size == 0:
            continue
        score = 0
        # Cálculo da frequência de cada valor de classe (p) e do score p^2
        for class_val in classes:

```

```

        p = [row[-
1] for row in group].count(class_val) / size # .count() retorna o número de
ocorrências de class_val
        score += p*p # soma dos quadrados das frequências de cada classe
no grupo
        # Ponderar o score do grupo pelo seu tamanho relativo
        gini += (1.0 - score)*(size / n_instances) # Quantifica o ganho gera
do pela divisão
    return gini

## Divisão (particionamento) dos dados
''' A divisão dos dados para CART significa os dados em duas listas de linha
s dado o índice do atributo e
    o valor condicional de divisão para este atributo. '''
# Função para o split dos dados
def test_split(index, value, dataset):
    left, right = list(), list() # Lista para armazenamento dos dados dos su
b-nós
    for row in dataset:
        if row[index] < value: # Critério de divisão
            left.append(row)
        else:
            right.append(row)
    return left, right # Retorna as listas de dados para cada sub-nó gerado

# Avaliar de todas as divisões possíveis
''' Com a função para o cálculo do coeficiente the Gini e a função de split
test agora temos o necessário para avaliar
    as divisões. Dado a base de dados, devemos checar todo valor em cada atr
ibuto como um candidato de divisão, avaliar
    o custo da divisão e determinar a melhor divisão possível que se pode re
alizar. Uma vez que a melhor divisão é
    determinada, podemos utilizá-la na árvore de decisão. '''
# Função para determinação da melhor divisão
def get_split(dataset):
    class_values = list(set(row[-
1] for row in dataset)) # set() não permite valores repetidos
    b_index, b_value, b_score, b_groups = 9999, 9999, 9999, None
    for index in range(len(dataset[0])-1): # Não contar a coluna de classes
        for row in dataset:
            groups = test_split(index, row[index], dataset)
            gini = gini_index(groups, class_values)
            #print('X%d < %.3f Gini=%.3f' % ((index+1), row[index], gini))
            if gini < b_score: # Quanto menor o gini melhor
                b_index, b_value, b_score, b_groups = index, row[index], gin
i, groups
    return {'index': b_index, 'value': b_value, 'groups': b_groups}

## Contrução da árvore de decisão

```



```

''' Para gerar o nó raiz da árvore basta aplicarmos get_split() para o dataset original. Já para gerar os sub-nós subsequentes é necessário aplicar get_split() de forma recursiva para cada sub-nó gerado. '''
# Nós terminais. É necessário determinar o momento de interromper o crescimento da árvore.
''' Para isso iremos utilizar o Maximum Tree Depth (máximo número de nós permitidos) na árvore e o Minimum Node Records (número mínimo de dados presentes em um nó). Uma outra condição que se deve levar em conta também é o caso em que todos os dados em um nó pertencem à mesma classe e o nó não pode ser mais dividido. Quando interrompemos o crescimento da árvore em um dado nó, esse nó é chamado de nó terminal e ele é utilizado para realizar a predição final. Isso é feito de forma a escolher a classe mais frequente no grupo de dados no nó. '''
# Função para seleção de uma classe em um nó terminal. Retorna a classe mais comum no nó
def to_terminal(group):
    outcomes = [row[-1] for row in group]
    return max(set(outcomes), key=outcomes.count) # Retorna o item mais comum na lista

# Função para a realização do split recursivo dos nós. Cria sub-nós filhos de um nó ou declara o nó como terminal
def split(node, max_depth, min_size, depth):
    left, right = node['groups']
    del(node['groups'])
    # Checa se algum dos nós criados está vazio
    if not left or not right:
        node['left'] = node['right'] = to_terminal(left + right)
        return
    # Checa se max_depth foi atingida
    if depth >= max_depth:
        node['left'], node['right'] = to_terminal(left), to_terminal(right)
        return
    # Split do nó esquerdo (left child)
    if len(left) <= min_size:
        node['left'] = to_terminal(left)
    else:
        node['left'] = get_split(left)
        split(node['left'], max_depth, min_size, depth+1)
    # Split do nó direito (right child)
    if len(right) <= min_size:
        node['right'] = to_terminal(right)
    else:
        node['right'] = get_split(right)
        split(node['right'], max_depth, min_size, depth+1)

```

```

# Função para construção da árvore
def build_tree(train, max_depth, min_size):
    root = get_split(train)
    split(root, max_depth, min_size, 1)
    return root

# Função para print da árvore
def print_tree(node, depth=0):
    if isinstance(node, dict): # Checa se node é um dicionário
        print('%s[X%d < %.3f]' % ((depth*' ', (node['index']+1), node['value
        '])))
        print_tree(node['left'], depth+1)
        print_tree(node['right'], depth+1)
    else:
        print('%s[%s]' % ((depth*' ', node)))

## Predição de classificação
# Função que faz predição de um dado com a árvore de decisão gerada
def predict(node, row):
    if row[node['index']] < node['value']:
        if isinstance(node['left'], dict):
            return predict(node['left'], row)
        else:
            return node['left']
    else:
        if isinstance(node['right'], dict):
            return predict(node['right'], row)
        else:
            return node['right']

# Função contendo o algoritmo CART
def decision_tree(train, test, max_depth, min_size):
    tree = build_tree(train, max_depth, min_size)
    predictions = list()
    for row in test:
        prediction = predict(tree, row)
        predictions.append(prediction)
    return predictions

## __Main()__
# Teste do algoritmo CART para dados de autenticação de banco
seed(1)
# Importação e preparação dos dados
filename = 'data_banknote_authentication_dataset.csv'
dataset = load_csv(filename)
# Conversão de atributos string para float
for i in range(len(dataset[0])):
    str_column_to_float(dataset, i)
# Avaliação da performance do algoritmo

```

