Aprendizaje Automático

Universidad de Granada

Trabajo 3

Ajuste de Modelos Lineales

Ignacio Vellido Expósito

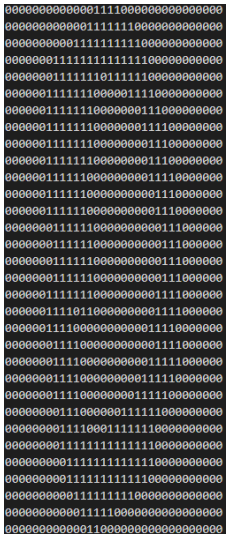
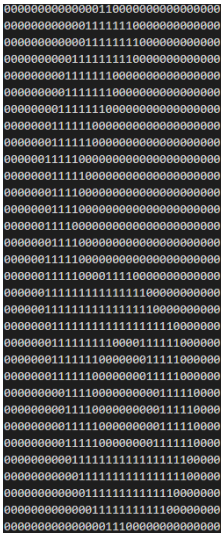
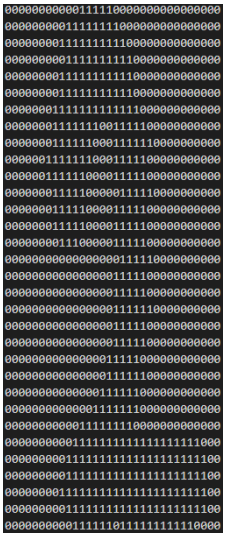
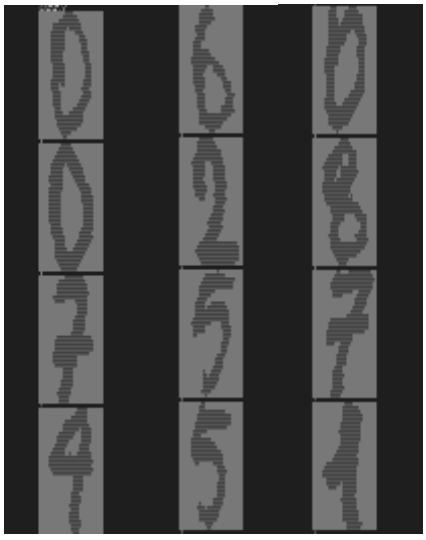
Grupo Viernes

Problema de clasificación

Comprensión del problema

Contamos con una base de datos de dígitos escritos (entre 0 y 9). Se nos proporciona un archivo en el que, partiendo una imagen de un dígito en 64 trozos, se cuenta el número de píxeles en cada trozo y se mide su intensidad. Al final de cada línea se añade la clase a la que pertenece.

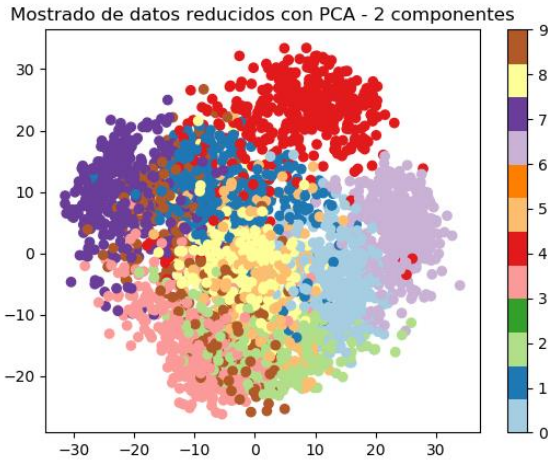
Datos no procesados:

Datos procesados:

```
0,1,6,15,12,1,0,0,0,7,16,6,6,10,0,0,0,8,16,2,0,11,2,0,0,5,16,3,0,5,7,0,0,7,13,3,0,8,7,0,0,4,12,0,1,13,5,0,0,0,14,9,15,9,0,0,0,0,6,14,7,1,0,0,0
0,0,10,16,6,0,0,0,0,7,16,8,16,5,0,0,0,11,16,0,6,14,3,0,0,12,12,0,0,11,11,0,0,12,12,0,0,8,12,0,0,7,15,1,0,13,11,0,0,0,16,8,10,15,3,0,0,0,10,16,15,3,0,0,0
0,0,8,15,16,13,0,0,0,0,1,11,9,11,16,1,0,0,0,0,0,7,14,0,0,0,0,3,4,14,12,2,0,0,1,16,16,16,16,10,0,0,2,12,16,10,0,0,0,0,0,2,16,4,0,0,0,0,0,9,14,0,0,0,0,7
0,0,0,3,11,16,0,0,0,0,5,16,11,13,7,0,0,3,15,8,1,15,6,0,0,11,16,16,16,16,10,0,0,1,4,4,13,10,2,0,0,0,0,0,15,4,0,0,0,0,0,3,16,0,0,0,0,0,0,1,15,2,0,0,4
0,0,5,14,4,0,0,0,0,0,13,8,0,0,0,0,0,3,14,4,0,0,0,0,0,6,16,14,9,2,0,0,0,4,16,3,4,11,2,0,0,0,14,3,0,4,11,0,0,0,10,8,4,11,12,0,0,0,4,12,14,7,0,0,6
0,0,11,16,10,1,0,0,0,4,16,10,15,8,0,0,0,4,16,3,11,13,0,0,0,1,14,6,9,14,0,0,0,0,0,0,12,10,0,0,0,0,0,6,16,6,0,0,0,0,5,15,15,8,8,3,0,0,10,16,16,16,16,6,2
```

Nuestro objetivo es poder aprender de estos datos y poder clasificar nuevos dígitos que nos vengan en este mismo formato.



Preprocesado de datos

Podríamos probar sacando dos características como la intensidad y simetría, y clasificar en base a estas. Pese a que esa idea nos puede venir muy bien para diferenciar algunos dígitos (como el 1 del 0) en otros tengo la sensación de que puede clasificar incorrectamente. Por tanto, nos basamos en las columnas proporcionadas para realizar el aprendizaje.

Como contamos con un número alto de variables (64), la dimensionalidad es mayor que lo que queremos, por lo que aplicamos varias técnicas para reducirla:

Primeramente, a partir de las imágenes se aprecia que en general algunas columnas contienen un valor constante 0, por lo que comprobamos y las quitamos. De cara al problema esto tiene sentido ya que por la forma de los dígitos algunas zonas del papel no suelen pintarse independientemente del número que sea.

```
# Buscamos las columnas
distintas = np.zeros((1, len(train_x[0])))

for row in train_x:
  distintas += row

distintas /= len(train_x)

insignificantes = np.zeros(np.shape(distintas))

for i in range(len(distintas[0])):
  if (distintas[0][i] < 0.5):
    insignificantes[0][i] = 1
  else:
    insignificantes[0][i] = 0
```

```
# Quitamos las de menos variabilidad
train_x = np.delete(train_x, np.where(insignificantes == 1), 1)
test_x = np.delete(test_x, np.where(insignificantes == 1), 1)
```

```
Variación de las columnas:
[[ 0.          0.30133403  5.48182056 11.80591159 11.4514779   5.50536228
   1.3873921   0.14229663  0.0020926   1.96050222 10.57729532 11.71540675
  10.62490191  8.29557939  2.20010463  0.15197489  0.00496992  2.59586712
   9.58069579  6.73502485  7.18650275  8.04839132  2.04603714  0.04917604
   0.0010463   2.33560031  9.23907926  9.13366466  9.67329323  7.86764321
   2.34030866  0.0031389   0.00130787  2.04289825  7.65942977  9.23803296
  10.34763275  9.20010463  2.91263406  0.          0.02746534  1.40570233
   6.45670939  7.18728747  7.9215276   8.67486267  3.5103322   0.01987968
   0.01778708  0.82003662  7.86895109  9.88569187  9.76484436  9.28328538
   3.74391839  0.14831284  0.00026157  0.2830238   5.85587235 11.94297672
  11.46115616  6.70049699  2.10567617  0.20219723]]

Columnas con poca variabilidad:
[[1. 0. 0. 0. 0. 0. 0. 0. 1. 0. 0. 0. 0. 0. 0. 0. 1. 0. 0. 0. 0. 0. 0. 0.
  1. 0. 0. 0. 0. 0. 0. 1. 1. 0. 0. 0. 0. 0. 0. 1. 0. 0. 0. 0. 0. 0. 0. 0.
  0. 0. 0. 0. 0. 0. 0. 0. 1. 0. 0. 0. 0. 0. 0. 0.]]
```

