Aula 08 - Exercício prático aprendizado supervisionado

October 17, 2023

1 Implemente o algoritmo KNN, Naive Bayes e Hunt e aplique no dataset IRIS:

https://www.kaggle.com/uciml/iris Não use bibliotecas prontas, implemente toda a lógica do algoritmo. Separe aleatoriamente 70% dos dados para treino e 30% para teste e reporte com um print da saída qual a acurácia do algoritmo (número de acertos).

Aluno: Vitor Albuquerque de Paula

1.1 Pré processamento dos dados

[2]:
```
import pandas as pd
from sklearn.model_selection import train_test_split

url = 'https://gist.githubusercontent.com/netj/8836201/raw/
 ↪6f9306ad21398ea43cba4f7d537619d0e07d5ae3/iris.csv'
iris_data = pd.read_csv(url)

train_data, test_data = train_test_split(iris_data, test_size=0.3,␣
 ↪random_state=42)
```

1.2 Algoritmo KNN no conjunto de dados Iris.

[20]:
```
import numpy as np
import pandas as pd
from sklearn.model_selection import train_test_split
from collections import Counter

# Função para calcular a distância euclidiana
def euclidean_distance(x, y):
    return np.sqrt(np.sum((x - y) ** 2))

# Função para retornar os 'k' vizinhos mais próximos
def get_k_neighbors(k, train, test_sample):
    distances = []
    for i in range(len(train)):
        distances.append((i, euclidean_distance(train.iloc[i, :-1],␣
 ↪test_sample)))
```

```
    distances.sort(key=lambda x: x[1])
    neighbors = [train.iloc[x[0], -1] for x in distances[:k]]
    return neighbors

# Função para fazer previsões usando o KNN
def knn_predict(k, train, test):
    predictions = []
    for _, test_sample in test.iterrows():
        neighbors = get_k_neighbors(k, train, test_sample[:-1])
        majority_class = Counter(neighbors).most_common(1)[0][0]
        predictions.append(majority_class)
    return predictions

# Função para calcular a acurácia das previsões
def accuracy(y_true, y_pred):
    return round(sum([yt == yp for yt, yp in zip(y_true, y_pred)]) /␣
 ↪len(y_true) * 100, 2)

url = 'https://gist.githubusercontent.com/netj/8836201/raw/
 ↪6f9306ad21398ea43cba4f7d537619d0e07d5ae3/iris.csv'
iris_data = pd.read_csv(url)

train_data, test_data = train_test_split(iris_data, test_size=0.3,␣
 ↪random_state=42)

k = 5  # Por exemplo, escolhendo k=5
predictions = knn_predict(k, train_data, test_data)
accuracy = accuracy(test_data.iloc[:, -1], predictions)

print(f"Acurácia do KNN (k={k}): {accuracy}%")
```

```
Acurácia do KNN (k=5): 100.0%
```

1.3 Algoritmo Naive Bayes no conjunto de dados Iris.

[22]:
```
import numpy as np
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import LabelEncoder
from math import exp, sqrt, pi, log

# Função para calcular a função PDF (Probability Density Function - Função␣
 ↪Densidade de Probabilidade) Gaussiana dado x, média e variação
def gauss_pdf(x, mean, var):
    const = sqrt(2 * pi * var)
    exponent = exp(-(x - mean) ** 2 / (2 * var))
    return exponent / const
```

```
# Função renomeada 'accuracy()' para 'get_accuracy()'
def get_accuracy(y_true, y_pred):
    return round(sum([yt == yp for yt, yp in zip(y_true, y_pred)]) /␣
 ↪len(y_true) * 100, 2)

# Função principal para treinar e prever usando Naive Bayes Gaussiano
def naive_bayes(train, test, epsilon=1e-9):
    X_train = train.iloc[:, :-1]  # Divide as características do conjunto de␣
 ↪treinamento
    y_train = train.iloc[:, -1]  # Extrai os rótulos do conjunto de treinamento
    X_test = test.iloc[:, :-1]  # Divide as características do conjunto de teste
    y_test = test.iloc[:, -1]  # Extrai os rótulos do conjunto de teste

    # Codifica os rótulos das classes em valores numéricos
    label_encoder = LabelEncoder().fit(y_train)
    y_train_encoded = label_encoder.transform(y_train)
    label_classes = label_encoder.classes_

    num_features = X_train.shape[1]

    test_class_probs = []
    for index, test_sample in X_test.iterrows():
        class_probs = []
        for label in label_classes:
            class_prob = log(get_class_prob(y_train_encoded, label) + epsilon) ␣
 ↪# Acrescenta epsilon para evitar erro de log(0)

            conditional_probs = []
            for feature_idx, feature in enumerate(test_sample):
                feature_values = X_train[y_train == label].iloc[:, feature_idx]
                mean, var = feature_values.mean(), feature_values.var()

                # Calcula a probabilidade de uma amostra pertencer a uma classe
                # com base na distribuição Gaussiana das características
                likelihood = gauss_pdf(feature, mean, var)
                conditional_probs.append(log(likelihood + epsilon))  #␣
 ↪Acrescenta epsilon para evitar erro de log(0)

            class_probs.append(class_prob + sum(conditional_probs))

        test_class_probs.append(np.argmax(class_probs))

    return label_encoder.inverse_transform(test_class_probs)

url = 'https://gist.githubusercontent.com/netj/8836201/raw/
 ↪6f9306ad21398ea43cba4f7d537619d0e07d5ae3/iris.csv'
```

```
iris_data = pd.read_csv(url)

train_data, test_data = train_test_split(iris_data, test_size=0.3,␣
↪random_state=42)

predictions = naive_bayes(train_data, test_data)
acc = get_accuracy(test_data.iloc[:, -1], predictions)  # Altera 'accuracy'␣
↪para 'get_accuracy'

print(f"Acurácia do Naive Bayes Gaussiano: {acc}%")
```

```
Acurácia do Naive Bayes Gaussiano: 97.78%
```

1.4 Algoritmo Hunt no conjunto de dados Iris.

[19]:
```
import numpy as np
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import KBinsDiscretizer

# Classe para armazenar os nós da árvore de decisão
class TreeNode:
    def __init__(self, feature=None, value=None, class_label=None, left=None,␣
↪right=None):
        self.feature = feature  # Característica usada para dividir o nó
        self.value = value  # Valor associado à característica no nó
        self.class_label = class_label  # Rótulo da classe para nó folha
        self.left = left  # Filho à esquerda da árvore
        self.right = right  # Filho à direita da árvore

# A função de recorrência principal para construir a árvore de decisão usando o␣
↪algoritmo de Hunt
def hunt_algorithm(data, features):
    # Se todas as amostras no conjunto de dados têm a mesma classe, retorne um␣
↪nó folha com esse rótulo de classe
    if len(data['variety'].unique()) == 1:
        return TreeNode(class_label=data['variety'].unique()[0])

    # Se não há mais recursos para dividir, retorne um nó folha com a classe␣
↪mais frequente no conjunto de dados
    if len(features) == 0:
        return TreeNode(class_label=data['variety'].value_counts().idxmax())

    # Encontre o melhor recurso para dividir o conjunto de dados atual
    best_feature = None
    best_gain_ratio = -1
    S = entropy(data)
```

```
    for feature in features:
        gain, split_info = info_gain(data, feature, S)
        gain_ratio = gain / split_info if split_info != 0 else gain

        if gain_ratio > best_gain_ratio:
            best_gain_ratio = gain_ratio
            best_feature = feature

    # Dividir o conjunto de dados no melhor recurso
    left_data = data[data[best_feature] == 0].drop(columns=[best_feature])
    right_data = data[data[best_feature] == 1].drop(columns=[best_feature])
    remaining_features = [f for f in features if f != best_feature]

    # Se algum dos subconjuntos resultantes for vazio, retorne um nó folha com␣
 ↪a classe mais frequente no conjunto de dados
    if left_data.empty or right_data.empty:
        return TreeNode(class_label=data['variety'].value_counts().idxmax())

    # Construa a subárvore esquerda e direita recursivamente e retorne o nó␣
 ↪raiz atual
    left_child = hunt_algorithm(left_data, remaining_features)
    right_child = hunt_algorithm(right_data, remaining_features)

    return TreeNode(feature=best_feature, left=left_child, right=right_child)

# Função para medir a impureza do conjunto de dados (entropia)
def entropy(data):
    prob = data['variety'].value_counts(normalize=True)  # Mudamos 'class' para␣
 ↪'variety'
    return -np.sum(prob * np.log2(prob))

# Função para calcular o ganho de informação ao dividir um conjunto de dados em␣
 ↪um determinado recurso
def info_gain(data, feature, S):
    prob_split = data[feature].value_counts(normalize=True)
    entropy_after_split = -np.sum(prob_split * np.log2(data.
 ↪groupby(feature)['variety'].value_counts(normalize=True)))  # Mudamos␣
 ↪'class' para 'variety'
    gain = S - entropy_after_split
    split_info = -np.sum(prob_split * np.log2(prob_split))
    return gain, split_info

# Função para prever a classe de uma amostra com base na árvore de decisão
def predict_sample(tree, sample):
    if tree.class_label is not None:
        return tree.class_label
```

```
    feature_value = sample[tree.feature]
    return predict_sample(tree.left if feature_value == 0 else tree.right,␣
 ↪sample)

# Função para prever as classes de um conjunto de teste usando a árvore de␣
 ↪decisão
def hunt_predict(tree, test_data):
    return [predict_sample(tree, sample) for _, sample in test_data.iterrows()]

url = 'https://gist.githubusercontent.com/netj/8836201/raw/
 ↪6f9306ad21398ea43cba4f7d537619d0e07d5ae3/iris.csv'
iris_data = pd.read_csv(url)

train_data, test_data = train_test_split(iris_data, test_size=0.3,␣
 ↪random_state=42)

# Discretize Iris features
# Decidi discretizar as características do conjunto de dados Iris porque o␣
 ↪algoritmo Hunt funciona melhor
# com características categóricas.
# A discretização é feita dividindo os valores numéricos das características em␣
 ↪intervalos, convertendo-os
# em valores categóricos.
binning = KBinsDiscretizer(n_bins=3, encode='ordinal', strategy='uniform',␣
 ↪subsample=200_000)
train_data_binned = train_data.copy()
test_data_binned = test_data.copy()
column_names = train_data_binned.columns[:-1]

train_data_binned[column_names] = binning.
 ↪fit_transform(train_data_binned[column_names])
test_data_binned[column_names] = binning.
 ↪transform(test_data_binned[column_names])

columns = train_data_binned.columns[:-1]
decision_tree = hunt_algorithm(train_data_binned, columns)
predictions = hunt_predict(decision_tree, test_data_binned)

acc = get_accuracy(test_data.iloc[:, -1], predictions)

print(f"Acurácia da Árvore de Decisão (algoritmo de Hunt): {acc}%")
```

```
Acurácia da Árvore de Decisão (algoritmo de Hunt): 71.11%
```