P1RomeroChantadaMAAAII_1

March 7, 2025

1 Practica 1

1.1 ## Pablo Chantada Saborido y José Romero Conde

1.1.1 Indice

[23]:
```
from utils.mnist_reader import load_mnist
import matplotlib.pyplot as plt
from matplotlib.gridspec import GridSpec
from  sklearn.metrics import adjusted_rand_score, normalized_mutual_info_score,␣
 ↪silhouette_score, fowlkes_mallows_score, confusion_matrix
from sklearn.cluster import KMeans, DBSCAN, AgglomerativeClustering, MeanShift,␣
 ↪OPTICS, estimate_bandwidth, AffinityPropagation, SpectralClustering
import hdbscan
from sklearn.decomposition import PCA
import sklearn.manifold
from  sklearn.model_selection import GridSearchCV
import numpy as np
from time import time
import umap
import torch
```

```
from torch import nn, tensor, optim
from torch.utils.data import DataLoader
import pandas as pd
import seaborn as sns
```

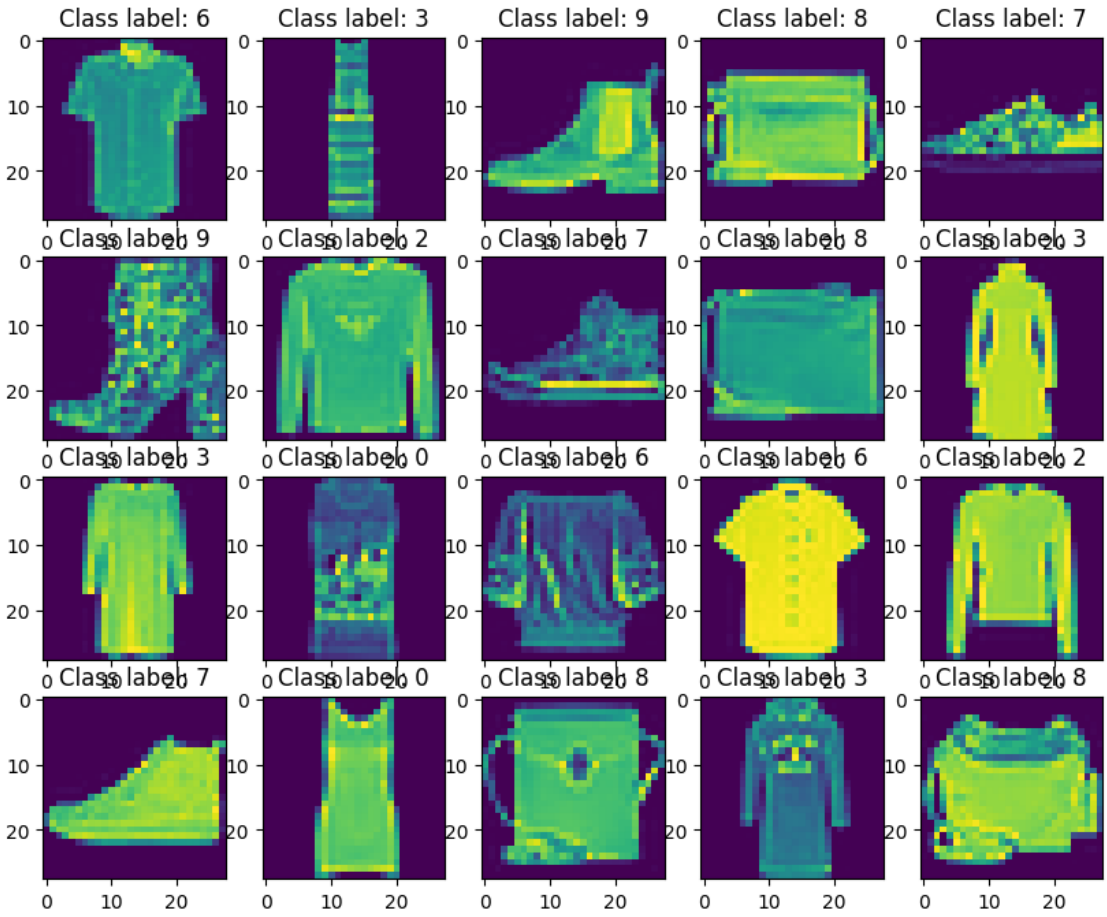
1.2 1. Analisis exploratorio

[2]:
```
X_train, y_train = load_mnist('data/fashion', kind='train')
X_test, y_test = load_mnist('data/fashion', kind='t10k')
X_train, X_dev = X_train[:50000], X_train[50000:]
y_train, y_dev = y_train[:50000], y_train[50000:]

print(f'X_train: {X_train.shape}, y_train: {y_train.shape}')
print(f'X_dev: {X_dev.shape}, y_dev: {y_dev.shape}')
print(f'X_test: {X_test.shape}, y_test: {y_test.shape}')
```

```
X_train: (50000, 784), y_train: (50000,)
X_dev: (10000, 784), y_dev: (10000,)
X_test: (10000, 784), y_test: (10000,)
```

1.2.1 1.1 Visualización

[3]:
```
indexes = np.random.choice(len(X_train),20)
plt.figure(figsize=(10, 8))
for i, index in enumerate(indexes):
    plt.subplot(4,5,i+1)
    plt.title(f'Class label: {y_train[index]}')
    plt.imshow(np.resize(X_train[index],(28,28)))

plt.show()
```



Vemos el dataset, consta de ropa y las clases oficialmente son: | Clase | Tipo de prenda | | — | — | | 0 | Camiseta | | 1 | Pantalón | | 2 | Jersei | | 3 | Vestido | | 4 | Abrigo | | 5 | Sandalia | | 6 | Camisa | | 7 | Tenis | | 8 | Bolso | | 9 | Bota |

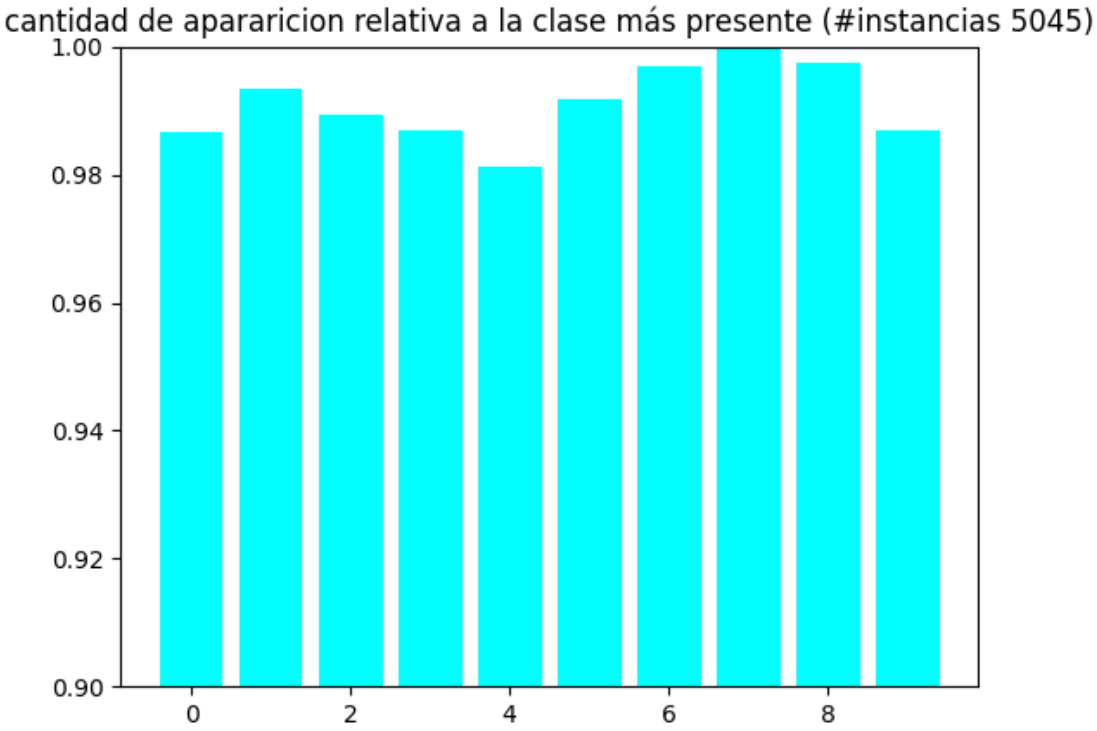
Como es evidente, no todas las clases son igual de parecidas entre si ni distintas con respecto a las otras. Los Jerseis serán bastante parecidos entre sí mientras que las sandalias pueden ser más distintas (formas realmente diferentes). Esto tiene que ver con el concepto de *Eps* (una medida de distancia entre un punto de un *cluster* y sus vecinos; a cuanto más grande *Eps* más alejados estarán los considereados vecinos. https://www.dbs.ifi.lmu.de/Publikationen/Papers/KDD-96.final.frame.pdf), que esperaríamos que varíe de cluster a cluster (lo cual dificulta el problema).

