UNIVERSIDAD DE GRANADA

PRÁCTICA 3: Ajuste de Modelos Lineales

Aprendizaje Automático || Curso 2018-2019

Alumno: José María Sánchez Guerrero
DNI: 76067801Q
Correo: jose26398@correo.ugr.es
Grupo: A3 – Viernes 17:30

Contenido

1. Comprender el problema a resolver

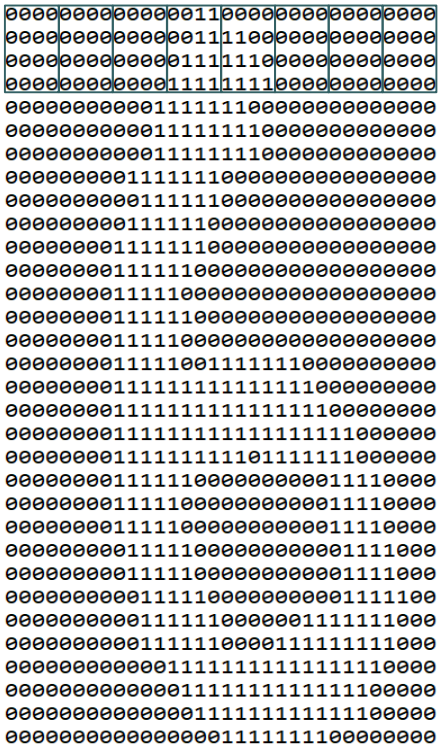
El objetivo de esta práctica será ajustar y seleccionar el mejor modelo lineal para un problema determinado, y posteriormente discutir los resultados (y el error) de éste modelo elegido. Para llevarlo a cabo, se utilizarán dos conjuntos de datos sacados del *"UCI Machine Learning Repository"*.

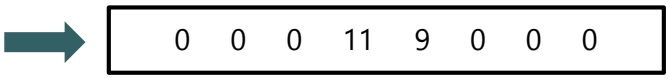
Optical Recognition of Handwritten Digits Data Set

Este conjunto de datos está formado por una serie de filas que representan un número y 65 columnas, formadas por 64 características de un número entero (del 0 al 9) que estará en la última columna de cada fila.

Los datos originalmente son mapas de bits normalizados de dígitos manuscritos de un formulario preimpreso. Estos mapas son de tamaño 32x32 y cada uno de los bits estará activo o no, dependiendo de si en el formulario está pintado o no. Estos mapas de bits se dividirán en bloques no superpuestos de 4x4 y el número de bits activos en este bloque será el número que encontramos en nuestro dataset. Al ser de este tamaño, se generarán 64 valores enteros (número de características del dataset) que irán del 0 al 16. Esto reduce la dimensionalidad y da invarianza a pequeñas distorsiones.

Veamos un ejemplo:





Si transformamos la primera fila de matrices 4x4 en números enteros como acabamos de explicar, dará este resultado.

Estos 8 valores será los primeros que aparezcan en la fila correspondiente a este número, seguido de los 54 números que completan el mapa de bits.

Obviamente, seguirán el orden lógico de izquierda a derecha y de arriba abajo.

Como hemos dicho antes, el último número será el propio dígito manuscrito (en este caso se ve claramente que es el 6).

Los propios datos del repositorio ya vienen divididos en *training* y *test*. También te proporcionan los datos originales con los mapas de bits completos y sin normalizar, sin embargo, yo voy utilizar los que ya están procesados. Estos dos ficheros se llaman 'optdigits.tra' y 'optdigits.tes' y los utilizare en un directorio 'datos' localizado en el mismo sitio que el código.

Los datos de estos archivos son:

Número de instancias:

optdigits.tra	Training	3823
optdigits.tes	Testing	1797

Distribución de clases:

Clase	Nº ejemplos training	Nº ejemplos testing
0	376	178
1	389	182
2	380	177
3	389	183
4	387	181
5	376	182
6	377	181
7	387	179
8	380	174
9	382	184

Airfoil Self-Noise Data Set

Este segundo conjunto de datos es más sencillo que el anterior. Las filas se componen sólo de 6 columnas, formadas por 5 características que determinan el nivel de presión sonora escalado, en decibelios. Este nivel será el valor de la última columna.

Los datos no están transformados ni modificados como en el ejemplo anterior, simplemente son los datos recogidos por la NASA para diferentes perfiles aerodinámicos NACA 0012. Estos perfiles comprenderán diferentes tamaños a varias velocidades en el túnel de viento y diferentes ángulos de ataque. La duración del perfil aerodinámico y la posición del observador fueron las mismas en todos los experimentos.

Veamos ahora a qué pertenece cada una de las 5 características mencionadas anteriormente, ordenadas por el número de columna al que pertenecen:

0. Frecuencia, en hercios.
1. Ángulo de ataque, en grados.
2. Longitud de la cuerda (trigonométricamente hablando), en metros.
3. Velocidad de flujo, en metros por segundo.
4. Grosor de desplazamiento lateral de aspiración, en metros.

Los datos vienen todos en un fichero 'airfoil_self_noise.dat' y, a diferencia del anterior sin dividir en *training* y *testing*. Este fichero también estará en el directorio 'datos' localizado en el mismo sitio que el código.

Los datos de estos archivos son:

Número de instancias:

airfoil_self_noise.dat	Training y testing	1503

La distribución de clases, al ser tan extensa y tener tantos valores diferentes, no la podremos mostrar en una tabla como el anterior.

2. Preprocesado de los datos

Tanto en el primer conjunto de datos como en el segundo, realizaremos unas modificaciones para que el ajuste sea mejor. Estas modificaciones será las mismas en la mayoría de los modelos, sin embargo, se realizará pruebas cambiando parámetros y otras opciones.

Se utilizará una regularización '*Ridge*' o '*l2*' para eliminar las características que no son relevantes del conjunto de datos. Es el método de regularización más utilizado para los problemas sin una solución única. Agrega una penalización equivalente al cuadrado de la magnitud de los coeficientes. A diferencia de la regularización 'Lasso' o '*l1*', no reduce a cero algunos de los coeficientes, si no que reduce los coeficientes a un valor cercano de cero.

