MMACD

Matheus Vitoreli de Oliviera

Objetivo da atividade Objetivo desta atividade e então avaliar as implementações existentes de MLP em problemas reais, cujos dados podem ser encontrados em repositórios de dados públicos. Além disso, também deseja-se compreender melhor o comportamento do algoritmo.**4**

• Explicar o dataset escolhido;

Dataset manipulado em Python utilizando *scikit learn* uma biblioteca de *machine learning* desenvolvida para aplicação prática em Python,ela oferece uma ampla variedade de ferramentas eficientes e simples para análise preditiva de dados, modelagem estatística, análise e mineração de dados, além de suporte ao aprendizado supervisionado e não supervisionado.

A escolha e devido ao *Dataset* ser bastante empregado em testes de treinamento e aprendizado onde devemos aplicar um pré-processamento dos dados categóricos, mapeamento e discretizacão e nova atribuição das classes (Diagnóstico (maligno 0, benigno 1) bem como fitar e usar a predição e pôr fim a acurácia da massa de dados em questão e visualizar utilizando alguns *plots.* Utilizou-se dois algoritmos *Decision trees (*classificador de dados baseado em uma série de condicionais), *and k-neighbors* (atribui a classe que ocorre com maior frequência na massa de dados, utilizou-se para uma classificação mais rápida).

Disponível em: <https://www.kaggle.com/code/buddhiniw/breast-cancer-prediction >. Acesso em 22 abril 2023.

• **Mostrar algumas estatísticas interessantes do problema, e justificar porque ele foi escolhido como objetivo de investigação;**

Mapeamento e discretizacão das classes e sua nova atribuição, utilizando uma função simples (data.head()) podemos visualizar os 5 primeiros itens da base em questão, que conta com 569 linha e 32 colunas da qual no alvo e a coluna

Diagnosis classificada em maligno e benigno, posteriormente foram discretizados (0,1) utilizando a função:

```
# Atributo alvo é a coluna diagnosis
y = data["diagnosis"]

# Convertendo atributo categórico em numérico
le = LabelEncoder()
y = le.fit_transform(y)
print(y)
```

```
# Atributo alvo é a coluna diagnosis
y = data["diagnosis"]

# Convertendo atributo categórico em numérico
le = LabelEncoder()
y = le.fit_transform(y)
print(y)
```

```
[1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 0 0 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1
 0 1 1 1 1 1 1 1 1 0 1 0 0 0 0 0 1 1 0 1 1 0 0 0 0 1 0 1 1 0 0 0 0 1 0 1 1
 0 1 0 1 1 0 0 0 1 1 0 1 1 1 0 0 0 1 0 0 1 1 0 0 0 1 1 0 0 0 0 1 0 0 1 0 0
 0 0 0 0 0 0 1 1 1 0 1 1 0 0 0 1 1 0 1 0 1 1 0 1 1 0 0 1 0 0 1 0 0 0 0 1 0
 0 0 0 0 0 0 0 0 1 0 0 0 0 1 1 0 1 0 0 1 1 0 0 1 1 0 0 0 0 1 0 0 1 1 1 0 1
 0 1 0 0 0 1 0 0 1 1 0 1 1 1 1 0 1 1 1 0 1 0 1 0 0 1 0 1 1 1 1 0 0 1 1 0 0
 0 1 0 0 0 0 0 1 1 0 0 1 0 0 1 1 0 1 0 0 0 0 1 0 0 0 0 0 1 0 1 1 1 1 1 1 1
 1 1 1 1 1 1 1 0 0 0 0 0 0 1 0 1 0 0 1 0 0 1 0 1 1 0 0 0 0 0 0 0 0 0 0 0 0
 0 1 0 0 1 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 1 0 1 0 0 0 0 1 1 1 0 0
 0 0 1 0 1 0 1 0 0 0 1 0 0 0 0 0 0 0 1 1 1 0 0 0 0 0 0 0 0 0 0 1 1 0 1 1
```

```
# Normalizando os dados
scaler = StandardScaler()
X = scaler.fit_transform(X)
pd.DataFrame(X)
```

