PÓS-GRADUAÇÃO EM SEGURANÇA CIBERNÉTICA

2022PGS1M1

Introdução a I.A.

Trabalho Final

Filipe Sousa - 163610

Gelson Filho - 160157

Otavio Marelli - 153381

Rodrigo Camargo - 132036

Prof.: Paulo R. Nietto

Sorocaba / SP

02/09/2022

LISTA DE FIGURAS

SUMÁRIO

1. INTRODUÇÃO

Com o grande avanço tecnológico que tivemos nas últimas décadas, obteve-se grande progresso também na área de estudos da Inteligência Artificial, pois assuntos relacionados a este tema, que há alguns anos era mais restrito aos pesquisadores e desenvolvedores, hoje pode ser encontrada no dia a dia de pessoas que não têm nenhuma formação na área de tecnologia, seja através do uso de *chatbots* em sites de compras; assistentes pessoais que controlam agendas, iluminações de ambientes, dispositivos eletrodomésticos; algoritmos que auxiliam na busca de determinados conteúdos e na recomendação destes; algoritmos que oferecem a rota com menos trânsito em um determinado trajeto; dentre outras diversas aplicações.

Uma dessas muitas aplicações da Inteligência Artificial é o aprendizado de máquina, que pode ser compreendido como um método que é utilizado para automatizar a construção de modelos analíticos, que através da aprendizagem de dados, busca identificar padrões e tomar decisões com o menor nível de interferência ou decisões humanas envolvidas nestes processos. Tal metodologia permite que o algoritmo se adapte de forma independente quando novos dados são enviados a ele, baseando-se em dados que foram computados em momentos anteriores.

Apesar de não ser tão atual, com a grande quantidade de dados e informações que têm sido geradas por meio da utilização de aplicativos, smartphones e outros dispositivos, torna-se quase impossível realizar a análise desta massa de dados manualmente, sendo necessária a utilização de ferramentas que se beneficiam dessa característica de aprendizado de máquina.

2. OBJETIVOS

Através deste relatório, espera-se realizar a demonstração e aplicação do conteúdo lecionado durante as aulas de Introdução à Inteligência Artificial, especificando termos referentes ao aprendizado de máquina e aos processos desta metodologia, como os tipos de aprendizados, a metodologia escolhida para ser aplicada e a base de dados que foi selecionada para realizar o treinamento do algoritmo, bem como os resultados obtidos através deste desafio.

3. DESENVOLVIMENTO

Para realizar este relatório, fez-se necessário organizar e colocar em prática os conhecimentos adquiridos acerca de base de dados e tipos de algoritmos, selecionar um algoritmo para realizar a tentativa de aplicação de aprendizado de máquina e então realizar a análise dos dados obtidos.

O aprendizado de máquina pode ser realizado através de diferentes métodos de aprendizado, sendo estes o aprendizado supervisionado, o aprendizado não supervisionado, o aprendizado semi-supervisionado e o aprendizado por esforço.

No aprendizado supervisionado o algoritmo é alimentado com dados que possuem rótulos, portanto em seu banco de dados existem dados que, sabe-se que estão “corretos” ou não, e através destes parâmetros, o algoritmo passa a identificar dados futuros.

O aprendizado não supervisionado, por outro lado, não tem a informação de quais dados estão “corretos”, ele apenas recebe exemplos não rotulados, e cabe ao algoritmo realizar a identificação de padrões para definir se os dados pertencem ou não a uma mesma classe.

Tem-se o aprendizado semi-supervisionado, que abrange os dois conceitos citados anteriormente, e é utilizado geralmente em casos em que essa rotulação dos dados tem um custo muito elevado.

Por fim, o algoritmo de aprendizado por esforço realiza diversas tentativas para identificar quais escolhas ele deve tomar, através do método de tentativa e erro.

3.1 Base de dados selecionada

Com o intuito de realizar a aplicação de um modelo de aprendizado de máquinas, buscou-se uma base da dados que estivesse relacionada ao tema de cyber segurança.

O repositório utilizado para a seleção desta base foi o *“UCI Machine Learning Repository”*, conforme sugerido na documentação de auxílio de elaboração desta atividade. O *UCI* foi criado em 1987 e desde então tem contribuído em diversos documentos e pesquisas relacionados ao aprendizado de máquinas.

No repositório do *UCI*, buscou-se bases de dados que estivessem interligadas com a área do curso de pós graduação, a *cyber* segurança, e dentre diversas bases, incluindo algumas sobre dados de *Firewalls*, dispositivos *IoTs*, e ataques a redes, optou-se por utilizar uma base de dados sobre *phishing*, visto que esta é uma categoria de ataque cibernético que tem se destacado nos últimos anos, crescendo exponencialmente durante a pandemia, tendo um aumento de 80% de tentativas de ataques somente em 2021, de acordo com dados da Federação Brasileira de Bancos (Febraban).

Esta base é composta por dados de 1353 diferentes *websites*, de diversas fontes, dos quais 548 são sites legítimos, enquanto 702 são sites de *phishing*, sendo os demais sites que apresentam características de sites legítimos, porém ao mesmo tempo apresentam características de sites de *phishing*, caracterizando-os como sites suspeitos.

O intuito da utilização desta base de dados com o algoritmo é fazer com que ele consiga realizar a classificação de sites confiáveis, tentativas de *phishing* e possíveis tentativas de *phishing*, mediante o aprendizado que é realizado com 10 atributos. Por ser uma aplicação que envolve um problema de classificação, optou-se pela utilização do algoritmo KNN, que será abordado no tópico 3.3.

Os dados que compõem o *dataset*, a princípio, passaram por um pré-processamento de modo a categorizá-los. Tal processo, assim como uma breve descrição de cada tipo de dado está disposta abaixo, no subcapitulo 3.1.X.

3.1.1 *URL Anchor*

Âncoras são elementos *HTML* definidos pela *tag* <a> que são utilizados para direcionar o usuário para outra parte da mesma página ou outras *URL's*.

Âncoras suspeitas podem ser identificadas quando:

- A *tag* <a> direciona a *URL's* de domínios diferentes do original, similar ao caso dos "*Request* URL"
- A âncora não se referência a própria página utilizando um dos recursos de HTML:
    - <a href= "#">
    - <a href= "#content">
    - <a href= "#skip">
    - <a href= "JavaScript::void(0)">

Com base nos critérios propostos acima os dados contidos no *dataset* seguem a regra abaixo para sua classificação:

$$Regras\ para\ Ancoras: \begin{cases} \%\ Ancoras\ suspeitas < 31\% \rightarrow Legitimo\ (1) \\ \%\ Ancoras\ suspeitas \geq 31\%\ e\ < 67\% \rightarrow Suspeito\ (0) \\ Outros\ casos\ \rightarrow Phishing\ (-1) \end{cases}$$

3.1.2 *Request URL*

Em geral URL's que possuem imagens, vídeos ou áudios acessam tais recursos de seus próprios domínios, na grande maioria ao menos, porém em casos de *phishing* a URL busca tais recursos de outros domínios, gerando solicitações de acesso a outros endereços.

Com base nesse conceito foi implementada a seguinte regra ao *dataset* e a dados para análise que considera as seguintes características de solicitações de acesso a URL's terceiras:

$$Regras\ para\ Request\ URL: \begin{cases} \%\ acesso\ a\ outras\ URL's < 22\% \rightarrow Legitimo\ (1) \\ \%\ acesso\ a\ outras\ URL's \geq 22\%\ e\ < 61\% \rightarrow Suspeito\ (0) \\ Outros\ casos\ \rightarrow Phishing\ (-1) \end{cases}$$

3.1.3 *Server Form Handler (SFH)*

Por questões de segurança e organização, portais financeiros/e-commerce exigem autenticação (login e senha). Para realização de login as páginas possuem formulários a serem preenchidos com tais dados, por exemplo:

<form action="/inetSearch/index.jsp" method="post" target="top"> in http://www.chase.com

Em casos de *phishing* o formulário preenchido acima direcionaria as URL's diferentes ou são mantidos sem redirecionamento, sendo especificado como '*about:blank*'.

Portanto os dados inseridos ao *dataset* e analisados posteriormente devem seguir a regra descrita abaixo:

$$Regras\ para\ SFH: \begin{cases} SFH\ com\ 'about:blank'\ ou\ em\ branco \rightarrow Phishing\ (-1) \\ SFH\ que\ direcionam\ a\ outras\ URL's \rightarrow Suspeito\ (0) \\ Outros\ casos\ \rightarrow Legitimo\ (1) \end{cases}$$

3.1.4 *URL Lenght*

O comprimento das URL's pode ser utilizado como um recurso pelos *phishers* para disfarçar partes duvidosas do link, como por exemplo:

http://mercadolivre.com.br/3f/aze/ab51e2e319e54802f416dbe46b773a5e/?cmd=_home&dispatch=11004d54465174f8dc1e7c2e8dd4105e811004d58f5b74f8dc1e7c2e8dd4105e8@phishing.website.html@phishing.website.html

Sendo assim os autores do *dataset* realizaram um estudo de comprimento de *links*, determinando a média de caracteres por URL e verificou-se que os casos de *phishing* começam a ser recorrentes em links com mais de 54 caracteres, levando a elaboração da seguinte regra:

$$Regras\ para\ tamanho\ de\ URL's: \begin{cases} URL < 54\ caracteres \rightarrow Legitimo\ (1) \\ URL\ [> 54]\ e\ [< 74]\ caracteres \rightarrow Suspeito\ (0) \\ Outros\ casos\ \rightarrow Phishing\ (-1) \end{cases}$$

3.1.5 *Having* "@"

O uso do "@" é um método de burlar a leitura das URL's pelo navegador, onde ao inserir um link com "@" o que precede o símbolo será ignorado e o acesso será feito ao que procedo o símbolo.