Esta regularización también reducirá a 0 las características que son constantes, lo que significa que estas características tienen el mismo valor para todas las muestras.

Otro método que utilizaré para transformar los datos será escalar las características y luego normalizarlas:

- **Escalar.** He utilizado la función '*MaxAbsScaler()*', la cual escala cada característica por su valor máximo absoluto. Este estimador escala y traduce cada característica individualmente, de modo que el valor absoluto máximo de cada característica en el conjunto de entrenamiento será de 1. No desplaza o centra los datos, y por lo tanto no destruye ninguna dispersión.
- **Normalizar.** Simplemente aplicando la función '*normalize(X_train)*' escala las muestras individuales para tener una norma de unidad. Como ya veremos, esto es menos relevante después de realizar el escalado, no obstante, si no se realiza viene bastante bien sobre todo si planea usar una forma cuadrática como el producto puntual.

Veamos la implementación de estos dos cambios:

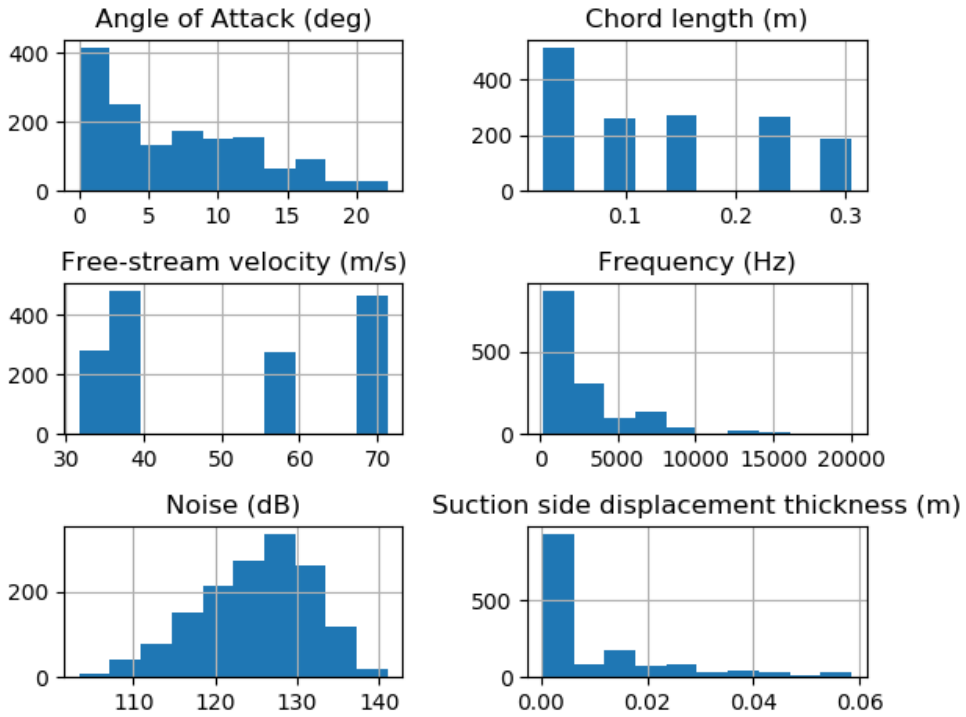
```
# Leemos a partir del fichero
df = pd.read_csv('datos/optdigits.tra')

# Guardamos en una variable auxiliar los datos pero sin el
# valor de las etiquetas
df_aux = df.copy()
df_aux = df_aux.iloc[:, :-1]

# Asignamos un escalador y lo aplicamos al conjunto de características
scaled_df = df_aux.copy()
scaler = MaxAbsScaler()
```

```
scaled = scaler.fit_transform(df_aux)
scaled_df.loc[:,:] = scaled

# Metemos en la variables el nuevo conjunto escalado
trainX_optdigits = np.array(scaled_df)[:,:-1]
trainY_optdigits = np.array(df)[:,-1:]

# Normalizamos los datos del conjunto de entrenamiento
trainX_optdigits = normalize(trainX_optdigits)
```

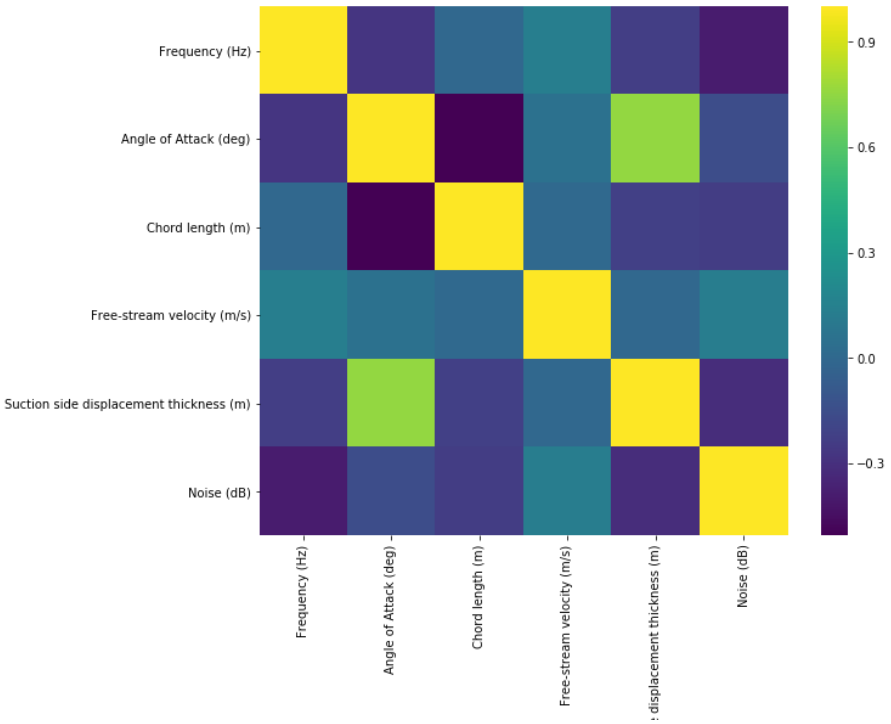
En este caso sólo lo estoy mostrando para el primer conjunto. Para el segundo también realizaremos lo mismo, lo único que cambiará será el nombre de las variables y del conjunto de datos.

