IFT3700_Devoir1_Rapport

November 19, 2021

IFT3700 Devoir 1

Weiyue Cai
Steve Lévesque

1 Introduction

Proposer une notion de similarité originale et spécifiquement construite pour être utilisée avec MNIST et ADULT. L'objectif est d'augmenter la performance de divers algorithmes de partitions. Comparer la performance des algorithmes suivants en utilisant la distance euclidienne et votre mesure proposé.

k-medoïde
Partition binaire (Regroupement hiérarchique)
PcoA (c'est un cas particulier de MDS)
Isomap
Bonus : T-SNE (pour complémenter PcoA et Isomap
KNN

1.1 Persistence des résultats (logs) entre le rapport et le code

Il y a une différence mineur par rapport au temps de génération des graphiques sur le rapport, les logs du code et les images elles-mêmes. Cela est le cas puisqu'une version est démarré en jupyter notebook pour générer le fichier pdf et l'autre sur PyCharm.

Sinon, les résultats des algorithmes et des graphiques (les données) sont reproductibles fidèlement. Un seed (le même pour le rapport et le code) est utilisé.

1.2 Bonus

On a ajouté 2 aspects bonus pour complémenter le devoir : - Utilisation de plusieurs métriques sur chaque similarité et algorithme pour approfondir les possibilités et les explications. - Euclidien - Manhattan - Cosine - Chebyshev - Canberra - Ajout de T-SNE pour complémenter la partie PcoA (MDS) et Isomap.

1.3 Comment avons nous généralement trouvés les informations pour les notions de similarité, le choix (de la métrique) et motivations de nos métriques/similarités?

Nous tenons à dire que nous avons toujours en premier temps généré les résultats pour comparer lequels performent mieux entre-eux. Ceci nous permet d'avoir une idée de début. Nous n'avons pas assez de connaissance des algorithmes, métriques et similarités pour nous lancer et faire des choix éclairés directement.

Ensuite, nous avons fait des recherches plus poussées pour justifier les disparitées et changements dans les résultats. Nous avons ajouté les liens des recherches et du code aux endroits appropriées (le plus proche de la source, avec les fonctions dans le code et rapport, et proche du texte en question dans le rapport).

Finalement, nous avons révisé pour enlever les incohérence (le plus possible) et pour corriger et/ou changer les choix et motivations précédents en lumières de nos nouvelles connaissances avec tous ce qu'on a fait.

NB : Desfois, nous nous sommes permis de référer les notions d'un algorithme précédent puisque le principe est le même pour alléger le rapport au lieu de copier-coller le texte ou juste rephraser pour dire la même chose.

Bonne lecture,

Steve et Weiyue

[1]:
```
# basic
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.image as mpimg
import seaborn as sns
import warnings
warnings.filterwarnings('ignore')
from time import time
import random
from itertools import product
from scipy import ndimage
from matplotlib import offsetbox
from scipy.cluster.hierarchy import dendrogram

# models
from sklearn import decomposition
from sklearn.decomposition import PCA
from sklearn import neighbors
from sklearn.neighbors import KNeighborsClassifier
from sklearn.linear_model import LogisticRegression
from sklearn.cluster import AgglomerativeClustering
from sklearn import manifold
```

```
# metrics
from sklearn import metrics
from sklearn.metrics import accuracy_score
from sklearn.metrics import precision_recall_fscore_support
from sklearn.neighbors import DistanceMetric
from sklearn.metrics.pairwise import euclidean_distances

# models training
from sklearn.pipeline import make_pipeline

# data preparation
from sklearn import datasets
from sklearn.datasets import load_digits
from sklearn.preprocessing import scale
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from sklearn import preprocessing

# data augmentation
from keras.preprocessing.image import ImageDataGenerator
from scipy.ndimage.interpolation import shift
from scipy.ndimage import rotate

# sklearn extra
from sklearn_extra.cluster import KMedoids
```

2 Préparation des données: MNIST

[2]:
```
data1, labels = load_digits(return_X_y=True)
digits = load_digits()
X1 = scale(digits.data)
y1 = digits.target
n_digits = len(np.unique(digits.target))
X1.shape, y1.shape
```

[2]: ((1797, 64), (1797,))

3 Préparation des données: ADULT

[3]:
```
data2 = pd.read_csv('../data/adult.csv')
```

[4]:
```
# categorical attributes
cat_cols_data2 = data2.columns[data2.dtypes == 'object']
cat_cols_data2
```

[4]:
```
Index(['workclass', 'education', 'marital-status', 'occupation',
       'relationship', 'race', 'gender', 'native-country', 'income'],
      dtype='object')
```

[5]:
```
for cat in cat_cols_data2:
  le = preprocessing.LabelEncoder()
  le.fit(data2[cat])
  data2[cat] = le.transform(data2[cat])
```

[6]:
```
data2
```

[6]:

	age	workclass	fnlwgt	education	educational-num	marital-status	\
0	25	4	226802	1	7	4	
1	38	4	89814	11	9	2	
2	28	2	336951	7	12	2	
3	44	4	160323	15	10	2	
4	18	0	103497	15	10	4	
…	…	…	…	…	…	…	
48837	27	4	257302	7	12	2	
48838	40	4	154374	11	9	2	
48839	58	4	151910	11	9	6	
48840	22	4	201490	11	9	4	
48841	52	5	287927	11	9	2	

	occupation	relationship	race	gender	capital-gain	capital-loss	\
0	7	3	2	1	0	0	
1	5	0	4	1	0	0	
2	11	0	4	1	0	0	
3	7	0	2	1	7688	0	
4	0	3	4	0	0	0	
…	…	…	…	…	…	…	
48837	13	5	4	0	0	0	
48838	7	0	4	1	0	0	
48839	1	4	4	0	0	0	
48840	1	3	4	1	0	0	
48841	4	5	4	0	15024	0	

	hours-per-week	native-country	income
0	40	39	0
1	50	39	0
2	40	39	1
3	40	39	1
4	30	39	0
…	…	…	
48837	38	39	0
48838	40	39	1
48839	40	39	0

```
48840              20              39       0
48841              40              39       1

[48842 rows x 15 columns]
```

[7]:
```
X2_all = data2.iloc[:, :-1]
y2_all = data2.iloc[:, -1]
```

[8]:
```
# We use the first 2000 data to do the analyse
X2 = X2_all.iloc[1:2000, :]
y2 = y2_all[1:2000]
```

4 Prétraitement des données: MNIST

Dans l'énonce, il est précisé que si on effectue une légère translation de l'image, cela ne devrait pas affecter sa similarité.
On pourrait augmenter les données par rotation et translation.

[9]:
```
# Code adapted from
# https://towardsdatascience.com/
 ↪improving-accuracy-on-mnist-using-data-augmentation-b5c38eb5a903

# data augmentation by rotating and shifting
def rotate_shift(image, angle, dx, dy):
  image = image.reshape((8, 8))
  rotated_image = rotate(image, angle, reshape=False)
  shifted_rotated_image = shift(rotated_image, [dy, dx], cval=0,␣
 ↪mode="constant")
  return shifted_rotated_image.reshape([-1])

# Creating Augmented Dataset
X1_augmented = [image for image in X1]
y1_augmented = [image for image in y1]

for image, label in zip(X1, y1):
  lower_shift = -0.2
  upper_shift = 0.2
  lower_angle = 0
  upper_angle = 20
  dx = lower_shift + random.random() * (upper_shift - lower_shift)
  dy = lower_shift + random.random() * (upper_shift - lower_shift)
  angle = lower_angle + random.random() * (upper_angle - lower_angle)
  X1_augmented.append(rotate_shift(image, angle, dx, dy))
  y1_augmented.append(label)


# Shuffle the dataset
```

```
shuffle_idx = np.random.permutation(len(X1_augmented))
X1_augmented = np.array(X1_augmented)[shuffle_idx]
y1_augmented = np.array(y1_augmented)[shuffle_idx]

X1_augmented.shape, y1_augmented.shape
```

[9]:
```
((3594, 64), (3594,))
```

5 Fonction de benchmark

Voici les étapes de la fonction benchmark :

1. Créer un pipeline qui va faire une mise à l'échelle des données à l'aide d'un "StandardScaler";
2. Entraîner et compter le temps que le pipeline prends;
3. Mesurer la performance des groupes obtenues avec l'aide de plusieurs métriques.

[10]:
```
# Code adapted from
# A demo of K-Means clustering on the handwritten digits data
# https://scikit-learn.org/stable/auto_examples/cluster/plot_kmeans_digits.html

def benchmark(model, data, labels):
    t0 = time()
    estimator = make_pipeline(StandardScaler(), model).fit(data)
    fit_time = time() - t0
    results = [fit_time]

    # Define the metrics which require only the true labels and estimator
    # labels
    selected_metrics = [
        metrics.accuracy_score,
        metrics.homogeneity_score,
        metrics.completeness_score,
        metrics.v_measure_score
    ]

    results += [each_metric(labels, estimator[-1].labels_) for each_metric in␣
↪selected_metrics]
    results += [metrics.f1_score(labels, estimator[-1].labels_,␣
↪average='macro')]
    # Show the results
    formatter_result = ("{:.3f}\t{:.3f}\t{:.3f}\t{:.3f}\t{:.3f}\t{:.3f}")
    print(formatter_result.format(*results))
```

[11]:
```
def run_benchmark(selected_models, data, labels):
  for i, (model, description) in enumerate(selected_models):
    print(description)
```

```
    print(80 * '_')
    print('time\taccu\thomo\tcompl\tv-meas\tf1')
    each_model = model.fit(data)
    benchmark(model, data, labels)
    print(80 * '_')
```

6 KMedoids (k-medoïde)

[12]:
```
# define KMedoids models by using different metrics
# https://scikit-learn.org/stable/modules/generated/sklearn.neighbors.
 ↪DistanceMetric.html

'''
Valid metrics are ['euclidean', 'l2', 'l1', 'manhattan', 'cityblock',␣
 ↪'braycurtis',
'canberra', 'chebyshev', 'correlation', 'cosine', 'dice', 'hamming', 'jaccard',
'kulsinski', 'mahalanobis', 'matching', 'minkowski', 'rogerstanimoto',␣
 ↪'russellrao',
'seuclidean', 'sokalmichener', 'sokalsneath', 'sqeuclidean', 'yule',␣
 ↪'wminkowski',
'nan_euclidean', 'haversine'],
'''
def KMedoids_models(n):
  selected_KMedoids_models = [
        (
            KMedoids(init='k-medoids++', metric="euclidean", n_clusters=n),
            "KMedoids (euclidean, k-medoids++)"
        ),

        (
            KMedoids(init='k-medoids++', metric="manhattan", n_clusters=n),
            "KMedoids (manhattan, k-medoids++)"
        ),


        (
            KMedoids(init='k-medoids++', metric="cosine", n_clusters=n),
            "KMedoids (cosine, k-medoids++)"
        ),


        (
            KMedoids(init='k-medoids++', metric="chebyshev", n_clusters=n),
            "KMedoids (chebyshev, k-medoids++)"
        ),
```

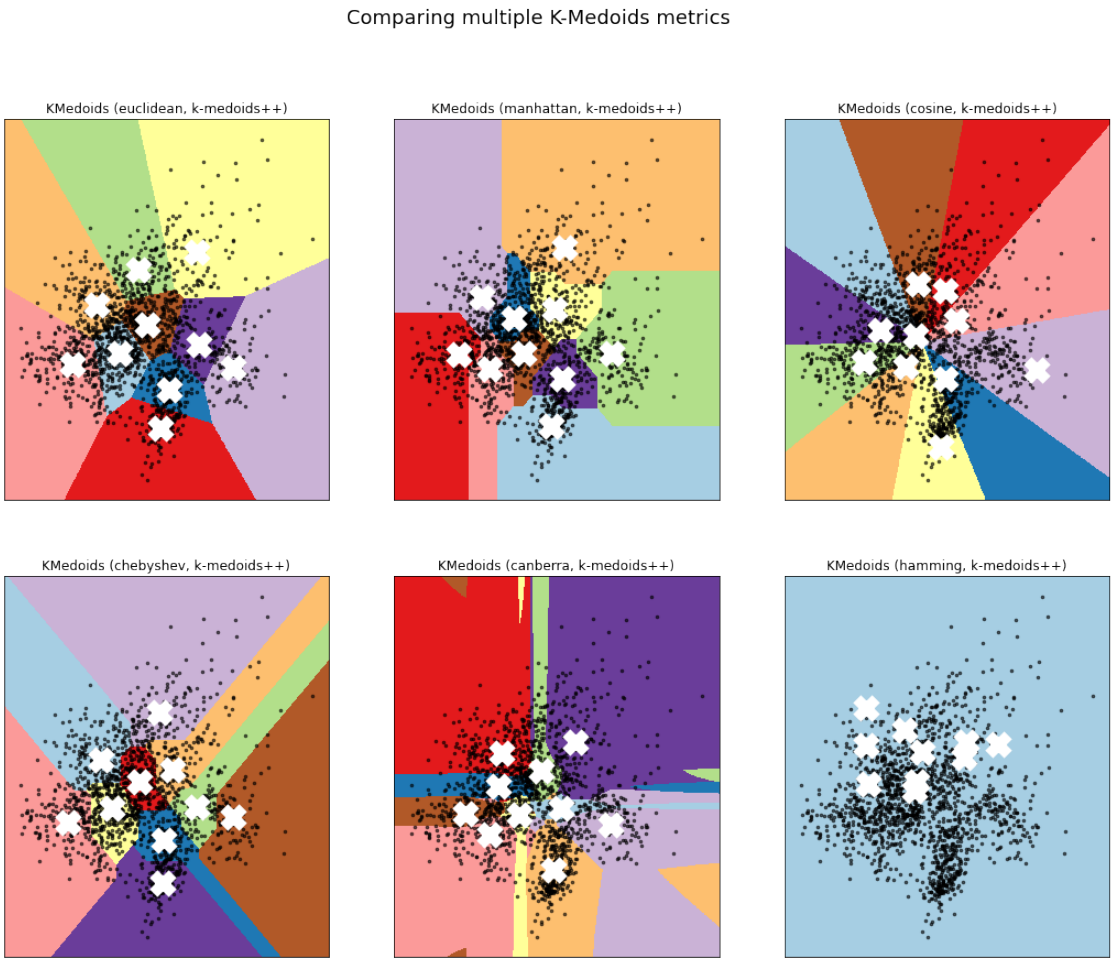
```
        (
            KMedoids(init='k-medoids++', metric="canberra", n_clusters=n),
            "KMedoids (canberra, k-medoids++)"
        ),

        (
            KMedoids(init='k-medoids++', metric="hamming", n_clusters=n),
            "KMedoids (hamming, k-medoids++)"
        )
    ]
  return selected_KMedoids_models
```

