UNIVERSIDADE AUTÓNOMA DE LISBOA
LUÍS DE CAMÕES

DEPARTAMENTO DE CIÊNCIAS E TECNOLOGIAS
LICENCIATURA EM INFORMÁTICA DE GESTÃO

(INTELIGÊNCIA ARTIFICIAL – TRABALHO PRÁTICO)

João Cajado: 30007671
João Falcão: 30007061
Tiago Gamboa: 30007315
Pedro Sarmento: 30007313

Janeiro de 2023
Lisboa

(Página em Branco)

Índice

1 Introdução

Este trabalho foi realizado no âmbito da cadeira de Inteligência Artificial do terceiro ano da licenciatura de informática de gestão na Universidade Autónoma de Lisboa.

Com a realização deste trabalho pretende-se demonstrar a consolidação dos conhecimentos lecionados nas aulas, nomeadamente a implementação dos algoritmos Naive Bayes, Perceptrão e AC-3.

Este trabalho prático proposto pelo professor apresenta dois exercícios distintos, cujo objetivo é implementar e solucioná-los através dos algoritmos referidos. A primeira parte do trabalho é composta pelo *Filtro de Spam*, com base no algoritmo de Naive Bayes e com base no algoritmo do Perceptrão. A segunda parte do trabalho pretende-se resolver um *Sudoku*, com base no algoritmo AC-3. Todos os algoritmos foram implementados utilizando a linguagem de programação Python 3.9.13.

O relatório do trabalho prático irá conter a descrição detalhada dos enunciados de cada exercício e a teoria dos mesmos, de modo a ser explicado com todo o detalhe.

2 Filtro Spam

Enunciado

Pretende-se implementar dois filtros de spam para documentos de texto (sms ou emails) com base no algoritmo de Naive Bayes e no algoritmo do Perceptrão respetivamente.

1. Deverá utilizar a linguagem de programação Python ou Julia.
2. Poderá utilizar o conjunto de sms previamente classificados como spam ou ham disponíveis no e-learning (deverá passar esses dados por um antivírus para garantir que não coloca vírus na sua máquina já que esses dados foram retirados de repositórios públicos). Em alternativa poderá usar outros dados semelhantes de sms ou emails que facilmente encontrará em diversos repositórios de dados. Deverá utilizar um conjunto de treino, um conjunto de teste e um conjunto de validação (70/15/15).
3. Pode eventualmente achar útil pré-processar os documentos a fim de retirar conteúdo não relevante (por exemplo, carateres ou strings não informativas).
4. Utilizará para o algoritmo do Perceptrão a representação de cada documento no modelo de *bag of words*. Nesse modelo cada documento é representado por um vetor de frequências absolutas com base num léxico previamente definido.
5. Deverá apresentar métricas de perfomance dos filtros.

Fundamentos Teóricos

O algoritmo de Naive Bayes é um classificador probabilístico baseado no teorema de Bayes. É chamado de "naive" porque assume que todas as características são independentes entre si, o que nem sempre é verdade. No entanto, mesmo com essa suposição simplificadora, ele é muito eficaz em muitas tarefas de classificação.

Este algoritmo é amplamente utilizado em sistemas de filtragem de spam, uma vez que é capaz de lidar com muitas características diferentes e é relativamente rápido para treinar. A principal ideia por trás do algoritmo é a seguinte, dado um conjunto de dados de treino com rótulos de classe conhecidos (por exemplo, *spam* ou *ham*), o algoritmo de Naive Bayes aprende a associar determinadas características dos dados com determinadas classes. Uma vez treinado, o algoritmo de Naive Bayes pode usar o conhecimento adquirido para classificar novos exemplos de dados. Ele calcula a probabilidade de um novo exemplo pertencer a cada uma das classes e atribui o rótulo da classe com maior probabilidade.

Visto que o algoritmo de Naive Bayes assume que todas as características são independentes, pode não ser tão preciso quanto alguns algoritmos mais avançados, no entanto é um bom ponto de partida para muitos problemas de classificação.

Em relação ao Perceptrão, é um algoritmo utilizado para classificação binária. É um modelo linear e é chamado de Perceptrão porque foi originalmente desenvolvido como uma simulação do funcionamento do nervo sensorial em animais.

O Perceptrão é treinado usando um conjunto de dados de treino rotulados, onde cada exemplo é uma combinação de características de entrada e um rótulo de classe correspondente (por exemplo, *spam* ou *ham*). O objetivo deste treino é ajustar os pesos das características de entrada de maneira a minimizar o número de erros de classificação, sendo feito através de um processo iterativo, onde cada exemplo é avaliado e os pesos são ajustados de acordo com o erro de classificação. Após ser treinado, o algoritmo do Perceptrão pode ser utilizado para classificar novos exemplos de dados.

Este algoritmo apresenta algumas limitações, uma vez que só pode ser utilizado para classificação binária e assume que os dados são linearmente separáveis, o que nem sempre é verdade, no entanto, mesmo com essas limitações, o algoritmo do Perceptrão é amplamente utilizado em muitas aplicações, incluído a filtragem de spam.

Implementação

Como foi referido, ambos os algoritmos precisam de um conjunto de emails devidamente rotulados com spam ou ham para que consiga realizar a sua classificação. Nesta implementação utilizámos o ficheiro CSV fornecido pelo professor.

O primeiro passo desta implementação foi criar um ficheiro Python com o nome **csv_reader.py** de modo a implementar uma função que se encarrega de ler todo o ficheiro que contém todos os emails.

```
import csv
from random import shuffle

blacklist = ["!", "?", ",", "-", ".", "/", ";", "_", "{", "}", "%",
":", "(", ")", "<", ">"]  # filtragem dos caracteres que não sao
palavras

def csv_reader():
    treino = [] # -> 70%
    teste = [] # -> 15%
    validacao = [] # -> 15%
    with open("spam.csv", "r") as ficheiro:
        reader = csv.reader(ficheiro)
        emails_todos = []
        for email in reader:
            for word in email[1].lower().split(" "):
                word = word.lower()
                for char in blacklist:
                    word.replace(char, "")
            emails_todos.append(email)
        emails_todos.remove(emails_todos[0])
        shuffle(emails_todos)
        for i in range(len(emails_todos)):
            if i < round(len(emails_todos) * 0.7):
                treino.append(emails_todos[i])
            elif i < round(len(emails_todos) * 0.7 + len(emails_todos)
* 0.15):
                teste.append(emails_todos[i])
            else:
                validacao.append(emails_todos[i])
        return treino, teste, validacao
```

Como referido no enunciado, era necessário efetuar uma divisão em três partes (70/15/15) e optámos por criar quatro arrays inicialmente vazios: *treino*, *teste*, *validacao* e *emails_todos*. Após a criação dos mesmos, utilizámos a biblioteca CSV para abrir o nosso ficheiro com todos os emails e, de seguida, demos início a um *for loop* de modo a inserir todos os emails do ficheiro csv no nosso novo array *emails_todos* para prosseguirmos para a divisão e colocar os emails nos outros arrays (*treino*, *teste* e *validacao*).