	0	1	2	3	4	5	6	7	8	9	...
0	-2.073335	1.269934	0.984375	1.568466	3.283515	2.652874	2.532475	2.217515	2.255747	2.489734	...
1	-0.353632	1.685955	1.908708	-0.826962	-0.487072	-0.023846	0.548144	0.001392	-0.868652	0.499255	...
2	0.456187	1.566503	1.558884	0.942210	1.052926	1.363478	2.037231	0.939685	-0.398008	1.228676	...
3	0.253732	-0.592687	-0.764464	3.283553	3.402909	1.915897	1.451707	2.867383	4.910919	0.326373	...
4	-1.151816	1.776573	1.826229	0.280372	0.539340	1.371011	1.428493	-0.009560	-0.562450	1.270543	...
...	...	...	...	...	...	...	...	...	...	...	...
564	0.721473	2.060786	2.343856	1.041842	0.219060	1.947285	2.320965	-0.312589	-0.931027	2.782080	...
565	2.085134	1.615931	1.723842	0.102458	-0.017833	0.693043	1.263669	-0.217664	-1.058611	1.300499	...
566	2.045574	0.672676	0.577953	-0.840484	-0.038680	0.046588	0.105777	-0.809117	-0.895587	0.184892	...
567	2.336457	1.982524	1.735218	1.525767	3.272144	3.296944	2.658866	2.137194	1.043695	1.157935	...
568	1.221792	-1.814389	-1.347789	-3.112085	-1.150752	-1.114873	-1.261820	-0.820070	-0.561032	-0.070279	...

569 rows × 29 columns

O *StandardScaler ()* transforma os dados para ter uma média igual a zero e um desvio padrão igual a 1, ou seja, segue a distribuição normal padrão (SND). Ela é útil para os recursos que seguem uma distribuição normal, para que os valores das diferentes características estejam na mesma escala, evitando que recursos com maior variância dominem o processo de treinamento dos algoritmos, prejudicando o desempenho dos recursos com menor variância, ocasionando um possível *overfit.*

Onde os atributos descritivos da massa de dados (x), a escolha do *set de* dados deve-se por esse motivo, a resposta alvo será mais facilmente encontrada no processo de iteração e a discretizacão das mesmas de maligno e benigno para 0 e 1.

Divisão do conjunto de dados em teste e treinamento:

```
# Dividindo o conjunto de dados em treino e teste
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size
=0.3, random_state=42)
```

```
# visualizando o conjunto de treinamento como data frame
pd.DataFrame(X_train).head()
```

	0	1	2	3	4	5	6	7	8	9	...
0	-0.321053	-0.158542	-0.198772	-1.204139	-0.769070	-0.753164	-0.919018	-1.236277	-0.991984	-0.560049	...
1	-0.674768	-0.241747	-0.288361	-1.794101	-0.589220	-0.098925	-0.539588	-1.422476	-0.647506	-0.870802	...
2	-1.235591	0.257479	0.003444	0.479635	1.502076	0.705598	0.363201	1.001751	1.596561	0.508999	...
3	-0.688730	-0.388796	-0.393877	-1.206274	-0.960480	-0.628619	-0.648697	0.063458	-0.868652	-0.761442	...
4	0.916946	-0.196849	-0.232332	-0.277565	-0.698760	-0.741488	-0.631673	-0.538947	-0.678694	-0.213564	...

5 rows × 29 columns

```
[ ] # visualizando o conjunto de teste como data frame
    pd.DataFrame(X_test).head()
```

	0	1	2	3	4	5	6	7	8	9	...
0	-0.160486	-0.448110	-0.491999	0.234114	0.027651	-0.109847	-0.276232	0.413949	0.132176	-0.032743	...
1	0.470149	1.302886	1.351264	-0.446227	-0.027309	0.241064	0.789060	-0.838325	-1.160679	1.384594	...
2	0.044296	0.400820	0.267377	0.913744	0.340350	0.725686	0.824140	0.435855	-0.685782	0.249497	...
3	-0.374576	-0.432457	-0.532101	0.643316	0.516599	-0.142993	-0.539846	-0.002259	1.165609	-0.824604	...
4	-1.128546	-0.713374	-0.716683	0.247636	0.145150	-0.269044	-0.592724	0.023298	0.711976	-0.457547	...