Como vamos a emplear un modelo cuadrático, para acelerar el proceso de aprendizaje normalizamos el conjunto de datos. Aunque el rango de valores que aceptan los datos es pequeño (está entre 0 y 16), situarlos en el rango 0-1 facilita la convergencia del modelo.

```
scaler = preprocessing.StandardScaler().fit(train_x)
train_x = scaler.transform(train_x)
test_x = scaler.transform(test_x)
```

Seguimos teniendo demasiadas columnas, por lo que aplicamos PCA.

```
pca = PCA(n_components=0.8, svd_solver="full")
pca.fit(train_x)
train_x = pca.transform(train_x)
test_x = pca.transform(test_x)
```

Este método nos reduce las columnas de a 15, consiguiendo una gran mejora en el tiempo de cómputo del aprendizaje.

Selección de la clase de funciones

En una primera instancia no se ha realizado transformaciones a la clase de funciones de los datos, y al haber obtenido buenos resultados con ella se ha decido mantenerla.

Definición de conjuntos de datos

Pese a que se nos proporciona la separación de datos en *training* y *test*, hemos visto que una técnica más profesional y generalmente con mejores resultados es *cross-validation*, que es la que empleamos en este caso.

Este procedimiento no es totalmente ventajoso, pues la iteración a través de las distintas particiones de los conjuntos origina un coste computacional mayor. Aunque al haber reducido dimensionalidad y no contar con un conjunto de datos excesivamente grande proseguimos con el método.

Para tener una medida exacta de la calidad del clasificador mantenemos el conjunto de *test* que nos proporcionan. Se podría usar *CV* con el conjunto entero de datos y considerar la media de errores como la medida, pero ya que sin el conjunto de *test* se consiguen buenos resultados mantenemos la separación.

Como generalmente está considerado k=10 un buen número de particiones, se utiliza este valor. Tras ajustar probando con otro tipo de tamaños no se han conseguido diferencias significativas y se ha mantenido este valor.

Regularización

Por lo visto en clase de teoría aplicar regularización nunca es una mala idea, puesto que en el peor de los casos el valor del parámetro será 0 indicando que no es necesario ninguna cantidad de regularización, y en el caso de que sea mayor, estamos evitando el sobreajuste en la muestra.

Siguiendo esta mentalidad, se implementa un bucle en el que se lanza *CV* con diferentes valores en el parámetro de regularización, eligiendo aquel que nos proporcione mejores resultados en la muestra.

Modelo usado

Se descarta el perceptrón ya que es un clasificador binario, y en nuestro caso contamos con 10 clases. Se podría entrenar 10 perceptrones (*one-vs-all*), utilizando cada uno para diferenciar cada dígito, pero el problema se daría si varios perceptrones afirman que el dato pertenece a su clase, lo que obligaría a asignar probabilidades y complicaría la clasificación.

También se podría utilizar el perceptrón multicapa, pero no es un clasificador lineal y por tanto no entra en el objetivo de la práctica.

Por tanto, se ha optado por un modelo de **regresión logística** ya que estamos tratamos con un problema de clasificación.

Ajuste

En un principio se utiliza una tolerancia de **10^{-4}** y un máximo de iteraciones de **10.000**, el resto de parámetros no mencionados (*learning_rate, factor de regularización* y *función de pérdida*) se mantienen a los que ofrece por defecto *Scikit-Learn*, puesto que son los recomendados por la librería.