Para el primer conjunto de datos, estas dos modificaciones van a ser útiles respecto a que tiene muchas características y muchas de ellas con valores constantes (o que varían muy poco) y se podrá reducir considerablemente el número de características a evaluar.

En el segundo conjunto de datos, lo de reducir características no va a ser tan relevante, puesto que cada dato sólo tiene 5. No obstante, escalar los datos va a ser muy importante por lo siguiente. Como hemos dicho antes, al existir un gran número de variables de clasificación no podíamos hacer una tabla, sin embargo, se mostrarán seis histogramas que representarán a cada una de las características del data set respectivamente.

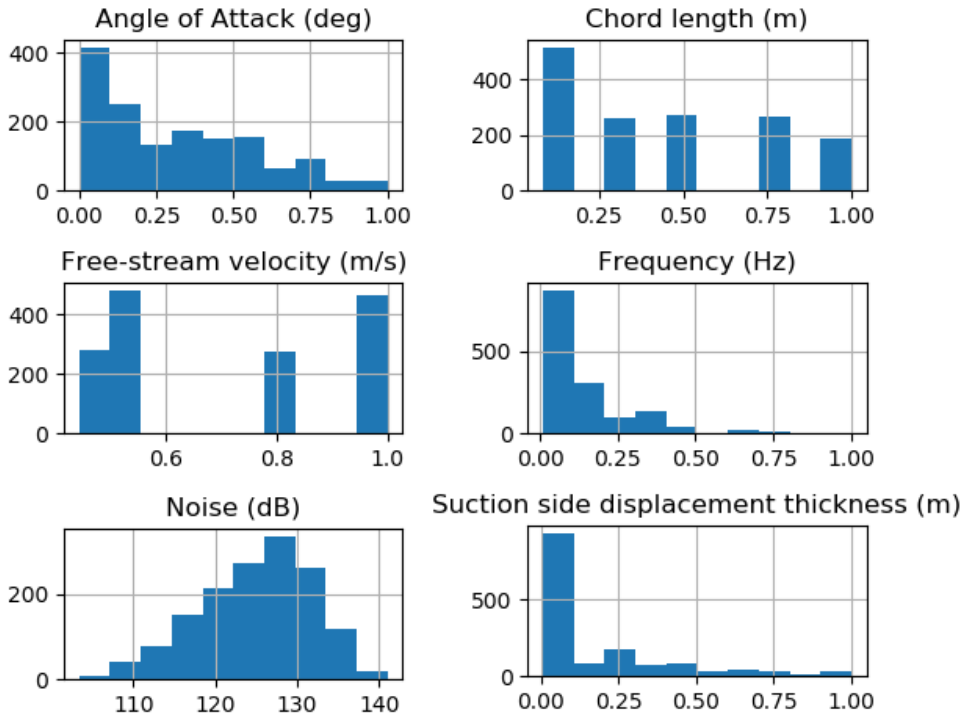


Y su matriz de correlación:



Podemos ver que no existe ninguna relación directa entre parámetros que cambie mucho el resultado final, pero en los histogramas si hemos visto que para cada característica los valores son muy dispares. Por ejemplo, los valores de frecuencia van entre 0 y 20000 mientras que los valores del ángulo de ataque entre 0 y 25.

Cuando realizamos el escalado, obtenemos los siguientes valores:



Como vemos los dos histogramas son exactamente iguales, con la única diferencia de que ahora los valores están comprendidos entre 0 y 1, lo cual facilitará mucho la regresión posterior. A estos datos después se les realizará la regularización y el ajuste que hemos comentado antes, pero eso lo veremos cuando expliquemos cada modelo.

Mientras tanto, lo que vamos a hacer con los valores ya transformados va a ser dividirlos en los conjuntos de train y test.

3. Definición de los conjuntos training, validación y test

Para el primer conjunto de datos, como la división en training y test ya está realizada no vamos a utilizar ninguna función para ello, simplemente entrenaremos y validaremos con los datos de los ficheros proporcionados.

En el segundo conjunto de datos sólo se nos proporciona un fichero con todos los datos juntos, asi que tendremos que realizar una división en entrenamiento y test. Usualmente, los datos para testear se obtienen después de los datos de entrenamiento.

Pero como no nos ofrecen otro fichero de pruebas, esta división la tendremos que realizar por nuestra cuenta. Para ello nos ayudaremos de una función de '*sklearn* ' llamada '*train_test_*split ' y cuya implementación es la siguiente:

```
# Importamos el paquete
from sklearn.model_selection import train_test_split

# Asignamos a estas variables la división en training y test
trainX_airfoil, testX_airfoil, trainY_airfoil, testY_airfoil = \
    train_test_split(data[:,:data.shape[1]-1],
                     data[:,data.shape[1]-1],
                     test_size=0.2)
```

Como podemos ver, esta función devuelve los dos conjuntos divididos y también separando en características y etiquetas. El parámetro que le pasamos, aparte de los parámetros de datos, es el tamaño que va a tener el test; y en mi caso lo he dejado con un 20% del conjunto total. El 80% restante pertenecerá será para realizar el entrenamiento.

4. Modelos lineales usados

A continuación, mostraremos los modelos lineales que se utilizarán tanto para clasificar el primer conjunto de datos como para hacer la regresión en el segundo. Por esta razón, estos modelos no serán los mismos para cada conjunto, sin embargo, he intentado utilizar los más parecidos posibles (por ejemplo, *'SGDClassifier* ' y '*SGDRegressor* ' para el primero y el segundo respectivamente).

También he elegido los modelos teniendo en cuenta los que hemos visto en clase más en profundidad y los que hemos visto/implementado en clase de prácticas. Pese a esto, todos han sido sacados de '*sklearn* ', es decir, que no he utilizado mis propias implementaciones por optimizar un poco más el trabajo.

Para mostrar los resultados utilizaremos dos funciones, una que muestre los resultados de los ejercicios de clasificación y otra para los de regresión. Las funciones no tienen mucho misterio, por lo que no pondré la implementación aquí. Para el primer caso simplemente utilizo '*metrics'* para mostrar cómo se han clasificado y su matriz de confusión; y para el segundo, utilizando '*metrics'* también, imprimo los errores MAE y MSE, y muestro una gráfica para ver cómo ha ido la regresión.