5 rows × 29 columns

```
[ ]  # visualizando os rotulos do conjunto de treinamento como data frame
     pd.DataFrame(y_train).head()
```

	0
0	0
1	0
2	0
3	0
4	0

```
[ ]  # visualizando os rotulos do conjunto de treinamento como data frame
     pd.DataFrame(y_test).head()
```

	0
0	0
1	1
2	1
3	0
4	0

• Explicar a implementação de MLP escolhida, seus hiperparâmetros, e características gerais de uso;

Os hiperparâmetros mais comuns em um MLP incluem o número de camadas ocultas, o número de neurônios em cada camada, a função de ativação utilizada pelos neurônios, o algoritmo de otimização e a taxa de aprendizagem. Além disso, também é comum utilizar técnicas de regularização, como *dropout* e L2 *regularization*, para evitar overfitting, na base de dados não se utilizou a técnica de regularização por ser um *set* de dados com saída binária.

Os hiperparâmetros são, onde:

```
mlp = MLPClassifier(hidden_layer_sizes=(10), solver='adam', alpha=0
.001, max_iter=1000)

print(mlp)
```

- *Hidden_layer_size* é a quantidade de neurônio na camada oculta essa configuração e usada na topologia da MLP, no exemplo utilizado há 10 neurônios ocultos.
- *Solver='adam'* refere-se a um parâmetro que pode ser usado com o algoritmo MLPClassifier ou MLPRegressor da biblioteca Scikit-learn para treinar redes neurais artificiais, este parâmetro é usado para especificar, otimizador e minimizar a função de perda durante o treinamento da rede neural, o 'adam' e baseado em gradiente estocástico, ele funciona bem em conjuntos de dados relativamente grandes, com milhares de amostras de treinamento ou mais, em termos de tempo de treinamento e pontuação de validação.
- *Alpha* é um hiperparâmetros que controla a força da regularização L2 (também conhecida como penalidade de peso) aplicada aos pesos da rede neural. Valores maiores de alpha irão aumentar a intensidade da regularização e, portanto, diminuir a magnitude dos pesos aprendidos pela rede neural.
- *Max_iter* é um parâmetro que controla o número máximo de iterações que o algoritmo de otimização usará para ajustar os pesos da rede neural. O algoritmo otimiza iterativamente os pesos até que a função de perda convirja ou o número máximo de iterações seja alcançado.

• Descrever a metodologia experimental: como os dados foram manipulados entre treino e teste, qual medida de desempenho foi usada, e quais tipos de investigações você realizou;

Utilizou-se a função *train_test_split* que é uma função da biblioteca scikit-learn do Python que divide um conjunto de dados em conjuntos de treinamento e teste para o modelo MLP (Multilayer Perceptron), a ideia é treinar o modelo usando o conjunto de treinamento e, em seguida, avaliar a performance do modelo no conjunto de teste.

A divisão dos dados é feita aleatoriamente e é importante que seja feita de forma a minimizar o potencial de viés na avaliação e validação do modelo, no caso da base utilizada empregou-se um *test_size*=*0.3* (30%) da quantidade dos dados para o treinamento do modelo.

A opção *"random_state"* é um parâmetro da função *train_test_split()* do *scikit-learn* que permite definir uma semente aleatória para a divisão dos dados em conjuntos de treinamento e teste. Essa semente é usada pelo gerador de números pseudoaleatórios, que permite que a divisão dos dados seja reproduzível e consistente, garantindo que a mesma divisão seja obtida sempre que o código for executado com a mesma semente. A investigação foi a mudança da quantidade de neurônios bem como a função de ativação (relu, linear).

```
[ ]  # Dividindo o conjunto de dados em treino e teste
     X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3, random_state=42)
```