Tendo em vista o conceito foi implementada a seguinte regra ao *database* e novos dados a serem analisados:

$$Regras\ para\ URL's\ com\ "@": \begin{cases} Possue\ "@" \rightarrow Phishing\ (-1) \\ Outros\ casos\ \rightarrow Legitimo\ (1) \end{cases}$$

3.1.6 *Prefix/Suffix*

O uso do símbolo de traço dificilmente é empregado em URL's legitimas, e em muitos os casos os *phishers* se aproveitam desse símbolo para adicionar prefixos ou sufixos ao domínio de *phishing*, a fim de confundir os usuários tentando se passar por URL's legítimos, como o exemplo abaixo demonstra:

http://www.confirme-mercadolivre.com.br/

Com base nessa análise, links no *dataset* que possuem o símbolo de traço e URL's que serão analisados devem passar pela seguinte regra de classificação:

$$Regras\ para\ Prefixos\ e\ Sufixos: \begin{cases} Possue\ "-" \rightarrow Phishing\ (-1) \\ Outros\ casos\ \rightarrow Legitimo\ (1) \end{cases}$$

3.1.7 IP

URL's que são formadas por IP em muitos os casos são de grande suspeita, outro formato empregado pelos *phishers* é a utilização do *link* utilizando o endereço IP em formato hexadecimal, conforme exemplo:

"http://0x58.0xCC.0xCC.0x62/5/paypal.com/index.html"

Em quase todos os casos do gênero, trata-se de sites de *phishing* e, portanto, a seguinte regra foi adotada aos dados do *dataset*:

$$Regras\ para\ URL's por\ IP: \begin{cases} Utiliza\ endereço\ IP\ como\ dominio \rightarrow Phishing\ (-1) \\ Outros\ casos\ \rightarrow Legitimo\ (1) \end{cases}$$

3.1.8 *Sub Domain*

Para compreender como a regra de subdomínio funciona iremos analisar o seguinte link: *https://www.facens.edu.br/estudante/*. O nome de domínio, a princípio, pode incluir o código de pais junto aos domínios de primeiro nível (ccTLD) que no exemplo é "*br*", e junto temos o "*edu"* que se refere a instituições de educação e se classifica como domínio de primeiro nível secundário (sTLD).

Para analisar as URL's e aplicar a uma regra é necessário também ignorar o "*www.*" e o "*edu.br*" sobrando apenas o nome principal do domínio ("*facens.*"), com isso é possível realizar a contagem de pontos e classificar com base na regra estabelecida no dataset.

$$Regra\ para\ Sub\ Domain: \begin{cases} Numero\ \ de\ subdominios\ (pontos) = 1\ \rightarrow Legitimo\ (1) \\ Numero\ \ de\ subdominios\ (pontos) = 2\ \rightarrow Suspeito\ (0) \\ Outros\ casos\ \rightarrow Phishing\ (-1) \end{cases}$$

3.1.9 *Web Traffic*

O input do tipo *"Web Traffic"*, visa verificar a popularidade do site, através do número de visitantes e o número de páginas que eles visitam, tendo em vista que sites *phishing* não possuem um longo tempo de vida, o que limita sua popularidade.

Nesses casos sites desse gênero não estarão presentes em *ranks* de sites mais visitados, como: *Similarweb, Semrush, Ahrefs, etc..* Em específico o *dataset* utiliza como base os dados fornecidos pelo antigo *website Alexa* (*Alexa the Web Information Company*), descontinuado em 2022.

Com base nesse conceito o *database* avaliou essa informação fornecida pelo antigo *Alexa.com* e atribuiu valores de –1, 0 e 1, seguindo a seguinte regra:

$$Regra\ para\ Web\ Traffic: \begin{cases} Posição\ no\ website\ rank < 100.000\ \rightarrow Legitimo\ (1) \\ Posição\ no\ website\ rank > 100.000\ \rightarrow Suspeito\ (0) \\ Fora\ do\ rank\ \rightarrow Phishing\ (-1) \end{cases}$$

3.1.10 *Domain age*

Tendo em vista que seria impossível remover pontos de entrada de dados, é necessário realizar a aplicação de filtro dos dados que são submetidos a aplicação, isso é um pré-processamento que seleciona e remove possíveis pacotes maliciosos ou suspeitos.

Baseado no fato de que sites de *phishing* possuem uma vida curta, os autores do *dataset* propuseram uma regra de avalia o tempo pelo qual o domínio está *online*, uma vez que domínios de procedência confiável se permanecem por longos períodos, sendo assim, os dados inseridos ao *dataset* e de possível análise devem seguir a seguinte regra de classificação:

$$Regras\ para\ Idade\ do\ dominio: \begin{cases} Idade\ do\ dominio\ \leq 1\ ano \rightarrow Phishing\ (-1) \\ Outros\ casos\ \rightarrow Legitimo\ (1) \end{cases}$$

3.1.11 *Class*

Com base nos demais parâmetros levantados cada URL pode se enquadrar em uma das classificações final como resultado, sendo elas:

$$Possiveis\ resultados: \begin{cases} URL\ Legitima\ (1) \\ URL\ Suspeita\ (0) \\ URL\ de\ Phishing\ (-1) \end{cases}$$

3.2 Aprendizagem baseada em instâncias

O algoritmo selecionado para a realização da tarefa proposta, visto que a mesma exigia uma solução de classificação, é o “K vizinhos mais próximos” (KNN), que se enquadra na classificação de aprendizado de máquina baseado em instâncias, sendo este um sistema de aprendizado supervisionado, que realiza a classificação de novas instâncias com base nas instâncias de treinamento que são enviados a ele. É também conhecido como aprendizado baseado em memória, e sua complexidade de tempo se dá de acordo com o tamanho dos dados de treinamento.

As principais desvantagens envolvendo este tipo de algoritmo se dá pelo fato de que os custos são mais elevados, visto que é necessária muita memória para realizar o armazenamento destes dados de treinamento, além de que cada nova consulta realiza a identificação partindo de um novo modelo local. Por outro lado,

suas vantagens se dão no fato de que este tipo de algoritmo pode se adaptar facilmente a novas instâncias de treinamento.

3.3 KNN

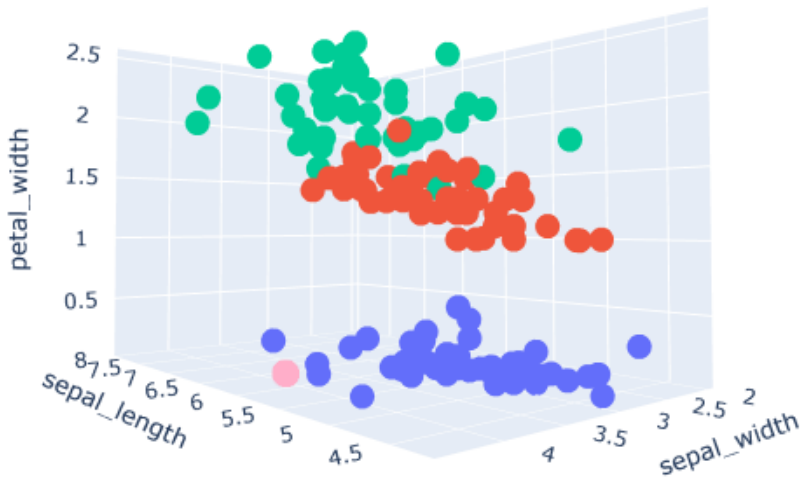
O “*K Nearest Neighbors*” (KNN) ou “K Vizinhos mais próximos”, como descrito anteriormente, é um algoritmo que se encaixa na categoria de aprendizado supervisionado, o que indica que os dados de treinamento transmitidos a ele são rotulados, e é comumente utilizado para solucionar problemas de classificações, apesar de poder ser utilizado também em problemas que envolvem regressão linear.

Todos os dados de treinamento (instâncias) são armazenados, e para cada novo dado a ser classificado, estima-se a qual classe este novo dado apresentado pertence com base na distância entre este e os já classificados.

O funcionamento do algoritmo KNN se dá de forma que um parâmetro, chamado nesse contexto de “K”, será avaliado pelo algoritmo em relação à distância que possui dos vizinhos mais próximos. Caso os vizinhos mais próximos pertençam todos a uma mesma classe, o parâmetro avaliado será considerado como participante desta classe.

Supondo-se que há um conjunto de dados composto por três classes, sendo estas representadas nas cores azul, vermelho, e verde, é então apresentado a esse conjunto de dados uma nova amostra não classificada, que é representada na cor rosa, conforme a figura 1.

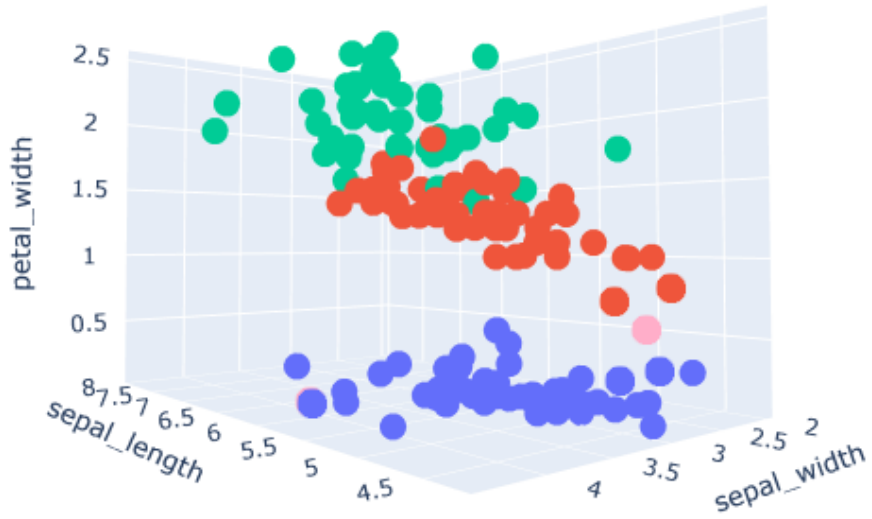
Figura 1 - Exemplo de classificação KNN 1



Fonte: Autores

Utilizando-se do KNN, seleciona-se os vizinhos mais próximos a fim de identificar a qual classe esse novo dado pertence. Analisando a figura 2, entende-se que devido ao fato dos três vizinhos mais próximos pertencerem à classe “azul”, o novo dado também será pertencente a essa classe.

