Curso 2021-2022

Iñaki Diez Lambies y Manuel Diaz Pastor

Percepción

Proyecto de prácticas

Reconocimiento de dígitos manuscritos: MNIST

Contenido

[2 Ejercicio 2.1 - Principal Component Analysis 2](#_Toc101820031)

[2.1 Ejercicio 2.2 – Comprobación PCA 2](#_Toc101820032)

[3 Ejercicio obligatorio – KNN + PCA 2](#_Toc101820033)

[3.1 pca+knn-exp.py 2](#_Toc101820034)

[3.1.1 Resultados 3](#_Toc101820035)

[3.2 pca+knn-eva.py 3](#_Toc101820036)

[3.2.1 Resultados 3](#_Toc101820037)

[4 Ejercicio opcional – Algoritmo de Wilson 4](#_Toc101820038)

[4.1 mnn 4](#_Toc101820039)

[4.2 knnV 4](#_Toc101820040)

[4.3 wilson 4](#_Toc101820041)

[4.4 pca+knn+wilson-exp.py 5](#_Toc101820042)

[4.4.1 Resultados 5](#_Toc101820043)

[4.5 pca+knn+wilson-eva.py 5](#_Toc101820044)

[4.5.1 Resultados 5](#_Toc101820045)

Entrega 1

# Ejercicio 2.1 - Principal Component Analysis

Para este ejercicio se nos ha pedido realizar una implementación del algoritmo de Principal Component Analysis (PCA). En nuestro caso, la función que designaremos tiene como parámetro de entrada los datos de entrenamiento dispuestos por filas y, como resultado, no devolverá dos elementos: el vector media de los valores (media entre todas sus dimensiones) y la matriz de proyección *W*. Esta matriz se compone por los vectores propios de la matriz de entrada dispuestos en filas y ordenados de mayor a menor valor propio asociado.

## Ejercicio 2.2 – Comprobación PCA

Se nos pide comprobar el correcto funcionamiento del algoritmo anterior. Para esto hemos desarrollado un pequeño script que nos permite realizar el visionado de los vectores propios de los datos dados.

Para conseguir esto primero aplicamos PCA a los datos y seguidamente realizamos la representación vector propio a vector propio siguiendo lo realizado en la práctica 0. El script está diseñado para mostrarlos todos, de forma que para no tener que observar todo el rango de la matriz se debe parar la ejecución del programa a través de línea de comandos.

# Ejercicio obligatorio – KNN + PCA

Este ejercicio consiste en estudiar el comportamiento del clasificador por *k* vecinos más cercanos (KNN) respecto a la redimensionalidad aplicada a la matriz de muestras de entrenamiento. Esta última se consigue a partir de aplicar las *k* columnas de la matriz de proyección *W* a las muestras de entrenamiento (para *k* dimensiones, *k* columnas).

Vamos a completar dos programas para experimentar con esto.

## pca+knn-exp.py

Este programa nos permite designar, a partir de un conjunto de muestras de entrenamiento, una parte dedicada al entrenamiento y otra a realizar los test. Además, podemos designar un conjunto de dimensionalidades a aplicar y que, gracias a la función *knn*, podemos obtener el error de clasificación por vecinos más cercanos para unas muestras de entrenamiento y test dadas.

Así pues, nuestro programa separa de un conjunto de muestras en entrenamiento y test (líneas 23-27) después de ser aleatorizadas las muestras (líneas 19-21). Seguidamente realizamos el cálculo de la matriz de proyección para, a continuación, calcular la tasa de error para cada posible proyección en *k* dimensiones.

Siguiendo las directrices del boletín, y con el objetivo de replicar la gráfica dada, realizamos un experimento con el 90% de las muestras para entrenamiento y el 10% para test, así como pruebas con redimensionalidad en 1, 2, 5, 10, 20, 50, 100, 200 y 500 dimensiones.

Estos resultados son anotados siguiendo el formato indicado para realizar después la representación con GNUPlot. El resultado de esto se puede visualizar en el archivo *pca+knn-exp.eps*.en forma de dibujado vectorial. También se pueden consultar los resultados en el fichero *pca+knn-exp.out*.

