

# PEC1\_GarciaRuiz\_Ricardo

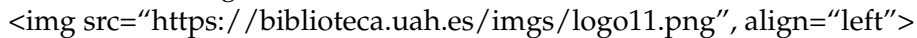
February 15, 2019

## Table of Contents

- 1 Introducción al dataset Fashion-Mnist
  - 1.1 Jugando con la moda MNIST
  - 1.2 Entrar en la moda MNIST
- 2 Métodos supervisados
  - 2.1 Carga de datos
    - 2.1.1 Descripción del conjunto de datos
    - 2.1.2 Definición del conjunto de entrenamiento y test
    - 2.1.3 Utilización de algoritmo PCA para reducción dimensión del dataset
  - 2.2  $k$  vecinos más cercanos
    - 2.2.1 Cálculo hiperparámetros óptimos
    - 2.2.2 Entrenamiento de un modelo  $k$ -nn
    - 2.2.3 La matriz de confusión de PCA
  - 2.3 Support Vector Machines
    - 2.3.1 Cálculo del valor óptimo de hiperparámetros
    - 2.3.2 Modelo SVM
    - 2.3.3 La matriz de confusión de SVM
  - 2.4 Redes neuronales
    - 2.4.1 Número óptimo de neuronas
    - 2.4.2 Número óptimo de neuronas (conjunto completo)
    - 2.4.3 La matriz de confusión del modelo
  - 2.5 Optimización de métricas
    - 2.5.1 Nueva función personalizada de coste
    - 2.5.2 Aplicación al modelo de la nueva función de coste
- 3 Combinación de clasificadores
  - 3.1 Combinación paralela de clasificadores base similares
    - 3.1.1 Bagging
      - 3.1.1.1 Random forest simple
      - 3.1.1.2 Out-of-bag
      - 3.1.1.3 Probabilidad por clase
      - 3.1.1.4 Importancia de las variables
      - 3.1.1.5 Número de clasificadores
      - 3.1.1.6 Volumen de datos
    - 3.1.2 Boosting
  - 3.2 Combinación secuencial de clasificadores base diferentes
    - 3.2.1 Stacking
    - 3.2.2 Cascading

3.2.2.1 Cascading simple

3.2.2.2 Cascading con variables adicionales



EN26 - HERRAMIENTAS DE ANÁLISIS ü PEC1

ENTORNOS DE ANÁLISIS DE DATOS (PYTHON)

2018-2019 ü Máster universitario en Ciencia de datos (Data science)

## 1 Introducción al dataset Fashion-Mnist

### 1.1 Jugando con la moda MNIST

Recientemente, los investigadores de Zalando, una empresa de comercio electrónico, presentaron a Fashion MNIST como un reemplazo directo del conjunto de datos original de MNIST. Al igual que MNIST, Fashion MNIST consiste en un conjunto de entrenamiento que consiste en 60.000 ejemplos pertenecientes a 10 clases diferentes y un conjunto de prueba de 10.000 ejemplos. Cada ejemplo de entrenamiento es una imagen en escala de grises, de tamaño 28x28.

### 1.2 Entrar en la moda MNIST

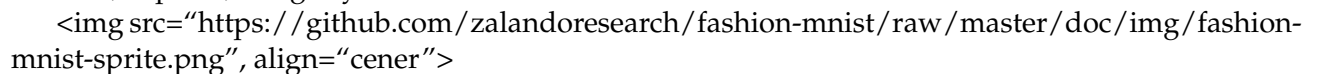
Al publicar este conjunto de datos, los investigadores de Zalando hicieron las siguientes observaciones en MNIST:

MNIST es demasiado fácil. Las redes convolucionales pueden alcanzar el 99.7% en MNIST. Los algoritmos clásicos de aprendizaje automático también pueden alcanzar el 97% fácilmente. Echa un vistazo a nuestro punto de referencia de lado a lado para Fashion-MNIST vs. MNIST, y lee “La mayoría de los pares de dígitos MNIST se pueden distinguir bastante bien con solo un píxel”.

MNIST está sobreutilizado. En un hilo de Twitter de abril de 2017, el investigador científico de Google Brain y experto en aprendizaje profundo, Ian Goodfellow, hizo un llamamiento a los investigadores a alejarse del conjunto MNIST.

MNIST no puede representar tareas CV modernas, como se señala en este hilo de Twitter de abril de 2017, el experto en aprendizaje profundo/autor de Keras, François Chollet.

Los investigadores introdujeron a Fashion-MNIST como un reemplazo del conjunto de datos MNIST. El nuevo conjunto de datos contiene imágenes de diversos artículos de ropa, como camisas, zapatos, abrigos y otros artículos de moda.



El dataset Fashion-MNIST comparte la misma estructura dividida de train-test como MNIST. Mientras que en el caso del conjunto de datos MNIST las etiquetas de clase eran dígitos del 0-9, Las etiquetas de clase para Fashion-MNIST son las siguientes:

Label

Description

0

T-shirt/top
1
Trouser
2
Pullover
3
Dress
4
Coat
5
Sandal
6
Shirt
7
Sneaker
8
Bag
9
Ankle boot

</table></p>

## 2 Métodos supervisados

En esta primera parte de la práctica vamos a trabajar en diferentes métodos supervisados aplicados sobre el conjunto de datos [Fashion MNIST](#) y trataremos de optimizar diferentes métricas.

- Carga de datos
- $k$  vecinos más cercanos
- Support vector machines
- Redes neuronales
- Optimización de métricas

### 2.1 Carga de datos

#### 2.1.1 Descripción del conjunto de datos

El conjunto de datos Fashion MNIST proporcionado por Zalando consta de 70.000 imágenes con 10 clases diferentes de ropa repartidas uniformemente. No obstante, para esta práctica utilizaremos únicamente un subconjunto de 5.000 imágenes que consiste en 1.000 imágenes de 5 clases diferentes.

Las imágenes tienen una resolución de 28x28 píxeles en escala de grises, por lo que se pueden representar utilizando un vector de 784 posiciones.

El siguiente código cargará las 5.000 imágenes en la variable `images` y las correspondientes etiquetas (en forma numérica) en la variable `labels`. Podemos comprobar que la carga ha sido correcta obteniendo las dimensiones de estas dos variables.

```
In [1]: import pickle
import numpy as np
```

```

with open("data.pickle", "rb") as f:
    data = pickle.load(f)

images = data["images"]
labels = data["labels"]
n_classes = 5
labels_text = ["T-shirt", "Trouser", "Pullover", "Dress", "Coat"]

print("Dimensiones del vector de imágenes: {}".format(images.shape))
print("Dimensiones del vector de etiquetas: {}".format(labels.shape))

```

Dimensiones del vector de imágenes: (5000, 784)  
Dimensiones del vector de etiquetas: (5000,)

Con el siguiente código podemos ver un ejemplo de imagen de cada una de las clases. Para ello reajustamos el vector de 784 dimensiones que representa cada imagen en una matriz de tamaño 28x28 y la transponemos para mostrarla:

```

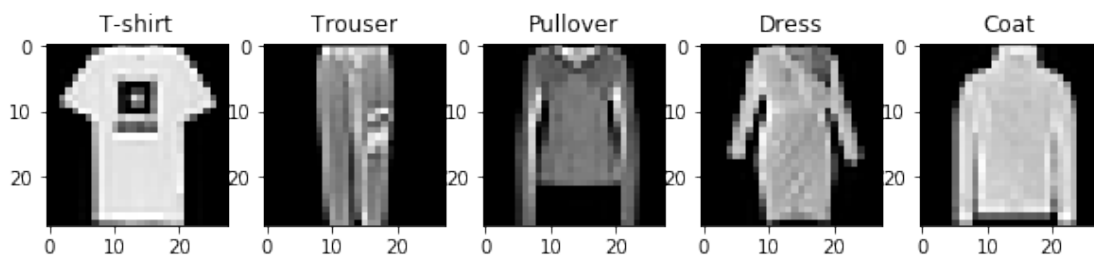
In [2]: import matplotlib.pyplot as plt
        %matplotlib inline

fig, ax = plt.subplots(1, n_classes, figsize=(10,10))

idxs = [np.where(labels == i)[0] for i in range(n_classes)]

for i in range(n_classes):
    k = np.random.choice(idxs[i])
    ax[i].imshow(images[k].reshape(28, 28), cmap="gray")
    ax[i].set_title("{}".format(labels_text[i]))

```



### 2.1.2 Definición del conjunto de entrenamiento y test

De las 5.000 imágenes distintas de subset de datos utilizaremos 4.000 imágenes para entrenar los diferentes modelos y 1.000 imágenes para validar los resultados. Con el siguiente código separamos los datos que hemos cargado anteriormente en dos conjuntos, train y test, de forma estratificada, es decir, en cada uno de los conjuntos las clases aparecen en la misma proporción que en el conjunto original.

### 2.1.3 Utilización de algoritmo PCA para reducción dimensión del dataset

En lugar de trabajar directamente con un vector de 784 dimensiones para cada imagen aplicaremos primero el algoritmo PCA para reducir la dimensión de los ejemplos a 100. El proceso de entrenamiento de PCA lo hacemos con las imágenes de train y luego lo aplicamos también sobre las imágenes de test, de forma que no utilizamos ninguna información de las imágenes en el conjunto de test para entrenar los modelos.

```
In [3]: from sklearn.model_selection import train_test_split
        from sklearn.decomposition import PCA

        X_train, X_test, y_train, y_test = train_test_split(images, labels, test_size=0.2, random_state=2017)

        pca = PCA(n_components=100, random_state=2017)
        pca_fit = pca.fit(X_train)
        X_train_pca = pca_fit.transform(X_train)
        X_test_pca = pca_fit.transform(X_test)

        def proporcion_etiquetas(y):
            _, count = np.unique(y, return_counts=True)
            return np.true_divide(count, y.shape[0])

        print("Número de imágenes para entrenar: {}".format(X_train_pca.shape[0]))
        print("Número de imágenes para test: {}".format(X_test_pca.shape[0]))

        print("Proporción de las etiquetas en el conjunto original: {}".format(proporcion_etiquetas(y_train)))
        print("Proporción de las etiquetas en el conjunto de entrenamiento: {}".format(proporcion_etiquetas(y_train)))
        print("Proporción de las etiquetas en el conjunto de test: {}".format(proporcion_etiquetas(y_test)))
```

Número de imágenes para entrenar: 4000

Número de imágenes para test: 1000

Proporción de las etiquetas en el conjunto original: [0.2 0.2 0.2 0.2 0.2]

Proporción de las etiquetas en el conjunto de entrenamiento: [0.2 0.2 0.2 0.2 0.2]

Proporción de las etiquetas en el conjunto de test: [0.2 0.2 0.2 0.2 0.2]

## 2.2 $k$ vecinos más cercanos

El primer algoritmo que utilizaremos para clasificar las imágenes de ropa es el  $k$ -nn. En este paso del análisis del dataset ajustaremos dos hiperparámetros del algoritmo:

$k$ : el número de vecinos que se consideran para clasificar un nuevo ejemplo. Probaremos con todos los valores entre 1 y 10.

pesos: importancia que se da a cada uno de los vecinos considerados. En este caso probaremos dos opciones:

<ol>

<li>pesos uniformes, donde todos los vecinos se consideran igual</li>

<li>pesos según distancia, donde los vecinos más cercanos tienen más peso en la clasificación</li>

</ol>

Para decidir cuáles son los hiperparámetros óptimos utilizaremos una búsqueda de rejilla (grid search), es decir, entrenaremos un modelo para cada combinación de hiperparámetros posible y la evaluaremos utilizando validación cruzada (cross validation) con 4 particiones estratificadas. Posteriormente escogeremos la combinación de hiperparámetros que mejor resultados haya dado.