```
n_folds = 5
max_depth = 5
min_size = 10
scores = evaluate_algorithm(dataset, decision_tree, n_folds, max_depth, min_size)
print('Scores: %s' % scores)
print('Mean Accuracy: %.3f%%' % (sum(scores)/float(len(scores))))
```

Figura 53 - Código em Python para a implementação do CART

### 9.3 BIBLIOGRAFIA

BREIMAN, L. et al. Classification and regression trees. [s.l.] CRC press, 1984

WU, Xindong; KUMAR, Vipin (Ed.). The top ten algorithms in data mining. CRC press, 2009.

## 10 REDE NEURAL FEEDFOWARD

As redes neurais artificiais são modelos computacionais que foram introduzidos nos anos 1950s como modelos baseados em sistemas nervosos presentes em seres vivos. Estes modelos podem ser descritos como um conjunto de unidades de processamento (neurônios artificiais) interligados por diversas conexões (sinapses artificiais) capazes de adquirir e manter conhecimentos baseados em dados. A implementação destes modelos é realizada por meio de simples operações lineares de vetores e matrizes (pesos sinápticos). Ao longo dos anos vários modelos matemáticos foram propostos para as redes neurais entre os quais destacam-se o modelo básico Perceptron idealizado por (ROSENBLATT, 1958), o modelo ADALINE (WIDROW; HOFF, 1960) e o modelo MADALINE. Apesar de sua capacidade de capturar conhecimento, por muitos anos estes métodos foram caracterizados por suas limitações de aprendizado, uma vez que eram compostos por uma única camada de unidades de processamento e seus métodos de treinamento eram baseados em funções lógicas básicas. Um grande marco no estudo de redes neurais artificiais foi a publicação do livro “Parallel Distributed Processing” (MCCLELLAND et. al., 1986) de Rumelhart, Hinton e William, o qual introduziu o algoritmo que permitiu o treinamento de redes neurais mais complexas, o chamado “backpropagation”. A criação do algoritmo de “backpropagation” reanimou os estudos sobre redes neurais motivando surgimento de novos modelos de redes neurais (redes recorrentes, redes convolucionais, autoencoders, etc), além de outros algoritmos de treinamento (ADAM, SGD, etc). Neste contexto, as redes neurais vêm sendo aplicadas com sucesso para diversos problemas de ciência e engenharia, oferecendo funcionalidades tais como: ajuste de curvas (regressão), controle de processos, classificação e reconhecimento de padrões, agrupamento de dados, predição, otimização de sistemas, memória associativa, entre outras. Uma breve descrição dos componentes para implementação de uma rede neural é realizada a seguir.

### 10.1 UNIDADE DE PROCESSAMENTO

A estrutura das redes neurais artificiais foi idealizada por meio de modelos conhecidos de sistemas nervosos biológicos. As unidades de processamento, referenciadas como neurônios artificiais, são modelos bem simplificados dos neurônios biológicos encontrados que realizam simples funções de unir sinais de entrada, manipulá-los de acordo com uma função operacional matemática, e produzir uma resposta considerando uma função ativação. O modelo de unidade de processamento mais utilizado em redes neurais é apresentado na Figura 54. Neste modelo, múltiplos sinais de entrada oriundos de um ambiente externo são representados por um conjunto  $\{x_1, x_2, x_3, \dots, x_m\}$  e a relevância de cada sinal  $\{x_i\}$  é calculada pela multiplicação de seu parâmetro peso  $\{w_i\}$  correspondente. O agrupamento destes sinais ponderado é realizado em um agregador linear ( $\Sigma$ ). A resposta  $v$  gerada pelo agregador consiste na adição entre a soma ponderada dos sinais e um parâmetro bias ( $b$ ), o qual é o valor limite que  $v$  deve ter para que uma resposta seja gerada pela unidade de processamento.

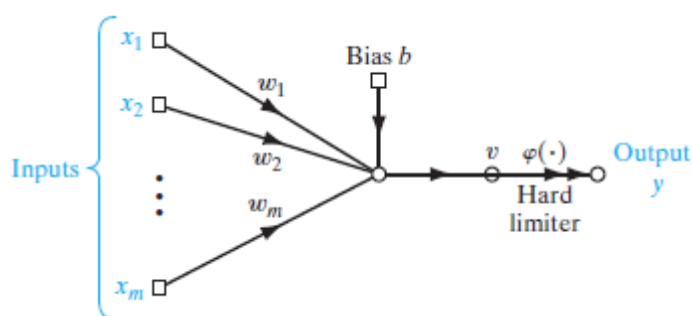


Figura 54 - Unidade de processamento

Após ser gerada pelo agregador linear,  $v$  é fornecida para uma função de ativação ( $\varphi(\cdot)$ ), a qual limita que a resposta final  $y$  da unidade de processamento esteja contida em um intervalo de valores. Uma vez que  $y$  é gerada pode ser utilizada como entrada de outras unidades de processamento. As operações realizadas pela unidade de processamento podem ser sintetizadas por duas expressões:

$$v = \sum_{i=1}^m w_i x_i + b$$

$$y = \varphi(u)$$

Entre as funções de ativação podem ser utilizadas funções tais como:

- Função logística (Sigmóide):

$$\varphi(v) = \frac{1}{1 + e^{-\beta v}}$$

- Função tangente hiperbólico (TanH):

$$\varphi(v) = \frac{1 - e^{-\beta v}}{1 + e^{-\beta v}}$$

- Função Gaussiana:

$$\varphi(v) = e^{-\frac{(v-c)^2}{2\sigma^2}}$$

- Função retificadora linear (ReLU):

$$\varphi(v) = \max(0, v)$$

## 10.2 ARQUITETURAS DE REDES NEURAIS

A arquitetura de uma rede neural artificial define como suas unidades de processamento são arranjadas. Esses arranjos são estruturados pelo direcionamento de conexões entre as unidades de processamento, definindo a topologia das redes neurais. Em geral, uma rede neural artificial pode ser dividida em três partes, a camada de entrada (responsável por receber as informações / dados), as camadas ocultas, as quais são compostas por unidades de processamento responsáveis por extrair padrões nos dados analisados, e a camada de saída, composta por unidades de processamento responsáveis por produzir e apresentar as repostas da rede. As principais arquiteturas de redes neurais podem ser divididas em diversos tipos, sendo a arquitetura feedforward a mais conhecida historicamente. Exemplos de outros tipos de arquiteturas são as redes neurais recorrentes, as redes neurais convolucionais, as redes RBF e os autoencoders.

### 10.2.1 ARQUITETURA FEEDFORWARD

As redes neurais com arquitetura feedforward são caracterizadas pelo fluxo de informação em uma única direção, o qual se inicia na camada de entrada e se encerra na camada de saída. A estrutura mais comum de arquitetura feedforward consiste em redes neurais com uma camada de entrada, uma camada de saída e múltiplas camadas ocultas entre elas. Este tipo de rede neural é aplicado para diversos problemas como por exemplo aproximação de funções, classificação de padrões, e sistema de predição.

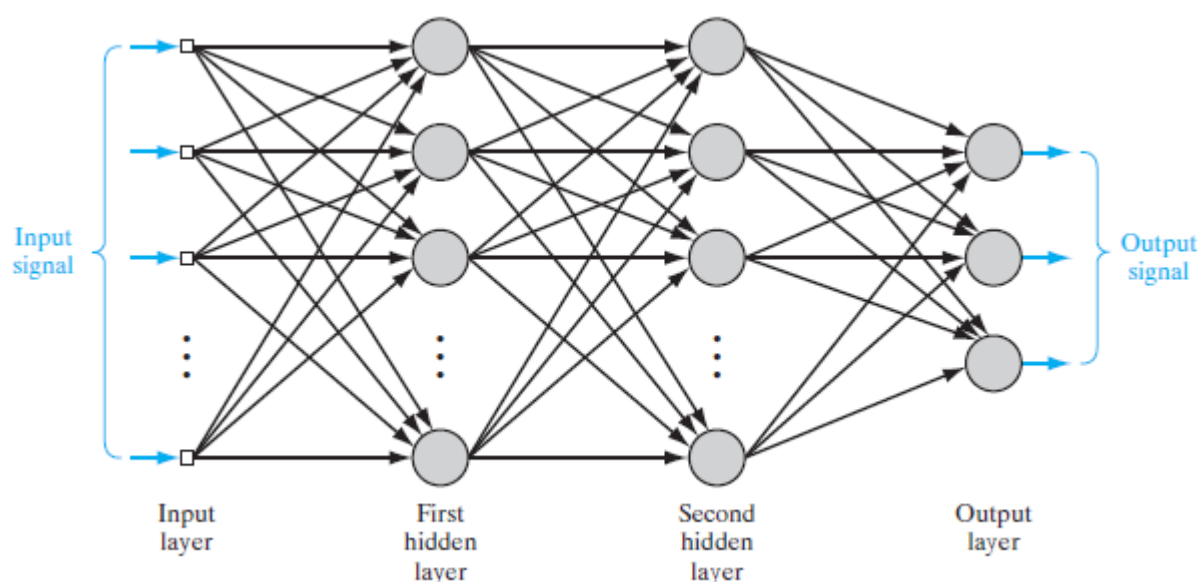


Figura 55 - Rede Neural Feedforward Multicamada

A Figura 55 ilustra uma rede feedforward composta por uma camada de entrada, duas camadas ocultas, e uma camada de saída que apresenta as respostas do problema sendo analisado. Neste tipo de redes neurais, também chamado de Multilayer Perceptron (MLP), os sinais de entrada são processados camada por camada por meio das operações lineares matriciais e transformações não-lineares presentes nas

unidades processamento até a camada de saída. Nota-se que o número de unidades de processamento da primeira camada oculta não é necessariamente igual ao número de sinais de entrada que compõe a camada de entrada. A escolha do número de camadas ocultas e o número respectivo de unidades de processamento normalmente depende da natureza e complexidade do problema sendo analisado, tal como a quantidade e qualidade dos dados disponíveis. Já o número de saídas da rede neural sempre irá coincidir com o número de unidades de saída.

### 10.3 PROCESSO DE TREINAMENTO

A capacidade de aprendizado por meio de dados é uma das características mais relevantes de uma rede neural. Ao aprender a relação entre dados de entrada e seus respectivos valores de saída, a rede neural se torna capaz a realizar generalizações, ou seja, a rede passa a gerar soluções bem próximas às respostas esperadas para qualquer dado de entrada. Para o aprendizado das relações entre dados de entrada e saída, um processo de treinamento é necessário. O processo de treinamento de uma rede neural consiste basicamente na aplicação de passos ordenados de ajuste dos parâmetros pesos e bias de suas unidades de processamento de forma a minimizar o erro entre as respostas fornecidas pela rede e as repostas esperadas. Este conjunto de passos ordenados compõe um algoritmo de treinamento, o qual possibilita que componentes relevantes para o ajuste dos parâmetros da rede neural sejam extraídos dos dados. Tal como citado anteriormente, o algoritmo mais relevante para o processo de treinamento em redes neurais é o algoritmo “backpropagation”.

#### 10.3.1 BACKPROPAGATION

O processo de treinamento de uma rede neural MLP utilizando o algoritmo “backpropagation” é realizado basicamente em duas etapas, a propagação “forward” (para frente) e propagação “backward” (para trás) (Figura 56).

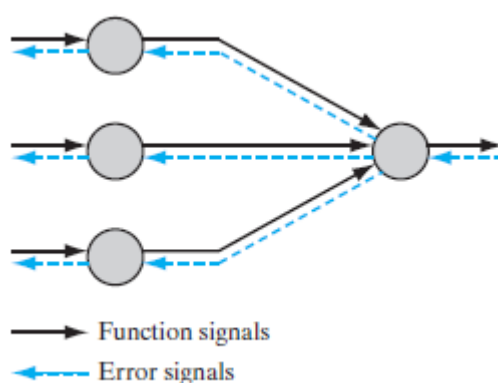


Figura 56 – Propagação forward (preto). Propagação backward(azul)

O primeiro estágio (propagação “forward”) consiste na propagação dos sinais dos dados de entrada de camada em camada até produção de sua resposta correspondente na camada de saída. Portanto, este estágio é unicamente responsável pela obtenção de respostas considerando os parâmetros pesos e bias correntes da rede neural durante a execução do estágio. Em seguida, as repostas

produzidas pela rede neural são comparadas respectivamente as respostas desejadas, o erro obtido é utilizado para o segundo estágio de treinamento, a propagação “backward”. Ao contrário do primeiro estágio, modificações e ajustes nos parâmetros peso e bias das unidades de processamento são realizadas nesta segunda propagação por meio de um processo de otimização. O algoritmo de “backpropagation” normalmente utiliza técnicas de otimização via gradiente descendente, as quais fornecem uma aproximação da trajetória no espaço dos parâmetros ajustados rumo a algum mínimo local.

Considerando um conjunto de dados de treino contendo  $N$  amostras  $\{(x(n), d(n))\}_{n=1}^N$ , onde  $x(n)$  é um dado a ser fornecido para o treinamento de uma rede neural e  $d(n)$  é a resposta esperada para esse dado, o processo de “backpropagation” pode ser descrito pelos seguintes passos:

1. **Inicialização:** Assumindo que nenhuma informação a priori esteja disponível, os valores dos parâmetros peso e bias das unidades de processamento são randomicamente inicializados por meio de distribuições probabilísticas de valores, como por exemplo uma distribuição uniforme ou distribuição normal.
2. **Introdução dos Dados de Treinamento:** Um conjunto de dados para treinamento são introduzidos à rede neural. Cada amostra presente no conjunto de dados é submetida à sequência de propagação “forward” e propagação “backward” descritas nos passos 3 e 4, respectivamente.
3. **Propagação Forward:** Dado a amostra presente no conjunto de dados denotada por  $(x(n), d(n))$ , com vetor de entrada  $x(n)$  aplicado para a camada de entrada e o vetor resposta esperada  $d(n)$  na camada de saída, esta é propagada de camada para camada tal que em cada unidade de processamento é realizada a agregação de sinais e a geração de uma resposta. A agregação de sinais na unidade de processamento  $j$  na camada  $l$  é denotada por:

$$v_j^{(l)}(n) = \sum_i w_{ji}^{(l)}(n) y_i^{(l-1)}(n)$$

Onde  $y_i^{(l-1)}(n)$  é a resposta da unidade  $i$  na camada anterior  $l - 1$  na iteração  $n$ ,  $w_{ji}^{(l)}(n)$  é o parâmetro peso do unidade  $j$  na camada  $l$  que é submetido à resposta gerada pela unidade  $i$  da camada  $l - 1$ . Para  $i = 0$ , tem-se que  $y_0^{(l-1)}(n) = 1$  e  $w_{j0}^{(l)}(n) = b_j^{(l)}(n)$  é o parâmetro bias aplicado para a unidade  $j$  na camada  $l$ . A geração da resposta da unidade de processamento é denotada pela aplicação da função ativação ao valor da agregação de sinais da unidade:

$$y_j^{(l)} = \varphi_j(v_j(n))$$



Se a unidade  $j$  está na primeira camada oculta  $l = 1$  então:

$$y_j^{(0)} = x_j(n)$$

Onde  $x_j(n)$  é o  $j$ ésimo elemento do vetor entrada  $x(n)$ . Se a unidade  $j$  estiver na camada de saída  $l = L$  então:

$$y_j^{(L)} = o_j(n)$$

O erro gerado pela comparação da resposta gerada pela rede e a resposta esperada é denotado por:

$$e_j(n) = d_j(n) - o_j(n)$$

Onde  $d_j(n)$  é o  $j$ ésimo elemento do vetor resposta esperada  $d(n)$ .

- 4. Propagação Backward:** Durante a propagação “backward” os valores de gradientes locais  $\delta s$  da rede neural são calculados. Esse cálculo é realizado da seguinte forma:

$$\delta_j^{(l)}(n) = \begin{cases} e_j^{(L)}(n) \varphi_j'(v_j^{(L)}(n)) & \text{para unidade } j \text{ na camada de saída } L \\ \varphi_j'(v_j^{(l)}(n)) \sum_k \delta_k^{(l+1)}(n) w_{kj}^{(l+1)}(n) & \text{para unidade } j \text{ na camada oculta } l \end{cases}$$

Onde  $\varphi_j'$  denota a diferenciação da função ativação com respeito ao argumento. O ajuste dos parâmetros pesos e bias da rede neural na camada  $l$  é realizado pela regra de atualização:

$$w_{ji}^{(l)}(n+1) = w_{ji}^{(l)}(n) + \alpha [\Delta w_{ji}^{(l)}(n-1)] + \eta \delta_j^{(l)}(n) y_i^{(l-1)}(n)$$

Onde  $\eta$  é a taxa de aprendizado e  $\alpha$  é uma constante de momento. A taxa de aprendizado  $\eta$  controla a escala em que os ajustes dos parâmetros são realizados, tal que taxas elevadas aceleram o treinamento, mas geram instabilidade nas redes neurais, enquanto que taxas menores resultam em treinamento mais lento, porém mais estável. A constante de momento  $\alpha$  é utilizado como um componente que possibilite o uso de taxas de treinamento mais elevadas sem que instabilidade seja gerada na rede neural

- 5. Iteração:** Iterações das propagações “forward” e “backward” são realizadas por meio dos passos 3 e 4 ao apresentar novas amostras de treinamento para rede neural até que um critério de parada seja obedecido.

Em resumo, as sucessivas aplicações dos estágios “forward” e “backwards” permitem que os parâmetros pesos e bias das unidades de processamento sejam

automaticamente ajustados em cada iteração, resultando na gradual redução da soma dos erros produzidos na comparação entre respostas produzidas pela rede e respostas esperadas. A Figura 57 ilustra o fluxo de sinais durante o processo de “backpropagation” de uma rede neural composta por uma camada de entrada com 3 entradas de sinais, duas camadas ocultas com três unidades de processamento cada uma, e uma camada de saída com três saídas. As linhas em preto representam o fluxo de informação durante a propagação “forward” e as linhas em azul representam o fluxo de informações durante a propagação “backward”.

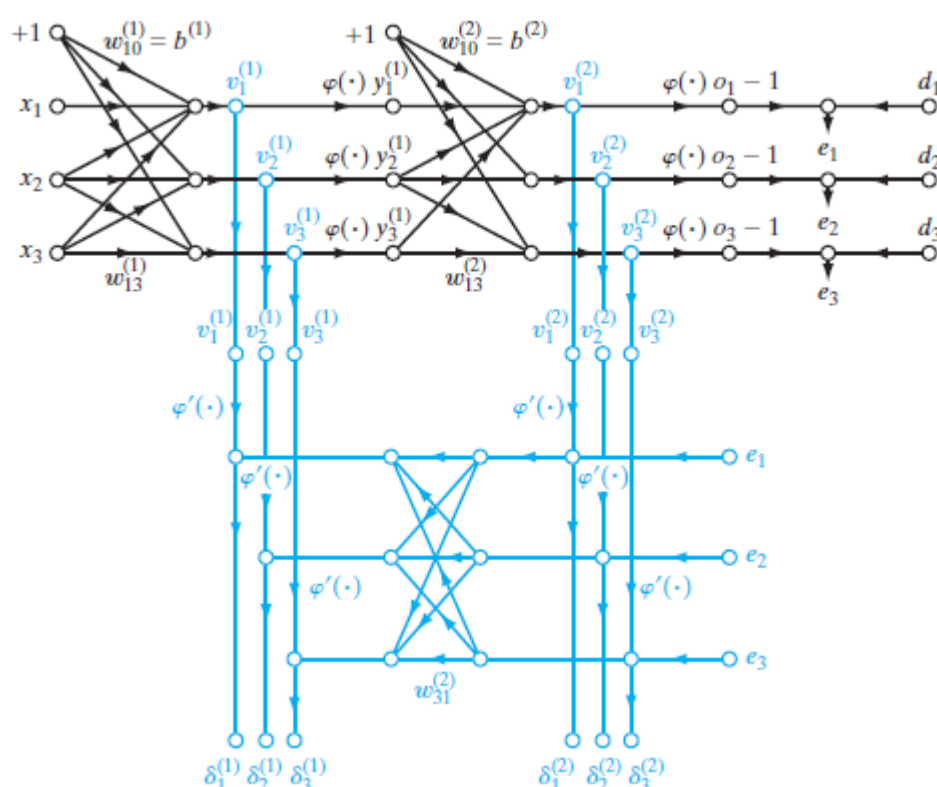


Figura 57 - Fluxo de sinais durante a aplicação do “backpropagation”. Parte superior: estágio “forward”. Parte inferior: estágio “backward”

## 10.4 APLICAÇÃO DA REDE NEURAL FEEDFORWARD

Para exemplificar aplicação de uma rede neural feedforward, um problema de classificação de dígitos manuscritos baseado na análise de imagens do conjunto de dados MNIST (“<http://yann.lecun.com/exdb/mnist/>”) é proposto. O problema consiste em treinar um modelo de forma a torná-lo capaz de identificar quais são os dígitos manuscritos nas imagens analisadas. Sendo assim, para solução deste problema propõe-se a implementação de uma rede neural com 4 camadas, contendo 64 unidades de processamento em cada uma das três camadas ocultas e 10 unidades na camada de saída (uma saída para cada classe possível  $\{0,1,2,3,4,5,6,7,8,9\}$  para classificação das imagens analisadas). As amostras coletadas do conjunto de dados MNIST são imagens com resolução (28x28) pixels (Figura 58). De forma a avaliar a performance de generalização do modelo após treino, estas são divididas em dois

grupos: dados de treinamento (utilizados no processo de aprendizado do modelo) e dados de teste (utilizados para avaliar a performance do modelo).

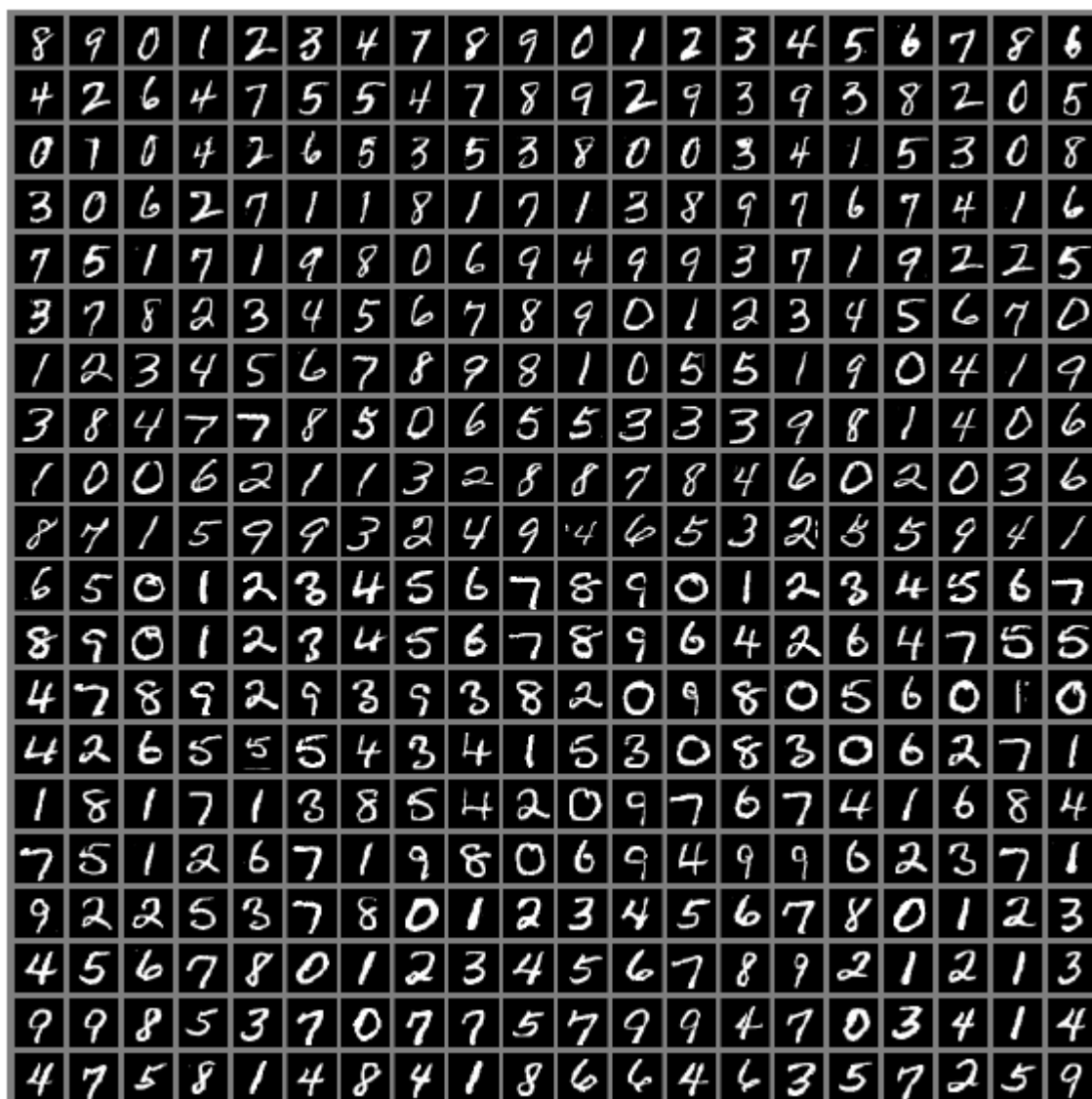


Figura 58 - Amostras do conjunto de dados MNIST

Para inserção dessas amostras na rede neural é realizado achatamento das imagens analisadas em vetores de dimensão (1x784) contendo os valores dos 784 pixels da imagem. Para as unidades de processamento nas camadas ocultas opta-se pelo uso da função de ativação ReLU, e nas unidades da camada de saída opta-se pelo uso da função de ativação LogSoftmax, a qual é específica para geração de respostas em problemas de classificação. O treinamento da rede neural é realizado por meio do algoritmo de “backpropagation” otimizado pelo algoritmo Adam (variação do algoritmo de gradiente descendente) e a função erro utilizada é a NegativeLogLikelihood (Log Verossimilhança Negativa), utilizada para comparar respostas de problemas de classificação. Após o treinamento da rede neural, o processo de teste utilizando amostras nunca vistas pela rede treinada é realizado para avaliar a performance do modelo proposto. A rede neural treinada apresentou uma boa performance durante o

teste apresentando uma acurácia de 97%. Exemplos de classificações realizadas pelo modelo durante o processo de teste são ilustradas na Figura 59.

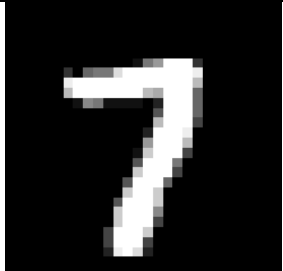
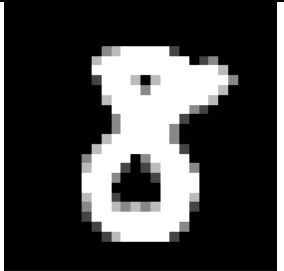
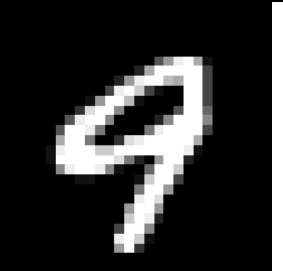
Amostra fornecida para o modelo			
Classificação pelo modelo	7	8	4

Figura 59 – Exemplos de classificações realizadas pela rede neural feedforward

## 10.5 CÓDIGO

