UNIVERSIDADE FEDERAL DE UBERLÂNDIA

FACULDADE DE COMPUTAÇÃO

CURSO DE SISTEMAS DE INFORMAÇÃO

Algoritmo de classificação k-nearest neighbors

Distância Euclidiana

Atividade apresentada para obtenção de nota semestral na disciplina de Mineração de Dados do curso de Sistemas de Informação, ministrada pelo Prof. Dr. Marcos Luiz de Paula Bueno.

Danilo Pereira de Oliveira 31721BSI005

Monte Carmelo

2019

No reconhecimento de padrões, o **algoritmo *k*- vizinhos mais próximos** (***k*-NN**) é um método não paramétrico usado para classificação e regressão. Nos dois casos, a entrada consiste nos *k* exemplos de treinamento mais próximos no espaço de recursos . A saída depende se *k* -NN é usado para classificação ou regressão:

- Na *classificação k-NN*, a saída é uma associação de classe. Um objeto é classificado pelo voto de pluralidade de seus vizinhos, sendo o objeto atribuído à classe mais comum entre os *k* vizinhos mais próximos (*k* é um número inteiro positivo, geralmente pequeno). Se *k*=1, o objeto é simplesmente atribuído à classe do único vizinho m,ais próximo.
- Na *regressão k-NN*, a saída é o valor da propriedade para o objeto. Este valor é a média dos valores de *k* vizinhos mais próximos.

k-NN é um tipo de aprendizado baseado em instância, ou aprendizado lento, em que a função é aproximada apenas localmente e todo o cálculo é adiado até a classificação.

Tanto para classificação quanto para regressão, uma técnica útil pode ser atribuir pesos às contribuições dos vizinhos, para que os vizinhos mais próximos contribuam mais com a média do que os mais distantes. Por exemplo, um esquema de ponderação comum consiste em atribuir a cada vizinho um peso de 1/*d*, em que *d* é a distância do vizinho.

Os vizinhos são obtidos de um conjunto de objetos para os quais a classe (para a classificação *k*-NN) ou o valor da propriedade do objeto (para a regressão *k*-NN) é conhecida. Isso pode ser pensado como o conjunto de treinamento para o algoritmo, embora nenhuma etapa explícita do treinamento seja necessária.

Uma peculiaridade do algoritmo *k*-NN é que ele é sensível à estrutura local dos dados.

Trabalharemos com um conjunto de dados que contém informações físico-químicas dos vinhos portugueses (vinho verde), bem como sua qualidade, conforme observado pelos seres humanos. O problema, nesses dados, é prever automaticamente a qualidade com base nessas informações, a fim de auxiliar o trabalho de avaliação dos enólogos, melhorar a produção de vinho e direcionar o gosto dos consumidores nos mercados. nicho. Este conjunto de dados está disponível nos arquivos da UCI: http://archive.ics.uci.edu/ml/machine-learning-databases/wine-quality/

No que diz respeito a análise de dados feita no decorrer deste trabalho, para auxiliar na execução do algoritmo foram utilizadas as seguintes bibliotecas/pacotes:

```
12
13 import pandas as pd
14 import numpy as np  #Biblioteca para computação científica
15 import math #Biblioteca para funções matemáticas
16 import matplotlib.pyplot as plt  #biblioteca para visualização de dados
17 from sklearn import dummy  #classificador para fazer previsões usando regras simples
18 from sklearn import model_selection   #Dividir matrizes em subconjuntos aleatórios de train e test
19 from sklearn.model_selection import KFold  #Fornece índices de treinamento / teste para dividir dados em conjuntos de treinamento / teste
20 from sklearn import neighbors, metrics   #Classificador que implementa o voto dos k-vizinhos mais próximos.
21 from sklearn import preprocessing   #pacote fornece várias funções utilitárias comuns e classes de transformadores para alterar
22 #vetores de recursos brutos em uma representação mais adequada para os estimadores a jusante.
23 import warnings
24 warnings.filterwarnings(action='ignore')
```

Na preparação dos dados, vamos receber o arquivo "winequality-white.csv" que é a nossa base de dados:

```
27 data = pd.read_csv('winequality-white.csv', sep=";")   #preparação dos dados
28 data.head()
```