También esperariamos que las camisas sean más parecidas a las camisetas que a los tenis, e incluso se podría confundir camisas con camisetas. Esto es razonable que ocurra pero puede acarrear la consecuencia de confundir clases en solo una. Solventar este problema es especialmente complejo porque necesitamos que el *clusterizador* pueda distinguir con las n carácteristicas $(x_1, ..., x_n)$ lo que es una camiseta o una camisa. No es evidente, en cambio, como un *clusterizador* puede hacer esto si las caracteristicas son en valor de intensidad de un pixel. Por tanto, para solucionar el problema de camisas vs camisetas necesitariamos (o nos ayudaría enormemente) una buena representación, de alto nivel de los datos. Idealmente aprendidas con métodos profundos (ver sección 1.5).

Aprovechamos este momento para discutir inicialmente el número de clusters apropiado. Si bien parece razonable adoptar las clases originales ($k = 10$) como buenas, la distintcion entre las prendas de ropa no es tan arbitraria; podría ser muy especifica (20 clústers por ejemplo) y separar manga corta de larga, tipo de tela, etc. También podríamos irnos al otro extremo, y usar unicamente dos clusters para prendas superiores e inferiores. Por tanto no tenemos una granuralidad fija de cuanta división queremos obtener, por ello a lo largo de la práctica usaremos métricas que supondrán que $k = 10$ es lo ideal al usar y_{train}. Esto, no obstante, podría no ser lo ideal para alguna aplicición. Se volverá a tratar este punto en la sección 1.4 .

1.2.2 1.2 Balanceo de datos

[4]:
```
counts, _ = np.histogram(y_train)
plt.bar(range(len(counts)), counts/max(counts) ,color='cyan')
plt.ylim((0.9,1))
plt.title(f'cantidad de apararicion relativa a la clase más presente␣
 ↪(#instancias {max(counts)})');
```



En el anterior gráfico se muestra la presencia relativa (número de instancias) para cada etiqueta con respecto a la clase con más instancias. Observamos que las clases están distribuidas de forma mas o menos equitativa; por lo que no sería necesario aplicar tecnicas como *over/undersampling*. Inicialmente esto facilita el análisis, sin embargo no importa que hayan 5000 camisetas si son todas muy parecidas, ya que no nos permitirian obtener una buena generalización.

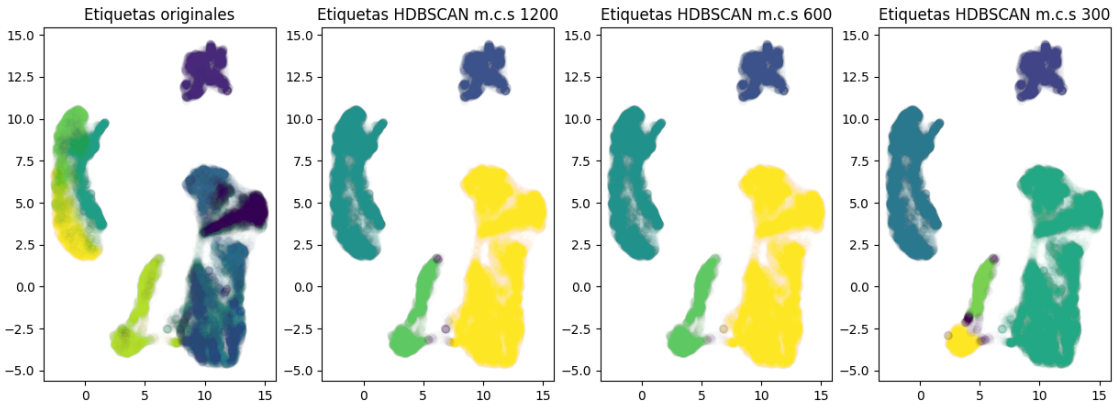
1.3 Atípicos

[11]:
```
reducer = umap.UMAP()
```

[12]:
```
X_train_reduced = reducer.fit_transform(X_train)
hdbscan1200 = HDBSCAN(min_cluster_size = 1200)
hdbscan600 = HDBSCAN(min_cluster_size = 600)
hdbscan300 = HDBSCAN(min_cluster_size = 300)
labelsHDBSCAN1200 = hdbscan1200.fit_predict(X_train_reduced)
labelsHDBSCAN600 = hdbscan600.fit_predict(X_train_reduced)
labelsHDBSCAN300 = hdbscan300.fit_predict(X_train_reduced)
```

```
/Users/pepe/carrera/3/2/ma2/ma2/lib/python3.13/site-
packages/sklearn/utils/deprecation.py:151: FutureWarning: 'force_all_finite' was
renamed to 'ensure_all_finite' in 1.6 and will be removed in 1.8.
  warnings.warn(
```

[7]:
```
plt.figure(figsize=(15, 5))
plt.subplot(1,4,1)
plt.title('Etiquetas originales')
plt.scatter(X_train_reduced[:,0],X_train_reduced[:,1],alpha=1e-2,c=y_train)
plt.subplot(1,4,2)
plt.title('Etiquetas HDBSCAN m.c.s 1200')
plt.scatter(X_train_reduced[:,0],X_train_reduced[:
 ↪,1],alpha=1e-2,c=labelsHDBSCAN1200)
plt.subplot(1,4,3)
plt.title('Etiquetas HDBSCAN m.c.s 600')
plt.scatter(X_train_reduced[:,0],X_train_reduced[:
 ↪,1],alpha=1e-2,c=labelsHDBSCAN600)
plt.subplot(1,4,4)
plt.title('Etiquetas HDBSCAN m.c.s 300')
plt.scatter(X_train_reduced[:,0],X_train_reduced[:
 ↪,1],alpha=1e-2,c=labelsHDBSCAN300);
```



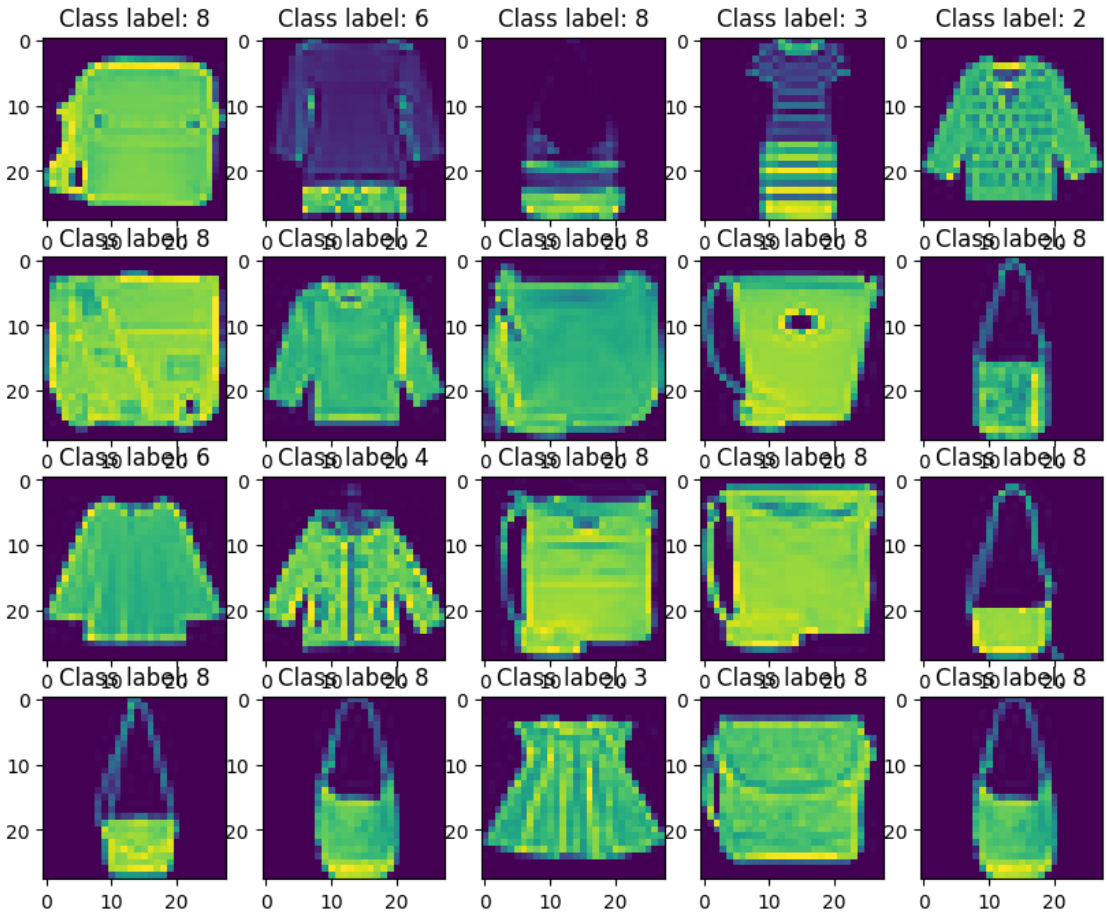
En el gráfico anterior vemos los datos reducidos a dos dimensiones con UMAP