### 2.2.1 Cálculo hiperparámetros óptimos

Implementación: Calcularemos el valor óptimo de los hiperparámetros  $k$  y pesos. Vamos a utilizar los módulos GridSearchCV y KNeighborsClassifier de sklearn.

```
In [4]: from sklearn.model_selection import GridSearchCV
        from sklearn.neighbors import KNeighborsClassifier
        from time import time

        clf = KNeighborsClassifier()

        param_grid = {"n_neighbors": range(1, 11), "weights": ["uniform", "distance"]}

        grid_search = GridSearchCV(clf, param_grid=param_grid, cv=4)

        start = time()
        grid_search.fit(X_train_pca, y_train)
        end = time()

        print("La búsqueda llevó {} segundos".format(end - start))

        means = grid_search.cv_results_["mean_test_score"]
        stds = grid_search.cv_results_["std_test_score"]
        params = grid_search.cv_results_['params']
        ranks = grid_search.cv_results_['rank_test_score']

        for rank, mean, std, pms in zip(ranks, means, stds, params):
            print("{} Precisión media: {:.2f} +/- {:.2f} con parámetros {}".format(rank, mean, std, pms))
```

La búsqueda llevó 69.53597712516785 segundos

17) Precisión media: 83.55 +/- 1.32 con parámetros {'weights': 'uniform', 'n\_neighbors': 1}  
17) Precisión media: 83.55 +/- 1.32 con parámetros {'weights': 'distance', 'n\_neighbors': 1}  
20) Precisión media: 82.53 +/- 1.10 con parámetros {'weights': 'uniform', 'n\_neighbors': 2}  
17) Precisión media: 83.55 +/- 1.32 con parámetros {'weights': 'distance', 'n\_neighbors': 2}  
15) Precisión media: 85.20 +/- 0.78 con parámetros {'weights': 'uniform', 'n\_neighbors': 3}  
13) Precisión media: 85.38 +/- 0.66 con parámetros {'weights': 'distance', 'n\_neighbors': 3}  
16) Precisión media: 84.92 +/- 0.78 con parámetros {'weights': 'uniform', 'n\_neighbors': 4}  
10) Precisión media: 85.58 +/- 0.61 con parámetros {'weights': 'distance', 'n\_neighbors': 4}  
2) Precisión media: 85.95 +/- 0.86 con parámetros {'weights': 'uniform', 'n\_neighbors': 5}  
4) Precisión media: 85.90 +/- 0.87 con parámetros {'weights': 'distance', 'n\_neighbors': 5}  
12) Precisión media: 85.40 +/- 0.91 con parámetros {'weights': 'uniform', 'n\_neighbors': 6}  
1) Precisión media: 86.10 +/- 0.72 con parámetros {'weights': 'distance', 'n\_neighbors': 6}  
8) Precisión media: 85.70 +/- 0.63 con parámetros {'weights': 'uniform', 'n\_neighbors': 7}

2) Precisión media: 85.95 +/- 0.82 con parámetros {'weights': 'distance', 'n\_neighbors': 7}  
 11) Precisión media: 85.55 +/- 1.09 con parámetros {'weights': 'uniform', 'n\_neighbors': 8}  
 4) Precisión media: 85.90 +/- 0.79 con parámetros {'weights': 'distance', 'n\_neighbors': 8}  
 13) Precisión media: 85.38 +/- 1.13 con parámetros {'weights': 'uniform', 'n\_neighbors': 9}  
 6) Precisión media: 85.82 +/- 1.24 con parámetros {'weights': 'distance', 'n\_neighbors': 9}  
 9) Precisión media: 85.65 +/- 1.01 con parámetros {'weights': 'uniform', 'n\_neighbors': 10}  
 6) Precisión media: 85.82 +/- 0.93 con parámetros {'weights': 'distance', 'n\_neighbors': 10}

Análisis: A partir de los resultados anteriores podemos considerar responder de manera sencilla las siguientes cuestiones:

¿Qué parámetros han dado mejores resultados?

<li>¿Qué variación hay entre las diferentes combinaciones de parámetros?</li>

<li>¿Es significativa la variación entre las diferentes combinaciones?</li>

<li>¿Hay algún parámetro que influya más que el otro?</li>

<li>¿Era de esperar?</li>

Respuesta:

<ul>

La mejor solución es con  $k = 6$  y los pesos calculados con la distancia.

La máxima diferencia entre las precisiones medias es de unos 3.5 puntos porcentuales, con desviaciones estándar del orden de 1 punto porcentual podemos afirmar que hay opciones claramente mejores que otras.

Para  $k = 1$  los pesos no importan, ya que siempre se clasifican los nuevos ejemplos con la clase del vecino más cercano.

Parece que la precisión depende más de  $k$  que del tipo de los pesos, aún así es interesante notar que en prácticamente todos los casos es mejor utilizar pesos con distancias en vez de uniformes, lo cual es lógico porque al utilizar las distancias el algoritmo tiene más información.

## 2.2.2 Entrenamiento de un modelo $k$ -nn

Implementación: Ahora procederemos a realizar un entrenamiento de un modelo  $k$ -nn con los valores de los hiperparámetros óptimos utilizando todo el conjunto  $X_{train\_pca}$  y mostraremos la precisión de la predicción del modelo en el conjunto  $X_{test\_pca}$ .

La codificación siguiente realiza el entrenamiento  $k$ -nn, y finalmente muestra la precisión del conjunto de test.

```
In [5]: print("Valor óptimo para k: {}".format(grid_search.best_params_["n_neighbors"]))
        print("Valor óptimo para weights: {}".format(grid_search.best_params_["weights"]))

        clf = KNeighborsClassifier(n_neighbors=grid_search.best_params_["n_neighbors"], weights=
        clf.fit(X_train_pca, y_train)

        preds = clf.predict(X_test_pca)

        accuracy = np.true_divide(np.sum(preds == y_test), preds.shape[0])*100
        print("Precisión en el conjunto de test: {:.2f}%".format(accuracy))
```

Valor óptimo para k: 6  
Valor óptimo para weights: distance  
Precisión en el conjunto de test: 86.60%

### 2.2.3 La matriz de confusión de PCA

Implementación: Finalmente mostramos la matriz de confusión del modelo y algunas imágenes que el modelo ha clasificado incorrectamente junto con la etiqueta asignada por el modelo y la etiqueta original.

```
In [6]: import itertools
        from sklearn.metrics import confusion_matrix

        cnf_matrix = confusion_matrix(y_test, preds)

        def plot_confusion_matrix(cm, classes):
            cm = cm.astype('float') / cm.sum(axis=1)[:, np.newaxis]

            cmap=plt.cm.Blues

            plt.imshow(cm, interpolation='nearest', cmap=cmap)
            plt.colorbar()
            tick_marks = np.arange(len(classes))
            plt.xticks(tick_marks, classes, rotation=45)
            plt.yticks(tick_marks, classes)

            thresh = cm.max() / 2.
            for i, j in itertools.product(range(cm.shape[0]), range(cm.shape[1])):
                plt.text(j, i, format(cm[i, j], ".2f"),
                        horizontalalignment="center",
                        color="white" if cm[i, j] > thresh else "black")

            plt.tight_layout()
            plt.ylabel('True label')
            plt.xlabel('Predicted label')

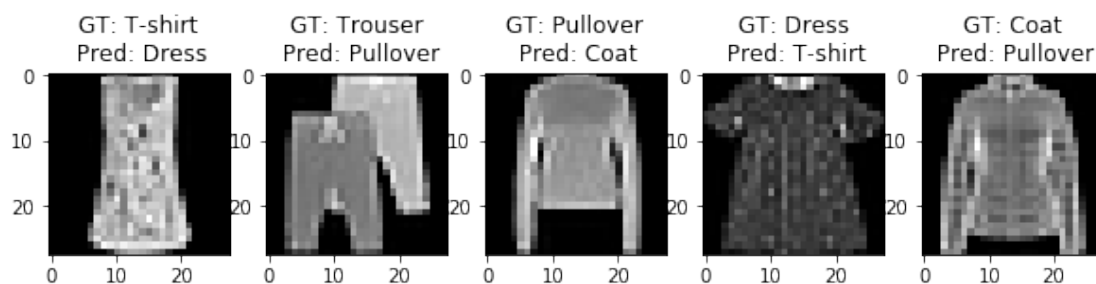
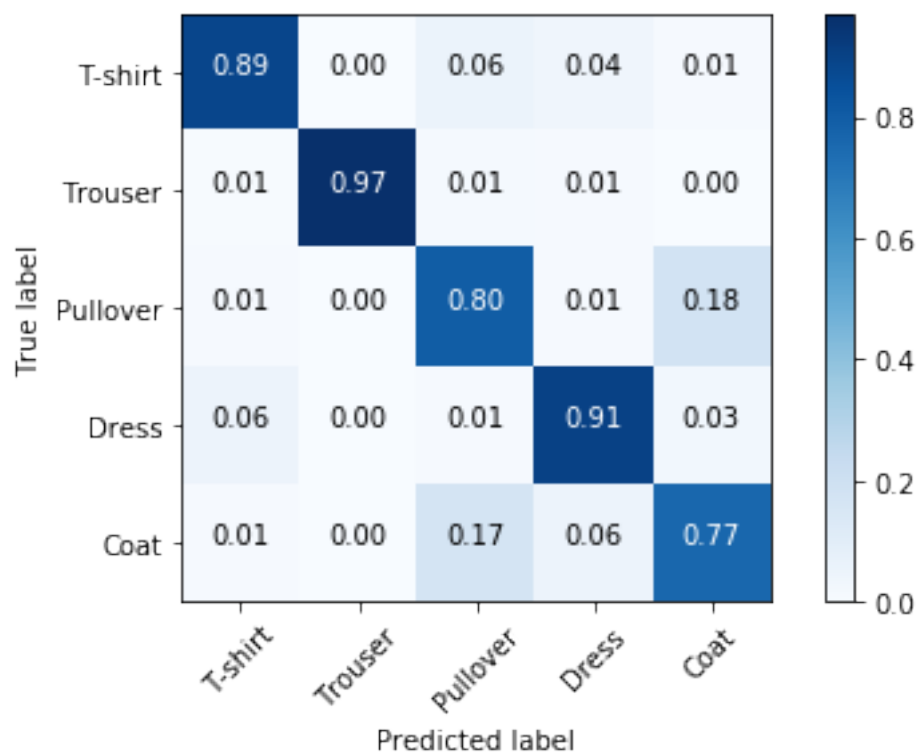
        plot_confusion_matrix(cnf_matrix, classes=labels_text)

        fig, ax = plt.subplots(1, n_classes, figsize=(10,10))

        idxs = [np.where((y_test == i) & (preds != i))[0] for i in range(n_classes)]

        for i in range(n_classes):
            k = np.random.choice(idxs[i])
            ax[i].imshow(X_test[k].reshape(28, 28), cmap="gray")
            ax[i].set_title("GT: {} \n Pred: {}".format(labels_text[y_test[k]], labels_text[preds[k]]))
```





Análisis: A la vista de los resultados obtenidos y de la matriz de confusión podemos plantearnos responder a las cuestiones:

¿Cómo son los errores?

¿Parecen razonables?

Respuesta:

<ul>

Es interesante notar que cuando el modelo predice “Trousers” casi nunca se equivoca: 97% acierto.