Figura 2 - Exemplo de classificação KNN 2



Fonte: Autores

Adicionando um novo dado, agora em outro ponto do gráfico, e aumentando de 3 para 5 o valor de K vizinhos mais próximos, o algoritmo há de classificar este novo dado como pertencente à classe “azul”, visto que dos 5 vizinhos mais próximos, 2 são vermelhos, enquanto 3 são azuis.

Quanto ao valor de K, caso este valor seja demasiadamente pequeno, a classificação tende a se tornar mais sensível a regiões próximas, podendo levar a problemas de overfitting, que consiste em desempenhos excelentes quando utilizados os dados de treino, porém esse desempenho torna-se ruim quando se utiliza os dados de testes. Isso indica que o algoritmo decorou as relações existentes nos dados de treinamento, porém não as consegue generalizar para classificar os novos dados.

Caso o valor de K seja elevado, tende-se a obter uma classificação menos disposta a ruídos, porém é necessário levar em consideração a possibilidade de se obter *underfitting*, que é o cenário em que o desempenho apresentado pelo algoritmo é ruim mesmo com os dados de treinamento, não sendo possível obter qualquer relação entre as variáveis.

É comum utilizar a distância Euclidiana para realizar o cálculo entre as distâncias destes pontos (dados) no espaço euclidiano, porém existem outros métodos que podem ser utilizados, como a distância de *Hamming*, para quando se trata de dados binários, e a distância de *Matching*, quando se trata de dados

categóricos, além da utilização de cálculos para dados contínuos como a distância Euclidiana, que é a mais utilizada, a correlação de Pearson, utilizada para o tratamento de dados biológicos e a medida do Cosseno, para textos.

3.4 Aplicação do algoritmo

Atualmente há uma linguagem de programação que é muito utilizada e que facilita muito o desenvolvimento de algoritmos de *Machine Learning*, trata-se do *Python*. O *Python* possui uma documentação muito rica em seu site oficial além de possuir muitas bibliotecas disponibilizadas na internet e compartilhamento de conhecimento em ambientes habitualmente frequentados por outros desenvolvedores (*Stack Overflow* ou fóruns genéricos, por exemplo). Para aplicações de *Machine Learning*, comumente é utilizada a biblioteca *SKLearn*. Portanto para medirmos o quão eficiente o algoritmo foi na obtenção dos resultados utilizamos esta biblioteca. Porém, anteriormente optamos por fazer um primeiro código sem o uso da biblioteca para compreendermos plenamente e passo-a-passo o funcionamento da técnica KNN.

3.4.1 Implementação sem biblioteca

Na primeira versão de código, sem a utilização da biblioteca, utilizamos apenas a métrica de “distância Euclidiana” para os testes dada a facilidade de sua implementação além de ter o prévio conhecimento que ela é muito utilizada. O código completo pode ser visualizado no Apêndice A – Algoritmo KNN sem uso de biblioteca. No *dataset* basicamente precisamos alterar a última coluna (coluna do resultado classificado) escrevendo de fato as strings: *Legitimate* (para resultado = 1), *Suspicious* (para resultado = 0) e *Phishing* (para resultado = -1). Precisamos também recortar alguns dados para testes. Ademais basta adicionar na devida linha de execução os dados de testes para podermos ter como retorno a classificação que o algoritmo definiu para os determinados dados. Ao verificar a primeira página do apêndice anteriormente citado é intuitivo entender como o algoritmo pode ser utilizado.

3.4.2 Conceitos iniciais para implementação com bibliotecas

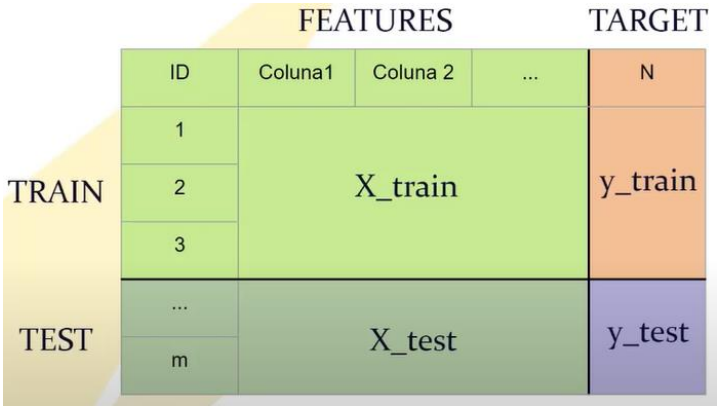
Para levantamento de dados estatísticos, acurácia e informações mais ricas sobre o algoritmo aplicado ao banco escolhido, a melhor decisão foi utilizar a biblioteca *SKLearn*.

Antes, o primeiro conceito a ser compreendido foi o da divisão dos atributos em aspectos (*Features*) e alvo (*Target*). Como anteriormente descrito e categorizado, temos no banco escolhido para a aplicação 9 atributos que consideramos como "*Features*" e 1 (a coluna *Result*) que consideramos como *Target*.

Ademais é necessário compreender a necessidade de retirar a maior quantidade de dados para treinarmos o algoritmo e a menor quantidade para teste do mesmo. Para o teste na versão do algoritmo utilizando SKLearn, poderíamos selecionar qualquer proporção arbitrária, todavia optamos por capturar aleatoriamente 20% dos dados para teste e 80% dos dados para treinamento satisfazendo o famoso Princípio de Pareto (que afirma que para muitos eventos 80% dos efeitos vêm de 20% das causas).

Juntando os conceitos dos dois parágrafos anteriores podemos entender que teremos 4 listas de dados: X_train (*features* de treino), y_train (*targets* de treino), X_test (*features* de teste) e y_test (*targets* de test). A figura 3 representa graficamente este conceito.

Figura 3 - Concatenação dos conceitos de feature, target, train and test



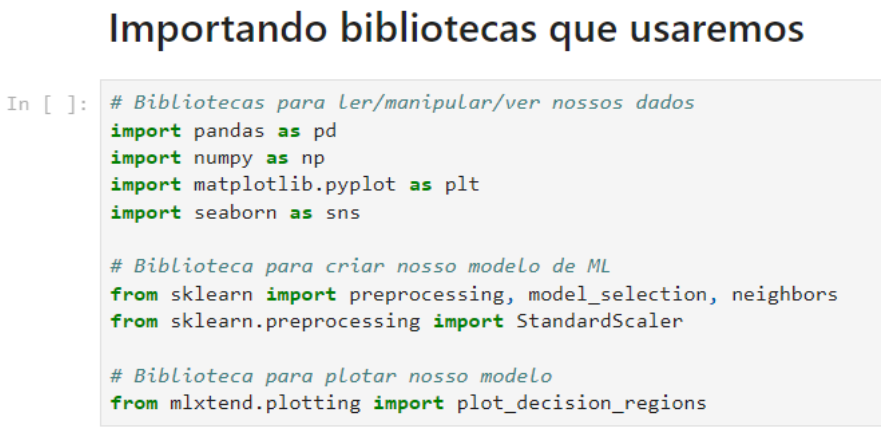
Fonte: Autores

3.4.3 Implementação com utilizações de bibliotecas

O código completo consta no Apêndice B - Algoritmo KNN com uso de biblioteca. Nos parágrafos a seguir será explorado cada parte do código bem como seus eventuais retornos e adendos. Após entender o conceito introdutório do tópico

anterior pode-se prosseguir para o código e inicialmente importar as bibliotecas. O código foi desenvolvido em um arquivo de extensão *.ipynb*, ou seja, utilizando a extensão do *Jupyter Notebook* no editor de código *Visual Studio Code*.

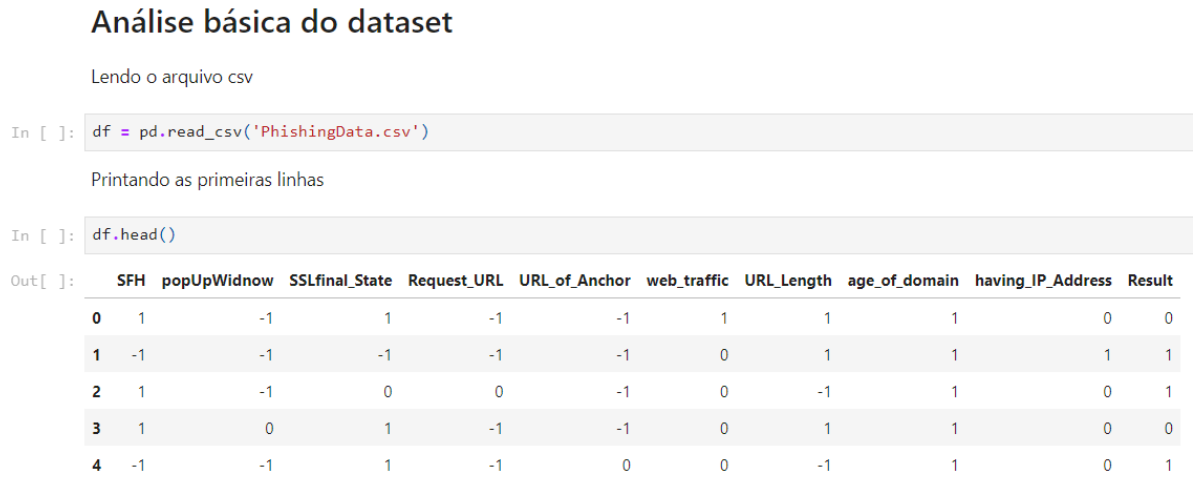
Figura 4 - Importação de bibliotecas



Fonte: Autores

Após a importação pode-se criar o *dataframe* com os dados *csv* bem como visualizar as 5 primeiras linhas para efeito de teste.