(https://arxiv.org/pdf/1802.03426). Sobre esos datos hemos clusterizado usando HDBSCAN (https://hdbscan.readthedocs.io/en/latest/how_hdbscan_works.html) porque bajo su definición de *cluster* se considera el concepto de *ruído* o *anomalía*. Esto es posible porque para los algoritmos derivados de DBSCAN todos los puntos cumplen alguna de las siguientes condiciones: - Tiene en un a una distancia Eps a por lo menos $MinPoints$ puntos. Llamemoslos puntos centrales. - No tiene a una distancia Eps a por lo menos $MinPoints$ puntos pero alguno de los puntos que están a distancia Eps si que tienen a una distancia Eps a por lo menos $MinPoints$ puntos. No es un punto central pero esta cerca de un punto central, llamemoslos puntos bordes. - Ni son puntos centrales ni tienen a uno a una distancia Eps. Estos serán los ruidos o anomalías.

Naturalmente, lo que para un conjunto de datos concretos es un punto anómalo, dependerá de la elección de hiperparámetros. Probamos el hiperparámetro `min_cluster_size` con valores $100 \times 3i \mid i \in \{1, 2, 4\}$, y observamos que la grafica con `min_cluster_size=1200` tiene ruído predominante en dos regiones: - Encima, en la punta del cluster de abajo a la izquierda - Entre los dos clusters de abajo

Escogemos entonces las etiquetas producidas por esa clusterización y a continuación ploteamos unas cuantas prendas de la clase *ruido* (-1). Esperaríamos ver, por tanto dos tipos distintos de ruído.

[9]:
```
anomalies = np.where(labelsHDBSCAN1200 == -1)[0]
indexes = np.random.choice(anomalies,20)
plt.figure(figsize=(10, 8))
for i, index in enumerate(indexes):
    plt.subplot(4,5,i+1)
    plt.title(f'Class label: {y_train[index]}')
    plt.imshow(np.resize(X_train[index],(28,28)))
plt.show()
```

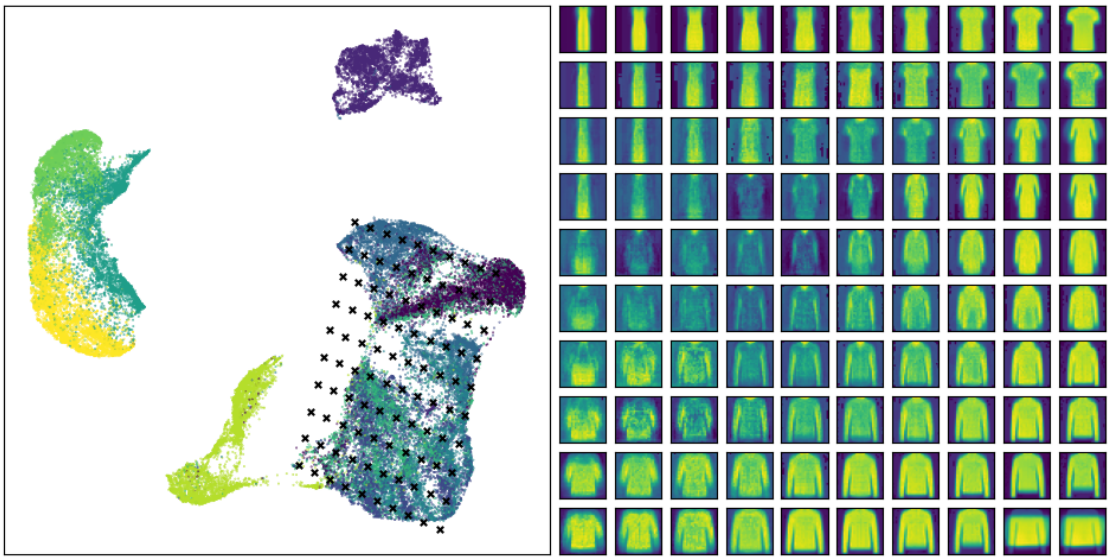


1.4 Discusión de número de clusters Ahora que hemos explorado ligeramente la distribución de los datos en el espacio-de-pixeles-reducido-por-umap, vamos a aprovechar para retomar la discusión del número apropiado de clusters. En concreto vamos a explorar la región de abajo a la derecha, que originalmente (según las etiquetas proporcionadas) son muchas clases pero, por ejemplo los HDBSCANes pensaban que sólo era una clase. No obstante, HDBSCAN se equivoca garrafalmente porque está haciendo la clasificación sobre la representación en $\mathbb{R}^2$, la cual falla en capturar la diferencia real entre elementos.

[ ]:
```
# Definimos ahora una region del espacio R^2 delimitada
# por 'corners'. Serán las esquinas de nuestra parcela.
# Se han definido para que cubran la región de abajo a la
# derecha.

corners = np.array([

    [9, 7],
    [14, 5],
    [7, -2.5],
```

```
    [12, -5]

])
# Interpolamos entre esos puntos y obtenemos los puntos
# 'test_pts', luego calcularmos la función inversa y así
# tendremos las imagenes muestreadas.

test_pts = np.array([
    (corners[0]*(1-x) + corners[1]*x)*(1-y) +
    (corners[2]*(1-x) + corners[3]*x)*y
    for y in np.linspace(0, 1, 10)
    for x in np.linspace(0, 1, 10)
])

inv_transformed_points = reducer.inverse_transform(test_pts)
```

[ ]:
```
# Código referenciado de https://umap-learn.readthedocs.io/en/latest/
 ↪inverse_transform.html

fig = plt.figure(figsize=(12,6))
gs = GridSpec(10, 20, fig)
scatter_ax = fig.add_subplot(gs[:, :10])
digit_axes = np.zeros((10, 10), dtype=object)
for i in range(10):
    for j in range(10):
        digit_axes[i, j] = fig.add_subplot(gs[i, 10 + j])


scatter_ax.scatter(reducer.embedding_[:, 0], reducer.embedding_[:, 1],
                   c=y_train.astype(np.int32),  s=0.1)
scatter_ax.set(xticks=[], yticks=[])

scatter_ax.scatter(test_pts[:, 0], test_pts[:, 1], marker='x', c='k', s=15)

for i in range(10):
    for j in range(10):
        digit_axes[i, j].imshow(inv_transformed_points[i*10 + j].reshape(28,␣
 ↪28))
        digit_axes[i, j].set(xticks=[], yticks=[])
```



En la grafica anterior se samplearon puntos en el espacio reducido y se generaron las correspondientes imágenes en el espacio de píxeles. Esto es posible gracias a que HDBSCAN tiene inversa.