El proceso consiste en calcular el mejor parámetro de regularización mediante *CV* para luego ajustarlo con el conjunto de entrenamiento al completo. Al tener un número mayor de datos, es probable (pero no seguro) que obtengamos el mejor ajuste de los modelos probados.

```
# Ajuste
"""
alpha    - Para regularización
max_iter - Número máximo de iteraciones
tol      - Criterio de parada
"""
tol = 1e-4
max_iter = 10_000

best_classifier = 0
best_scores = 0
best_media = 0

for i in range(3):
  alpha = 0.0001 * (10 ** i)
  classifier = SGDClassifier(max_iter=max_iter, tol=tol, alpha=alpha,
                             loss="log", penalty="none")

  scores = cross_val_score(classifier, train_x, train_y, cv=10)

  media = scores.mean()

  # Nos quedamos con la mejor
  if media > best_media:
    best_classifier = classifier
    best_scores = scores
    best_media = media


best_classifier.fit(train_x, train_y)
```

Posteriormente se han realizado pruebas con diferentes parámetros y nos hemos quedado con aquel que nos devuelve el valor más alto de puntuación fuera de la muestra.

Tolerancia = **10^{-4}**

```
Porcentaje de aciertos en train: 0.9539628563955009
Porcentaje de aciertos en test:  0.9220923761825265
```

Tolerancia = **10^{-7}**

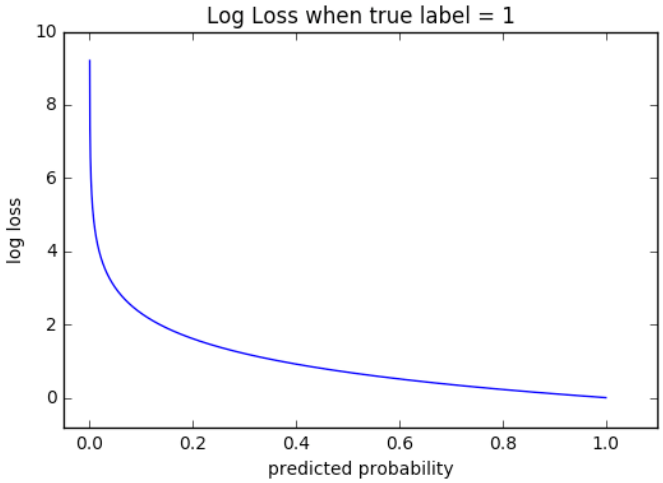
```
Porcentaje de aciertos en train: 0.9450693172900864
Porcentaje de aciertos en test:  0.9165275459098498
```

Learning rate manual a **0,01** y tolerancia = **10^{-4}**

```
Porcentaje de aciertos en train: 0.9479466387653676
Porcentaje de aciertos en test:  0.910962715637173
```

Métrica usada

Scikit-Learn utiliza la función de pérdida cuando se le pone regresión logística a SGD. Lo bueno de esta función es que es monótona decreciente, facilitando la convergencia.



Estimación del error

Por un lado, tenemos las puntuaciones en cada partición de *CV* y la mejor media obtenida. Esta generalmente suele ser menor que la de *train* ya que para esta última se utiliza el conjunto de *training* al completo.

```
Mejores puntuaciones con CV:
 [0.92467532 0.94285714 0.96354167 0.93489583 0.9296875  0.95833333
 0.94240838 0.94195251 0.95778364 0.94429708]
Y su mejor media:  0.9440432407814525

Porcentaje de aciertos en train: 0.9539628563955009
Porcentaje de aciertos en test:  0.9220923761825265
```

Por otro lado, calculamos una matriz de confusión y la cantidad de fallos en cada dígito para darnos una idea de los resultados.

```
Matriz de confusión:
 [[173   0   1   0   1   3   0   0   0   0]
 [  0 162   2   0   1   1   0   0   5  11]
 [  0   1 170   1   0   0   0   4   1   0]
 [  2   0   3 161   0   6   0   2   1   8]
 [  0   0   0   0 177   0   0   1   3   0]
 [  0   1   0   0   1 177   1   0   1   1]
 [  2   1   0   0   2   0 175   0   1   0]
 [  0   1   0   0   1   6   0 164   0   7]
 [  0  15   1   0   0   3   3   0 142  10]
 [  0   4   0   1   9   5   0   0   5 156]]

Lista de fallos por dígito:
 [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
 [ 5. 20.  7. 22.  4.  5.  6. 15. 32. 24.]
```