```
# importação das bibliotecas a serem utilizadas
# pytorch: biblioteca para implementação de modelos de deep learning / redes neurais
import torch
import torchvision
from torchvision import transforms, datasets
import torch.nn as nn
import torch.nn.functional as F
import torch.optim as optim

# coleta dos dados: serão analisados amostras da base de dados MNIST para reconhecimento de dígitos manuscritos
# o pytorch já oferece funções para a coleta desses dados
# dados originais: http://yann.lecun.com/exdb/mnist/

# coleta de dados para treinamento
train = datasets.MNIST('', train=True, download=True, transform=transforms.Compose([transforms.ToTensor()]))
# coleta de dados para teste
test = datasets.MNIST('', train=False, download=True, transform=transforms.Compose([transforms.ToTensor()]))

# preparação dos dados: dados serão analisados em bateladas de 10 amostras
trainset = torch.utils.data.DataLoader(train, batch_size=10, shuffle=True)
testset = torch.utils.data.DataLoader(test, batch_size=10, shuffle=False)

# implementação do modelo / arquitetura de rede neural a ser utilizado
# optou-se por uma rede neural feedforward de 4 camadas de unidades de processamento
class Net(nn.Module):
    # definição da estrutura das camadas
    def __init__(self):
        super().__init__()
```

```

    # imagens analisadas possuem dimensão 28x28 pixels
    # logo os dados de entrada são vetores de dimensão (1, 28x28) = flat
ten image
    self.fc1 = nn.Linear(28*28, 64) # camada oculta 1: 64 unidades de pr
ocessamento
    self.fc2 = nn.Linear(64, 64) # camada oculta 2: 64 unidades de proce
ssamento
    self.fc3 = nn.Linear(64, 64) # camada oculta 3: 64 unidades de proce
ssamento
    self.fc4 = nn.Linear(64, 10) # camada de saída: 10 unidades de proce
ssamento
    # amotras são classificadas por meio de 10 classes (0,1,2,3,4,5,6,7,
8,9)
    # o mesmo número de unidades de saída

    # implementação da propagação forward
    def forward(self, x):
        # amostra x é propagada de camada em camada
        # para 3 primeiras camadas ocultas a função de ativação ReLU é utili
zada
        x = F.relu(self.fc1(x))
        x = F.relu(self.fc2(x))
        x = F.relu(self.fc3(x))
        # para a camada de saída utiliza-
se a função de ativação LogSoftmax, comum em problemas de classificação
        x = F.log_softmax(self.fc4(x), dim=1)
        return x

    # pytorch: valores dos parâmetros são iniciados randomicamente utilizand
o distribuições uniformes

# iniciação do modelo de rede neural implementado
net = Net()

# escolha do algoritmo de treinamento
# opta-
se pelo uso do algoritmo de otimização Adam para o algoritmo de backprogratio
n
optimizer = optim.Adam(net.parameters(), lr=0.001) # taxa de aprendizado lr
= 0.001

# processo de treinamento
loss_log = []
for epoch in range(3): # 3 passagens sobre todas as amostras de treinamento
são realizadas
    total_loss = torch.tensor([[0]])
    for data in trainset: # iteração da propagação de amostras de treinament
o
        X, y = data # amostras de treinamento

```

```

net.zero_grad() # zera os valores de gradiente da rede neural
# propagação forward
output = net(X.view(-1, 28*28))
# cálculo do erro das repostas geradas na propagação forward
loss = F.nll_loss(output, y) # opta-
se pelo uso da função erro Negative Log Likelihood
                                # para problemas de classificação

# propagação backward
# pytorch: gradientes são calculados automaticamente durante a propa
gação forward
loss.backward()
# ajuste dos valores dos parâmetros pesos e bias
optimizer.step() # algoritmo Adam otimiza os valores dos parâmetros
pela propagação dos gradientes da rede
total_loss += loss
# valores de erro
loss_log.append(total_loss.item())

# processo de teste
correct = 0
total = 0
# desativa o cálculo dos gradientes
with torch.no_grad():
    for data in testset: # iteração da propagação das amostras de teste
        X, y = data # amostras de teste
        output = net(X.view(-1, 28*28)) # respostas geradas pela rede neural
        # checagem de resposta gerada está correta
        for idx, i in enumerate(output):
            if torch.argmax(i) == y[idx]:
                correct += 1
        total += 1
# Acurácia:
print("Accuracy: ", round(correct/total, 3))

```

Figura 60 – Código em Python para implementação da rede neural feedforward

## 10.6 BIBLIOGRAFIA

HAYKIN, Simon. Neural networks: a comprehensive foundation. Prentice Hall PTR, 1994.

DA SILVA, Ivan Nunes et al. Artificial neural networks. Cham: Springer International Publishing, p. 39, 2017.

MCCLELLAND, James L. et al. Parallel distributed processing. Explorations in the Microstructure of Cognition, v. 2, p. 216-271, 1986.

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

GOODFELLOW, Ian; BENGIO, Yoshua; COURVILLE, Aaron. Deep learning. MIT press, 2016.

WIDROW, Bernard; HOFF, Marcian E. Adaptive switching circuits. Stanford Univ Ca Stanford Electronics Labs, 1960.

## 11 REDE NEURAL CONVOLUCIONAL

As redes neurais convolucionais (LECUN et al., 1989) consistem em um tipo especializado de redes neurais para processamento de dados estruturados em grid, tal como dados de séries temporais (amostras coletadas em intervalos de tempo regulares que podem ser estruturados em grids 1D) e dados de imagens (as quais podem ser organizados em grids 2D de pixels). As redes convolucionais tem se destacado pelo seu sucesso em aplicações complexas de reconhecimento de imagens (KRIZHEVSKY; SUTSKEVER; HINTON, 2012). Estas redes neurais são chamadas de convolucionais pelo fato de empregarem a operação matemática de convolução (tipo de operação linear) no lugar da simples multiplicação de matrizes com valores de parâmetros normalmente aplicada em redes neurais como as feedforward (Figura 61).

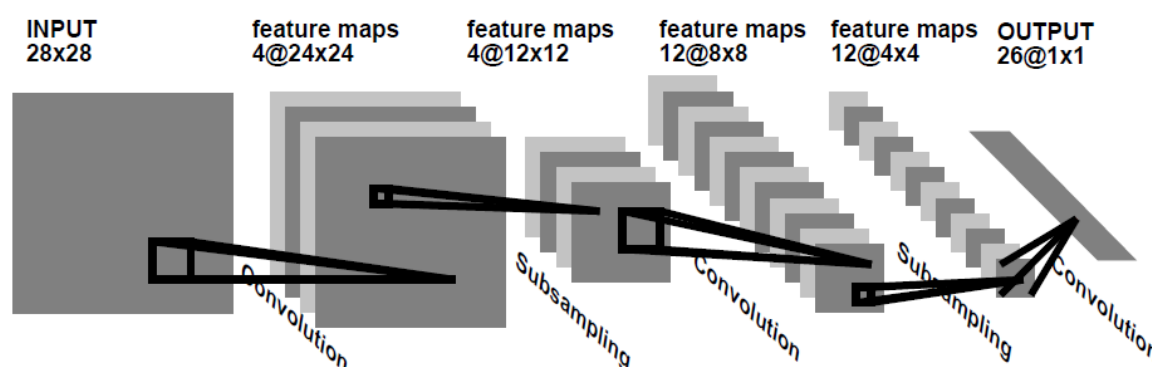


Figura 61 - Rede convolucional comumente utilizada para reconhecimento de imagens

Para descrever o funcionamento das redes neurais convolucionais, duas operações matemáticas devem ser explicadas, a operação de convolução e a operação de pooling (subsampling).

### 11.1 OPERAÇÃO DE CONVOLUÇÃO

De forma geral, a convolução consiste em uma operação que envolve duas funções com argumentos reais. Tome como exemplo um problema de rastreamento da localização de uma aeronave por meio de sensores a laser. O sensor fornecerá uma única resposta  $x(t)$  (a localização da aeronave no tempo  $t$ ). Ambos  $x$  e  $t$  são valores reais tal que diferentes medidas do sensor podem ser obtidas em cada instante de tempo. Supondo que as medidas do sensor sofrem com ruído, para obter uma estimativa da posição da aeronave é desejado que se tire a média de várias medições. Sabendo medidas mais recentes são consideradas mais relevantes, a estimativa da localização então deve ser realizada por meio de uma média ponderada. A ponderação das medições pode ser feita com funções pesos  $w(a)$ , tal que  $a$  representa o tempo decorrente após a realização da medição  $x(a)$ . Ao aplicar tal ponderação para cada instante, uma nova função  $s$  é obtida, a qual fornece uma estimativa sem ruídos da localização da aeronave:



$$s(t) = \int x(a)w(t-a)da$$

Esta operação é um exemplo de convolução. A operação de convolução é tipicamente denotada com um asterisco:

$$s(t) = (x * w)(t)$$

A convolução é genericamente definida por quaisquer funções que definem a integral acima e pode ser utilizada para diversas outras funções além de gerar médias ponderadas. O primeiro argumento da operação de convolução ( $x$ ) é referenciado como o valor de entrada e o segundo argumento ( $w$ ) é chamado de kernel. A resposta da operação ( $s$ ) é comumente chamada de mapa de componentes. O exemplo proposto para operação de convolução não deve ser considerado realístico uma vez que um sensor real não pode realizar medidas em qualquer instante de tempo. Normalmente medições são realizadas em intervalos específicos de tempo, o que impossibilitaria o cálculo da integral de convolução para todos os instantes de tempo. Sendo assim, considerando que  $t$  seja representado por valores discretizados do tempo, e que  $x$  e  $w$  sejam definidos apenas nesses instantes de tempo, é possível definir uma operação de convolução discretizada:

$$s(t) = (x * w)(t) = \sum_{a=-\infty}^{\infty} x(a)w(t-a)$$

Onde para valores de  $a$  para quais não foram realizadas medições, as funções possuem valor nulo.

No caso de aplicações de machine learning, o valor de entrada em uma operação de convolução é normalmente uma matriz multidimensional contendo dados e o kernel é uma matriz multidimensional de parâmetros que são ajustados por um algoritmo de aprendizado. As operações de convolução são normalmente aplicadas simultaneamente em mais de uma dimensão. Por exemplo, no caso de uma imagem bidimensional  $I$  utilizada como entrada, é desejado que também se utilize um kernel bidimensional  $K$ :

$$S(i, j) = (I * K)(i, j) = \sum_m \sum_n I(m, n)K(i-m, j-n)$$

A operação de convolução é comutativa, tal que:

$$S(i, j) = (K * I)(i, j) = \sum_m \sum_n I(i-m, j-n)K(m, n)$$

Esta propriedade de comutação se deve ao fato de ser possível rotacionar o kernel em relação a entrada. Apesar destas serem as expressões que descrevem a operação de convolução no caso de entradas bidimensionais, muitas bibliotecas de machine

learning optam pela implementação da função de correlação-cruzada, a qual é mesma da operação de convolução, porém sem rotacionar o kernel:

$$S(i, j) = (I * K)(i, j) = \sum_m \sum_n I(i + m, j + n) K(m, n)$$

A Figura 62 ilustra um exemplo de convolução (sem rotação do kernel) aplicado para uma entrada bidimensional.

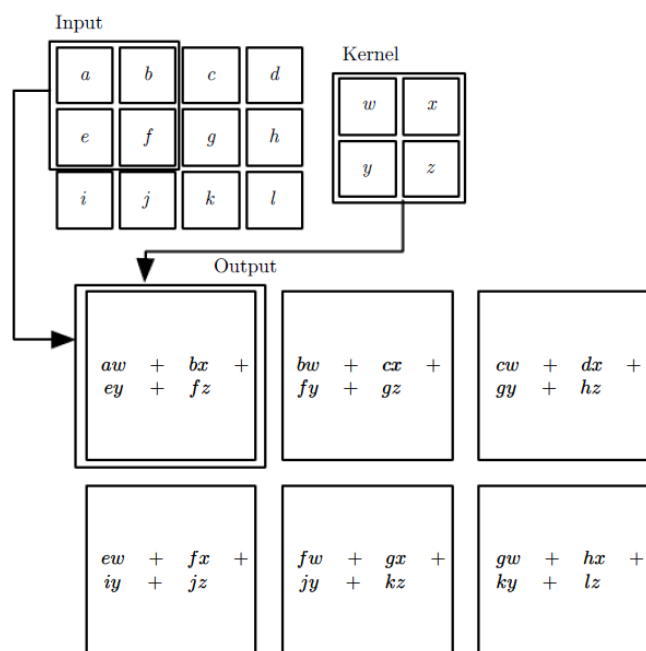


Figura 62 - Exemplo de convolução bidimensional

Tal como pode ser observado na Figura 62, o processo de convolução discreta pode ser vista como uma multiplicação matricial entre partições da entrada e a matriz kernel. Qualquer matriz que realize operações matriciais, mas que não dependa de propriedades relacionadas às estruturas das matrizes devem operar bem com convolução, sem que nenhuma alteração na rede seja necessária.

### 11.1.1 MOTIVAÇÃO PARA O USO DE CONVOLUÇÃO EM REDES NEURAIAS

As camadas de redes neurais feedforward tradicionais utilizam multiplicações de matrizes de parâmetros para descrever relação entre unidades de entrada e unidades de saída camada, o que significa que cada unidade de saída interage com todas as unidades da entrada. Nas redes convolucionais, entretanto, as interações entre as unidades de uma camada são consideradas esparsas, uma vez que as multiplicações matriciais que ocorrem na operação de convolução utilizam kernels com dimensões bem menores que as da matriz de entrada. Por exemplo, ao analisar uma imagem que contenha milhares de pixel, é possível detectar pequenos componentes relevantes na imagem ao utilizar kernels que ocupam apenas dezenas de pixels para o mapeamento da imagem. Isto significa que um menor número de parâmetros precisa ser armazenado e a memória exigida pelo modelo é reduzida. Além disso, as redes

convolucionais possuem a vantagem de utilizar os mesmos valores de parâmetros em mais de uma função do modelo. Em redes neurais tradicionais cada elemento da matriz de pesos é utilizado apenas uma única vez no cálculo da saída de uma unidade de processamento. Já no caso das redes convolucionais, cada elemento da matriz kernel é utilizado em cada posição da matriz de entrada (com exceção dos elementos de borda), ou seja, os parâmetros do kernel são utilizados em mais de uma ocasião. Isto significa que ao invés de treinar um conjunto de parâmetros para cada elemento de entrada, apenas um único conjunto de parâmetros é treinado para todos elementos de entrada, reduzindo também a quantidade de memória exigida pelo modelo. Além de melhorar o uso da memória das redes neurais, o uso convolução possibilitam por exemplo que os kernels atuem como filtros de uma imagem, melhorando assim a detecção de componentes relevantes para análise do conteúdo da imagem, como formas, contornos, arestas, etc (Figura 63).

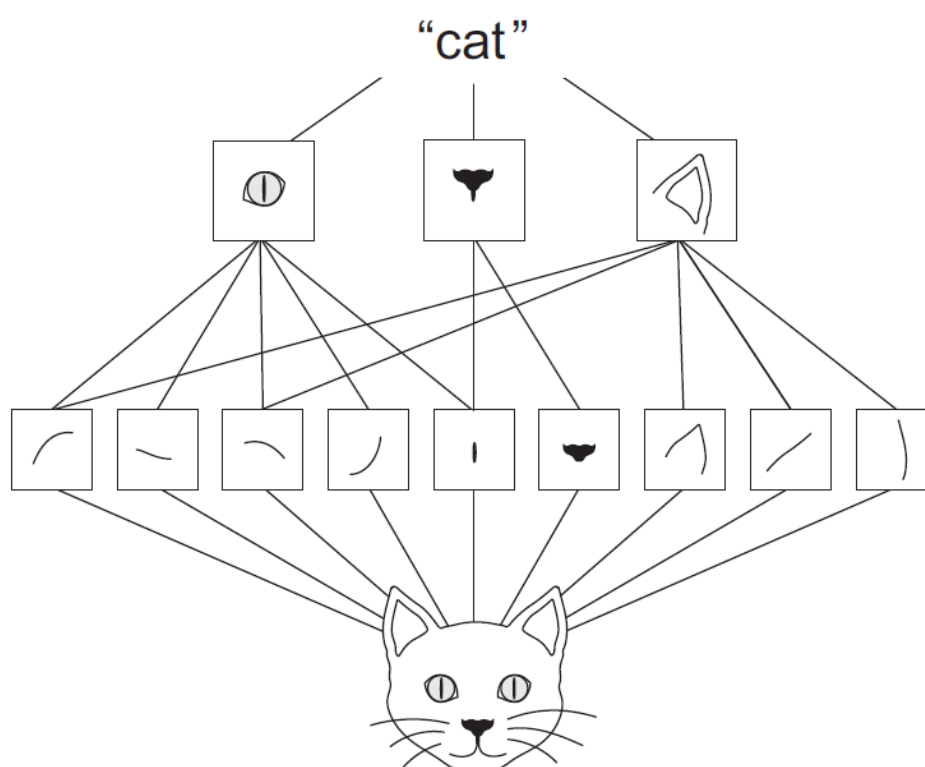
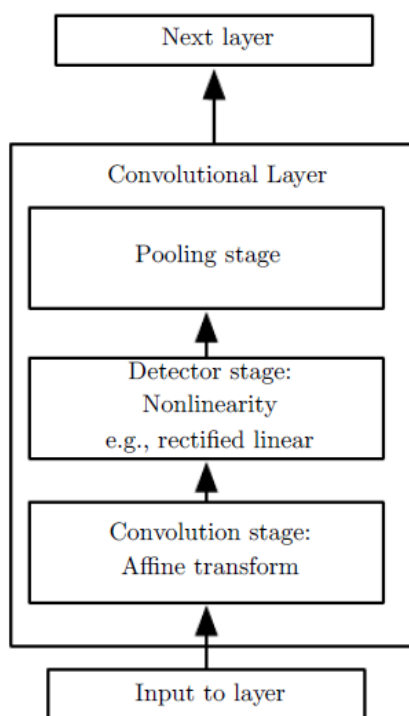


Figura 63 - Filtragem de componentes de uma imagem quando aplicada convolução

## 11.2 POOLING (SUBSAMPLING)

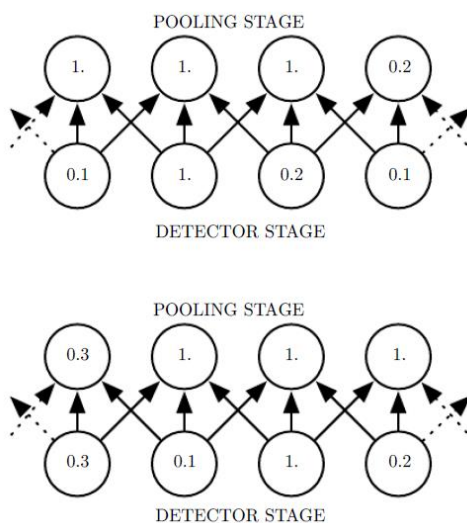
Uma camada de convolução em uma rede neural pode ser descrita por três estágios (Figura 64). O primeiro estágio realiza várias operações de convolução em paralelo para produzir ativações lineares. No segundo estágio, cada ativação linear é aplicada a uma função de ativação não linear, como por exemplo a função ReLU. Este segundo estágio é chamado de detector. Já no terceiro estágio uma função pooling é utilizada para modificar a resposta da camada. A função pooling realiza a substituição de um certo elemento da matriz de resposta gerada no segundo estágio por um valor resultante de cálculo estatístico de elementos respostas em volta deste elemento

específico. Por exemplo, a função pooling mais utilizada na operação de convolução é a função max pooling, a qual retorna a resposta com máximo valor na vizinhança retangular de um certo elemento da matriz resposta (Figura 65).



**Figura 64 - Estágio de uma camada de convolução**

Em todos os casos em que é aplicado a função pooling ajuda a tornar as respostas da camada de convolução invariante a translação elementos da entrada. A invariância a translações locais pode ser bem útil quando se está mais interessado em checar se certo componente está presente do que checar onde está esse componente Figura 66.



**Figura 65 - Exemplo da ação da função max pooling**

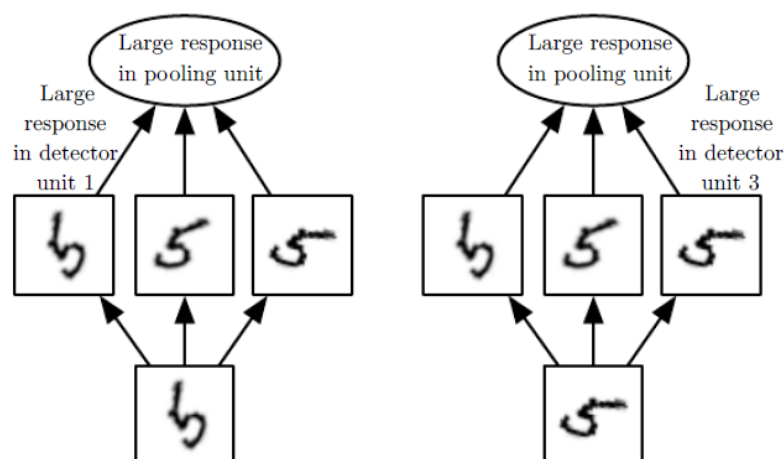


Figura 66 – Exemplo de invariância gerada ao aplicar função pooling

### 11.3 PROCESSO DE TREINAMENTO

O método de treinamento para as redes neurais convolucionais é bem semelhante ao aplicado para redes neurais tradicionais e se baseia também no algoritmo “backpropagation”. A única diferença no processo de treinamento das redes convolucionais é que, ao invés de se ajustar os valores dos parâmetros das unidades de processamento, os parâmetros dos kernels aplicados nas operações de convolução são o foco do processo de otimização.

### 11.4 APLICAÇÃO DA REDE NEURAL CONVOLUCIONAL

Para exemplificar aplicação de uma rede neural convolucional, um problema de classificação de imagens Cachorro vs Gatos é proposto. O problema consiste em treinar um modelo de forma a torná-lo capaz de identificar se a imagem analisada contém um cachorro ou um gato. Esse problema foi proposto para uma competição da plataforma Kaggle (<https://www.kaggle.com/c/dogs-vs-cats>). Os dados para este problema podem ser encontrados no site da Microsoft (<https://www.microsoft.com/en-us/download/details.aspx?id=54765>). Sendo assim, para solução deste problema propõe-se a implementação de uma rede neural convolucional com 3 camadas convolucionais e 2 camadas de unidades de processamento para classificação das amostras. A primeira camada convolucional utilizará 32 canais com kernels (5x5), a segunda camada utilizará 64 canais com kernels (5x5) e a terceira camada utilizará 128 canais com kernels (5x5). Todas as repostas resultantes das operações de convolução são submetidas a funções de ativação ReLU e entre cada uma das camadas de convolução são utilizadas funções max pooling com janelas (2x2). A primeira camada após as camadas convolução conterá 512 unidades de processamento e é utilizada para achatar as respostas das operações de convolução. Já a camada de saída conterá 2 unidades de saída (uma saída para cada classe possível {Gato, Cachorro}) e realizará a classificação das imagens analisadas. Para as unidades de processamento na camada de achatamento opta-se pelo uso da função de ativação ReLU, e nas unidades da camada de saída opta-se pelo uso da

função de ativação Softmax. As amostras de imagens RGB coletadas são convertidas para imagens em escala de cinza com resolução (50x50) pixels ( ).

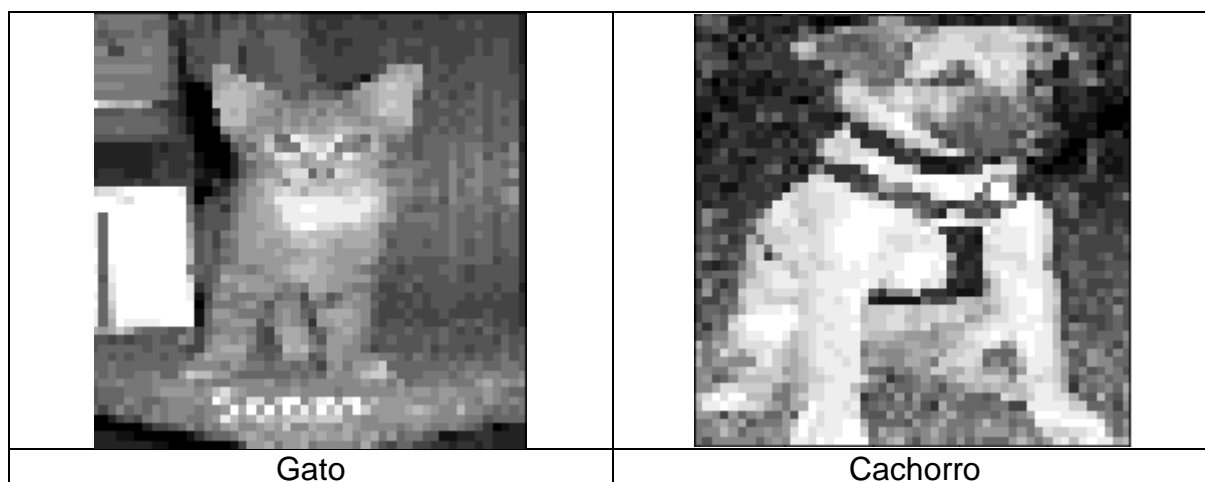


Figura 67 - Exemplo de imagens de gato e cachorro analisadas pela rede neural convolucional

De forma a avaliar a performance de generalização do modelo após treino, estas são divididas em dois grupos: dados de treinamento (utilizados no processo de aprendizado do modelo) e dados de teste (utilizados para avaliar a performance do modelo). O treinamento da rede neural é realizado por meio do algoritmo de “backpropagation” otimizado pelo algoritmo Adam (variação do algoritmo de gradiente descendente) e a função erro utilizada é a MSE (erro quadrado médio). Após o treinamento da rede neural, o processo de teste utilizando amostras nunca vistas pela rede treinada é realizado para avaliar a performance do modelo proposto. A rede neural treinada apresentou uma razoável performance durante o teste apresentando uma acurácia de 75%. Exemplos de classificações realizadas pelo modelo durante o processo de teste são ilustradas na Figura 68.



Figura 68 - Exemplos de classificações realizadas pela rede neural convolucional

## 11.5 CÓDIGO