[13]:
```
# Visualize and compare different metrics
# Code adapted from: A demo of K-Medoids clustering on the handwritten digits␣
 ↪data
# https://scikit-learn-extra.readthedocs.io/en/stable/auto_examples/cluster/
 ↪plot_kmedoids_digits.
 ↪html#sphx-glr-auto-examples-cluster-plot-kmedoids-digits-py

# Authors: Timo Erkkilä <timo.erkkila@gmail.com>
#          Antti Lehmussola <antti.lehmussola@gmail.com>
#          Kornel Kiełczewski <kornel.mail@gmail.com>
# License: BSD 3 clause

def plot_kmedoids(X, y, n):
    np.random.seed(42)
    data_scale = scale(X)
    reduced_data = PCA(n_components=2).fit_transform(data_scale)

    # Step size of the mesh. Decrease to increase the quality of the VQ.
    h = 0.02  # point in the mesh [x_min, m_max]x[y_min, y_max].

    # Plot the decision boundary. For that, we will assign a color to each
    x_min, x_max = reduced_data[:, 0].min() - 1, reduced_data[:, 0].max() + 1
    y_min, y_max = reduced_data[:, 1].min() - 1, reduced_data[:, 1].max() + 1
    xx, yy = np.meshgrid(np.arange(x_min, x_max, h), np.arange(y_min, y_max, h))

    plt.figure(figsize = (18, 14))
    plt.clf()

    plt.suptitle(
        "Comparing multiple K-Medoids metrics",
        fontsize=18,
    )

    models = KMedoids_models(n)
    plot_rows = 3
```

```
    plot_cols = int(np.ceil(len(models) / 3.0))

    for i, (model, description) in enumerate(models):
        # Obtain labels for each point in mesh. Use last trained model.
        model.fit(reduced_data)
        Z = model.predict(np.c_[xx.ravel(), yy.ravel()])

        # Put the result into a color plot
        Z = Z.reshape(xx.shape)
        plt.subplot(plot_cols, plot_rows, i + 1)
        plt.imshow(
            Z,
            interpolation="nearest",
            extent=(xx.min(), xx.max(), yy.min(), yy.max()),
            cmap=plt.cm.Paired,
            aspect="auto",
            origin="lower",
        )

        plt.plot(
            reduced_data[:, 0], reduced_data[:, 1], "k.", markersize=5, alpha=0.
↪6
        )

        # Plot the centroids as a white X
        centroids = model.cluster_centers_
        plt.scatter(
            centroids[:, 0],
            centroids[:, 1],
            marker="x",
            s=300,
            linewidths=10,
            color="w",
            zorder=10,
        )
        plt.title(description)
        plt.xlim(x_min, x_max)
        plt.ylim(y_min, y_max)
        plt.xticks(())
        plt.yticks(())


    plt.show()
```

##MNIST

[14]:
```
plot_kmedoids(X1, y1, 10)
```



[15]:
```
# Run the benchmark on original data (X_train and y_train)
run_benchmark(KMedoids_models(10), X1, y1)
```

```
KMedoids (euclidean, k-medoids++)

--------------------------------------------------------------------------------
time    accu    homo    compl   v-meas  f1
0.076   0.206   0.597   0.658   0.626   0.167

--------------------------------------------------------------------------------
KMedoids (manhattan, k-medoids++)

--------------------------------------------------------------------------------
time    accu    homo    compl   v-meas  f1
0.213   0.165   0.509   0.538   0.523   0.130

--------------------------------------------------------------------------------
KMedoids (cosine, k-medoids++)

--------------------------------------------------------------------------------
time    accu    homo    compl   v-meas  f1
0.095   0.029   0.638   0.644   0.641   0.029

--------------------------------------------------------------------------------
```

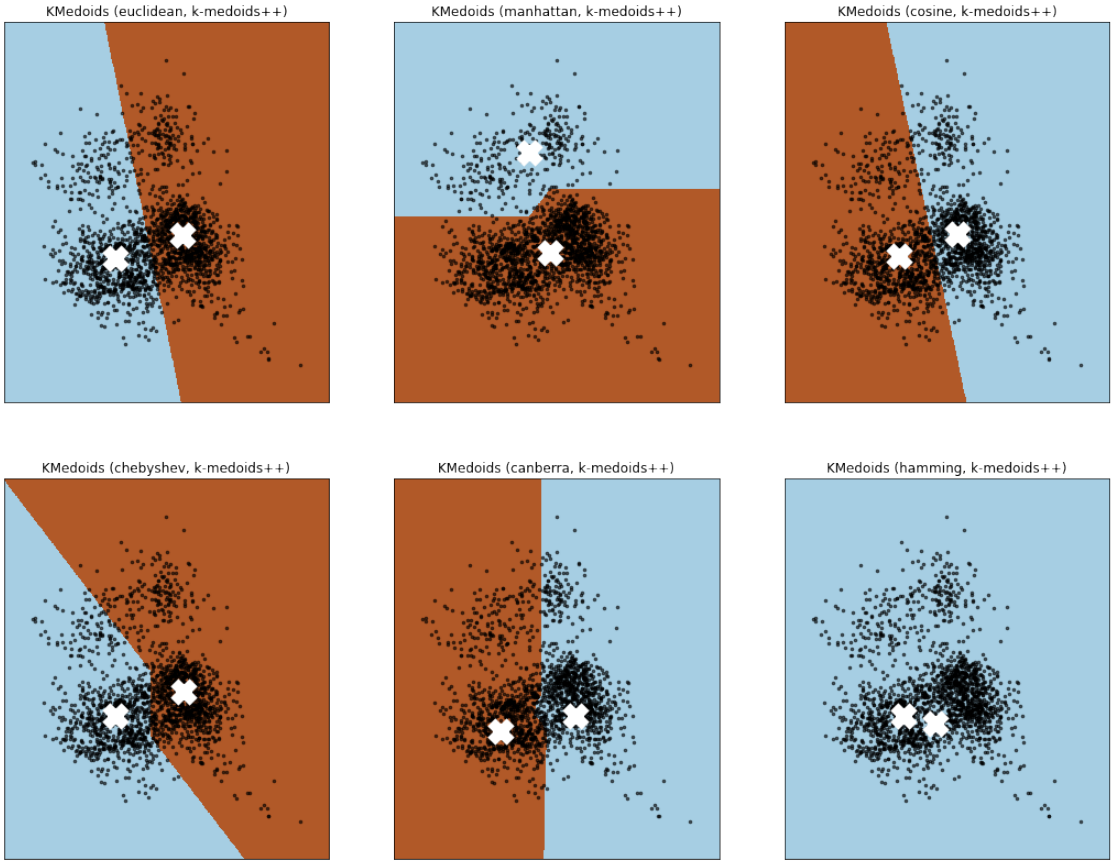
```
KMedoids (chebyshev, k-medoids++)

--------------------------------------------------------------------------------
time    accu    homo    compl   v-meas  f1
0.173   0.076   0.176   0.244   0.204   0.057

--------------------------------------------------------------------------------
KMedoids (canberra, k-medoids++)

--------------------------------------------------------------------------------
time    accu    homo    compl   v-meas  f1
0.165   0.014   0.566   0.583   0.575   0.014

--------------------------------------------------------------------------------
KMedoids (hamming, k-medoids++)

--------------------------------------------------------------------------------
time    accu    homo    compl   v-meas  f1
0.328   0.026   0.362   0.393   0.377   0.030

--------------------------------------------------------------------------------
```

[16]:
```
# Run the benchmark on augmented data (X_augmented, y_augmented)
run_benchmark(KMedoids_models(10), X1_augmented, y1_augmented)
```

```
KMedoids (euclidean, k-medoids++)

--------------------------------------------------------------------------------
time    accu    homo    compl   v-meas  f1
0.503   0.113   0.485   0.531   0.507   0.111

--------------------------------------------------------------------------------
KMedoids (manhattan, k-medoids++)

--------------------------------------------------------------------------------
time    accu    homo    compl   v-meas  f1
0.874   0.045   0.516   0.545   0.530   0.041

--------------------------------------------------------------------------------
KMedoids (cosine, k-medoids++)

--------------------------------------------------------------------------------
time    accu    homo    compl   v-meas  f1
0.460   0.291   0.558   0.566   0.562   0.278

--------------------------------------------------------------------------------
KMedoids (chebyshev, k-medoids++)

--------------------------------------------------------------------------------
time    accu    homo    compl   v-meas  f1
0.653   0.085   0.113   0.199   0.144   0.039

--------------------------------------------------------------------------------
KMedoids (canberra, k-medoids++)

--------------------------------------------------------------------------------
time    accu    homo    compl   v-meas  f1
0.730   0.118   0.180   0.184   0.182   0.113

--------------------------------------------------------------------------------
KMedoids (hamming, k-medoids++)

--------------------------------------------------------------------------------
time    accu    homo    compl   v-meas  f1
0.978   0.102   0.115   0.123   0.119   0.092
```

##ADULT

[17]:
```
plot_kmedoids(X2, y2, 2)
```



[18]:
```
run_benchmark(KMedoids_models(2), X2, y2)
```

```
KMedoids (euclidean, k-medoids++)

--------------------------------------------------------------------------------
time    accu    homo    compl   v-meas  f1
0.176   0.438   0.078   0.064   0.070   0.352

--------------------------------------------------------------------------------
KMedoids (manhattan, k-medoids++)

--------------------------------------------------------------------------------
time    accu    homo    compl   v-meas  f1
0.154   0.596   0.092   0.074   0.082   0.580

--------------------------------------------------------------------------------
```

```
KMedoids (cosine, k-medoids++)

--------------------------------------------------------------------------------
time    accu    homo    compl   v-meas  f1
0.201   0.390   0.098   0.079   0.087   0.326

--------------------------------------------------------------------------------
KMedoids (chebyshev, k-medoids++)

--------------------------------------------------------------------------------
time    accu    homo    compl   v-meas  f1
0.201   0.280   0.011   0.015   0.013   0.273

--------------------------------------------------------------------------------
KMedoids (canberra, k-medoids++)

--------------------------------------------------------------------------------
time    accu    homo    compl   v-meas  f1
0.162   0.592   0.015   0.012   0.013   0.541

--------------------------------------------------------------------------------
KMedoids (hamming, k-medoids++)

--------------------------------------------------------------------------------
time    accu    homo    compl   v-meas  f1
0.179   0.492   0.060   0.052   0.056   0.376

--------------------------------------------------------------------------------
```

7 K-Médoïdes - Similarité

7.0.1 Notion de similarité :

MNIST Cosine: le cosinus de l'angle entre deux vecteurs dans un espace à n dimensions. C'est le produit scalaire des deux vecteurs divisé par le produit des normes des deux vecteurs.

- La similarité d'image est une tâche principalement liée à la sélection des caractéristiques de l'image. Ceci est similaire aux "word embedding" où la distance peut être calculée.
- Le cosine est une méthode classique pour mesurer la similarité des mots et des textes. De même, l'utilisation de cosine pour calculer la similarité d'images pourrait être un bon choix. (https://stackoverflow.com/questions/64901547/calculate-distance-between-images)
- Une propriété de cosine est qu'il peut produire un petit angle (respectivement une similarité haute) pour deux données qui sont loins l'une de l'autre. Cela est pertinent dans notre cas avec le MNIST puisque nous traitons des caractères typographiques pouvant se ressembler fortement et ne pas être dans la même classe, et l'inverse aussi possible.

