

```
[2]: %matplotlib inline
```

```
[3]: %load_ext autoreload  
%autoreload 2
```

Ejemplo CAE sobre dataset de MNIST 1

En este notebook se muestra la aplicación de un Autoencoder Convolucional sobre el Dataset de MNIST.

El Dataset de MNIST es uno de los más utilizados en cursos y formaciones con redes neuronales debido a su sencillez y que no requiere ningún preprocesado de las imágenes. Contiene un total de 60.000 muestras (50k de entrenamiento y 10k de test) etiquetadas de números de un dígito escritos a mano y el tamaño de todas las imágenes es de 28x28 píxeles (784 variables).

El objetivo de la aplicación del CAE será ver si es posible reducir estas 784 variables a un espacio latente mucho menor sin una alta pérdida de información. Posteriormente se aplicarán modelos de ML tanto supervisados como no supervisados para el análisis de la validez de estas variables latentes generadas.

Librerías

Básicas

```
[4]: import urllib  
import gzip  
import pickle  
import os  
import yaml  
from datetime import datetime
```

Datos

```
[5]: import numpy as np  
import pandas as pd
```

Machine Learning

```
[6]: from keras.callbacks import TensorBoard  
  
from sklearn.decomposition import PCA  
from sklearn.manifold import TSNE  
  
from sklearn.cluster import KMeans  
from sklearn.linear_model import LogisticRegression  
from sklearn.metrics import classification_report  
from sklearn.metrics import confusion_matrix
```

Using TensorFlow backend.

Visualización

```
[7]: from matplotlib import pyplot as plt  
from matplotlib import image as mpimage  
import seaborn as sns
```

Propias

```
[8]: from model.cae import CAE
      from model import metrics

      from utils import plot_paired_imgs
      from utils import plot_sample_imgs
```

```
[9]: def reshape_mnist(vectors):
      images = np.reshape(vectors, (-1,28,28,1))
      return images
```

Lectura de datos

Los datos están disponibles online y en un objeto binario serializado que es descargado y leído con *pickle*.

```
[10]: mnistfile = 'mnist.pkl.gz'

      if not os.path.isfile(mnistfile):
          url = urllib.request.URLopener()
          url.retrieve("http://deeplearning.net/data/mnist/mnist.pkl.gz", mnistfile)

      with gzip.open(mnistfile, 'rb') as f:
          training_set, validation_set, testing_set = pickle.load(f, encoding='latin1')
```

```
[11]: X_train, y_train = training_set
      X_test, y_test = testing_set
```

Los datos cargados son arrays unidimensionales de 784 valores que son necesarios redimensionar para mostrar las imágenes.

```
[12]: X_train.shape, X_test.shape
```

```
[12]: ((50000, 784), (10000, 784))
```

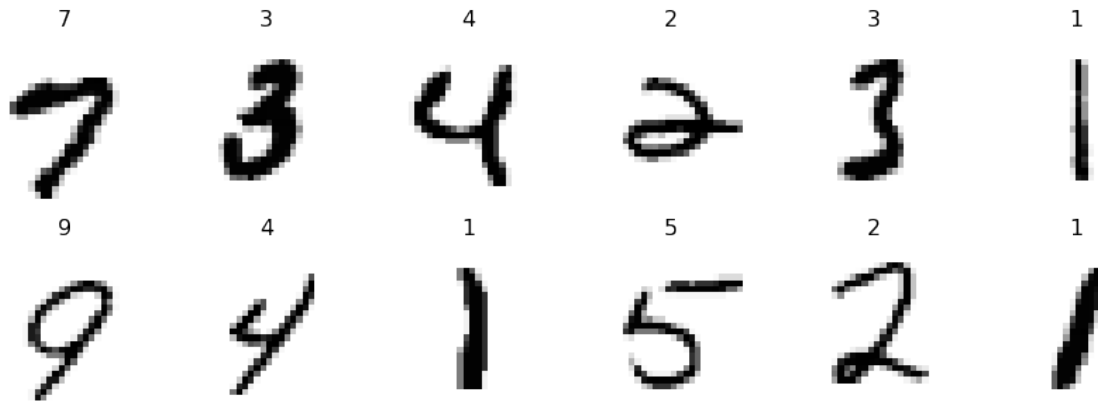
```
[13]: X_train = reshape_mnist(X_train)
      X_test = reshape_mnist(X_test)
```

```
[14]: X_train.shape, X_test.shape
```

```
[14]: ((50000, 28, 28, 1), (10000, 28, 28, 1))
```

Muestra un subset de imágenes. Es relativamente fácil para el ojo humano reconocer cada uno de los dígitos a los que se corresponde. Los vectores Y contienen la etiqueta del dígito representado.

```
[69]: plot_sample_imgs(X_train, 2, 6, color=False, labels=y_train, size=2.5)
```



Entrenamiento del Modelo

Inicializa la arquitectura

La clase CAE se ha creado para poder añadir un nivel mayor de abstracción sobre Keras cuando se define la arquitectura de un autoencoder convolucional. Los parámetros son los siguientes:

- *input_shape*: dimensiones de entrada. Deberán de ser iguales a las dimensiones de las imágenes que entran, en este caso 28x28x1. La última dimensión es el número de canales, que será 1 para imágenes en blanco y negro y 3 para imágenes a color RGB.
- *latent_features*: dimensiones de la capa oculta de variables latentes, esto, el número de variables a las que queremos comprimir las imágenes.
- *filters*: capas convolucionales, cada elemento de la lista es el número de kernels de la capa convolucional.
- *kernel*: tamaño del kernel.
- *stride*: al número de píxeles que avanza la ventana en cada cálculo de la convolución.
- *pool*: tamaño del pooling para realizar el submuestreo después de la capa de convolución.
- *optimizer*: nombre del optimizador a utilizar.
- *lossfn*: métrica para el cálculo de la función de coste.
- *path*: ruta donde guardar/cargar el CAE entrenado y los parámetros.
- *load*: True/False. Cuando se marca como True se ignoran todos los parámetros de definición de la arquitectura y se carga el CAE que esté almacenado en el *path*.

En este caso se han seleccionado 20 variables latentes en la capa oculta y dos capas convolucionales de 8 y 10 kernels.

```
[16]: cae = CAE(input_shape=X_train[0].shape, latent_features=20, filters=[8,16], path='/tmp/
      ↪model')
```

WARNING:tensorflow:From /Users/portizdegallisteo/anaconda3/lib/python3.7/site-packages/tensorflow/python/framework/op_def_library.py:263: colocate_with (from tensorflow.python.framework.ops) is deprecated and will be removed in a future version.

Instructions for updating:
Colocations handled automatically by placer.

Entrenamiento

Para observar los resultados en tiempo real del entrenamiento se utiliza TensorBoard, para ello hay que definir el *callback* e indicar la ruta donde almacenar el *log*. El servidor de TensorBoard debería estar apuntando a esta misma ruta.

```
[17]: tb = TensorBoard(log_dir='tmp/logs/mnist' + datetime.now().strftime('%Y%m%d_%H%M'),  
                        write_grads=True, write_images=True, histogram_freq=1)
```

Es importante darse cuenta que a la hora de entrenar el CAE **la entrada y la salida deben de ser la misma**. Se está intentando reducir codificar una imagen reduciendo el error de reconstrucción al máximo. Se reserva un 10% de los datos de training para validación.

```
[ ]: cae.model.fit(X_train, X_train, epochs=5, callbacks=[tb], validation_split=0.1)
```

Guarda el modelo

El método *save* guarda el modelo en la ruta indicada. Se guardan 3 ficheros distintos: * *params.yaml*: parámetros de la arquitectura. * *encoder.h5*: pesos de las capas del encoder. * *decoder.h5*: pesos de las capas de decoder.

```
[20]: cae.save()
```

```
[21]: os.listdir('/tmp/model')
```

```
[21]: ['decoder.h5', 'encoder.h5', 'params.yaml']
```

```
[22]: with open('/tmp/model/params.yaml') as f:  
        params = yaml.load(f)  
  
params
```

```
[22]: {'filters': [8, 16],  
      'input_shape': (28, 28, 1),  
      'kernel': (3, 3),  
      'latent_features': 20,  
      'lossfn': 'mse',  
      'optimizer': 'adamax',  
      'pool': (2, 2),  
      'stride': 1}
```

Es posible cargar el modelo entrenado únicamente indicando el path donde está guardado.

```
[23]: cae = CAE(path='/tmp/model', load=True)
```

Análisis de la salida

Además de la abstracción sobre la definición de la arquitectura en Keras la clase CAE incluye también como atributos, además del CAE entrenado al completo, el encoder y el decoder como modelos independientes. De esta forma es muy sencillo realizar la codificación y la decodificación de imágenes llamando al método *predict* de estos objetos.

```
[24]: cae.model, cae.encoder, cae.decoder
```

```
[24]: (<keras.engine.training.Model at 0x1c3de1a5c0>,  
      <keras.engine.sequential.Sequential at 0x1a3aba3588>,  
      <keras.engine.sequential.Sequential at 0x1c3dec5048>)
```

Pérdida de reconstrucción

Se puede evaluar la pérdida de reconstrucción de la imagen calculando el error cuadrático medio entre la imagen original y la reconstruida.

```
[25]: cae.model.evaluate(X_train, X_train)
```

```
50000/50000 [=====] - 5s 93us/step
```

```
[25]: 0.01178719367802143
```

```
[26]: cae.model.evaluate(X_test, X_test)
```

```
10000/10000 [=====] - 1s 97us/step
```

```
[26]: 0.011510138456523418
```

En train y test son muy similares por lo que se concluye que no hay overfitting.