Figura 5 - Análise básica do dataset parte 1



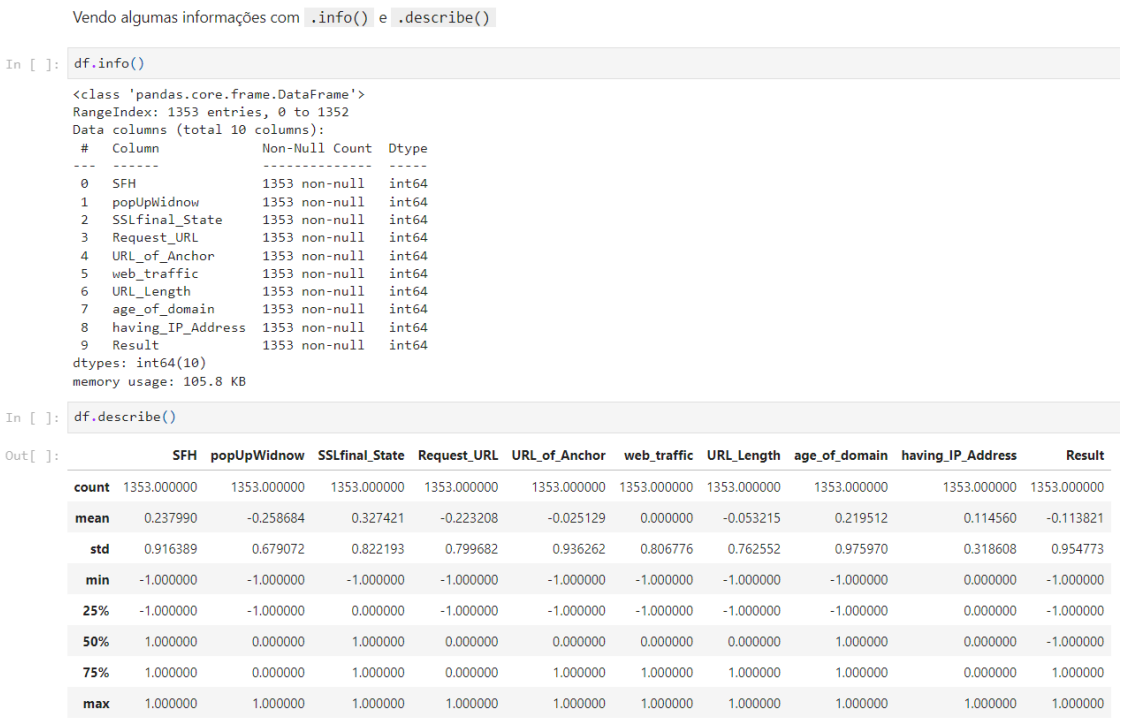
	SFH	popUpWidnow	SSLfinal_State	Request_URL	URL_of_Anchor	web_traffic	URL_Length	age_of_domain	having_IP_Address	Result
0	1	-1	1	-1	-1	1	1	1	0	0
1	-1	-1	-1	-1	-1	0	1	1	1	1
2	1	-1	0	0	-1	0	-1	1	0	1
3	1	0	1	-1	-1	0	1	1	0	0
4	-1	-1	1	-1	0	0	-1	1	0	1

Fonte: Autores

Pode-se também com a função da biblioteca pandas .info() analisar o tipo de dados de cada coluna *(atributo)* para confirmar que são valores inteiros. Pudemos perceber que estava tudo sobre controle, já que anteriormente o banco já foi categorizado em -1, 0 ou 1 que são realmente valores do tipo inteiro. Já com a

função *.describe()* alguns dados de pré-visualização foram explorados: média, quantidade de linhas, valores mínimos, valores máximos, etc.

Figura 6 - Análise básica do dataset parte 2



Vendo algumas informações com `.info()` e `.describe()`

In [ ]:
```
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 1353 entries, 0 to 1352
Data columns (total 10 columns):
 #   Column             Non-Null Count  Dtype
---  ------             --------------  -----
 0   SFH                1353 non-null   int64
 1   popUpWidnow        1353 non-null   int64
 2   SSLfinal_State     1353 non-null   int64
 3   Request_URL        1353 non-null   int64
 4   URL_of_Anchor      1353 non-null   int64
 5   web_traffic        1353 non-null   int64
 6   URL_Length         1353 non-null   int64
 7   age_of_domain      1353 non-null   int64
 8   having_IP_Address  1353 non-null   int64
 9   Result             1353 non-null   int64
dtypes: int64(10)
memory usage: 105.8 KB
```

In [ ]:
```
df.describe()
```

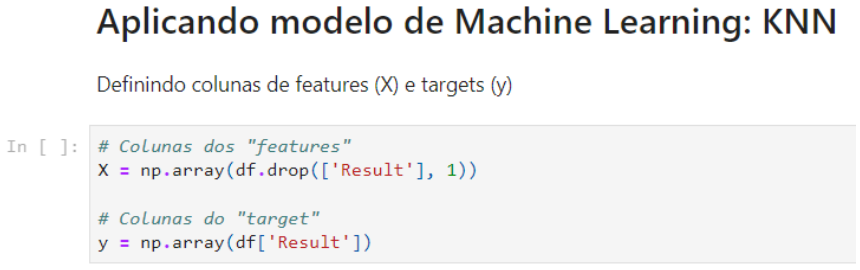
Out[ ]:

	SFH	popUpWidnow	SSLfinal_State	Request_URL	URL_of_Anchor	web_traffic	URL_Length	age_of_domain	having_IP_Address	Result
count	1353.000000	1353.000000	1353.000000	1353.000000	1353.000000	1353.000000	1353.000000	1353.000000	1353.000000	1353.000000
mean	0.237990	-0.258684	0.327421	-0.223208	-0.025129	0.000000	-0.053215	0.219512	0.114560	-0.113821
std	0.916389	0.679072	0.822193	0.799682	0.936262	0.806776	0.762552	0.975970	0.318608	0.954773
min	-1.000000	-1.000000	-1.000000	-1.000000	-1.000000	-1.000000	-1.000000	-1.000000	0.000000	-1.000000
25%	-1.000000	-1.000000	0.000000	-1.000000	-1.000000	-1.000000	-1.000000	-1.000000	0.000000	-1.000000
50%	1.000000	0.000000	1.000000	0.000000	0.000000	0.000000	0.000000	1.000000	0.000000	-1.000000
75%	1.000000	0.000000	1.000000	0.000000	1.000000	1.000000	1.000000	1.000000	0.000000	1.000000
max	1.000000	1.000000	1.000000	1.000000	1.000000	1.000000	1.000000	1.000000	1.000000	1.000000

Fonte: Autores

Para a utilização via *SKLearn*, o banco original foi utilizado sem fazer os *“replaces”* de nomenclatura para classes diferente do que fora feito anteriormente no algoritmo sem utilização da biblioteca. Pois ao utilizarmos a biblioteca Pandas criamos um *dataframe* com todos os dados e simplesmente excluímos com um ”*drop*” a coluna que não necessitamos. Isso pode ser verificado na figura 7 ao iniciarmos de fato a aplicação do modelo KNN. Ao excluirmos a coluna dos resultados obtivemos na variável X as nossas *features* (os atributos, os dados de entrada, os aspectos) e na variável y apenas a nossa *target* (coluna de resultado).

Figura 7 - Capturando as features e a target



Aplicando modelo de Machine Learning: KNN

Definindo colunas de features (X) e targets (y)

In [ ]:
```
# Colunas dos "features"
X = np.array(df.drop(['Result'], 1))

# Colunas do "target"
y = np.array(df['Result'])
```

Fonte: Autores

Após isso, enfim o modelo é aplicado como demonstra a figura 8 com comentário nas linhas de código mais importantes. Decidiu-se utilizar como métrica a "distância de *Manhattan*" e a quantidade de vizinhos k = 8. O motivo desta escolha será explicado nos próximos parágrafos.

Figura 8 - Finalmente treinando o modelo com base no KNN

Separando o Dataset em treino e teste Como os dados já foram categorizados anteriormente não há a necessidade de fazer a normalização Pois os dados já estão normalizados (os dados sempre são -1, 0 ou 1)

In [ ]:
```
X_train, X_test, y_train, y_test = model_selection.train_test_split(
                                                    X, y, test_size = 0.2, random_state=0)
```

Definindo nosso modelo (KNN)

In [ ]:
```
# clf = neighbors.KNeighborsClassifier(n_neighbors=3, metric='euclidean')
clf = neighbors.KNeighborsClassifier(n_neighbors=8, metric='manhattan')
#o motivo de utilizar-se o manhattan com k=8 estara nas linhas a seguir
```

Treinando nosso modelo

In [ ]:
```
clf.fit(X_train, y_train)
```

Out[ ]:
```
KNeighborsClassifier(metric='manhattan', n_neighbors=8)
```

Fonte: Autores

3.4.4 Otimização do valor de K e escolha da métrica ideal

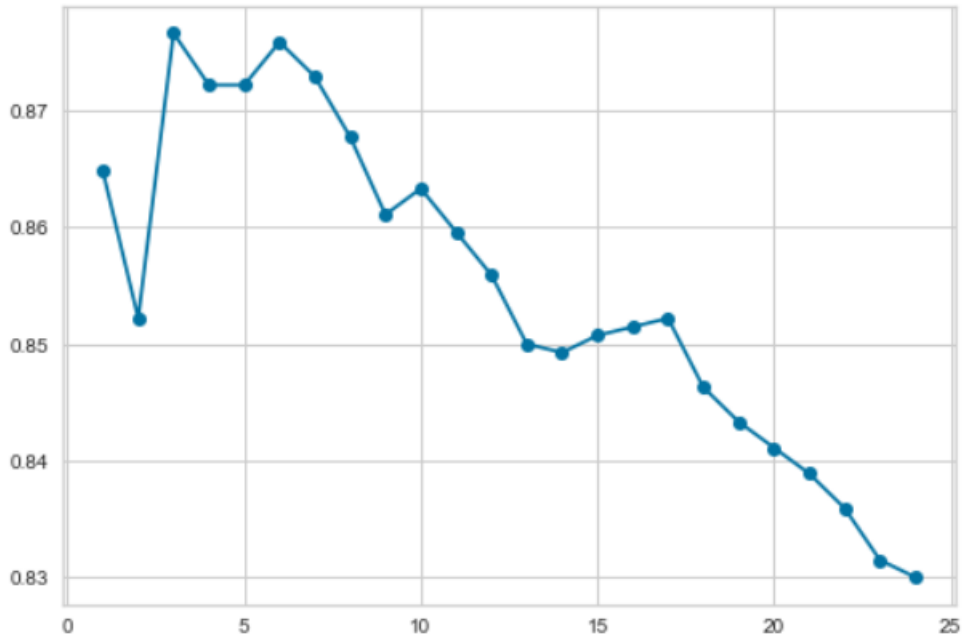
Para escolher um bom fator de K poderíamos intuitivamente com base no método "tentativa e erro" achar a melhor solução com base na acurácia de resultados. Se dos dados de teste uma porcentagem maior de acurácia fosse adquirida ao aumentar o K, então testaríamos K maiores. Se dos dados de teste uma maior porcentagem de acurácia fosse adquirida ao diminuir o K, então testaríamos K menores. Poderíamos repetir esse processo até encontrar um K ideal. O K ideal é aquele que satisfaz a maior parte possível dos dados de teste considerando a métrica de distância implementada.

