UNIVERSIDADE FEDERAL DO RIO GRANDE DO NORTE
DEPARTAMENTO DE ENGENHARIA ELÉTRICA
ELE0606 - TÓPICOS ESPECIAIS EM INTELIGENCIA ARTIFICIAL

Docente:
José Alfredo Ferreira Costa

Discente:
Douglas Wilian Lima Silva
Semestre 2023.2 - Turma 01

Desenvolvimento do K-Nearest Neighbors na base Wine

Natal - RN
Setembro de 2023

Sumário

1 Apresentação

O presente trabalho tem por objetivo a aplicação do algoritmo K Vizinhos Mais Próximos (*K-Nearest Neighbors - KNN*) utilizando a base de dados *wine* disponível na Scikit-Learn. A ideia do projeto é justamente utilizar as inúmeras informações acerca dos vinhos, para que, utilizando o algoritmo de predição e classificação, seja possível associar cada vinho a sua classe determinada.

De forma geral, a classificação de padrões e a busca por semelhanças são tarefas fundamentais em muitos campos da ciência da computação e da análise de dados. O algoritmo KNN é uma técnica amplamente utilizada para resolver esses tipos de problemas. O objetivo principal do KNN é classificar um objeto desconhecido com base na maioria das classes dos K vizinhos mais próximos a ele em um espaço de atributos multidimensional.

A ideia de aplicação do algoritmo consiste justamente em agrupar termos de acordo com a distância euclidiana entre eles no conjunto de dados. Após escolher esse valor de K (número de elementos agrupados), o algoritmo atribui ao objeto desconhecido a classe que é mais comum entre esses vizinhos. Essa decisão é tomada por votação, onde cada vizinho contribui igualmente para a decisão final. O KNN é um algoritmo de aprendizado preguiçoso (lazy), o que significa que ele não cria um modelo explícito durante a fase de treinamento. Em vez disso, ele armazena todo o conjunto de dados de treinamento e realiza a classificação apenas quando um novo objeto precisa ser classificado.

De forma ilustrativa, podemos observar na imagem abaixo uma ideia de aplicação do algoritmo, em que ele usa essa distância entre os elementos mais próximos em uma tentativa de predição acerca da classe ou tipo do elemento desejado.

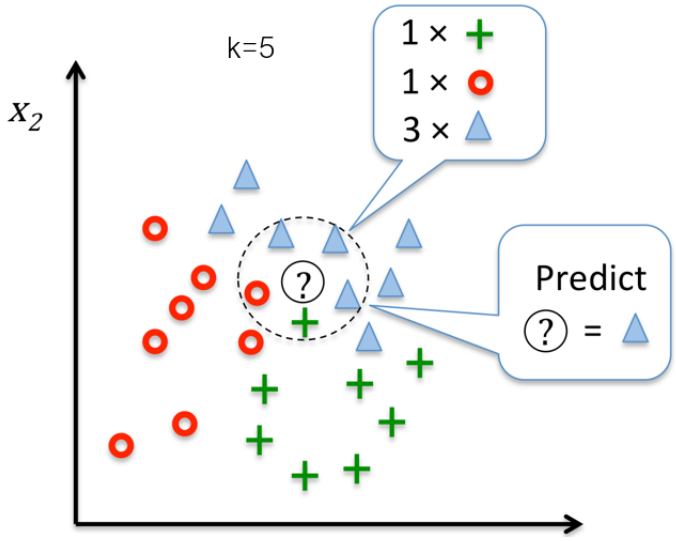

Figura 1: Exemplo ilustrativo.

Observa-se que, utilizando um agrupamento de 5 vizinhos, o algoritmo identificou que o número de elementos pertencentes à classe "triângulo"foi superior, fazendo com que

sua predição do elemento desejado informasse que ele provavelmente pertença realmente à essa classe.