```
# importando bibliotecas
import os
import cv2
import numpy as np
from tqdm import tqdm
import torch
import matplotlib.pyplot as plt
```

```

import torch.nn as nn
import torch.nn.functional as F
import torch.optim as optim

# coleta e pré-processamento dos dados
# dados para esse problemas foram baixados dos site: https://www.microsoft.com/en-us/download/details.aspx?id=54765
# ao baixar os dados, armazená-
# los em uma pasta nomeada PetImages no mesmo diretório do código
REBUILD_DATA = False
# classe para definição dos dados de treinamento
class DogsVSCats():
    IMG_SIZE = 50
    CATS = "PetImages/Cat"
    DOGS = "PetImages/Dog"
    TESTING = "PetImages/Testing"
    LABELS = {CATS: 0, DOGS: 1}
    training_data = []
    catcount = 0
    dogcount = 0
    def make_training_data(self):
        for label in self.LABELS:
            for f in tqdm(os.listdir(label)):
                if "jpg" in f:
                    try:
                        path = os.path.join(label, f) # caminho para os dados
                        img = cv2.imread(path, cv2.IMREAD_GRAYSCALE) # imagens RGB transformadas em imagens grayscale
                        img = cv2.resize(img, (self.IMG_SIZE, self.IMG_SIZE)) # imagens serão 50x50 pixels
                        self.training_data.append([np.array(img), np.eye(2)[self.LABELS[label]]]) # np.eye(2) -> identity matrix 2x2
                        # contagem de amostras
                        if label == self.CATS:
                            self.catcount += 1
                        elif label == self.DOGS:
                            self.dogcount += 1
                    except Exception as e:
                        pass

        np.random.shuffle(self.training_data) # embaralha as amostras
        np.save('training_data.npy', self.training_data) # salva dados
        print('Cats:', self.catcount) # número de amostras com gatos
        print('Dogs', self.dogcount) # número de amostras com cachorros

if REBUILD_DATA:
    dogsvcats = DogsVSCats()

```

```

dogsvcats.make_training_data()

# dado de treinamento
training_data = np.load("training_data.npy", allow_pickle = True)
# dados de entrada armazenados em um tensor
X = torch.Tensor([i[0] for i in training_data]).view(-1,50,50)
# normalização dos pixels dos dados de entrada
X = X/255.0
# respostas esperadas para os dados de entrada armazenados em um tensor
y = torch.Tensor([i[1] for i in training_data])

# exemplo de imagem
plt.imshow(X[0], cmap='gray')
print(y[0])

# divisão dos dados em dados de treinamento e dados de teste
VAL_PCT = 0.1 # 10% dos dados serão utilizados para teste
val_size = int(len(X)*VAL_PCT)
# divisão
# dados de treinamento
train_X = X[:-val_size]
train_y = y[:-val_size]
# dados de teste
test_X = X[-val_size:]
test_y = y[-val_size:]
print(len(train_X), len(test_X))

# implementação do modelo de rede convolucional a ser aplicada para o problema
# opta-se pela implementação de uma rede neural de 3 camadas de convolução, uma camada de achatamento
# e uma camada de saída
class Net(nn.Module):
    def __init__(self):
        super().__init__()
        # camada 1 de convolução
        self.conv1 = nn.Conv2d(1, 32, 5) # entrada: 1 imagem (50x50), 32 canais de saída, (5x5) kernel
        # camada 2 de convolução
        self.conv2 = nn.Conv2d(32, 64, 5) # 32 entradas, 64 canais de saída, (5x5) kernel
        # camada 3 de convolução
        self.conv3 = nn.Conv2d(64, 128, 5) # 64 entradas, 128 canais de saída, (5x5) kernel

        # descobre a dimensão da resposta da terceira camada de convolução
        x = torch.randn(50,50).view(-1, 1, 50, 50)
        self.flatten_x_len = int(len(self.convs(x)))

```



```

        # camadas de unidades de processamento
        self.fc1 = nn.Linear(self.flatten_x_len, 512) # camada com 512 unidades de processamento
        self.fc2 = nn.Linear(512, 2) # camada com 2 unidades de saída (número de classes possíveis)

    # propagação forward nas camadas de convolução
    # entre cada camada de convolução aplica-se a função max pooling com janela (2x2)
    def convs(self, x):
        # max pooling over 2x2
        x = F.max_pool2d(F.relu(self.conv1(x)), (2,2)) # função max pooling com janela (2x2)
        x = F.max_pool2d(F.relu(self.conv2(x)), (2,2)) # função max pooling com janela (2x2)
        x = F.max_pool2d(F.relu(self.conv3(x)), (2,2)) # função max pooling com janela (2x2)
        # resposta da terceira camada de convolução é achatada para classificação da amostra
        x = torch.flatten(x) # flattening -
> there is a torch.flatten(Tensor) function
        return x

    # propagação forward na rede neural
    def forward(self, x):
        x = self.convs(x)
        x = x.view(-1, self.flatten_x_len)
        x = F.relu(self.fc1(x))
        x = self.fc2(x) # output layer
        return F.softmax(x, dim=1)

# inicia rede neural convolucional
net = Net()
print(net) # resumo da rede neural implementada

# otimizador utilizado para treinamento da rede
optimizer = optim.Adam(net.parameters(), lr=0.001) # opta-se por utilizar o algoritmo Adam
# função erro para avaliar respostas geradas pela rede neural
loss_function = nn.MSELoss() # opta-se por utilizar a função de erro quadrado médio (MSELoss)

# treinamento da rede neural
BATCH_SIZE = 100 # tamanho das bateladas de amostras a serem utilizadas no treinamento
EPOCHS = 3 # quantas passagens sobre todas as amostras de treinamento serão utilizadas

```

```

loss_log = []
for epoch in range(EPOCHS):
    loss_epoch = []
    for i in tqdm(range(0, len(train_X), BATCH_SIZE)): # iteração de passagens forward e backward
        batch_X = train_X[i:i+BATCH_SIZE].view(-1,1,50,50) # batelada de dados de entrada
        batch_y = train_y[i:i+BATCH_SIZE] # respostas esperadas para os dados de entrada
        # zera os gradientes da rede neural
        net.zero_grad()
        # propagação forward
        outputs = net(batch_X)
        # cálculo dos erros das respostas geradas pela rede neural para amostras de entradas
        loss = loss_function(outputs, batch_y)
        # propagação backward
        loss.backward()
        # ajuste dos parâmetros da rede neural
        optimizer.step()
        # armazenamento dos erros das bateladas
        loss_epoch.append(loss.item())
    # armazenamento dos erros da passagem
    loss_log.append(sum(loss_epoch))
    print(f"Epoch: {epoch}. Loss: {loss}")
# plot função erro x epochs
plt.plot(range(EPOCHS), loss_log)
plt.xlabel("Epochs")
plt.ylabel("Loss")
plt.show()