Todavia a alternativa escolhida foi outra mais eficaz: utilizar a função *GridSearchCV* da biblioteca *SKLearn*. Desta forma é possível obter um gráfico com base em uma lista de parâmetros K e métricas correspondentes onde o eixo das abcissas representava o valor de K e o eixo das ordenadas a acurácia em porcentagem. As próximas figuras, mostram o código comentado em *Python* e a figura gerada da Acurácia x Valor de K para métrica "distância euclidiana", "distância de hamming" e "distância de matching" consecutivamente. Pôde-se observar que os melhores K para cada uma dessas métricas foram k=3, k=7 e k=8 respectivamente.

Figura 9 - Otimização de K para "distância euclidiana"

In [ ]:
```
#incluindo mais uma função do sklearn
from sklearn.model_selection import GridSearchCV

#criando uma lista de 1 a 25 para testes de possiveis 'k'
#utilizando métrica da distancia euclidiana
k_list = list(range(1,25))
metric_list = [ 'euclidean']
parameters = {'n_neighbors':k_list,'metric':metric_list}

#aqui é testado os parametros com base no score de acurácia
grid = GridSearchCV(clf,parameters,cv=5,scoring='accuracy')
grid.fit(X,y)

#obtedo o resultado e armazenando em um dataframe
scores = pd.DataFrame(grid.cv_results_)

#plotando no gráfico para visualização
plt.plot(k_list,scores['mean_test_score'],marker='o')
```

Out[ ]:
```
[<matplotlib.lines.Line2D at 0x1cdc1acdcd0>]
```

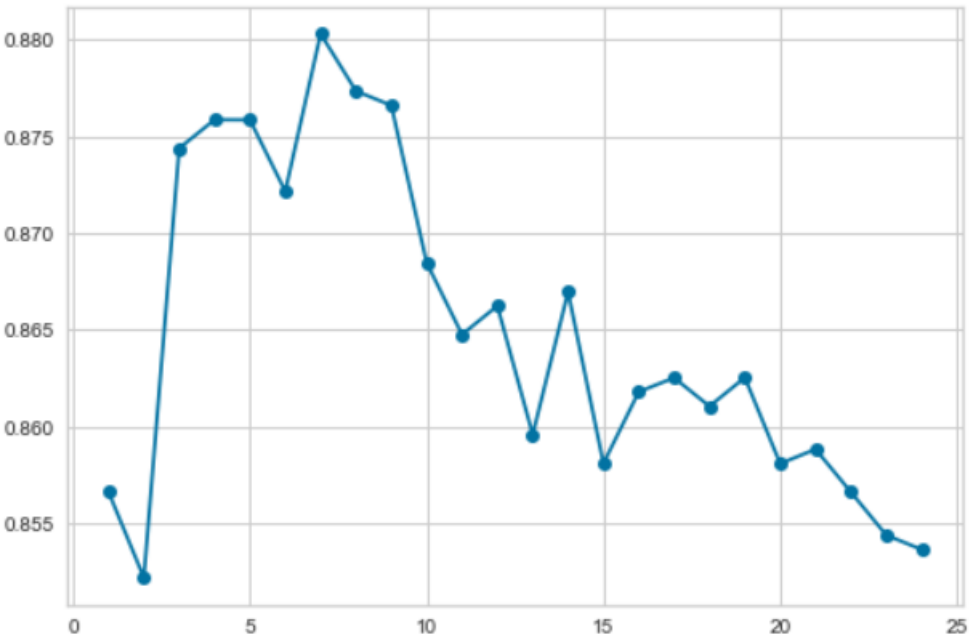


Fonte: Autores

Figura 10 - Otimização de K para "distância de hamming"

In [ ]:
```
#incluindo mais uma função do sklearn
from sklearn.model_selection import GridSearchCV

#criando uma lista de 1 a 25 para testes de possiveis 'k'
#utilizando métrica de distancia de hamming
k_list = list(range(1,25))
metric_list = ['hamming']
parameters = {'n_neighbors':k_list,'metric':metric_list}

#aqui é testado os parametros com base no score de acurácia
grid = GridSearchCV(clf,parameters,cv=5,scoring='accuracy')
grid.fit(X,y)

#obtedo o resultado e armazenando em um dataframe
scores = pd.DataFrame(grid.cv_results_)

#plotando no gráfico para visualização
plt.plot(k_list,scores['mean_test_score'],marker='o')
```

Out[ ]:
```
[<matplotlib.lines.Line2D at 0x1cdc1b53a60>]
```

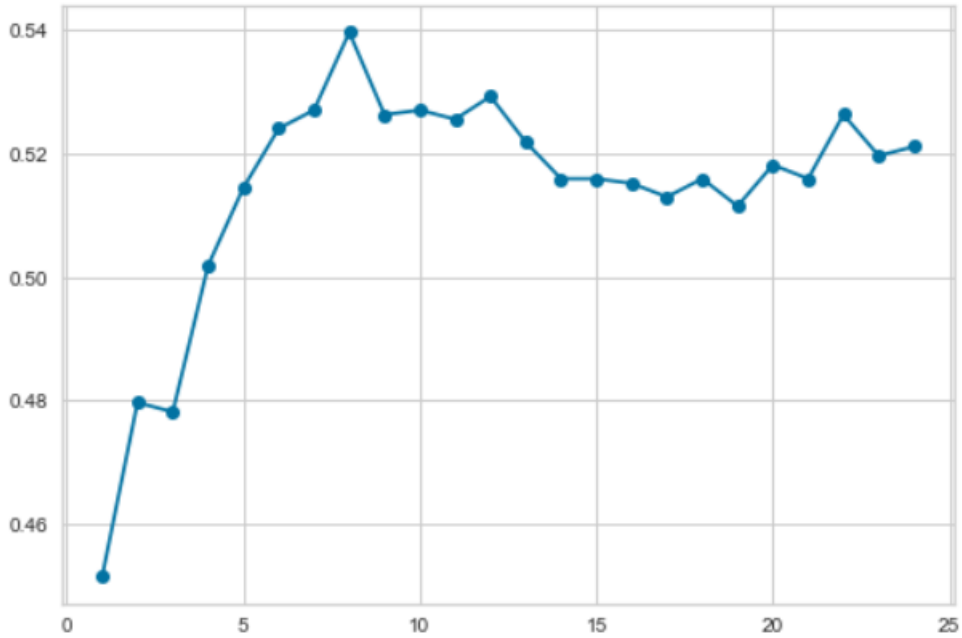


Fonte - Autores

Figura 11 - Otimização de K para "distância de matching"

In [ ]:
```
#incluindo mais uma função do sklearn
from sklearn.model_selection import GridSearchCV

#criando uma lista de 1 a 25 para testes de possiveis 'k'
#utilizando métrica de distancia de matching
k_list = list(range(1,25))
metric_list = ['matching']
parameters = {'n_neighbors':k_list,'metric':metric_list}

#aqui é testado os parametros com base no score de acurácia
grid = GridSearchCV(clf,parameters,cv=5,scoring='accuracy')
grid.fit(X,y)

#obtedo o resultado e armazenando em um dataframe
scores = pd.DataFrame(grid.cv_results_)

#plotando no gráfico para visualização
plt.plot(k_list,scores['mean_test_score'],marker='o')
```

Out[ ]:
```
[<matplotlib.lines.Line2D at 0x1cdc1bb3a30>]
```



Fonte: Autores

Percebe-se que “distância euclidiana” e “distância de *hamming*” retornaram melhores resultados. Já a “distância de *matching*” teve resultados inferiores. Para escolher a métrica ideal poderíamos simplesmente considerar a que mais faz sentido para o tipo de dado que temos no *dataset* considerando: se são categorizados, se são dados biológicos, se são binários ou valores que variam muito em termos de escala, etc. Entretanto, um modo excelente para adquirir a melhor métrica pode ser apenas adicionar várias métricas possíveis nos parâmetros de entrada da função

GridSearchCV que pode analisar um intervalo de valores para K para uma lista possível de métricas. Neste caso não foi um gráfico o retorno que nos foi apresentado, mas sim a própria “solução ótima”. Ao fazer essa última análise, percebeu-se que a melhor configuração a ser utilizada para o *dataset* escolhido seria k=8 sob a métrica “distância de *manhattan*”.

