UNIVERSIDADE FEDERAL DE PERNAMBUCO
CENTRO DE INFORMÁTICA

PROJETO DA DISCIPLINA SISTEMAS INTELIGENTES
ANÁLISE DOS PARÂMETROS DOS CLASSIFICADORES: ÁRVORE DE DECISÃO, KNN E REGRESSÃO LOGÍSTICA

Matheus Alves Almeida (maa4), Gabriel Nogueira Leite (gnl2), Pietro Bernardo Santos Masur (pbsm), João Vitor Oliveira de Araújo (jvoa), Jailson José da Silva Junior (jjsj2)

Recife, 2021

Introdução

O presente relatório descreve com detalhes o desenvolvimento do projeto da disciplina Sistemas Inteligentes, ministrada pelo professor Francisco de A. T. de Carvalho. O objetivo do projeto é aplicar uma análise de parâmetros de diferentes classificadores aplicados ao mesmo *dataset.* O conjunto de dados utilizados nesse projeto foi o *Wireless Indoor Localization Data Set*; os dados foram coletados em um espaço interno, observando a intensidade do sinal de sete sinais WiFi visíveis em um smartphone.

Foram aplicados três algoritmos de classificação neste conjunto de dados, sendo eles: 1) Árvore de decisão; 2) K-vizinhos próximos; 3) Regressão logística. Para isso, foi utilizada uma biblioteca em Python chamada *Scikit Learn*. Vale ressaltar que a leitura dos dados e o tratamento do *dataset* foi feito da mesma forma para todos os arquivos: 1) foi realizada a leitura do conjunto de dados usando a biblioteca Pandas e tabulação /t; 2) As colunas foram nomeadas de A1 até A7, a única diferença é a última coluna que foi renomeada para “Saída”; 3) Foram divididos 30% do conjunto de dados para teste e o restante para treino. A imagem a seguir mostra como isso foi feito.

```
#Leitura dos dados

df = pd.read_csv('wifi_localization.txt', header = None, delimiter = "\t")
```

```
df.columns=['A1','A2', 'A3', 'A4', 'A5', 'A6', 'A7', 'Saida']
```

```
x_train, x_test, y_train, y_test = train_test_split(df.drop('Saida', axis = 1), df['Saida'], test_size=0.30)
```

A descrição de cada algoritmo aplicado será apresentada a seguir.

Árvores de decisão

Após a etapa de leitura que foi explicada na introdução, foi aplicado o método *DecisionTreeClassifier()* com os parâmetros *default* para a análise de decisão dos dados e depois foi aplicado o método *fit()* para calcular os parâmetros de aprendizagem do modelo e realizar o treinamento. Com isso, foi feita a predição usando o método *predict()* e já se obteve um resultado excelente para o modelo, como descrito na primeira imagem de resultados. Então foram feitas algumas modificações testando como o modelo se comportaria com alterações nos parâmetros de *max_depth* e *min_samples_leaf*. A imagem a seguir mostra as alteraçẽos nos parâmetros. Para uma melhor análise, foram fixados 2 parâmetros enquanto um terceiro era alterado.

```
result0 = DecisionTreeClassifier()
result1 = DecisionTreeClassifier(max_depth=3, criterion='entropy', min_samples_leaf=2)
result2 = DecisionTreeClassifier(max_depth=5, criterion='entropy', min_samples_leaf=2)
result3 = DecisionTreeClassifier(max_depth=5, criterion='entropy', min_samples_leaf=3)
#Treinando o modelo
result0.fit(x_train, y_train)
result1.fit(x_train, y_train)
result2.fit(x_train, y_train)
result3.fit(x_train, y_train)
```