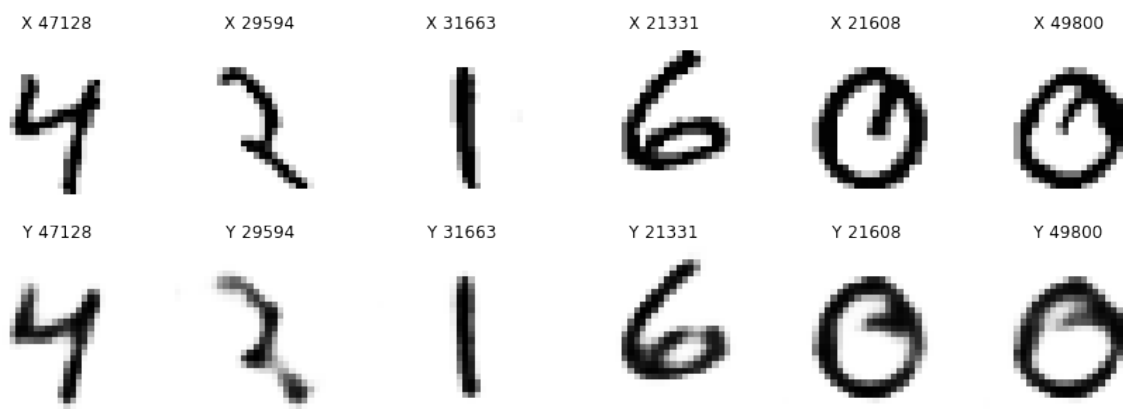
Muestra de imágenes reconstruidas

Con la llamada al método predict el objeto model se obtienen los arrays de las imágenes reconstruidas.

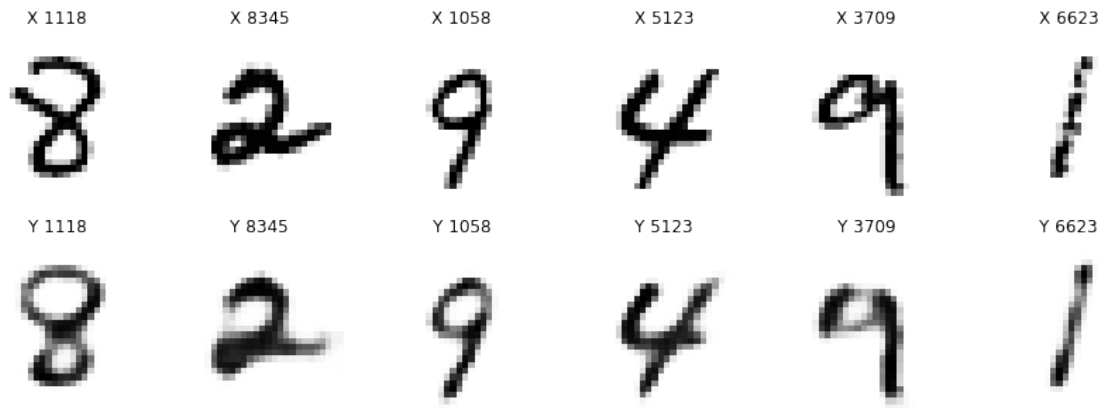
```
[27]: X_train_pred = cae.model.predict(X_train)  
      X_test_pred = cae.model.predict(X_test)
```

Las imágenes reconstruidas son muy similares a las originales tanto en el conjunto de train como en el de test.

```
[30]: plot_paired_imgs(X_train, X_train_pred, N=6, orient='h', color=False, size=2.5)
```



```
[38]: plot_paired_imgs(X_test, X_test_pred, N=6, orient='h', color=False, size=2.5)
```



Obtención de variables latentes

Llamando al método *predict* del encoder se obtendrán los valores de las 20 variables generadas para cada imagen.

```
[31]: X_train_enc = cae.encoder.predict(X_train)
      X_test_enc = cae.encoder.predict(X_test)
```

```
[32]: X_train_enc.shape, X_test_enc.shape
```

```
[32]: ((50000, 20), (10000, 20))
```

Modelo Supervisado

Una de las opciones para analizar la calidad de las variables generadas, además de la pérdida de reconstrucción, es utilizar estas variables como entrada a otros modelos. Por ejemplo en este caso se va a utilizar una Regresión Logística sobre estas variables para predecir el dígito. Es importante notar que debido a la alta dimensionalidad no hubiera sido posible o se hubieran obtenido unos resultados muy pobres utilizando las 784 variables originales como entrada del modelo.

```
[36]: lr = LogisticRegression(multi_class='ovr', solver='lbfgs', max_iter=200)
```

```
[37]: lr.fit(X_train_enc, y_train)
```

```
[37]: LogisticRegression(C=1.0, class_weight=None, dual=False, fit_intercept=True,
      intercept_scaling=1, max_iter=200, multi_class='ovr',
      n_jobs=None, penalty='l2', random_state=None, solver='lbfgs',
      tol=0.0001, verbose=0, warm_start=False)
```

```
[40]: y_train_pred = lr.predict(X_train_enc)
      y_test_pred = lr.predict(X_test_enc)
```