### Resultados

Como podemos observar en la imagen, para una baja dimensionalidad el error excede en gran medida el del modelo original. Ahora bien, conforme aumentamos la dimensionalidad encontramos una mayor cercanía al punto de partida.

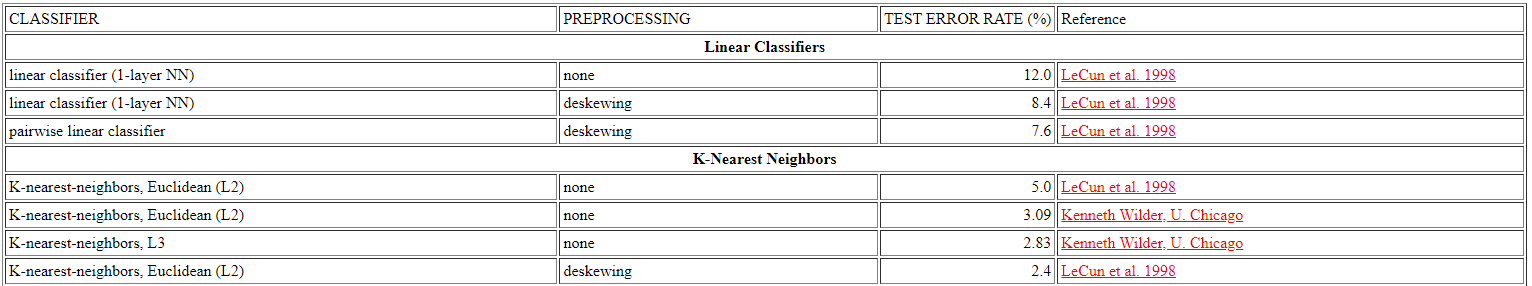
Incluso podemos encontrar un conjunto de dimensionalidades óptimas que la original, donde el error es inferior. En nuestro caso hemos obtenido que para 50 dimensiones el error se reduce a un 2.3%.

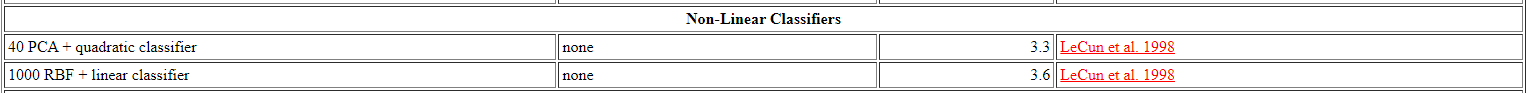
## pca+knn-eva.py

Ahora calculamos gracias a este script la tasa de error para todas las muestras de los datos MNIST. En concreto utilizaremos la dimensionalidad que, según nuestros resultados, aporta una tasa de error incluso que la original. Esta es 50 dimensiones.

Así pues, hemos realizado un experimento muy similar solo que para una dimensión y aplicando esta vez todo el conjunto de muestras de entrenamiento y las muestras de test de MNIST. Como resultado esto nos ha dado un error de 2.6597%.

### Resultados

Si comparamos este resultado con los valores presentados en la página oficial de MNIST, podremos comprobar que la tasa de error obtenida mejora significativamente a la de clasificadores lineales. También podemos observar una leve mejoría con respecto a clasificadores de k vecinos más cercanos con distancias L2 y L3. A partir de varias técnicas de preprocesado que se han llevado a cabo en los ejemplos aquí expuestos, los resultados que hemos obtenidos resultan peores en comparación, quedando por debajo de la mayoría de redes neuronales y máquinas de vectores soporte (SVMs), pero ligeramente mejor que ciertos clasificadores no lineales (cuadráticos y RBF):



# Ejercicio opcional – Algoritmo de Wilson

Hemos escogido implementar este ejercicio siguiendo las recomendaciones dadas en el anexo. Así pues, en el archivo *wilson.py* hemos realizado la implementación de los tres métodos descritos en el boletín.