Optical Recognition of Handwritten Digits Data Set

Los modelos utilizados, pese a ser diferentes, tienen varios parámetros en común, los cuales los he puesto todos iguales para que la comparación sea justa. Posteriormente, se cambiarán estos parámetros en los mejores modelos para ver cómo afecta. Estos parámetros son los siguientes:

- **Penalty.** Hace referencia a la regularización, y en este caso voy a usar la regularización '*Ridge* ' o '*l2* ' para todos los casos.
- **Max_iter**. Número máximo de iteraciones, por defecto puesto en 15000.
- **Tol.** Referente a la tolerancia, es decir, el umbral en el que se detendrá la ejecución. Por defecto puesto en 1e-4.

Pasemos a ver ahora los modelos lineales de clasificación utilizados.

Perceptron

Se implementará el algoritmo del Perceptron. Como ya vimos en la práctica anterior, este algoritmo calcula el hiperplano que divide una muestra clasificada de forma binaria. La función se llama **Perceptron(penalty, max_iter, tol)** y sus parámetros son los que acabamos de ver.

La implementación del algoritmo es la siguiente:

```
def Perceptron(penality, max iter, tol):
    print('\n\n\nPERCEPTRON')

    # Creamos el clasificador y lo asignamos a una variable
    classifier = linear_model.Perceptron(penalty=penality,
                                         max_iter=max_iter, tol=tol)

    # Entrenamos nuestro clasificador con los datos de entrenamiento
    classifier.fit(trainX_optdigits, trainY_optdigits.ravel())

    # Creamos dos variables: valor esperado y el que predecimos
    # gracias al vector de entrenamiento
    expected = testY_optdigits
    predicted = classifier.predict(testX_optdigits)

    analisisClasificacion(expected, predicted)

Perceptron('l2', 15000, 1e-4)
```

Regresión Logística

Algoritmo que también vimos en la práctica anterior en el que se utiliza el gradiente descendente para hacer la clasificación, pero con unas cuantas diferencias. La función se llama **RegresionLogistica(penality, max_iter)** y, pese a que tiene más parámetros, son los que vienen por defecto en la implementación de '*sklearn*'. Yo los he puesto para evitar unos *warnings* en la ejecución del mismo. La implementación del algoritmo es la siguiente:

```
def RegresionLogistica(penality, max_iter):
    print('\n\n\nREGRESION LOGISTICA')

    # Creamos el clasificador y lo asignamos a una variable
    classifier = linear_model.LogisticRegression(penalty=penality,
max_iter=max_iter, solver='liblinear', multi_class='ovr')

    # Entrenamos nuestro clasificador con los datos de entrenamiento
    classifier.fit(trainX_optdigits, trainY_optdigits.ravel())

    # Creamos dos variables: valor esperado y el que predecimos
    # gracias al vector de entrenamiento
    expected = testY_optdigits
    predicted = classifier.predict(testX_optdigits)

    # Funcion que muestra por pantalla los resultados
    analisisClasificacion(expected, predicted)

RegresionLogistica('l2', 15000)
```

Gradiente Descendente Estocástico (SGD)

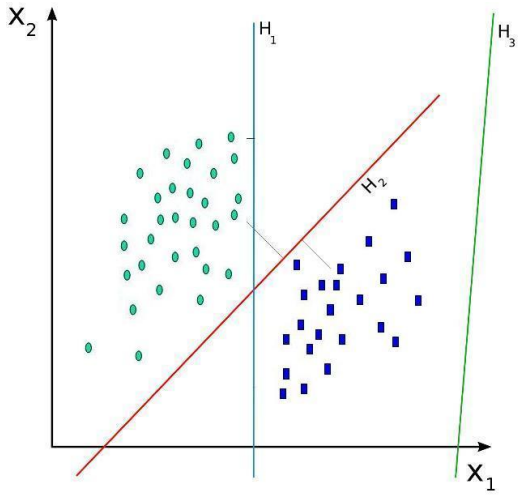
Este algoritmo sirve para minimizar funciones el cual empieza en un punto y va descendiendo por la pendiente más pronunciada. Para realizarlo no se coge la muestra completa, si no una parte de ella llamada *minibatch.* La función se llama **SGDClassifier(penality, max_iter, alpha, tol)** y tiene un parámetro extra llamado *alpha* el cual será la tasa de aprendizaje del algoritmo. La he puesto tal y como viene por defecto a 0.001. La implementación del algoritmo es la siguiente:

```
def SGDClassifier(penality, max_iter, alpha, tol):
    print('\n\n\nGRADIENTE DESCENDENTE ESTOCASTICO (Classifier)')

    # Creamos el clasificador y lo asignamos a una variable
    classifier = linear_model.SGDClassifier(penalty=penality,
max_iter=max_iter, alpha=alpha, tol=tol)

    # Entrenamos nuestro clasificador con los datos de entrenamiento
    classifier.fit(trainX_optdigits, trainY_optdigits.ravel())

    # Creamos dos variables: valor esperado y el que predecimos
    # gracias al vector de entrenamiento
    expected = testY_optdigits
    predicted = classifier.predict(testX_optdigits)

    # Funcion que muestra por pantalla los resultados
    analisisClasificacion(expected, predicted)

SGDClassifier('l2', 15000, 0.001, 1e-4)
```

Support Vector Machine (SVM)

Por último, vamos a utilizar este algoritmo que no habíamos visto en prácticas anteriores. Este algoritmo busca un hiperplano que separe de forma óptima a los puntos de una clase de la de otra. Al decir de forma óptima nos referimos a que busca el hiperplano que tenga la máxima distancia (margen) con los puntos que estén más cerca de él mismo.

Para tener más claro lo que hace este algoritmo, mostraremos una imagen de cómo separa los datos dejando el pasillo más grande posible:



La función se llama **LinearSVC(penalty, C, max_iter, tol)** y tiene un parámetro extra llamado *C* que será el error permitido para este pasillo, es decir, contra más pequeño sea el valor, mayor error va a permitir y más pequeño será el pasillo que divide los datos.