Após a preparação já é possível distinção de valores para cada coluna da nossa base:

data - DataFrame

Index	fixed acidity	volatile acidity	citric acid	residual sugar	chlorides	ree sulfur dioxid	otal sulfur dioxid	density	pH	sulphates	alcohol	quality
0	7	0.27	0.36	20.7	0.045	45	170	1.001	3	0.45	8.8	6
1	6.3	0.3	0.34	1.6	0.049	14	132	0.994	3.3	0.49	9.5	6
2	8.1	0.28	0.4	6.9	0.05	30	97	0.9951	3.26	0.44	10.1	6
3	7.2	0.23	0.32	8.5	0.058	47	186	0.9956	3.19	0.4	9.9	6
4	7.2	0.23	0.32	8.5	0.058	47	186	0.9956	3.19	0.4	9.9	6
5	8.1	0.28	0.4	6.9	0.05	30	97	0.9951	3.26	0.44	10.1	6
6	6.2	0.32	0.16	7	0.045	30	136	0.9949	3.18	0.47	9.6	6
7	7	0.27	0.36	20.7	0.045	45	170	1.001	3	0.45	8.8	6

Format | Resize | Background color | Column min/max | Save and Close | Close

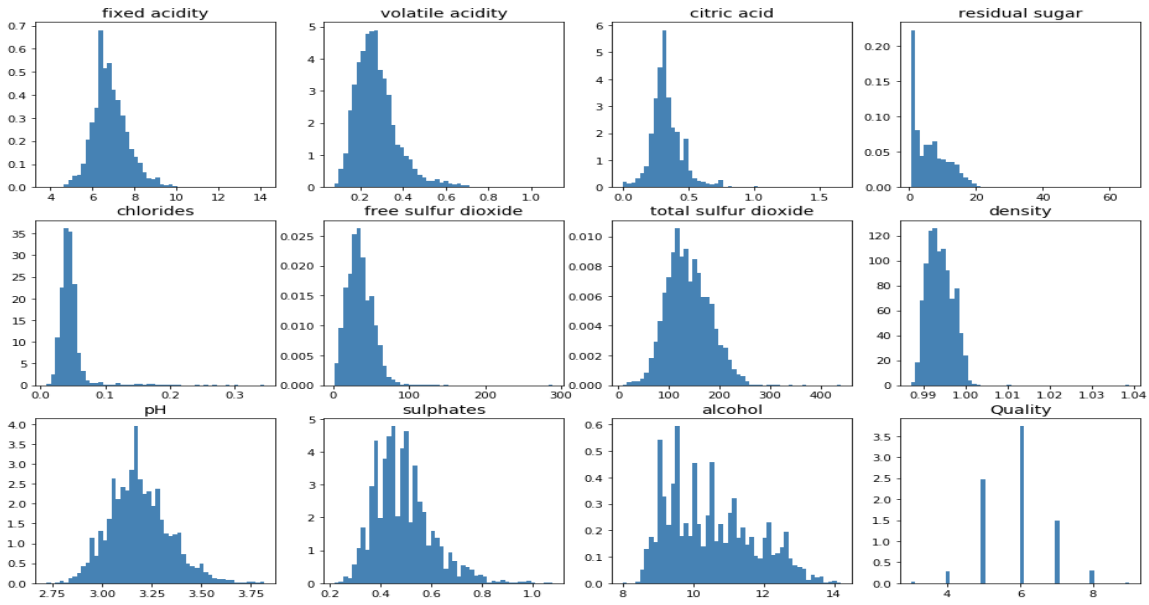
Nossa base contém 12 colunas, 11 que correspondem a vários indicadores físico-químicos e 1 que é a qualidade do vinho. Min/max visualizado no DataFrame acima.

Extrairemos duas matrizes numpy desses dados, uma contendo os pontos e a outra contendo as tags:

```
34
35 X = data.as_matrix(data.columns[:-1])
36 y_m = data.as_matrix([data.columns[-1]])
37 y = y_m.flatten()
38
```

Agora podemos exibir um histograma para cada uma de nossas variáveis:

```
42 fig = plt.figure(figsize=(16, 12))
43 for feat_idx in range(X.shape[1]):
44     ax = fig.add_subplot(3,4, (feat_idx+1))
45     h = ax.hist(X[:, feat_idx], bins=50, color='steelblue',
46                 normed=True, edgecolor='none')
47     ax.set_title(data.columns[feat_idx], fontsize=14)
48
49 ax = fig.add_subplot(3,4, 12)
50 h = ax.hist(y, bins=50, color='steelblue',
51             normed=True, edgecolor='none')
52 ax.set_title('Quality', fontsize=14)
53 plt.show()
54
```



