TP4_Classification_kafando

November 12, 2023

L'objectif de ce TP est de programmer et de tester deux algorithmes de classification, très simples mais très efficaces : l'algorithme du Plus Proche Voisin (PPV) et le Classifieur Bayesien Naïf (CBN). Nous n'étudions ici que les versions les plus simple de ces algorithmes.

1 Plus Proche Voisin Nearest Neighbor Classifier NNC

1. Créez une fonction PPV(X,Y) qui prend en entrée des données X et des étiquettes Y et qui renvoie une étiquette, pour chaque donnée, prédite à partir du plus proche voisin de cette donnée. Ici on prend chaque donnée, une par une, comme donnée de test et on considère toutes les autres comme données d'apprentissage. Cela nous permet de tester la puissance de notre algorithme selon une méthode de validation par validation croisée (cross validation) de type "leave one out".

[128]:
```
# imports
from sklearn.metrics.pairwise import euclidean_distances
from sklearn.neighbors import KNeighborsClassifier
from sklearn import datasets
import numpy as np
from collections import Counter
```

[129]:
```
def PPV(X, Y):
    predicted_labels = []

    for i in range(len(X)):
        # Utilisez toutes les données sauf la i-ème comme données␣
↪d'apprentissage
        X_train = np.delete(X, i, axis=0)
        Y_train = np.delete(Y, i, axis=0)

        # Calculez les distances euclidiennes entre la donnée de test et les␣
↪données d'apprentissage
        distances = euclidean_distances(X[i].reshape(1, -1), X_train)

        # Trouvez l'indice de la donnée d'apprentissage la plus proche
        closest_id = np.argmin(distances)

        # Obtenez l'étiquette de la donnée la plus proche
```

```
        predicted_label = Y_train[closest_id]

        # Ajoutez l'étiquette prédite à la liste des étiquettes prédites
        predicted_labels.append(predicted_label)

    return np.array(predicted_labels)
```

2. La fonction PPV calcule une étiquette prédite pour chaque donnée. Modifiez la fonction pour calculer et renvoyer l'erreur de prédiction : c'est à dire le pourcentage d'étiquettes mal prédites.

[130]:
```
def PPV(X, Y):
    predicted_labels = []
    errors = 0
    for i in range(len(X)):
        # Utilisez toutes les données sauf la i-ème comme données␣
↪d'apprentissage
        X_train = np.delete(X, i, axis=0)
        Y_train = np.delete(Y, i, axis=0)

        # Calculez les distances euclidiennes entre la donnée de test et les␣
↪données d'apprentissage
        distances = euclidean_distances(X[i].reshape(1, -1), X_train)

        # Trouvez l'indice de la donnée d'apprentissage la plus proche
        closest_id = np.argmin(distances)

        # Obtenez l'étiquette de la donnée la plus proche
        predicted_label = Y_train[closest_id]

        # Ajoutez l'étiquette prédite à la liste des étiquettes prédites
        predicted_labels.append(predicted_label)
        if (predicted_label !=Y[i]):
            errors = errors + 1

    err = errors / len(Y)*100

    return np.array(predicted_labels),err
```

3. Testez sur les données Iris.

[131]:
```
iris = datasets.load_iris()
X = iris.data
Y = iris.target

ppv_resultat,error = PPV(X, Y)

print("Erreur en %:",error)
```

```python
print("Predicted labels: ",ppv_resultat)
```

```
Erreur en %: 4.0
Predicted labels:  [0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0
 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 2 1 2 1
 1 1 1 1 1 1 1 1 1 2 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 2 2 2 2 2 2 1 2 2 2 2
 2 2 2 2 2 2 2 2 1 2 2 2 2 2 2 2 2 2 2 2 2 2 1 2 2 2 2 2 2 2 2 2 2 2 2 2 2
 2 2]
```

4. Testez la fonction des K Plus Proches Voisins de sklearn (avec ici K = 1). Les résultats sont-ils différents ? Testez avec d'autres valeurs de K

[132]:
```python
neigh = KNeighborsClassifier(n_neighbors=1)
neigh.fit(X,Y)
sk_resultat = neigh.predict(X)
sk_resultat
```