La implementación del algoritmo es la siguiente:

```
def LinearSVC(penality, C, max_iter, tol):
    print('\n\n\nSUPPORT VECTOR MACHINE (SVC)')

    # Creamos el clasificador y lo asignamos a una variable
    classifier = svm.LinearSVC(penalty=penality, C=C,
                               max_iter=max_iter, tol=tol)

    # Entrenamos nuestro clasificador con los datos de entrenamiento
    classifier.fit(trainX_optdigits, trainY_optdigits.ravel())

    # Creamos dos variables: valor esperado y el que predecimos
    # gracias al vector de entrenamiento
    expected = testY_optdigits
    predicted = classifier.predict(testX_optdigits)

    # Funcion que muestra por pantalla los resultados
    analisisClasificacion(expected, predicted)

LinearSVC('l2', 1.0, 15000, 1e-4)
```

Airfoil Self-Noise Data Set

En los siguientes algoritmos de regresión, como los parámetros son bastante distintos entre unos y otros, los explicaré por separado. Aun así, cuando se elijan los mejores modelos, también cambiaremos algunos parámetros para ver cómo afectaban al algoritmo.

Regresión Lineal

Desde el punto de vista de la implementación, esto es simplemente mínimos cuadrados ordinarios que hacen la función de predecir los valores. Por esta razón tampoco tiene parámetros. La función se llama **RegresionLineal().** La implementación del algoritmo es la siguiente:

```
def RegresionLineal():
    print('\n\n\nREGRESION LINEAL')
    # Creamos el regresor y lo asignamos a una variable
    regressor = linear_model.LinearRegression()

    # Lo entrenamos con los datos de entrenamiento
    regressor.fit(trainX_airfoil, trainY_airfoil.ravel())

    # Creamos dos variables: valor esperado y el que predecimos
    # gracias al vector de entrenamiento
    expected = testY_airfoil
    predicted = regressor.predict(testX_airfoil)

    # Funcion que muestra por pantalla los resultados
    analisisRegresion(expected, predicted)

RegresionLineal()
```

Gradiente Descendente Estocástico (SGD)

Este algoritmo es como el que hemos visto para clasificación, ya que el interior del algoritmo (lo que hace) es igual para los dos, pero este adaptado para regresión. La función se llama **SGDRegressor(penality, max_iter, alpha, tol)** y cómo podemos observar, los parámetros son los mismo que teníamos antes. La implementación del algoritmo es la siguiente:

```
def SGDRegressor(penality, max_iter, alpha, tol):
    print('\n\n\nGRADIENTE DESCENDENTE ESTOCASTICO')
    # Creamos el regresor y lo asignamos a una variable
    regressor = linear_model.SGDRegressor(penalty=penality,
                        max_iter=max_iter, alpha=alpha, tol=tol)

    # Lo entrenamos con los datos de entrenamiento
    regressor.fit(trainX_airfoil, trainY_airfoil.ravel())

    # Creamos dos variables: valor esperado y el que predecimos
    # gracias al vector de entrenamiento
    expected = testY_airfoil
    predicted = regressor.predict(testX_airfoil)

    # Funcion que muestra por pantalla los resultados
    analisisRegresion(expected, predicted)

SGDRegressor('l2', 15000, 0.001, 1e-4)
```

Support Vector Machine (SVM)

Como en el algoritmo anterior, éste también es el mismo que hemos visto en clasificación, pero adaptado a regresión. La función se llama **LinearSVR(C, max_iter, tol)** y a diferencia del anterior, éste no tiene el parámetro de regularización, ya que es orientado a regresión.

La implementación del algoritmo es la siguiente:

```
def LinearSVR(C, max_iter, tol):
    print('\n\n\nSUPPORT VECTOR MACHINE (SVR)')

    # Creamos el regresor y lo asignamos a una variable
    regressor = svm.LinearSVR(C=C, max_iter=max_iter, tol=tol)

    # Lo entrenamos con los datos de entrenamiento
    regressor.fit(trainX_airfoil, trainY_airfoil.ravel())

    # Creamos dos variables: valor esperado y el que predecimos
    # gracias al vector de entrenamiento
    expected = testY_airfoil
    predicted = regressor.predict(testX_airfoil)

    # Funcion que muestra por pantalla los resultados
    analisisRegresion(expected, predicted)

LinearSVR(1.0, 15000, 1e-4)
```

Estos son todos los modelos lineales implementados. A continuación, pasemos al análisis de los resultados obtenidos al ejecutarlos.

5. Análisis de los resultados obtenidos por cada modelo

Ahora se pasará a analizar el resultado de cada método mediante los distintos valores que mostramos tanto por consola como a través de gráficas. El orden que seguiremos al analizarlos será el mismo que hemos utilizado para describirlos en el apartado anterior.

Optical Recognition of Handwritten Digits Data Set

Para los siguientes modelos de clasificación vamos a sacar por consola los siguientes datos:

- **Informe de clasificación.** Muestra una tabla con distintos parámetros sobre cómo ha clasificado para cada uno de los números. Al final de esta tabla tendremos los 3 valores más importantes, que son: *micro_avg*, que representa la media del total de verdaderos positivos, falsos negativos y falsos positivos; *macro_avg*, que representa la media no ponderada de todas las etiquetas; y *weighted_avg*, que representa la media anterior ponderada.
- **Matriz de confusión.** Muestra otra tabla que contiene la matriz de confusión, con la cual podremos ver cuántos valores han sido bien clasificados y ver dónde ha metido a los que no ha clasificado bien.