Em particular, essas variáveis recebem os valores em conjuntos diferentes. Por exemplo, “sulfatos” varia de 0 a 1, enquanto “dióxido de enxofre total” varia de 0 a 440.
Portanto, será necessário padronizar os dados para que o segundo não domine completamente o primeiro.
Começaremos por transformar esse problema em um problema de classificação: será uma questão de separar os bons vinhos dos medíocres:

```
61 y_class = np.where(y<6, 0, 1)
```

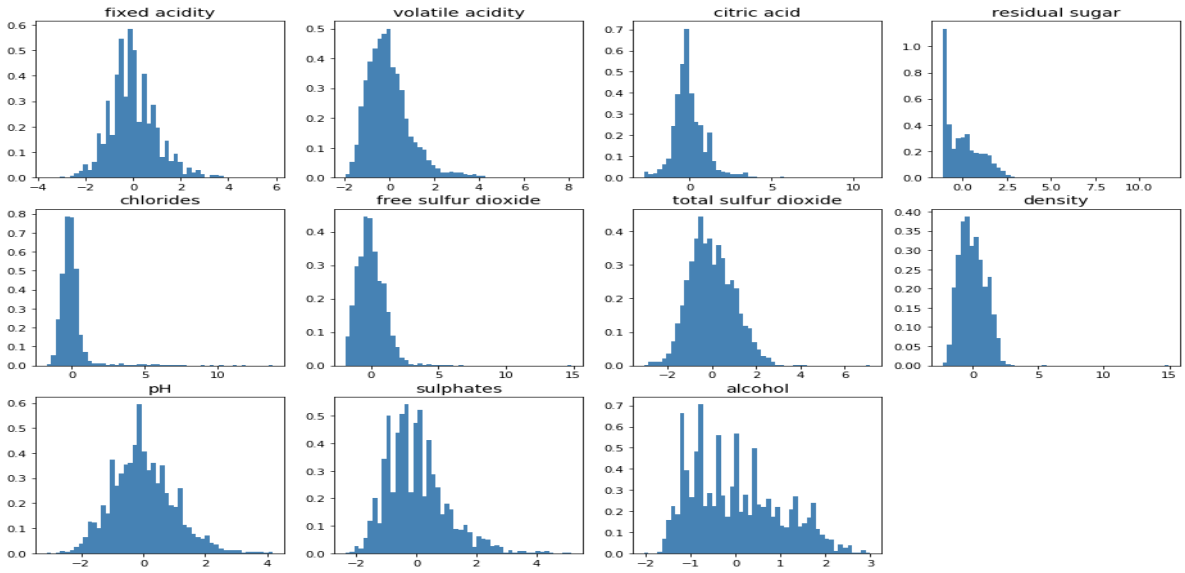
Vamos dividir nossos dados em treinamento e teste. O conjunto de testes conterá 10% dos dados. Definimos 'random_state' com o valor da semente para que a separação dos dados seja fixa e reproduzível.

```
69 X_train, X_test, y_train, y_test = \
70     model_selection.train_test_split(X, y_class,
71                                      test_size=0.1, # 10% dos dados no conjunto de teste
72                                       random_state = seed
73                                      )
74
```

Agora podemos padronizar os dados de treinamento e aplicar a mesma transformação aos dados de teste:

```
72 std_scale = preprocessing.StandardScaler().fit(X_train)
73 X_train_std = std_scale.transform(X_train)
74 X_test_std = std_scale.transform(X_test)
75
```

Os dados podem ser visualizados novamente para garantir que as diferentes variáveis recebam valores que agora possuem ordens de magnitude semelhantes.