Ao limitar a altura máxima da árvore para 3, houve uma perda de aproximadamente 0,3 em precisão, 0,2 em recall e 0,2 em f1-score em relação ao que tinha nos parâmetros default, porém os resultados ainda se mostraram excelentes.

```
In [16]: predict = result0.predict(x_test)
         score = list()
         score.append(accuracy_score(y_test, predict))
         print(classification_report(y_test, predict))
         print('\n')
```

```
              precision    recall  f1-score   support

           1       0.99      0.99      0.99       141
           2       0.98      0.97      0.98       168
           3       0.95      0.97      0.96       153
           4       0.99      0.99      0.99       138

    accuracy                           0.98       600
   macro avg       0.98      0.98      0.98       600
weighted avg       0.98      0.98      0.98       600
```

Resultados do *result0*

Agora, aumentando a altura máxima para 5 e mantendo os outros parâmetros, notou-se um ganho em todos os parâmetros de resultado. Permitindo a conclusão de que 3 não é o melhor valor para esse parâmetro.

```
In [17]: predict = result1.predict(x_test)
         score = list()
         score.append(accuracy_score(y_test, predict))
         print(classification_report(y_test, predict))
         print('\n')
```

```
              precision    recall  f1-score   support

           1       0.99      0.98      0.99       141
           2       0.98      0.94      0.96       168
           3       0.91      0.97      0.94       153
           4       0.99      0.99      0.99       138

    accuracy                           0.97       600
   macro avg       0.97      0.97      0.97       600
weighted avg       0.97      0.97      0.97       600
```

Resultados do *result1*

```
In [18]: predict = result2.predict(x_test)
         score = list()
         score.append(accuracy_score(y_test, predict))
         print(classification_report(y_test, predict))
         print('\n')
```

```
              precision    recall  f1-score   support

           1       0.99      0.99      0.99       141
           2       0.98      0.95      0.97       168
           3       0.94      0.97      0.96       153
           4       0.99      0.99      0.99       138

    accuracy                           0.98       600
   macro avg       0.98      0.98      0.98       600
weighted avg       0.98      0.98      0.98       600
```

Resultados do *result2*

Agora, mantendo 5 para a altura máxima e colocando o *min_samples_leaf* como 3, houve um ganho ainda maior em relação à troca de parâmetros anterior.

```
In [19]: predict = result3.predict(x_test)
         score = list()
         score.append(accuracy_score(y_test, predict))
         print(classification_report(y_test, predict))
         print('\n')
```