Los resultados del modelo son bastante buenos, con un *accuracy* medio del 90% en el conjunto de test.

```
[41]: print(classification_report(y_train, y_train_pred))
```

```
precision    recall  f1-score   support
```

0	0.93	0.96	0.95	4932
1	0.95	0.97	0.96	5678
2	0.87	0.86	0.87	4968
3	0.87	0.86	0.87	5101
4	0.90	0.88	0.89	4859
5	0.83	0.83	0.83	4506
6	0.91	0.94	0.92	4951
7	0.90	0.92	0.91	5175
8	0.86	0.83	0.84	4842
9	0.85	0.83	0.84	4988
micro avg	0.89	0.89	0.89	50000
macro avg	0.89	0.89	0.89	50000
weighted avg	0.89	0.89	0.89	50000

```
[42]: print(classification_report(y_test, y_test_pred))
```

	precision	recall	f1-score	support
0	0.93	0.98	0.95	980
1	0.96	0.98	0.97	1135
2	0.89	0.86	0.88	1032
3	0.88	0.89	0.88	1010
4	0.92	0.91	0.91	982
5	0.86	0.84	0.85	892
6	0.91	0.92	0.91	958
7	0.92	0.91	0.91	1028
8	0.86	0.87	0.86	974
9	0.87	0.85	0.86	1009
micro avg	0.90	0.90	0.90	10000
macro avg	0.90	0.90	0.90	10000
weighted avg	0.90	0.90	0.90	10000

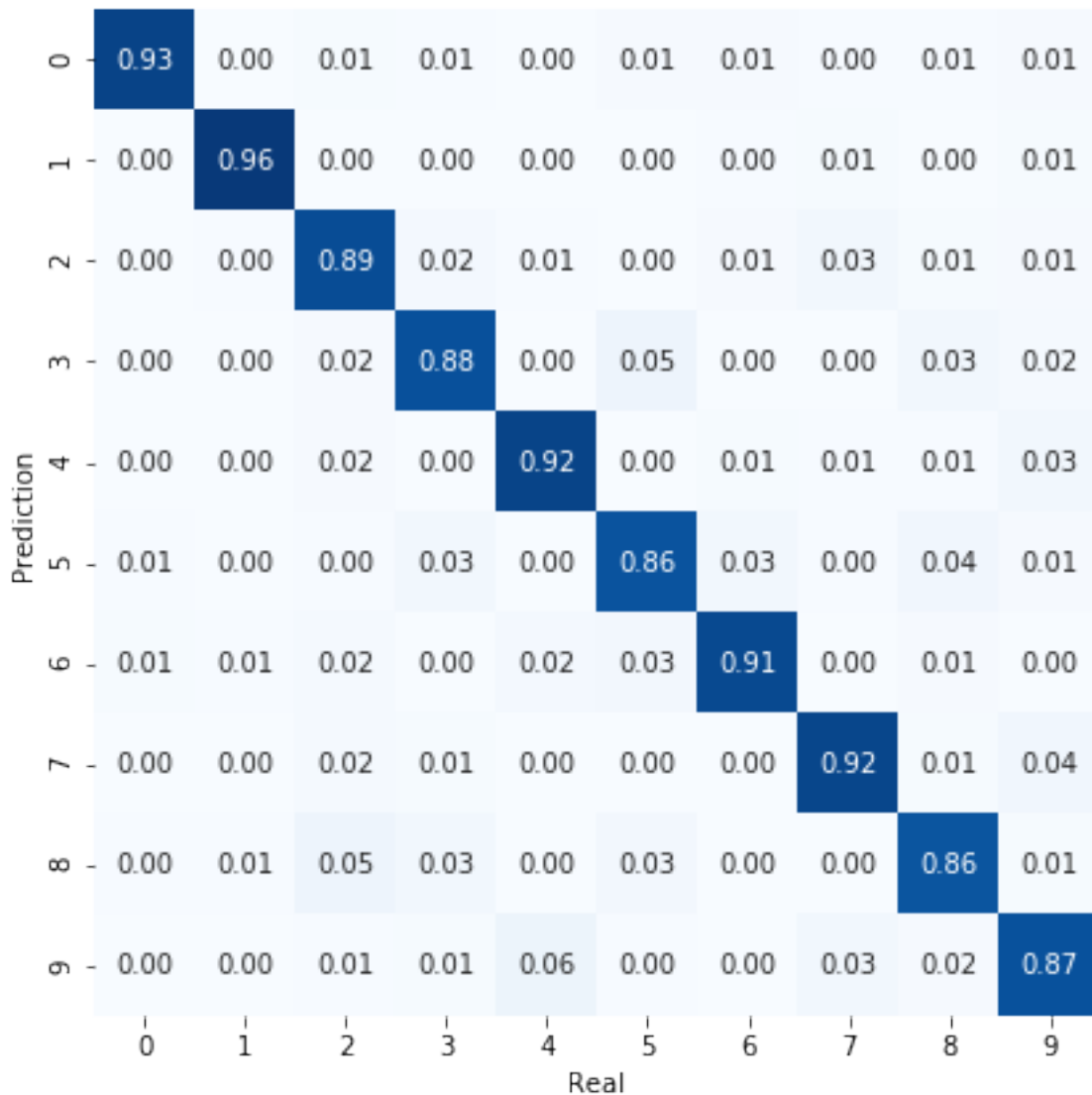
Mediante la matriz de confusión se puede identificar en qué números se ha cometido más error. Por ejemplo, como era de esperar, los errores en el 4 suelen ser predichos como 9.

```
[43]: plt.figure(figsize=(8,7))

sns.heatmap(pd.crosstab(y_test_pred, y_test, normalize='index'),
            cmap='Blues', annot=True, fmt='.2f', square=True, vmax=1, vmin=0, cbar=False)

plt.xlabel('Real')
plt.ylabel('Prediction')
```

```
[43]: Text(83.94, 0.5, 'Prediction')
```