2 Implementação

2.1 Processamento dos dados

A implementação desse algoritmo para a base dos vinhos será realizada de forma bem semelhante ao exemplo ilustrativo, no entanto, para que possamos classificar uma base de dados, é importante em primeiro lugar, explorar suas características e processá-las de forma correta.

A base de dados trabalhada é composta por 178 linhas e 14 colunas, em que em 13 dessas estão contidos diferentes parâmetros acercas do vinhos coletados como alcalinidade, nível de magnésio etc, e na coluna restante, a classificação acerca do tipo do vinho: tipo 1, 2 ou 3.

Classes	3
Samples per class	[59,71,48]
Samples total	178
Dimensionality	13
Features	real, positive

Figura 2: Wine - ScikitLearn.

Importando essa base de dados e gerando um dataframe usando a biblioteca Pandas, podemos observar toda a disposição dos elementos e os valores numéricos correspondentes a cada estrutura.

```
#Carregamento da base de dados

url = 'https://archive.ics.uci.edu/ml/machine-learning-databases/wine/
    wine.data'

# Ajustando os nomes das colunas
column_names = [
    'Class',
    'Alcohol',
    'Malic_acid',
    'Ash',
    'Alcalinity_of_ash',
    'Magnesium',
    'Total_phenols',
    'Flavanoids',
    'Nonflavanoid_phenols',
```

```
    'Proanthocyanins',
    'Color_intensity',
    'Hue',
    'OD280_OD315_of_diluted_wines',
    'Proline'
]
df = pd.read_csv(url, names = column_names)
```

Listing 1: Importação e visualização do dataframe.

	Class	Alcohol	Malic_acid	Ash	Alcalinity_of_ash	Magnesium	Total_phenols	Flavanoids	Nonflavanoid_phenols
0	1	14.23	1.71	2.43	15.6	127	2.80	3.06	0.28
1	1	13.20	1.78	2.14	11.2	100	2.65	2.76	0.26
2	1	13.16	2.36	2.67	18.6	101	2.80	3.24	0.30
3	1	14.37	1.95	2.50	16.8	113	3.85	3.49	0.24
4	1	13.24	2.59	2.87	21.0	118	2.80	2.69	0.39
...	...	...	...	...	...	...	...	...	...
173	3	13.71	5.65	2.45	20.5	95	1.68	0.61	0.52
174	3	13.40	3.91	2.48	23.0	102	1.80	0.75	0.43
175	3	13.27	4.28	2.26	20.0	120	1.59	0.69	0.43
176	3	13.17	2.59	2.37	20.0	120	1.65	0.68	0.53
177	3	14.13	4.10	2.74	24.5	96	2.05	0.76	0.56

178 rows × 14 columns

Figura 3: Visualização parcial dos dados.

2.2 Normalização

Tendo acesso ao dataframe, para que possamos trabalhar com o algoritmo de classificação, precisamos realizar a normalização dos dados de forma que todos possam se encontrar no mesmo *range* de 0 à 1, já que, observando os dados estudados a variação é bem elevada.

Existem diversos métodos de normalização de dataframes, usando valores máximos e mínimos, algo que pode ser realizado manualmente sem grandes problemas. No entanto, tendo em vista a facilidade e a diversidade da linguagem trabalhada, foi utilizado o método MinMaxScaler() presente na já citada Scikit-Learn. Assim, usando esse método, foi gerado um novo dataframe usando os valores normalizados.

```
from sklearn.preprocessing import MinMaxScaler

Zscore = MinMaxScaler()

#Dataframe normalizado
dfn = pd.DataFrame(Zscore.fit_transform(df), columns = df.columns)

display(dfn)
```

Listing 2: Normalização dos dados.