La confusión más grande aparece entre “Pullover” y “Coat”: 18% Pullover y 17% Coat.

Viendo algunos ejemplos parece razonable que el modelo confunda algunas imágenes de “Pullover” con “Coat” y viceversa. También las confusiones entre “T-shirt” y “Pullover” o “Dress” y “Coat” podrían justificarse.

## 2.3 Support Vector Machines

Como continuación de la práctica, en esta segunda parte clasificaremos las imágenes de ropa utilizando el algoritmo SVM con el kernel radial. En este caso, en lugar de utilizar una búsqueda de rejilla para ajustar los hiperparámetros del algoritmo utilizaremos una búsqueda aleatoria, es decir, probaremos combinaciones de parámetros al azar. Los hiperparámetros a optimizar son:

C: el valor de penalización de los errores en la clasificación. Marca el compromiso entre obtener el hiperplano con el mayor margen posible y clasificar el máximo número de ejemplos correctamente. Probaremos valores aleatorios distribuidos uniformemente entre 1 y 500.

gamma: coeficiente que multiplica la distancia entre dos puntos en el kernel radial. Probaremos valores aleatorios distribuidos uniformemente entre 0.001 y 0.1

Igual que en el caso anterior, para validar el rendimiento del algoritmo con cada combinación de hiperparámetros utilizaremos validación cruzada (cross-validation) con 4 particiones estratificadas.

### 2.3.1 Cálculo del valor óptimo de hiperparámetros

Implementación: En este primer paso calculamos el valor óptimo de los hiperparámetros C y gamma utilizando 10 combinaciones de parámetros elegidas al azar. Para ello utilizaremos los módulos RandomizedSearchCV y svm de sklearn, así como el módulo uniform de scipy.

```
In [7]: from sklearn.model_selection import RandomizedSearchCV
        from sklearn import svm
        from scipy.stats import uniform as sp_rand
        from time import time

        clf = svm.SVC()

        param_dist = {"C": sp_rand(loc=1, scale=500), "gamma": sp_rand(loc=0.001, scale=0.1)}

        n_iter_search = 10
        random_search = RandomizedSearchCV(clf, param_distributions=param_dist, n_iter=n_iter_s

        start = time()
        random_search.fit(X_train_pca, y_train)
        end = time()

        print("La búsqueda llevó {} segundos".format(end - start))

        means = random_search.cv_results_["mean_test_score"]
        stds = random_search.cv_results_["std_test_score"]
        params = random_search.cv_results_['params']
        ranks = random_search.cv_results_['rank_test_score']
```

```
for rank, mean, std, pms in zip(ranks, means, stds, params):
    print("{} Precisión media: {:.2f} +/- {:.2f} con parámetros {}".format(rank, mean,
```

La búsqueda llevó 85.28187775611877 segundos

```
3) Precisión media: 88.25 +/- 1.02 con parámetros {'gamma': 0.028326801618082466, 'C': 373.639
3) Precisión media: 88.25 +/- 0.96 con parámetros {'gamma': 0.028951695071907035, 'C': 498.184
1) Precisión media: 88.33 +/- 0.83 con parámetros {'gamma': 0.039490234193371644, 'C': 418.641
9) Precisión media: 86.90 +/- 1.15 con parámetros {'gamma': 0.09663101891123066, 'C': 29.36027
5) Precisión media: 87.55 +/- 0.98 con parámetros {'gamma': 0.0778384471203179, 'C': 14.344899
7) Precisión media: 87.20 +/- 1.07 con parámetros {'gamma': 0.08844792047132262, 'C': 253.5427
8) Precisión media: 86.98 +/- 1.11 con parámetros {'gamma': 0.09390054773874328, 'C': 329.7318
6) Precisión media: 87.48 +/- 0.94 con parámetros {'gamma': 0.07708254047265004, 'C': 339.7271
1) Precisión media: 88.33 +/- 0.83 con parámetros {'gamma': 0.039441400938736826, 'C': 481.532
10) Precisión media: 86.80 +/- 1.20 con parámetros {'gamma': 0.006892306444705908, 'C': 100.53
```

Análisis: A la vista de los resultados obtenidos con la codificación previa, podemos proceder a responder las siguientes cuestiones:

¿Qué parámetros han dado mejores resultados?

¿Qué variación hay entre las diferentes combinaciones de parámetros?

¿Es significativa la variación entre las diferentes combinaciones?

¿Hay algún parámetro que influya más que el otro?

Respuesta:

<ul>

La mejor combinación se da con  $C = 245$  y  $\gamma = 0.034$ .

Las diferencias en la precisión no son demasiado grandes, por lo que, teniendo en cuenta la desviación estandar es difícil afirmar que haya combinaciones claramente mejores que otras.

Las mejores soluciones se dan con  $\gamma$  del orden de 0.03-0.04. Se puede ver que en las mejores soluciones el valor de  $C$  es bastante variable, por lo que podemos deducir que el parámetro  $\gamma$  tiene mucho más peso, lo cual es típico al utilizar el kernel radial.

## 2.3.2 Modelo SVM

Implementación: A continuación vamos a proceder a realizar un entrenamiento de un modelo SVM con los valores de los hiperparámetros óptimos utilizando todo el conjunto  $X_{train\_pca}$  y mostraremos la precisión de la predicción del modelo en el conjunto  $X_{test\_pca}$ .

```
In [8]: print("Valor óptimo para C: {}".format(random_search.best_params_["C"]))
        print("Valor óptimo para gamma: {}".format(random_search.best_params_["gamma"]))

clf = svm.SVC(C=random_search.best_params_["C"], gamma=random_search.best_params_["gamma"])
clf.fit(X_train_pca, y_train)

preds = clf.predict(X_test_pca)
```

```
accuracy = np.true_divide(np.sum(preds == y_test), preds.shape[0])*100
print("Precisión en el conjunto de test: {:.2f}%".format(accuracy))
```

Valor óptimo para C: 418.641252041766

Valor óptimo para gamma: 0.039490234193371644

Precisión en el conjunto de test: 87.50%

### 2.3.3 La matriz de confusión de SVM

Implementación: Finalmente mostraremos la matriz de confusión del modelo y algunas imágenes que el modelo ha clasificado incorrectamente junto con la etiqueta asignada por el modelo y la etiqueta original.

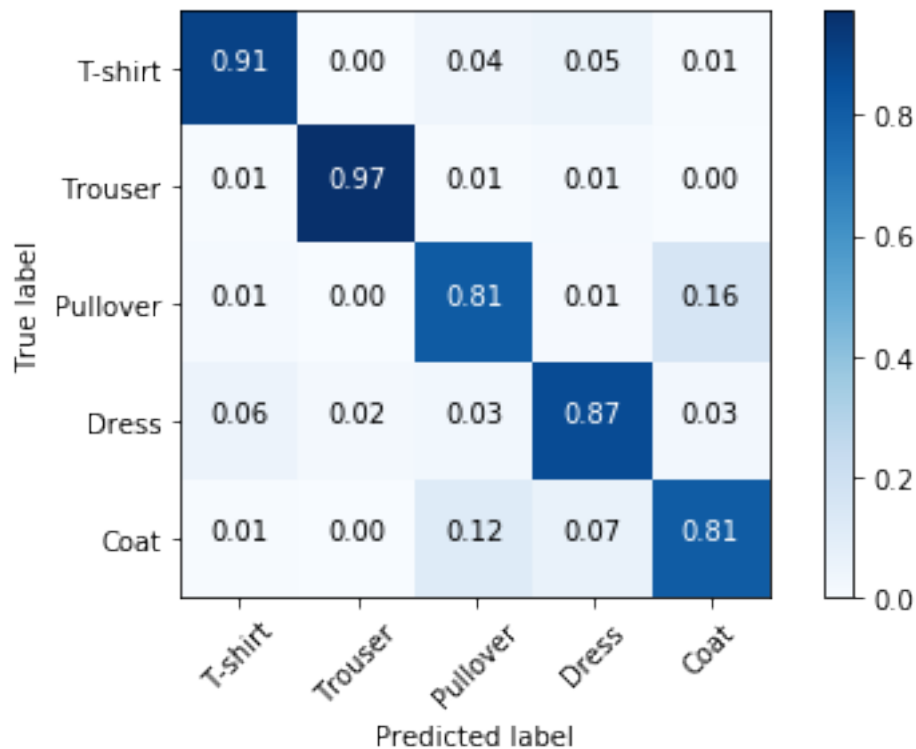
```
In [9]: cnf_matrix = confusion_matrix(y_test, preds)

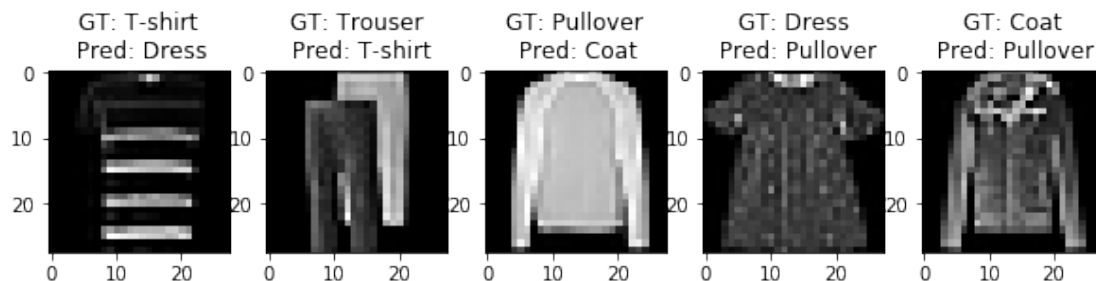
plot_confusion_matrix(cnf_matrix, classes=labels_text)

fig, ax = plt.subplots(1, n_classes, figsize=(10,10))

idxs = [np.where((y_test == i) & (preds != i))[0] for i in range(n_classes)]

for i in range(n_classes):
    k = np.random.choice(idxs[i])
    ax[i].imshow(X_test[k].reshape(28, 28), cmap="gray")
    ax[i].set_title("GT: {}\n Pred: {}".format(labels_text[y_test[k]], labels_text[preds[k]]))
```





Análisis: A la vista de los resultados obtenidos y de la matriz de confusión podemos plantearnos responder a las cuestiones:

£Cómo son los errores?

£Parecen razonables?

Respuesta:

<ul>

Este modelo parece clasificar ligeramente mejor las imágenes con “Coat” y “Pullover” que el modelo anterior, pero un poco peor las imágenes de “Dress”.

La mayor confusión para este modelo está también entre “Pullover” y “Coat”, seguido de la confusión entre “T-shirt” y “Dress”.

En este caso el modelo se equivoca en alguna predicción de “Trouser”, pero también es la prenda que mejor clasifica como en el modelo anterior.

Viendo algunos ejemplos de errores estos parecen bastante razonables.

## 2.4 Redes neuronales

A continuación y como tercera parte de los análisis de métodos supervisados, vamos a utilizar una red neuronal para clasificar las imágenes de ropa. Utilizaremos también ahora una búsqueda aleatoria para ajustar los hiperparámetros de la red neuronal. En particular, utilizaremos una red monocapa con 4 salidas (una para cada clase del conjunto de datos) entrenada con el método de retropropagación y el optimizador SGD. Las neuronas de la capa oculta tendrán como activación la función sigmoide. Los hiperparámetros a ajustar en este caso son los siguientes:

Número de neuronas de la capa oculta: probaremos valores entre 20 y 200.

Número de épocas de entrenamiento: probaremos valores entre 10 y 50.