De modo a efetuar um tratamento de dados com a finalidade de remover os caracteres que não são palavras e que poderiam interferir na classificação do algoritmo, foi necessário criar um array *blacklist* com todos os caracteres que o grupo achou que eram indesejados e assim ficar com um ficheiro mais “limpo.

Após adicionarmos todos os emails ao array *emails_todos* já devidamente tratados, outro *for loop* foi iniciado de modo a repartir os emails de acordo com as percentagens pedidas.

Implementação do algoritmo Naive Bayes

Para a implementação do algoritmo Naive Bayes, foi criado um ficheiro Python **NaiveBayes.py** onde o algoritmo é implementado e conta com as seguintes funções: **train()**, **classify()**, **melhor_c()** e **classify_list()**.

```
import math

contagem_de_palavras = {}

w_spam = 0 # nº de palavras spam
w_ham = 0 # nº de palavras ham
m_spam = 0 # nº de mails spam
m_ham = 0 # nº de mails ham

c = 0.009
```

Este ficheiro inicia com a importação da biblioteca *math* que mais adiante irá ser utilizada para a fórmula da inicialização do threshold de rejeição. De seguida, decidimos não seguir a sugestão do professor que era criar uma *bag of words*, em vez disso optámos pela criação de um dicionário *contagem_de_palavras* que serve para guardar todas as palavras encontradas do ficheiro com a frequência das mesmas associada. A criação das variáveis iniciadas a zero, é necessária para efetuar a contagem das palavras spam/ham e do número de emails *spam*/*ham*. Inicializamos também a variável *c* que lhe é atribuído um valor definido pelo grupo e que mais para a frente pode ser sujeita a alterações.

A primeira função a ser implementada é a função **train()**. Esta função fica encarregue de calcular as frequências absolutas e relativas dos emails.

```
def train(X):
    global w_ham, w_spam, m_ham, m_spam
    # Cálculo das frequencias absolutas
    for mail in X:
        if mail[0] == "spam":
            m_spam +=1
            for palavra in mail[1].lower().split(" "):
                if palavra.lower() not in contagem_de_palavras:
                    contagem_de_palavras[palavra] = [2, 1, 0, 0]
                else:
                    contagem_de_palavras[palavra][0] += 1
                w_spam += 1

        elif mail[0] == "ham":
            m_ham += 1
            for palavra in mail[1].lower().split(" "):
                if palavra.lower() not in contagem_de_palavras:
                    contagem_de_palavras[palavra] = [1, 2, 0, 0]
                else:
                    contagem_de_palavras[palavra][1] += 1
                w_ham += 1

    # Cálculo das frequencias relativas
    for palavra in contagem_de_palavras:
        contagem_de_palavras[palavra][2] = contagem_de_palavras[pala-
vra][0] / w_spam
        contagem_de_palavras[palavra][3] = contagem_de_palavras[pala-
vra][1] / w_ham
```

A função inicia com um *for loop* em que se itera sobre a lista X de modo a verificar se o email é *spam* ou *ham*. É efetuada a distinção através das condições *if* e *elif*, isto é, caso na primeira posição se encontre a *string* "spam", então a variável *m_spam* é incrementada em 1. Se na primeira posição estiver a *string* "ham", a variável *m_ham* é incrementada em 1. Após o número de emails *spam* ser incrementado, é iniciado outro *for* loop onde o email é dividido num conjunto de palavras independentes, todas convertidas para letras minúsculas e separadas por espaço.

No caso da palavra que se encontra em iteração não tiver sido adicionada anteriormente ao dicionário *contagem_de_palavras*, então a mesma é adicionada ao mesmo, sendo a *key* a palavra e o *value* um array dividido em quatro informações: frequência absoluta para spam, frequência absoluta para ham, frequência relativa para spam e frequência relativa para ham, seguindo esta ordem descrita. Deste modo, os valores são inseridos respetivamente da seguinte

forma para emails *spam*: [2,1,0,0]. A frequência absoluta para spam apresenta o valor ‘2’, pois as palavras devem ser inicializadas com ‘1’, de modo que o logaritmo de 0 não seja utilizado ($log_0 \rightarrow$ indefinido) e como a palavra é encontrada ocorre o incremento de ‘1’, originando o valor ‘2’. Para o valor ‘1’ das frequências absolutas para ham, a explicação é a mesma do logaritmo de 0 não ser definido, tendo de ser necessário usar o valor ‘1’ e como não é *ham*, não se acrescenta nenhum valor. Caso a palavra já se encontre em *contagem_de_palavras*, é apenas adicionado ‘1’ ao seu valor de frequências absolutas para spam. Após isto, é incrementado em um o valor da variável *w_spam*, que contibiliza o número de palavras spam.

No caso de emails *ham*, o raciocínio utilizado foi o mesmo, apresentando uma única diferença na incrementação, neste caso é a frequência absoluta de *ham* que é incrementada.

De modo a introduzir a frequência relativa para cada palavra no dicionário, é iniciado um *for loop* que percorre todo o dicionário *contagem_de_palavras* e através da divisão da frequência absoluta de cada palavra pelo número total de palavras *spam*/*ham* adiciona o valor da frequência relativa na posição correta caso a palavra seja *spam* ou *ham*.

A segunda função, denominada **classify()**, classifica email a email (um de cada vez).

```
def classify(email, c = c):
    threshold= -(math.log(c) + math.log(m_ham) - math.log(m_spam))

    for palavra in email.lower().split(" "):
        if palavra.lower() in contagem_de_palavras:
            threshold+= (math.log(contagem_de_palavras[palavra][0]) -
math.log(contagem_de_palavras[palavra][1]))

    return "spam" if threshold> 0 else "ham"
```

A classificação é efetuada através da fórmula da inicialização do threshold de rejeição ($treshold = -(math.\log(c) + math.\log(m\_ham) - math.\log(m\_spam)$).