## mnn

En primer lugar, realizamos *mnn*. Esta función tiene como objetivo que, dada una matriz con los datos de entrenamiento por filas *X*, un vector columna *xl* con las etiquetas de estos y el número *m* de vecinos más cercanos que queremos almacenas; calcular la matriz *V* que almacena por columnas los índices de los *m* vecinos más cercanos de cada prototipo (fila) de *X*.

El funcionamiento de esta es el siguiente:

1. Inicializamos la matriz *V* a una matriz de ceros con la dimensionalidad deseada (*m*, *nº prototipos*).
2. Para cada muestra realizamos el calculo de la distancia L2 adaptando una de las implementaciones dadas en el documento *L2dist.py* a nuestro calculo iterativo.
   1. Después de este cálculo utilizamos la función *argsort* para conseguir los índices de las distancias más pequeñas a la muestra en cuestión.
   2. Guardamos los *m* resultados más cercanos en la matriz *V*. Esto lo hacemos obviando el elemento más cercano (distancia 0, una muestra consigo misma).
3. Después de realizar el cálculo devolvemos *V* como resultado.

En cuanto al cálculo de la distancia, hemos realizado una aproximación iterativa (frente a la directa ya propuesta y más eficiente temporalmente) debido a la incapacidad de almacenar una matriz de tamaño (60000, 60000). Nuestros sistemas no han podido con esa cantidad de datos y es por ello por lo que hemos optado por sacrificar eficiencia temporal a cambio de una eficiencia espacial del algoritmo.

## knnV

Seguidamente realizamos la implementación del método *knnV*. Este nos permite devolver la clase a la cual pertenece el prototipo *i*. Esto lo hace a partir de la columna con los índices de los prototipos más cercanos *Vi*, el conjunto de índices aún disponibles de Wilson *ind*, las clases de las muestras *xl* y el número de vecinos más cercanos a tener en cuenta *k*.

El funcionamiento de este es el siguiente:

1. Filtramos en *idx* los índices de *Vi* que aún no han sido eliminados.
2. Escogemos los *k* primeros a considerar.
3. Realizamos la clasificación (siguiendo lo propuesto en el clasificador *knn*).
4. Devolvemos la clase más probable.

## wilson

Por último, nos enfrentamos a aplicar todo esto en la función *wilson* la cual implementa el algoritmo deseado. Nuestro enfoque tuvo como objetivo replicar lo visto en teoría de la forma más fidedigna. En nuestro caso esta función recoge el conjunto de muestras *X* y sus etiquetas *xl* así como el número *k* de vecinos más cercanos a tener en cuenta y devuelve los índices de las muestras

Su funcionamiento es tal que:

1. Inicializamos *ind* con los índices de todas las muestras que permanecerán en el conjunto.
2. Calculamos la matriz *V* para las muestras que
3. Realizamos lo siguiente siempre que eliminemos alguna muestra en el bucle.
   1. Para cada muestra de las disponibles comprobamos que su clase coincidiría con la más cercana por *knn*
   2. En caso de que esto no sea así la eliminamos del conjunto de muestras

## pca+knn+wilson-exp.py

Hemos realizado una implementación paralela a la del ejercicio obligatorio. Indicando el porcentaje de muestras de entrenamiento, las de test y las dimensionalidades a probar.

### Resultados

Como podemos observar en la imagen los resultados son muy similares a los obtenidos sin aplicar el algoritmo de Wilson. Aún así cabe destacar que de forma general presenta una mayor tasa de error a partir de la dimensionalidad óptima.

Esto nos permite confirmar que aún presentando una mayor una tasa de error, nos permite tener un menor número de muestras de entrenamiento (dependiendo de la dimensionalidad entre un 7.36% y un 26.74%). Esto tendrá un efecto significativo para entrenamientos con un gran número de prototipos sin sacrificar en gran medida el error de nuestro clasificador.

## pca+knn+wilson-eva.py

De nuevo una aproximación similar a la mostrada en el ejercicio obligatorio. La única diferencia en que aquí hemos permitido la posibilidad de incluir múltiples dimensionalidades.

### Resultados

Los resultados son realmente similares entre las muestras de test y las de evaluación. No hay nada especialmente a destacar sobre estos resultados aportados.