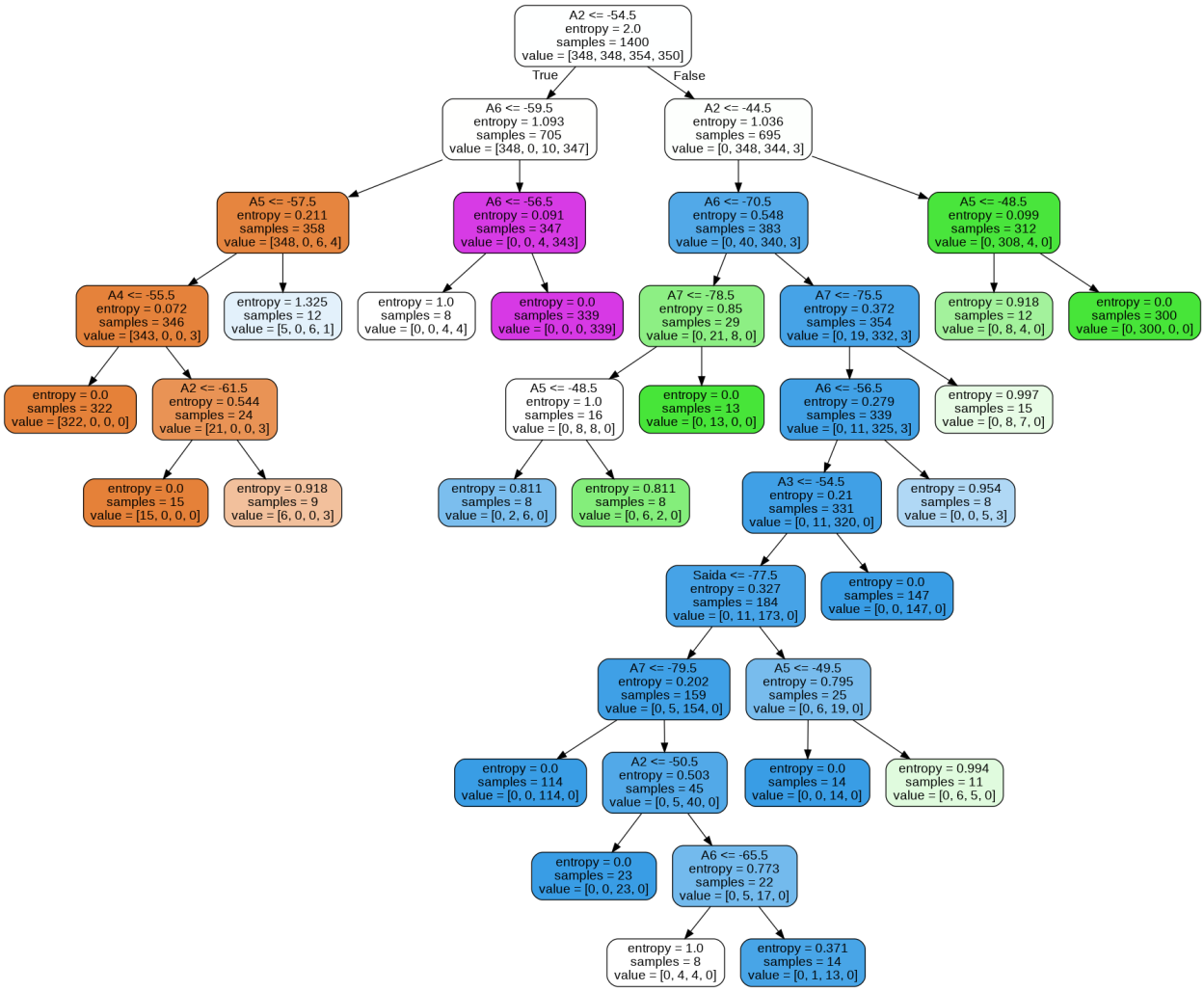
```
              precision    recall  f1-score   support

           1       0.99      0.99      0.99       141
           2       0.99      0.95      0.97       168
           3       0.93      0.98      0.96       153
           4       1.00      0.99      0.99       138

    accuracy                           0.97       600
   macro avg       0.98      0.98      0.98       600
weighted avg       0.98      0.97      0.98       600
```

Resultados do *result3*

Com isso, chegamos a conclusão que *max_depth = 5, min_samples_leaf = 3* e usando Entropia como critério são os melhores parâmetros para esse classificador dado esse conjunto de dados. Também foi notado que aumentar os valores dos parâmetros para maiores que estes mencionados não resulta em nenhum ganho para o modelo. A imagem a seguir mostra a árvore final do modelo:



K-vizinhos próximos (KNN)

Utilizando a base de dados requerida, desenvolvemos um algoritmo usando *Python* e *Scikit Learn* para treinar o método KNN de classificação.

O código em si pode ser visto abaixo:

```
import pandas as pd
import numpy as np

import matplotlib.pyplot as plt
import seaborn as sns

from sklearn.model_selection import train_test_split
from sklearn.neighbors import KNeighborsClassifier
from sklearn.preprocessing import StandardScaler
from sklearn.metrics import accuracy_score, classification_report, co

# Organizando os dados...
df = pd.read_csv('wifi_localization.txt', header=None, delimiter="\t"
df.columns = ['A1', 'A2', 'A3', 'A4', 'A5', 'A6', 'A7', 'Saida']
x_train, x_test, y_train, y_test = train_test_split(
    df.drop('Saida', axis=1), df['Saida'], test_size=0.30)

# Normalizando...
scaler = StandardScaler()
scaler.fit(x_train)
x_train = scaler.transform(x_train)
x_test = scaler.transform(x_test)

# Treinando...
classifier = KNeighborsClassifier(n_neighbors=1)
classifier.fit(x_train, y_train)

# Realizando previsões...
y_pred = classifier.predict(x_test)

# Mostrando classificações...
score = list()
score.append(accuracy_score(y_test, y_pred))
print(classification_report(y_test, y_pred))
print('\n')
```

Código Python usando o KNN

Foi usado um tamanho de 30% para o conjunto de treinamento, com os dados sendo escolhidos aleatoriamente dentro deste conjunto.