Utilizou-se uma matriz de confusão para visualização do desempenho do modelo, onde há as métricas de acurácia bem como a precisão alcançada.

```
[ ]  from sklearn.metrics import classification_report,confusion_matrix
     print(confusion_matrix(y_train,predict_train))
     print(classification_report(y_train,predict_train))
```

```
[[249   0]
 [  3 146]]
              precision    recall  f1-score   support

           0       0.99      1.00      0.99       249
           1       1.00      0.98      0.99       149

    accuracy                           0.99       398
   macro avg       0.99      0.99      0.99       398
weighted avg       0.99      0.99      0.99       398
```

- **Apresentar suas conclusões com relação aos resultados obtidos, e descrever algumas possíveis limitações/vantagem do algoritmo no problema escolhido.**

Devido a base de dados utilizada nos proporcionar uma saída binária a topologia da MLP criada foi relativamente simples com 10 neurônios na camada oculta, porém, posteriores testes nos trouxeram que apenas 3 neurônios seriam capazes de aprender e nos entregar uma acurácia acima dos 98%, se aumentássemos as quantidades de neurônios ocultas corre-se o risco de perda dos gradientes que fazem os ajustes sinápticos proporcionando assim um *overfit* ou que a MLP caísse em um mínimo local. Na atomização usou-se "solver='adam'" se refere ao otimizador de gradiente estocástico. O otimizador "adam" é um algoritmo de otimização que pode ser usado em vez do procedimento clássico de descida de gradiente estocástico para atualizar os pesos da rede neural com base nos dados de treinamento. Quanto a eficiência pelo volume de dados da base utilizada creio que o adam e sobre avaliado, realizou-se alguns testes com as funções "sgd" que manteve alto desempenho como mostra a matrix.

```
# Criando o MLPClassifier
# mlp = MLPClassifier(hidden_layer_sizes=(30, 30, 30), activation='
relu', solver='sgd', alpha=0.001, max_iter=1000)

mlp = MLPClassifier(hidden_layer_sizes=(10), solver='sgd', alpha=0.
001, max_iter=1000)
print(mlp)
```

```
[[246   3]
 [ 13 136]]
              precision    recall  f1-score   support

           0       0.95      0.99      0.97       249
           1       0.98      0.91      0.94       149

    accuracy                           0.96       398
   macro avg       0.96      0.95      0.96       398
weighted avg       0.96      0.96      0.96       398
```

```
# Avaliando o modelo
print("Acurácia:", mlp.score(X_test, y_test))
```

```
Acurácia: 0.9766081871345029
```

Código Fonte:

```
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.neural_network import MLPClassifier
from sklearn.metrics import accuracy_score
from sklearn.preprocessing import LabelEncoder, OneHotEncoder
from sklearn.preprocessing import StandardScaler
from sklearn.model_selection import train_test_split


data = pd.read_csv('https://raw.githubusercontent.com/jefferson-
oliva/databases/main/breast_cancer.csv')


# Atributo alvo é a coluna diagnosis
y = data["diagnosis"]

# Convertendo atributo categórico em numérico
le = LabelEncoder()
y = le.fit_transform(y)
print(y)


# X são os atributos descritivos - excluir o sample id (coluna 0),
e a classe (diagnosis) que é a ultima columa
X = data.iloc[:, 2:len(data.columns)-1]
X.head()


# Normalizando os dados
scaler = StandardScaler()
X = scaler.fit_transform(X)
pd.DataFrame(X)


# Dividindo o conjunto de dados em treino e teste
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size
=0.3, random_state=42)


# Criando o MLPClassifier
# mlp = MLPClassifier(hidden_layer_sizes=(30, 30, 30), activation='
relu', solver='adam', alpha=0.001, max_iter=1000)

mlp = MLPClassifier(hidden_layer_sizes=(10), solver='adam', alpha=0
.001, max_iter=1000)
print(mlp)


# Treinando o modelo
mlp.fit(X_train, y_train)
```

```
from sklearn.metrics import classification_report,confusion_matrix
print(confusion_matrix(y_train,predict_train))
print(classification_report(y_train,predict_train))

# Avaliando o modelo
print("Acurácia:", mlp.score(X_test, y_test))
```