Velocidad de aprendizaje (learning rate): probaremos valores entre 0.001 y 0.2.

El procedimiento para validar el rendimiento del modelo para cada combinación de parámetros que se utilizará será el mismo que ya se ha venido utilizando en los casos anteriores: validación cruzada con 4 particiones generadas de forma estratificada.

### 2.4.1 Número óptimo de neuronas

Implementación: Como parte fundamental del proceso debemos realizar el cálculo del valor óptimo del número de neuronas de la capa oculta, el número de épocas de entrenamiento y la velocidad de aprendizaje utilizando 10 combinaciones de parámetros elegidas al azar. Para este trabajo vamos utilizar los módulos Sequential, Dense y SGD de keras, además de uniform y randint de scipy y StratifiedKFold de sklearn.

```
In [10]: import keras
         from keras.models import Sequential
         from keras.layers import Softmax
         from keras.layers import Activation, Dense
         from keras.optimizers import SGD
         from sklearn.model_selection import StratifiedKFold
         from scipy.stats import uniform as sp_rand
         from scipy.stats import randint
         from time import time

         kf = StratifiedKFold(n_splits=4)

         n_iter_search = 10
         params = np.zeros((n_iter_search, 3))

         n_neurons_dist = randint(low=20, high=200)
         n_epochs_dist = randint(low=10, high=50)
         lr_dist = sp_rand(loc=0.001, scale=0.2)

         best_it = -1
         max_score = 0

         start = time()
         for i in range(n_iter_search):
             n_neurons = n_neurons_dist.rvs()
             n_epochs = n_epochs_dist.rvs()
             lr = lr_dist.rvs()

             params[i, 0] = n_neurons
             params[i, 1] = n_epochs
             params[i, 2] = lr

             print("Prueba {}".format(i))
             print("Número de neuronas: {}; Número de épocas: {}; Velocidad de aprendizaje: {}".format(n_neurons, n_epochs, lr))

             scores = np.zeros(4)
             j = 0
             for train_index, test_index in kf.split(X_train_pca, y_train):
                 X_train_kf, X_test_kf = X_train_pca[train_index], X_train_pca[test_index]
                 y_train_kf, y_test_kf = y_train[train_index], y_train[test_index]
```

```

y_train_kf = keras.utils.to_categorical(y_train_kf, num_classes=n_classes)
y_test_kf = keras.utils.to_categorical(y_test_kf, num_classes=n_classes)

model = Sequential()
model.add(Dense(n_neurons, input_shape=(100,), activation="sigmoid"))
model.add(Dense(n_classes, activation="softmax"))

model.compile(optimizer=SGD(lr=lr), loss='categorical_crossentropy', metrics=

model.fit(X_train_kf, y_train_kf, epochs=n_epochs, verbose=0)

score = model.evaluate(X_test_kf, y_test_kf, verbose=0)
scores[j] = score[1]
j += 1

score_mean = np.mean(scores)
score_std = np.std(scores)
print("Precisión media: {:.2f} +/- {:.2f}".format(score_mean*100, score_std*100))

if (score_mean > max_score):
    max_score = score_mean
    best_it = i

end = time()

print("La búsqueda llevó {} segundos".format(end - start))

```

Using TensorFlow backend.

Prueba 0

Número de neuronas: 112; Número de épocas: 17; Velocidad de aprendizaje: 0.06227011358599537  
Precisión media: 85.75 +/- 1.02

Prueba 1

Número de neuronas: 41; Número de épocas: 24; Velocidad de aprendizaje: 0.15659340480693826  
Precisión media: 86.83 +/- 1.06

Prueba 2

Número de neuronas: 180; Número de épocas: 47; Velocidad de aprendizaje: 0.08986082726614962  
Precisión media: 86.95 +/- 1.10

Prueba 3

Número de neuronas: 102; Número de épocas: 25; Velocidad de aprendizaje: 0.10306330361645728  
Precisión media: 86.05 +/- 0.64

Prueba 4

Número de neuronas: 171; Número de épocas: 31; Velocidad de aprendizaje: 0.09531607444757116  
Precisión media: 86.28 +/- 1.60

Prueba 5

Número de neuronas: 108; Número de épocas: 17; Velocidad de aprendizaje: 0.09936269191534093  
Precisión media: 86.50 +/- 0.88

Prueba 6

Número de neuronas: 164; Número de épocas: 40; Velocidad de aprendizaje: 0.058797417656567055

Precisión media: 86.40 +/- 1.06

Prueba 7

Número de neuronas: 159; Número de épocas: 40; Velocidad de aprendizaje: 0.11559052536216151

Precisión media: 86.53 +/- 0.87

Prueba 8

Número de neuronas: 193; Número de épocas: 14; Velocidad de aprendizaje: 0.09444927001195838

Precisión media: 85.60 +/- 1.03

Prueba 9

Número de neuronas: 84; Número de épocas: 18; Velocidad de aprendizaje: 0.14530596825397996

Precisión media: 86.67 +/- 1.37

La búsqueda llevó 143.8752293586731 segundos

Análisis: A la vista de los resultados obtenidos con la codificación previa, podemos proceder a responder las siguientes cuestiones:

¿Qué parámetros han dado mejores resultados?

¿Qué variación hay entre las diferentes combinaciones de parámetros?

¿Es significativa la variación entre las diferentes combinaciones?

¿Hay algún parámetro que influya más que el otro?

Respuesta:

<ul>

La mayor precisión se obtiene con 23 neuronas en la capa oculta y entrenando 40 épocas con una velocidad de aprendizaje de 0.119.

En este caso podemos ver mayor variabilidad entre las precisiones medias para cada combinación por lo que, aun teniendo en cuenta las desviaciones estandard podemos afirmar que algunas soluciones son mejores que otras.

Con pocas pruebas y las desviaciones estandard altas es difícil extraer relaciones claras entre hiperparámetros, pero parece aflorar que las velocidades de aprendizaje más bajas dan peores resultados en la precisión, lo cual es quiere decir que probablemente se debería haber entrenado durante más épocas.

## 2.4.2 Número óptimo de neuronas (conjunto completo)

Implementación: Ahora vamos a entrenar una red neuronal con los valores de los hiperparámetros óptimos utilizando todo el conjunto  $X_{train\_pca}$  y mostraremos la precisión de la predicción del modelo en el conjunto  $X_{test\_pca}$ .

```
In [11]: print("Número de neuronas óptimo: {}".format(params[best_it, 0]))
         print("Número de épocas óptimo: {}".format(params[best_it, 1]))
         print("Velocidad de aprendizaje óptima: {}".format(params[best_it, 2]))

model = Sequential()
model.add(Dense(int(params[best_it, 0]), input_shape=(100,)))
model.add(Dense(n_classes, activation='softmax'))
```



```

model.compile(optimizer=SGD(lr=params[best_it, 2]), loss='categorical_crossentropy', m
model.fit(X_train_pca, keras.utils.to_categorical(y_train, num_classes=n_classes), ep

preds = np.argmax(model.predict(X_test_pca, verbose=0), axis=1)

accuracy = np.true_divide(np.sum(preds == y_test), preds.shape[0])*100
print("Precisión en el conjunto de test: {:.2f}%".format(accuracy))

```

Número de neuronas óptimo: 180.0

Número de épocas óptimo: 47.0

Velocidad de aprendizaje óptima: 0.08986082726614962

Precisión en el conjunto de test: 84.30%

### 2.4.3 La matriz de confusión del modelo

Implementación: Finalmente mostramos la matriz de confusión del modelo y algunas imágenes que el modelo ha clasificado incorrectamente junto con la etiqueta asignada por el modelo y la etiqueta original.

```

In [12]: cnf_matrix = confusion_matrix(y_test, preds)

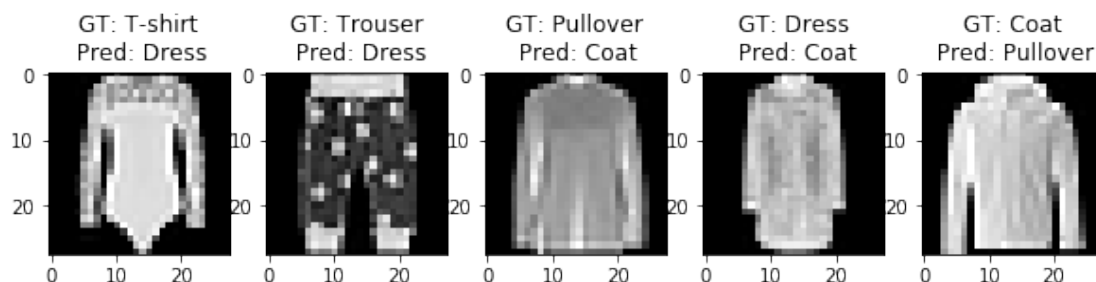
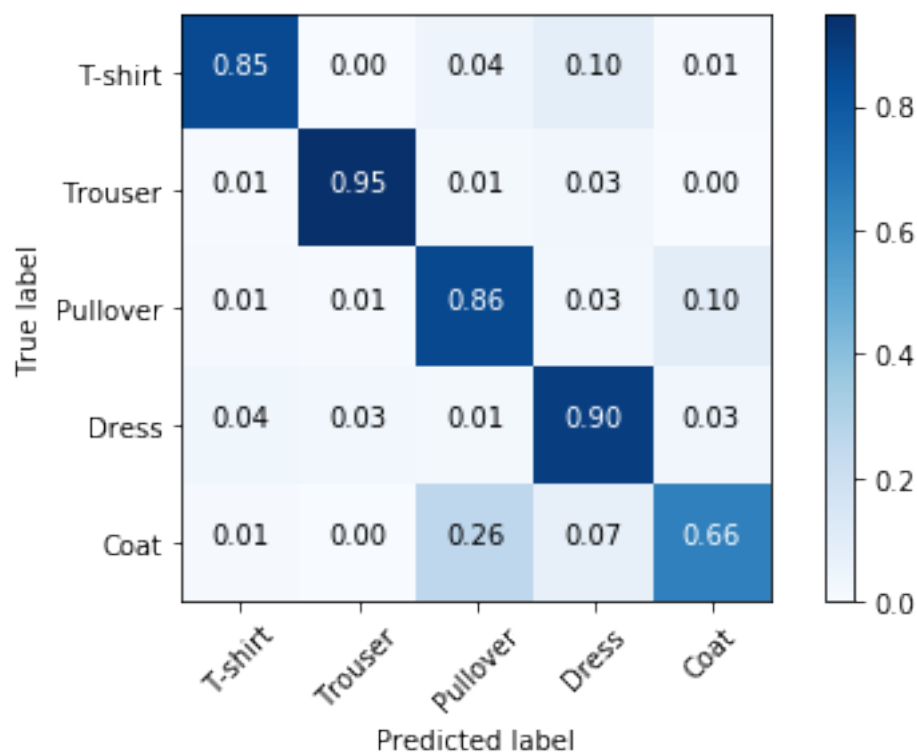
plot_confusion_matrix(cnf_matrix, classes=labels_text)

fig, ax = plt.subplots(1, n_classes, figsize=(10,10))

idxs = [np.where((y_test == i) & (preds != i))[0] for i in range(n_classes)]

for i in range(n_classes):
    k = np.random.choice(idxs[i])
    ax[i].imshow(X_test[k].reshape(28, 28), cmap="gray")
    ax[i].set_title("GT: {} \n Pred: {}".format(labels_text[y_test[k]], labels_text[pr

```



Análisis: A la vista de los resultados obtenidos y de la matriz de confusión podemos plantearnos responder a las cuestiones:

¿Cómo son los errores?