Llama la atención en especial la confusión entre el dígito 1 y el 8, y la del 9 con el resto de números. Pese a eso, vemos que generalmente tenemos unos buenos resultados.

```
# Evaluamos el ajuste
print("Mejores puntuaciones con CV:\n", best_scores)
print("Y su mejor media: ", best_media, "\n")
print("Porcentaje de aciertos en train:", best_classifier.score(train_x, train_y))
print("Porcentaje de aciertos en test: ", best_classifier.score(test_x, test_y))


predict = best_classifier.predict(test_x)
cm = confusion_matrix(test_y, predict)

errores = np.zeros(10)
for i, row in enumerate(cm):
  errores[i] = sum([value for j,value in enumerate(row) if j != i])

print("\nMatriz de confusión:\n", cm)
print("\nLista de fallos por dígito:\n", list(range(10)), "\n", errores)
```

Justificación del modelo

Empezamos basándonos en el resultado del error obtenido para ver que el porcentaje de aciertos es bastante alto. Suponemos que la muestra es realista y por tanto puede existir ruido en la muestra, o dígitos escritos con mala caligrafía (algo frecuente en la realidad), por lo que exigir un porcentaje mayor no es razonable.

Puestos a mejorar se debería intentar bajar los errores con el número 9, ya que parte del trazo de escritura es similar al de otros dígitos como el 1 y el 8 y puede ser lo que confunda al clasificador.

Por otra parte, el hecho de utilizar *cross-validation* y *regularización* nos ayudan a tener confianza en que se ha intentado evitar el sobreajuste, y, por tanto, una mayor probabilidad de generalización.

Problema de regresión

Comprensión del problema

Para regresión contamos con un conjunto de datos sobre alerones y la presión acústica que estos originan en diferentes túneles de viento. Las características medidas son:

- Frecuencia
- Ángulo de ataque
- Dimensión del ala
- Velocidad del aire
- Desplazamiento por succión