Si bien originalmente podríamos pensar que toda esa región era un mismo cluster (la division a ojo o por HDBSCAN, de puntos en $\mathbb{R}^2$ parecía sugerirlo) ahora vemos que no. Que a lo largo de la región hay distintos tipos de prendas. Fácilemente se pueden nombrar: vestidos de tubo, jerseis, camisetas de manga corta, blusas y sudaderas. Por otro lado también es posible nombrarlos a todos como *"ropa superior"*, y entonces toda la región sí sería un único cluster.

Como curiosidad final, en el plot de la izquierda se ven puntos que caen fuera del cluster y justamente esos mismos puntos son los que con más dificultad se reconstruyen (por eso aparece una lengua emborronada en la mitad izquierda del plot de la derecha).

1.5 Necesidad de transformación La gráfica en la que se veía el dataset en $\mathbb{R}^2$ y sampleados en ese espacio sus reconstrucciones en el espacio de pixeles demuestran que umap no lo hace nada mal. No obstante tenemos que hablar del elefante en la habitación y es que estas imagenes estan meticulosamente recortadas y alineadas. Esto no solo simplifica la tarea de umap, si no que la hace posible en primer lugar. Sabemos que no es razonable tratar con imagenes en el espacio de pixeles puro, y alguna detección de bordes, esquinas, puntos de interés o alguna convolución con algún filtro sabemos que ayudan en la difícil tarea de la visión por computador. Sabiendo esto intentamos hacer un autoencoder para reducir las imagenes a un espacio en el que tenga más sentido tratarlas.

Desgraciadamente, por cuestiones de computo (en especial de memoria) esto no fue posible para nosotros (**mostrar referencia a uno que funcione**). No obtante queremos mostrar nuestro intento. Primero definimos la arquitectura y el bucle de entretenimiento.

[13]:
```
class Autoencoder(nn.Module):
    def __init__(self):
        super(Autoencoder, self).__init__()
```

```
        # Encoder

        self.encoder = nn.Sequential(
            nn.Conv2d(1, 16, kernel_size=3, stride=1, padding=1),
            nn.ReLU(),
            nn.MaxPool2d(kernel_size=2, stride=2),
            nn.Conv2d(16, 8, kernel_size=3, stride=1, padding=1),
            nn.ReLU(),
            nn.MaxPool2d(kernel_size=2, stride=2)
        )
        self.decoder = nn.Sequential(
            nn.ConvTranspose2d(8, 16,
                               kernel_size=3,
                               stride=2,
                               padding=1,
                               output_padding=1),
            nn.ReLU(),
            nn.ConvTranspose2d(16, 1,
                               kernel_size=3,
                               stride=2,
                               padding=1,
                               output_padding=1),
            nn.Sigmoid()
        )

    def forward(self, x):
        encoded = self.encoder(x)
        decoded = self.decoder(encoded)
        return decoded

    def trainModel(self, train_loader):
        criterion = nn.MSELoss()  # Mean Squared Error Loss
        lr = 3e-3
        optimizer = optim.Adam(self.parameters(), lr=lr)
        num_epochs = 2000
        device = torch.device("mps")
        self.to(device)
        losses = []
        for epoch in range(num_epochs):
            self.train()
            running_loss = 0.0
            for batch_idx, data in enumerate(train_loader):
                # Forward pass

                data = data.to(device)
                output = self.forward(data)
                loss = criterion(output, data)
```

```
                # Backward pass and optimization
                optimizer.zero_grad()
                loss.backward()
                optimizer.step()
                optimizer.zero_grad()

                running_loss += loss.item()
                losses.append(loss.item)
            if epoch%5 == 0:
                torch.save(self.state_dict(), 'modelParams')
                if epoch%3 == 0:
                    self.plotCurrentAbility()

            #print(f">>epoch [{epoch}], Loss: {running_loss/len(train_loader):.
↪4f}")



            print(f"Epoch [{epoch+1}/{num_epochs}], Loss: {running_loss/
↪len(train_loader):.4f}")

        plt.plot(losses)


    def plotCurrentAbility(self):
        self.eval
        self.to('cpu')

        plt.subplot(3,2,1)
        plt.imshow(np.reshape(autoencoder(toNNformat(X_train))[0][0].detach().
↪numpy(),(28,28)))
        plt.subplot(3,2,2)
        plt.imshow(np.reshape(X_train[0],(28,28)))
        plt.subplot(3,2,3)
        plt.imshow(np.reshape(autoencoder(toNNformat(X_train))[0][1].detach().
↪numpy(),(28,28)))
        plt.subplot(3,2,4)
        plt.imshow(np.reshape(X_train[1],(28,28)))
        plt.subplot(3,2,5)
        plt.imshow(np.reshape(autoencoder(toNNformat(X_train))[0][2].detach().
↪numpy(),(28,28)))
        plt.subplot(3,2,6)
        plt.imshow(np.reshape(X_train[2],(28,28)))

        self.train()
        self.to('mps')
```

```
def toNNformat(x):
    xformated = tensor(x).float() / 255.0
    return xformated.unsqueeze(0)
```

[14]:
```
autoencoder = Autoencoder()
```

Las dos siguientes celdas son relativas al entrenamiento, no aconsejamos su ejecución.

[69]:
```
X_train_tensor = toNNformat(X_train)
train_loader = DataLoader(X_train_tensor, batch_size=128, shuffle=True)
```

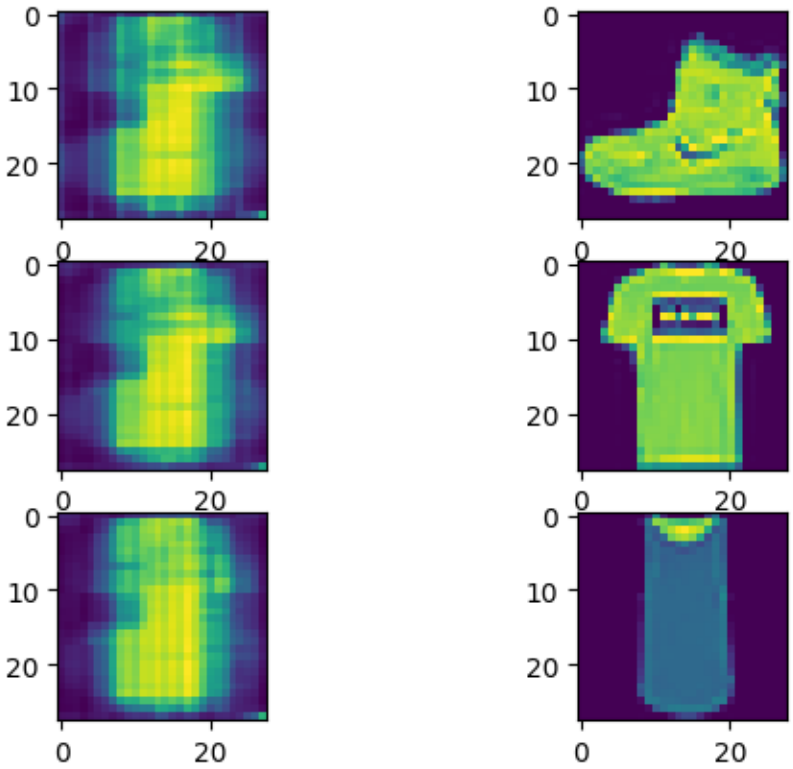
[ ]:
```
autoencoder.load_state_dict(torch.load('modelParams'))
autoencoder.trainModel(train_loader)

# Epoch [1/2000], Loss: 0.1927
# Epoch [2/2000], Loss: 0.1894
# Epoch [3/2000], Loss: 0.1855
# Epoch [4/2000], Loss: 0.1809
# Epoch [5/2000], Loss: 0.1760
# Epoch [6/2000], Loss: 0.1708
# ...
# Epoch [100/2000], Loss: 0.0797
# Epoch [101/2000], Loss: 0.0796
# Epoch [102/2000], Loss: 0.0795
# Epoch [103/2000], Loss: 0.0794
# Epoch [104/2000], Loss: 0.0793
# Epoch [105/2000], Loss: 0.0792
```

La celda que sigue, en cambio, puede ejecutarse con cautela. Si se ejecuto la definición de clase y la instanciación, debería funcionar. Actualmente está *hardcodeada* para procesadores Apple Silicon.