ADULT Chebyshev: la distance entre deux points comme différence maximale sur n'importe quelle de valeur d'axe (x ou y dans le cas 2D)

- En visualisant les données de manière attentive, nous pouvons apprendre que les caractéristiques ("features") ne sont pas toutes égales dans le sens que la plupart sont catégorielles et certains autres continues.
- Parmis les "features" catégoriels et continues de ADULT, on peut constater qu'en appliquant une transformation pour distinguer numériquement ceux étant catégoriels qu'il y a une grande

différence avec les énumérations (i.e martial-status entre 0 et ~5) et les valeurs continues (i.e. l'argent/capital dans les 0 jusqu'à 10k+).

- En distinguant d'avantages les "features" continues, il est possible d'en trouver un qui peut fortement agir sur les données en comparaison à d'autres avec la propriété de Chebyshev (rappelons nous, le maximum) qui est le "capital-gain". Celui-ci est généralement important par rapport au classement "income".

7.0.2 Clarté des explications:

Se référer au(x) graphique(s) pour des détails aux explications et au fichier README fourni dans le code pour reproduire fidèlement tous les résultats.

7.0.3 Motivations :

MNIST Il est pertinent d'utiliser la similarité d'image puisque nous voulons comparer celle-ci avec des données qui représente un caractère typographique où une similarité graphique est très importante.

Nous avons choisi cette similarité (cosine) puisque nous avons fait la comparaison avec 5 autres similarités en utilisant 5 métriques pour s'assurer qu'il n'y ait pas un alternatif meilleur que nos idées préalables. Il s'avère que cosine est plus rapide, performant en termes de métriques.

i.e. deux caractères pouvant être proches l'un de l'autre comme 3 et 7 si écrit similairement mais n'étant pas de la même classe, l'inverse aussi où les 2 caractères ne sont pas dans le même groupement mais ceux-ci ont une similarité haute.

ADULT Comme mentionné dans la section "Notion de similarité", Chebyshev calcule la distance maximale parmi toutes les dimensions (features) entre deux points. Il est pertinent de l'utiliser avec les notions que nous avons citées.

Puisque nous avons transformé les "features" catégoriels en énumérations de nombres à incrément d'une unité (0,1,2,..), il ne faut pas les prendre en compte numériquement pour éviter le biais (i.e. 0 d'une catégorie n'est pas nécessairement égal à 0 d'une autre).

Un "feature" avec beaucoup d'influence et une classification binaire (donc seulement 2 choix) ne nécessite pas de vérifier plusieurs ambiguïtés ou de trancher dans la "zone grise" comme MNIST.

7.0.4 Analyse des résultats :

MNIST Pour tester K-Médoïdes, nous avons utilisé 6 similarités différentes avec l'aide de 5 métriques pour les appuyer ainsi qu'avec une version augmentée des données.

En observant les résultats, nous pouvons conclure que cosine est le plus performant globalement en comparant toutes les métriques.

ADULT En observant les résultats, nous pouvons conclure que chebyshev est le plus performant au niveau des métriques de classification (accuracy et f1).

Par contre, chebyshev est faible au niveau des métriques de partitionnement où les données ne portent pas attention à des caractéristiques propres entre elles.

8 Hierarchical Clustering (Partition binaire / Regroupement hiérarchique)

[19]:
```
# define Hierarchical Clustering models by using different metrics
# https://scikit-learn.org/stable/modules/generated/sklearn.cluster.
 ↪AgglomerativeClustering.html
# https://scikit-learn.org/stable/modules/clustering.
 ↪html#hierarchical-clustering
def agglomerative_clustering(n):
  selected_HierarchicalClustering_models = [
      (
          AgglomerativeClustering(n_clusters=n, affinity='euclidean',␣
 ↪linkage='average'),
          "Hierarchical Clustering (euclidean)",
      ),


      (
          AgglomerativeClustering(n_clusters=n, affinity='manhattan',␣
 ↪linkage='average'),
          "Hierarchical Clustering (manhattan)",
      ),


      (
          AgglomerativeClustering(n_clusters=n, affinity='cosine',␣
 ↪linkage='average'),
          "Hierarchical Clustering (cosine)",
      ),

      (
          AgglomerativeClustering(n_clusters=n, affinity='canberra',␣
 ↪linkage='average'),
          "Hierarchical Clustering (canberra)",
      ),

  ]
  return selected_HierarchicalClustering_models
```

8.1 MNIST

[20]:
```
# the benchmark on original data (X_train, y_train)
run_benchmark(agglomerative_clustering(10), X1, y1)
```

```
Hierarchical Clustering (euclidean)

--------------------------------------------------------------------------------
time    accu    homo    compl   v-meas  f1
```

```
0.167   0.100   0.007   0.238   0.014   0.019

--------------------------------------------------------------------------------
Hierarchical Clustering (manhattan)

--------------------------------------------------------------------------------
time    accu    homo    compl   v-meas  f1
0.158   0.102   0.018   0.301   0.033   0.024

--------------------------------------------------------------------------------
Hierarchical Clustering (cosine)

--------------------------------------------------------------------------------
time    accu    homo    compl   v-meas  f1
0.158   0.006   0.678   0.763   0.718   0.007

--------------------------------------------------------------------------------
Hierarchical Clustering (canberra)

--------------------------------------------------------------------------------
time    accu    homo    compl   v-meas  f1
0.225   0.095   0.607   0.791   0.687   0.062

--------------------------------------------------------------------------------
```

[21]:
```
# run the benchmark on augmented data
run_benchmark(agglomerative_clustering(10), X1_augmented, y1_augmented)
```

```
Hierarchical Clustering (euclidean)

--------------------------------------------------------------------------------
time    accu    homo    compl   v-meas  f1
0.777   0.099   0.004   0.268   0.008   0.018

--------------------------------------------------------------------------------
Hierarchical Clustering (manhattan)

--------------------------------------------------------------------------------
time    accu    homo    compl   v-meas  f1
0.693   0.100   0.004   0.244   0.007   0.020

--------------------------------------------------------------------------------
Hierarchical Clustering (cosine)

--------------------------------------------------------------------------------
time    accu    homo    compl   v-meas  f1
0.709   0.008   0.535   0.681   0.599   0.006

--------------------------------------------------------------------------------
Hierarchical Clustering (canberra)

--------------------------------------------------------------------------------
time    accu    homo    compl   v-meas  f1
0.895   0.092   0.113   0.152   0.129   0.068

--------------------------------------------------------------------------------
```

Nous pouvons observer que quand on utilise les données originales et cosinus comme métrique, l'algorithme est plus performante au niveau des métriques de clustering (homogénéité, complétude et v-mesures).

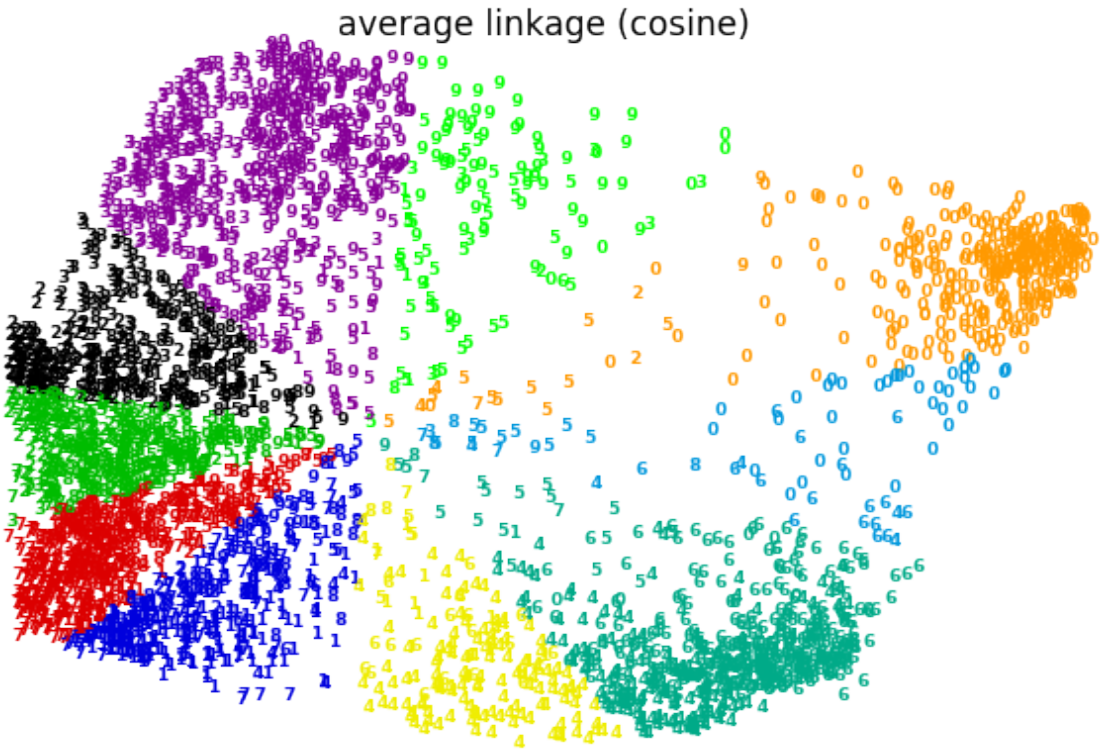
[22]:
```
# Visualize Hierarchical Clustering
# code adapted from
```

```
# Various Agglomerative Clustering on a 2D embedding of digits
# https://scikit-learn.org/stable/auto_examples/cluster/plot_digits_linkage.
 ↪html#sphx-glr-auto-examples-cluster-plot-digits-linkage-py

np.random.seed(42)

def nudge_images(X, y):
    # Having a larger dataset shows more clearly the behavior of the
    # methods, but we multiply the size of the dataset only by 2, as the
    # cost of the hierarchical clustering methods are strongly
    # super-linear in n_samples
    shift = lambda x: ndimage.shift(x.reshape((8, 8)),
                                    .3 * np.random.normal(size=2),
                                    mode='constant',
                                    ).ravel()
    X = np.concatenate([X, np.apply_along_axis(shift, 1, X)])
    Y = np.concatenate([y, y], axis=0)
    return X, Y


X11, y11 = nudge_images(X1, y1)

# Visualize the clustering
def plot_clustering(X_red, y, labels, n, title=None):
    x_min, x_max = np.min(X_red, axis=0), np.max(X_red, axis=0)
    X_red = (X_red - x_min) / (x_max - x_min)

    plt.figure(figsize=(8, 6))
    for i in range(X_red.shape[0]):
        plt.text(X_red[i, 0], X_red[i, 1], str(y[i]),
                 color=plt.cm.nipy_spectral(labels[i] / (n*1.0)),
                 fontdict={'weight': 'bold', 'size': 9})

    plt.xticks([])
    plt.yticks([])
    if title is not None:
        plt.title(title, size=17)
    plt.axis('off')
    plt.tight_layout(rect=[0, 0.03, 1, 0.95])
```

[23]:
```
# We got a better result when using cosine and original data
X1_red = manifold.SpectralEmbedding(n_components=2).fit_transform(X11)
clustering_cosine = AgglomerativeClustering(10, affinity="cosine",␣
 ↪linkage='average')
t0 = time()
clustering_cosine.fit(X1_red)
print("average linkage (manhattan) :\t%.2fs" % (time() - t0))
```

```
plot_clustering(X1_red, y11, clustering_cosine.labels_, 10, "average linkage␣
↪(cosine)")
plt.show()
```

```
average linkage (manhattan) :   0.27s
```