¿Parecen razonables?

Respuesta:

Este modelo confunde muchas “T-shirt” con “Dress” y muchos “Pullover” con “Coat”. Esto por su parte conlleva a la vez que el modelo prediga mejor la clase “Coat” que los otros modelos.

En el caso de “Trouser” lo clasifica con mucha seguridad, pero aún así se equivoca ligeramente más que los modelos anteriores.

Viendo algunos ejemplos parece razonable que el modelo confunda algunas imágenes de “T-shirt” con “Dress” y viceversa.

## 2.5 Optimización de métricas

En las etapas de análisis anteriores hemos buscado siempre el modelo que mejor precisión obtiene en general, pero esto no es siempre lo más adecuado. Por ejemplo, imaginemos que necesitamos el modelo para una empresa que únicamente vende pantalones y está haciendo un estudio sobre las imágenes de pantalones que obtiene de Internet. En este escenario, imaginemos que la empresa quiere estudiar el máximo número posible de imágenes de pantalones, por lo que está muy interesada en que el modelo no clasifique erróneamente imágenes de pantalones (asumiendo si es necesario que para ello habrá imágenes clasificadas como pantalones que en realidad no lo sean).

La misma idea de utilidad del modelo se puede encontrar, aunque con un ejemplo más complejo, en [este enlace](#).

### 2.5.1 Nueva función personalizada de coste

Implementación: Definimos a continuación una función que, dada la predicción del modelo para un conjunto de imágenes y las etiquetas reales de los datos, devuelva un coste de forma que los errores de clasificar un pantalón como otra prenda tengan el doble de peso que los otros errores.

```
In [13]: def coste(preds, labels):
        p = np.argmax(preds, 1)
        l = np.argmax(labels, 1)
        errors = np.sum(p != l)
        trousers = np.where(l == 1)[0]
        errors_trousers = np.sum(p[trousers] != l[trousers])

        return errors + errors_trousers
```

### 2.5.2 Aplicación al modelo de la nueva función de coste

Implementación: Utilizamos la función definida anteriormente junto con el código de entrenamiento de la red neuronal para optimizar los hiperparámetros de la red según la nueva métrica.

```
In [14]: start = time()
        for i in range(n_iter_search):
            n_neurons = n_neurons_dist.rvs()
            n_epochs = n_epochs_dist.rvs()
            lr = lr_dist.rvs()

            params[i, 0] = n_neurons
            params[i, 1] = n_epochs
            params[i, 2] = lr

            print("Prueba {}".format(i))
            print("Número de neuronas: {}; Número de épocas: {}; Velocidad de aprendizaje: {}".format(n_neurons, n_epochs, lr))

            scores = np.zeros(4)
            j = 0
            for train_index, test_index in kf.split(X_train_pca, y_train):
                X_train_kf, X_test_kf = X_train_pca[train_index], X_train_pca[test_index]
```

```

y_train_kf, y_test_kf = y_train[train_index], y_train[test_index]

y_train_kf = keras.utils.to_categorical(y_train_kf, num_classes=n_classes)
y_test_kf = keras.utils.to_categorical(y_test_kf, num_classes=n_classes)

model = Sequential()
model.add(Dense(n_neurons, input_shape=(100,), activation="sigmoid"))
model.add(Dense(n_classes, activation='softmax'))

model.compile(optimizer=SGD(lr=lr), loss='categorical_crossentropy', metrics=

model.fit(X_train_kf, y_train_kf, epochs=n_epochs, verbose=0)

preds = model.predict(X_test_kf, verbose=0)

score = coste(preds, y_test_kf)

scores[j] = score
j += 1

score_mean = np.mean(scores)
score_std = np.std(scores)
print("Precisión media: {:.2f} +/- {:.2f}".format(score_mean*100, score_std*100))

if (score_mean > max_score):
    max_score = score_mean
    best_it = i

end = time()

print("La búsqueda llevó {} segundos".format(end - start))

```

Prueba 0

Número de neuronas: 182; Número de épocas: 23; Velocidad de aprendizaje: 0.10833462438168667  
Precisión media: 14450.00 +/- 1213.47

Prueba 1

Número de neuronas: 79; Número de épocas: 27; Velocidad de aprendizaje: 0.1619478669035974  
Precisión media: 14575.00 +/- 1488.92

Prueba 2

Número de neuronas: 153; Número de épocas: 24; Velocidad de aprendizaje: 0.12265323722077036  
Precisión media: 14775.00 +/- 794.91

Prueba 3

Número de neuronas: 186; Número de épocas: 21; Velocidad de aprendizaje: 0.1583979225276578  
Precisión media: 14900.00 +/- 863.13

Prueba 4

Número de neuronas: 58; Número de épocas: 43; Velocidad de aprendizaje: 0.1063801229707025  
Precisión media: 13950.00 +/- 887.41

Prueba 5

Número de neuronas: 113; Número de épocas: 48; Velocidad de aprendizaje: 0.1576013661346941  
Precisión media: 13725.00 +/- 1188.22  
Prueba 6  
Número de neuronas: 123; Número de épocas: 21; Velocidad de aprendizaje: 0.09144483615365742  
Precisión media: 14825.00 +/- 749.58  
Prueba 7  
Número de neuronas: 123; Número de épocas: 45; Velocidad de aprendizaje: 0.03916805682202471  
Precisión media: 14575.00 +/- 1198.70  
Prueba 8  
Número de neuronas: 56; Número de épocas: 36; Velocidad de aprendizaje: 0.146017521145336  
Precisión media: 13775.00 +/- 1063.90  
Prueba 9  
Número de neuronas: 159; Número de épocas: 29; Velocidad de aprendizaje: 0.0752219505972483  
Precisión media: 14925.00 +/- 1123.33  
La búsqueda llevó 231.61324739456177 segundos

Análisis: La aplicación en el modelo de nuestra nueva función de **coste** permite plantear las preguntas:

¿Han cambiado significativamente los mejores valores de los hiperparámetros?

¿Cuál crees que puede ser la razón?

Respuesta:

Efectivamente, han cambiado significativamente los resultados de la prueba.

Los valores óptimos de los hiperparámetros son bastante diferentes.

Aún así, hay valores de hiperparámetros parecidos a los óptimos del ejercicio anterior que dan resultados similares a los valores óptimos de este problema.

Esto viene dado porque la métrica que estamos utilizando ahora no cambia los datos ni el proceso de entrenamiento, si no que nos sirve para seleccionar, de entre las soluciones que dan buenos resultados, qué solución es la más apropiada en el caso de que los pantalones tengan más peso en la métrica.

Al cambiar las metricas estandard 'metrics=['accuracy']' por las personalizadas que hemos construido en la función 'coste()', se consigue que la red entrene de manera mas eficiente las capas de aprendizaje.

### 3 Combinación de clasificadores

En esta segunda parte del trabajo vamos a trabajar sobre diferentes métodos de combinación de clasificadores aplicados sobre el conjunto de datos [Fashion MNIST](#).

Combinación paralela de clasificadores base similares

Bagging

Random Forest simple

Out-of-bag

Probabilidad por clase

Importancia de las variables

Número de clasificadores

Volumen de datos

</ul>

- Boosting
- Combinación secuencial de clasificadores base diferentes
- Stacking
- Cascading

</ul>

- Cascading simple
- Cascading con variables adicionales

</ul>

</ul>

## 3.1 Combinación paralela de clasificadores base similares

### 3.1.1 Bagging

**Random forest simple** La idea básica del *bagging* es utilizar el conjunto de entrenamiento original para generar centenares o miles de conjuntos similares usando muestreo con reemplazo. En este concepto está basado el algoritmo *Random Forest*, la combinación de varios árboles de decisión, cada uno entrenado con una realización diferente de los datos. La decisión final del clasificador combinado (la *Random Forest*) se toma por mayoría, dando el mismo peso a todas las decisiones parciales tomadas por los clasificadores base (los árboles).

Implementación: Usando los conjuntos *X\_train\_pca* e *y\_train\_pca*, entrenamos un modelo *Random Forest* con 100 árboles de decisión y estimamos la precisión del modelo con una estrategia de *cross-validation* en los mismos conjuntos.

Usaremos los módulos *RandomForestClassifier* y *cross\_val\_score* de *sklearn*. Sobre el funcionamiento y aplicaciones de *cross validation* y sobre como usar estos módulos, los siguientes enlaces son fundamentales:

- [RandomForestClassifier](#)
- [CrossValidation](#)
- [model\\_selection.cross\\_val\\_score](#)

</ul>

```
In [15]: from sklearn.model_selection import cross_val_score
         from sklearn import ensemble
         from time import time

         clf = ensemble.RandomForestClassifier(n_estimators=100)

         cvscores = cross_val_score(clf, X_train_pca, y_train, cv=5)

         print("Precisión media obtenida con cross-validation (CV): {:.2f} +/- {:.2f} %".format(
Precisión media obtenida con cross-validation (CV): 85.62 +/- 1.85 %
```

**Out-of-bag** Una ventaja del *bagging* usado en el *Random Forest* es que cada uno de los árboles de decisión ha sido entrenado con una combinación diferente de los datos (muestreo con reemplazo), o sea que cada uno de los árboles no ha visto una determinada parte de los datos originales. Esto define una especie de conjunto de test para cada uno de los árboles, llamado *out-of-bag*, que puede ser usado para estimar el error del modelo sin necesidad de usar el conjunto de test real que creamos previamente, ni de usar estrategias de *cross-validation*.

Implementación: Usando los conjuntos *X\_train\_pca* e *y\_train\_pca*, entrenamos a continuación un modelo Random Forest con 100 árboles de decisión. Mostramos la precisión de este modelo en el *out-of-bag* y en el conjunto *X\_test\_pca*.

Usamos el módulo *RandomForestClassifier* de *sklearn*. Como referencias sobre *out-of-bag* y sobre como usar este módulo (incluyendo el atributo *oob\_score\_*), los siguientes enlaces son los que hemos consultado:

- RandomForestClassifier
- plot\_ensemble\_oob

</ul>

```
In [16]: clf = ensemble.RandomForestClassifier(n_estimators=100, oob_score=True, random_state=0)

clf.fit(X_train_pca, y_train)

print("Precisión de este modelo con uso de Out-of-bag: {:.2f} %".format(clf.oob_score_))

preds_rfc = clf.predict(X_test_pca)
accuracy = np.true_divide(np.sum(preds_rfc == y_test), preds_rfc.shape[0])*100
print("Precisión en el conjunto de test: {:.2f}%".format(accuracy))
```

Precisión de este modelo con uso de Out-of-bag: 84.95 %

Precisión en el conjunto de test: 85.10%

Análisis: A la vista de los resultados obtenidos:

La precisión obtenida en el *out-of-bag* y en el conjunto de test ¿son comparables?

¿Era de esperar?

Respuesta:

La precisión medida con el uso de 'out-of-bag' y la obtenida en el conjunto de test son muy parecidas (aprox. una diferencia de 0,15).

Este resultado era de esperar porque:

la estimación del error con el out-of-bag es un método robusto y nos da el error del modelo en el conjunto de datos *X\_train\_pca*

el error en el conjunto de datos de test *X\_test\_pca* es muy parecido porque *X\_test\_pca* tiene la misma estructura (fue creado con separación aleatoria estratificada) que el conjunto de datos de entrenamiento *X\_train\_pca*

**Probabilidad por clase** Otra ventaja del *bagging* usado en el *Random Forest* es que cada uno de los árboles de decisión, entrenado con una combinación diferente de los datos, puede obtener un resultado diferente. En los problemas de clasificación, el resultado de cada árbol se considera como un voto diferente, y la predicción final del modelo es la clase que haya obtenido más votos teniendo en cuenta todos los árboles.