[15]:
```
autoencoder.load_state_dict(torch.load('modelParams'))


autoencoder.eval
autoencoder.to('cpu')

plt.subplot(3,2,1)
plt.imshow(np.reshape(autoencoder(toNNformat(X_train))[0][0].detach().
 ↪numpy(),(28,28)))
plt.subplot(3,2,2)
plt.imshow(np.reshape(X_train[0],(28,28)))
plt.subplot(3,2,3)
plt.imshow(np.reshape(autoencoder(toNNformat(X_train))[0][1].detach().
 ↪numpy(),(28,28)))
plt.subplot(3,2,4)
```

```
plt.imshow(np.reshape(X_train[1],(28,28)))
plt.subplot(3,2,5)
plt.imshow(np.reshape(autoencoder(toNNformat(X_train))[0][2].detach().
  ↪numpy(),(28,28)))
plt.subplot(3,2,6)
plt.imshow(np.reshape(X_train[2],(28,28)));
```

[15]: `<matplotlib.image.AxesImage at 0x17ccfbd90>`



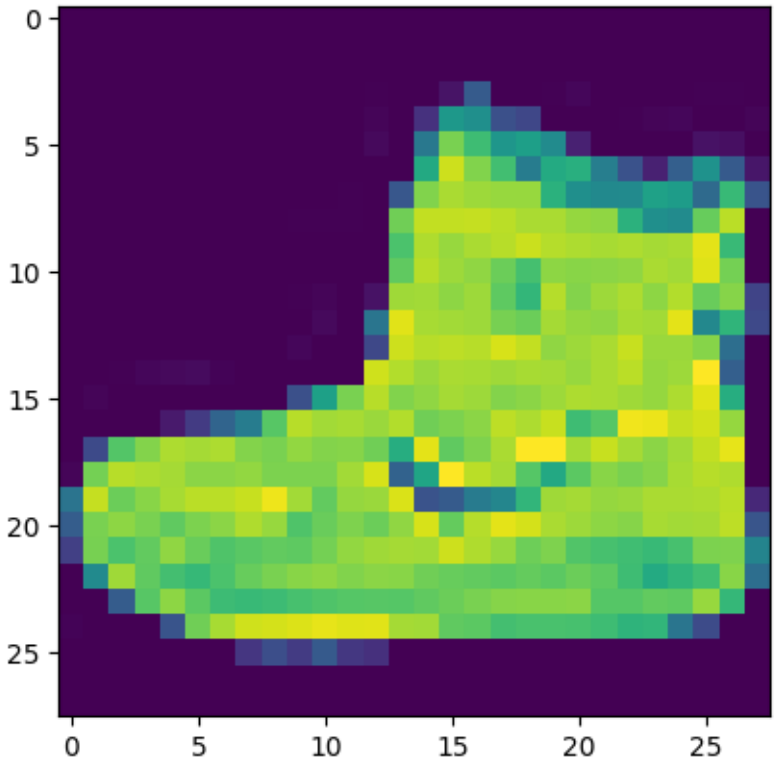
Como se puede ver, el autoencoder después de 100 epoch ofrece muy malos resultados, fallando en reconstruir en la columna de la izquierda las imágenes que se ven en la derecha. No creemos, no obstante que se deba a que la idea de usar un autoencoder es mala, ni creemos que la arquitectura y resto de parámetros de optimización sean desafortunados; creemos que hace falta más cómputo. No solo llevó mucho entrenarlo, sólo evaluar la celda superior lleva su tiempo. Habiendo visto los resultados que ofrece UMAP, no creemos oportuno insitir en la idea del autoencoder (aunque inicialmente nos ilusionaba).

[3]:
```
X_train_5D = umap.UMAP( n_components=5).fit_transform(X_train)
# vemos como queda el primer elemento
print(f'A partir de ahora trabajaremos en la 5D, eso significa \nque confiamos␣
  ↪en la hipótesis de que el siguiente vector e imagen \nportan la misma␣
  ↪información.\n\nVector:{X_train_5D[0]}\nImagen:')
```

```
plt.imshow(np.resize(X_train[0],(28,28)));
```

```
/Users/pepe/carrera/3/2/ma2/ma2/lib/python3.13/site-
packages/sklearn/utils/deprecation.py:151: FutureWarning: 'force_all_finite' was
renamed to 'ensure_all_finite' in 1.6 and will be removed in 1.8.
  warnings.warn(
OMP: Info #276: omp_set_nested routine deprecated, please use
omp_set_max_active_levels instead.
```

```
A partir de ahora trabajaremos en la 5D, eso significa
que confiamos en la hipótesis de que el siguiente vector e imagen
portan la misma información.

Vector:[9.390443  5.066862  2.461857  6.2722044 6.642136 ]
Imagen:
```



1.3 2. Clusterización

2.1 Primer intento de grid search (usando una matriz 50000x5) A partir de la idea de *No Free Lunch* (https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=585893) (y de los requisitos de la práctica), decidimos probar varios algoritmos de clusterización. Usamos `GridSearchCV` con `cv = 2`, es decir, para cada tupla (modelo, parametrós), se entrenó dos veces con conjuntos disjuntos. Para garantizar robustez en los resultados.

Para ello, tuvimos que heredar los algoritmos de sklearn y definir `predict` para aquellos que no tenían definido (idea encontrada en https://stackoverflow.com/questions/68920638/dbscan-object-has-no-attribute-predict-using-gridsearchcv-pipeline/70291704#70291704).

Algunos algoritmos piden y necesitan el número de clusters de antemano, y otros se arreglan ellos mismos para conseguir ese número, como vamos a comparar la calidad de los algoritmos con respecto a las labels, no tiene mucho sentido no fijar el k a 10 para aquellos que lo pidan; sabemos de antemano que los mejores resultados (si los comparamos con y_{label}) serán aquellos que por lo menos compartan número de k (frente a los que no). Esto es una injusticia porque los algoritmos que tienen que descubrir k ellos mismos, tendrán que esforzarse y (como hemos comentado en la sección 1) sigue sin ser claro o objetivo que tenga que ser 10 el número de clusters.

[18]:
```
class AgglomerativeClusteringWrapper(AgglomerativeClustering):
    def predict(self,X):
      return self.labels_.astype(int)

class DBSCANWrapper(DBSCAN):
    def predict(self,X):
      return self.labels_.astype(int)

class HDBSCANWrapper(HDBSCAN):
    def predict(self,X):
      return self.labels_.astype(int)

class OPTICSWrapper(OPTICS):
    def predict(self,X):
        return self.labels_.astype(int)

class SpectralClusteringWrapper(SpectralClustering):
    def predict(self,X):
        return self.labels_.astype(int)

clustering_methods = {
    'KMeans': KMeans(),
    'DBSCAN': DBSCANWrapper(),
    'HDBSCAN': HDBSCANWrapper(),
    'MeanShift': MeanShift(),
    'OPTICS': OPTICSWrapper(),
    #'AffinityPropagation': AffinityPropagation()
    #'SpectralClustering': SpectralClusteringWrapper(),
    #'Agglomerative': AgglomerativeClusteringWrapper()

}



param_grids = {
    'KMeans': {
```

```
        'n_clusters': [10],
        'init': ['k-means++', 'random'],
        #'max_iter': [300, 500]
    },
    'DBSCAN': {
        'eps': [10],
        'min_samples': [5, 50]
    },

    'HDBSCAN': {
        'min_cluster_size': [50, 100, 500],
        'max_cluster_size': [4500, 5500, 6500],
    },

    'MeanShift': {
        'bandwidth': [3.7,4.5]
    },

    'OPTICS': {
        'min_samples': [50, 100]
    }#,

    #'AffinityPropagation': {
    #    'damping': [0.5]
    #}
    #'SpectralClustering': {
    #    'n_clusters': [9]
    #},

    #,
    #'AgglomerativeClustering': {
    #    'n_clusters': [ 9]
    #    #'linkage': ['ward']#, 'complete', 'average']
    #}

}

# Function to evaluate each model and its hyperparameters
def try_clustering(X_train, y_train,  clustering_methods, param_grids, verbose␣
 ↪= True):
    results = {}
    best_score = -float('inf')
    models = []
    for name, model in clustering_methods.items():
        start = time()
```

```
        grid_search = GridSearchCV(model, param_grids[name], cv=2,␣
↪scoring='adjusted_rand_score', n_jobs=4)
        grid_search.fit(X_train, y_train)


        score = grid_search.best_score_
        models.append(grid_search.best_estimator_)
        params = grid_search.best_params_
        end = time()

        if verbose: print(f'Modelo: {name}. \nAdjMutualInformation: {score}.␣
↪\nParametros: {params}. \nTiempo: {round(end-start,2)}s (o␣
↪{round(end-start,2)/60}min) .\n\n')

        results[name] = [params, score, round(end-start,2)]

    results_df = pd.DataFrame.from_dict(results, orient='index',␣
↪columns=['Mejores parámetros', 'Adjusted Rand Score', 'Seconds'])

    # Reset index to make the model name a column
    results_df.reset_index(inplace=True)
    results_df.rename(columns={'index': 'Algoritmo'}, inplace=True)

    # Sort the DataFrame by 'Adjusted Mutual Info Score' in descending order
    results_df = results_df.sort_values(by='Adjusted Rand Score',␣
↪ascending=False)

    models = [models[i] for i in results_df.index.tolist()]  # esto ordena los␣
↪modelos en una lista


    return results_df, models
```