Reducción de dimensionalidad

La gran mayoría de algoritmos de reducción de dimensionalidad como PCA y especialmente tSNE no admiten un número de variables tan alto como las 784 iniciales que se tenían. En este caso se va a utilizar la salida codificada como entrada a los algoritmos de reducción de dimensionalidad para realizar una representación bidimensional de la distribución de los dígitos.

PCA

Con el algoritmo de PCA se hace una reducción identificando las componentes principales mediante combinaciones lineales de las otras variables. Para representar los dígitos en 2 dimensiones se obtienen las dos primeras componentes.

```
[118]: pca = PCA(n_components=2)
```

```
[119]: X_train_enc_pca = pca.fit_transform(X_train_enc)
```

Con las dos primeras componentes se representan el 39% de la varianza de los datos. No se espera ver grandes resultados en la representación 2D debido a que gran parte de la varianza queda fuera.


```
[122]: pca.explained_variance_ratio_
```

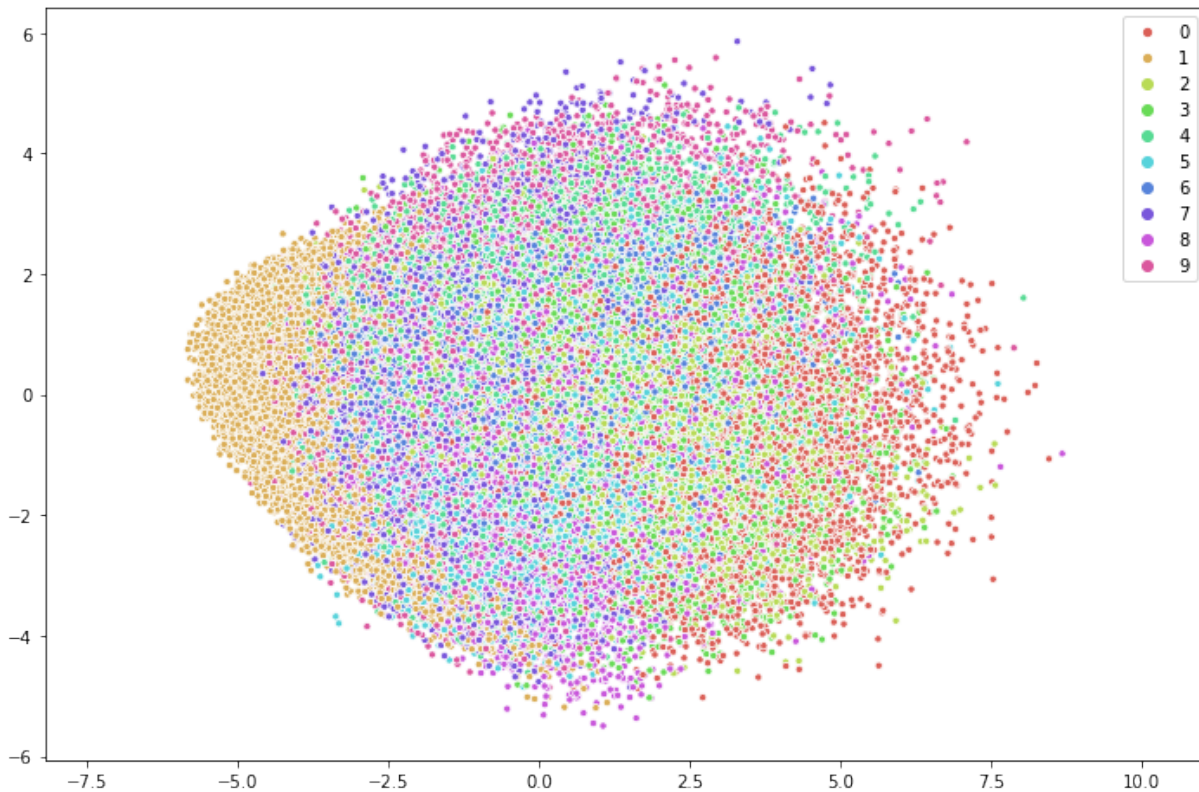
```
[122]: array([0.24032657, 0.1349783 ], dtype=float32)
```

```
[120]: X_train_enc_pca[:,1]
```

```
[120]: array([ 0.6406583 , -0.24814816,  0.6898029 , ..., -2.3894398 ,  
          -0.5899435 , -0.08575194], dtype=float32)
```