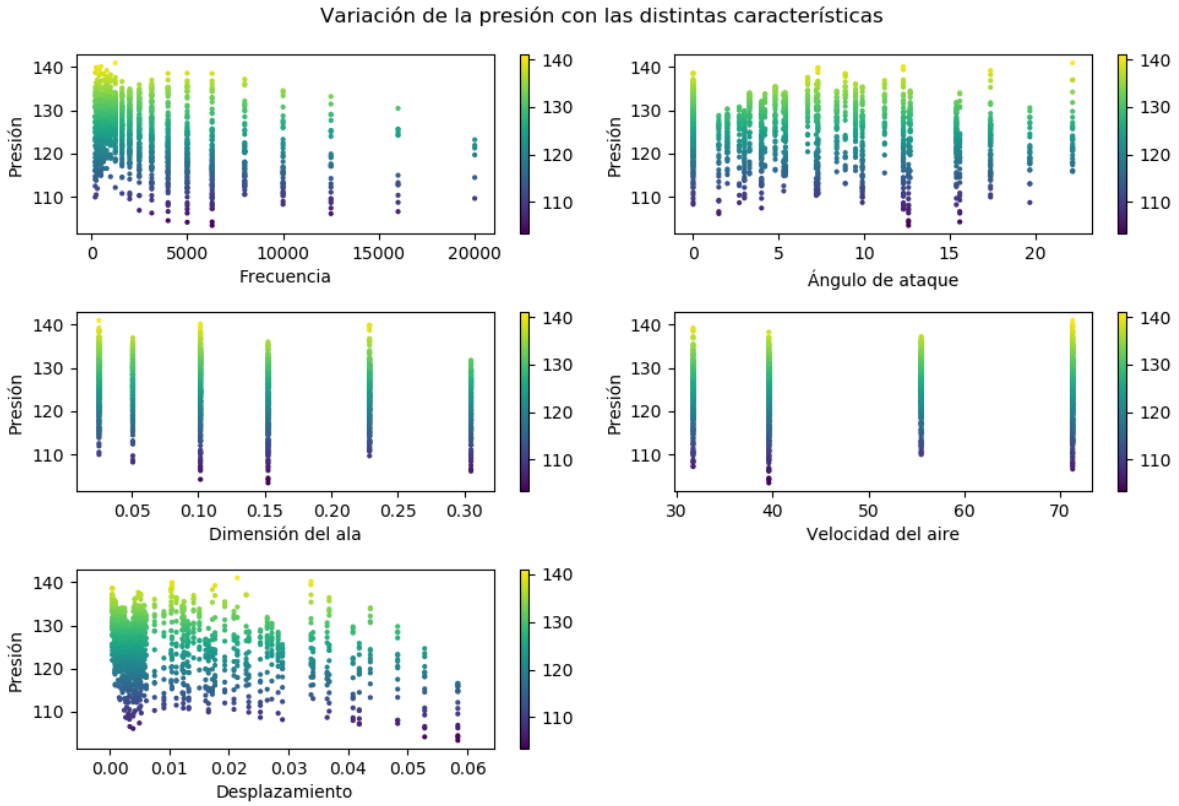
y

- Presión acústica originada

Nuestro objetivo es poder estimar la presión acústica de nuevos datos.

1000	0	0.3048	71.3	0.00266337	125.201
1250	0	0.3048	71.3	0.00266337	125.951
1600	0	0.3048	71.3	0.00266337	127.591
2000	0	0.3048	71.3	0.00266337	127.461
2500	0	0.3048	71.3	0.00266337	125.571
3150	0	0.3048	71.3	0.00266337	125.201
4000	0	0.3048	71.3	0.00266337	123.061
5000	0	0.3048	71.3	0.00266337	121.301
6300	0	0.3048	71.3	0.00266337	119.541
8000	0	0.3048	71.3	0.00266337	117.151
10000	0	0.3048	71.3	0.00266337	115.391
12500	0	0.3048	71.3	0.00266337	112.241
16000	0	0.3048	71.3	0.00266337	108.721
500	0	0.3048	55.5	0.00283081	126.416
630	0	0.3048	55.5	0.00283081	127.696
800	0	0.3048	55.5	0.00283081	128.086
1000	0	0.3048	55.5	0.00283081	126.966
1250	0	0.3048	55.5	0.00283081	126.086
1600	0	0.3048	55.5	0.00283081	126.986
2000	0	0.3048	55.5	0.00283081	126.616
2500	0	0.3048	55.5	0.00283081	124.106
3150	0	0.3048	55.5	0.00283081	123.236
4000	0	0.3048	55.5	0.00283081	121.106
5000	0	0.3048	55.5	0.00283081	119.606
6300	0	0.3048	55.5	0.00283081	117.976
8000	0	0.3048	55.5	0.00283081	116.476
10000	0	0.3048	55.5	0.00283081	113.076
12500	0	0.3048	55.5	0.00283081	111.076

Para tener una idea del efecto de cada variable sobre el valor a predecir, mostramos:



Por la forma de las gráficas se da a entender que las características no aportan información independiente, sino que el conjunto de sus valores influye en la presión originada.

Preprocesado de datos

Al igual que en el otro ejercicio, y a pesar de tener ya de por sí pocas variables, aplicamos normalización y PCA.

```
# Normalizado
scaler = preprocessing.StandardScaler()
scaler.fit(data_x)
data_x = scaler.transform(data_x)

# PCA
pca = PCA(n_components=0.99, svd_solver="full")
data_x = pca.fit_transform(data_x)

# Separamos en train y en test
train_x, test_x, train_y, test_y = train_test_split(data_x, data_y, test_size=0.2)
```

Selección de la clase de funciones

Como primera opción mantenemos los datos intactos, obteniendo:

```
Error cuadrático en train: 0.5163955862321661
Error cuadrático en test:  0.5122109441814217
Mean square error: 25.284812125029738
Mean absolute error: 3.956505241951651
```