Perceptron	precision	recall	f1-score	support
0	0.97	0.95	0.96	177
1	0.67	0.97	0.79	182
2	0.97	0.96	0.97	177
3	0.90	0.95	0.92	183
4	0.95	0.86	0.90	181
5	0.84	0.98	0.91	182
6	0.99	0.93	0.96	181
7	0.95	0.94	0.95	179
8	0.94	0.67	0.79	174
9	0.92	0.75	0.83	180
micro avg	**0.90**	**0.90**	**0.90**	**1796**
macro avg	**0.91**	**0.90**	**0.90**	**1796**
weighted avg	**0.91**	**0.90**	**0.90**	**1796**

Perceptron *Matriz de confusión*

169	0	0	0	1	7	0	0	0	0
0	176	0	0	0	1	0	0	0	5
0	4	170	1	0	0	0	2	0	0
0	0	2	173	0	2	0	4	0	2
0	23	0	0	155	0	0	1	1	1
1	2	0	0	0	179	0	0	0	0
0	11	0	0	2	0	168	0	0	0
0	1	0	0	0	6	0	169	1	2
1	38	2	3	0	9	1	2	117	1
4	6	1	15	6	8	0	0	5	135

Regresión Logística	precision	recall	f1-score	support
0	1.00	0.98	0.99	177
1	0.89	0.91	0.90	182
2	0.98	0.98	0.98	177
3	0.98	0.92	0.95	183
4	0.96	0.97	0.96	181
5	0.91	0.98	0.94	182
6	0.98	0.97	0.98	181
7	0.97	0.92	0.95	179
8	0.88	0.86	0.87	174
9	0.86	0.92	0.89	180
micro avg	**0.94**	**0.94**	**0.94**	**1796**
macro avg	**0.94**	**0.94**	**0.94**	**1796**
weighted avg	**0.94**	**0.94**	**0.94**	**1796**

Regresión Logística *Matriz de confusión*

173	0	0	0	1	3	0	0	0	0
0	165	3	0	0	0	0	0	6	8
0	1	173	1	0	0	0	1	1	0
0	0	0	169	0	3	0	3	3	5
0	3	0	0	175	0	0	1	2	0
0	0	1	0	0	178	1	0	0	2
0	2	0	0	2	0	176	0	1	0
0	0	0	0	1	7	0	165	1	5
0	13	0	1	0	3	2	0	149	6
0	1	0	1	4	2	0	0	6	166

SGD Clas	*precision*	*recall*	*f1-score*	*support*
0	1.00	0.97	0.99	177
1	0.94	0.90	0.92	182
2	0.95	0.98	0.96	177
3	0.97	0.93	0.95	183
4	0.95	0.98	0.96	181
5	0.90	0.98	0.94	182
6	0.99	0.98	0.98	181
7	0.98	0.91	0.94	179
8	0.89	0.87	0.88	174
9	0.88	0.91	0.89	180
micro avg	**0.94**	**0.94**	**0.94**	**1796**
macro avg	**0.94**	**0.94**	**0.94**	**1796**
weighted avg	**0.94**	**0.94**	**0.94**	**1796**

SGD Classifier *Matriz de confusión*

172	0	0	0	1	4	0	0	0	0
0	164	5	0	0	0	0	0	5	8
0	1	174	1	0	0	0	1	0	0
0	0	2	171	0	3	0	2	2	3
0	0	0	0	177	0	0	1	3	0
0	0	1	0	0	178	1	0	0	2
0	1	0	0	2	0	177	0	1	0
0	0	0	0	2	7	0	163	2	5
0	8	1	3	1	4	1	0	151	5
0	1	1	2	4	2	0	0	6	164

SVC	precision	recall	f1-score	support
0	1.00	0.99	0.99	177
1	0.92	0.93	0.93	182
2	0.99	0.98	0.99	177
3	0.96	0.95	0.95	183
4	0.97	0.97	0.97	181
5	0.92	0.98	0.95	182
6	0.99	0.98	0.98	181
7	0.98	0.93	0.95	179
8	0.91	0.88	0.89	174
9	0.89	0.94	0.92	180
micro avg	**0.95**	**0.95**	**0.95**	**1796**
macro avg	**0.95**	**0.95**	**0.95**	**1796**
weighted avg	**0.95**	**0.95**	**0.95**	**1796**

SVC Matriz de confusión

175	0	0	0	0	2	0	0	0	0
0	170	0	0	0	0	0	0	6	6
0	1	173	1	0	0	0	1	1	0
0	0	0	173	0	3	0	2	1	4
0	2	0	0	175	0	0	1	3	0
0	0	1	1	0	179	1	0	0	0
0	1	0	0	2	0	177	0	1	0
0	0	0	0	1	7	0	166	1	4
0	9	0	3	0	2	1	0	153	6
0	1	0	3	2	2	0	0	2	170

Como podemos ver los resultados de todos los modelos son bastante buenos (con una media de acierto del más del 90% para todos). Entre todos ellos, el que mejor clasifica los datos es el *Support Vector Machine*, con un 95% de media; aunque tanto la regresión lineal como el *SGD* tienen un 94% de media, el cual es un valor muy cercano al del SVM.

Viendo las matrices de confusión y comparándola con la clasificación, podemos analizar más en profundidad la clasificación.

Por ejemplo, en el perceptron, pese a tener una buena media (91%), si nos fijamos en la clasificación del número 1, sólo un 67% de precisión. Si nos vamos a su matriz de confusión, podemos ver que estos valores los confunde sobre todo con el 4 (23 datos) y con el 8 (38 datos).

En los siguientes modelos vemos que también hay errores, pero no son tan grandes como en este. Todos los valores tienen más de un 85% de precisión, llegando algunos incluso a tener una efectividad del 100% (por ejemplo, el propio número 0 clasificado con SMV).

Airfoil Self-Noise Data Set

Para los siguientes modelos de regresión vamos a sacar por consola los siguientes datos:

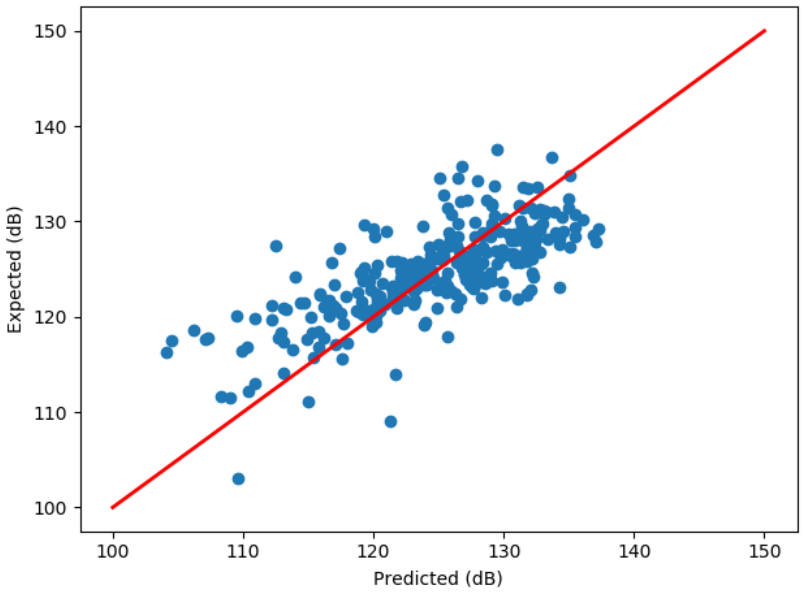
- **Mean Absolut Error (MAE).** Como el propio nombre indica, el MAE es una media de los errores absolutos $e_i = |y_i - x_i|$, dónde y_i es la predicción y x_i es el valor esperado.
- **Mean Square Error (MSE).** Es una media de los errores absolutos pero esta vez al cuadrado, es decir, $e_i = |y_i - x_i|^2$.

Y también vamos a mostrar un gráfico de cómo ha ido la regresión, siendo los valores del eje X los que hemos predicho y los del eje Y los esperados. Esto nos servirá para ver gráficamente cómo ha ido la predicción. Si la predicción fuese perfecta, todos los puntos estarían sobre la línea (el valor predicho es el mismo que el esperado).

REGRESION LINEAL:

Mean Absolut Error: 3.5942918794749463

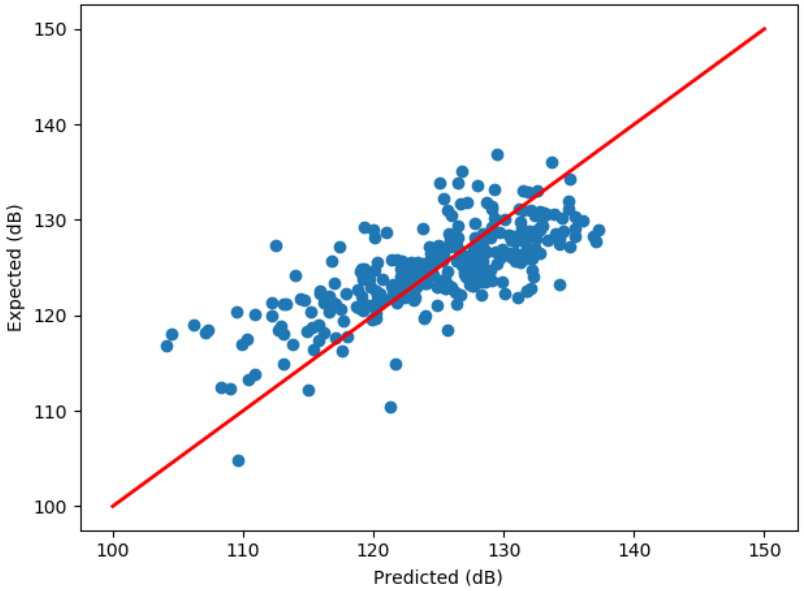
Mean Squared Error: 21.366053724621846



SGD Regressor:

Mean Absolut Error: 3.628813336168349

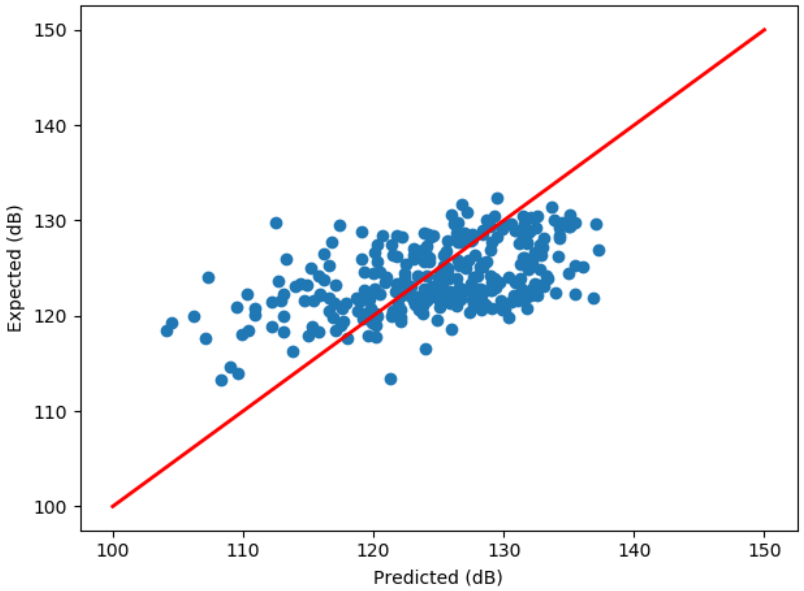
Mean Squared Error: 21.508934732480917



SVR:

Mean Absolut Error: 4.713543998529675

Mean Squared Error: 34.21430312574688



A diferencia de los modelos de regresión, en este la regresión lineal y el Gradiente Descendente estocástico ofrecen mejores resultados que el *Support Vector Machine*. Pese a esto, los errores no son excesivamente grandes en ninguno de los tres casos.

Observando los gráficos, vemos cómo el valor de los errores se cumple. Con un simple vistazo podemos ver que los puntos en regresión lineal y en el SGD (bastante parecidos los dos) se aproximan mucho más a la línea que en el SVM.

Ahora vamos a proceder a cambiar varios parámetros de los modelos. Como en la clasificación el mejor ha sido el SVC, intentaremos mejorar el SGD para ver si puede conseguir un acierto igual o superior. También aquí vamos a probar a cambiar las regularizaciones *Ridge* y *Lasso*.

Por otra parte, para el modelo de regresión haremos todo lo contrario. Intentaremos modificar los parámetros del SVR para que pueda conseguir un error tan bajo como el de los anteriores. También vamos a poner parámetros que empeoren el modelo, para comprobar la importancia que tiene no poner unos parámetros correctos a la hora de ejecutar.