Percebe-se que se fez necessário normalizar os dados de treino e de teste para uma melhor acurácia no uso do algoritmo KNN, como fora recomendado.

O KNN utiliza apenas um parâmetro customizável, o número de vizinhos próximos ao novo elemento a ser classificado que serão utilizados na classificação em si. Dada sua simplicidade, decidimos testar, por tentativa e erro e bruta-força, qual o resultado de precisão na classificação do conjunto de testes. Considerando que 30% da quantidade total dos dados eram 600 dados, modificamos um pouco o código e iteramos o knn de 1 a 600, construindo assim um gráfico de desempenho de classificação.

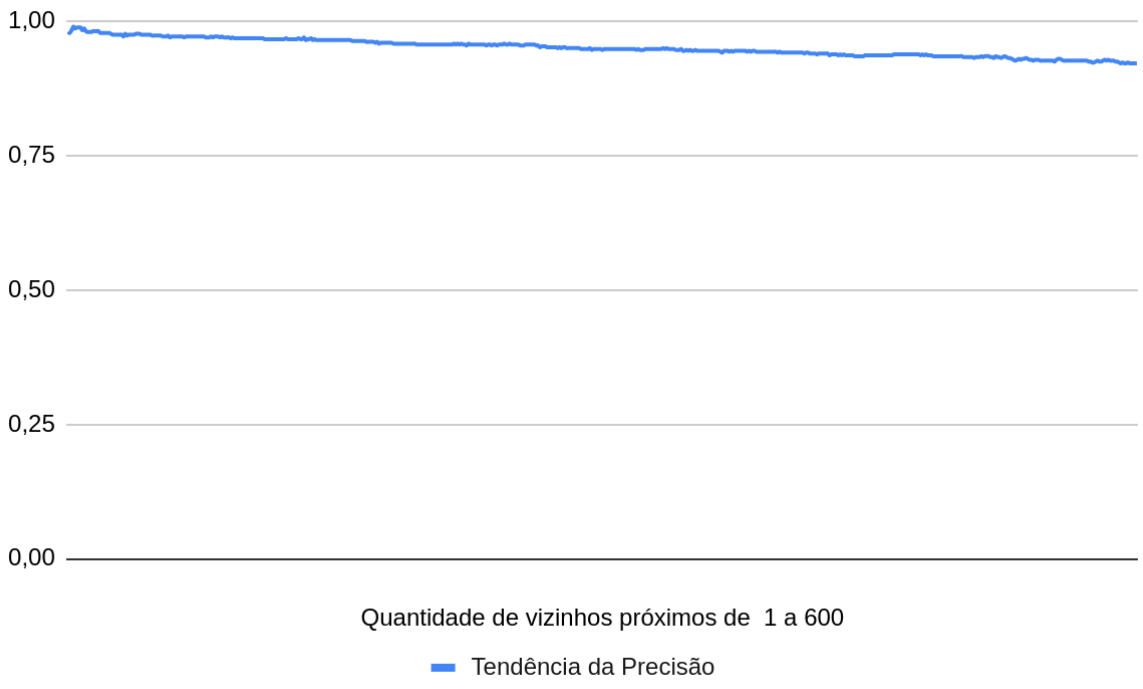
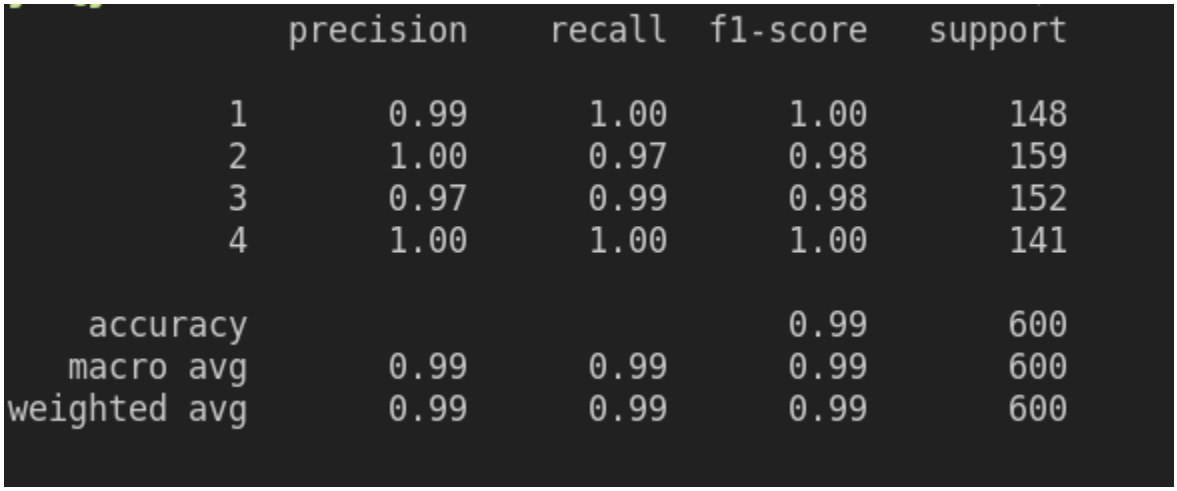