Posteriormente probamos a añadir los cuadrados:

```
Error cuadrático en train: 0.5234695465172867
Error cuadrático en test:  0.5858591472026042
Mean square error: 20.468468958788883
Mean absolute error: 3.4569399584833094
```

Y en esta ocasión los cuadrados únicamente:

```
Error cuadrático en train: 0.3492712495971976
Error cuadrático en test:  0.4445077609918563
Mean square error: 27.603859950774627
Mean absolute error: 4.211092432295087
```

Viendo que añadiendo más variables el modelo mejora un poco, proseguimos añadiendo los cubos además de los cuadrados, pero al no conseguir mejora nos quedamos con los cuadrados:

```
Error cuadrático en train: 0.5342354266468868
Error cuadrático en test:  0.547888913227313
Mean square error: 21.220660813280922
Mean absolute error: 3.5246355951404857
```

Definición de conjuntos de datos

En este caso se nos proporciona un único conjunto de datos, pero, por las mismas razones que en el problema de clasificación, se opta por una separación *training-test* y la aplicación de *cross-validation* sobre el conjunto de *training*.

El número de particiones vuelve a ser 10, puesto que valores más pequeños no nos proporcionan resultados mejores.

También se prueba con *leave-one-out* y de igual manera no existe mejora.

```
Error cuadrático en train: 0.5370911583724274
Error cuadrático en test:  0.5353036555809376
Mean square error: 21.708191670662938
Mean absolute error: 3.498723910996911
```

Regularización

Por los mismos motivos que los del ejercicio anterior, se itera con distintos valores de regularización mediante *CV*.

Modelo usado

Puesto que estamos intentando predecir un valor continuo ni el perceptron ni regresión logística nos valen, por lo que empleamos un modelo de regresión lineal proporcionado por *Scikit-Learn*.

Tras elegir el parámetro de regularización con *CV*, ajustamos el modelo para evaluar su calidad.

Ajuste

El modelo regresión ofrecido por la librería solo permite alterar la tolerancia (*tol*), el número máximo de iteraciones (*max_iter*) y el factor de regularización (*alpha*).

Puesto que se implementa un bucle con distintos valores de *alpha*, se lanza el experimento con varios valores de tolerancia quedándonos con el que mejor resultados ofrece.

Tolerancia = 10^{-10}

```
Error cuadrático en train: 0.5217112285323066
Error cuadrático en test:  0.5667715649670111
Mean square error: 21.776787593653076
Mean absolute error: 3.5292628581622045
```

Tolerancia = 10^{-5}

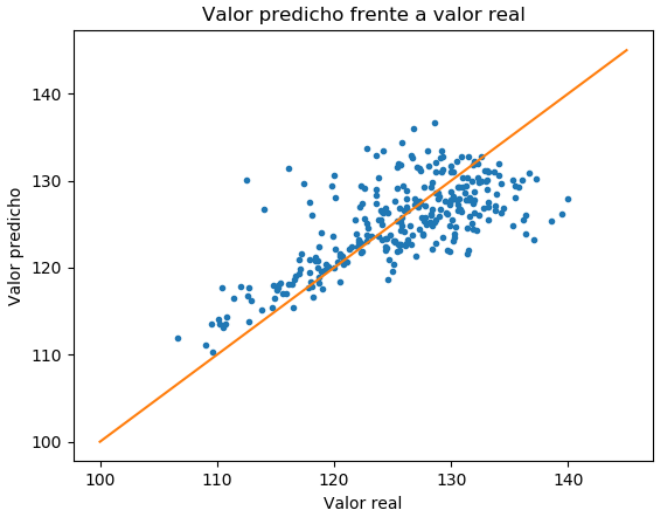
```
Error cuadrático en train: 0.5280620309084956
Error cuadrático en test:  0.5674475977057727
Mean square error: 19.761021155642457
Mean absolute error: 3.4706682917978964
```