	Class	Alcohol	Malic_acid	Ash	Alcalinity_of_ash	Magnesium	Total_phenols	Flavanoids	Nonflavanoid_phenols
0	1	0.842105	0.191700	0.572193	0.257732	0.619565	0.627586	0.573840	0.283019
1	1	0.571053	0.205534	0.417112	0.030928	0.326087	0.575862	0.510549	0.245283
2	1	0.560526	0.320158	0.700535	0.412371	0.336957	0.627586	0.611814	0.320755
3	1	0.878947	0.239130	0.609626	0.319588	0.467391	0.989655	0.664557	0.207547
4	1	0.581579	0.365613	0.807487	0.536082	0.521739	0.627586	0.495781	0.490566
...	...	...	...	...	...	...	...	...	...
173	3	0.705263	0.970356	0.582888	0.510309	0.271739	0.241379	0.056962	0.735849
174	3	0.623684	0.626482	0.598930	0.639175	0.347826	0.282759	0.086498	0.566038
175	3	0.589474	0.699605	0.481283	0.484536	0.543478	0.210345	0.073840	0.566038
176	3	0.563158	0.365613	0.540107	0.484536	0.543478	0.231034	0.071730	0.754717
177	3	0.815789	0.664032	0.737968	0.716495	0.282609	0.368966	0.088608	0.811321

178 rows × 14 columns

Figura 4: Dataframe normalizado.

Observe que, embora todos os dados do dataframe foram normalizados, a coluna das classes deve se manter intacta, já que através dela que serão geradas as classificações. Assim, essa coluna do dataframe normalizado foi associada a coluna do original para que não existisse a normalização.

2.3 Grupos de desenvolvimento e teste

Usando como base a referência apresentada pelo professor da aplicação do KNN na base iris, foi possível realizar a separação do dataframe em duas classes de estudo: a de treinamento ou desenvolvimento, como chamada e a de testes.

Basicamente, realizar essa separação consiste em "*randomizar*"a base de dados de forma a remover a sequência do dataframe, para então, sortear valores aleatórios para que possam ser usados para o treinamento do algoritmo, enquanto os demais serão usados nos testes de predição.

Para tal, foi usada a biblioteca NumPy para gerar a aleatoriadade e então foram definidos, inicialmente que 60% do dataframe será utilizada para treinamento e o restante para teste.

```
indices = np.random.permutation(dfn.shape[0])
div = int(0.6*len(indices))
desen_id , test_id = indices[:div], indices[div:]

cj_desen, cj_test = dfn.loc[desen_id,:], dfn.loc[test_id,:]

y_d = cj_desen['Class']
y_t = cj_test['Class']
del cj_desen['Class']
del cj_test['Class']
```

Listing 3: Separação das classes.

Pode-se observar que as colunas de classificação dos vinhos foram removidas das classes de treinamento e teste e armazenadas nas variáveis y_d e y_t, respectivamente, de forma a manter a aleatoriedade, mas também a correlação entre os conjuntos. Assim, todas as informações aleatórias escolhidas, antes presentes no dataframe normalizado, estão contidas nos conjuntos cj_desen e cj_test.

	Alcohol	Malic_acid	Ash	Alcalinity_of_ash	Magnesium	Total_phenols	Flavanoids	Nonflavanoid_phenols
26	0.621053	0.203557	0.673797	0.283505	0.250000	0.644828	0.548523	0.396226
141	0.613158	0.359684	0.529412	0.484536	0.206522	0.144828	0.033755	0.452830
99	0.331579	0.480237	0.454545	0.381443	0.195652	0.644828	0.559072	0.603774
129	0.265789	0.703557	0.545455	0.587629	0.108696	0.386207	0.297468	0.547170
88	0.160526	0.260870	0.588235	0.567010	0.152174	0.334483	0.284810	0.660377
...	...	...	...	...	...	...	...	...
7	0.797368	0.278656	0.668449	0.360825	0.554348	0.558621	0.457806	0.339623
69	0.310526	0.088933	0.208556	0.319588	0.880435	0.300000	0.198312	0.018868
21	0.500000	0.604743	0.689840	0.412371	0.347826	0.493103	0.436709	0.226415
63	0.352632	0.077075	0.427807	0.432990	0.184783	0.868966	0.582278	0.113208
28	0.747368	0.229249	0.770053	0.453608	0.402174	0.679310	0.554852	0.452830

106 rows × 13 columns

Figura 5: Conjunto de treinamento.