Figura 12 - Melhor valor de K e melhor métrica

```
In [ ]: #portanto dá pra fazer o teste
        #das duas coisas ao mesmo tempo
        #para saber qual K e qual metrica trará o melhor resultado

        #gerando lista com várias métricas para teste
        metric_list = [ 'euclidean',
                        'hamming',
                        'matching',
                        'minkowski',
                        'chebyshev',
                        'manhattan',
                        'jaccard',
                        'dice',
                        'kulsinski',
                        'rogerstanimoto',
                        'russellrao',
                        'sokalmichener',
                        'sokalsneath',
                        'braycurtis',
                        'canberra']

        #gerando lista com k de 1 ate 25
        k_list = list(range(1,25))

        #gerando o dicionario com todos os parametros que deseja testar entre si
        parameters = {'n_neighbors':k_list,'metric':metric_list}

        #aqui mudamos o cv de 5 (que é o padrão) para 10
        #na tentativa e erro percebeu-se uma melhora quando cv = 10
        #por meio da estratégia Kfold o dataset é dividido entre 10 partes
        #para fazer 10 testes diferentes
        #entre os dados de treino e os dados de teste
        #baseando-se sempre na acurácia
        grid = GridSearchCV(clf,parameters,cv=10,scoring='accuracy')
        grid.fit(X,y) #treinando
        scores = pd.DataFrame(grid.cv_results_) #capturando os score em um dataframe

        # capturando logo o melhor parametro ja que o gráfico agora
        # não poderá ser gerado por excesso de dados
        print("Melhor solução: "+ str(grid.best_params_))
```

```
Melhor solução: {'metric': 'manhattan', 'n_neighbors': 8}
```

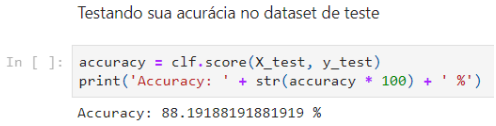
Fonte: Autores

4. RESULTADOS OBTIDOS

Com a obtenção do melhor valor de K e a melhor métrica, pudemos realizar o levantamento de alguns dados estatísticos para compreender o resultado que o algoritmo nos trouxe considerando o *dataset* escolhido.

A figura a seguir mostra o código realizado para adquirir a acurácia do nosso modelo bem como o valor de mesma (88.19%).

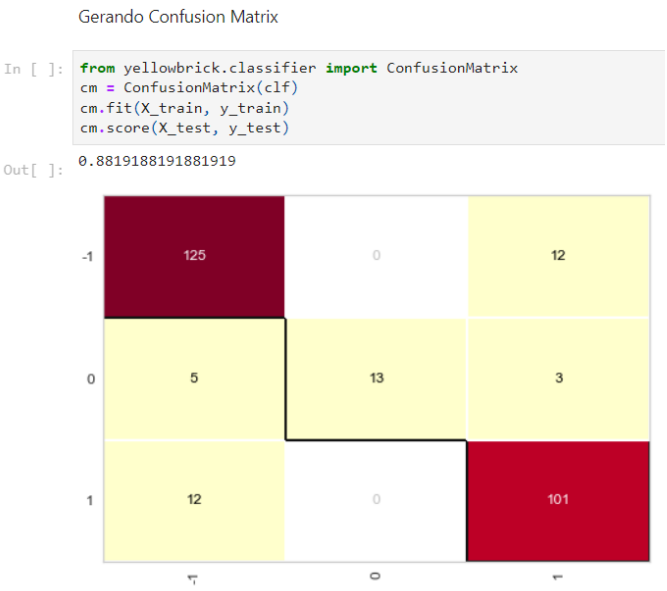
Figura 13 - Acurácia (em porcentagem) do modelo para o dataset escolhido



Fonte: Autores

A seguir a geração da matriz de confusão pôde nos esclarecer mais alguns pontos importantes. A diagonal principal da matriz nos mostrou novamente que o modelo teve bons resultados, a grande parte dos testes tiveram êxito. Quando um resultado era pra ser "*Phishing*" em 125 vezes ele acertou, em 12 vezes ele errou achando que tratava-se de um site "Legítimo" e em 5 vezes ele achou que o site poderia ser somente "Suspeito". Já quando o site era somente "Suspeito" ele acertou todas as vezes. Por fim, nos resultados para sites na categoria "Legítimo" ele acertou 101 vezes a classe desejada, errou 12 vezes achando que se tratava de site "*Phishing*" e 3 vezes achando que se tratava de site apenas "Suspeito". Essas informações e o código para gerá-las estão na figura a seguir.

Figura 14 - Matriz de confusão



Fonte: Autores

Por fim, mas não menos importante, foi gerado um relatório de classificação para enriquecermos ainda mais nossos dados estatísticos e para respaldar o bom resultado que o modelo KNN retornou para o *dataset* escolhido.

Com o relatório de classificação, pôde-se observar que o algoritmo conseguiu detectar 91% dos sites que eram “Phishing”, porém para estes sites detectados ele acertou 88% das vezes a classificação. Já no caso dos sites “suspeitos” ele conseguiu encontrar 62% dos sites, porém com a precisão de 100%. Outrossim, ele detectou 89% dos sites que eram legítimos e acertou em 87% das vezes que classificou assim. Estas informações e as linhas de código que as geraram constam na figura a seguir.

Figura 15 - Relatório de classificação

Gerando a "Classification Report"

```
In [ ]: from sklearn.metrics import classification_report
previsoes_classificadas = clf.predict(X_test)
print(classification_report(y_test, previsoes_classificadas))
```

```
              precision    recall  f1-score   support

          -1       0.88      0.91      0.90       137
           0       1.00      0.62      0.76        21
           1       0.87      0.89      0.88       113

    accuracy                           0.88       271
   macro avg       0.92      0.81      0.85       271
weighted avg       0.89      0.88      0.88       271
```

Fonte: Autores

5 CONCLUSÃO

Como descrito nos tópicos anteriores, o método de *Machine Learning* escolhido foi o do tipo K-NN por apresentar bons resultados e em com baixa carga de processamento, o que a princípio foi executado em um código independente utilizando o método de métrica por distância euclidiana.

Com código independente foram realizados testes com 3 linhas de dados previamente retirados do *dataset* a fim de não estarem presentes no treinamento. No caso o algoritmo conseguiu classificar corretamente os dados selecionados.

Todavia a amostra de testes se apresentava um tanto pobre. Além disso, havia um grande questionamento sobre o método de métrica euclidiana ser o mais eficiente para um *dataset* categorizado. Tais situações serviram de incentivo para os testes aprofundados e baseados em estatística com o uso de algumas

bibliotecas para *Python* com destaque para a mais explorada neste trabalho: a biblioteca *SKLearn*.

Tais testes aprofundados com o uso da biblioteca apresentaram resultados de grande valia, uma vez que foi possível comparar o aprendizado e a assertividade entre os métodos de métrica e a amplitude de vizinhos (k) por métrica.

Utilizando cerca de 80% do *dataset* como treinamento e os 20% restantes como teste observou-se através do algoritmo de comparação entre métricas e vizinhos que o melhor resultado foi utilizando a métrica *Manhattan* com 8 vizinhos.

Com tal configuração a assertividade ficou próximo a 89%, o que indica um resultado satisfatório, tendo em vista que os algoritmos de treinamento supervisionados são suscetíveis a *overfitting* e *underfitting*, de tal modo que taxas de assertividade muito elevadas indicam um *overfitting* e muito baixas o *underfitting*. Tais conceitos foram explorados também neste trabalho.

Com a geração de matriz de confusão podemos concluir que sites “*Phishing*”, quando detectados de maneira errada, eram na maioria das vezes classificados como a outra extrema classificação de sites “Legítimos”, mas raramente como apenas “Suspeitos”. O contrário também se notou, pois, sites erroneamente detectados como “Legítimos” na maioria das vezes eram “*Phishing*” e raramente “Suspeitos”. Portanto, genericamente, ou o algoritmo acertava consideravelmente bem ou errava drasticamente mal, raramente ficava no meio termo, provavelmente por algum atributo que, pela maneira de categorização, acabava por gerar tal extremismo ao modelo aplicado. Já quando o site era apenas “Suspeito” o algoritmo obteve 100% de precisão, isso é fato, provavelmente pela boa categorização e padronização prévia de valores dos atributos.

Destarte, o notório resultado também pôde ser comprovado no relatório de classificação que confirmou tais conclusões satisfatórias sob a classificação de um site em “*Phishing*”, “Legítimo” ou “Suspeito”.

REFERÊNCIAS BIBLIOGRÁFICAS

ABDELHAMID et al.,(2014a) **Phishing Detection based Associative Classification Data Mining**. Expert Systems With Applications (ESWA), 41 (2014).

DUA, D. and Graff, C. (2019). **UCI Machine Learning Repository**. Irvine, CA: University of California, School of Information and Computer Science. Disponível em: http://archive.ics.uci.edu/ml

IMANDOUST, Sadegh Bafandeh; BOLANDRAFTAR, Mohammad. **Application of k-nearest neighbor (knn) approach for predicting economic events: Theoretical background.** International Journal of Engineering Research and Applications, v. 3, n. 5, p. 605-610, 2013.

PRATIWI, M. E., T. A. LOROSAE, and F. W. Wibowo. **"Phishing site detection analysis using artificial neural network."** Journal of Physics: Conference Series. Vol. 1140. No. 1. IOP Publishing, 2018. Disponivel em: https://iopscience.iop.org/article/10.1088/1742-6596/1140/1/012048/pdf

SOLUTIONS, Stefanini It. **As 7 principais aplicações de inteligência artificial nas empresas**. Jaguariúna: Stefanini Group, 2022. Disponível em: https://stefanini.com/pt-br/trends/artigos/as-7-principais-aplicacoes-de-inteligencia-artificial-nas-empres#:~:text=Hoje%20em%20dia%2C%20%C3%A9%20poss%C3%ADvel,a%20produtividade%20e%20economizar%20tempo. Acesso em: 02 set. 2022.

BITCOIN, Mercado. **Brasil registra aumento de 80% nas tentativas de ataques de phishing**. São Paulo: Valor Econômico, 2022. Disponível em: https://valor.globo.com/patrocinado/mercado-bitcoin/noticia/2022/07/07/brasil-registra-aumento-de-80percent-nas-tentativas-de-ataques-de-phishing.ghtml. Acesso em: 02 set. 2022.

INC, Sas Institute. **Machine Learning**: o que é e qual sua importância?. São Paulo: Sas, 2022. Disponível em: https://www.sas.com/pt_br/insights/analytics/machine-learning.html#:~:text=O%20aprendizado%20de%20m%C3%A1quina%20(em,o%20m%C3%ADnimo%20de%20interven%C3%A7%C3%A3o%20humana. Acesso em: 02 set. 2022.