[132]:
```
array([0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
       0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
       0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
       1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
       1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2,
       2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2,
       2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2])
```

[133]:
```python
# Les résultats sont-ils différents
print("\n Is equal ? "+str(np.array_equal(sk_resultat, ppv_resultat)))
```

```

 Is equal ? False
```

Testez avec d'autres valeurs de K

[134]:
```python
#K=2
neigh = KNeighborsClassifier(n_neighbors=2)
neigh.fit(X,Y)
sk_resultat = neigh.predict(X)
print(sk_resultat)
print("\n Is equal ? "+str(np.array_equal(sk_resultat, ppv_resultat)))
```

```
[0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 2 2 2 2 2 2 1 2 2 2 2
 2 2 2 2 2 2 2 2 1 2 2 2 2 2 2 2 2 2 2 2 2 2 1 2 2 2 2 2 2 2 2 2 2 2 2 2 2
 2 2]

 Is equal ? False
```

[135]:
```
#K=3
neigh = KNeighborsClassifier(n_neighbors=3)
neigh.fit(X,Y)
sk_resultat = neigh.predict(X)
print(sk_resultat)
print("\n Is equal ? "+str(np.array_equal(sk_resultat, ppv_resultat)))
```

```
[0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 2 1 2 1
 1 1 1 1 1 1 1 1 2 1 1 1 1 1 1 1 1 1 1 1 1 1 1 2 2 2 2 2 1 2 2 2 2
 2 2 2 2 2 2 2 2 1 2 2 2 2 2 2 2 2 2 2 2 1 2 2 2 2 2 2 2 2 2 2 2 2 2
 2 2]

 Is equal ? True
```

Les résultats sont égales pour K=3

5. BONUS : Modifiez la fonction PPV pour qu'elle prenne en entrée un nombre K de voisins (au lieu de 1). La classe prédite sera alors la classe majoritaire parmi les Kvoisins.

[136]:
```
def PPVMK(X, Y, K=1):
    predicted_labels = []
    errors = 0

    for i in range(len(X)):
        # Utilize all data except the i-th as training data
        X_train = np.delete(X, i, axis=0)
        Y_train = np.delete(Y, i, axis=0)

        # Calculate Euclidean distances between the test data and training data
        distances = euclidean_distances(X[i].reshape(1, -1), X_train)

        # Find the indices of the K nearest neighbors
        closest_ids = np.argpartition(distances, K)[:,:K]

        # Get the labels of the K nearest neighbors
        k_nearest_labels = Y_train[closest_ids]

        # Find the majority class among the K nearest neighbors
        predicted_label, _ = mode(k_nearest_labels, axis=1, keepdims=True)

        # Add the predicted label to the list of predicted labels
        predicted_labels.append(predicted_label[0][0])  # Access the actual␣
↪predicted label

        # Check for errors
        if predicted_label != Y[i]:
            errors += 1
```

```
    error_rate = errors / len(Y) * 100

    return list(predicted_labels), error_rate

predicted_labels,error = PPVMK(X, Y, 2)
print("Pourcentage d'erreurs de prédiction :", error)
print("predictions:",predicted_labels)
```

```
Pourcentage d'erreurs de prédiction : 5.333333333333334
predictions: [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 2, 1, 2, 1, 1, 1,
1, 1, 1, 1, 1, 1, 1, 2, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 2, 2, 2,
2, 2, 2, 1, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 1, 2, 2, 2, 2, 2, 2, 2, 2, 2,
2, 2, 2, 1, 1, 2, 2, 2, 1, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2]
```

2 Classifieur Bayesien Naïf

1. Créez une fonction CBN(X,Y) qui prend en entrée des données X et des étiquettes Y et qui renvoie une étiquette, pour chaque donnée, prédite à partir de la classe la plus probable selon l'équation (1). Ici encore, on prend chaque donnée, une par une, comme donnée de test et on considère toutes les données comme données d'apprentissage. Il est conseillé de calculer d'abord les barycentres et les probabilités à priori P (k) pour chaque classe, puis de calculer les probabilités conditionnelles P (xi/ k) pour chaque classe et chaque variable.