8.2 ADULT

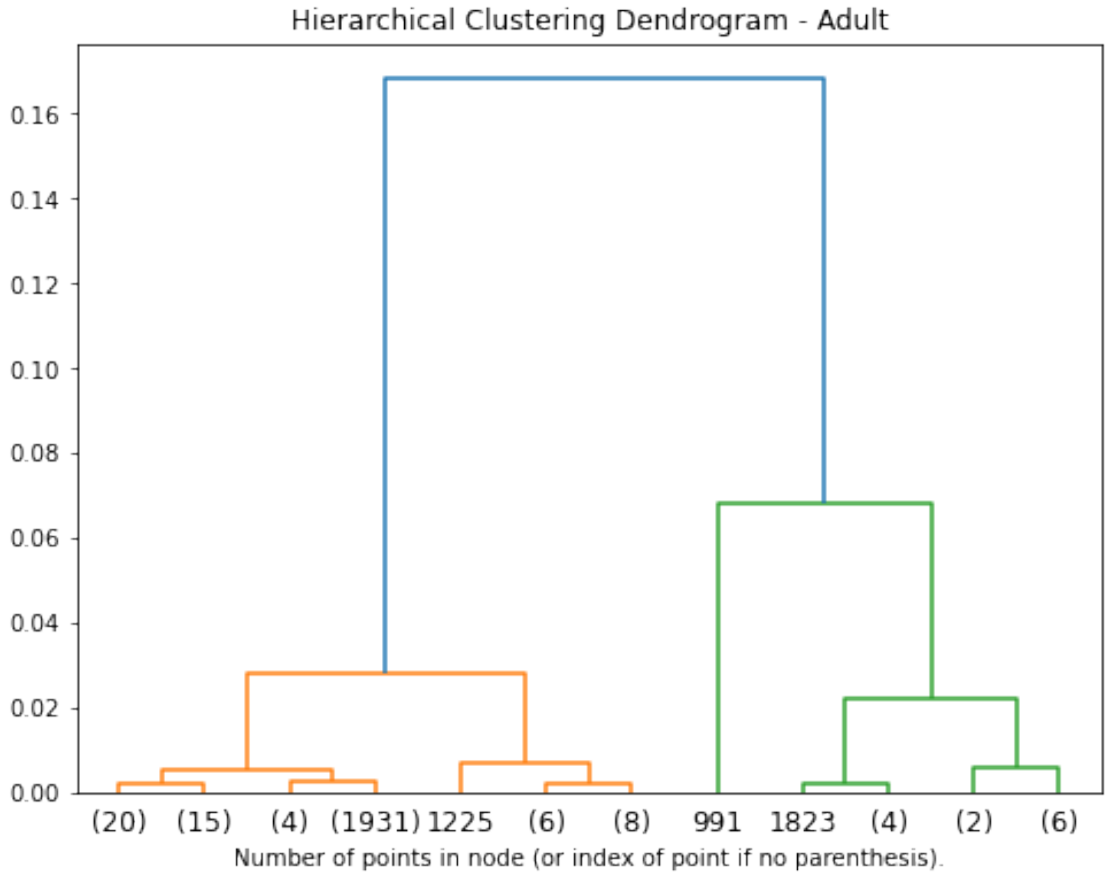
[24]:
```
run_benchmark(agglomerative_clustering(2), X2, y2)
```

```
Hierarchical Clustering (euclidean)

--------------------------------------------------------------------------------
time    accu    homo    compl   v-meas  f1
0.130   0.767   0.020   0.244   0.036   0.464

--------------------------------------------------------------------------------
Hierarchical Clustering (manhattan)

--------------------------------------------------------------------------------
time    accu    homo    compl   v-meas  f1
0.126   0.767   0.020   0.244   0.036   0.464

--------------------------------------------------------------------------------
Hierarchical Clustering (cosine)

--------------------------------------------------------------------------------
```

```
time    accu    homo    compl   v-meas  f1
0.118   0.742   0.127   0.111   0.118   0.684

--------------------------------------------------------------------------------
Hierarchical Clustering (canberra)

--------------------------------------------------------------------------------
time    accu    homo    compl   v-meas  f1
0.188   0.758   0.000   0.035   0.001   0.431

--------------------------------------------------------------------------------
```

[25]:
```python
# https://scikit-learn.org/stable/auto_examples/cluster/
↪plot_agglomerative_dendrogram.html
def plot_dendrogram(model, **kwargs):
    # Create linkage matrix and then plot the dendrogram

    # create the counts of samples under each node
    counts = np.zeros(model.children_.shape[0])
    n_samples = len(model.labels_)
    for i, merge in enumerate(model.children_):
        current_count = 0
        for child_idx in merge:
            if child_idx < n_samples:
                current_count += 1  # leaf node
            else:
                current_count += counts[child_idx - n_samples]
        counts[i] = current_count

    linkage_matrix = np.column_stack(
        [model.children_, model.distances_, counts]
    ).astype(float)

    # Plot the corresponding dendrogram
    dendrogram(linkage_matrix, **kwargs)
```

[26]:
```python
# setting distance_threshold=0 ensures we compute the full tree.
model2_reg_hie = AgglomerativeClustering(distance_threshold=0,␣
↪affinity="cosine", linkage="average", n_clusters=None)

model2_reg_hie = model2_reg_hie.fit(X2)
plt.figure(figsize=(8,6))
plt.title("Hierarchical Clustering Dendrogram - Adult")
# plot the top three levels of the dendrogram
plot_dendrogram(model2_reg_hie, truncate_mode="level", p=3)
plt.xlabel("Number of points in node (or index of point if no parenthesis).")
plt.show()
```



9 Regroup. hiérarchique - Similarité

9.0.1 Notion de similarité :

MNIST Cosine

Idem à K-Médoïdes.

ADULT Cosine

- Le regroupement hiérarchique fait des regroupements binaires de données similaires. De ce fait, le problème revient plus ou moins à des regroupements ("clusters") représentants plusieurs classes comme MNIST car il s'agit de la partition binaire dans les deux cas.
- La majorité des caractéristiques du jeu de données ADULT est catégorielle. Une similarité classique de distance (comme euclidien et manhattan) ne devrait pas pouvoir être pertinente contrairement aux autres choix lors de faire le regroupement hiérarchique.
- Avec les énumérations catégorielles au format numérique, le même index ne veut pas nécessairement dire la même chose. Aussi, celles-ci ne sont pas des distances à priori.

9.0.2 Clarté des explications:

Se référer au(x) graphique(s) pour des détails aux explications et au fichier README fourni dans le code pour reproduire fidèlement tous les résultats.

9.0.3 Motivations :

MNIST Idem à K-Médoïdes.

ADULT Les autres similarités performent mieux à côté de la complétude comparé à cosine. Cependant, cette métrique n'est pas pertinente dans notre cas puisque nos données ADULT sont par rapport à la classification binaire. Donc, nous voulons prioriser le score f1 et la précision, ce que cosine excelle en comparaison.

Du même ordre d'idées que précédemment, la complétude n'est pas pertinente puisque nous avons des "features" d'énumérations en format numérique qui peuvent biaiser les résultats.

Cosine est plus performant en terme de performance pour (~ -0.02 précision) et même supérieur pour les métriques que nous n'avons pas besoin.

9.0.4 Analyse des résultats :

MNIST Il est possible de voir en analysant les résultats que Cosine a une précision terrible en comparaison aux autres similarités. Cependant, nous nous intéressons à l'homogénéité, la complétude et la v-mesure.

Une complétude plus basse pour cosine contrairement à canberra (~ -0.03) est acceptable si nous observons le temps d'exécution qui est 2 fois moindre.

ADULT Cosine n'est pas le plus précis ($0.742 < 0.767$), mais nous pouvons observer que le score f1 est relativement plus haut que le reste des similarités (~ +20%). Ce que nous voulons basant sur la notion et la motivation que nous appuyons.

10 PcoA

Aussi appelé classique ou métrique MDS.

10.1 MNIST

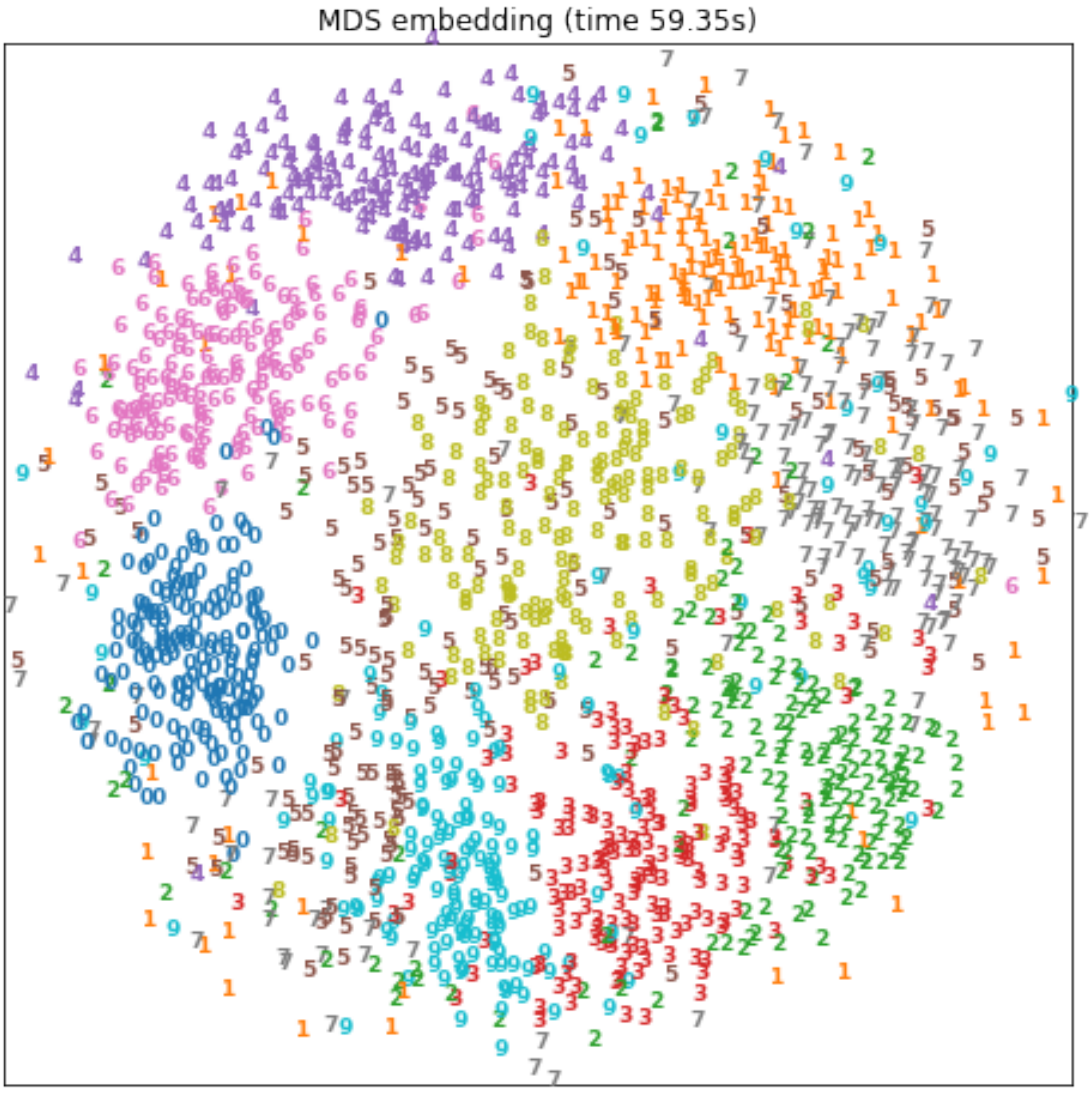
[27]:
```
# code adpated from Manifold learning on handwritten digits
# https://www.programcreek.com/python/example/102441/matplotlib.offsetbox.
 ↪AnnotationBbox

def plot_embedding(X, y, n, title=None):
    x_min, x_max = np.min(X, 0), np.max(X, 0)
    X = (X - x_min) / (x_max - x_min)

    plt.figure(figsize = (8, 8))
    ax = plt.subplot(111)
    for i in range(X.shape[0]):
```

```
        plt.text(X[i, 0], X[i, 1], str(y[i]),
                 color=plt.cm.tab10(y[i] / (n*1.0)),
                 fontdict={'weight': 'bold', 'size': 9})

    if hasattr(offsetbox, 'AnnotationBbox'):
        # only print thumbnails with matplotlib > 1.0
        shown_images = np.array([[1., 1.]])  # just something big
        for i in range(X.shape[0]):
            dist = np.sum((X[i] - shown_images) ** 2, 1)
            if np.min(dist) < 4e-3:
                # don't show points that are too close
                continue
            shown_images = np.r_[shown_images, [X[i]]]

    plt.xticks([]), plt.yticks([])
    if title is not None:
        plt.title(title)
```

[28]:
```
digits = load_digits()
X1_org = digits.data
y1_org = digits.target
```

[29]:
```
# MDS  embedding of the digits dataset
# dissimilarity = 'euclidean'
'''
dissimilarity{‘euclidean’, ‘precomputed’}, default=’euclidean’
Dissimilarity measure to use:

‘euclidean’:
Pairwise Euclidean distances between points in the dataset.

‘precomputed’:
Pre-computed dissimilarities are passed directly to fit and fit_transform.

'''

print("Computing MDS embedding")
np.random.seed(42)
mds1_l2 = manifold.MDS(n_components=2, metric=True, n_init=5, max_iter=100,␣
 ↪dissimilarity='euclidean')
t0 = time()
mds1_l2.fit(X1_org.copy())
X1_mds_l2 = mds1_l2.embedding_

plot_embedding(X1_mds_l2, y1_org, 10,
               "MDS embedding (time %.2fs)" %
               (time() - t0))
```

```
Computing MDS embedding
```



[30]:
```
print("Classification score by using KNN and MDS with L2 as metric")
model1_mds_l2 = KNeighborsClassifier()
model1_mds_l2.fit(X1_mds_l2, np.array(y1))
model1_mds_l2.score(X1_mds_l2, np.array(y1))
```

```
Classification score by using KNN and MDS with L2 as metric
```