Estos votos individuales de los árboles también se pueden usar para estimar la probabilidad con la que el modelo prevé cada una de las clases, siendo la probabilidad para cada clase igual al número de votos obtenidos para aquella clase dividido entre el número de árboles.

Implementación: Para cada clase (etiqueta), muestra un ejemplo de imagen que el modelo haya clasificado incorrectamente junto con la etiqueta asignada por el modelo y la etiqueta original. Muestra también las probabilidades que el modelo ha atribuido a cada clase para estas imágenes.

Vamos a usar el modelo que entrenado en el paso anterior con el módulo *RandomForestClassifier* de *sklearn* y las previsiones que calculamos para el conjunto de datos de test. Para mostrar las imágenes, usaremos el código proporcionado en la carga de datos original. Sobre el módulo *RandomForestClassifier* de *sklearn* (incluyendo el método *predict\_proba*), el siguiente enlace es el que hemos consultado:

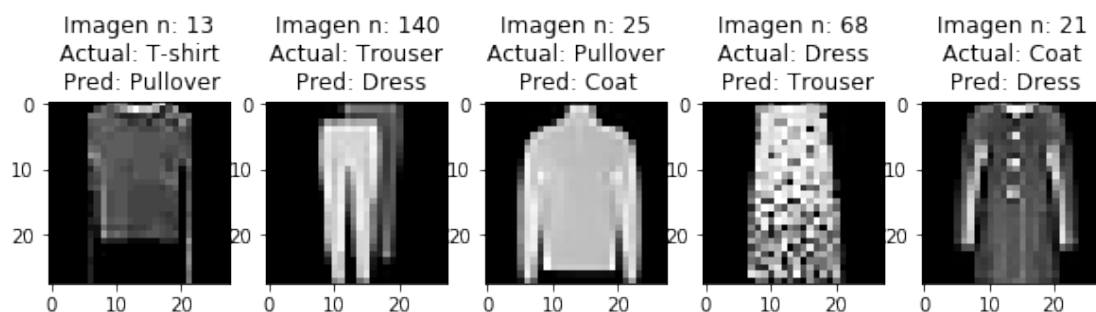
`RandomForestClassifier`

</ul>

```
In [17]: import matplotlib.pyplot as plt
         %matplotlib inline

         # Obtenemos los índices de la primera imagen mal identificada para cada clase:
         idxs = [np.where((y_test == i) & (preds_rfc != i))[0][0] for i in range(n_classes)]

         # Mostramos las imágenes junto con las etiquetas actuales y previstas:
         fig, ax = plt.subplots(1, n_classes, figsize=(10,10))
         for i in range(n_classes):
             k = idxs[i]
             ax[i].imshow(X_test[k].reshape(28, 28), cmap="gray")
             ax[i].set_title("Imagen n: {} \n Actual: {} \n Pred: {}".format(k, labels_text[y_test[k]], labels_text[preds_rfc[k]]))
```



```
In [18]: # Calculamos las probabilidades de cada clase para cada imagen:
         probs_rfc = clf.predict_proba(X_test_pca)

         # Mostramos las probabilidades junto con la clase prevista (probabilidad más alta) y
         print("Etiquetas: \t\t\t{}".format(labels_text))
         for i in range(n_classes):
             k = idxs[i]
```



```

print("Probabilidades imagen {}: \t{}; Actual: '{}'; Pred: '{}'.format(k, probs_rfc[k], y_test[k], y_pred[k])

# Adicional: calculamos cuantas veces la segunda clase con más probabilidad era la correcta
n_max_prob_correct = 0
n_2nd_max_prob_correct = 0
for i in range(len(y_test)):
    if y_test[i] == np.argmax(probs_rfc[i]):
        n_max_prob_correct += 1
    elif y_test[i] == probs_rfc[i].argsort()[-2]:
        n_2nd_max_prob_correct += 1
print("\nCuántas veces el modelo acertó? (precisión): {}%\nDe las veces que no acertó,
      .format(float(n_max_prob_correct)/len(y_test)*100,
              float(n_2nd_max_prob_correct)/(len(y_test)-n_max_prob_correct)*100))

```

```

Etiquetas:          ['T-shirt', 'Trouser', 'Pullover', 'Dress', 'Coat']
Probabilidades imagen 13: [0.24 0.06 0.35 0.2 0.15]; Actual: 'T-shirt'; Pred: 'Pullover'
Probabilidades imagen 140: [0.1 0.3 0.07 0.41 0.12]; Actual: 'Trouser'; Pred: 'Dress'
Probabilidades imagen 25: [0.02 0.05 0.38 0.08 0.47]; Actual: 'Pullover'; Pred: 'Coat'
Probabilidades imagen 68: [0.28 0.32 0.12 0.14 0.14]; Actual: 'Dress'; Pred: 'Trouser'
Probabilidades imagen 21: [0.03 0.04 0.21 0.4 0.32]; Actual: 'Coat'; Pred: 'Dress'

```

Cuántas veces el modelo acertó? (precisión): 85.1%

De las veces que no acertó, cuántas veces la segunda clase más votada era la correcta?: 79.2%

**Análisis:** En estos casos en los que el modelo se equivocó, ¿estaba cerca de prever la etiqueta correcta?

**Respuesta:** Sí, aunque el modelo se equivocó, ha estado siempre muy cerca (relativamente) de obtener el resultado correcto. En 4 de los 5 casos analizados, la segunda clase con una probabilidad más alta era la correcta. Esta proporción se mantiene en todo el conjunto de datos de test.

**Importancia de las variables** Otra ventaja del algoritmo *Random Forest* es que permite medir la importancia relativa de cada variable, gracias a que cada uno de los árboles fue entrenado con un subconjunto diferente de las variables originales.

En el problema de clasificación de imágenes analizado aquí, la importancia de las variables nos permite saber cuáles son generalmente los píxeles más importantes para poder clasificar la imagen.

**Implementación:** Vamos a proceder a realizar el entrenamiento de un clasificador *Random Forest* con el conjunto de datos de entrenamiento original *X\_train*, en los que cada variable es la intensidad de cada píxel (en vez de ser las variables PCA que usamos anteriormente). Mostraremos cuáles son las 10 variables más importantes. También presentamos un gráfico en el que se visualizará que zonas de una imagen son más importantes para el clasificador.

Usaremos el módulo *RandomForestClassifier* de *sklearn* para calcular la importancia de las variables. Para representar gráficamente la importancia de cada píxel de la imagen, usaremos el código proporcionado en la carga de datos original. Sobre el módulo *RandomForestClassifier* de *sklearn* (incluyendo el método *feature\_importances\_*), el siguiente enlace es el que hemos consultado:

`RandomForestClassifier`

</ul>

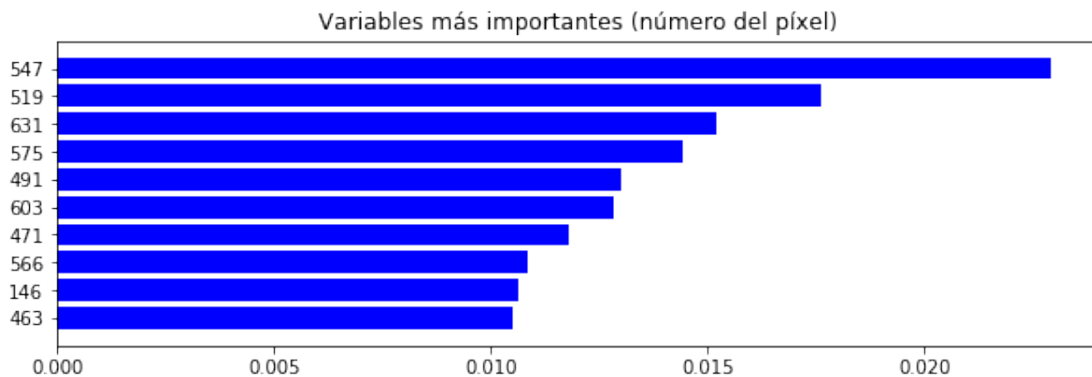
```

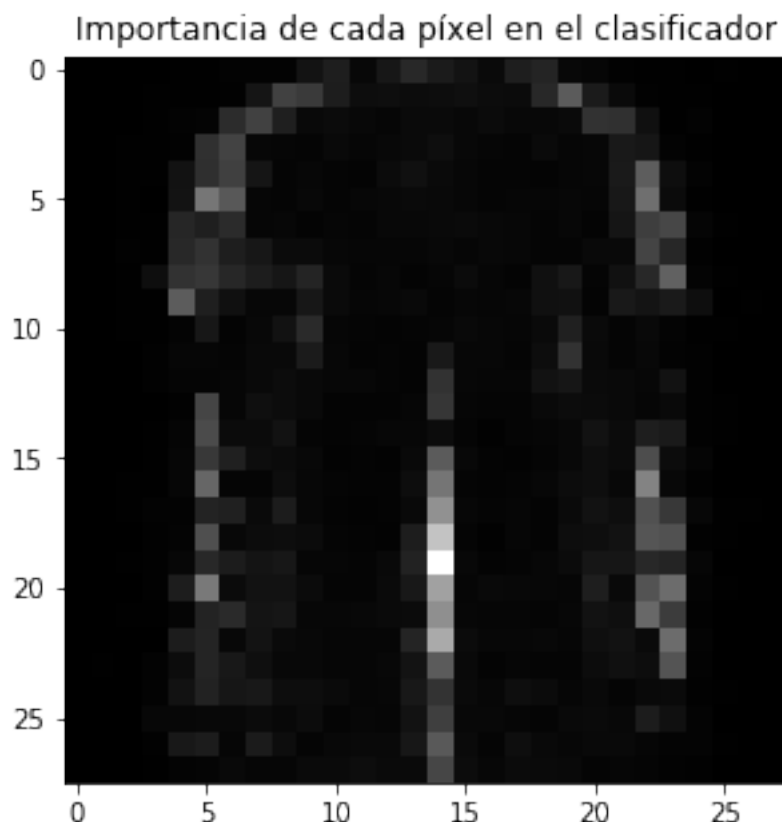
In [19]: # Entrenamos el clasificador con datos originales, sin PCA, para que cada variable se
        clf = ensemble.RandomForestClassifier(n_estimators=80)
        clf.fit(X_train, y_train)

        # calculamos la importancia de las variables y representación gráfica de las top 10:
        importances = clf.feature_importances_
        topN = 10
        indices = np.argsort(importances)
        n_feats = min(len(importances), topN)
        labels = np.arange(1, np.shape(X_train)[1]+1) # nombres de las variables, en este caso
        plt.figure(figsize=(10,3))
        plt.title("Variables más importantes (número del píxel)")
        plt.barh(range(n_feats), importances[indices][-n_feats:], color="b")
        plt.yticks(range(n_feats), labels[indices][-n_feats:])
        plt.ylim([-1, n_feats])
        plt.show()

        # representación gráfica de la importancia de cada píxel:
        plt.figure(figsize=(5,5))
        plt.title("Importancia de cada píxel en el clasificador")
        plt.imshow(importances.reshape(28, 28), cmap="gray")
        plt.show()

```





**Análisis:** A la vista del resultado obtenido podemos plantearnos si nos parece plausible el resultado que hemos obtenido. ¿Porqué lo creemos así?