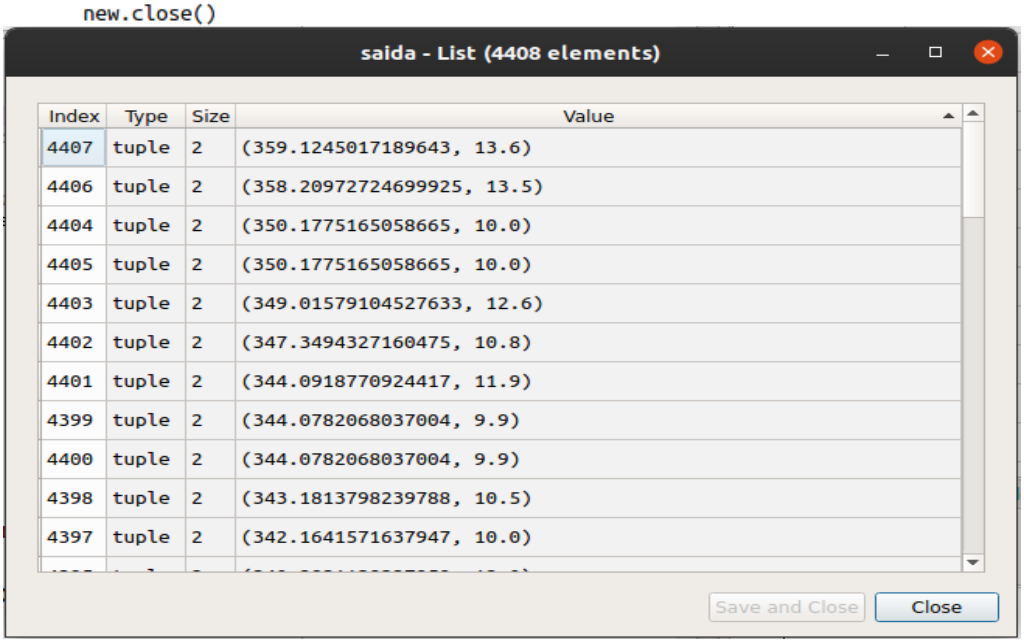


Aplicando a distância euclidiana:

```
def dist_euclidiana(v1, v2):
    v1, v2 = np.array(v1), np.array(v2)
    diff = v1 - v2
    quad_dist = np.dot(diff, diff)
    return math.sqrt(quad_dist)

n1,n2 = np.shape(X_test)

for lin in X_test[:490]:
    saida = []
    new = open("saida-teste.txt","w")
    for lin1 in X_train:
        saida.append((dist_euclidiana(lin,lin1),lin1[10]))
    saida.sort()
    for ii in saida:
        new.writelines(str(ii[0])+" "+str(ii[1])+"\n")
    new.close()
```



Criamos a dobra de 5k usando a semente para que essas dobras sejam reproduzíveis:

```
kf = KFold(n_splits=5, shuffle=True, random_state=seed)
for train_index, test_index in kf.split(X_train_std):
    print("TRAIN:", "taille", len(train_index), ", 5ers indices", train_index[0:5],
          "|| TEST:", "taille", len(test_index), ", 5ers indices", test_index[0:5])
```

```
TRAIN: taille 3526 , 5ers indices [0 1 2 3 4] || TEST: taille 882 , 5ers indices [ 8 14 17 19 23]
TRAIN: taille 3526 , 5ers indices [1 2 3 4 5] || TEST: taille 882 , 5ers indices [ 0  6  7 12 25]
TRAIN: taille 3526 , 5ers indices [0 2 3 4 5] || TEST: taille 882 , 5ers indices [ 1 13 18 20 22]
TRAIN: taille 3527 , 5ers indices [0 1 4 6 7] || TEST: taille 881 , 5ers indices [ 2  3  5 10 15]
TRAIN: taille 3527 , 5ers indices [0 1 2 3 5] || TEST: taille 881 , 5ers indices [ 4  9 11 16 34]
```