Aún así se puede ver como con la primera componente se diferencian claramente los 1s contra el resto de números.

```
[72]: plt.figure(figsize=(12,8))  
      sns.scatterplot(x=X_train_enc_pca[:,0], y=X_train_enc_pca[:,1], hue=y_train, size=1,  
                     palette=sns.color_palette("hls", 10))  
      plt.axis('equal')  
      pass
```

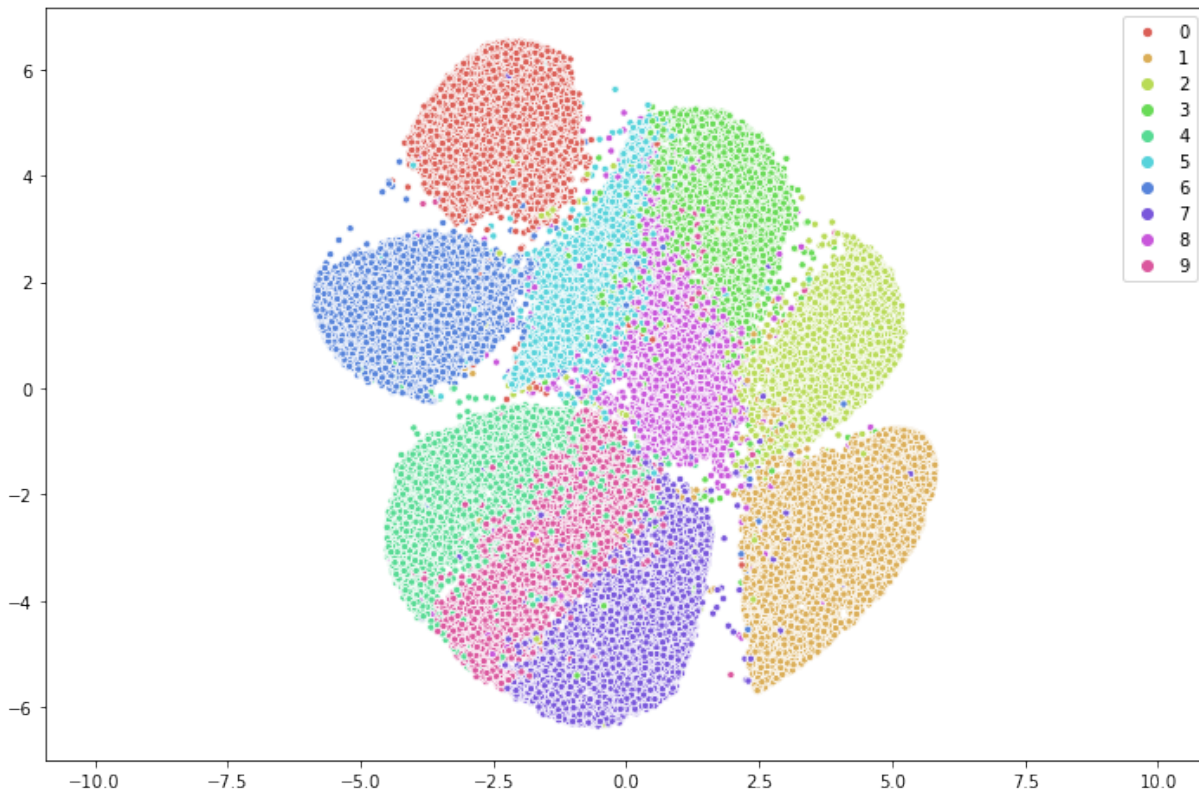


tSNE

Este algoritmo está diseñado especialmente para visualización en 2 o 3 dimensiones de conjuntos de datos y se observa que los resultados de la agrupación son mucho mejores que PCA.

```
[ ]: tsne = TSNE(n_components=2, verbose=1, perplexity=40, n_iter=300)  
      X_train_enc_tsne = tsne.fit_transform(X_train_enc)
```

```
[57]: plt.figure(figsize=(12,8))
      sns.scatterplot(x=X_train_enc_tsne[:,0], y=X_train_enc_tsne[:,1], hue=y_train, size=1,
                      palette=sns.color_palette("hls", 10))
      plt.axis('equal')
      pass
```



Clustering: KMeans

Por último se evalúan los resultados sobre un modelo no supervisado. Este caso es una simulación mejor el caso de las imágenes histológicas ya que de estas se tienen etiquetas a nivel imagen completa pero no a nivel *patch*.