Esta fórmula é aplicada a cada palavra do email que está a ser iterado. É retornado “spam” caso o threshold seja maior que zero, caso contrário é retornado “ham”.

A terceira função implementada, **melhor_c()** tem como objetivo encontrar o melhor valor de *c* para o nosso algoritmo.

```
def procurar_o_melhor_c():
    pass
```

A maior dificuldade encontrada pelo grupo na implementação do algoritmo de Naive Bayes foi implementar a função que ao longo do treino do algoritmo iria descobrir qual *c* seria o mais eficiente para o algoritmo, ou seja, cada vez que corrêssemos o algoritmo, o mesmo iria descobrir o melhor *c* e assim daria valores diferentes e isso teria impacto na perfomance. Visto que não conseguimos implementar esta função, o algoritmo corre sempre com o mesmo *c* que lhe atribuímos um valor de '0.009'.

A última função definida neste ficheiro foi **classify_list()** que foi implementada com o propósito de classificar todos os emails do conjunto de validação recorrendo às informações que foram adquiridas no decorrer da execução do algoritmo de Naive Bayes. Esta função tem como argumento uma lista de validação (*lista_validacao*) e vai iterar sobre todos os emails, um a um, correndo a função **classify**(email) para cada email.

```
def classify_list(lista_validacao):
    num_guesses = 0
    num_correct_guesses = 0
    num_incorrect_guesses = 0
    num_spam = 0
    num_ham = 0
    verdadeiro_positivo = 0
    verdadeiro_negativo = 0
    falso_positivo = 0
    falso_negativo = 0

    for email in lista_validacao:
        classificador = classify(email[1])
        if classificador == email[0]:
            num_correct_guesses += 1
            if classificador == 'spam':
                verdadeiro_positivo += 1
            if classificador == 'ham':
                verdadeiro_negativo += 1
        else:
            num_incorrect_guesses += 1
            if classificador == 'spam':
                falso_positivo += 1
            if classificador == 'ham':
                falso_negativo += 1

        if email[0] == 'spam':
            num_spam += 1
        if email[0] == 'ham':
            num_ham += 1
        num_guesses += 1
```

É iniciado um conjunto de variáveis a zero que serviram depois para a avaliação da classificação efetuada como também para as métricas de perfomance, de modo a chegarmos a conclusões sobre a eficiência do nosso algoritmo Naive Bayes.

Após ocorrer a classificação do email, a função compara se o resultado da função **classify(**email**)** é igual ao rótulo do email que está a iterar. Caso sejam iguais, concluímos que a classificação foi efetuada corretamente, incrementando assim em um a variável *num_correct_guesses*, caso contrário, incrementa a variável *num_incorrect_guesses*. Também é feita a verificação dos resultados para a distinção se o mesmo é *spam* ou *ham*, de modo a contabilizar o número acertos de emails *spam* e *ham* separadamente.

De modo a disponibilizar as métricas de classificação, utilizámos os seguintes prints:

```
print("--->Algoritmo Naive Bayes 70/15/15 <---")
    print("O algoritmo percorreu:", num_guesses, "emails!")
    print(" ")
    print("Número de emails spam avaliados:", num_spam)
    print("-> Avaliou corretamente", verdadeiro_positivo)
    print("-> Avaliou incorretamente:", num_spam - verdadeiro_positivo)
    print(" ")
    print("Número de emails ham avaliados:", num_ham)
    print("-> Avaliou corretamente", verdadeiro_negativo)
    print("-> Avaliou incorretamente:", num_ham - verdadeiro_negativo)
    print("")
    print("O algoritmo obteve uma taxa de sucesso igual a:", ((num_cor-
rect_guesses / num_guesses) * 100), "%")
    print("O algoritmo obteve uma taxa de insucesso igual a:",  (100 -
(num_correct_guesses / num_guesses) * 100), "%")
    print(" ")

    print("Métricas de Classificação:")
    print("    Accuracy:", (num_correct_guesses / num_guesses))
    print("    Error rate:", (num_incorrect_guesses / num_guesses))
    print("    Sensivity:", (verdadeiro_positivo / (verdadeiro_positivo
+ falso_negativo)))
    print("    Specificity:", (verdadeiro_negativo / (verdadeiro_nega-
tivo + falso_positivo)))
    print("    Precision:", (verdadeiro_positivo / (verdadeiro_positivo
+ falso_positivo)))
    print("    Recall:", (verdadeiro_positivo / (verdadeiro_positivo +
verdadeiro_negativo)))
    print(" ")
```

Implementação do algoritmo do Perceptrão

Para a implementação do algoritmo do Perceptrão, foi criado um ficheiro Python **perceptrao.py** onde o algoritmo é implementado e conta com as seguintes funções: **perceptron()**, **classify()**, **perceptron_classify_list()**, **convert_tag()**, **email_to_word_counter()**, **create_teta()**.

A primeira função a ser implementada foi a **perceptron()**. Esta função recebe dois argumentos, *train_data* e *T*, representado o conjunto de emails de treino e o número de vezes que o algoritmo percorre o conjunto de emails de treino, respetivamente.

```
def convert_tag(lista_emails):
    for email in lista_emails:
        if email[0] == "spam":
            email[0] = -1
        if email[0] == "ham":
            email[0] = 1
    return lista_emails

def email_to_word_counter (lista_emails):
    for email in lista_emails:
        email_word_counter = {}
        for word in email[1].split(" "):
            if word not in email_word_counter:
                email_word_counter[word] = 1
            else:
                email_word_counter[word] += 1
        email[1] = email_word_counter
    return lista_emails

def create_teta(emails_dictionary):
    teta = {}
    for email in emails_dictionary:
        for word, count in email[1].items():
            if word not in teta:
                teta[word] = 0
    return teta
```