Agora usaremos o método "GridSearchCV" para validar cruzado o parâmetro k de um kNN (o número de vizinhos mais próximos) no jogo de treinamento:

```
# Defina os valores dos hiperparâmetros a serem testados
param_grid = {'n_neighbors':[3, 5, 7, 9, 11, 13, 15]}
# Escolha uma pontuação para otimizar, aqui a precisão (proporção de previsões corretas)
score = 'accuracy'

# Cria um classificador kNN com pesquisa de hiperparâmetro de validação cruzada
clf = model_selection.GridSearchCV(neighbors.KNeighborsClassifier(), # um classificador kNN
                                   param_grid, # hiperparâmetros para testar
                                   cv=kf, # dobras para validação cruzada
                                   scoring=score # pontuação para otimizar
                                   )
# Otimiza esse classificador no treinamento
clf.fit(X_train_std, y_train)

# Visualizar o(s) hiperparâmetro(s) ideal(is)
print("Melhores hiperparâmetros na base de treinamento: ")
print(clf.best_params_)

#Ver o desempenho correspondente
print("Resultados da validação cruzada: ")
for mean, std, params in zip(clf.cv_results_['mean_test_score'], # pontuação média
                             clf.cv_results_['std_test_score'], # desvio padrão da pontuação
                             clf.cv_results_['params'] # valor do hiperparâmetro
                             ):
    print("\t%s = %0.3f (+/-%0.03f) for %r" % (score, # critério usado
                                               mean, # pontuação média
                                               std * 2, # barra de erro
                                               params # hiperparâmetro
                                               ))

pd.DataFrame(clf.cv_results_)[['params', 'mean_test_score', 'rank_test_score']]
```

```
Melhores hiperparâmetros na base de treinamento:
{'n_neighbors': 3}
Resultados da validação cruzada:
        accuracy = 0.771 (+/-0.028) for {'n_neighbors': 3}
        accuracy = 0.768 (+/-0.010) for {'n_neighbors': 5}
        accuracy = 0.762 (+/-0.015) for {'n_neighbors': 7}
        accuracy = 0.758 (+/-0.010) for {'n_neighbors': 9}
        accuracy = 0.766 (+/-0.018) for {'n_neighbors': 11}
        accuracy = 0.767 (+/-0.011) for {'n_neighbors': 13}
        accuracy = 0.766 (+/-0.012) for {'n_neighbors': 15}
```

O melhor desempenho (acurácia ≅ 0,771) é alcançado aqui com 3 vizinhos.
Agora podemos verificar o desempenho sobre a base de teste (o GridSearchCV treinou automaticamente o melhor modelo em toda a base de treinamento):

```
y_pred = clf.predict(X_test_std)
```

Pontuação de precisão da base de teste: 0.751

```
def cvf(X, y, n_neighbors, n_fold, verbose=False):
    """
    X : dados
    y : valor a prever
    n_neighbors : lista de K para testar o K-NN
    n_fold : número de dobras a serem criadas
    verbose : Falso por padrão, exibe informações de código
    """

    kf = KFold(n_splits=n_fold, shuffle=True, random_state=seed)
    if verbose:
        for train_index, test_index in kf.split(X):
            print("TRAIN:", "taille", len(train_index), ", 5ers indices", train_index[0:5],
                  "|| TEST:", "taille", len(test_index), ", 5ers indices", test_index[0:5])

    # inicializa a tabela que conterá os diferentes resultados
    accuracy_mean = []

    # faz um loop para testar os diferentes valores de K para o KNN
    for kn in n_neighbors:
        if verbose: print("kn", kn)
        knn = neighbors.KNeighborsClassifier(kn) # inicialização do classificador

        accuracy = [] # inicialização de uma lista que conterá a precisão de cada dobra k e cada knn

        # faz um loop nas diferentes dobras
        for train_index, test_index in kf.split(X):
            if verbose: print("TRAIN:", "taille", len(train_index), ", 5ers indices", train_index[0:5],
                      "|| TEST:", "taille", len(test_index), ", 5ers indices", test_index[0:5])
            X_train, X_test = X[train_index], X[test_index]
            y_train, y_test = y[train_index], y[test_index]

            # fiz o knn na dobra de treinamento
            knn.fit(X_train, y_train)

            # testa a precisão na dobra de teste e a adiciona à tabela 'precisão':
            accuracy.append([knn.score(X_test, y_test)])
        if verbose: print(np.mean(accuracy))

        # os resultados são agregados mantendo apenas a média da precisão das diferentes dobras para cada k
        accuracy_mean.append([kn, round(np.mean(accuracy), 3)])

    return accuracy_mean
```

Fazemos a iteração novamente na função com os mesmos valores usados para o GridSearchCV:

```
n_neighbors = [3, 5, 7, 9, 11, 13, 15, 22]
n_fold = 5
cvf = cvf(X_train_std, y_train, n_neighbors, n_fold, verbose=False)
# A tabela de precisão média para cada K é obtida:
cvf_df = pd.DataFrame(cvf, columns=['knn', 'mean_accuracy'])
cvf_df
cvf_df[cvf_df['mean_accuracy'] == cvf_df['mean_accuracy'].max()]
#O K a ser selecionado é, portanto, 3, com uma precisão média de 0,771,
#o que corresponde ao que o GridSearchCV retornou com os mesmos dados.
#Agora podemos verificar o desempenho da base de teste.
knn3 = neighbors.KNeighborsClassifier(3)
knn3.fit(X_train_std, y_train)
```