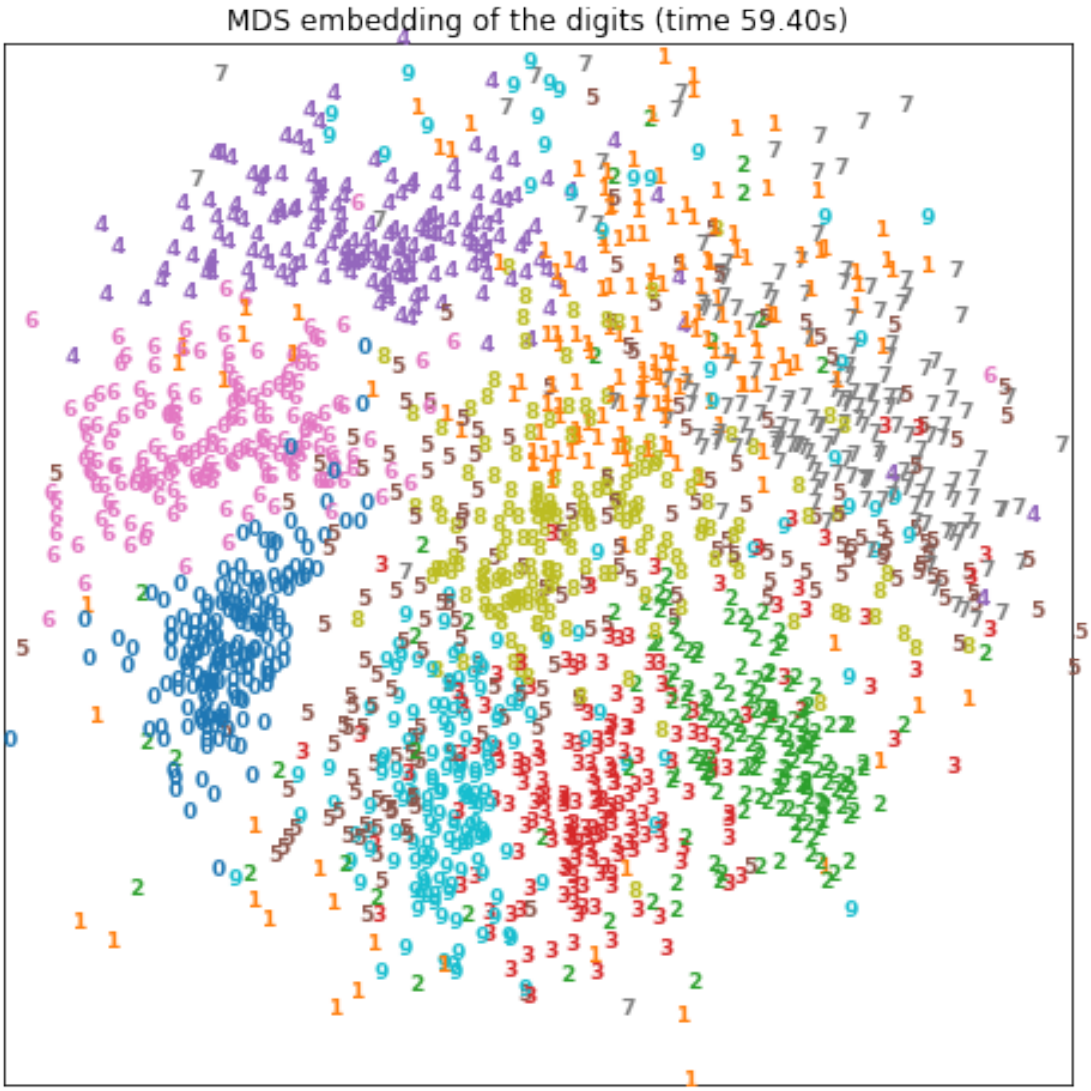
[30]:
```
0.7890929326655537
```

[31]:
```
# https://towardsdatascience.com/
↪multidimensional-scaling-mds-for-dimensionality-reduction-and-data-visualization-d5252c8bc4
```

```
def MDS_custom_distance(X, y, n, distance, title, plot=False, cosine=False):
    if cosine == False:
        dist = DistanceMetric.get_metric(distance)
        dist_matrix = dist.pairwise(X)
    else:
        dist_matrix = metrics.pairwise.cosine_distances(X)

    np.random.seed(42)
    mds = manifold.MDS(n_components=2, metric=True, n_init=5, max_iter=100,␣
 ↪dissimilarity='precomputed')
    t0 = time()
    mds.fit(dist_matrix)
    x_mds = mds.embedding_

    model_mds = KNeighborsClassifier()
    model_mds.fit(x_mds, y)

    print(title)
    print("Classification score by using KNN and MDS")
    if plot == True:
      plot_embedding(x_mds, y, n, "MDS embedding of the digits (time %.2fs)" %␣
 ↪(time() - t0))
    print(model_mds.score(x_mds, y))
```

[32]:
```
MDS_custom_distance(X1_org.copy(), y1_org.copy(), 10, "", 'cosine_distance',␣
 ↪True, True)
```

```
cosine_distance
Classification score by using KNN and MDS
0.7729549248747913
```



10.2 ADULT

[33]:
```
print("Computing MDS embedding")
np.random.seed(42)
mds2_l2 = manifold.MDS(n_components=2, metric=True, n_init=5, max_iter=100,␣
 ↪dissimilarity='euclidean')
t0 = time()
X2_mds_l2 = mds2_l2.fit_transform(X2.copy())
model_mds_l2 = KNeighborsClassifier()
model_mds_l2.fit(X2_mds_l2, np.array(y2.copy()))
model_mds_l2.score(X2_mds_l2, np.array(y2.copy()))
```

```
Computing MDS embedding
```

[33]: 0.8014007003501751

[34]:
```
MDS_custom_distance(X2.copy(), np.array(y2.copy()), 2, "chebyshev",␣
↪'chebyshev', False, False)
```

```
chebyshev
Classification score by using KNN and MDS
0.7938969484742371
```

[35]:
```
MDS_custom_distance(X2.copy(), np.array(y2.copy()), 2, "manhattan",␣
↪'manhattan', False, False)
```

```
manhattan
Classification score by using KNN and MDS
0.8114057028514257
```

[36]:
```
MDS_custom_distance(X2.copy(), np.array(y2.copy()), 2, "canberra", 'canberra',␣
↪False, False)
```

```
canberra
Classification score by using KNN and MDS
0.8419209604802401
```

11 PcoA - Similarité

11.0.1 Notion de similarité :

MNIST L2 (Euclidien) ou cosine

- Idem à K-Médoïdes (pour cosine).
- Le jeu de données peut avoir plus d'exemplaires avec des groupements homogène (i.e. si on a des gens qui écrivent bien) tandis qu'il peut y avoir d'autres jeux de données avec des caractères mals écrits (i.e. des 1 et 7 qui se ressemblent et des 1 qui sont des 7 et vise versa).

ADULT Canberra:

$$\sum(|x-y|/(|x|+|y|))$$

- Puisque nous avons traité les données adultes assez naïvement, la distance Canberra nous permet de représenter la distance comme une somme des ratios au lieu d'utiliser les distances classiques comme l1 et l2 qui sont représentées par les gros chiffres.
- Les données peuvent être reliées dans des mêmes groupes lorsque celles-ci ne sont pas nécessairement proches ou distantes entres-elles. (i.e. des données opposées avec la même classe ou le contraire avec des données proches et classées différemment.)
- Les données avec leurs caractéristiques catégorielles en grande partie et la classification binaire peut être vue comme un choix de sorte (catégories, income plus grand/petit) plutôt que de degré d'exactitude/similarité (comme MNIST).

11.0.2 Clarté des explications:

Se référer au(x) graphique(s) pour des détails aux explications et au fichier README fourni dans le code pour reproduire fidèlement tous les résultats.

11.0.3 Motivations :

MNIST

- Puisque le cosinus est excellent dans les méthodes de partitions précédentes lors d'analyse du jeu de données MNIST, nous voulons l'essayer aussi comme similarité dans MDS. Même si le cosinus est moins performant avec KNN par rapport à la distance euclidienne, il reste un choix si on choisit l'autre classificateur pour faire la prédiction.
- L2 est un bon choix puisque celui-ci agit sur l'espace vectoriel et les distances de manière plus forte que cosine, qui lui prends en compte la similarité.
- Nous tenons à choisir cosine ET L2 Euclidien puisque le fait que nous données sont bien groupés ou qu'une similarité possible par rapport à des ressemblances (i.e. des 1 et 7 qui sont presque pareils) dépends du jeu de donnée.

ADULT

- Canberra est moins influencé par les grandes valeurs que Manhattan. Avec le fait que nous avons plus de "features" catégoriels que de grands nombres, il est pertinent de l'utiliser. (http://www.code10.info/index.php?option=com_content&view=article&id=49:article_canberra-distance&catid=38:cat_coding_algorithms_data-similarity&Itemid=57)
- Nous voulons pouvoir identifier la sorte des données (selon notre classification binaire) et appliquer cette même logique sur les "features" puisque ceux-ci sont catégoriels. Sinon, nous allons voir des distances entre les catégories et cela est totalement à éviter

11.0.4 Analyse des résultats :

MNIST Au niveau du score du classificateur KNN, la distance euclidienne > cosine > les autres tentatives (manhattan, cityblock, canberra, etc.) Notre jeu de données doit avoir des groupements biens homogènes.

ADULT Au niveau du score du classificateur KNN, canberra > manhattan > euclidean > chebyshev. Avec les réductions de dimensions, la notion de distance prend le dessus et on peut le voir avec les pourcentages que chebyshev est maintenant classé en dernier.

12 Isomap

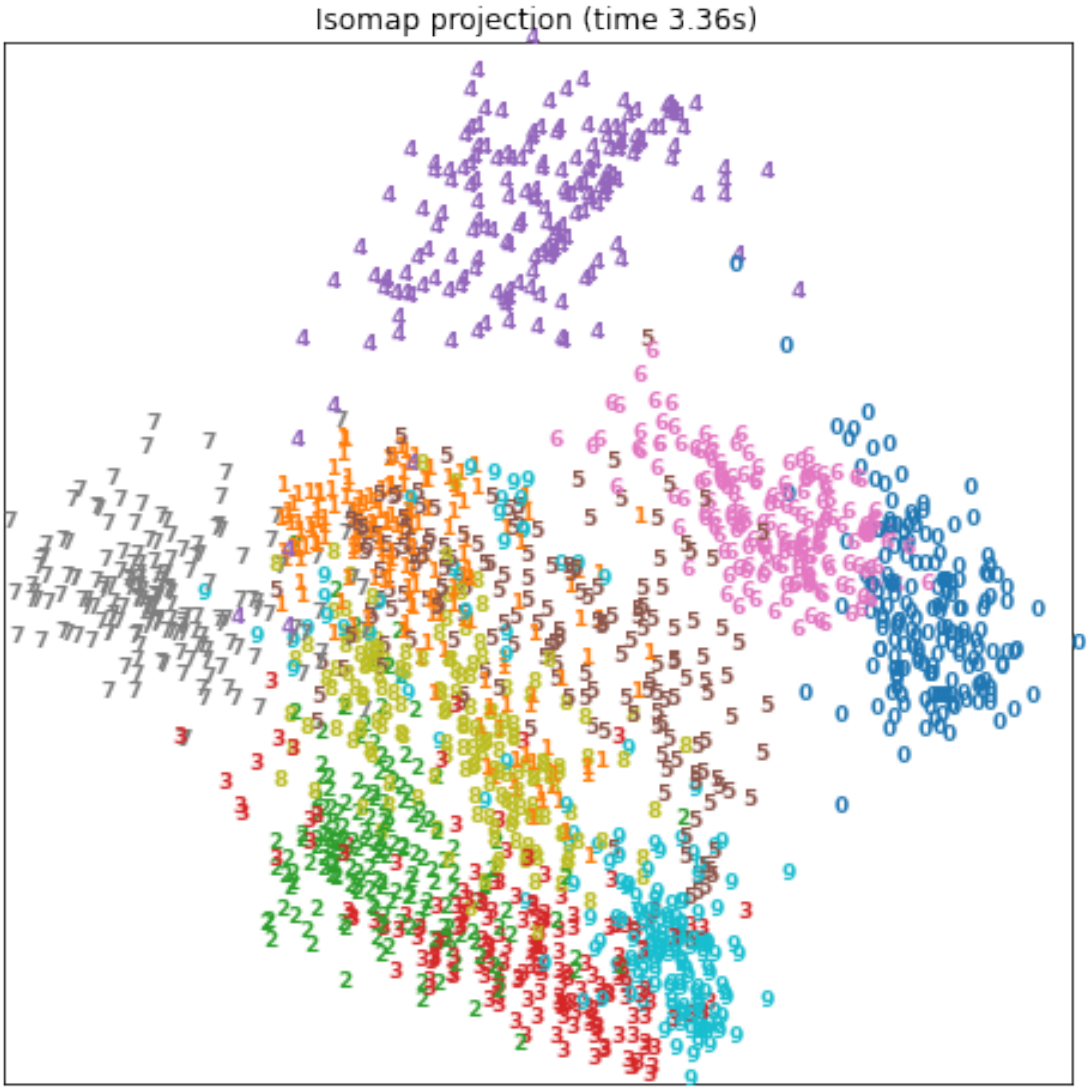
[37]:
```
def isomap(X, y, n, custom_metric, plot=False):
  print("Computing Isomap projection")
  t0 = time()
  X_isomap = manifold.Isomap(n_neighbors=30, n_components=2, metric =␣
 ↪custom_metric).fit_transform(X)
  print("Done.")
```

```
  model_isomap = KNeighborsClassifier()
  model_isomap.fit(X_isomap, y)
  if plot == True:
    plot_embedding(X_isomap, y, n, "Isomap projection (time %.2fs)" % (time() -␣
↪t0))
  print(model_isomap.score(X_isomap, y))
```