Com o auxílio das funções **convert_tag()**, **email_to_word_counter() e create_teta()**, inicializámos a função **perceptron()** de modo a manipular os emails. A primeira função (**convert_tag()**) tem a finalidade de converter o rótulo de “spam” para ‘-1’ ou “ham” para ‘1’. A segunda função (**email_to_word_counter()**) tem a finalidade de percorrer todos os emails e fazer a contagem das palavras com o objetivo de transformar o conjunto de palavras do email,

para um dicionário chamado *email_word_counter*, onde a *key* é a palavra e o *value* corresponde à frequência absoluta do número de vezes que a palavra se repete. A função **create_teta()** é responsável por criar um dicionário *teta*, em que a *key* são as palavras de todos os emails do conjunto de treino e o *value* corresponde a '0'.

```
def perceptron(train_data, T):
    train_data = convert_tag(train_data)
    train_data = email_to_word_counter(train_data)

    teta = create_teta(train_data)
    teta_zero = 0
    for _ in range(T):
        for email in train_data:
            if email[0] * classify(teta, email[1], teta_zero) <= 0:
                for word, count in email[1].items():
                    if word in teta:
                        teta[word] += email[0] * count
                teta_zero += email[0]

    return teta, teta_zero
```

Voltando à função **perceptron()**, após o que foi referido anteriormente, é criada uma variável *teta_zero* que é iniciada a zero. Após a inicialização desta variável, iniciamos um *for loop* que itera o número de vezes de T, garantido assim que o número de iterações não é desrespeitado.

É necessário encontrar um equilíbrio no que toca à atribuição do valor de *T*, uma vez que, quanto maior for *T*, mais vezes o classificador é treinado e, portanto, tende a ter um desempenho melhor. No entanto, um número muito grande pode levar ao "overfitting", ou seja, o classificador pode ajustar-se muito bem a esse conjunto de treino, mas não conseguir generalizar para novos dados e obter um desempenho baixo, diminuindo assim a sua taxa de acerto. O mesmo se verifica se o valor de *T* for muito pequeno, o classificador pode não ser treinado adequadamente e obter um mau desempenho e, consequentemente, apresentar uma taxa de acerto muito baixa. Ou seja, é preciso ter cuidado no que toca à atribuição de valor a *T*.

Após o primeiro *for loop*, é inicializado outro *for loop*, que itera email a email do conjunto de treino e realiza o cálculo de classificação do email através da fórmula presente no pseudocódigo fornecido do algoritmo do Perceptrão.

É na condição *if* que esta fórmula é aplicada (`if email[0] * classify(teta, email[1], teta_zero) <= 0:`) e a mesma verifica se o rótulo do email é igual à classificação do email, visto que se forem diferentes o resultado será inferior a zero. Caso os valores sejam diferentes conclui-se

que a classificação está incorreta e é necessário ocorrer atualização dos valores. A classificação é realizada na função **classify(**teta, email[1], teta_zero**)**. Se o resultado da multiplicação for menor ou igual a zero, entramos no *for loop* que faz a iteração sobre os items do dicionário do email e faz a verificação, através da condição *if*, se a palavra iterada se encontra em teta e procede para a alteração do seu valor. No final, ocorre a incrementação do valor de teta_zero, que adiciona o valor do rótulo daquele email.

De seguida, foi implementada a função **classify()**. Esta função é composta por três parâmetros: *teta*, *email_dic* e *teta_zero*. O parâmetro *teta* representa o dicionário *teta*, o parâmetro *email_dic* representa um email no formato de dicionário, sendo a *key* uma palavra presente no email e o *value* corresponde à frequência absoluta da palavra (quantas vezes a mesma palavra se repete no email). O parâmetro *teta_zero* representa o teta_zero do Perceptrão. O principal objetivo da implementação desta função é avaliar um email e é importante na função **perceptron()**, para ocorrer a avaliação de um email específico e caso seja necessário proceder à atualização dos valores de *teta* e de *teta_zero*.

```
def classify(teta, email_dic, teta_zero):
    total_sum = 0
    for word, count in email_dic.items():
        if word in teta:
            total_sum += teta[word] * count
    return total_sum + teta_zero
```

Iniciamos esta função ao criar uma variável *total_sum* iniciada a zero. Esta variável representa o produto escalar/interno, que é importante para calcular a saída do classificador. Nesta função o produto interno é calculado da seguinte forma, para cada palavra (*word*) no dicionário de palavras (*email_dic*), o peso da palavra (*teta[word]*) é multiplicado pela contagem da palavra (*count*) e o resultado é adicionado à variável *total_sum*. No final, *teta*_zero é adicionado a *total*_sum. O resultado de *total_sum* é a saída do classificador. Se a saída for positiva, o classificador classifica o email como ham, caso seja negativa, é classificado como spam.

Por fim, é implementada a função **perceptron_classify_list()**. Possui três argumentos, sendo eles, *validation_list*, *teta*, *teta_zero*. Tal como na função **perceptron()** iniciamos esta função com a formatação adequada dos dados recorrendo às funções **convert_tag()** e **email_to_word_counter()**.

```
def classify_list(lista_validacao, teta, teta_zero):
    lista_validacao = convert_tag(lista_validacao)
    lista_validacao = email_to_word_counter(lista_validacao)

    num_guesses = 0
    num_correct_guesses = 0
    num_incorrect_guesses = 0
    num_spam = 0
    num_ham = 0
    verdadeiro_positivo = 0
    verdadeiro_negativo = 0
    falso_positivo = 0
    falso_negativo = 0

    for email in lista_validacao:
        classificador = classify(teta, email[1], teta_zero)
        classificador = math.copysign(1, classificador)
        if math.copysign(1, classificador) == email[0]:
            num_correct_guesses += 1
            if classificador == -1:
                verdadeiro_positivo += 1
            elif classificador == 1:
                verdadeiro_negativo += 1
        else:
            num_incorrect_guesses += 1
            if classificador == -1:
                falso_positivo += 1
            elif classificador == 1:
                falso_negativo += 1
        if email[0] == -1:
            num_spam += 1
        if email[0] == 1:
            num_ham += 1
        num_guesses += 1
```

De seguida, procedemos à criação de variáveis, todas ela inicializadas a zero, que tal como no algoritmo de Naive Bayes, vão ajudar a avaliar as classificações efetuadas e a apresentar as métricas de classificação do nosso algoritmo do Perceptrão, de modo a chegarmos a uma conclusão em relação à sua perfomance.