Acurácia na base de teste: 0.751

Encontramos a mesma precisão com o GridSearchCV

Percebe-se que encontramos a mesma precisão com o GridSearchCV

Partimos agora para a avaliação da classificação do knn:

```
y_pred_proba = clf.predict_proba(X_test_std)[:, 1]
[fpr, tpr, thr] = metrics.roc_curve(y_test, y_pred_proba)
plt.plot(fpr, tpr, color='coral', lw=2)
plt.xlim([0.0, 1.0])
plt.ylim([0.0, 1.05])
plt.xlabel('1 - specificite', fontsize=14)
plt.ylabel('Sensibilite', fontsize=14)
plt.show()

```

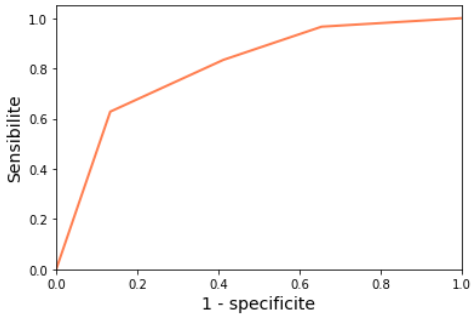
Temos:

```
ROC = 0.803
Sensibilidade : 0.97
Especificidade : 0.35
Limiar : 0.33
RMSE : 0.70
```

Uma das principais formas de avaliar um classificador é através das ROC. As curvas ROC é uma curva que é desenhada utilizando a taxa de positivos verdadeiros e a taxa de falso positivo:

Variando o valor do limiar, obtém-se cada ponto da curva.

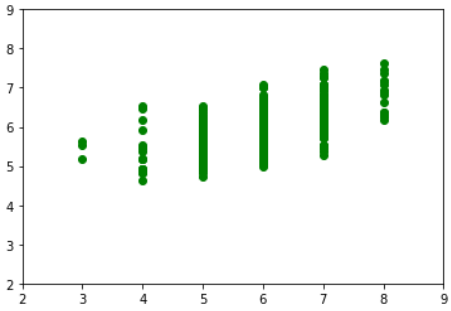
Como funciona a limiar na prática? A limiar varia para cada classificador. Por exemplo, nas redes neurais é a confiança de predição, nas árvores de decisão são a quantidade de instancia que chegam naquela folha e no KNN é a quantidade de positivos em cada k questionado.



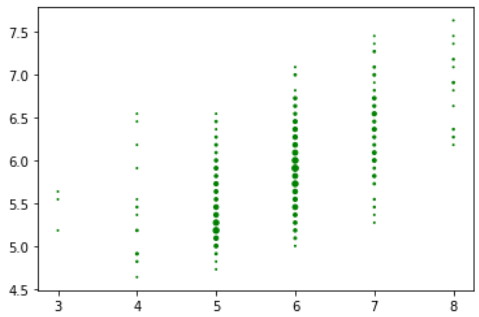
Avaliando com abordagens ingênuas, treinando um KNN= 11 com nossa base de dados:

Calculo erro médio quadrático da raiz (RMSE)

Corresponde à raiz quadrada da diferença média entre os valores de resultados conhecidos observados e os valores previstos.



RMSE: 0.70



RMSE: ≅1.94

Portanto, quando treinado um knn com k= 11 nos dados em questão, verificou-se que obtemos um RMSE muito superior ao RMSE obtido pelo nosso modelo knn k=3, sendo RMSE:≅1.94 e RMSE:0.70 respectivamente. Logo, o modelo utilizado conseguiu aprender melhor do que um modelo aleatório.

Para visualizar melhor os dados, podemos usar como círculos marcadores cujo tamanho é proporcional ao número de pontos presentes nas coordenadas entre 3 e 8, que são os valores dos rótulos. Notamos, portanto, um acúmulo de previsões corretas na diagonal, afinal muitos dos vinhos analisados tem uma classificação de 6, e muitas das previsões estão em torno desse valor.