[19]:
```
results, models = try_clustering(X_train_5D, y_train, clustering_methods,␣
↪param_grids)
```

```
Modelo: KMeans.
AdjMutualInformation: 0.4861175102419829.
Parametros: {'init': 'random', 'n_clusters': 10}.
Tiempo: 1.49s (o 0.024833333333333332min) .


Modelo: DBSCAN.
AdjMutualInformation: 0.0.
Parametros: {'eps': 10, 'min_samples': 5}.
Tiempo: 103.82s (o 1.7303333333333333min) .
```

```
/Users/pepe/carrera/3/2/ma2/ma2/lib/python3.13/site-
packages/joblib/externals/loky/process_executor.py:752: UserWarning: A worker
stopped while some jobs were given to the executor. This can be caused by a too
short worker timeout or by a memory leak.
  warnings.warn(

Modelo: HDBSCAN.
AdjMutualInformation: 7.887468891222135e-06.
Parametros: {'max_cluster_size': 5500, 'min_cluster_size': 100}.
Tiempo: 28.23s (o 0.47050000000000003min) .


Modelo: MeanShift.
AdjMutualInformation: 0.30161579041501896.
Parametros: {'bandwidth': 3.7}.
Tiempo: 311.58s (o 5.193min) .


Modelo: OPTICS.
AdjMutualInformation: 4.654200682021259e-06.
Parametros: {'min_samples': 100}.
Tiempo: 82.98s (o 1.383min) .
```

[20]:
```
results
```

[20]:

	Algoritmo	Mejores parámetros	Adjusted Rand Score	Seconds
0	KMeans	{'init': 'random', 'n_clusters': 10}	0.486118	1.49
3	MeanShift	{'bandwidth': 3.7}	0.301616	311.58
2	HDBSCAN	{'max_cluster_size': 5500, 'min_cluster_size':…	0.000008	28.23
4	OPTICS	{'min_samples': 100}	0.000005	82.98
1	DBSCAN	{'eps': 10, 'min_samples': 5}	0.000000	103.82

En la celda superior se puede ver una tabla con el resultado de 5 algoritmos con sus parámetros. Extraemos la siguiente información: - La elección de hiperparámetros es crucial y hace que algoritmos que sabemos que son muy buenos como HDBSCAN (https://youtu.be/dGsxd67IFiU?si=eFei7SSAKWZ7C_Ql) ofrezacan un mal rendimiento. - Aunque sabemos que a priori KMeans es peor (hipótesis de cluster con forma gausiana, convexa que por las gráficas anteriores sabemos que no se cumplen), como tiene la ventaja de saber el número

de clusters, ofrece el mejor score. - Los algoritmos que estan comentados no han sido ejecutados por lo intratable que era. Esto ocurre porque los recursos de tiempo y memoria que consume cada algoritmo pueden diferir enormemente entre sí, aunque hagan cosas parecidas. Además influye enormemente la instancia del algoritmo y la implementación. En nuestro caso encontramos que el archivo adjunto `gridsearch.py`, después de toda una noche de ejecución solo pudo calcular los que arriba se presentan ahora.

Motivados por la intratabilidad de los algoritmos menos eficientes y por la necesidad de explorar más y mejores hiperparámetros, vamos a volver a hacer pruebas pero con muchos menos ejemplos

2.2 Segundo intento de grid search (usando una matriz de 5000x5)

[36]:
```
indexes = np.random.choice(50_000, 5_000)
X_train5D_1000_sampled = X_train_5D[indexes,:]
y_train_1000_sampled = y_train[indexes]

clustering_methods = {
    'KMeans': KMeans(),
    'DBSCAN': DBSCANWrapper(),
    'HDBSCAN': HDBSCANWrapper(),
    'MeanShift': MeanShift(),
    'OPTICS': OPTICSWrapper(),
    'SpectralClustering': SpectralClusteringWrapper(),
    'AgglomerativeClustering': AgglomerativeClusteringWrapper()

}


param_grids = {
    'KMeans': {
        'n_clusters': [10],
        'init': ['k-means++', 'random'],
        #'max_iter': [300, 500]
    },
    'DBSCAN': {
        'eps': [0.1, 0.5, .75, 1, 2, 5, 10],
        'min_samples': [2, 3, 5,10,25, 50]
    },

    'HDBSCAN': {
        'min_cluster_size': [2, 3, 5, 10, 15, 20, 25],
        'max_cluster_size': [10, 15, 30, 40, 50, 60, 70],
    },

    'MeanShift': {
        'bandwidth': [3.7,4.5]
    },
```

```
    'OPTICS': {
        'min_samples': [10, 50, 100, 500, 600],
        'metric': ['cityblock', 'cosine', 'euclidean', 'l1', 'l2', 'manhattan']
    },

    'SpectralClustering': {
        'n_clusters': [10],
        'affinity': ['nearest_neighbors', 'rbf'],
        'n_neighbors': [5, 10, 20, 50]
    },

    'AgglomerativeClustering': {
        'n_clusters': [10],
        'linkage': ['ward', 'complete', 'average', 'single'],
    }

}

results, models = try_clustering(X_train5D_1000_sampled, y_train_1000_sampled,␣
↪clustering_methods, param_grids, verbose=False)
```

```
/Users/pepe/carrera/3/2/ma2/ma2/lib/python3.13/site-
packages/sklearn/manifold/_spectral_embedding.py:329: UserWarning: Graph is not
fully connected, spectral embedding may not work as expected.
  warnings.warn(
/Users/pepe/carrera/3/2/ma2/ma2/lib/python3.13/site-
packages/sklearn/manifold/_spectral_embedding.py:329: UserWarning: Graph is not
fully connected, spectral embedding may not work as expected.
  warnings.warn(
/Users/pepe/carrera/3/2/ma2/ma2/lib/python3.13/site-
packages/sklearn/manifold/_spectral_embedding.py:329: UserWarning: Graph is not
fully connected, spectral embedding may not work as expected.
  warnings.warn(
/Users/pepe/carrera/3/2/ma2/ma2/lib/python3.13/site-
packages/sklearn/manifold/_spectral_embedding.py:329: UserWarning: Graph is not
fully connected, spectral embedding may not work as expected.
  warnings.warn(
/Users/pepe/carrera/3/2/ma2/ma2/lib/python3.13/site-
packages/sklearn/manifold/_spectral_embedding.py:329: UserWarning: Graph is not
fully connected, spectral embedding may not work as expected.
  warnings.warn(
/Users/pepe/carrera/3/2/ma2/ma2/lib/python3.13/site-
packages/sklearn/manifold/_spectral_embedding.py:329: UserWarning: Graph is not
fully connected, spectral embedding may not work as expected.
  warnings.warn(
/Users/pepe/carrera/3/2/ma2/ma2/lib/python3.13/site-
packages/sklearn/manifold/_spectral_embedding.py:329: UserWarning: Graph is not
fully connected, spectral embedding may not work as expected.
```

```
  warnings.warn(
/Users/pepe/carrera/3/2/ma2/ma2/lib/python3.13/site-
packages/sklearn/manifold/_spectral_embedding.py:329: UserWarning: Graph is not
fully connected, spectral embedding may not work as expected.
  warnings.warn(
/Users/pepe/carrera/3/2/ma2/ma2/lib/python3.13/site-
packages/sklearn/manifold/_spectral_embedding.py:329: UserWarning: Graph is not
fully connected, spectral embedding may not work as expected.
  warnings.warn(
```