12.1 MNIST

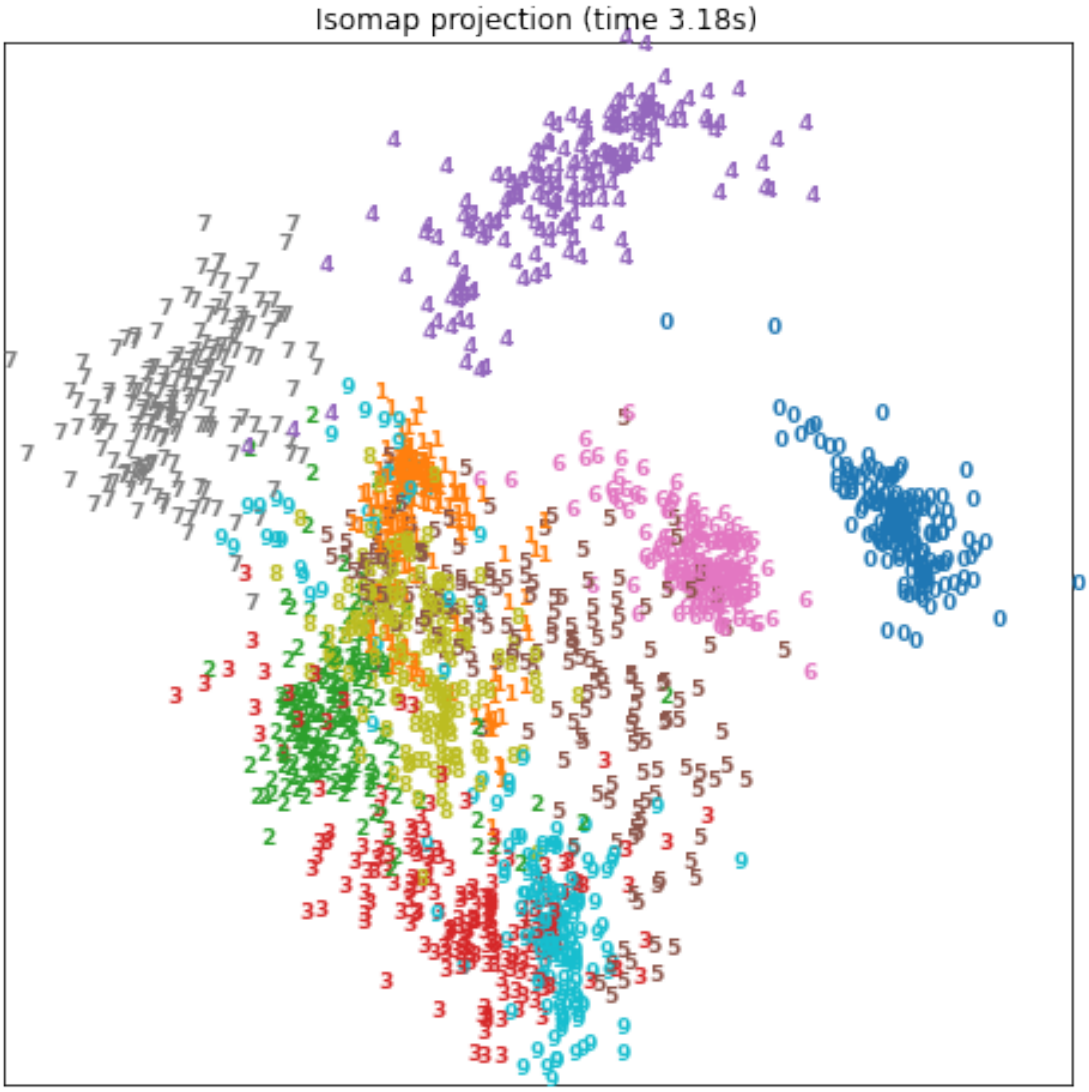
[38]:
```
isomap(X1_org, y1_org, 10, 'minkowski', True) # = euclidean
```

```
Computing Isomap projection
Done.
0.8096828046744574
```



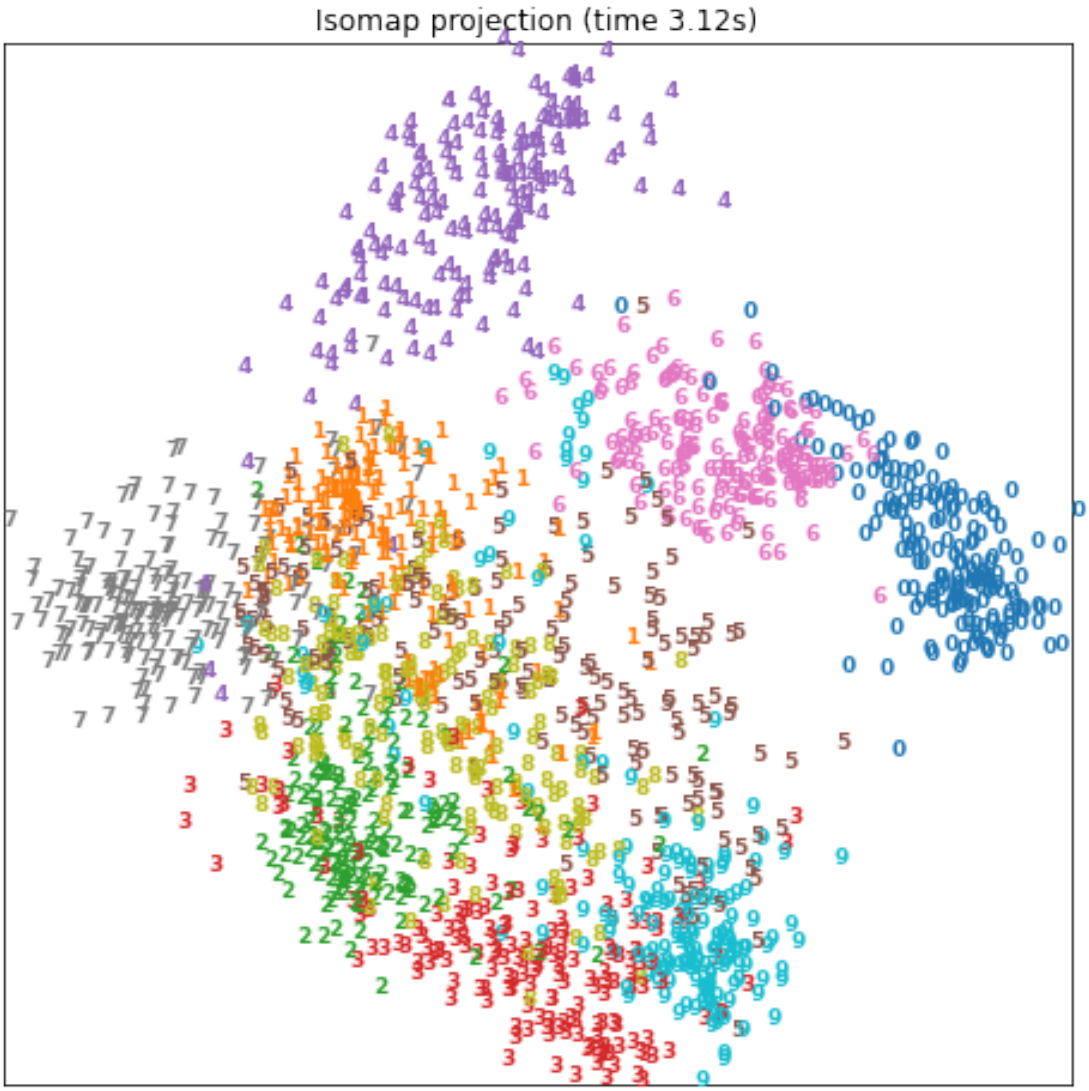
[39]:
```
isomap(X1_org, y1_org, 10, 'cosine', True)
```

```
Computing Isomap projection
Done.
0.8480801335559266
```



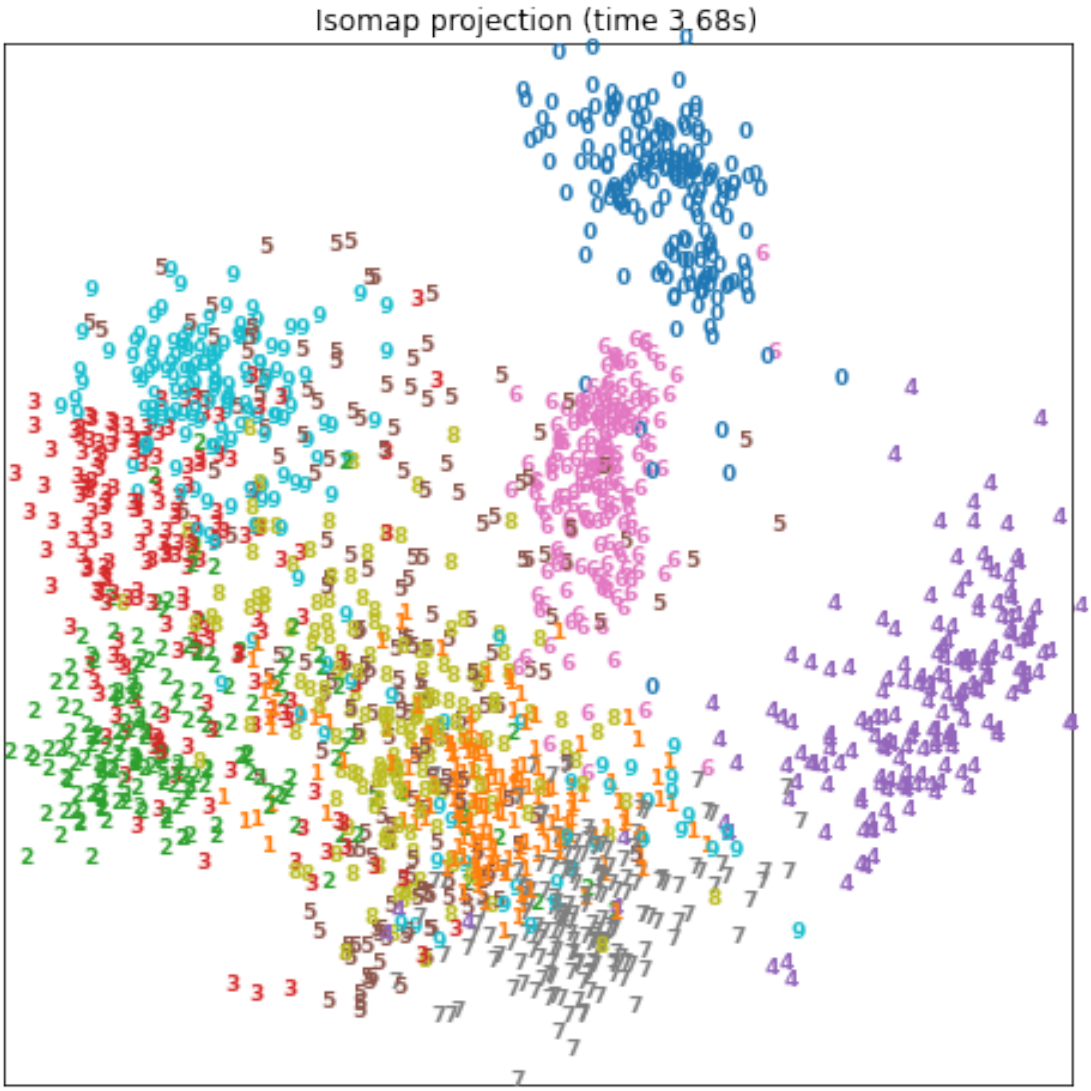
[40]:
```
isomap(X1_org, y1_org, 10, 'manhattan', True)
```

```
Computing Isomap projection
Done.
0.8046744574290484
```



[41]:
```
isomap(X1_org, y1_org, 10, 'canberra', True)
```

```
Computing Isomap projection
Done.
0.7813021702838063
```



12.2 ADULT

[42]:
```
isomap(X2, np.array(y2), 2, 'minkowski', False)
```

```
Computing Isomap projection
Done.
0.8099049524762382
```

[43]:
```
isomap(X2, np.array(y2), 2, 'cosine', False)
```

```
Computing Isomap projection
Done.
0.8269134567283641
```

[44]:
```
isomap(X2, np.array(y2), 2, 'manhattan', False)
```

```
Computing Isomap projection
Done.
0.8094047023511756
```

[45]:
```
isomap(X2, np.array(y2), 2, 'canberra', False)
```

```
Computing Isomap projection
Done.
0.8224112056028015
```

[46]:
```
isomap(X2, np.array(y2), 2, 'chebyshev', False)
```

```
Computing Isomap projection
Done.
0.7978989494747374
```

13 Isomap - Similarité

13.0.1 Notion de similarité :

MNIST Cosine

- Idem à K-Médoïdes, KNN* et PcoA (MDS) puisque les étapes ont des aspects de chaque algorithme mentionné.

ADULT Cosine

- Les données ont maintenant une vue différente par rapport à Isomap où la distance est plus importante puisque l'algorithme à des aspects de KNN*. Cela veut dire que les "features" catégoriels n'ont pas le même poids que d'habitude (i.e. avec les autres algorithmes du devoir).
- Idem à PcoA (MDS), puisque MDS est appliqué en dernier dans la suite d'étapes de l'algorithme Isomap.