# teste da rede neural
correct = 0
total = 0
with torch.no_grad(): # desativa o cálculo de gradientes
    for i in tqdm(range(len(test_X))): # propagação dos dados de teste
        real_class = torch.argmax(test_y[i])
        net_out = net(test_X[i].view(-1,1,50,50))[0] # repostas geradas para os dados teste
        predicted_class = torch.argmax(net_out)
        # avaliação da acurácia do modelo
        if predicted_class == real_class:
            correct += 1
        total += 1
print("Accuracy: ", round(correct/total, 3))

```

Figura 69 - Código em Python para implementação da rede neural convolucional

## **11.6 BIBLIOGRAFIA**

LECUN, Yann et al. Convolutional networks for images, speech, and time series. The handbook of brain theory and neural networks, v. 3361, n. 10, p. 1995, 1995.

KRIZHEVSKY, Alex; SUTSKEVER, Ilya; HINTON, Geoffrey E. Imagenet classification with deep convolutional neural networks. In: Advances in neural information processing systems. 2012. p. 1097-1105.

GOODFELLOW, Ian; BENGIO, Yoshua; COURVILLE, Aaron. Deep learning. MIT press, 2016.

FRANÇOIS, Chollet. Deep learning with Python. 2017.

## 12 REDE NEURAL RECORRENTE LSTM

As redes neurais recorrentes (RUMELHART et. al., 1986) são uma família de redes neurais especializadas no processamento de dados sequenciais, como por exemplo dados de texto, som e séries temporais. Redes neurais deste tipo são caracterizadas por apresentarem conexões de retroalimentação (feedback loop) de informações já processadas entre unidades e camadas de processamento. A Figura 70 ilustra uma rede neural recorrente, onde  $A$  representa uma rede neural alimentada por uma entrada  $X_t$ . Ao processar  $X_t$ , a rede gera uma resposta  $h_t$ , a qual é realimentada em  $A$  e utilizada no processamento do dado seguinte.

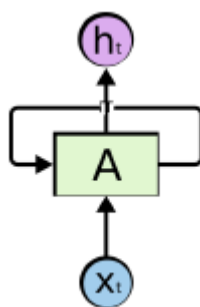


Figura 70 - Exemplo de rede neural recorrente com loop de retroalimentação

Pode-se pensar que a estrutura de redes neurais com loops de retroalimentação e a estrutura de redes neurais tradicionais sejam bem distintas, entretanto, na realidade elas são bem similares. Uma rede neural recorrente pode ser vista como múltiplas cópias de uma mesma rede neural tradicional conectadas em sequência, tal que a informação gerada por uma rede é passada para a rede seguinte (Figura 71).

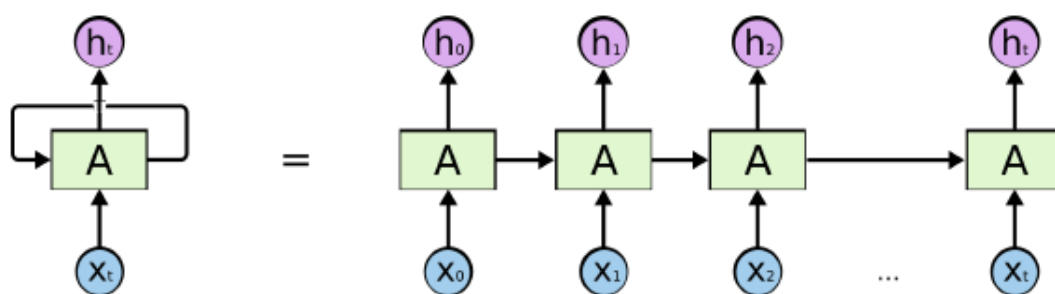


Figura 71 - Rede neural recorrente como uma sequência de redes neurais

As redes neurais recorrentes possuem diversas propriedades que as tornam atrativas para problemas com dados sequenciais: elas são flexíveis no uso de informações contextual (são capazes de aprender qual informação deve ser armazenada e qual informação deve ser ignorada), aceitam diversos tipos de representações de dados sequenciais, e são capazes de reconhecer padrões sequenciais na presença de distorções sequenciais (GRAVES, 2012). Apesar destas vantagens, as redes neurais recorrentes com simples loops de retroalimentação (como apresentados acima) apresentam limitações quando aplicadas para problemas reais de dados sequenciais. Estas redes recorrentes tradicionais apresentam dificuldade em armazenar

informações extraídas por um longo período de tempo, o que limita a análise de dados que dependem de informações de dados anteriores, uma vez que apenas uma limitada quantidade de dados passados pode ser acessada. As redes Long Short-Term Memory (LSTM) (HOCHREITER; SCHMIDHUBER, 1997) são um tipo específico de redes neurais recorrentes criadas com intuito superar estas limitações enfrentadas pelas simples redes recorrentes. As redes LSTM são caracterizadas pelo uso de unidades de memória que possibilitam o armazenamento de informações na rede neural por um maior período de tempo, mas também possibilitam o esquecimento de informações que já não são mais relevantes para rede, ou seja, estas unidades de memória otimizam o gerenciamento de informações na rede neural. Recentemente, as redes LSTM vêm se destacando pelo seu sucesso em problemas de reconhecimento de som, modelagem de linguagem, tradução, análise de séries temporais, as quais são aplicações que dependem da persistência de informações. Sendo assim, um estudo mais detalhado deste tipo de rede neural recorrente é considerado relevante.

### 12.1 LONG SHORT-TERM MEMORY

Tal como citado anteriormente, a principal limitação das redes neurais recorrentes tradicionais é a sua dificuldade de manter informações por um longo período de tempo. Esta limitação se deve ao problema de desaparecimento de gradiente (“vanishing gradient problem”) o qual ocorre pelo fato da influência de uma certa entrada em uma camada oculta (e consequentemente na camada de saída) se reduzir exponencialmente a zero ao decorrer dos ciclos recorrência. Na Figura 72, a qual ilustra o problema de “vanishing gradient”, nota-se que a influência da primeira entrada 1 (representada em escala de cinza) vai decaindo com o decorrer do tempo, o que demonstra o rápido esquecimento da informação fornecida pela primeira entrada.

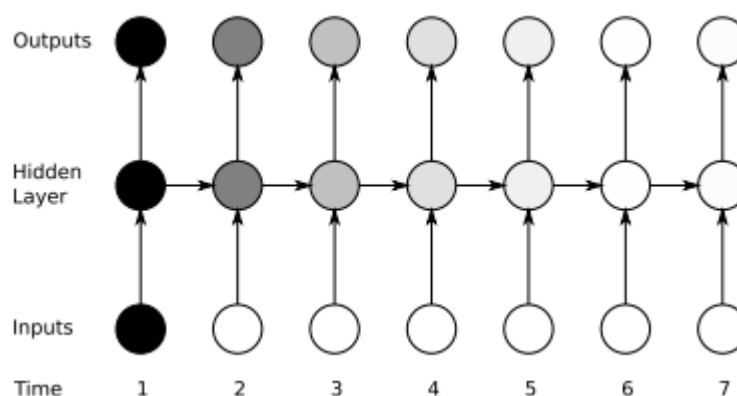


Figura 72 – Gradient vanishing problem

Para evitar o problema de “vanishing gradient”, nas redes LSTM é proposto o uso de um conjunto de subredes conectadas recorrentemente, chamado de unidade de memória, as quais possibilitam a persistência de informações na rede neural por um maior período de tempo. Tal como foi descrito anteriormente, as redes neurais podem ser estruturadas por meio de cadeias de módulos repetidos de redes neurais. No caso das redes recorrentes tradicionais, o módulo de rede neural implementado é bem

simples, sendo composto por exemplo por uma única camada de processamento com função de ativação  $\tanh$ :

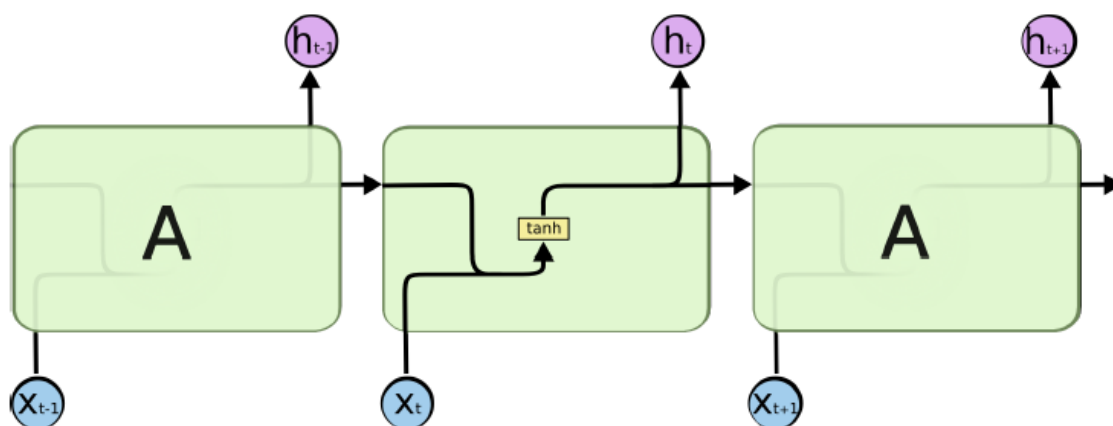


Figura 73 - Módulo de repetição utilizado em redes recorrentes tradicionais

As redes LSTM também podem ser estruturadas em cadeias, entretanto o módulo a ser repetido durante sequência de redes neurais é mais complexo. Ao invés de utilizar uma simples camada de processamento, as redes LSTM utilizam uma unidade de memória composta por quatro camadas de processamento:

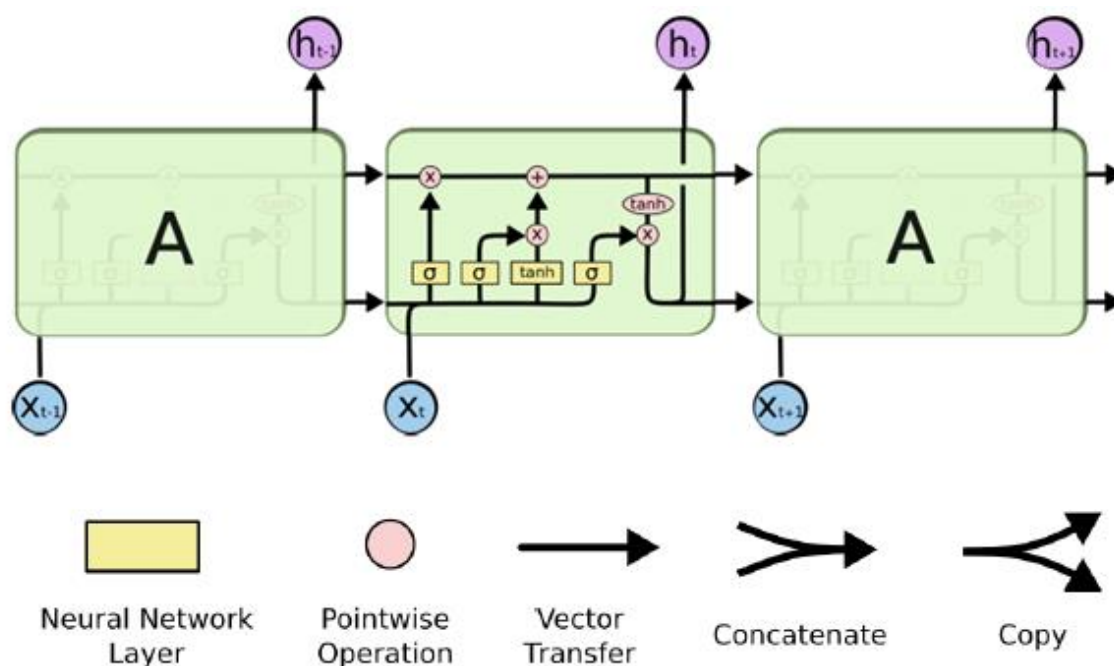


Figura 74 - Módulo de repetição nas redes LSTM

Nesta célula de memória, cada representa a transmissão de informação em formato de vetor, da saída de um nó, para a entrada de outros nós. Os círculos em rosa representam operações realizadas na unidade, enquanto que caixas amarelas representam camadas de processamento a serem treinadas.

O componente chave da rede LSTM é a sua célula de estado (linha horizontal no topo da unidade de memória da rede):

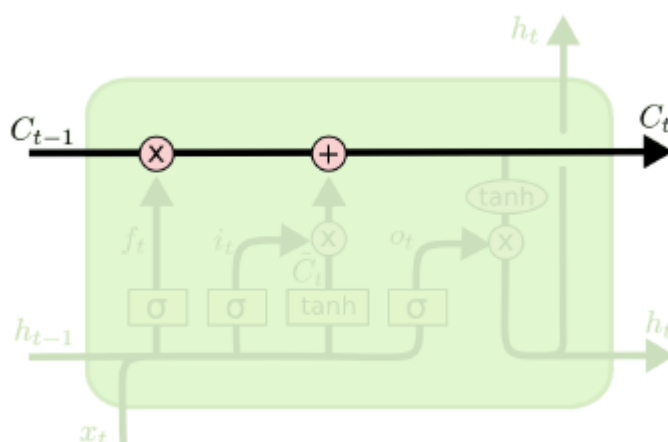


Figura 75 - Célula de estado

A rede LSTM tem a capacidade de remover e adicionar informação à célula de estado por meio de estruturas de regulação de informação chamados de gates. Cada gate é composto por uma camada de unidades de processamento com funções de ativação sigmóide e uma operação de multiplicação:

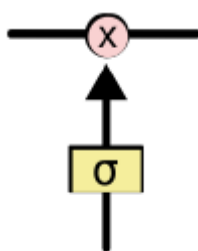


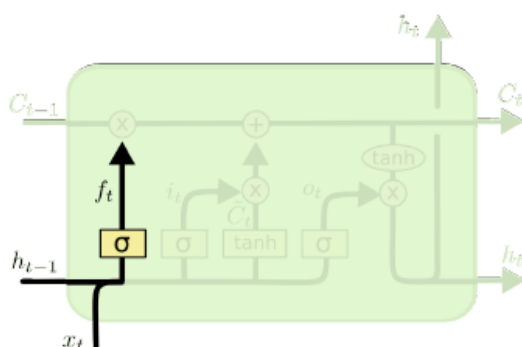
Figura 76 - Gate da célula de memória

As unidades de processamento com funções sigmóides geram respostas entre 0 e 1, descrevendo quanto de cada informação deve continuar sendo propagada, onde 0 significa que nada deve ser propagado, enquanto 1 significa que toda informação deve ser propagada. Cada unidade de memória da rede LSTM possui três gates de controle e proteção da célula de estado: o gate de esquecimento (“forget gate”), o gate de entrada (“input gate”) e o gate de saída (“output gate”). Cada um desses gates é explicado a seguir.

### 12.1.1 FORGET GATE

O primeiro passo na unidade de memória da rede LSTM é decidir qual informação será removida (deletada) da célula de estado. Esta decisão é realizada pelo gate de esquecimento (“forget gate”) (Figura 77). Este gate recebe como entradas  $x_t$  (dado de entrada) e  $h_{t-1}$  (resposta gerada pelo módulo anterior), e fornece como resposta um número entre 0 e 1 para cada número na célula de estado  $C_{t-1}$ . Se a resposta for 1 a

informação completa deverá ser mantida e se a resposta for 0 toda a informação deve ser removida.

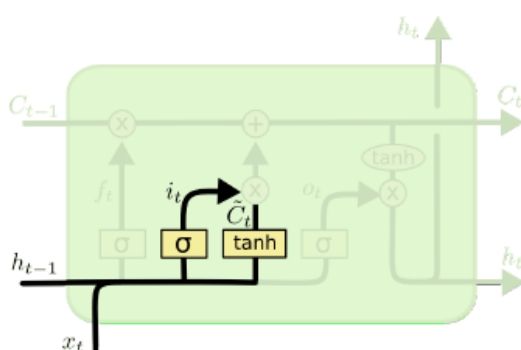


$$f_t = \sigma(W_f \cdot [h_{t-1}, x_t] + b_f)$$

Figura 77 - Forget gate

### 12.1.2 INPUT GATE

Após decidir quais informações devem ser mantidas, o próximo passo é decidir quais novas informações serão armazenadas na célula de estado. Este passo possui duas etapas. Na primeira etapa, a camada de unidades de processamento sigmóide, chamada de gate de entrada ("input gate"), tem como função decidir quais informações da célula de estado serão atualizadas. Na segunda etapa, a camada de processamento com funções de ativação  $\tanh$  cria um vetor com novos valores para  $\tilde{C}_t$  que são candidatos a serem adicionados à célula de estado. Ao combinar o resultado destas etapas, valores para atualização são fornecidos para a célula de estado (Figura 78).



$$i_t = \sigma(W_i \cdot [h_{t-1}, x_t] + b_i)$$

$$\tilde{C}_t = \tanh(W_C \cdot [h_{t-1}, x_t] + b_C)$$

Figura 78 - Input gate

Para atualizar a célula de estado primeiro é realizado a multiplicação do estado anterior  $C_{t-1}$  por  $f_t$  para remoção das informações consideradas descartáveis pelo "forget gate" e em seguida se adiciona  $i_t * \tilde{C}_t$  (candidatos a novos valores na célula de estado) (Figura 79).



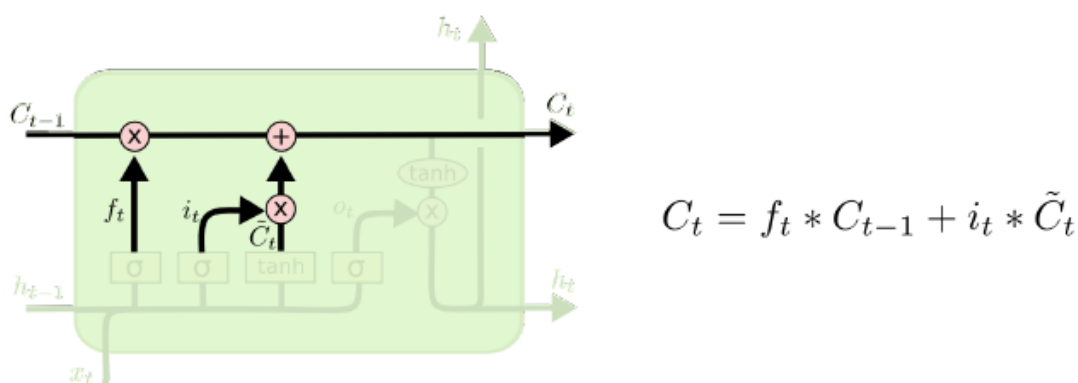


Figura 79 - Atualização da célula de estado

### 12.1.3 OUTPUT GATE

Por fim, a unidade de memória decide quais informações serão fornecidas como resposta. Para a geração da resposta aplica-se um filtro aos valores da célula de estado. Para isso, primeiramente utiliza-se uma camada de processamento com funções sigmóides (“output gate”) para decidir quais informações da célula de estado serão fornecidas como resposta. Em seguida as informações da célula de estado são introduzidas a uma função  $\tanh$  (para limitar seus valores ao intervalo de -1 a 1) e são multiplicadas pelo resultado gerado pelo “output gate”, tal que apenas informações definidas como resposta sejam fornecidas.

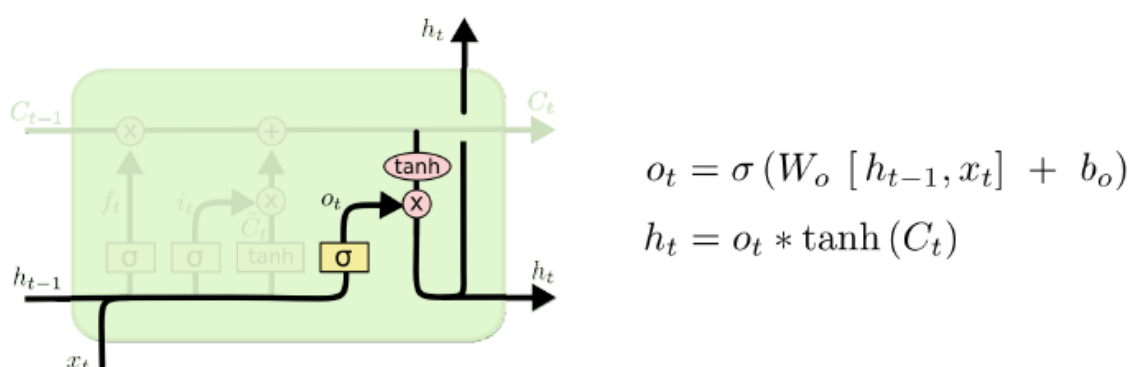


Figura 80 - Output gate

## 12.2 PROCESSO DE TREINAMENTO

Pelo fato de as redes neurais recorrentes em geral apresentarem estruturas de múltiplas redes neurais conectadas em cadeia, o algoritmo tradicional de “backpropagation” utilizado para redes neurais tradicionais não pode ser utilizado para as redes LSTM. Para o treinamento de redes recorrentes, como as redes LSTM, geralmente opta-se pelo uso do algoritmo de “backpropagation through time” (BPTT). Como o algoritmo tradicional de “backpropagation”, o algoritmo BPTT também consiste a repetida aplicação da regra da cadeia ao longo durante a propagação “backward” na rede neural. Entretanto, diferente do caso de redes neurais tradicionais,

nas redes recorrentes a função erro depende da influência das ativações das camadas ocultas na camada de saída, mas também depende da influência destas ativações no timestep seguinte. Sendo assim, os cálculos dos gradientes no algoritmo BPTT são diferentes comparado aos do algoritmo tradicional. O algoritmo BPTT é detalhado em (WILLIAMS; ZILPSE, 1995).

### 12.3 APLICAÇÃO DE REDE NEURAL RECORRENTE LSTM

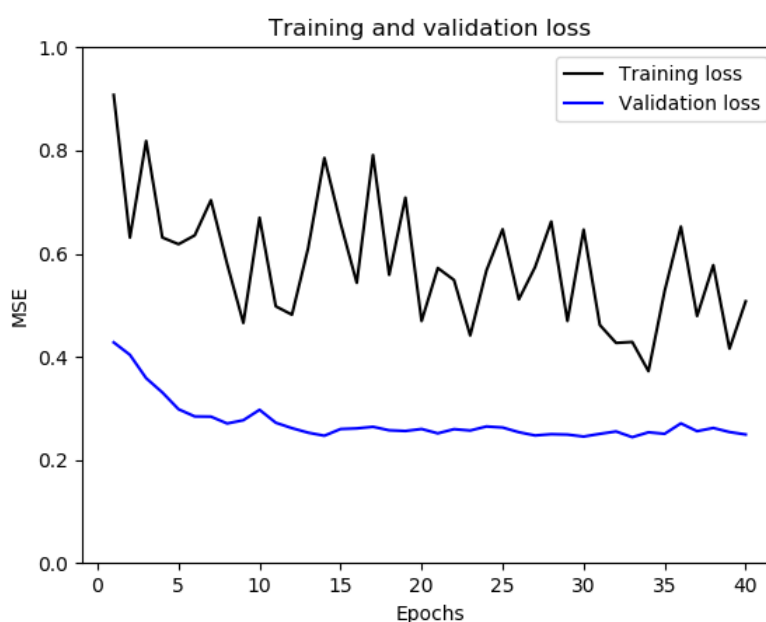
Com objetivo de explorar o potencial das redes neurais recorrentes LSTM, um caso de estudo com dados de séries temporais de um processo real da indústria de papel e celulose foi proposto. Os dados para este caso de estudo consistem em dados de operação de um processo Kraft para produção de celulose. Basicamente, o processo Kraft consiste na aplicação de um licor de digestão na madeira designada para produção de celulose. Este licor, chamado de licor branco, é composto por hidróxido de sódio, sulfeto de sódio e carbonato de sódio. Além da celulose, o processo Kraft também gera um licor resultante da reação do licor branco com a madeira, chamado de licor preto. Este licor preto é caracterizado por conter componentes restos da madeira, mas também por conter uma grande quantidade de substâncias resultantes do licor branco, como sulfato de sódio e carbonato de sódio, os quais devem ser recuperados. Sendo assim, o licor preto é geralmente submetido a um processo de recuperação química. Este processo de recuperação química envolve a concentração do licor preto em evaporadores, seguido da combustão em um forno de recuperação para redução do sulfato de sódio em sulfeto de sódio, gerando o licor verde. O licor verde então é direcionado para um processo de causticização onde reage com carbonato de cálcio, para geração de hidróxido de sódio e assim a recuperação do licor branco, que é reciclado no processo Kraft. Para avaliar o processo de recuperação química dois índices são utilizados: o álcali total (AT) e o álcali efetivo (AE). Estes índices são calculados por meio das concentrações de hidróxido de sódio, carbonato de sódio e sulfeto de sódio presentes no licor branco recuperado e são de extrema importância para a avaliação da performance do processo de recuperação.

$$TA = [\text{NaOH}] + [\text{Na}_2\text{CO}_3] + [\text{Na}_2\text{S}]$$

$$EA = [\text{NaOH}] + [\text{Na}_2\text{S}]/2$$

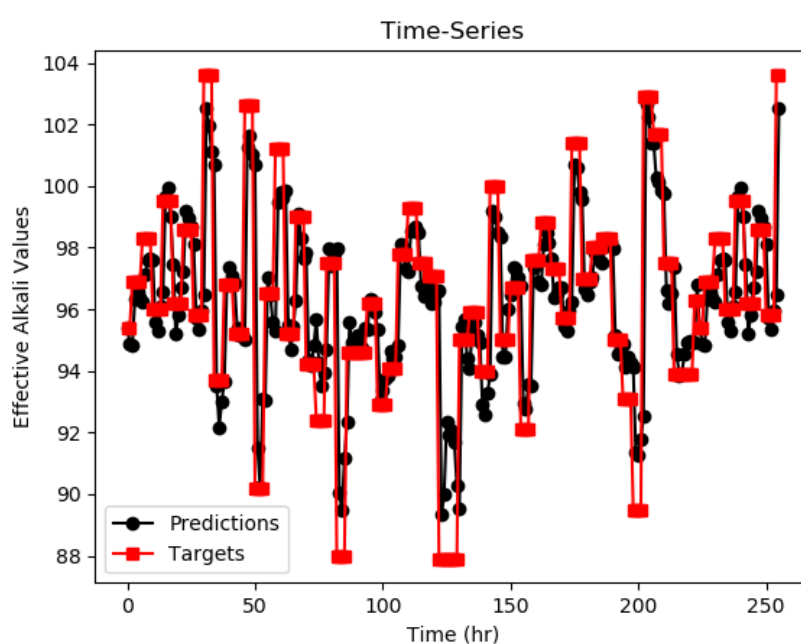
Sendo assim, o estudo de caso propõe a elaboração de um regressor baseado em redes LSTM para previsão do índice AE. Para treinamento deste modelo é proposto a utilização de 6 variáveis do processo de recuperação: vazão mássica de óxido de sódio puro, vazão mássica de carbonato de cálcio presente na corrente de óxido de sódio, o índice AT do licor verde, a temperatura de alimentação de licor verde e a temperatura de operação durante a reação de causticização. A rede LSTM proposta para a solução do problema contém uma camada com 32 unidades de memória LSTM e uma camada de saída com apenas uma unidade processamento (em problemas de regressão a saída da rede neural só deve conter uma saída). O algoritmo de otimização é o RMSProp e a função erro utilizada é o erro quadrado médio (MSE). O

treinamento é realizado com 40 passagens sobre os dados de treino. A curva da função erro durante o treinamento pode ser observada na Figura 81.



**Figura 81 - Função erro ao longo do treinamento da rede LSTM**

Para avaliar a capacidade de generalização do modelo treinado, realizou-se uma operação de teste com fornecimento de dados de operação nunca vistos pela rede neural treinada. O teste resultou em um erro quadrado médio de 0.22, o que é resultado satisfatório. A comparação dos valores de AE gerados pela rede neural e os valores esperados de AE é ilustrada na Figura 82.



**Figura 82 - Comparação: valores previstos x valores esperados**

### 12.3.1 CÓDIGO