La idea detrás de este proceso es que, mediante un algoritmo de clustering, se agrupen las distintas imágenes según su forma. Si el resultado de el clustering es bueno cada grupo deberá tener un tipo de etiqueta (dígito en ese caso) predominante.

Para esto se utilizará un KMeans sobre la salida bidimensional del TSNE para agilizar el entrenamiento y mejorar los resultados.

```
[58]: K = 20
      kmeans = KMeans(n_clusters=K)
```

```
[59]: y_train_clust = kmeans.fit_predict(X_train_enc_tsne)
```

La siguiente imagen muestra el porcentaje de cada dígito en cada grupo generado.

```
[60]: freq_table = pd.crosstab(y_train, y_train_clust, rownames='N', colnames='C',
                               <-normalize='columns')
```

```
[61]: plt.figure(figsize=(14,7))

sns.heatmap(freq_table, cmap='Blues', square=True,
            annot=True, fmt='.2f', vmax=1, vmin=0, cbar=False)

plt.xlabel('Cluster', fontsize=12, color='navy')
plt.ylabel('Number', fontsize=12, color='navy')

plt.vlines(list(range(1, K)) ,0, 10, lw=3, color='navy')
```

[61]: <matplotlib.collections.LineCollection at 0x1c4b18a898>



Una manera de evaluar esta agrupación, ya que se tienen todas las muestras debidamente etiquetadas, es utilizar métricas de impureza como GINI o la entropía. Los cluster con menor GINI serán aquellos donde haya menos ruido de otros dígitos. El clúster 4, es el que mayor GINI tiene, era de esperar ya que su dígito más contenido representa únicamente el 50% de las muestras.

Otro dato interesante sería la diferencia entre el grupo 10 y el 11, ambos tienen un porcentaje alto de dígitos 9s pero mientras en uno el siguiente más representado es el 4 otro es el 7. Se observan las muestras de imágenes se ve que el cluster 11 tiene aquellos 9s en los que el palo está inclinado, es decir, similar al del número 7. Similar al caso de los grupos 16 y 19, con 1s inclinados y rectos.

Esto prueba que, no sólo ha sido posible separar automáticamente según el dígito si no que se han separado también según su forma de escritura.

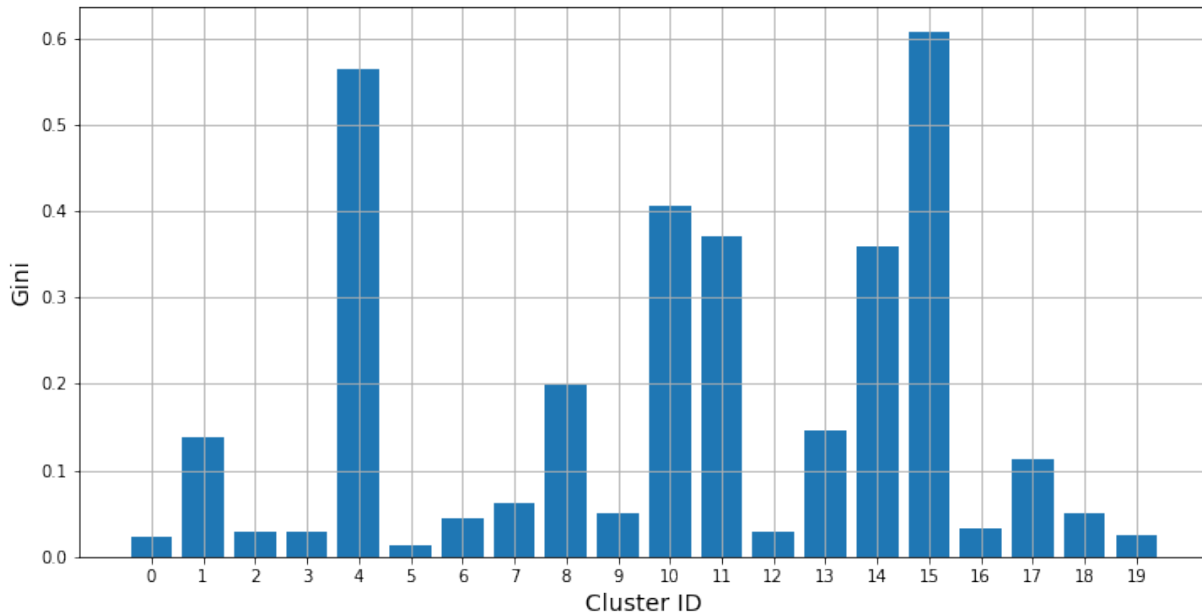
```
[62]: ginis = metrics.cluster_gini(y_train_clust, y_train)
```

```
[63]: plt.figure(figsize=(12,6))

plt.bar(ginis.keys(), ginis.values())
plt.xticks(list(ginis.keys()))
```