**Respuesta:** En el gráfico con la 'Importancia de cada píxel en el clasificador' vemos que hay una zona vertical central que es la más importante, así como una especie de número ocho difuso alrededor. Parece muy razonable ya que la línea central permite diferenciar los pantalones, mientras que la zona importante alrededor está situada aproximadamente para localizar el contorno de las diferentes prendas de vestir.

**Número de clasificadores** En los pasos anteriores hemos combinado 100 clasificadores simples en nuestro clasificador combinado. ¿Será posible que la precisión del clasificador combinado aumente indefinidamente su desempeño si añadimos más clasificadores?

Para responder a esta pregunta vamos a representar una curva de validación. La curva de validación es una representación gráfica del desempeño de un modelo variando uno de sus parámetros. Mientras que la búsqueda de rejilla nos permite encontrar la combinación de parámetros que da mejores resultados, la curva de validación nos permite entender cuál es el impacto de un determinado parámetro en el desempeño de un modelo.

**Implementación:** Entrenamos varios modelos de *Random Forest* con un número de árboles cada vez mayor. Para cada modelo, calcularemos su precisión en el conjunto de test o usando *cross-validation* en el conjunto de entrenamiento. Adicional: También vamos a representar gráficamente la evolución de la precisión con el número de árboles para ayudarnos en el posterior análisis de los resultados.

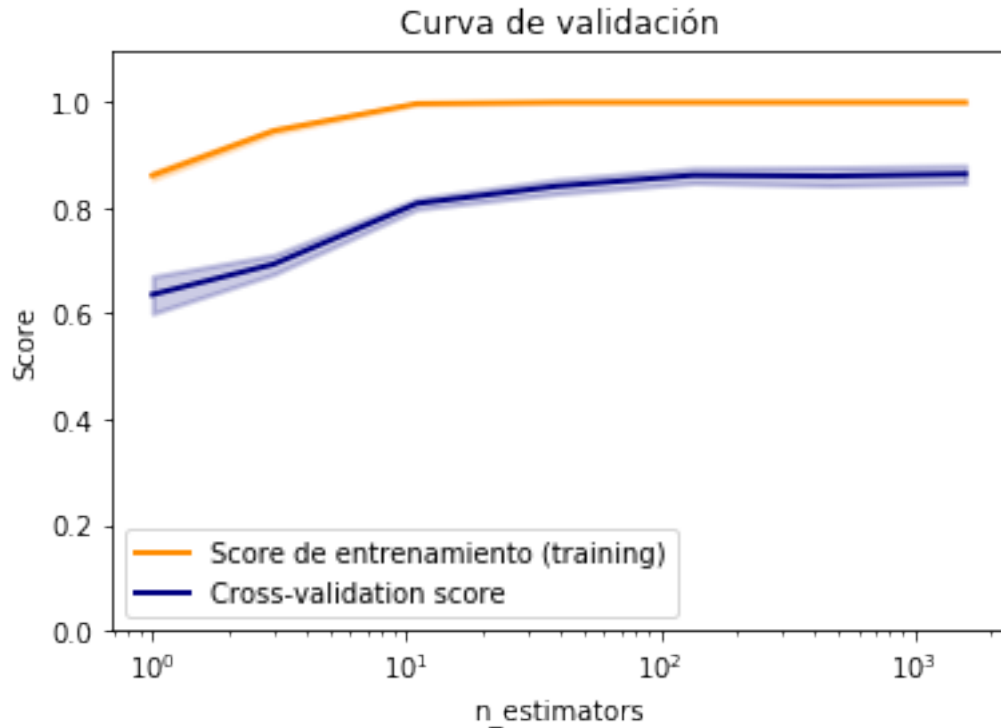
Usaremos el módulo *validation\_curve* de sklearn. Los siguientes enlaces son los que hemos consultado para poder realizar esta parte del análisis:

`model_selection.validation_curve`  
`validation-curve`

```
In [20]: from sklearn.model_selection import validation_curve
```

```
param_range = np.logspace(0, 3.2, 7).astype(np.int)
param_name="n_estimators"
train_scores, test_scores = validation_curve(
    ensemble.RandomForestClassifier(), X_train_pca, y_train,
    param_name=param_name, param_range=param_range,
    cv=5, scoring="accuracy", n_jobs=4)
train_scores_mean = np.mean(train_scores, axis=1)
train_scores_std = np.std(train_scores, axis=1)
test_scores_mean = np.mean(test_scores, axis=1)
test_scores_std = np.std(test_scores, axis=1)

plt.title("Curva de validación")
plt.xlabel(param_name)
plt.ylabel("Score")
plt.ylim(0.0, 1.1)
lw = 2
plt.semilogx(param_range, train_scores_mean, label="Score de entrenamiento (training)",
             color="darkorange", lw=lw)
plt.fill_between(param_range, train_scores_mean - train_scores_std,
                train_scores_mean + train_scores_std, alpha=0.2,
                color="darkorange", lw=lw)
plt.semilogx(param_range, test_scores_mean, label="Cross-validation score",
             color="navy", lw=lw)
plt.fill_between(param_range, test_scores_mean - test_scores_std,
                test_scores_mean + test_scores_std, alpha=0.2,
                color="navy", lw=lw)
plt.legend(loc="best")
plt.show()
```



Análisis: A la vista del gráfico anterior, ¿podemos decir que se incrementa indefinidamente la precisión con el número de árboles combinados? Si satura, ¿lo hace a la precisión máxima o a otro valor? ¿Por qué?

Respuesta: La precisión no aumenta indefinidamente con el número de árboles combinados, sino que se satura a un valor alrededor del 85%. Añadir más árboles sólo es útil cuando estos añaden información nueva, porque han sido entrenados con datos distintos que permiten al algoritmo mejorar las decisiones tomadas. La parcialidad de un clasificador, por falta de riqueza en los datos o falta de atributos esenciales, no puede ser totalmente solucionada añadiendo nuevos clasificadores parciales.

**Volumen de datos** Podemos pensar que entrenando el modelo con más datos (más imágenes) ¿el modelo aprendería a clasificar con mejor precisión? Es muy útil intentar responder a esta pregunta antes de lanzarse a conseguir más datos, ya que este puede ser un proceso difícil, caro, o que implique esperar mucho tiempo.

Para responder a esta pregunta, analizaremos cómo evoluciona la precisión del modelo en los conjuntos de entrenamiento y test para diferentes volúmenes de datos de creciente tamaño. Representar los resultados en una curva de aprendizaje (*learning curve*) nos permitirá analizar visualmente estas cantidades.

Implementación: Entrenamos varios modelos de *Random Forest* con un volumen de datos cada vez mayor. Para cada modelo, calcularemos su precisión en el conjunto de entrenamiento y de test, y representaremos los resultados en un gráfico.

Usaremos el módulo *learning\_curve* de sklearn. Hemos consultado los siguientes enlaces para su utilización:

[learning-curve](#)

model\_selection.learning\_curve

```
In [21]: from sklearn.model_selection import learning_curve
```

```
"""
```

```
Esta función corresponde con la original de la fuente:
```

```
https://scikit-learn.org/stable/auto\_examples/model\_selection/plot\_learning\_curve.htm
```

```
Se han hecho variaciones en el código original
```

```
"""
```

```
def plot_learning_curve(estimator, title, X, y, ylim=None, cv=None,  
                        n_jobs=1, train_sizes=np.linspace(.1, 1.0, 5)):
```

```
    """
```

```
Generate a simple plot of the test and training learning curve.
```

```
Parameters
```

```
-----
```

```
estimator : object type that implements the "fit" and "predict" methods  
An object of that type which is cloned for each validation.
```

```
title : string  
Title for the chart.
```

```
X : array-like, shape (n_samples, n_features)  
Training vector, where n_samples is the number of samples and  
n_features is the number of features.
```

```
y : array-like, shape (n_samples) or (n_samples, n_features), optional  
Target relative to X for classification or regression;  
None for unsupervised learning.
```

```
ylim : tuple, shape (ymin, ymax), optional  
Defines minimum and maximum yvalues plotted.
```

```
cv : int, cross-validation generator or an iterable, optional  
Determines the cross-validation splitting strategy.
```

```
Possible inputs for cv are:
```

- None, to use the default 3-fold cross-validation,*
- integer, to specify the number of folds.*
- An object to be used as a cross-validation generator.*
- An iterable yielding train/test splits.*

```
For integer/None inputs, if ``y`` is binary or multiclass,  
:class:`StratifiedKFold` used. If the estimator is not a classifier  
or if ``y`` is neither binary nor multiclass, :class:`KFold` is used.
```

*Refer :ref:`User Guide <cross\_validation>` for the various cross-validators that can be used here.*

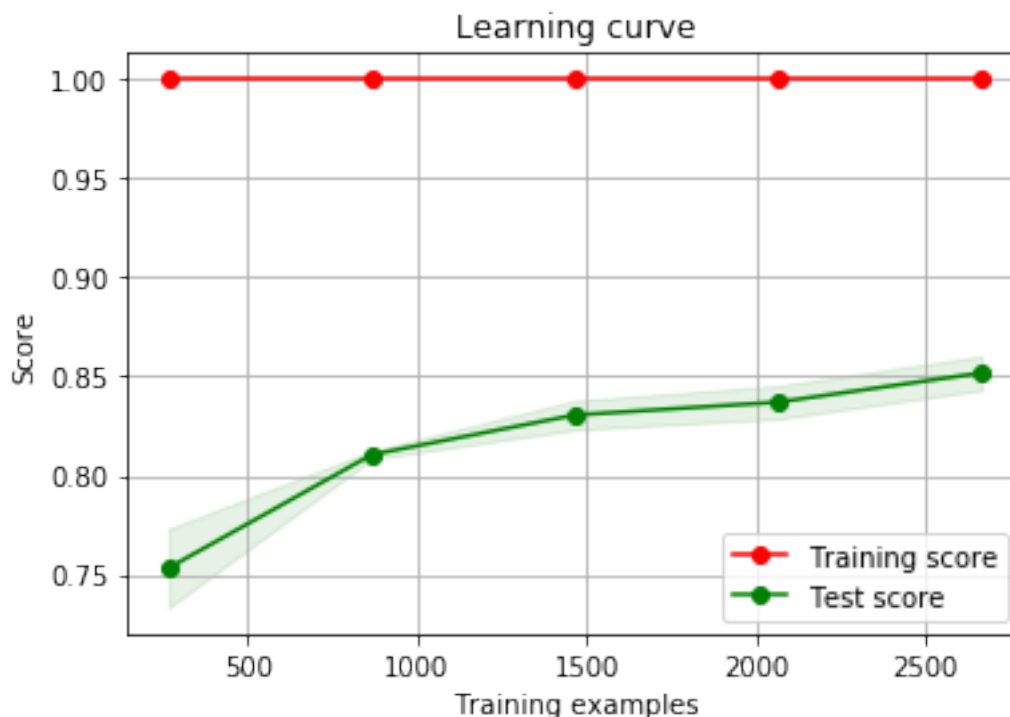
```
n_jobs : integer, optional
    Number of jobs to run in parallel (default 1).
"""
plt.figure()
plt.title(title)
# Las siguientes 2 lineas del código original no se usarán:
# if ylim is not None:
#     plt.ylim(*ylim)
plt.xlabel("Training examples")
plt.ylabel("Score")
train_sizes, train_scores, test_scores = learning_curve(
    estimator, X, y, cv=cv, n_jobs=n_jobs, train_sizes=train_sizes)
train_scores_mean = np.mean(train_scores, axis=1)
train_scores_std = np.std(train_scores, axis=1)
test_scores_mean = np.mean(test_scores, axis=1)
test_scores_std = np.std(test_scores, axis=1)
plt.grid()

plt.fill_between(train_sizes, train_scores_mean - train_scores_std,
                 train_scores_mean + train_scores_std, alpha=0.1,
                 color="r")
plt.fill_between(train_sizes, test_scores_mean - test_scores_std,
                 test_scores_mean + test_scores_std, alpha=0.1, color="g")
plt.plot(train_sizes, train_scores_mean, 'o-', color="r",
         label="Training score")
plt.plot(train_sizes, test_scores_mean, 'o-', color="g",
         label="Test score")

plt.legend(loc="best")
return plt

plot_learning_curve(ensemble.RandomForestClassifier(n_estimators=50),
                    "Learning curve",
                    X_train_pca, y_train,
                    cv = 3)
```

Out[21]: <module 'matplotlib.pyplot' from 'f:\\Users\\Ricardo\\Anaconda3\\envs\\my\_py35\\lib\\'



Análisis: A la vista del resultado, si obtuviésemos más datos de entrenamiento (más imágenes clasificadas) ¿mejoraría el modelo? ¿Porqué sí o porqué no?