Após a criação das variáveis, iniciamos um *for loop* que itera os emails, um a um, da nossa lista de validação (*validation_list*) que é depois classificado recorrendo à nossa função **classify()** e o valor desta classificação é guardado em *classifier*. Foi necessário recorrer à função **copysign()** da biblioteca math, de modo a retornar apenas '-1' ou '1'. Após guardarmos o valor

em *classifier*, iniciamos uma condição *if* que verifica se o rótulo do email que se encontra em iteração é igual ao valor da variável *classifier*. Caso seja igual, isto significa que o algoritmo fez uma avaliação correta e, por isso, procedemos à incrementação da variável *num_correct_guesses*. Caso contrário, a variável incrementada é *num_incorrect_guesses*, uma vez que o algoritmo fez uma avaliação incorreta. É necessário também verificar se o email que está a ser avaliado é *spam* ou *ham*, de modo a incrementar *true_positive*/*true_positive* ou *false_positive*/*false_negative*.

Por fim, fazemos uma verificação recorrendo a duas condições *if* para verificar o rótulo do email de modo a incrementar a variável *numb_spam* ou *numb_ham*.

De modo a disponibilizar as métricas de classificação, utilizámos os seguintes prints:

```
 print("---> Algoritmo do Perceptrão 70/15/15 <---")
    print("O algoritmo percorreu:", num_guesses, "emails!")
    print(" ")
    print("Número de emails spam avaliados:", num_spam)
    print("-> Avaliou corretamente", verdadeiro_positivo)
    print("-> Avaliou incorretamente:", num_spam - verdadeiro_positivo)
    print(" ")
    print("Número de emails ham avaliados:", num_ham)
    print("-> Avaliou corretamente", verdadeiro_negativo)
    print("-> Avaliou incorretamente:", num_ham - verdadeiro_negativo)
    print("")
    print("O algoritmo obteve uma taxa de sucesso igual a:", ((num_cor-
rect_guesses / num_guesses) * 100), "%")
    print("O algoritmo obteve uma taxa de insucesso igual a:",  (100 -
(num_correct_guesses / num_guesses) * 100), "%")
    print(" ")

    print("Métricas de Classificação:")
    print("    Accuracy:", (num_correct_guesses / num_guesses))
    print("    Error rate:", (num_incorrect_guesses / num_guesses))
    print("    Sensivity:", (verdadeiro_positivo / (verdadeiro_positivo
+ falso_negativo)))
    print("    Specificity:", (verdadeiro_negativo / (verdadeiro_nega-
tivo + falso_positivo)))
    print("    Precision:", (verdadeiro_positivo / (verdadeiro_positivo
+ falso_positivo)))
    print("    Recall:", (verdadeiro_positivo / (verdadeiro_positivo +
verdadeiro_negativo)))
    print(" ")
```

3 Sudoku

Enunciado

Pretende-se resolver o problema de sudoku que se apresenta na Fig. 1 a). Este problema em concreto pode resolver-se implementando apenas a consistência de arco e a solução (que é única) mostra-se na Fig. 1 b).

Deverá resolver este problema implementando o algoritmo AC-3 de inferência de restrições de consistência de arco. Para tal utiliza a linguagem Python ou Julia. *Importante: não se pretende, neste exercício, a implementação do algoritmo de backtracking mas sim o AC-3 apenas. A apresentação do algoritmo de backtracking não terá qualquer valor pois não é o pretendido.*

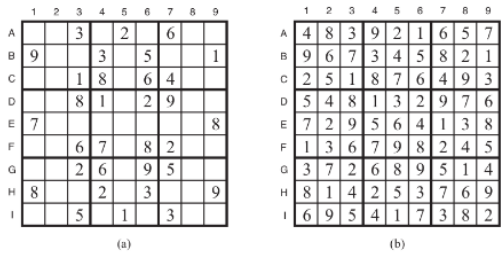

Figura 1: a) Problema de Sudoku. b) Solução do problema de sudoku proposto em a).

Fundamentos Teóricos

O algoritmo AC-3 é um algoritmo de pesquisa de Satisfação de Restrições (CSP) que é usado para determinar se é possível atribuir valores a variáveis de modo a satisfazer todas as restrições do problema. O AC-3 faz isso ao verificar se há inconsistências entre as restrições e à medida que verifica, vai removendo os valores possíveis das variáveis que não podem ser satisfeitas. É baseado no método consistência de arcos, que envolve a verificação de cada arco em um grafo de restrições para garantir que todos os valores possíveis de uma variável sejam possíveis de acordo com as restrições.

Apesar de ser um algoritmo eficiente em muitos casos, ele apresenta algumas limitações, tais como, só pode ser usado em problemas de CSP que têm restrições binárias, ou seja, restrições que envolvem apenas duas variáveis; o AC-3 não é capaz de resolver todos os problemas de CSP; pode acabar por ser lento em problemas com muitas variáveis e com um grande número de restrições; não é capaz de lidar com restrições que não sejam lineares, ou seja, se o problema apresentar restrições não lineares, não é capaz de o resolver.

Implementação AC-3

O Sudoku pode ser interpretado como um Problema de Satisfação de Restrições (CSP) apresentando um conjunto de 81 variáveis, visto que, neste caso, o Sudoku é uma tabela 9x9 que está subdividida em nove blocos 3x3.

Para a resolução de um Sudoku é necessário seguir regras, nas quais, é necessário preencher a tabela com número de 1 a 9, de modo que não ocorra repetições de números na mesma linha, coluna ou bloco 3x3.

O AC3 sendo um algoritmo destinado a resolver um CSP, o mesmo é composto por variáveis, conjunto de domínios e conjunto de restrições.

Na resolução do exercício proposto, a célula (‘00’) não pode possuir o mesmo número que as células da mesma linha que são designadas por (‘01’), (‘02’), (‘03’), (‘04’), (‘05’), (‘06’), (‘07’), (‘08’), tal como não pode ser igual às células da coluna em que se encontra, designadas por (‘10’), (‘20’), (‘30’), (‘40’), (‘50’), (‘60’), (‘70’), (‘80’) e, por fim, também tem de ser diferente das células presente no mesmo bloco 3x3, designadas por (‘11’), (‘12’), (‘21’), (‘22’). Ou seja, existe um grande número de restrições, contabilizamos 1620 restrições formadas, sobre o formato de ['V00', 'V01'].

De modo a entender melhor os conceitos referidos anteriormente, iremos explicar detalhadamente e mostrar como os dados são armazenados na nossa implementação.

Vaiáveis (Variables):

As variáveis representam cada uma das células do Sudoku e estão inseridas numa lista com todas as posições possíveis do Sudoku.

```
['V00', 'V01', 'V02', 'V03', 'V04', 'V05', 'V06', 'V07', 'V08', 'V10', 'V11', 'V12', 'V13', 'V14', 'V15',
'V16', 'V17', 'V18', 'V20', 'V21', 'V22', 'V23', 'V24', 'V25', 'V26', 'V27', 'V28', 'V30', 'V31', 'V32',
'V33', 'V34', 'V35', 'V36', 'V37', 'V38', 'V40', 'V41', 'V42', 'V43', 'V44', 'V45', 'V46', 'V47', 'V48',
'V50', 'V51', 'V52', 'V53', 'V54', 'V55', 'V56', 'V57', 'V58', 'V60', 'V61', 'V62', 'V63', 'V64', 'V65',
'V66', 'V67', 'V68', 'V70', 'V71', 'V72', 'V73', 'V74', 'V75', 'V76', 'V77', 'V78', 'V80', 'V81', 'V82',
'V83', 'V84', 'V85', 'V86', 'V87', 'V88']
```

Restrições (Constraints):

As restrições são as regras do Sudoku que dizem, como referido anteriormente, que uma célula não pode ter o mesmo valor que outra célula na mesma linha, coluna ou bloco 3x3. Os seguintes dados foram encurtados, uma vez que são bastantes restrições guardadas.

```
[['V00', 'V01'], ['V00', 'V02'], ['V00', 'V03'], ['V00', 'V04'], ['V00', 'V05'], ['V00', 'V06'], ['V00',
'V07'], ['V00', 'V08'], ['V00', 'V10'], ['V00', 'V20'], ['V00', 'V30'], ['V00', 'V40'], ['V00', 'V50'],
['V00', 'V60'], ['V00', 'V70'], ['V00', 'V80'], ['V00', 'V11'], ['V00', 'V12'], ['V00', 'V21'], ['V00',
'V22'], ['V01', 'V00'], ['V01', 'V02'], ['V01', 'V03'], ['V01', 'V04'], ['V01', 'V05'], ['V01', 'V06'],
['V01', 'V07'], ['V01', 'V08'], ['V01', 'V11'], ['V01', 'V21'], ['V01', 'V31'], ['V01', 'V41'], ['V01',
'V51'], ['V01', 'V61'], ... , ['V88', 'V76'] , ['V88', 'V77']]
```

Domínios (Domains):

Cada variável tem um domínio de valores possíveis. Optamos por criar um dicionário sendo a *key* a posição da célula (variable) e o *value* uma lista com todos os valores que a posição pode assumir.

```
{'V00': [1, 2, 3, 4, 5, 6, 7, 8, 9],
'V01': [1, 2, 3, 4, 5, 6, 7, 8, 9],
....
}
```

Vizinhança (Neighbors):

Os *neighbors* são as outras células que compartilham uma linha, coluna ou bloco 3x3 com a célula em questão, ou seja, como podemos verificar no seguinte dicionário, que uma vez mais foi encurtado devido à grande quantidade de dados, para a célula ‘V00’, tem como *neighbors* ['V01', 'V02', 'V03', 'V04', 'V05', 'V06', 'V07', 'V08', 'V10', 'V20', 'V30', 'V40', 'V50', 'V60', 'V70', 'V80', 'V11', 'V12', 'V21', 'V22', 'V01', 'V02', 'V03', 'V04', 'V05', 'V06', 'V07', 'V08', 'V10', 'V11', 'V12', 'V20', 'V21', 'V22', 'V30', 'V40', 'V50', 'V60', 'V70', 'V80'], ou seja, este dicionário tem como *key* a posição da célula e como *values* todas as posições das células *neighbors*.

```
{'V00': ['V01', 'V02', 'V03', 'V04', 'V05', 'V06', 'V07', 'V08', 'V10', 'V20', 'V30', 'V40', 'V50',
'V60', 'V70', 'V80', 'V11', 'V12', 'V21', 'V22', 'V01', 'V02', 'V03', 'V04', 'V05', 'V06', 'V07', 'V08',
'V10', 'V11', 'V12', 'V20', 'V21', 'V22', 'V30', 'V40', 'V50', 'V60', 'V70', 'V80'], 'V01': ['V00',
'V00', 'V02', 'V03', 'V04', 'V05', 'V06', 'V07', 'V08', 'V11', 'V21', 'V31', 'V41', 'V51', 'V61', 'V71',
'V81', 'V10', 'V12', .... , 'V76', 'V77']}
```

Dentro do ficheiro Python **AC3_final.py**, irá ocorrer a implementação do algoritmo AC-3 que inicia com a definição das duas funções presentes no pseudocódigo fornecido pelo professor, as funções **ac3()** e **revise()**.

Começando pela função **ac3()**, irá ser preciso a implementação de uma classe CSP que mais à frente irá ser implementada.

```
def ac3(csp):
    queue = csp.C
    while queue:
        arc_value = queue.pop(0)  # arc_value = [Xi, Xj]
        Xi = arc_value[0]
        Xj = arc_value[1]

        if revise(csp, Xi, Xj):
            if len(csp.D[Xi]) == 0:
                return False  # Se o CSP não tiver solução dá return
False
            for Xk in csp.N[Xi]:
                if Xk != Xj:
                    queue.append([Xk, Xi])
    return True
```

Esta função funciona iterando sobre a *queue* de pares de variáveis e vai tentando reduzir o domínio de cada variável de acordo com as restrições do problema.

Inicialmente a *queue* é preenchida com todas as restrições que devem ser consideradas e, em cada iteração, a primeira restrição é removida da fila e as variáveis envolvidas são criadas como Xi e Xj. Na condição *if*, a função **revise()** é chamada para tentar reduzir o domínio de Xi de acordo com a restrição com Xj. Se a função **revise()** retornar *true*, então significa que o domínio de Xi foi alterado e, portanto, todas as outras restrições que envolvem Xi necessitam de ser revistas. Essas restrições são adicionadas de volta à fila para serem alvos de revisão mais tarde. Caso o tamanho do domínio de Xi seja igual a zero (if len(csp.D[Xi]) == 0:), então significa que não há valores possíveis para Xi que satisfaçam todas as restrições, então a função retorna *false* indicando que o problema não tem solução.