Gráfico de tendência da precisão em relação ao aumento de vizinhos próximos

É perceptível que o desempenho se comporta de maneira decrescente, linear e quase cem por cento conciso em proporção ao aumento da quantidade dos vizinhos próximos. Portanto, conclui-se que, para este conjunto de dados, configurar tal parâmetro como o menor possível se adequa de maneira muito mais vantajosa para nossa classificação. Acrescenta-se que os testes mostraram que 5 vizinhos dá uma precisão de 99%, que em média é a maior registrada neste teste.



	precision	recall	f1-score	support
1	0.99	1.00	1.00	148
2	1.00	0.97	0.98	159
3	0.97	0.99	0.98	152
4	1.00	1.00	1.00	141
accuracy			0.99	600
macro avg	0.99	0.99	0.99	600
weighted avg	0.99	0.99	0.99	600

Report de classificação para 5 vizinhos

Regressão Logística

A partir dos dados coletados, foi realizada a aplicação da técnica de regressão logística, que consiste em uma técnica de mineração de dados, que por sua vez, é um processo de extração e descoberta de padrões em conjuntos de dados envolvendo métodos na interseção de aprendizado de máquina, estatísticas e sistemas de banco de dados.

Após importar as funções necessárias para utilização da regressão logística (from sklearn.linear_model import LogisticRegression), fizemos a separação do conjunto de dados em conjunto de treino e conjunto de testes.

```
[3] dfTarget = df.iloc[0:2000,7]

[4] train, test, trainTarget, testTarget = train_test_split(df,dfTarget,test_size=0.3,random_state=3)
```

Feita a separação dos conjuntos de dados, foi realizado o treinamento do modelo, utilizando a função LogisticRegression, nesse ponto foi encontrado problemas ao utilizar o parâmetro *default*, referente ao número máximo de iterações (*max_iter*), que por padrão tem valor 100. O problema ocorre pois com o máximo de iterações padrão, o valor não foi suficiente para que o modelo convergisse, desta forma tivemos que buscar um valor por tentativa e erro, o valor mínimo encontrado foi 3684 iterações.

```
Treinamento do modelo

[ ] logmodel = LogisticRegression(max_iter=3684)
    logmodel.fit(train,trainTarget)

    LogisticRegression(C=1.0, class_weight=None, dual=False, fit_intercept=True,
                       intercept_scaling=1, l1_ratio=None, max_iter=3684,
                       multi_class='auto', n_jobs=None, penalty='l2',
                       random_state=None, solver='lbfgs', tol=0.0001, verbose=0,
                       warm_start=False)
```

Com o modelo previamente treinado, a próxima etapa foi realizar a predição do conjunto de teste, utilizando a função *predict()*, que retornou um array contendo em cada elemento o resultado da predição.