13.0.2 Clarté des explications:

Se référer au(x) graphique(s) pour des détails aux explications et au fichier README fourni dans le code pour reproduire fidèlement tous les résultats.

13.0.3 Motivations :

MNIST

- Cosine peut avoir un angle très bas (respectivement une plus grande similarité) par rapport à deux données qui sont loins l'une de l'autre. Ceci motive son utilisation et sa performance par rapport aux autres similarités.
- Contrairement à PcoA (MDS), l'algorithme à des aspects de KNN*. Donc, la distance est une notion d'importance.

ADULT

- Comme MNIST, puisque Isomap à des aspects de KNN*, cosine est pertinent si nous n'avons plus la possibilité d'utiliser Canberra (i.e, trop forte notion sur la distance et similarité).
- Le déclassement minime de Canberra est à cause de l'aspect KNN* de l'algorithme. Cela motive à passer par Cosine pour exploiter le résultat.

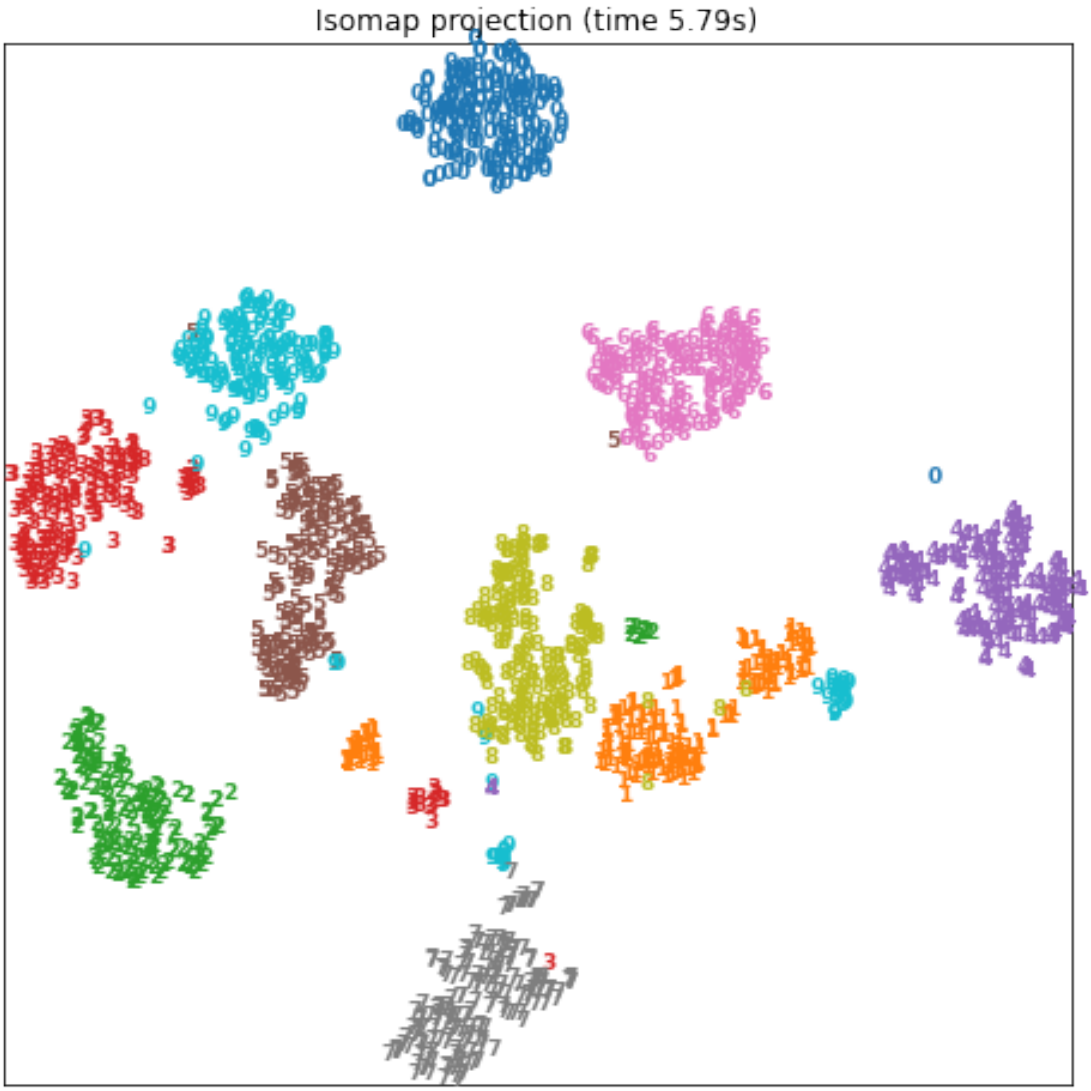
*Une plus grande discussion se retrouve dans la partie KNN respectivement pour alléger le contenu du rapport.

14 Bonus : T-SNE

[47]:
```
def tsne(X, y, n, plot=False):
  print("Computing t-SNE embedding")
  tsne = manifold.TSNE(n_components=2, init='pca', random_state=0)
  t0 = time()
  print("Done.")
  X_tsne = tsne.fit_transform(X)
  model_tsne = KNeighborsClassifier()
  model_tsne.fit(X_tsne, y)
  if plot == True:
    plot_embedding(X_tsne, y, n, "Isomap projection (time %.2fs)" % (time() -␣
 ↪t0))
  print("Classification score by using KNN and T-SNE")
  print(model_tsne.score(X_tsne, y))
```

[48]:
```
tsne(X1_org, y1_org, 10, True)
```

```
Computing t-SNE embedding
Done.
Classification score by using KNN and T-SNE
0.9905397885364496
```



[49]:
```
tsne(X2, np.array(y2), 2, False)
```

```
Computing t-SNE embedding
Done.
Classification score by using KNN and T-SNE
0.8124062031015508
```

15 T-SNE - Similarité

Nous ne sommes pas satisfait de PcoA (MDS) et Isomap pour leur résultats en général. De ce fait, nous nous sommes permis d'utiliser T-SNE pour générer une réduction plus représentative, qui nous donne les résultats très satisfaisant pour MNIST (> 0.99 comme socre de KNN) et 0.81 pour Adult.

16 KNN

[50]:
```
# KNN using different metrics on MNIST datasets
# https://medium.com/analytics-vidhya/
 ↪a-beginners-guide-to-knn-and-mnist-handwritten-digits-recognition-using-knn-from-scratch-df
# https://docs.scipy.org/doc/scipy/reference/generated/scipy.spatial.distance.
 ↪pdist.html


'''
Metrics
['braycurtis', 'canberra', 'chebyshev', 'cityblock', 'correlation', 'cosine',
 'dice', 'euclidean', 'hamming', 'haversine', 'jaccard', 'kulsinski', 'l1',␣
 ↪'l2',
 'mahalanobis', 'manhattan', 'matching', 'minkowski',␣
 ↪'nan_euclidean','precomputed',
 'rogerstanimoto', 'russellrao', 'seuclidean', 'sokalmichener', 'sokalsneath',
 'sqeuclidean', 'wminkowski', 'yule']
'''

X1_train, X1_test, y1_train, y1_test = train_test_split(X1, y1, test_size=0.25,␣
 ↪random_state=123)
X1_train.shape, X1_test.shape, y1_train.shape, y1_test.shape
```

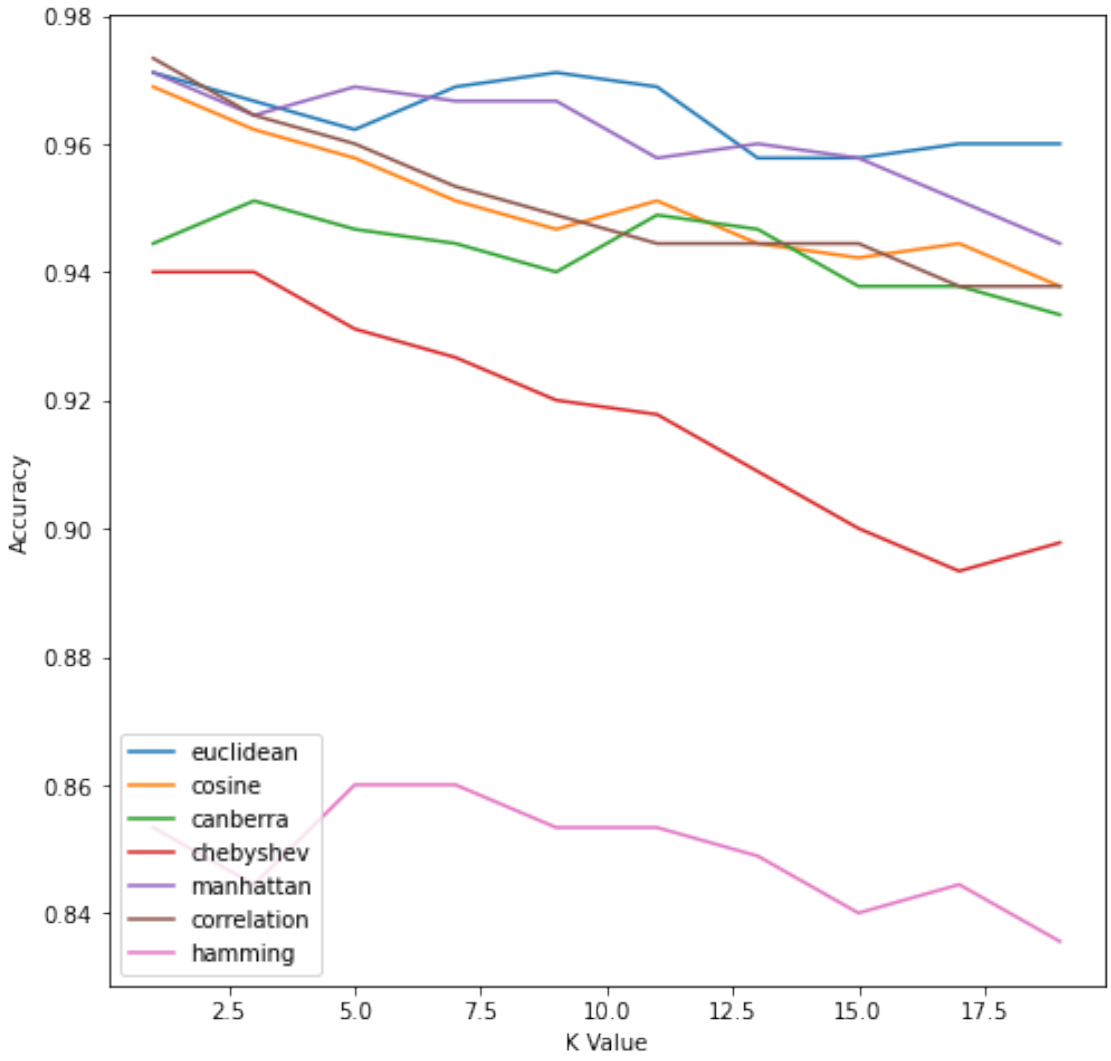
[50]:
```
((1347, 64), (450, 64), (1347,), (450,))
```

[51]:
```
def knn(X_train, y_train, X_test, y_test):
    np.random.seed(42)
    kVals = np.arange(1,20,2)
    metrics = ['euclidean', 'cosine', 'canberra', 'chebyshev', 'manhattan',␣
 ↪'correlation', 'hamming']
    plt.figure(figsize = (8, 8))

    for each_metric in metrics:
      print(each_metric)
      accuracies = []
      for k in kVals:
        model = KNeighborsClassifier(n_neighbors=k, p=2, metric=each_metric)
        model.fit(X_train, y_train)
        pred = model.predict(X_test)
        acc = accuracy_score(y_test, pred)
        accuracies.append(acc)
        print("K = "+str(k)+"; Accuracy: "+str(acc))
      plt.plot(kVals, accuracies, label=each_metric)
      plt.xlabel("K Value")
      plt.ylabel("Accuracy")
      plt.legend(loc=3)
```