De seguida, criamos a função **revise()** que é a função chamada pela função **ac3()** para tentar reduzir o domínio de uma variável de acordo com uma restrição.

```
def revise(csp, Xi, Xj):
    revised = False
    # Se o valor de Xi não existir no dominio de Xj
    for x in csp.D[Xi][:]:
        if not any(x != y for y in csp.D[Xj]):
            csp.D[Xi].remove(x)
            revised = True
    return revised
```

Esta função percorre todos os valores no domínio de Xi e verifica se há algum valor que não esteja do domínio de Xj. Caso não haja, então esse valor é removido do domínio de Xi e a

variável *revised* é definida como *true*. Isto significa que o domínio de Xi foi alterado e outras restrições envolvimento Xi precisam ser revistas.

Se no final da iteração, *revised* ainda for *false*, então significa que o domínio de Xi não foi alterado e, portanto, não há necessidade de efetuar uma revista a outras restrições que envolvem Xi.

Após implementarmos as funções **ac3()** e **revise()** de acordo com o pseudocódigo, criamos a classe chamada CSP que é importante para a função **ac3()**. Esta classe representa um problema de satisfação de restrições.

```
class CSP:
    def __init__(self):
        self.V = []  # lista de variáveis
        self.C = []  # lista de restrições(constrains)(arcs)
        self.D = {}  # dicionário de domínios para cada variável
        self.N = {}  # dicionário de neighbors para cada variável


    def add_variable(self, variable, domain):
        self.V.append(variable)
        self.D[variable] = domain
        self.N[variable] = []

    def add_constraint(self, variable1, variable2):
        self.N[variable1].append(variable2)
        self.N[variable2].append(variable1)
        self.C.append([variable1, variable2])
```

Esta classe é iniciada com a definição de duas listas e dois dicionários. A lista 'V' representa uma lista de variáveis do problema. A lista 'C' representa uma lista de restrições do problema, onde cada restrição é representada como uma lista de duas variáveis. O dicionário 'D' faz a correspondência *key-value* de cada variável para o seu domínio de valores possíveis e o dicionário 'N' faz a correspondência *key-value* de cada variável para uma lista da sua vizinhança (outras variáveis pela qual tem restrição).

Apresenta também dois métodos ***add_variable()*** que adiciona uma nova variável ao problema com um determinado domínio de valores possíveis e ***add_constraint()*** que adiciona uma nova restrição entre duas variáveis.

Os métodos são usados para construir o problema e armazená-lo na estrutura de dados da classe CSP e, depois, o algoritmo AC-3 pode ser usado para tentar resolver o problema.

Por fim, ocorre a implementação de **solve_sudoku()**. Esta função é responsável por resolver o Sudoku com o uso do algoritmo AC-3.

```
def solve_sudoku():
    puzzle = [
    [0, 0, 3, 0, 2, 0, 6, 0, 0],
    [9, 0, 0, 3, 0, 5, 0, 0, 1],
    [0, 0, 1, 8, 0, 6, 4, 0, 0],
    [0, 0, 8, 1, 0, 2, 9, 0, 0],
    [7, 0, 0, 0, 0, 0, 0, 0, 8],
    [0, 0, 6, 7, 0, 8, 2, 0, 0],
    [0, 0, 2, 6, 0, 9, 5, 0, 0],
    [8, 0, 0, 2, 0, 3, 0, 0, 9],
    [0, 0, 5, 0, 1, 0, 3, 0, 0]
]
    # cria objeto CSP
    csp = CSP()

    # adicionar variáveis ao objeto CSP
    for i in range(9):
        for j in range(9):
            if puzzle[i][j] == 0:
                csp.add_variable(f"V{i}{j}", [1, 2, 3, 4, 5, 6, 7, 8, 9])
            else:
                csp.add_variable(f"V{i}{j}", [puzzle[i][j]])

    # adicionar restrições ao objeto CSP
    for row in range(9):
        for column in range(9):
            # verifique se há restrições na linha
            for row_position in range(9):
                if row_position != column:
                    csp.add_constraint(f"V{row}{column}", f"V{row}{row_position}")

            # verifique se há restrições na coluna
            for column_position in range(9):
                if column_position != row:
                    csp.add_constraint(f"V{row}{column}",
f"V{column_position}{column}")

            # verifique se há restrições na grade 3x3
            row_start = row // 3 * 3
            col_start = column // 3 * 3
            # x -> row
            # y -> column
            for x in range(3):
                for y in range(3):
                    if row_start + x != row and col_start + y != column:
                        csp.add_constraint(f"V{row}{column}", f"V{row_start +
x}{col_start + y}")
```

A função inicia com a grelha do sudoku e com a criação de um objeto CSP.

De seguida, no *for loop* a função adiciona cada célula do Sudoku como uma variável ao objeto CSP. Se a célula tiver um valor conhecido no Sudoku, então o domínio da variável é definido como o valor da célula. Caso a célula não tenha um valor conhecido, então o domínio da variável é definido como todos os número de 1 a 9.

Após as variáveis serem adicionadas ao objeto CSP, é necessário adicionar restrições entre elas. Para cada posição de uma dada célula é efetuada a verificação de restrições dada a sua linha, coluna e o bloco 3x3 em que está inserida.

Após todas as restrições serem adicionadas, o AC3 é chamado para tentar resolver o Sudoku. Se o AC-3 retornar *true*, significa que o problema foi resolvido com sucesso, caso contrário, significa que o problema não tem solução.

```
    # resolver o CSP usando o algoritmo AC-3
    if ac3(csp):
        print("""Sudoku por resolver:
+-------+-------+-------+
| 0 0 3 | 0 2 0 | 6 0 0 |
| 9 0 0 | 3 0 5 | 0 0 1 |
| 0 0 1 | 8 0 6 | 4 0 0 |
|-------+-------+-------|
| 0 0 8 | 1 0 2 | 9 0 0 |
| 7 0 0 | 0 0 0 | 0 0 8 |
| 0 0 6 | 7 0 8 | 2 0 0 |
|-------+-------+-------|
| 0 0 2 | 6 0 9 | 5 0 0 |
| 8 0 0 | 2 0 3 | 0 0 9 |
| 0 0 5 | 0 1 0 | 3 0 0 |
+-------+-------+-------+
        """)
        # imprime o Sudoku resolvido dentro de uma grid
        print("Sudoku Resolvido:")
        print("+-------+-------+-------+")
        for i in range(9):
            for j in range(9):
                if j == 0:
                    print("| ", end='')
                print(csp.D[f"V{i}{j}"][0], end=" ")
                if j == 2 or j == 5:
                    print("|", end=" ")
            print("|")
            if i == 2 or i == 5:
                print("-" * 25)
        print("+-------+-------+-------+")
    else:
        print("Sem solução.")
```