```

import os
import numpy as np
from math import ceil
from matplotlib import pyplot as plt
from keras.models import Sequential
from keras import layers
from keras.optimizers import RMSprop
from sklearn.metrics import mean_squared_error

## coleta dos dados
# diretório dos dados
data_dir = '/Data/Directory/Path'
fname = os.path.join(data_dir, 'caustificacao_dados.csv')
# leitura dos dados
f = open(fname)
data = f.read()
f.close()
# convertendo em matriz
lines = data.split('\n')
num_rows = len(lines)
num_cols = len(lines[0].split(','))-1
float_data = np.zeros((num_rows, num_cols))
for i, line in enumerate(lines):
    values = [float(value) for value in line.split(',')[1:]]
    float_data[i, :] = values

## normalização dos dados de treinamento
mean_alc_ef = float_data[:4380, -1].mean(axis=0)
std_alc_ef = float_data[:4380, -1].std(axis=0)
mean = float_data[:4380].mean(axis=0)
float_data -= mean
std = float_data[:4380].std(axis=0)
float_data /= std

## função para carregamento dos dados para rede neural
def generator(data, lookback, delay, min_index, max_index, shuffle=False, batch_size=32, step=1):
    # data: the original array of floating-point data, which was normalized
    # lookback: how many timesteps back the input data should go (set it to 12 to go back 24 hours)
    # delay: how many timesteps in the future the target should be (set it to 0 to get targets at the current timestep)
    # min_index and max_index: indices in the data array that delimit which time-steps to draw from, useful
    # for keeping a segment of for validation and another for testing

```

```

# shuffle: whether to shuffle the samples or draw them in chronological
order
# batch_size: the number of samples per batch
# step: the period, in timesteps, at which you sample data (set it to 1
in order to draw one data point
# every 2 hours)
# a timestep is 2 hours

if max_index is None:
    max_index = len(data) - delay - 1
i = min_index + lookback
while 1:
    if shuffle:
        rows = np.random.randint(min_index + lookback, max_index, size =
batch_size)
    else:
        if i + batch_size >= max_index:
            i = min_index + lookback
        rows = np.arange(i, min(i + batch_size, max_index)) # create evenl
y spaced values
        i += len(rows)
        samples = np.zeros((len(rows), lookback // step, data.shape[-
1])) # 3D array (batch-size, lookback, num_cols)
        targets = np.zeros((len(rows),)) # 1D array for target values
        for j, row in enumerate(rows):
            indices = range(row - lookback, row, step) # indice of the datab
ase values that will be use as lookback values
            samples[j] = data[indices]
            targets[j] = data[row + delay][-1]
        yield samples, targets # generate one batch

# geração dos dados de treino, validação e teste
lookback = 12 # 24 hours before
step = 1 # analysis at each 2 hours
delay = 0 # forecast the value at current time step
batch_size = 32

# dados de treino
train_gen = generator(float_data, lookback=lookback, delay=delay, min_index=
0, max_index=4380,
                        shuffle=True, step=step, batch_size=batch_size)

# dados de validação
val_gen = generator(float_data, lookback=lookback, delay=delay, min_index=43
80, max_index=7284,
                    step=step, batch_size=batch_size)

# dados de teste
test_gen = generator(float_data, lookback=lookback, delay=delay, min_index=7
284, max_index=None,
                    step=step, batch size=batch size)

```

```

# dados de teste
target_gen = generator(float_data, lookback=lookback, delay=delay, min_index
=7284, max_index=None,
                        step=step, batch_size=batch_size)

# dados de teste
eval_test_gen = generator(float_data, lookback=lookback, delay=delay, min_in
dex=7284, max_index=None,
                        step=step, batch_size=batch_size)

# número de iterações por epoch durante treino
train_steps = ceil((4380 - 0 - lookback) / batch_size)
# número de iterações por epoch durante validação
val_steps = ceil((7284 - 4380 - lookback) / batch_size) # how many steps to
draw from val_gen in order to see the entire validation set

# número de iterações por epoch durante teste
test_steps = ceil((len(float_data) - 7284 - lookback) / batch_size) # how ma
ny steps to draw from test_gen in order to see the entire test set

## modelo da rede lstm
# opta-
se pela implementação de uma rede LSTM com uma camada com 32 unidades LSTM e
uma camada de saída
model = Sequential()
model.add(layers.LSTM(32, dropout=0.3, recurrent_dropout=0.3, input_shape=(N
one, float_data.shape[-1]))) # utiliza-se dropout para evitar overfitting
model.add(layers.Dense(1))
## treinamento
# otimizador utilizado para o backpropagation through time é o RMSprop
# função erro é MSE
model.compile(optimizer=RMSprop(), loss='mse')
# inicia treinamento
history = model.fit_generator(train_gen, steps_per_epoch=train_steps, epochs
=40, validation_data=val_gen,
                            validation_steps=val_steps)

## resultados de treinamento

# plot
loss = history.history['loss']
val_loss = history.history['val_loss']
epochs = range(1, len(loss) + 1)

mean_loss = np.mean(loss)
print(f'The mean training loss is {mean_loss}')

mean_val_loss = np.mean(val_loss)
print(f'The mean validation loss is {mean_val_loss}')

```

```

# loss
plt.figure()
plt.plot(epochs, loss, 'k', label='Training loss')
plt.plot(epochs, val_loss, 'b-', label='Validation loss')
plt.ylim((0, 1))
plt.xlabel('Epochs')
plt.ylabel('MSE')
plt.title('Training and validation loss')
plt.legend()
plt.show()

## teste
evaluation = model.evaluate_generator(eval_test_gen, steps=test_steps)
print(f'The test loss is {evaluation}')

# plot resultado do teste
# targets x predictions

predictions = model.predict_generator(test_gen, steps=test_steps)

samples, targets = next(target_gen)
for i in range(1, test_steps):
    sample, target = next(target_gen)
    samples = np.concatenate((samples, sample), axis=0)
    targets = np.concatenate((targets, target))

print(f'Normalized MSE {mean_squared_error(predictions, targets)}')

r_targets = (targets*std_alc_ef) + mean_alc_ef
r_predictions = (predictions*std_alc_ef) + mean_alc_ef

print(f'Unnormalized MSE {mean_squared_error(r_predictions, r_targets)}')

plt.figure()
plt.plot(r_targets, r_predictions, 'ro', label='Predictions')
plt.plot(r_targets, r_targets, 'k', label='x=y')
plt.xlabel('Target Values')
plt.ylabel('Prediction Values')
#plt.xlim((97, 100))
#plt.ylim((97, 100))
plt.title('Targets x Predictions')
plt.legend()
plt.show()

# targets x time vs predictions x time
time = range(0, len(targets))

plt.figure()
plt.plot(time, r_predictions, 'ko--', label='Predictions')

```

```
plt.plot(time, r_targets, 'rd--', label='Targets')
plt.xlabel('Time (hr)')
plt.ylabel('Effective Alkali Values')
plt.title('Time-Series')
plt.legend()
plt.show()
```

Figura 83 - Código em Python para implementação da rede neural LSTM

## 12.4 BIBLIOGRAFIA

RUMELHART, David E. et al. Sequential thought processes in PDP models. Parallel distributed processing: explorations in the microstructures of cognition, v. 2, p. 3-57, 1986.

GRAVES, Alex. Supervised sequence labelling. In: Supervised sequence labelling with recurrent neural networks. Springer, Berlin, Heidelberg, 2012. p. 5-13.

HOCHREITER, Sepp; SCHMIDHUBER, Jürgen. Long short-term memory. Neural computation, v. 9, n. 8, p. 1735-1780, 1997.

WILLIAMS, Ronald J.; ZIPSER, David. Gradient-based learning algorithms for recurrent. Backpropagation: Theory, architectures, and applications, v. 433, 1995.

OLAH, Christopher. Understanding LSTM Networks. url: <https://colah.github.io/posts/2015-08-Understanding-LSTMs/>, 2015.



### 13 SUPPORT VECTOR CLUSTERING (SVC)

O support vector clustering, ou SVC, é algoritmo de agrupamento de dados proposto por (BEN-HUR et al., 2001). O algoritmo foi desenvolvido como uma extensão dos algoritmos de support vector machine (SVM), os quais são extensivamente utilizados para problemas de classificação e regressão. O algoritmo SVC se baseia nos conceitos dos algoritmos SVM, porém em uma abordagem de aprendizado não supervisionado.

O algoritmo opera realizando o mapeamento dos dados analisados originais para um espaço de componentes com maior dimensão utilizando um kernel gaussiano. Com a imagem dos dados nesse espaço de maior dimensão, então realiza-se uma busca pela menor esfera que englobe todas as imagens. Uma vez determinada a esfera, esta é mapeada de volta para o espaço original dos dados, onde forma os contornos que englobam os dados. Estes contornos são interpretados como fronteiras que englobam os diferentes grupos de dados, os clusters. Dados englobados pelo mesmo contorno são associados ao mesmo cluster. O algoritmo consegue manipular dados outliers pelo uso de uma constante de margem suave que permite que a esfera no espaço de componentes de maior dimensão não necessite englobar todas as imagens de dados mapeadas. A implementação do algoritmo SVC pode ser dividido em duas principais etapas: a determinação da menor esfera e a determinação dos clusters. Uma descrição mais detalhada destas etapas é realizada nas próximas seções.

#### 13.1 ALGORITMO

##### 13.1.1 DETERMINAÇÃO DA HIPERESFERA

Para determinação da esfera, e conseqüentemente dos contornos dos clusters, formula-se uma descrição de vetores suporte sobre um conjunto de dados. Considere o conjunto de dados contendo  $N$  pontos  $\{x_i\} \subseteq \chi$  no espaço de dados  $\chi \subseteq \mathbb{R}^d$ . Utilizando uma transformação não-linear  $\phi$  do espaço original  $\chi$  para um espaço de componentes de maior dimensão, busca-se determinar a menor esfera com raio  $R$  que englobe todas as imagens dos dados no novo espaço. Este processo de busca é descrito pelo seguinte problema de otimização:

$$\min_{R,a} R^2$$

$$\|\phi(x_j) - a\|^2 \leq R^2 \quad \forall j$$

Onde  $\|\cdot\|$  representa o cálculo de norma euclidiana e  $a$  é o centro da esfera. Uma restrição suave é incorporada ao problema original pela adição de uma variável folga  $\xi_j$  à restrição e um termo de penalidade  $C \sum_j \xi_j$  à função objetivo:

$$\min_{R,a} R^2 + C \sum_j \xi_j$$

$$\|\phi(x_j) - a\|^2 \leq R^2 + \xi_j \quad \forall j$$

$$\xi_j \geq 0 \forall j$$

A adição  $\xi_j \geq 0$  torna o problema robusto a outliers. Para solução deste problema de otimização a função lagrangeana é introduzida:

$$\mathcal{L}(R, a, \xi, \beta, \mu) = R^2 + C \sum_j \xi_j - \sum_j \beta_j (R^2 + \xi_j - \|\phi(x_j) - a\|^2) - \sum_j \xi_j \mu_j$$

Onde  $\beta_j \geq 0$  e  $\mu_j \geq 0$  são multiplicadores de Langrange e  $C$  é uma contante. Zerando o gradiente de  $\mathcal{L}$  as condições de estabilidade são obtidas:

- $\nabla_R \mathcal{L} = 0$ :

$$\begin{aligned} \nabla_R \mathcal{L} &= 2R - \sum_j 2\beta_j R = 0 \\ \sum_j \beta_j &= 1 \end{aligned}$$

- $\nabla_a \mathcal{L} = 0$ :

$$\begin{aligned} \nabla_a \mathcal{L} &= \nabla_a \sum_j \beta_j (-\|\phi(x_j) - a\|^2) = 0 \\ \nabla_a \mathcal{L} &= 2a \sum_j \beta_j - 2 \sum_j \beta_j \phi(x_j) = 0 \\ a &= \sum_j \beta_j \phi(x_j) \end{aligned}$$

- $\nabla_{\xi_j} \mathcal{L} = 0$ :

$$\begin{aligned} \nabla_{\xi_j} \mathcal{L} &= C - \beta_j - \mu_j = 0 \\ \beta_j &= C - \mu_j \end{aligned}$$

Considerando também as condições KKT para o problema tem-se que:

$$\beta_j (R^2 + \xi_j - \|\phi(x_j) - a\|^2) = 0$$

$$\xi_j \mu_j = 0$$

Sendo assim, tendo em mãos estas condições, pode-se definir como vetores suportes (SVs: support vectors) como as imagens dos dados que se encontram na fronteira da esfera. Já os vetores suportes delimitados (BSVs: bounded support vectors) são as imagens de dados que se encontram fora da esfera. Para os dados BSVs sabe-se que que  $\xi_j > 0$ , então pelas condições de KKT e de estabilidade conclui-se que os BSVs serão aqueles dados em que  $\mu_j = 0$  e  $\beta_j = C$ . Os dados SV, por estarem na superfície da esfera, serão caracterizados por  $0 < \beta_j < C$ . Os demais pontos que estiverem no interior da esfera terão  $\xi_j = 0$  e  $\beta_j = 0$ . Caso  $C = 1$ , nenhum BSV irá existir, pois  $\sum_j \beta_j = 1$ .

Dado que o problema de otimização é continuamente diferenciável e convexo, tem-se que

$$\max_{\mu, \beta} \left( \min_{R, a, \xi} (\mathcal{L}(R, a, \xi, \beta, \mu)) \right)$$

O que é equivalente a:

$$\begin{aligned} \max_{\mu, \beta, R, a, \xi} \mathcal{L}(R, a, \xi, \beta, \mu) \\ \nabla_{R, a, \xi} \mathcal{L} = 0 \end{aligned}$$

Utilizando as condições de estabilidade e KKT,  $\mathcal{L}$  pode ser reduzido a uma função parametrizada somente por  $\beta$ , tal que, sob condições de estabilidade,  $\mathcal{L} = \hat{\mathcal{L}}(\beta)$ :

$$\begin{aligned} \hat{\mathcal{L}}(\beta) &= R^2 + C \sum_j \xi_j - \sum_j \beta_j (R^2 + \xi_j - \|\phi(x_j) - a\|^2) - \sum_j \xi_j \mu_j \\ &= R^2 \left( 1 - \sum_j \beta_j \right) + \sum_j \xi_j (C - \beta_j - \mu_j) + \sum_j \beta_j (\|\phi(x_j) - a\|^2) \\ &= \sum_j \beta_j (\|\phi(x_j) - a\|^2) \\ &= \sum_j \beta_j (\|\phi(x_j)\|^2 - 2a \cdot \phi(x_j) + \|a\|^2) \\ &= \sum_j \beta_j \left( \|\phi(x_j)\|^2 - 2 \sum_i \beta_i \phi(x_i) \cdot \phi(x_j) + \left\| \sum_i \beta_i \phi(x_i) \right\|^2 \right) \\ &= \sum_j \beta_j \phi(x_j) \phi(x_j) - \sum_{i,j} \beta_i \beta_j \phi(x_i) \phi(x_j) \end{aligned}$$

Deve destacar que para a função  $\phi(x)$  mapear os dados originais para um espaço com componentes de maior dimensão normalmente utiliza-se um truque de kernel. São comuns os casos em que a função  $\phi(x)$  por si só não consegue realizar o mapeamento dos dados para outro espaço. Desta forma, para que o mapeamento se torne possível normalmente opta-se por definir uma função kernel que consiste no produto interno de duas funções  $\phi(x)$  de fácil computação. Para o algoritmo SVC, a função kernel  $K(x_1, x_2)$  utilizada é o kernel gaussiano:

$$K(x_1, x_2) = \phi(x_1) \cdot \phi(x_2) = e^{-q\|x_1 - x_2\|^2}$$

Onde  $q$  é um hiperparâmetro da função gaussiana. Sendo assim, substituindo os produtos de  $\phi(x)$  por funções kernels gaussianos, obtém-se o problema dual de otimização para determinação da esfera:

$$\max_{\beta} \sum_j \beta_j K(x_j, x_j) - \sum_{i,j} \beta_i \beta_j K(x_i, x_j)$$

O qual consiste em um problema de programação quadrática que pode ser resolvido por diversos algoritmos de otimização presentes na literatura. Ao resolver este problema de otimização, valores de  $\beta_j$  para cada dado original são obtidos, assim o centro da esfera otimizada. Com estes valores em mão, então se torna possível definir um mapa de valores de distância das imagens dos dados no espaço auxiliar e o centro da esfera determinada  $R^2(X)$ :

$$R^2(X) = \|\phi(x_j) - a\|^2 = K(x, x) - 2 \sum_j \beta_j K(x_j, x) + \sum_{i,j} \beta_i \beta_j K(x_i, x_j)$$

Por meio de  $R^2(X)$  pode-se calcular o raio da esfera determinada:

$$R = \{R(x_i) \mid x_i \text{ é um vetor suporte}\}$$

E os contornos que englobam os pontos do espaço dos dados original:

$$\{x \mid R(x) = R\}$$

### 13.1.2 DETERMINAÇÃO DOS CLUSTERS

Mesmo após gerar os contornos dos clusters, o algoritmo não diferencia automaticamente quais dados pertencem aos mesmos clusters. Para que associação dos pontos aos diferentes clusters, o algoritmo SVC utiliza uma abordagem geométrica utilizando  $R(x)$  baseado na seguinte observação: dado um par de dados que pertencem a clusters diferentes, qualquer caminho de conexão entre as imagens destes dados no espaço auxiliar deve sair da esfera determinada em algum momento. Ou seja, qualquer caminho entre estes pontos conterá um segmento pontos  $y$  tal que  $R(y) > R$ . Sendo assim, para determinar se os pares de dados pertencem a um mesmo cluster, elabora-se uma matriz de adjacência  $A_{ij}$  dos diferentes pares de dados  $x_i$  e  $x_j$  tal que:

$$A_{ij} = \begin{cases} 1 & \text{se, para todos } y \text{ em um segmento conectando } x_i \text{ e } x_j, R(y) \leq R \\ 0, & \text{caso contrário} \end{cases}$$

Na prática, para checar a adjacência entre dois pontos, realiza-se a amostragem de pontos do segmento que interliga os pontos (por exemplo 10 pontos). Com os valores da matriz de adjacência calculados, os clusters podem ser definidos pelos componentes conectados de acordo com os grafos induzidos por  $A$ .

### 13.1.3 AJUSTE DOS HIPERPARÂMETROS

O formato dos contornos que definem os clusters são governados por dois hiperparâmetros:  $q$  (parâmetro de escala do kernel gaussiano) e  $C$  (constantes de margem suave). Quanto maior o valor de  $q$ , maior será o número de dados não interligados entre si, aumentando o número de clusters (Figura 84).

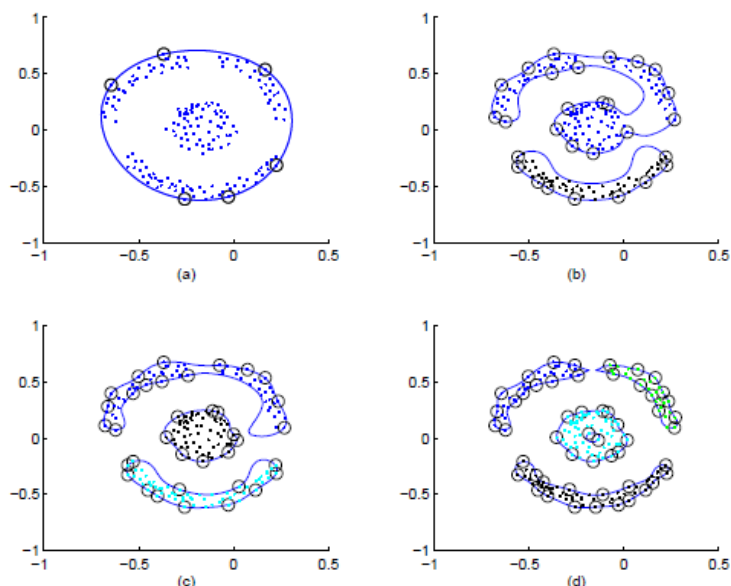


Figura 84 - Influência de  $q$  no número de clusters (a)  $q = 1$  (b)  $q = 20$  (c)  $q = 24$  (d)  $q = 48$

Já a constante  $C$  afeta o número de BSVs, o que afeta o número de outliers considerados. Por exemplo, caso  $C = 1$ , então o número de BSVs será nulo e nenhum dado será considerado, afetando o formato e o número de clusters (Figura 85).

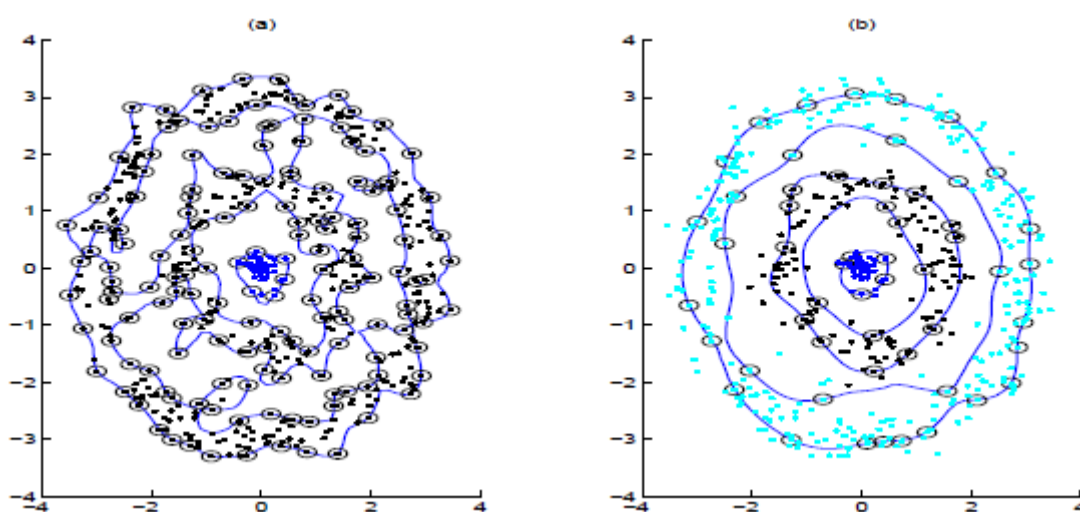


Figura 85 - Influência de  $C$  no formato e número de clusters (a)  $C = 1$  (b)  $C = 0.07$

Não existe uma regra definida para ajuste destes parâmetros, dependerá do contexto e dos dados analisados. Técnicas de grid para ajuste de parâmetros podem ser utilizados para determinação da melhor configuração de parâmetros.

## 13.2 APLICAÇÃO DO SVC

Para demonstrar a aplicação prática deste algoritmo dois casos de estudos são propostos. O primeiro consiste em um caso simples, no qual se utiliza um conjunto de dados brinquedo para demonstrar de forma visual a eficiência do algoritmo no agrupamento dos dados. Já o segundo caso consiste na análise de dados de operação de um processo da indústria de papel e celulose foi proposto para analisarmos a eficiência do algoritmo em um problema real.

### 13.2.1 APLCAÇÃO SVC: CASO SIMPLES

Para este caso de estudo simples é proposto a realização do agrupamento de dados de um conjunto de dados brinquedo obtido por meio da biblioteca scikit-learn para implementação de modelos de machine learning em python. A Figura 86 ilustra os dados analisados.

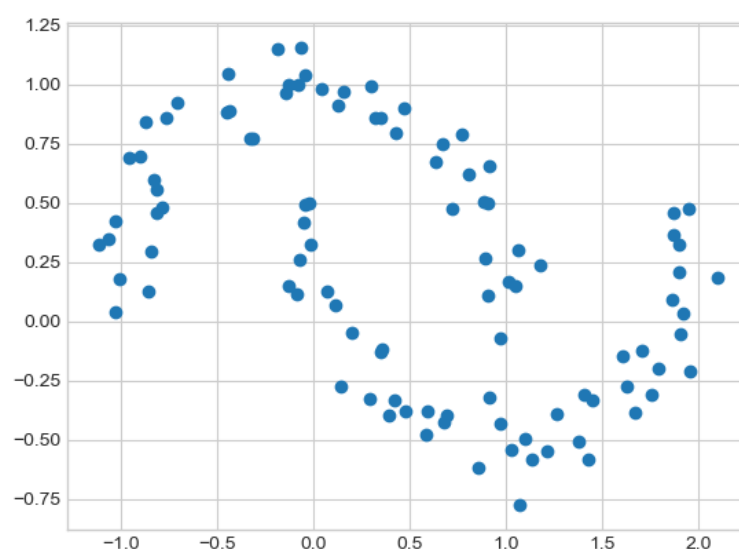


Figura 86 - Dados SVC (caso simple)

Os clusters obtidos pelo algoritmo SVC, com  $q = 6.5$  e  $C = 10$ , são ilustrados Figura 87. Tal como era esperados dois clusters bem definidos foram gerados. Os parâmetros  $q$  e  $C$  foram ajustados por tentativa e erro.

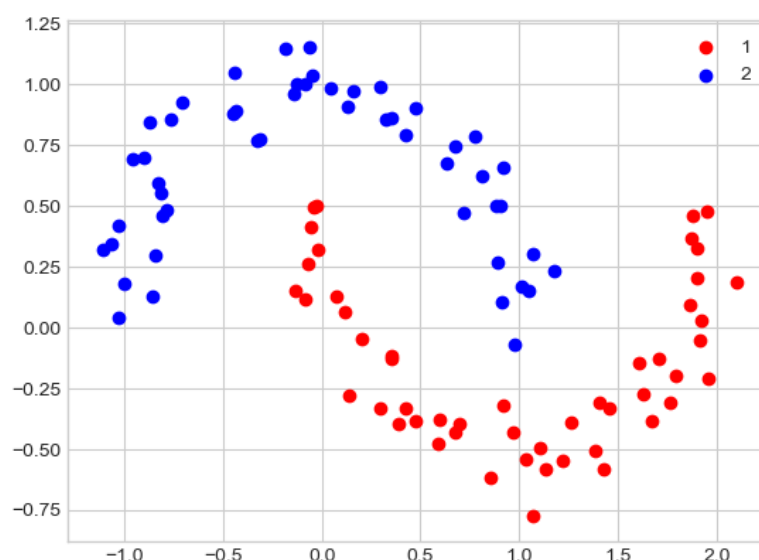


Figura 87 - Clusters gerados para  $q = 6.5$  e  $C = 10$

### 13.2.2 APLICAÇÃO DO SVC: CASO REAL

De forma a demonstrar a aplicabilidade do SVC para problemas com dados de situações reais, é proposto a análise de dados de operação de um processo Kraft presente em uma unidade de produção de celulose real. Os dados coletados consistem em dados de monitoramento de uma caldeira utilizada para queima de um licor proveniente da extração de celulose da madeira. A queima do licor na caldeira produz uma grande quantidade de gases quentes que são direcionados para superaquecedores utilizados para geração de vapor. Um grande problema na operação desta caldeira é fato dos gases, ao passarem pelos superaquecedores, gerarem depósitos sólidos na superfície do equipamento, os quais podem afetar a performance do processo. Sendo assim, com o intuito de elaborar um modelo de diagnóstico da presença de depósitos sólidos na caldeira, propõe-se o uso do SVC para o agrupamento de dados da caldeira em estado limpo e de dados da caldeira com depósito de sólidos. Para isso dados de operação em ambas as situações são fornecidos. Pelo fato de já se saber quais o estado da caldeira para cada amostra de dado, o problema elaborado pode ser considerado um caso de aprendizado semi-supervisionado, onde os dados são agrupados em duas classes já conhecidas. Para análise dos dados de monitoramento foi necessário realizar a seleção das variáveis mais relevantes para detecção da deposição de sólidos. Sendo assim, foram selecionadas 18 variáveis do processo. O algoritmo SVC foi aplicado para 200 amostras de dados de operação da caldeira, contendo 100 amostras da caldeira em estado limpo e 100 amostras de caldeira com depósito. Tal como era de esperar, a implementação do algoritmo resultou em dois clusters, um contendo só dados da caldeira em estado limpo e outro contendo só dados da caldeira com depósito de sólidos, o que demonstra a aplicabilidade do SVC para um caso industrial real. Com intuito de demonstrar o uso dos clusters gerados para um modelo de diagnóstico, propõe-se a classificação de novos dados de acordo com a proximidade aos clusters,

tal que o dado é considerado pertencente a classe do cluster mais próximo. O conjunto de 200 dados com estados já conhecidos foram classificados utilizando os clusters gerados, resultando em 96% de diagnósticos acertos e 4% de falsos alarmes de presença de deposição.

### 13.2.3 CÓDIGOS