Estas será las 4 funciones que ejecutaremos:

- SGDClassifier('l1', 15000, 0.001, 1e-4) # Cambiamos regularización
- SGDClassifier('l2', 15000, 0.0001, 1e-5) # Ponemos unos alpha y tolerancia menor
- LinearSVR(5.0, 15000, 1e-5) # Aumentamos el valor de C y menos tolerancia
- LinearSVR(0.1, 1000, 1e-3) # Disminuimos C y más tolerancia

Vamos a mostrar únicamente los valores de las medias (en caso de ver los resultados completamente se puede hacer en el script).

Para el SGD con la regularización cambiada obtenemos lo siguiente:

micro avg	0.94	0.94	0.94	1796
macro avg	0.94	0.94	0.94	1796
weighted avg	0.94	0.94	0.94	1796

El ajuste sigue siendo bueno (los valores son exactamente iguales a los anteriores), lo que quiere decir que es importante hacer la regularización, pero tanto decidir entre *Ridge* o *Lasso,* ya que ambas funcionan ambas bastante bien.

Ahora pasemos al SGD con parámetros más restrictivos y que supuestamente deberían mejorar el resultado:

micro avg	0.94	0.94	0.94	1796
macro avg	0.94	0.94	0.94	1796
weighted avg	0.94	0.94	0.94	1796

Podemos ver que el resultado sigue siendo el mismo, pese a poner una menor tolerancia y poniendo una tasa de aprendizaje menor. Esto nos demuestra que, pese a mucho ajustar, y como consecuencia ganar tiempo de cómputo, quizás no sea suficiente si el algoritmo no da para más.

Continuemos con los problemas de regresión. Empecemos por intentar mejoras el SVR a ver si llega al nivel del SGD:

Mean Absolut Error: 3.5912036941594256

Mean Squared Error: 21.737617988279744

Como podemos observar, con unos buenos parámetros hemos conseguido mejorar tanto el SGD (que tenía un error de aproximadamente 3.62) e incluso de la regresión lineal que nos ofrecía el error más bajo (aproximadamente 3.594).

Vamos a ver la otra cara de la moneda, es decir, metiéndole los parámetros malos al modelo:

Mean Absolut Error: 14.562038400529232

Mean Squared Error: 308.63523412611977

De este modo vemos como el error se nos va por las nubes. Esto es bastante representativo por lo que se ha dicho anteriormente, y es que los parámetros que le asignamos a la función determinan si hacemos una regresión buena o mala. Podemos pasar de tener el mejor algoritmo a uno que ni se acerca a los errores de los demás.

6. Modelo no lineal

Para ver cómo de buenos son estos modelos respecto a otros no lineales, he añadido uno al final del script. Este será el mismo tanto para clasificación como para regresión. Se trata del modelo KNeighbors, que realiza tanto la regresión como la clasificación utilizando la distancia euclídea con los *k* vecinos más cercanos.

Estos son los resultados obtenidos:

kNN	*precision*	*recall*	*f1-score*	*support*
0	1.00	1.00	1.00	177
1	0.93	1.00	0.97	182
2	0.99	0.98	0.99	177
3	0.98	0.96	0.97	183
4	0.99	0.99	0.99	181
5	0.98	0.99	0.99	182
6	1.00	1.00	1.00	181
7	0.99	0.97	0.98	179
8	0.96	0.94	0.95	174
9	0.96	0.97	0.96	180
micro avg	**0.98**	**0.98**	**0.98**	**1796**
macro avg	**0.98**	**0.98**	**0.98**	**1796**
weighted avg	**0.98**	**0.98**	**0.98**	**1796**

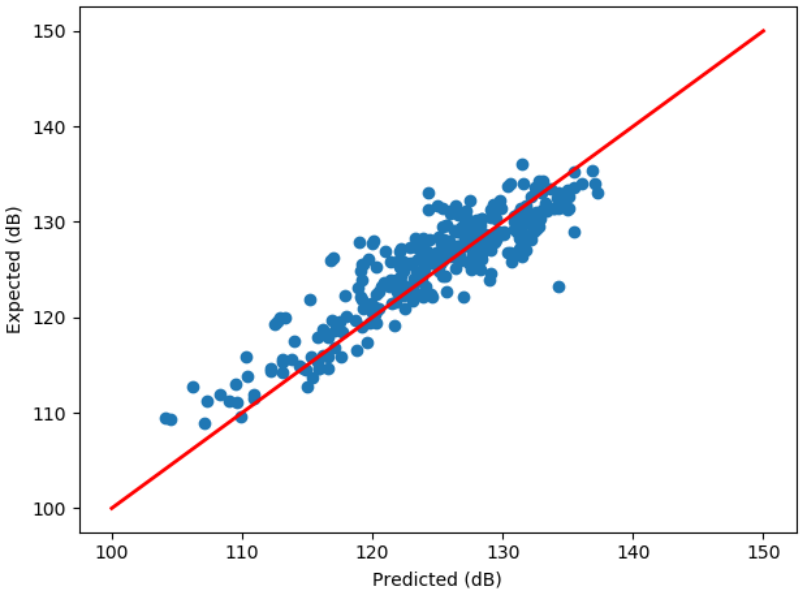
Como vemos la clasificación es mejor que en el modelo lineal, pero tampoco mucho mejor. Hay que tener en cuenta también que los modelos lineales ya tenían una tasa de acierto bastante alta, asi que es complicado decir cómo de bueno es con respecto a ellos.

Ahora veamos los resultados que ofrece para el de regresión:

kNN:

Mean Absolut Error: 2.41684318936877

Mean Squared Error: 9.583148304186041



Podemos ver que en este problema la mejora es más significativa que en el problema de clasificación. A simple vista, viendo la gráfica ya podemos ver que los puntos están mucho más cercanos a la línea que antes, pero comparando los valores que tienen los errores (sobre todo el MSE) vemos que este modelo es bastante mejor.

Podemos concluir que, pese a que los modelos lineales lo hacen bastante bien, los modelos no lineales son mejores en estas tareas. En mi caso, este ha sido el único modelo que he probado y con los parámetros por defecto. Estoy seguro que si hubiese ajustado más o probado algún otro modelo no lineal los resultados serían todavía más abultados.