	Alcohol	Malic_acid	Ash	Alcalinity_of_ash	Magnesium	Total_phenols	Flavanoids	Nonflavanoid_phenols
70	0.331579	0.171937	0.454545	0.505155	0.358696	0.041379	0.143460	0.452830
46	0.881579	0.563241	0.491979	0.278351	0.347826	0.782759	0.597046	0.264151
14	0.881579	0.223320	0.545455	0.072165	0.347826	0.800000	0.696203	0.301887
150	0.650000	0.470356	0.673797	0.690722	0.576087	0.144828	0.259494	0.169811
84	0.213158	0.029644	0.652406	0.381443	0.260870	0.420690	0.394515	0.169811
...	...	...	...	...	...	...	...	...
91	0.255263	0.152174	0.566845	0.587629	0.173913	0.162069	0.191983	0.698113
97	0.331579	0.132411	0.331551	0.278351	0.163043	0.541379	0.455696	0.301887
106	0.321053	0.195652	0.406417	0.432990	0.108696	0.231034	0.356540	0.452830
61	0.423684	0.122530	0.352941	0.319588	0.326087	0.358621	0.225738	0.754717
5	0.834211	0.201581	0.582888	0.237113	0.456522	0.789655	0.643460	0.396226

72 rows × 13 columns

Figura 6: Conjunto de testes.

2.4 Algoritmo KNN

Tendo todas as variáveis separadas, pode-se então partir para o desenvolvimento do algoritmo de classificação desejado. Para tal, foi utilizada novamente a biblioteca Scikit-Learn usando a implementação do método KNeighborsClassifier.

Através da biblioteca usada foi definida a função que recebe como parâmetros, os dados de treinamento: dados e classificação; os dados de testes e o agrupamento K, para que a partir desses dados ele possa classificar os dados de treinamento de acordo com a classe dos vinhos.

```
from sklearn.neighbors import KNeighborsClassifier

def knn_classification(X_train, y_train, X_test, k=3):

    knn_classifier = KNeighborsClassifier(n_neighbors=k)

    knn_classifier.fit(X_train, y_train)

    y_pred = knn_classifier.predict(X_test)

    return y_pred

y_pred = knn_classification(cj_desen, y_d, cj_test, k=3)
```

Listing 4: Algoritmo KNN.

A implementação dessa função permite com que o algoritmo retorne para a variável y_pred a lista correspondente a classificação dos vinhos prevista. Através dessa lista, podemos realizar as comparações e verificar a capacidade de previsão do modelo testado.

3 Resultados

Através de toda implementação aqui apresentada, foi possível realizar a análise dos resultados em diferentes situações de teste. Inicialmente, apenas como meio de confirmação e teste foi gerada a matriz de confusão para uma única situação de condições específicas. A seguir, a verificação de média de acertos (acurácia) para 10 testes consecutivos e por fim, a criação de uma tabela em que tanto os valores de K e de treinamento foram alterados.

3.1 Simulação única

Fixando o valor de K para 3 e fazendo 60% dos dados para treinamento e 40% para teste, foi construída a matriz de confusão dos resultados, como forma comparativa as previsões geradas pelo algoritmo.

Para essas verificações, foram usadas as bibliotecas Seaborn e as ferramentas da Scikit-Learn.

```
from sklearn.metrics import confusion_matrix
import matplotlib.pyplot as plt
from sklearn.metrics import accuracy_score
import seaborn as sns

y_pred = knn_classification(cj_desen, y_d, cj_test, k=3)

cm = confusion_matrix(y_t, y_pred)
accuracy = accuracy_score(y_t, y_pred)
```

```
10
11 plt.figure(figsize=(8, 6))
12 sns.heatmap(cm, annot=True, fmt='d', cmap='Blues', annot_kws={"size":
       16})
13 plt.xlabel('Previsoes')
14 plt.ylabel('Valores Verdadeiros')
15 plt.title('Matriz de Confusao')
16 plt.gca().set_xticklabels([1, 2, 3])
17 plt.gca().set_yticklabels([1, 2, 3])
18 plt.show()
19
20 print(f'Acuracia do modelo: {accuracy:.2f}')
```

Listing 5: Plotagem da matriz de confusão.