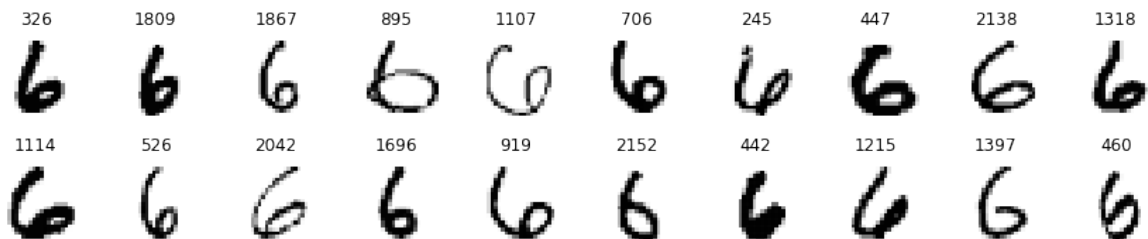
```
plt.xlabel('Cluster ID', fontsize=14)
plt.ylabel('Gini', fontsize=14)

plt.grid()
pass
```



Por último se muestran unas pocas imágenes de cada clúster para comprobar que tienen sentido.

```
[67]: for cluster in range(K):
        print('\033[1m Cluster {}'.format(cluster))
        plot_sample_imgs(X_train[y_train_clust == cluster], n_rows=2, n_cols=10, size=1.5,
        color=False)
        plt.show()
```



811	582	2267	2172	1592	1441	1581	1956	1816	1922
7	7	7	7	7	7	7	7	7	7
1693	1689	1190	1489	902	2133	1235	2118	461	2146
7	7	7	7	7	7	7	7	7	7

1684	1941	1705	1778	1419	1623	2238	1572	1852	1716
2	2	2	2	2	2	2	2	2	2
833	2168	1389	87	471	839	974	1642	2292	2010
2	2	2	2	2	2	2	2	2	2

376	867	2189	2224	1467	1050	1393	1685	2477	1895
7	7	7	7	7	7	7	7	7	7
2283	996	220	351	1677	2364	814	1499	376	686
7	7	7	7	7	7	7	7	7	7

2355	754	1252	1150	1656	1560	2107	1885	939	1424
8	8	3	3	8	3	3	8	3	3
149	656	1456	1143	1350	2329	1925	81	2562	1133
3	8	3	8	8	8	3	8	5	5

18	680	233	134	438	731	1139	744	157	2000
1015	2110	1157	288	1140	1668	1666	1862	2135	188

1517	254	942	789	2171	1258	183	1597	2161	1886
1188	1821	1927	1686	1522	368	217	2169	1428	1813

1461	778	450	125	326	1390	362	1907	1416	997
1023	1577	13	943	1057	1325	1617	1356	62	411

2278	297	361	2427	224	1267	888	702	1890	502
28	400	2557	2586	1983	1460	1772	1020	2144	58

260	304	2276	965	3103	1006	36	1056	1005	3003
3	3	3	3	3	3	3	3	5	3
2218	2060	1596	940	3380	3051	3269	2897	2744	2322
3	3	3	3	3	3	3	3	3	3

238	680	789	1208	2557	1313	474	970	2825	593
9	9	9	9	8	9	9	9	9	9
1848	1240	2215	1806	998	2336	389	52	2136	449
4	9	9	9	8	9	9	5	9	9

604	2786	2817	1892	541	634	588	76	13	2708
9	9	9	9	9	9	7	9	7	7
1827	1822	2242	1501	947	311	987	1155	1734	1074
9	7	9	9	9	7	9	9	9	4





282	1432	229	542	1244	1245	486	344	1173	990
0	0	0	0	0	0	0	0	0	0
150	779	341	1377	1150	441	158	1330	457	717
0	0	0	0	0	0	0	0	0	0







1877	2530	123	2185	1040	1980	434	1187	853	769
2	2	2	2	1	2	2	2	2	2
291	1722	1153	282	1564	1471	2388	1618	993	2165
2	2	2	2	2	2	2	2	2	2







1760	1421	166	1359	2420	1802	1607	2013	1548	2276
4	4	4	4	4	4	4	4	4	4
0	800	695	2449	2541	173	1193	2185	899	1526
4	4	4	5	4	4	4	4	4	4

2005	1488	862	995	2849	1967	1039	2513	2038	2321
5	5	5	5	5	5	0	5	0	0
208	817	1990	129	1242	2687	119	397	2136	1914
3	5	5	5	5	0	5	0	5	5

233	1187	385	114	1348	541	0	1438	333	447
/	/	/	/	/	/	/	/	/	/
329	472	1310	1369	566	1108	699	238	1349	362
/	/	/	/	/	/	/	/	/	/

2501	2278	2079	1519	1735	1725	182	448	1494	215
									
1472	2344	2735	2528	1940	2810	3422	3137	2602	669
									

1830	2288	2602	1395	2410	1931	1055	1093	263	2026
									
649	1281	2574	1413	1734	542	929	1201	2127	1420
									

1681	1399	144	217	1494	1670	850	470	1769	1583
									
1768	1075	1027	69	1773	1005	1167	641	234	525