4 Menu

Implementação AC-3

O grupo optou pela criação de um ficheiro Python com o nome **program.py** com a criação de um menu interativo no terminal de modo ao utilizador poder escolher qual dos três algoritmos à sua disposição (Naive Bayes, Perceptrão e AC-3) pretende utilizar.

```
import NaiveBayes, perceptrao, csv_reader, AC3_final
import sys
import time

sys.stdout.reconfigure(encoding="utf-8")

def opcao_escolhida():
    try:
        temp_pick = input()
    except EOFError:
        return None
    else:
        comandos = temp_pick.split(" ")
        comandos[0] = comandos[0].lower()
    return comandos

if __name__ == "__main__":

    while True:
        print("""
                 ==========================Menu==============================
                 Naive -> Executa o algoritmo de Naive Bayes(spam e ham);
                 Perceptrao -> Executa o algoritmo de perceptrão(spam e ham);
                 AC3 -> Executa o algoritmo de AC3(resolver sudoku);
                 Exit -> Encerra o programa;
                 ============================================================
                 """)

        escolha = opcao_escolhida()
        lista_treino, lista_teste, lista_validacao = csv_reader.csv_reader()
        if escolha[0] == "exit" or escolha[0] == " ":
            break
        if escolha[0] == "naive":
            start_time = time.time()
            NaiveBayes.train(lista_treino)
            NaiveBayes.classify_list(lista_validacao)
            print("Tempo de execução: %s segundos." % (time.time() - start_time))
        if escolha[0] == "perceptrao":
            start_time = time.time()
            teta, teta_zero = perceptrao.perceptron(lista_treino, 10)
            perceptrao.perceptron_classify_list(lista_validacao, teta, teta_zero)
            print("Tempo de execução: %s segundos." % (time.time() - start_time))
        if escolha[0] == "ac3":
            start_time = time.time()
            AC3_final.solve_sudoku()
            print("Tempo de execução: %s segundos." % (time.time() - start_time))
```

Caso o utilizador deseje percorrer o algoritmo de Naive Bayes para o filtro spam, basta percorrer o ficheiro Python **program.py** e escrever no terminal "naive" e será o mostrado o seguinte no terminal:

```
--->Algoritmo Naive Bayes 70/15/15 <---
O algoritmo percorreu: 836 emails!

Número de emails spam avaliados: 95
-> Avaliou corretamente 57
-> Avaliou incorretamente: 38

Número de emails ham avaliados: 741
-> Avaliou corretamente 716
-> Avaliou incorretamente: 25

O algoritmo obteve uma taxa de sucesso igual a: 92.46411483253588 %
O algoritmo obteve uma taxa de insucesso igual a: 7.535885167464116 %

Métricas de Classificação:
   Accuracy: 0.9246411483253588
   Error rate: 0.07535885167464115
   Sensivity: 0.6
   Specificity: 0.9662618083670715
   Precision: 0.6951219512195121
   Recall: 0.07373868046571798

Tempo de execução: 0.022021055221557617 segundos.
```

Caso o utilizador deseje percorrer o algoritmo do Perceptrão para o filtro spam, basta percorrer o ficheiro Python **program.py** e escrever no terminal "perceptrao" e será o mostrado o seguinte no terminal:

```
---> Algoritmo do Perceptrão 70/15/15 <---
O algoritmo percorreu: 836 emails!

Número de emails spam avaliados: 120
-> Avaliou corretamente 100
-> Avaliou incorretamente: 20

Número de emails ham avaliados: 716
-> Avaliou corretamente 710
-> Avaliou incorretamente: 6

O algoritmo obteve uma taxa de sucesso igual a: 96.88995215311004 %
O algoritmo obteve uma taxa de insucesso igual a: 3.1100478468899553 %

Métricas de Classificação:
   Accuracy: 0.9688995215311005
   Error rate: 0.03110047846889952
   Sensivity: 0.8333333333333334
   Specificity: 0.9916201117318436
   Precision: 0.9433962264150944
   Recall: 0.12345679012345678

Tempo de execução: 0.07307243347167969 segundos.
```

Caso o utilizador deseje percorrer o algoritmo AC-3 para resolver o Sudoku, basta percorrer o ficheiro Python **program.py** e escrever no terminal "ac3" e será o mostrado o seguinte no terminal:

```
Sudoku por resolver:
 +--------+--------+--------+
 | 0 0 3 | 0 2 0 | 6 0 0 |
 | 9 0 0 | 3 0 5 | 0 0 1 |
 | 0 0 1 | 8 0 6 | 4 0 0 |
 |--------+--------+--------|
 | 0 0 8 | 1 0 2 | 9 0 0 |
 | 7 0 0 | 0 0 0 | 0 0 8 |
 | 0 0 6 | 7 0 8 | 2 0 0 |
 |--------+--------+--------|
 | 0 0 2 | 6 0 9 | 5 0 0 |
 | 8 0 0 | 2 0 3 | 0 0 9 |
 | 0 0 5 | 0 1 0 | 3 0 0 |
 +--------+--------+--------+

Sudoku Resolvido:
+--------+--------+--------+
| 4 8 3 | 9 2 1 | 6 5 7 |
| 9 6 7 | 3 4 5 | 8 2 1 |
| 2 5 1 | 8 7 6 | 4 9 3 |
-----------------------------
| 5 4 8 | 1 3 2 | 9 7 6 |
| 7 2 9 | 5 6 4 | 1 3 8 |
| 1 3 6 | 7 9 8 | 2 4 5 |
-----------------------------
| 3 7 2 | 6 8 9 | 5 1 4 |
| 8 1 4 | 2 5 3 | 7 6 9 |
| 6 9 5 | 4 1 7 | 3 8 2 |
+--------+--------+--------+
Tempo de execução: 0.024524927139282227 segundos.
```

5 Conclusão

A realização deste trabalho teve uma contribuição muito positiva na aprendizagem dos conteúdos que foram lecionados pelo professor nas aulas, proporcionando a consolidação e a aplicação dos mesmos neste projeto. Para além dos conteúdos lecionados na aula o grupo realizou pesquisas na internet de modo a consolidar ainda mais o conteúdo dos três algoritmos implementados.