[37]:
```
results
```

[37]:

	Algoritmo	Mejores parámetros	Adjusted Rand Score	Seconds
0	KMeans	{'init': 'k-means++', 'n_clusters': 10}	0.473153	0.06
3	MeanShift	{'bandwidth': 3.7}	0.297455	11.73
2	HDBSCAN	{'max_cluster_size': 30, 'min_cluster_size': 3}	0.000926	1.14
5	SpectralClustering	{'affinity': 'nearest_neighbors', 'n_clusters'…	0.000697	3.33
6	AgglomerativeClustering	{'linkage': 'average', 'n_clusters': 10}	0.000625	0.41
1	DBSCAN	{'eps': 0.5, 'min_samples': 10}	0.000463	0.55
4	OPTICS	{'metric': 'cosine', 'min_samples': 100}	0.000398	16.90

Vemos que solo KMeans y MeanShift ofrecen buenos valores para Adjusted Rand Score. ¿Significa claramente que el resto son malos? Vamos a visualizar los resultados sobre una matriz $X_{5000,2}$. No podemos hacerlo directamente sobre $X_{50000,2}$ porque no todos los algoritmos pueden predecir sobre nuevos datos.

2.3 Visualización con 1000 puntos (usados en gridsearch)

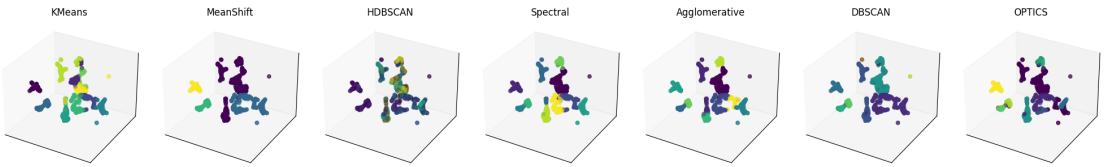
[45]:
```
labels = [models[i].labels_ for i in range(len(models))]
reducer3d = umap.UMAP(n_components=3)
X_train5D_1000_sampled_3D = reducer3d.fit_transform(X_train5D_1000_sampled)
```

```
/Users/pepe/carrera/3/2/ma2/ma2/lib/python3.13/site-
packages/sklearn/utils/deprecation.py:151: FutureWarning: 'force_all_finite' was
renamed to 'ensure_all_finite' in 1.6 and will be removed in 1.8.
  warnings.warn(
```

[8]:
```
def getModelName(i):
```

```
    return str(models[i].__repr__).split('of ')[1].split('(')[0].
 ↪split('Wrapper')[0].split('Clustering')[0] # se me ocurrió esto, y como␣
 ↪funciona y es rápido yo lo dejaba
```

[55]:
```
fig, axes = plt.subplots(1, 7, figsize=(20, 3), subplot_kw={'projection': '3d'})

# Loop over each subplot and plot a scatter plot
for i, ax in enumerate(axes):
    ax.scatter(X_train5D_1000_sampled_3D[:,0],X_train5D_1000_sampled_3D[:
 ↪,1],X_train5D_1000_sampled_3D[:,2],alpha=.1,c=labels[i])
    ax.set_title(getModelName(i))

    ax.set_xticks([])
    ax.set_yticks([])
    ax.set_zticks([])


plt.tight_layout()
plt.show()
```



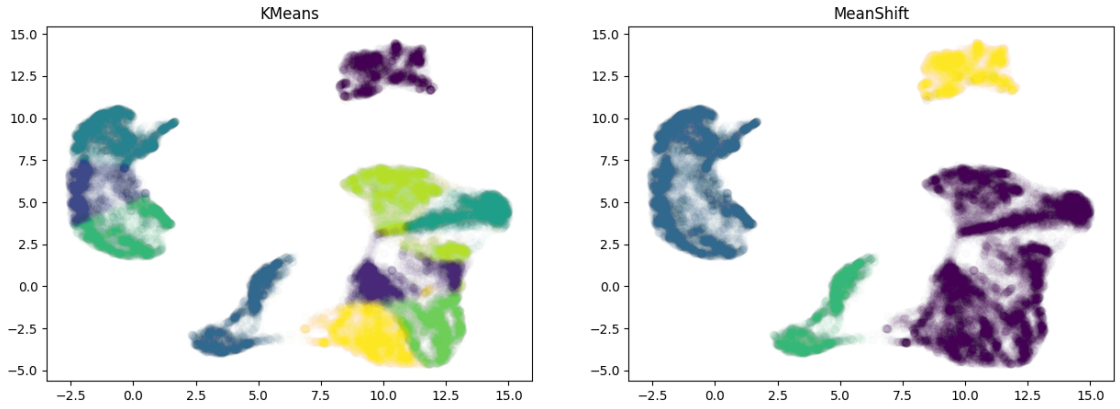
Ahora la interpretación es algo subjetiva, por la similitud de rendimiento de los diferentes modelos. Por ejemplo, sin duda parece que MeanShift lo hace mucho mejor que Kmeans. ¿Es esto cierto? Como comentábamos antes, no podemos usar todos los algoritmos directamente en la matriz grande, pero precisamente MeanShift y Kmeans si podemos usarlos para predecir sobre nuevos datos, hagámoslo y veamos.

2.4 Visualización con 50000 (algoritmos seleccionados prediciendo sobre datos no vistos)

[56]:
```
# sobre los totales
plt.figure(figsize=(15, 5))
labels = [models[i].predict(X_train_5D) for i in range(len(models))]

def getModelName(i):
    return str(models[i].__repr__).split('of ')[1].split('(')[0] # se me␣
 ↪ocurrió esto, y como funciona y es rápido yo lo dejaba

for i, model in enumerate(models[:2]):
    plt.subplot(1,len(models[:2]),i+1)
    plt.title(getModelName(i))
```

```
    print(getModelName(i))
    plt.scatter(X_train_reduced[:,0],X_train_reduced[:
↪,1],alpha=1e-2,c=labels[i])
```

```
KMeans
MeanShift
```



Ahora la cosa es distinta, vemos que los clusters superfluos que hacía KMeans con 1000 datos cobran sentido con esos nuevos datos adicionales. No obstante la patata de la derecha que habíamos examinado antes ahora esta muy mal segmentada. Esto es devido a que los clusters que existen dentro de la región no son convexos y KMeans no tiene forma de distinguirlos. Por otro lado MeanShift ofrece una clusterización conservadora pero más coherente.

Por su alta puntuación con respecto a las etiquetas originales y su no claramente mala visualización (lo que no nos gusta es que creemos que podría haber clusterizado en más clusters, pero para la decisión que tomó *(ya vimos al principio que no era una decisión trivial)* no lo hace mal), vamos a escoger a MeanShift y a compararlo de una forma más paramétrica contra HDBSCAN, que era nuestro favorito inicialmente (motivados en parte por https://youtu.be/AgPQ76RIi6A?si=N0MXpixgdhJEgrbI, https://youtu.be/dGsxd67IFiU?si=7XBYPe6LOacedKCz).