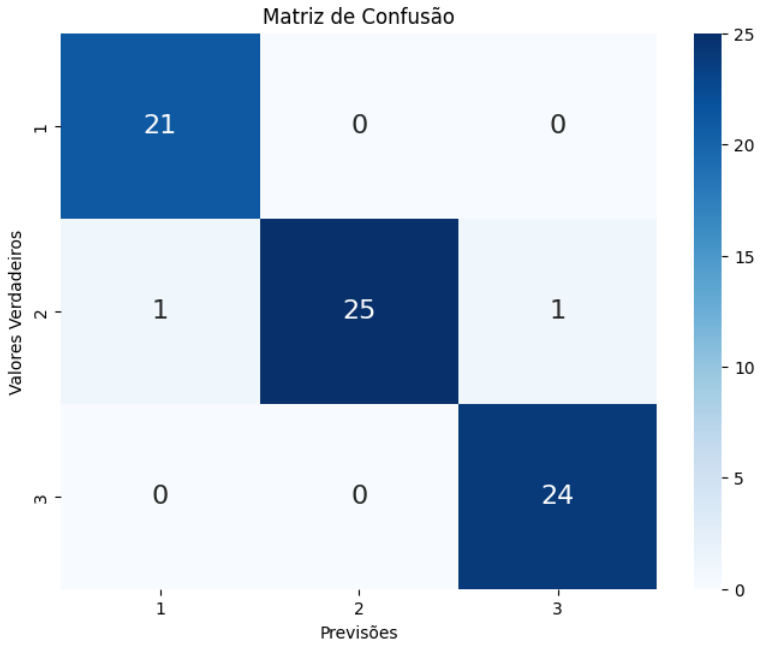

Figura 7: Matriz de Confusão.

Observando os resultados obtidos, percebemos que o modelo implementado, para esse caso específico, obteve uma acurácia de 97%, já que existiram dois valores que foram classificados incorretamente. Um deles, deveria estar classificado como classe 2 e o algoritmo o previu como classe 1 e o outro deveria também estar na classe 2 mas foi classificado como classe 3.

Idealmente, o modelo deveria gerar uma matriz completamente diagonal, no entanto, como as variações dos dados podem gerar vizinhos que mesmo próximos divergem da previsão, a matriz acaba por gerar alguns poucos valores incorretos.

3.2 Média de 10 realizações

Dando sequência a análise dos resultados, podemos verificar como o mesmo sistema se comporta para diversos testes usando variações diferentes dos dados utilizados. Então, nesse teste, o valor de K foi especificado como 10 e apenas 20% dos dados foram separados para treinamento a cada laço do loop gerado.

Com isso, a cada laço do loop, um novo conjunto de 20% vai ser gerado de forma aleatória, permitindo verificar o comportamento de acertos do programa para as diferentes variações.

```
media = 0
for n in range(10):
  indices = np.random.permutation(dfn.shape[0])
  div = int(0.2*len(indices))
  desen_id , test_id = indices[:div], indices[div:]
  cj_desen, cj_test = dfn.loc[desen_id,:], dfn.loc[test_id,:]
  y_d = cj_desen['Class']
  y_t = cj_test['Class']
  del cj_desen['Class']
  del cj_test['Class']

  y_pred = knn_classification(cj_desen, y_d, cj_test, k=10)
  media = media + accuracy_score(y_t, y_pred)

print(f'Media da acuracia dos 10 testes: {media/10}')
```

Listing 6: Média das 10 tentativas.