APÊNDICE A – Algoritmo KNN sem uso de biblioteca

09/09/2022 16:36 main.py

```python
from dado import dado
from knn import knn

if __name__ == '__main__':
    dataset_train=[]
    # Abertura e leitura do dataset.
    with open('.\\PhishingData.txt') as arquivo:
        for linha in arquivo.readlines():
            atributos = linha.rstrip().split(',')
            classe = atributos[-1]
            atributos = list(map(float, atributos[:-1]))
            dado_arquivo = dado(atributos, classe)
            dataset_train.append(dado_arquivo)
    arquivo.close()
    # Dados do input a ser classificado.
    entrada_verif = [
        [0,-1,0,-1,-1,1,-1,-1,0], '',
        [1,0,1,-1,-1,0,1,1,1], '',
        [1,0,-1,0,-1,1,1,-1,0], '' ]
    #entrada_verif = dado([7.6, 3, 6.6, 2.1], '')
    kneighbor = knn(3, dataset_train)
    result = 'Os resultados são: '
    for i in range(0,len(entrada_verif),2):
        t_data = dado(entrada_verif[i], entrada_verif[i+1])
        kneighbor.executar(t_data)
        result += t_data.get_classe() + ', '

    print(result)
```

```python
from sqlalchemy.sql.elements import conv

from dado import dado
from typing import List
import math


class knn:
    # Receber um dado não classificado (z),
    # o conjunto de dados classificados (X)
    # e o número de vizinhos (k);
    def __init__(self, k: int, conjunto_dados: List[dado]):
        self.__k__ = k
        self.__conjunto_dados__ = conjunto_dados

    def __metric_eclid__(self, new_dado: List[float], dado_clf: List[float]):
        somatorio: float = 0
        for carac_new_dado, carac_dado_clf in zip(new_dado, dado_clf):
            somatorio += math.pow((carac_new_dado - carac_dado_clf), 2)
        return math.sqrt(somatorio)

    def __next_to__(self):
        cnj_ordem = sorted(self.__conjunto_dados__, key=dado.get_distancia)
        print(cnj_ordem)
        return cnj_ordem[:self.__k__]

    def __most_rec__(self, dados_mais_proximos: List[dado]):
        dic_class = {}
        for close_data in dados_mais_proximos:
            if close_data.get_classe() in dic_class:
                dic_class[close_data.get_classe()] += 1
            else:
                dic_class[close_data.get_classe()] = 1
        return max(dic_class, key=dic_class.get)

    def executar(self, new_dado: dado):
        # Medir a distância de z para cada dado que já classificado;
        for dado_clf in self.__conjunto_dados__:
            distancia = self.__metric_eclid__(
                new_dado.get_atributos(), dado_clf.get_atributos())
            dado_clf.set_distancia(distancia)
        # Obter as k menores distâncias
        dados_mais_proximos = self.__next_to__()
        # Verificar a classe de cada um dos dados k dados de menor distância
        # e contar a quantidade de vezes que cada classe que aparece
        # Receber como resultado a classe mais recorrente
        classe = self.__most_rec__(dados_mais_proximos)
        # Classificar o novo dado com a classe mais recorrente
        new_dado.set_classe(classe)

```

```python
from typing import List

class dado:
    def __init__(self, atributos: List[float], classe: str):
        self.__atributos__ = atributos
        self.__classe__ =classe
        self.__distancia__: float = 0

    def get_atributos(self):
        return self.__atributos__

    def get_classe(self):
        return self.__classe__

    def get_distancia(self):
        return self.__distancia__

    def set_classe(self, classe:str):
        self.__classe__=classe

    def set_distancia(self, distancia:float):
        self.__distancia__ = distancia

```

APÊNDICE B – Algoritmo KNN com uso de biblioteca

Importando bibliotecas que usaremos

In [ ]:
```
# Bibliotecas para ler/manipular/ver nossos dados
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns

# Biblioteca para criar nosso modelo de ML
from sklearn import preprocessing, model_selection, neighbors
from sklearn.preprocessing import StandardScaler

# Biblioteca para plotar nosso modelo
from mlxtend.plotting import plot_decision_regions
```

Análise básica do dataset

Lendo o arquivo csv

In [ ]:
```
df = pd.read_csv('PhishingData.csv')
```

Printando as primeiras linhas

In [ ]:
```
df.head()
```

Out[ ]:

	SFH	popUpWidnow	SSLfinal_State	Request_URL	URL_of_Anchor	web_traffic	URL_Length	age_of_domain	having_IP_Address	Result
0	1	-1	1	-1	-1	1	1	1	0	0
1	-1	-1	-1	-1	-1	0	1	1	1	1
2	1	-1	0	0	-1	0	-1	1	0	1
3	1	0	1	-1	-1	0	1	1	0	0
4	-1	-1	1	-1	0	0	-1	1	0	1

Vendo algumas informações com `.info()` e `.describe()`

In [ ]:
```
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 1353 entries, 0 to 1352
Data columns (total 10 columns):
 #   Column             Non-Null Count  Dtype
---  ------             --------------  -----
 0   SFH                1353 non-null   int64
 1   popUpWidnow        1353 non-null   int64
 2   SSLfinal_State     1353 non-null   int64
 3   Request_URL        1353 non-null   int64
 4   URL_of_Anchor      1353 non-null   int64
 5   web_traffic        1353 non-null   int64
 6   URL_Length         1353 non-null   int64
 7   age_of_domain      1353 non-null   int64
 8   having_IP_Address  1353 non-null   int64
 9   Result             1353 non-null   int64
dtypes: int64(10)
memory usage: 105.8 KB
```

In [ ]:
```
df.describe()
```

Out[ ]:

	SFH	popUpWidnow	SSLfinal_State	Request_URL	URL_of_Anchor	web_traffic	URL_Length	age_of_domain	having_IP_Address	Result
count	1353.000000	1353.000000	1353.000000	1353.000000	1353.000000	1353.000000	1353.000000	1353.000000	1353.000000	1353.000000
mean	0.237990	-0.258684	0.327421	-0.223208	-0.025129	0.000000	-0.053215	0.219512	0.114560	-0.113821
std	0.916389	0.679072	0.822193	0.799682	0.936262	0.806776	0.762552	0.975970	0.318608	0.954773
min	-1.000000	-1.000000	-1.000000	-1.000000	-1.000000	-1.000000	-1.000000	-1.000000	0.000000	-1.000000
25%	-1.000000	-1.000000	0.000000	-1.000000	-1.000000	-1.000000	-1.000000	-1.000000	0.000000	-1.000000
50%	1.000000	0.000000	1.000000	0.000000	0.000000	0.000000	0.000000	1.000000	0.000000	-1.000000
75%	1.000000	0.000000	1.000000	0.000000	1.000000	1.000000	1.000000	1.000000	0.000000	1.000000
max	1.000000	1.000000	1.000000	1.000000	1.000000	1.000000	1.000000	1.000000	1.000000	1.000000

Aplicando modelo de Machine Learning: KNN

Definindo colunas de features (X) e targets (y)

In [ ]:
```
# Colunas dos "features"
X = np.array(df.drop(['Result'], 1))

# Colunas do "target"
y = np.array(df['Result'])
```

```
C:\Users\Lenovo\AppData\Local\Temp\ipykernel_25536\146368618.py:2: FutureWarning: In a future version of pandas all arguments of DataFrame.drop except for the argument 'labels' will b
nly.
  X = np.array(df.drop(['Result'], 1))
```

Separando o Dataset em treino e teste Como os dados já foram categorizados anteriormente não há a necessidade de fazer a normalização Pois os dados já estão normalizados (os dados sempre são -1, 0 ou 1)

In [ ]:
```
X_train, X_test, y_train, y_test = model_selection.train_test_split(
                                                    X, y, test_size = 0.2, random_state=0)
```

Definindo nosso modelo (KNN)

In [ ]:
```
# clf = neighbors.KNeighborsClassifier(n_neighbors=3, metric='euclidean')
clf = neighbors.KNeighborsClassifier(n_neighbors=8, metric='manhattan')
#o motivo de utilizar-se o manhattan com k=8 estara nas linhas a seguir
```

Treinando nosso modelo

In [ ]:
```
clf.fit(X_train, y_train)
```

Out[ ]:
```
KNeighborsClassifier(metric='manhattan', n_neighbors=8)
```

Testando sua acurácia no dataset de teste

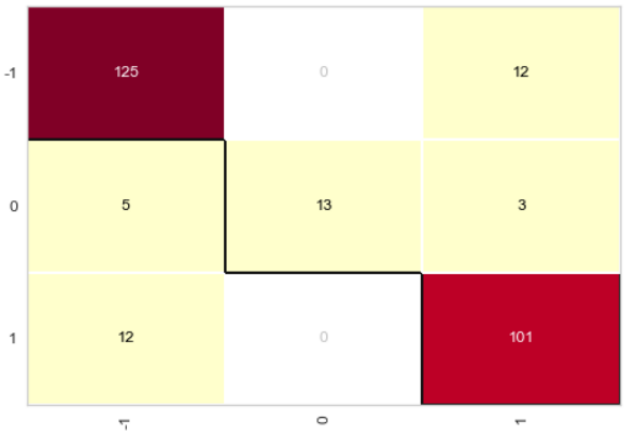
In [ ]:
```
accuracy = clf.score(X_test, y_test)
print('Accuracy: ' + str(accuracy * 100) + ' %')
```

```
Accuracy: 88.19188191881919 %
```

Gerando Confusion Matrix

In [ ]:
```
from yellowbrick.classifier import ConfusionMatrix
cm = ConfusionMatrix(clf)
cm.fit(X_train, y_train)
cm.score(X_test, y_test)
```

Out[ ]:
```
0.8819188191881919
```



Gerando a "Classification Report"

In [ ]:
```
from sklearn.metrics import classification_report
previsoes_classificadas = clf.predict(X_test)
print(classification_report(y_test, previsoes_classificadas))
```

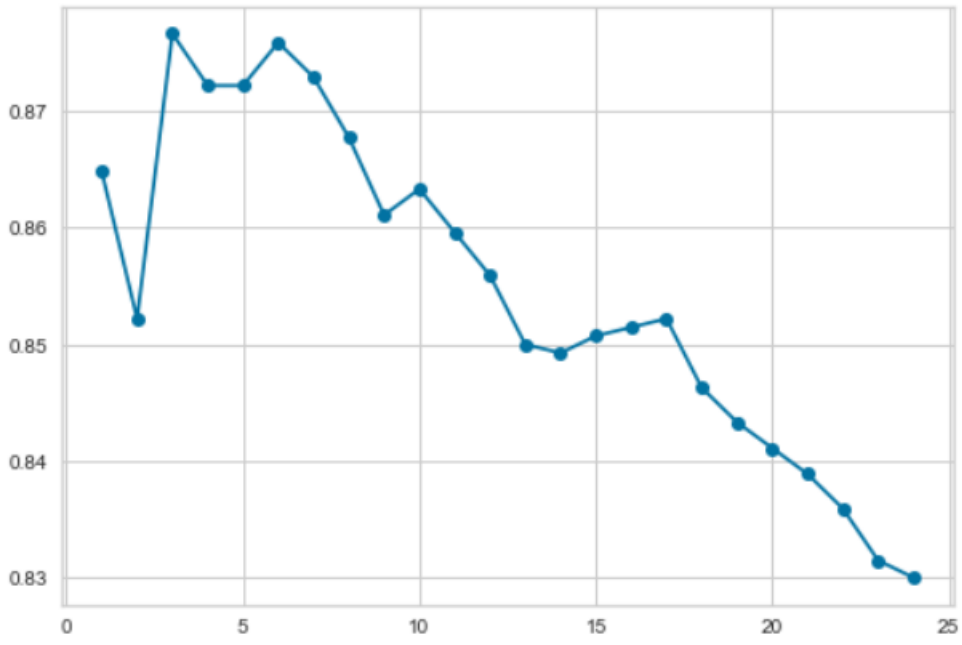
```
              precision    recall  f1-score   support

          -1       0.88      0.91      0.90       137
           0       1.00      0.62      0.76        21
           1       0.87      0.89      0.88       113

    accuracy                           0.88       271
   macro avg       0.92      0.81      0.85       271
weighted avg       0.89      0.88      0.88       271
```

Otimizando os parâmetros de treinamento

- O valor de 'k' ideal
- A métrica ideal a ser utilizada

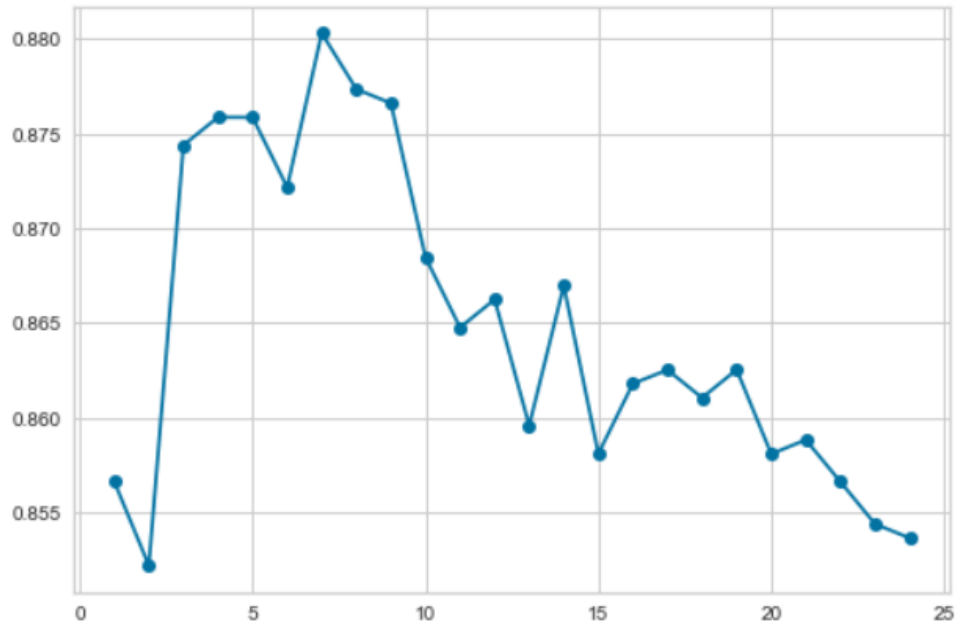
In [ ]:
```
#incluindo mais uma função do sklearn
from sklearn.model_selection import GridSearchCV

#criando uma lista de 1 a 25 para testes de possiveis 'k'
#utilizando métrica da distancia euclidiana
k_list = list(range(1,25))
metric_list = [ 'euclidean']
parameters = {'n_neighbors':k_list,'metric':metric_list}

#aqui é testado os parametros com base no score de acurácia
grid = GridSearchCV(clf,parameters,cv=5,scoring='accuracy')
grid.fit(X,y)

#obtedo o resultado e armazenando em um dataframe
scores = pd.DataFrame(grid.cv_results_)

#plotando no gráfico para visualização
plt.plot(k_list,scores['mean_test_score'],marker='o')
```

Out[ ]:
```
[<matplotlib.lines.Line2D at 0x1cdc1acdcd0>]
```



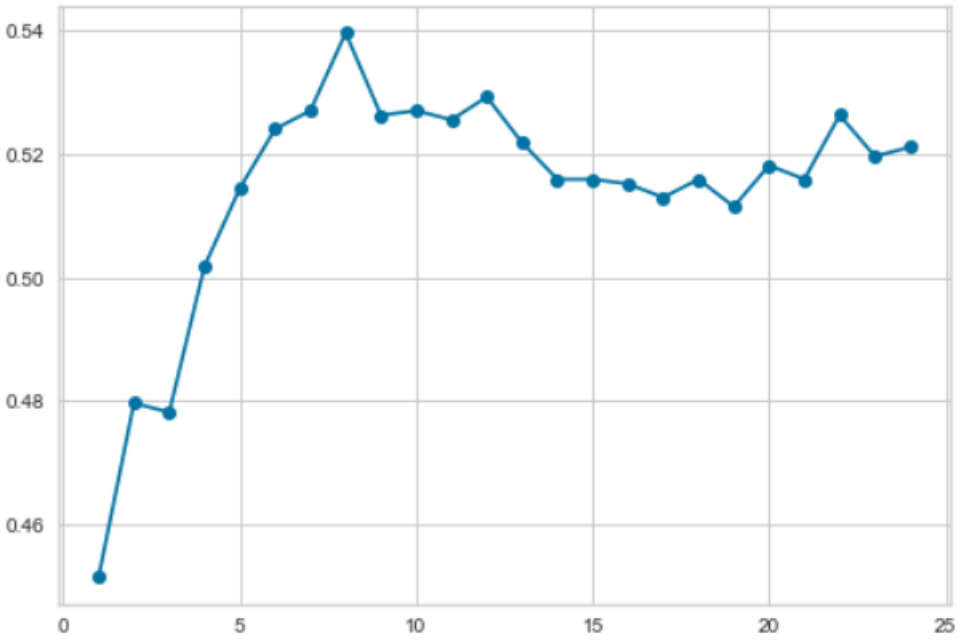
In [ ]:
```
#incluindo mais uma função do sklearn
from sklearn.model_selection import GridSearchCV

#criando uma lista de 1 a 25 para testes de possiveis 'k'
#utilizando métrica de distancia de hamming
k_list = list(range(1,25))
metric_list = ['hamming']
parameters = {'n_neighbors':k_list,'metric':metric_list}

#aqui é testado os parametros com base no score de acurácia
grid = GridSearchCV(clf,parameters,cv=5,scoring='accuracy')
grid.fit(X,y)

#obtedo o resultado e armazenando em um dataframe
scores = pd.DataFrame(grid.cv_results_)

#plotando no gráfico para visualização
plt.plot(k_list,scores['mean_test_score'],marker='o')
```

Out[ ]:
```
[<matplotlib.lines.Line2D at 0x1cdc1b53a60>]
```



In [ ]:
```
#incluindo mais uma função do sklearn
from sklearn.model_selection import GridSearchCV

#criando uma lista de 1 a 25 para testes de possiveis 'k'
#utilizando métrica de distancia de matching
k_list = list(range(1,25))
metric_list = ['matching']
parameters = {'n_neighbors':k_list,'metric':metric_list}

#aqui é testado os parametros com base no score de acurácia
grid = GridSearchCV(clf,parameters,cv=5,scoring='accuracy')
grid.fit(X,y)

#obtedo o resultado e armazenando em um dataframe
scores = pd.DataFrame(grid.cv_results_)

#plotando no gráfico para visualização
plt.plot(k_list,scores['mean_test_score'],marker='o')
```

Out[ ]:
```
[<matplotlib.lines.Line2D at 0x1cdc1bb3a30>]
```



In [ ]:
```
#portanto dá pra fazer o teste
#das duas coisas ao mesmo tempo
#para saber qual K e qual metrica trará o melhor resultado

#gerando lista com várias métricas para teste
metric_list = [ 'euclidean',
                'hamming',
                'matching',
                'minkowski',
                'chebyshev',
                'manhattan',
                'jaccard',
                'dice',
                'kulsinski',
                'rogerstanimoto',
                'russellrao',
                'sokalmichener',
                'sokalsneath',
                'braycurtis',
                'canberra']

#gerando lista com k de 1 ate 25
k_list = list(range(1,25))

#gerando o dicionario com todos os parametros que deseja testar entre si
parameters = {'n_neighbors':k_list,'metric':metric_list}

#aqui mudamos o cv de 5 (que é o padrão) para 10
#na tentativa e erro percebeu-se uma melhora quando cv = 10
#por meio da estratégia Kfold o dataset é dividido entre 10 partes
#para fazer 10 testes diferentes
#entre os dados de treino e os dados de teste
#baseando-se sempre na acurácia
grid = GridSearchCV(clf,parameters,cv=10,scoring='accuracy')
grid.fit(X,y) #treinando
scores = pd.DataFrame(grid.cv_results_) #capturando os score em um dataframe

# capturando logo o melhor parametro ja que o gráfico agora
# não poderá ser gerado por excesso de dados
print("Melhor solução: "+ str(grid.best_params_))
```

```
Melhor solução: {'metric': 'manhattan', 'n_neighbors': 8}
```