Se ve que apenas existe variación alterando los parámetros.

```python
# Ajuste
"""
alpha    - Para regularización
max_iter - Número máximo de iteraciones
tol      - Criterio de parada
"""
tol = 1e-10
max_iter = 100_000


best_classifier = 0
best_scores = 0
best_media = float('-inf')

for i in range(3):
  alpha = 0.0001 * (10 ** i)
  classifier = Ridge(max_iter=max_iter, tol=tol, alpha=alpha)

  scores = cross_val_score(classifier, train_x, train_y, cv=10)
  media = scores.mean()

  # Nos quedamos con la mejor
  if media > best_media:
    best_classifier = classifier
    best_scores = scores
    best_media = media


print("Mejores puntuaciones con CV: ", best_scores)
print("Mejor media: ", best_media, "\n")

best_classifier.fit(train_x, train_y)
```

Métrica usada

Se ha usado una versión de regresión lineal sobre una métrica de mínimos cuadrados.
Probando con *Lasso*, se obtienen resultados similares, a veces mejores y en ocasiones peores.



Lasso

```
Error cuadrático en train: 0.5469952640661253  Error cuadrático en train: 0.5479612335656519
Error cuadrático en test:  0.48445338450042386 Error cuadrático en test:  0.5317687386006724
Mean squared error: 21.919512082716437         Mean squared error: 25.96357745848067
Mean absolute error: 3.5538863933520233        Mean absolute error: 3.8432655014339345
```

Estimación del error

Utilizamos las funciones de la librería para comprobar la calidad del modelo.

Primeramente, al igual que con el problema de clasificación, elegimos el modelo con los parámetros que ofrecen mejor media en *CV*.

```
Mejores puntuaciones con CV:  [0.49509582 0.30205144 0.62020206 0.45776405 0.62923717 0.49340507
 0.45982773 0.47163371 0.60281323 0.48849374]
Mejor media:  0.5020524014941572
```

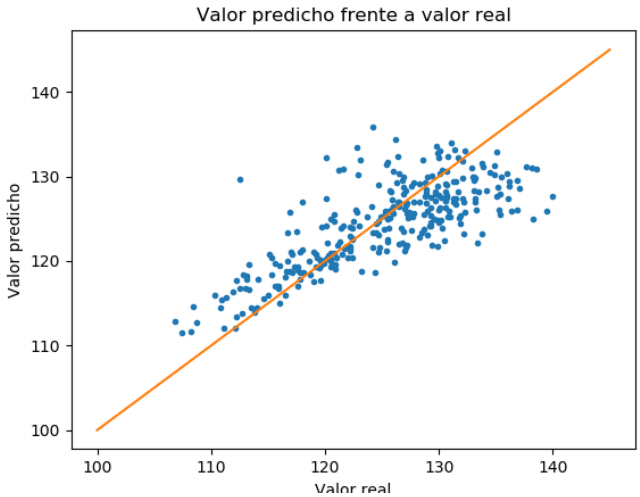
Una vez clasificado, podemos comprobar el error cuadrático:

```
Error cuadrático en train: 0.5297170302274004
Error cuadrático en test:  0.5645543451357861
```

Y los errores cuadrático y absoluto medio:

```
Mean squared error: 20.170783363394193
Mean absolute error: 3.423563331364162
```

Teniendo en una gráfica la diferencia entre valor predicho y valor real de los datos.



Justificación del modelo

Viendo la gráfica podemos afirmar que no se ha aprendido a predecir de manera exacta. Esto no quiere decir que el ajuste sea totalmente malo, todo depende de la precisión que se le requiera al programa. Es posible que lo que interese sea una aproximación de la presión y no un valor exacto, en cuyo caso el modelo podría servir.

En una aplicación real vería más conveniente intentar utilizar otro tipo de modelos. Por ejemplo, probando con modelos no lineales tenemos que *Random Forest* nos da un ajuste muchísimo mejor, pero *Singular Vector Decomposition* nos da resultados bastante similares a los de nuestro modelo.

```
Otros modelos no lineales:
SVM test:  0.6313492681837114
SVM train:  0.6797151144673582
SVM MSE:  18.35453467031381

RF test:  0.8873684681712153
RF train:  0.9812115321955314
RF MSE:  5.607745156877078
```