[52]:
```
knn(X1_train, y1_train, X1_test, y1_test)
```

```
euclidean
K = 1; Accuracy: 0.9711111111111111
K = 3; Accuracy: 0.9666666666666667
K = 5; Accuracy: 0.9622222222222222
K = 7; Accuracy: 0.9688888888888889
K = 9; Accuracy: 0.9711111111111111
K = 11; Accuracy: 0.9688888888888889
K = 13; Accuracy: 0.9577777777777777
K = 15; Accuracy: 0.9577777777777777
K = 17; Accuracy: 0.96
K = 19; Accuracy: 0.96
cosine
K = 1; Accuracy: 0.9688888888888889
K = 3; Accuracy: 0.9622222222222222
K = 5; Accuracy: 0.9577777777777777
K = 7; Accuracy: 0.9511111111111111
K = 9; Accuracy: 0.9466666666666667
K = 11; Accuracy: 0.9511111111111111
K = 13; Accuracy: 0.9444444444444444
K = 15; Accuracy: 0.9422222222222222
K = 17; Accuracy: 0.9444444444444444
K = 19; Accuracy: 0.9377777777777778
canberra
K = 1; Accuracy: 0.9444444444444444
K = 3; Accuracy: 0.9511111111111111
K = 5; Accuracy: 0.9466666666666667
K = 7; Accuracy: 0.9444444444444444
K = 9; Accuracy: 0.94
K = 11; Accuracy: 0.9488888888888889
K = 13; Accuracy: 0.9466666666666667
K = 15; Accuracy: 0.9377777777777778
K = 17; Accuracy: 0.9377777777777778
K = 19; Accuracy: 0.9333333333333333
chebyshev
K = 1; Accuracy: 0.94
K = 3; Accuracy: 0.94
K = 5; Accuracy: 0.9311111111111111
K = 7; Accuracy: 0.9266666666666666
K = 9; Accuracy: 0.92
K = 11; Accuracy: 0.9177777777777778
K = 13; Accuracy: 0.9088888888888889
K = 15; Accuracy: 0.9
K = 17; Accuracy: 0.8933333333333333
K = 19; Accuracy: 0.8977777777777778
manhattan
K = 1; Accuracy: 0.9711111111111111
```

```
K = 3; Accuracy: 0.9644444444444444
K = 5; Accuracy: 0.9688888888888889
K = 7; Accuracy: 0.9666666666666667
K = 9; Accuracy: 0.9666666666666667
K = 11; Accuracy: 0.9577777777777777
K = 13; Accuracy: 0.96
K = 15; Accuracy: 0.9577777777777777
K = 17; Accuracy: 0.9511111111111111
K = 19; Accuracy: 0.9444444444444444
correlation
K = 1; Accuracy: 0.9733333333333334
K = 3; Accuracy: 0.9644444444444444
K = 5; Accuracy: 0.96
K = 7; Accuracy: 0.9533333333333334
K = 9; Accuracy: 0.9488888888888889
K = 11; Accuracy: 0.9444444444444444
K = 13; Accuracy: 0.9444444444444444
K = 15; Accuracy: 0.9444444444444444
K = 17; Accuracy: 0.9377777777777778
K = 19; Accuracy: 0.9377777777777778
hamming
K = 1; Accuracy: 0.8533333333333334
K = 3; Accuracy: 0.8444444444444444
K = 5; Accuracy: 0.86
K = 7; Accuracy: 0.86
K = 9; Accuracy: 0.8533333333333334
K = 11; Accuracy: 0.8533333333333334
K = 13; Accuracy: 0.8488888888888889
K = 15; Accuracy: 0.84
K = 17; Accuracy: 0.8444444444444444
K = 19; Accuracy: 0.8355555555555556
```



[53]:
```
X2_train, X2_test, y2_train, y2_test = train_test_split(X2, y2, test_size=0.25,␣
↪random_state=123)
X2_train.shape, X2_test.shape, y2_train.shape, y2_test.shape
```

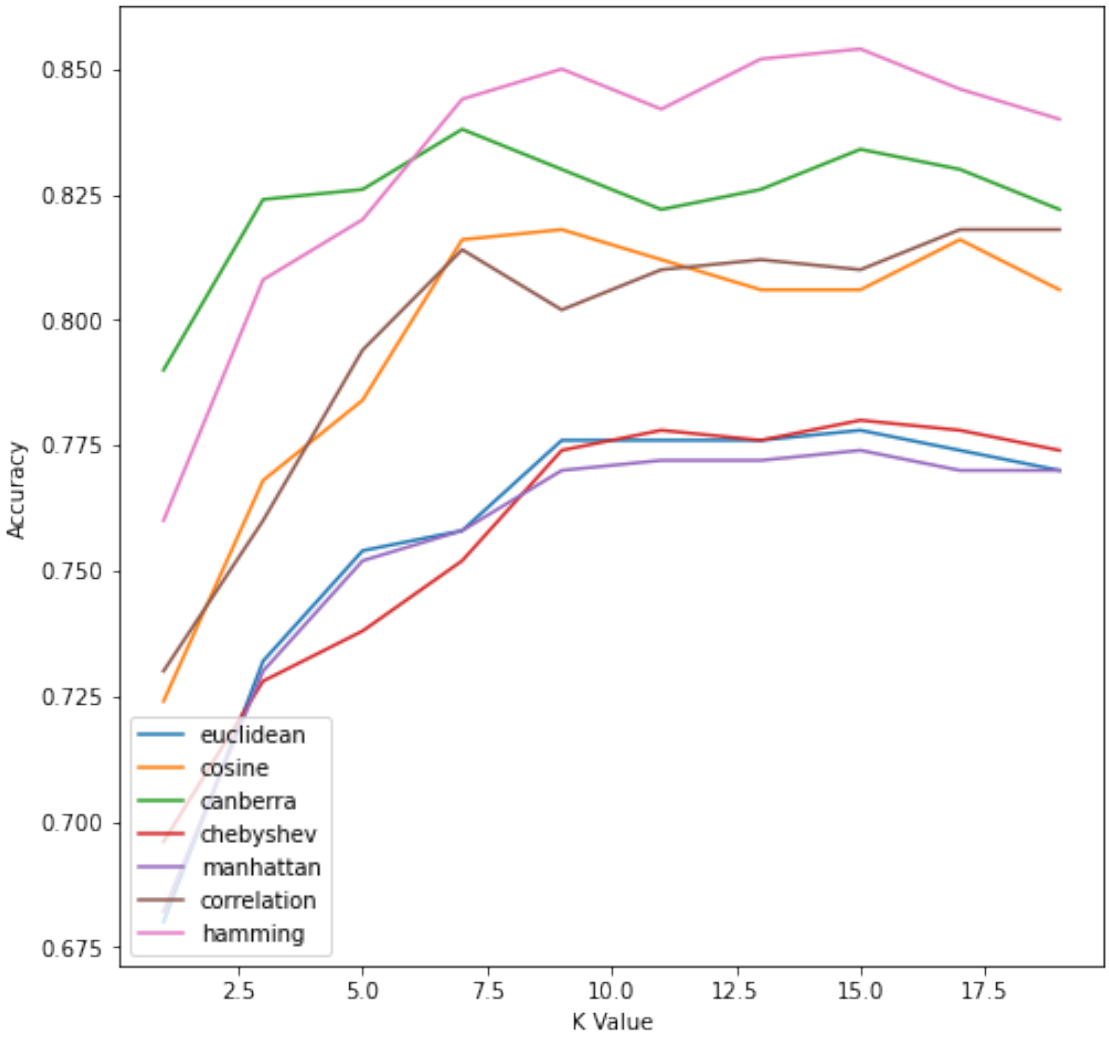
[53]:
```
((1499, 14), (500, 14), (1499,), (500,))
```

[54]:
```
knn(X2_train, y2_train, X2_test, y2_test)
```

```
euclidean
K = 1; Accuracy: 0.68
K = 3; Accuracy: 0.732
K = 5; Accuracy: 0.754
K = 7; Accuracy: 0.758
K = 9; Accuracy: 0.776
K = 11; Accuracy: 0.776
```

```
K = 13; Accuracy: 0.776
K = 15; Accuracy: 0.778
K = 17; Accuracy: 0.774
K = 19; Accuracy: 0.77
cosine
K = 1; Accuracy: 0.724
K = 3; Accuracy: 0.768
K = 5; Accuracy: 0.784
K = 7; Accuracy: 0.816
K = 9; Accuracy: 0.818
K = 11; Accuracy: 0.812
K = 13; Accuracy: 0.806
K = 15; Accuracy: 0.806
K = 17; Accuracy: 0.816
K = 19; Accuracy: 0.806
canberra
K = 1; Accuracy: 0.79
K = 3; Accuracy: 0.824
K = 5; Accuracy: 0.826
K = 7; Accuracy: 0.838
K = 9; Accuracy: 0.83
K = 11; Accuracy: 0.822
K = 13; Accuracy: 0.826
K = 15; Accuracy: 0.834
K = 17; Accuracy: 0.83
K = 19; Accuracy: 0.822
chebyshev
K = 1; Accuracy: 0.696
K = 3; Accuracy: 0.728
K = 5; Accuracy: 0.738
K = 7; Accuracy: 0.752
K = 9; Accuracy: 0.774
K = 11; Accuracy: 0.778
K = 13; Accuracy: 0.776
K = 15; Accuracy: 0.78
K = 17; Accuracy: 0.778
K = 19; Accuracy: 0.774
manhattan
K = 1; Accuracy: 0.682
K = 3; Accuracy: 0.73
K = 5; Accuracy: 0.752
K = 7; Accuracy: 0.758
K = 9; Accuracy: 0.77
K = 11; Accuracy: 0.772
K = 13; Accuracy: 0.772
K = 15; Accuracy: 0.774
K = 17; Accuracy: 0.77
K = 19; Accuracy: 0.77
```

```
correlation
K = 1; Accuracy: 0.73
K = 3; Accuracy: 0.76
K = 5; Accuracy: 0.794
K = 7; Accuracy: 0.814
K = 9; Accuracy: 0.802
K = 11; Accuracy: 0.81
K = 13; Accuracy: 0.812
K = 15; Accuracy: 0.81
K = 17; Accuracy: 0.818
K = 19; Accuracy: 0.818
hamming
K = 1; Accuracy: 0.76
K = 3; Accuracy: 0.808
K = 5; Accuracy: 0.82
K = 7; Accuracy: 0.844
K = 9; Accuracy: 0.85
K = 11; Accuracy: 0.842
K = 13; Accuracy: 0.852
K = 15; Accuracy: 0.854
K = 17; Accuracy: 0.846
K = 19; Accuracy: 0.84
```



17 KNN - Similarité

17.0.1 Notion de similarité :

MNIST Corrélation:
Définition

$$1 - \frac{(u - \overline{u}) \cdot (v - \overline{v})}{||(u - \overline{u})||_2 ||(v - \overline{v})||_2}$$

où

$$\overline{v}$$

est la moyenne du vecteur v.

- Le jeu de données contient plusieurs classes (10), il est donc plausible de supposer que chaque donnée à un niveau de similarité à prendre en considération pour ne pas se tromper dans nos

prédictions. De ce fait, un k très petit est nécessaire et justifié avec notre notion de similarité en lien avec cela.

- KNN est basé sur le fait que la donnée est le centre et qu'elle est classifiée en fonction de ses pairs proches. Cosine n'a donc plus sa place et il faut une notion qui prend en compte la densité des données proches de celle-ci.
- MNIST est un jeu de données où celles-ci sont généralement fortement corrélées entre-elles lorsqu'elles sont très proches l'une de l'autre. Ceci s'explique du fait que puisqu'une donnée est basée sur la représentation graphique d'un caractère. Il n'y a pas une infinité de structure de représentation possible (bien qu'il pourrait y avoir des variations mineurs quasi-infini) pour distinguer l'écriture du caractère (i.e. deux manières connues d'écrire 4 calligraphiquement, les variantes du style d'écriture, etc.).

ADULT Hamming : Définition La distance de Hamming consiste à compter quel ensemble de chiffres ou de lieux correspondants sont différents et quels sont les mêmes.

KNN est basé sur des distances entre données, et nous savons que les "features" sont catégoriels à priori et qu'il y en a beaucoup.

Les données ont peu de features continues comme par rapport au capital étant un grand nombre entre zéro et 1k+. Il est donc pertinent d'utiliser une similarité en lien avec les "features" catégoriels.

Les "features" catégoriels sont représentés avec des petits nombres (i.e. entre 0 et 3+ et ne dépassant pas 10 à priori). Il est pertinent d'utiliser une similarité qui prend cela en compte.

17.0.2 Clarté des explications:

Se référer au(x) graphique(s) pour des détails aux explications et au fichier README fourni dans le code pour reproduire fidèlement tous les résultats.

17.0.3 Motivations :

MNIST La nature des données est par rapport à la représentation de la calligraphie de chaque caractère. Il est pertinent de se baser sur la similarité par rapport à la typologie des caractères comme distance.

Puisque nous pouvons conclure que le cosine est le plus performant dans presque toutes les méthodes non-supervisées, l'utilisation de la similarité qui ressemble au cosine peut être aussi justifiée.

Le fait que la précision n'est pas parfaite même avec k=1 est lorsque par exemple nous voulons distinguer deux caractères de classe différente (i.e 3 et 5), mais que la calligraphie de ses caractères est très ressemblante. De ce fait, cela motive que la corrélation soit la similarité de choix puisque nous devons maximiser nos efforts sur cette propriété.

ADULT Utiliser le hamming est très pertinent puisque la distance sur des très petites valeurs agit sur les features catégoriels respectivement.

La performance et la précision est bonne en général sauf dans le cas ou k=1, mais nous pouvons omettre cela puisque k=1 est généralement mieux pour les cas où on a des données multi-classes comme MNIST tandis que nous avons besoin d'un k plus grands pour bien classifier un cas binaire.

Il faut prendre en considération que KNN va utiliser les “features” comme des distances, donc Hamming permet de faire une comparaison à titre de valeur comme des positions/catégories.

17.0.4 Analyse des résultats :

MNIST En observant les résultats et le graphique, nous pouvons conclure que la corrélation comme similarité et k = 1 (légèrement plus forte que le cosine) nous donne la meilleure précision.

ADULT En observant les résultats et le graphique, nous pouvons conclure que le hamming comme similarité et k = 15 nous donne la meilleur précision.