Com essa implementação, a média da acurácia se deu em 92,58% para as 10 tentativas percorridas pelo loop.

3.3 Simulações variáveis

Como forma de finalização do algoritmo, foram realizados laços concatenados de forma a realizar a variação K, a variação da porcentagem e para cada uma dessas configurações, testar a média dos 10 testes consecutivos. Assim, é possível obter uma tabela que nos mostra a variação da eficiência do algoritmo de acordo com a modificação desses parâmetros.

```
import pandas as pd
import numpy as np
from sklearn.metrics import accuracy_score

p1 = [10, 20, 30, 40, 50]
k3 = [1, 3, 5, 7, 9]

vis = pd.DataFrame(columns=k3, index=p1)

```

```
for pc in p1:
    for k2 in k3:
        media1 = 0
        for n in range(10):
            indices = np.random.permutation(dfn.shape[0])
            div = int((pc / 100) * len(indices))

            desen_id, test_id = indices[:div], indices[div:]

            cj_desen, cj_test = dfn.loc[desen_id, :], dfn.loc[test_id, :]
            y_d = cj_desen['Class']
            y_t = cj_test['Class']
            del cj_desen['Class']
            del cj_test['Class']

            y_pred = knn_classification(cj_desen, y_d, cj_test, k=k2)
            media1 = media1 + accuracy_score(y_t, y_pred)

        vis.at[pc, k2] = media1 / 10

display(vis)
```

Listing 7: Implementação das variações.

	1	3	5	7	9
10	0.871429	0.913043	0.914907	0.818012	0.585714
20	0.922378	0.943357	0.952448	0.940559	0.937063
30	0.948	0.9472	0.956	0.9576	0.9584
40	0.947664	0.948598	0.962617	0.950467	0.964486
50	0.942697	0.949438	0.949438	0.959551	0.967416

Figura 8: Tabela dos resultados.

Na tabela, observamos que o eixo vertical (10 à 50) mostra a variação das porcentagem utilizada para treinamento enquanto o eixo horizontal (1 à 9) mostra a variação de K.

Nota-se que a eficiência do algoritmo é relativamente boa e nos mostra um comportamento interessante: é necessário buscar um ponto ótimo de operação para K e porcentagem. Já que aumentar o K, mantendo uma baixa porcentagem pode não trazer resultados eficazes, como também aumentar apenas a porcentagem não trará.

Todo o algoritmo pode ser encontrado no link disponível nas referências.

4 Considerações

Como apresentado, foi possível a implementação do algoritmo K-Nearest Neighbors de forma eficiente e atendendo os requisitos previstos inicialmente. Levando também os conceitos que regem o algoritmo de classificação. Assim a implementação do algoritmo K-Nearest Neighbors (KNN) requer várias considerações cruciais. A escolha apropriada de K, o pré-processamento de dados para lidar com escalas diferentes, a eficiência computacional, a validação cruzada para avaliação robusta, a interpretabilidade limitada do modelo, os trade-offs em relação a outros algoritmos e a capacidade de aprendizado contínuo são todos aspectos a serem abordados. O sucesso na implementação do KNN requer equilíbrio e atenção a esses pontos-chave, tornando-o uma ferramenta valiosa para classificação e regressão de padrões em diversos contextos.

5 Referências

[1] KAPUR, I. k-NN on Iris Dataset. Disponível em: <https://towardsdatascience.com/k-nn-on-iris-dataset-3b827f2591e>.

[2] KUMAR, A. K-Nearest Neighbors Explained with Python Examples. Disponível em: <https://vitalflux.com/k-nearest-neighbors-explained-with-python-examples/>.

[3] sklearn.datasets.load_wine — scikit-learn 0.23.1 documentation. Disponível em: <http s://scikit- learn.org/stable/modules/generated/sklearn.datasets.load_wine.html>.

[4] LINK DO CÓDIGO COMPLETO.