Para analisar a taxa de acerto do modelo utilizamos a função *score()*, que retornou uma taxa altíssima de assertividade, sendo a taxa aproximadamente 99.6%.

```
[8]  score = logmodel.score(test,testTarget)
     print(score)

     0.9966666666666667

[9]  print(classification_report(results,testTarget))

                  precision    recall  f1-score   support

               1       1.00      1.00      1.00       161
               2       1.00      1.00      1.00       143
               3       1.00      0.99      0.99       148
               4       0.99      1.00      0.99       148

        accuracy                           1.00       600
       macro avg       1.00      1.00      1.00       600
    weighted avg       1.00      1.00      1.00       600
```

Na busca por uma taxa de acerto ainda mais elevada, foi decidido trabalhar com o parâmetro “C” da função *LogisticRegression()*, que equivale a 1/λ da função de custo no modelo de regressão logística.

A regularização é aplicar uma penalidade ao aumento da magnitude dos valores dos parâmetros a fim de reduzir o sobreajuste. Ao treinar um modelo, como um modelo de regressão logística, estão sendo escolhidos os parâmetros que fornecem o melhor ajuste aos dados. Isso significa minimizar o erro entre o que o modelo prevê para a variável dependente, dados seus dados, em comparação com o que a variável dependente realmente é.

O problema surge quando tem-se muitos parâmetros (muitas variáveis independentes), mas não muitos dados. Nesse caso, o modelo frequentemente ajustará os valores dos parâmetros às idiossincrasias em seus dados - o que significa que ele se ajusta quase perfeitamente aos seus dados. No entanto, como essas idiossincrasias não aparecem nos dados futuros que são vistos, o modelo prevê mal.

Para resolver isso, além de minimizar o erro como já discutido, é adicionado ao que está minimizado e também minimiza uma função que penaliza grandes valores dos parâmetros. Na maioria das vezes, a função é $\lambda\Sigma(\theta j)^2$, que é alguma constante λ vezes

a soma dos valores dos parâmetros quadrados $(\theta j)^2$. Quanto maior o λ é menos provável que os parâmetros aumentam em magnitude simplesmente para se ajustar a pequenas perturbações nos dados. Nesse caso, entretanto, em vez de especificar λ, foi especificado C = 1 / λ.

Após algumas tentativas inferimos que o melhor valor para o C era 1.63, isso nos proporcionou alcançar uma performance mais eficiente para o nosso modelo.

Treinamento do novo modelo

```
[10] logmodel2 = LogisticRegression(max_iter=6630, C=1.63)
     logmodel2.fit(train,trainTarget)

     LogisticRegression(C=1.63, class_weight=None, dual=False, fit_intercept=True,
                        intercept_scaling=1, l1_ratio=None, max_iter=6630,
                        multi_class='auto', n_jobs=None, penalty='l2',
                        random_state=None, solver='lbfgs', tol=0.0001, verbose=0,
                        warm_start=False)
```

Com isso obteve-se um f1-score e recall de 100%, ilustrado na imagem abaixo.

```
[12] score2 = logmodel2.score(test,testTarget)
     print(score2)

     1.0
```

```
print(classification_report(results,testTarget))
```

	precision	recall	f1-score	support
1	1.00	1.00	1.00	161
2	1.00	1.00	1.00	143
3	1.00	0.99	0.99	148
4	0.99	1.00	0.99	148
accuracy			1.00	600
macro avg	1.00	1.00	1.00	600
weighted avg	1.00	1.00	1.00	600