```
import numpy as np
import matplotlib.pyplot as plt
import cvxpy as cvx
from tqdm import tqdm
plt.style.use('seaborn-whitegrid')

class SupportVectorClustering():

    def __init__(self):
        pass

    def dataset(self, xs, xs_labels):
        self.xs = xs # dataset
        self.xs_labels = xs_labels
        self.N = len(xs) # number de instâncias

    def parameters(self, p=0.1, q=1):
        self.q = q # parâmetro kernel width
        self.p = p # fração de bounded support vectors (BSVs)
        self.C = 1/(self.N*p) # constante de penalização (1/C >= 1)

    def kernel(self, x1, x2):
        return np.exp(-self.q*np.sum((x1-x2)**2, axis=-1)) # gaussian kernel

    def kernel_matrix(self):
        self.km = np.zeros((self.N, self.N))
        for i in range(self.N):
            for j in range(self.N):
                self.km[i,j] = self.kernel(self.xs[i], self.xs[j])

    # método de otimização
    def find_beta(self):
        beta = cvx.Variable(self.N) # vetor de N dimensões
        objective = cvx.Maximize(cvx.sum(beta) - cvx.quad_form(beta, self.km)) # função objetivo
        constraints = [0 <= beta, beta <= self.C, cvx.sum(beta) == 1] # restrições
        prob = cvx.Problem(objective, constraints) # definição do problema de otimização
        prob.solve() # solução do problema de otimização
        self.beta = beta.value # valores ótimos das variáveis beta
```



```

# cálculo do raio da hiperesfera
def r_func(self, x):
    return self.kernel(x, x) - 2*np.sum([self.beta[i]*self.kernel(self.xs[i],x) for i in range(self.N)]) + self.beta.T@self.km@self.beta
    # python > 3.5 @ matrix multiplication

# amostragem de segmentos entre dois pontos
def sample_segment(self,x1,x2,r,n=10):
    adj = True
    for i in range(n):
        x = x1 + (x2-x1)*i/(n+1)
        if self.r_func(x) > r:
            adj = False
            return adj
    return adj

# definição da matriz de adjacência
def cluster(self):
    print('Calculating adjacency matrix... \n')
    svb_tmp = np.array(self.beta < self.C)*np.array(self.beta > 10**-
8) # svb: 0 < beta < C
    self.svb = np.where(svb_tmp == True)[0] # índice support vectors
    bsvb_tmp = np.array(self.beta >= self.C) # bsvb: beta == C ??
    self.bsvb = np.where(bsvb_tmp == True)[0] # índice bounded support v
ectors
    self.r = np.mean([self.r_func(self.xs[i]) for i in self.svb[:5]]) #
why 5??
    self.adj = np.zeros((self.N, self.N)) # matriz adjacência
    # checar adjacência entre pontos
    for i in tqdm(range(self.N)):
        if i not in self.bsvb:
            for j in range(i, self.N):
                if j not in self.bsvb:
                    self.adj[i,j] = self.adj[j,i] = self.sample_segment(
self.xs[i],self.xs[j],self.r)

# definição dos clusters
def return_clusters(self):
    ids = list(range(self.N))
    self.clusters = {}
    num_clusters = 0
    while ids:
        num_clusters += 1
        self.clusters[num_clusters] = []
        curr_id = ids.pop(0)
        queue = [curr_id]
        while queue:
            cid = queue.pop(0)

```

```

        for i in ids:
            if self.adj[i,cid]:
                queue.append(i)
                ids.remove(i)
            self.clusters[num_clusters].append(cid)
    print('\n')
    print(f'The number of clusters is {num_clusters}')

def results(self):
    clusters_results = np.zeros((len(self.clusters.keys()), 2))
    for i in self.clusters.keys():
        normal = 0
        fault = 0
        for j in self.clusters[i]:
            if self.xs_labels[j] == 0:
                normal += 1
            else:
                fault += 1
        clusters_results[i-1][0] = normal
        clusters_results[i-1][1] = fault
    print([0, 1])
    print(clusters_results)
    print(len(clusters_results))

# centróides dos clusters
def clusters_centroids(self):
    self.centroids = dict()
    for key, values in self.clusters.items():
        sum_cluster_data = np.zeros(len(self.xs[0]))
        for j in values:
            sum_cluster_data += self.xs[j]
        centroid = sum_cluster_data / len(values)
        self.centroids[key] = centroid
    print(self.centroids)

# teste do modelo
def test_clustering(self, xs_test, xs_labels_test):
    self.xs_test = xs_test
    self.xs_labels_test = xs_labels_test
    clusters_similarity = np.zeros((len(xs_test),len(self.centroids.keys
    ())))
    for i, x in enumerate(xs_test):
        for key, centroid in self.centroids.items():
            clusters_similarity[i,key-1] = np.linalg.norm(x-centroid, 2)
    cluster_assignment = np.argmin(clusters_similarity, axis = 1)
    print(cluster_assignment)

    if len(self.centroids.keys()) == 2:
        false_alarm = 0

```

```

        warn_miss = 0
        for i in range(len(self.xs_labels_test)):
            if (cluster_assignment[i] != self.xs_labels_test[i]):
                if ((cluster_assignment[i] == 1) and (self.xs_labels_test[i] == 0)):
                    false_alarm += 1
                else:
                    warn_miss += 1
            print(f'False Alarm {false_alarm*100/len(self.xs_labels_test)}%')
        print(f'Missed Warning {warn_miss*100/len(self.xs_labels_test)}%')
        print(f'Right Diagnostic {(len(self.xs_labels_test) - false_alarm - warn_miss)*100/len(self.xs_labels_test)}%')
    else:
        error = 0
        for i in range(len(self.xs_labels_test)):
            if cluster_assignment[i] != self.xs_labels_test[i]:
                error += 1
            print(f'Right Diagnostic {(len(self.xs_labels_test) - error)*100/len(self.xs_labels_test)}%')
            print(f'Wrong Diagnostic {(error)*100/len(self.xs_labels_test)}%')

    def plot_clusters(self):
        colors = {1:'r',2:'b',3:'g',4:'c',5:'m',6:'y',7:'k',8:'b',9:'c'}
        for num_cluster, samples in self.clusters.items():
            cluster = np.empty((len(samples), 2))
            for idx, sample in enumerate(samples):
                cluster[idx] = self.xs[sample]
            plt.scatter(cluster[:,0], cluster[:,1], c = colors[num_cluster],
label = num_cluster)
            plt.legend()
            plt.show()

```

Figura 88 - Código em Python para implementação do SVC

```

import numpy as np
import matplotlib.pyplot as plt
import sklearn.datasets
import time
plt.style.use('seaborn-whitegrid')

from support_vector_clustering import SupportVectorClustering
# importação dos dados
def define_data(REBUILD_DATA = False, N_SAMPLES = 50):
    if REBUILD_DATA == True:
        ms = sklearn.datasets.make_moons(n_samples=N_SAMPLES,noise=0.1)[0]
        np.save('simple_dataset.npy', ms)

```

```

X = np.load('simple_dataset.npy')
return X

if __name__ == '__main__':
    # define database
    X = define_data(REBUILD_DATA = False, N_SAMPLES=50)
    # iniciação do algoritmo
    start_time = time.time()
    svc = SupportVectorClustering()
    svc.dataset(X) # database
    svc.parameters(p=0.002, q=6.5) # define parâmetros
    svc.kernel_matrix() # cálculo matriz kernel
    svc.find_beta() # solução problema de otimização
    svc.cluster() # cálculo matriz adjacência
    svc.return_clusters() # define clusters
    print('\n')
    print(f'Processing Time: {time.time() - start_time} seconds')
    svc.plot_clusters() # plot

```

Figura 89 - Código em Python para solução do caso simples

```

import os
import numpy as np
import sklearn.datasets

from support_vector_clustering import SupportVectorClustering

if __name__ == '__main__':

    # importação dos dados
    f = open('caldeira_dataset.csv')
    data = f.read()
    f.close()
    # conversão dos dados para matriz
    lines = data.split('\n')
    num_rows = len(lines)
    num_cols = len(lines[0].split(';'))
    float_data = np.zeros((num_rows, num_cols-2))
    labels = np.zeros(len(float_data))
    for i, line in enumerate(lines):
        values = [float(value) for value in line.split(';')[1:num_cols-1]]
        float_data[i,:] = values
        labels[i] = float(line[-1])

    # dados de treinamento
    normal_data = float_data[:100]
    fault_data = float_data[2882:2982]
    training_data = np.concatenate((normal_data, fault_data), axis = 0)

```

```

# rótulos dos dados de treinamento
normal_labels = labels[:100]
fault_labels = labels[2882:2982]
training_labels = np.concatenate((normal_labels, fault_labels), axis = 0
)

# dados de teste
test_normal_data = float_data[1440:1540]
test_fault_data = float_data[4322:4422]
test_data = np.concatenate((test_normal_data, test_fault_data), axis = 0
)

# rótulos dos dados de teste
test_normal_labels = labels[1440:1540]
test_fault_labels = labels[4322:4422]
test_data_labels = np.concatenate((test_normal_labels, test_fault_labels
), axis = 0)
print(test_data_labels)

# normalização dos dados
mean = training_data.mean(axis=0)
std = training_data.std(axis=0)
training_data -= mean
training_data /= std
test_data -= mean
test_data /= std

# algoritmo SVC
svc = SupportVectorClustering()
svc.dataset(training_data, training_labels) # database
svc.parameters(p=0.005, q=0.24) # parâmetros # p = 0.005 q = 0.24 funcio
nou
svc.kernel_matrix() # matriz kernel
svc.find_beta() # solução problema de otimização
svc.cluster() # matriz de adjacência
svc.return_clusters() # define clusters
svc.results()
svc.clusters_centroids()
svc.test_clustering(test_data, test_data_labels)

```

Figura 90 - Código em Python para solução do caso industrial real

### 13.3 BIBLIOGRAFIA

BEN-HUR, Asa et al. Support vector clustering. Journal of machine learning research, v. 2, n. Dec, p. 125-137, 2001.