1.4 3. Evaluación y resultados

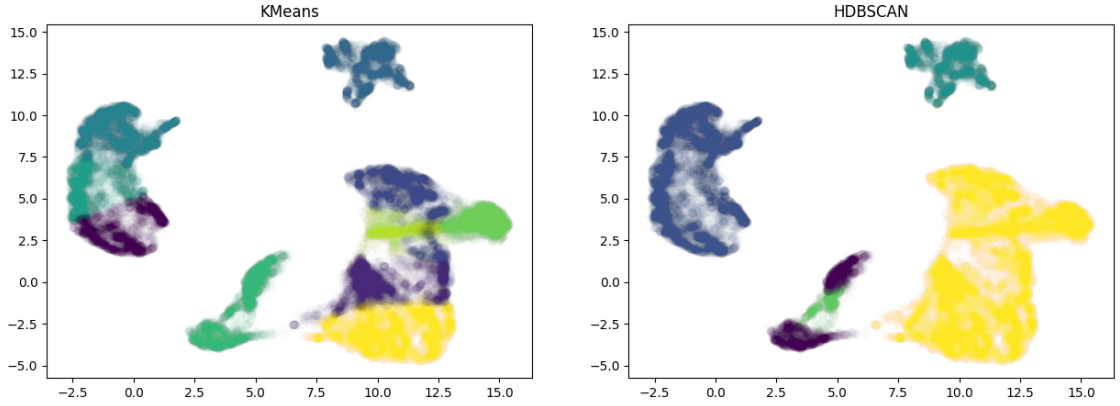
3.1 Comparativa final entre MeanShift y HDBSCAN En lo que prosigue se compararán de una forma más formal MeanShift y HDBSCAN para finalmente quedarnos con uno, procedemos primero a entrenarlos usando todo X_train con cinco columnas reducidas por UMAP. Insintimos en que nuestra decisión de realizar la comparativa final se basa en los siguentes puntos: - Suponiendono las etiquetas como algo muy importante y suponiéndo entonces que buenos clusters serán aquellos que se parezan a las etiquetas, entonces nuestros experimentos con gridsearch nos dejaron claro que tiene que ser o kmeans o meanshift. - Kmeans es mucho más veloz que mean shift, el cual no vimos mucha diferencia en cuanto a expresividad. - Suponiendo que las etiquetas son una guía o una refencia pero que nuestra tarea se basa en descubrir nuevo conocimiento, en encontrar patrones, en dividir el espacio en regiones densas… sabemos que el DBSCAN jerárquico que contempla diferentes *Eps* para diferentes clusters *(Sección 1.1)*, entonces HDBSCAN es un buen algoritmo para esto. -

Las plots en 2d de HSDBSCAN (Seccion 1.3) y KMeans (Seccion 2.3 y 2.4) nos convencen de que lo hacen bien.

[4]:
```
kmeans = KMeans(10).fit(X_train_5D)
```

[33]:
```
hdbscan = HDBSCAN(min_samples=4000).fit(X_train_5D)
```

[34]:
```
labels = [kmeans.labels_, hdbscan.labels_]
models = [kmeans, hdbscan]

plt.figure(figsize=(15, 5))

for i, model in enumerate(models):
    plt.subplot(1,len(models),i+1)
    plt.title(getModelName(i))
    plt.scatter(X_train_reduced[:,0],X_train_reduced[:
 ↪,1],alpha=1e-2,c=labels[i])
```



3.1.1 Métricas

[35]:
```
ari_kmeans = adjusted_rand_score(y_train, kmeans.labels_)
ari_hdbscan = adjusted_rand_score(y_train, hdbscan.labels_)

nmi_kmeans = normalized_mutual_info_score(y_train, kmeans.labels_)
nmi_hdbscan = normalized_mutual_info_score(y_train, hdbscan.labels_)

silhouette_kmeans = silhouette_score(X_train, kmeans.labels_)
silhouette_hdbscan = silhouette_score(X_train, hdbscan.labels_)

fmi_kmeans = fowlkes_mallows_score(y_train, kmeans.labels_)
fmi_hdbscan = fowlkes_mallows_score(y_train, hdbscan.labels_)

metrics_data = {
    "Metric": ["ARI", "NMI", "Silhouette", "FMI"],
```

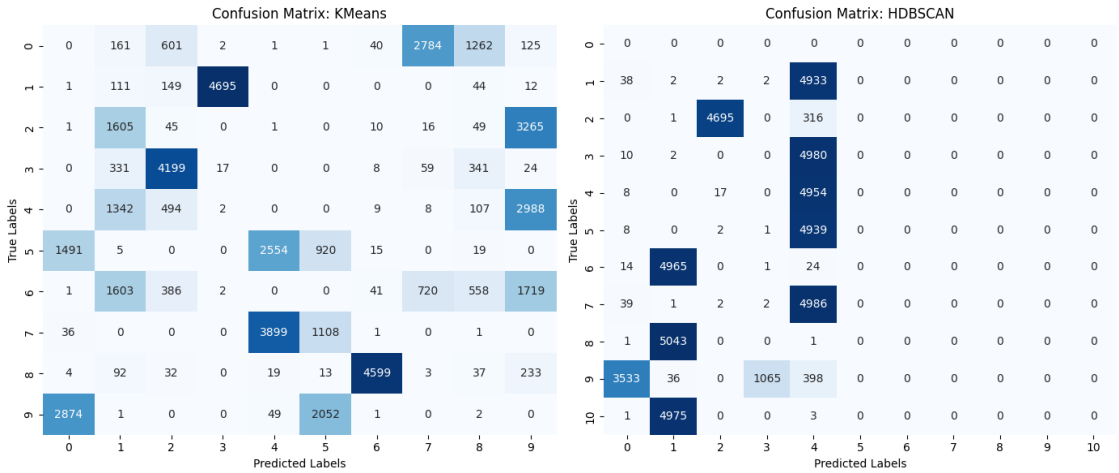
```
    "KMeans": [ari_kmeans, nmi_kmeans, silhouette_kmeans, fmi_kmeans],
    "HDBSCAN": [ari_hdbscan, nmi_hdbscan, silhouette_hdbscan, fmi_hdbscan]
}

metrics_df = pd.DataFrame(metrics_data)

metrics_df
```

[35]:

	Metric	KMeans	HDBSCAN
0	ARI	0.479808	0.289769
1	NMI	0.632515	0.611368
2	Silhouette	0.093816	0.092079
3	FMI	0.534833	0.488691

3.1.2 Matriz de confusion

[36]:
```
conf_matrix_mean_shift = confusion_matrix(y_train, kmeans.labels_)
conf_matrix_hdbscan = confusion_matrix(y_train, hdbscan.labels_)

# Plot confusion matrices side by side
fig, ax = plt.subplots(1, 2, figsize=(14, 6))

# Plot MeanShift confusion matrix
sns.heatmap(conf_matrix_mean_shift, annot=True, fmt="d", cmap="Blues",␣
 ↪cbar=False, ax=ax[0])
ax[0].set_title('Confusion Matrix: KMeans')
ax[0].set_xlabel('Predicted Labels')
ax[0].set_ylabel('True Labels')

# Plot HDBSCAN confusion matrix
sns.heatmap(conf_matrix_hdbscan, annot=True, fmt="d", cmap="Blues", cbar=False,␣
 ↪ax=ax[1])
ax[1].set_title('Confusion Matrix: HDBSCAN')
ax[1].set_xlabel('Predicted Labels')
ax[1].set_ylabel('True Labels')

plt.tight_layout()
plt.show()
```



1.4.1 3.2 Analisis

Sobre las métricas decir que KMeans supone una victoria consistente en todas ellas. De forma que, si deseásemos clusterizar para aplicarlo en un contexto semisupervisado, sin duda KMeans es mejor. Por otro lado, en un contexto real podríamos no tener etiquetas de nuevos datos, o podría existir una deriva de concepto, en general son varias las razones por las que se podría desconfiar de un uso ciego de las métricas.

Antes de comparar la matriz de confusión recordemos que:

0	Camiseta	1	Pantalón	2	Jersey	3	Vestido	4	Abrigo	5
Sandalia	6	Camisa	7	Tenis	8	Bolso	9	Bota		

Lo más distintivo y caracteristico es la tendencia de KMeans a ser más homogeneo que íntegro y por contrapartida, DBSCAN tiende a ser mucho más íntegro pero menos homogeneo.

1.5 3. Conclusión

La clusterización es un problema inherentemente difícil, incluso imposible desde ciertas perspectivas teóricas (https://proceedings.neurips.cc/paper_files/paper/2002/file/43e4e6a6f341e00671e123714de019a8-Paper.pdf). En nuestro enfoque, hemos priorizado decisiones fundamentadas en el conocimiento del dominio tras un análisis exhaustivo de los datos, evaluando múltiples algoritmos para obtener la solución más robusta posible. Aunque las limitaciones computacionales nos impidieron ejecutar algunos algoritmos potencialmente valiosos con el conjunto completo de datos, y reconocemos que un autoencoder habría mejorado significativamente la representación vectorial de características distintivas como la “camisidad” o “bolsidad”, los resultados obtenidos representan un compromiso óptimo entre precisión, eficiencia y recursos disponibles. Este trabajo establece una base sólida para futuras investigaciones que, con mayor capacidad computacional, podrían refinar aún más la caracterización de estos objetos complejos.