[137]:
```
from sklearn.metrics import euclidean_distances

def calculate_centroids_dict(X, Y):
    classes = np.unique(Y)
    centroids_dict = {}

    for c in classes:
        X_c = X[Y == c]
        centroid = np.mean(X_c, axis=0)
        centroids_dict[c] = centroid

    return centroids_dict

def calculate_prior_probs_dict(Y):
    classes = np.unique(Y)
    prior_probs_dict = {}

    for c in classes:
        prior_probs_dict[c] = np.sum(Y == c) / len(Y)

    return prior_probs_dict
```

```python
def calculate_conditional_probs_euclidean(x_test, centroids):
    # Initialize an empty dictionary to store conditional probabilities
    conditional_probs_dict = {}

    for key, centroid in centroids.items():
        # Calculate Euclidean distance between x_test and current centroid
        distance = euclidean_distances(x_test.reshape(1, -1), np.
 ↪array(centroid).reshape(1, -1))[0][0]
        conditional_probs_dict[key] = distance

    # Normalize the distances to get conditional probabilities
    total_distance = sum(conditional_probs_dict.values())
    conditional_probs_dict = {key: distance / total_distance for key, distance␣
 ↪in conditional_probs_dict.items()}

    return conditional_probs_dict

def CBN(X, Y):
    predictions = []

    # Pour chaque donnée de test
    for i in range(len(X)):
        X_train = np.delete(X, i, axis=0)  # Utiliser toutes les données sauf␣
 ↪la i-ème pour l'apprentissage
        Y_train = np.delete(Y, i)
        x_test = X[i]

        centroids = calculate_centroids_dict(X_train, Y_train)
        prior_probs = calculate_prior_probs_dict(Y_train)
        conditional_probs = calculate_conditional_probs_euclidean(x_test,␣
 ↪centroids)

        # Multiply the conditional probabilities with prior probabilities
        prediction = {key: prior_probs[key] * conditional_probs[key] for key in␣
 ↪prior_probs}

        # Get the class with the maximum value as the predicted class
        predicted_class = max(prediction, key=prediction.get)
        predictions.append(predicted_class)

    return predictions
```

2.La fonction CBN calcule une étiquette prédite pour chaque donnée. Modifiez la fonction pour calculer et renvoyer l'erreur de prédiction : c'est à dire le pourcentage d'étiquettes mal prédites. Testez sur les données Iris.

[138]:
```
def CBN(X, Y):
    errors = 0
    predictions = []
    # Pour chaque donnée de test
    for i in range(len(X)):
        X_train = np.delete(X, i, axis=0)  # Utiliser toutes les données sauf␣
↪la i-ème pour l'apprentissage
        Y_train = np.delete(Y, i)
        x_test = X[i]

        centroids = calculate_centroids_dict(X_train, Y_train)
        prior_probs = calculate_prior_probs_dict(Y_train)
        conditional_probs = calculate_conditional_probs_euclidean(x_test,␣
↪centroids)

        # Multiply the conditional probabilities with prior probabilities
        prediction = {key: prior_probs[key] * conditional_probs[key] for key in␣
↪prior_probs}

        # Get the class with the maximum value as the predicted class
        predicted_class = max(prediction, key=prediction.get)
        predictions.append(predicted_class)

        # Check if the predicted class is different from the actual class
        if predicted_class != Y[i]:
            errors += 1

    # Calculate the percentage of errors
    error_percentage = (errors / len(X)) * 100

    return predictions,error_percentage
```

[139]:
```
iris = datasets.load_iris()
X = iris.data
Y = iris.target
predicted_labels,error_percentage = CBN(X, Y)
print("Pourcentage d'erreurs de prédiction :", error_percentage)
print("Predictions:", predicted_labels)
```

```
Pourcentage d'erreurs de prédiction : 100.0
Predictions: [2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2,
2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2,
2, 0, 0, 0, 0, 0, 0, 0, 2, 0, 0, 2, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 2, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 2, 0, 0, 0, 0, 2, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
```

3. Testez la fonction du Classifieur Bayesien Naïf inclut dans sklearn. Cette fonction utilise une

distribution Gaussienne au lieu des distances aux barycentres. Les résultats sont-ils différents ?