Respuesta: En el gráfico vemos que la precisión del modelo en el conjunto de datos de entrenamiento es la máxima posible, por lo que el modelo consigue clasificar perfectamente esas imágenes. Sin embargo, la precisión en el conjunto de test está muy por debajo. Esto nos muestra que el modelo no generaliza suficientemente la lógica de clasificación de las imágenes, sino que esta lógica es demasiado específica del conjunto usado para entrenar el modelo. Este problema se conoce como *overfitting* y podría ser solucionado añadiendo más datos (más imágenes clasificadas). Por otro lado, en el gráfico también vemos que la precisión en el conjunto de test aumenta al aumentar el volumen de datos (aunque cada vez más lentamente, se comporta de manera asintótica hacia el valor 0.85), por lo que esperamos que la precisión continúe en aumento si tuviésemos más imágenes.

### 3.1.2 Boosting

En el sistema de *Boosting* se combinan varios clasificadores débiles secuencialmente, y en cada uno de ellos se da más peso a los datos que han sido erróneamente clasificados en las combinaciones anteriores, para que se concentre así en los casos más difíciles de resolver.

Implementación: Usando el conjunto `X_train_pca`, procederemos a entrenar un modelo *Gradient Boosting* y estimaremos la precisión del modelo con una estrategia de *cross-validation* en los mismos conjuntos. Seguidamente calcularemos las previsiones del modelo en el conjunto `X_test_pca` y su precisión en este conjunto.

Usaremos los módulos `GradientBoostingClassifier` y `cross_val_score` de `sklearn`. El siguiente enlace ha sido el consultado para realizar este proceso:

`GradientBoostingClassifier`



```
In [22]: clf = ensemble.GradientBoostingClassifier()

cvscores = cross_val_score(clf, X_train_pca, y_train, cv=5)

print("Precisión media obtenida con CV: {:.2f} +/- {:.2f} %".format(np.mean(cvscores),
                             np.std(cvscores)))

clf.fit(X_train_pca, y_train)

preds_gbc = clf.predict(X_test_pca)

# Calcular las probabilidades de cada clase para cada imagen:
probs_gbc = clf.predict_proba(X_test_pca)

Precisión media obtenida con CV: 86.15 +/- 1.72 %
```

Análisis: El boosting se basa en la combinación de clasificadores débiles. En la implementación que hemos utilizado en este ejercicio, ¿cuál es la profundidad de los árboles utilizados? Si la comparamos con la que ya utilizamos en los árboles de decisión del ejercicio de *bagging*, ¿qué podemos resolver?.

Respuesta: Por defecto en la implementación de sklearn se impone un límite de 3 niveles de profundidad en los árboles usados para el 'boosting', mientras que para el 'bagging' usamos árboles sin límite de profundidad.

## 3.2 Combinación secuencial de clasificadores base diferentes

### 3.2.1 Stacking

Un clasificador de *stacking* usa como atributos las predicciones hechas por otros clasificadores en lugar de los datos originales de entrada.

Para construir nuestro clasificador de *stacking* vamos a usar las predicciones hechas en el conjunto de test por los clasificadores: - utilizados en los ejercicios anteriores - K-Nearest neighbors Classifier (knc), Support Vector Machines Classifier (svmc) y Neural Network Classifier (nnc) - Discriminant Analysis (dac)

los dos últimos ya los tenemos generados y guardados en archivos adjuntos. Estas predicciones las vamos a cargar con el siguiente código:

```
In [23]: # carga de predicciones calculadas en la prueba PEC3:
preds_knc = np.load("preds_knc.pickle")
preds_svmc = np.load("preds_svmc.pickle")
preds_nnc = np.load("preds_nnc.pickle")

# carga de las predicciones por un modelo de Discriminant Analysis:
preds_dac = np.load("preds_dac.pickle")
```

Implementación: Tenemso la intención de construir un clasificador de *stacking* usando una *Random Forest* que use como atributos a las predicciones hechas en el conjunto de test por los algoritmos k-nn, SVM, red neuronal y Gradient Boosting. Calcularemos la precisión del modelo resultante con *cross-validation* en el conjunto de test.

Usaremos las funciones `column_stack` de *numpy* y *OneHotEncoder* de *sklearn* para preparar los datos. Para aprender a usar estas funciones hemos utilizado los siguientes enlaces:

`numpy.column_stack`

`OneHotEncoder`

`encoding-categorical-features`

```
In [24]: # Juntamos las predicciones de los distintos clasificadores:
```

```
X_test_stacking = np.column_stack((preds_rfc, preds_gbc, preds_knc, preds_svmc, preds_
print("Dimensiones de la matriz con las predicciones de todos los clasificadores: {}".format(X_test_stacking.shape))
```

```
# Transformamos las variables categóricas (son todas) con OneHotEncoder:
```

```
from sklearn.preprocessing import OneHotEncoder
```

```
enc = OneHotEncoder(categories='auto')
```

```
enc.fit(X_test_stacking)
```

```
X_test_stacking = enc.transform(X_test_stacking).toarray()
```

```
print("Dimensiones de la matriz para entrenar el clasificador de stacking: {}".format(X_test_stacking.shape))
```

```
# Calculamos la precisión de un RandomForestClassifier con estas variables usando CV:
```

```
clf = ensemble.RandomForestClassifier(n_estimators=100)
```

```
cvscores = cross_val_score(clf, X_test_stacking, y_test, cv=5)
```

```
print("Precisión media obtenida con CV: {:.2f} +/- {:.2f} %".format(np.mean(cvscores), np.std(cvscores)))
```

Dimensiones de la matriz con las predicciones de todos los clasificadores: (1000, 6)

Dimensiones de la matriz para entrenar el clasificador de stacking: (1000, 30)

Precisión media obtenida con CV: 88.60 +/- 1.46 %

Análisis: A la vista del resultado, ¿realmente hemos conseguido mejorar la precisión gracias al *stacking*? ¿Cuál es el motivo?.

Respuesta: La precisión obtenida gracias al ‘stacking’ es ligeramente superior a la obtenida con el mejor modelo anterior, gradient boosting. Aunque las desviaciones de las precisiones medias con CV son del mismo orden que la mejora obtenida, repitiendo el cálculo para diferentes ‘seeds’ nos da resultados consistentes en los que el ‘stacking’ es siempre ligeramente superior.

### 3.2.2 Cascading

**Cascading simple** El caso de *cascading* es parecido al de *stacking* pero utilizando no solamente las predicciones parciales de los clasificadores base, sino también los datos originales.

Implementación: Construiremos en este caso un clasificador de *cascading* usando una *Random Forest* que use como atributos a las predicciones hechas en el conjunto de test por los algoritmos k-nn, SVM, red neuronal y Gradient Boosting, así como también las variables originales. Calcularemos en ese punto la precisión del modelo resultante con *cross-validation* en el conjunto de test.

Vamos a usar el mismo conjunto de datos que en el paso previo pero añadiéndole el conjunto de test original *X\_test\_pca*.

```
In [25]: # Juntamos las predicciones de los distintos clasificadores y las variables originales:
```

```
X_test_cascading = np.column_stack((X_test_pca, X_test_stacking))
```

```
print("Dimensiones de la matriz: {}".format(np.shape(X_test_cascading)))
```

```
# Calculamos la precisión de un RandomForestClassifier con estas variables usando CV:
clf = ensemble.RandomForestClassifier(n_estimators=100)
cvscores = cross_val_score(clf, X_test_cascading, y_test, cv=5)
print("Precisión media obtenida con CV: {:.2f} +/- {:.2f} %".format(np.mean(cvscores), np.std(cvscores)))
```

Dimensiones de la matriz: (1000, 130)

Precisión media obtenida con CV: 88.20 +/- 1.36 %

Análisis: Con esta acción, ¿hemos conseguido mejorar la precisión gracias al *cascading*? ¿Porqué?

Respuesta: La precisión obtenida gracias al ‘cascading’ es equivalente o ligerísimamente superior a la obtenida con ‘stacking’. Las desviaciones de las precisiones medias con CV son superiores a mejora obtenida. Repitiendo el cálculo para diferentes ‘seeds’ vemos que el ‘cascading’ es consistentemente superior, pero el margen es muy pequeño.

**Cascading con variables adicionales** En el *cascading* también podemos añadir como variables del modelo a datos adicionales que se hayan podido generar durante la toma de decisiones de los clasificadores que combinamos.

Implementación: ¿Qué datos adicionales de los modelos anteriores podríamos usar para enriquecer al modelo? Construiremos un clasificador de *cascading* usando una *Random Forest* que gestione como atributos los usados en el paso anterior más otros que obtendremos de algunos de los clasificadores utilizados en otros de los pasos previos ya ejecutados de análisis. Finalmente, calcularemos la precisión del modelo resultante con *cross-validation* en el conjunto de test.

In [26]: # Cargamos datos:

```
probs_knc = np.load("probs_knc.pickle")
probs_svmc = np.load("probs_svmc.pickle")
probs_dac = np.load("probs_dac.pickle")

# Juntamos las predicciones de los distintos clasificadores incluyendo las probabilidades
X_test_cascading = np.column_stack((X_test_pca, probs_rfc, probs_gbc, probs_knc, probs_svmc, probs_dac))
print("Dimensiones de la matriz: {}".format(np.shape(X_test_cascading)))

# Calculamos la precisión de un RandomForestClassifier con estas variables usando CV:
clf = ensemble.RandomForestClassifier(n_estimators=100)
cvscores = cross_val_score(clf, X_test_cascading, y_test, cv=5)
print("Precisión media obtenida con CV: {:.2f} +/- {:.2f} %".format(np.mean(cvscores), np.std(cvscores)))
```

Dimensiones de la matriz: (1000, 155)

Precisión media obtenida con CV: 89.20 +/- 2.01 %

Análisis: ¿Hemos conseguido mejorar la precisión gracias a añadir datos adicionales al *stacking*? ¿Porqué?

Respuesta: Hemos añadido las probabilidades de cada clase predichas por cada uno de los modelos como datos adicionales. La precisión obtenida ‘stacking’ es ligeramente superior a la

obtenida anteriormente, aunque esta mejora está dentro de las desviaciones de las precisiones medias con CV. Los datos adicionales aportan nueva información al modelo, que ahora conoce con qué probabilidad fue prevista cada clase anteriormente, por lo que no nos sorprende que el modelo haya mejorado muy ligeramente.