[140]:
```
from sklearn.naive_bayes import GaussianNB
def CBNSKlearn(X, Y):
    predictions_gaussian = []
    true_labels = []

    for i in range(len(X)):
        X_train = np.delete(X, i, axis=0)
        Y_train = np.delete(Y, i)
        X_test = X[i]
        true_label = Y[i]

        # Utiliser GaussianNB
        clf = GaussianNB()
        clf.fit(X_train, Y_train)
        pred_gaussian = clf.predict([X_test])[0]
        predictions_gaussian.append(pred_gaussian)

        true_labels.append(true_label)

    return predictions_gaussian, true_labels

# Validation croisée avec GaussianNB et le classifieur personnalisé
pred_gaussian, true_labels = CBNSKlearn(X, Y)

print("Predicted Labels:", pred_gaussian)
```

```
Predicted Labels: [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 1, 1, 2, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 2, 1, 1, 1,
1, 1, 1, 2, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 2,
2, 2, 2, 2, 2, 1, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 1, 2, 2, 2, 2, 2, 2, 2, 2,
2, 2, 2, 2, 2, 1, 1, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2]
```

[141]:
```
print("\n Is equal ? "+str(np.array_equal(pred_gaussian, predicted_labels)))
```

```
 Is equal ? False
```

Les résultats sont différents, la fonction avec une distribution gaussienne est nettement meilleure que le premier

[144]:
```
import numpy as np
from scipy.stats import multivariate_normal

def calculate_conditional_probs_gaussian(x_test, centroids,x_train):
    # Initialize an empty dictionary to store conditional probabilities
```

```
    conditional_probs_dict = {}

    for key, centroid in centroids.items():
        # Calculate the mean and covariance matrix for the Gaussian distribution
        mean = centroid
        cov_matrix = np.cov(np.transpose(x_train))  # Assuming x_train is␣
 ↪available

        # Create a multivariate normal distribution
        mvn = multivariate_normal(mean=mean, cov=cov_matrix)

        # Calculate the probability density for x_test
        prob_density = mvn.pdf(x_test)

        conditional_probs_dict[key] = prob_density

    return conditional_probs_dict

def CBN(X, Y):
    predictions = []

    # Pour chaque donnée de test
    for i in range(len(X)):
        X_train = np.delete(X, i, axis=0)  # Utiliser toutes les données sauf␣
 ↪la i-ème pour l'apprentissage
        Y_train = np.delete(Y, i)
        x_test = X[i]

        centroids = calculate_centroids_dict(X_train, Y_train)
        prior_probs = calculate_prior_probs_dict(Y_train)
        conditional_probs = calculate_conditional_probs_gaussian(x_test,␣
 ↪centroids,X_train)

        # Multiply the conditional probabilities with prior probabilities
        prediction = {key: prior_probs[key] * conditional_probs[key] for key in␣
 ↪prior_probs}

        # Get the class with the maximum value as the predicted class
        predicted_class = max(prediction, key=prediction.get)
        predictions.append(predicted_class)

    return predictions

# Utiliser les fonctions pour prédire les étiquettes
predicted_labels = CBN(X, Y)
print("Predicted Labels:", predicted_labels)
```

```
Predicted Labels: [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0,
0, 0, 0, 1, 2, 1, 1, 1, 1, 2, 1, 1, 1, 1, 2, 1, 1, 1, 1, 2, 1, 1, 1, 2, 1, 1, 1,
1, 1, 1, 2, 1, 1, 1, 1, 1, 1, 2, 2, 2, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 2,
2, 2, 1, 2, 2, 1, 1, 1, 2, 2, 2, 2, 2, 2, 2, 2, 2, 1, 1, 2, 2, 1, 2, 2, 2, 2, 2,
2, 1, 1, 2, 2, 1, 1, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 1, 2, 2, 2]
```

[ ]:

[ ]:

[ ]:

[ ]: