Aprendizaje Automático (2018-2019)
Grado en Ingeniería Informática
Universidad de Granada

Informe práctica 3 : Ajuste de modelos lineales

Montserrat Rodríguez Zamorano

20 de mayo de 2019

Índice

1. Problema de reconocimiento de caracteres escritos **1**
1.1. Compresión del problema a resolver 1
1.2. Preprocesado de los datos 2
1.3. Selección de clases de funciones a usar 2
1.4. Conjuntos de training, validación y test usados en su caso 3
1.5. Necesidad de regularización 3
1.6. Modelos a usar. Estimación de parámetros e hiperparámetros 4
1.7. Selección y ajuste modelo final 4
1.8. Idoneidad de la métrica usada en el ajuste 5
1.9. Estimación del error E_{out} del modelo 6
1.10. Discusión y justificación de la calidad del modelo 6

2. Problema de regresión **7**
2.1. Compresión del problema 7
2.2. Preprocesado de los datos 7
2.3. Selección de clases de funciones a usar 8
2.4. Conjuntos de training, validación y test usados en su caso 8
2.5. Necesidad de regularización 9
2.6. Modelos a usar. Estimación de parámetros e hiperparámetros 9
2.7. Selección y ajuste modelo final 9
2.8. Idoneidad de la métrica usada en el ajuste 10
2.9. Estimación del error E_{out} del modelo 11
2.10. Discusión y justificación de la calidad del modelo 11

1. Problema de reconocimiento de caracteres escritos

1.1. Compresión del problema a resolver

Este primer problema es un problema de clasificación de dígitos escritos. Se dispone de un *data set* con 5620 muestras en la que han contribuido 43 personas: 30 al conjunto de entrenamiento y 13 al de test.

Los archivos de datos tienen 65 columnas. Las 64 primeras corresponden a los atributos, que tendrán valores entre 0 y 16. La última columna es el dígito con dichos atributos, y variará por tanto entre 0 y 9.

```
# Funcion para leer los datos
def classification_loadData(filetrain, filetest):
    print('Loading␣data␣from␣file␣', filetrain, '...')
    print('Loading␣data␣from␣file␣', filetest, '...')
    train = pd.read_csv(filetrain, header =None)
    test = pd.read_csv(filetest, header=None)
    #convertir los valores en array
    array_train = train.values
    array_test = test.values
    #separamos los datos en atributos y salida
    Xtrain = np.array(array_train[:,0:64], np.float64)
    ytrain = np.array(array_train[:,64], np.float64)
    Xtest = np.array(array_test[:,0:64], np.float64)
    ytest = np.array(array_test[:,64], np.float64)

    print('Done.')

    return Xtrain, Xtest, ytrain, ytest
```

Listing 1: Código para leer los archivos

Dibujamos algunos dígitos de nuestra base de datos. El código utilizado para mostrar dígitos y su resultado puede verse a continuación (2, 1.1).

```
#Funcion para dibujar digitos
def showDigits(x,y,num):
    plt.figure(figsize=(6,2))
    for i in range(num):
        plt.subplot(2,5, i+1)
        plt.imshow(np.split(x[i],8),cmap=plt.cm.gray_r)
        plt.axis('off')
    plt.show()
    #imprimimos las etiquetas
    print(y[0:num])
```

Listing 2: Código para mostrar dígitos

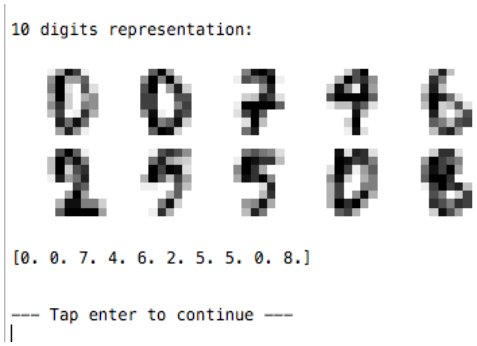

Figura 1.1: Representación de algunos dígitos

Para resolver el problema haremos uso de la **Regresión Logística**.

1.2. Preprocesado de los datos

En primer lugar cargamos los archivos y comprobamos las dimensiones de nuestros datos (1) para comprobar que son correctas. Como vemos, tenemos 3823 instancias para training y 1797 para test, dando un total de 5620 instancias, como queríamos.

```
Task 1. Classification problem

Loading data from file  datos/digits/optdigits.tra ...
Loading data from file  datos/digits/optdigits.tes ...
Done.
X_train shape: (3823, 64)
X_test shape: (1797, 64)


--- Tap enter to continue ---
```

Figura 1.2: Dimensiones de los datos

Como parte del preprocesado de los datos se normalizarán los valores de los atributos, que oscilan en el rango 0..,16.

```
#Funcion para normalizar datos
def normalizeData(x):
    norm = preprocessing.Normalizer()
    x_ = norm.fit_transform(x)
    return x_

#Funcion para preprocesar los datos para clasificacion
def classificationPreprocessing(x):
    print('Preprocessing data...')
    x_= normalizeData(x)
    print('Done.')
    return x_
```

Listing 3: Código preprocesado datos para clasificación

1.3. Selección de clases de funciones a usar

Se utilizará la clase de funciones lineales, como se indica en el ejercicio.

1.4. Conjuntos de training, validación y test usados en su caso

Se proporciona una base de datos de datos de entrenamiento y otra de datos de test, separadas, por lo que no hará falta separar estos dos conjuntos.

Por otra parte, se usará *cross validation*, que consistirá en dividir el conjunto de entrenamiento en n subconjuntos. Se tomarán $n-1$ conjuntos de datos para el entrenamiento y el conjunto restante para *test*. Se repetirá este proceso hasta que todos los conjuntos se hayan usado para *test*.

En este caso se ha dividido el conjunto en 10 subconjuntos. Es recomendable que tengan aproximadamente el mismo número de datos, pero en este caso, al ser el número de filas del conjunto de entrenamiento un número primo, no se ha podido dividir en subconjuntos iguales.

Al ser un clasificador, se ha utilizado $StratifiedKFold$. Puede verse el código utilizado a continuación (4).

```
def classificationCrossValidation(x,y, numsplits):
    #rango de exponentes para c
    c_s = range(-3,3)
    #dividimos el conjunto en numsplits subconjuntos
    skf = StratifiedKFold(n_splits=numsplits)
    skf.get_n_splits(x, y)
    bestc = 0
    bestacc = 0
    #probamos diferentes cs
    for i in c_s:
        c = 10**i
        accuracy = 0
        #iteramos en los subconjuntos
        for train_index, test_index in skf.split(x, y):
            #subconjuntos de train y test
            X_train_, X_test_ = x[train_index], x[test_index]
            y_train_, y_test_ = y[train_index], y[test_index]
            model = LogisticRegression(C=c, solver='lbfgs')
            #ajustar el modelo
            model.fit(X_train_, y_train_)
            #predicciones usando el conjunto test
            y_pred = model.predict(X_test_)
            #calcular score
            accuracy += metrics.accuracy_score(y_test_, y_pred)
        #score de media con c actual
        mean_accuracy = accuracy/numsplits
        #si el c actual da mejor resultado en media
        #lo guardamos con el mejor c
        if mean_accuracy > bestacc:
            bestacc = mean_accuracy
            bestc = c
    return bestc, bestacc
```

Listing 4: Código validación cruzada para clasificación

1.5. Necesidad de regularización

Es recomendable regularizar para evitar *overfitting*. De hecho, en regresión logística, la documentación de *scikit learn* (`https://scikit-learn.org/stable/modules/generated/sklearn.`

`linear_model.LogisticRegression.html`) alude a la regularización y al hecho de que se aplica por defecto, a través del parámetro c.

En la función programada en (4) el bucle exterior es el que se encarga de establecer el exponente de c. En este caso, tenemos $c = 100$.

1.6. Modelos a usar. Estimación de parámetros e hiperparámetros

El modelo a usar será, como se ha indicado, regresión logística. Uno de los hiperparámetros será c, que en la documentación se define como la inversa de la fuerza de la regularización. Otro hiperparámetro es *numsplits*, que indica el número de divisiones del conjunto en la validación cruzada.

1.7. Selección y ajuste modelo final

El modelo final escogido será aquel con el c que nos proporcione un menor porcentaje de fallos, en este caso $c = 100$. A continuación puede verse el código (6) usado para el ajuste del modelo.

```
#Funciones para ajustar el modelo y realizar predicciones
def adjustModel(xtrain, ytrain, xtest, ytest, model):
    #ajustar el modelo
    model.fit(xtrain, ytrain)
    #realizar predicciones
    y_pred = model.predict(xtest)
    #calcular score
    accuracy = metrics.accuracy_score(ytest, y_pred)
    return y_pred, accuracy

def logisticRegression(xtrain, ytrain, xtest, ytest, c):
    #seleccionamos como modelo regresion logistica
    model = LogisticRegression(C=c, solver='lbfgs')
    #ajustar el modelo, devuelve predicciones y score
    predictions, accuracy = adjustModel(xtrain, ytrain, xtest,
       ytest, model)
    return predictions, accuracy
```

Listing 5: Código regresión logística

A continuación puede verse la salida del programa, el resultado de la siguiente ejecución (6), que incluye la ejecución de la validación cruzada.

```
    start = time()
    #aplicar cross validation
    bestc, bestacc = classificationCrossValidation(xtrain_processed
       , y_train, 10)
    #tiempo de fin
    end = time() - start
    print('Time used in Crossvalidation : ', end)
    print('Best c:', bestc)
    #error con los datos de training
    print('Ein: {:.5f}'.format(1.0-bestacc))
    #aplicar regresion logistica
    y_pred, accuracy = logisticRegression(xtrain_processed, y_train
       , xtest_processed, y_test, bestc)
    print('Accuracy: {:.2f}'.format(accuracy*100),'%')
    #error con datos de test
```

```
    print('Eout: {:.5f}'.format(1.0-accuracy))
```

Listing 6: Código regresión logística (II)

```
Time used in Crossvalidation :  16.03740429878235
Best c: 100
Ein: 0.04129
Accuracy: 95.10 %
Eout: 0.04897


--- Tap enter to continue ---
```

Figura 1.3: Salida del programa

1.8. Idoneidad de la métrica usada en el ajuste

Se escoge como métrica la matriz de confusión. Me parece una buena decisión ya que podemos detectar dónde falla el modelo.

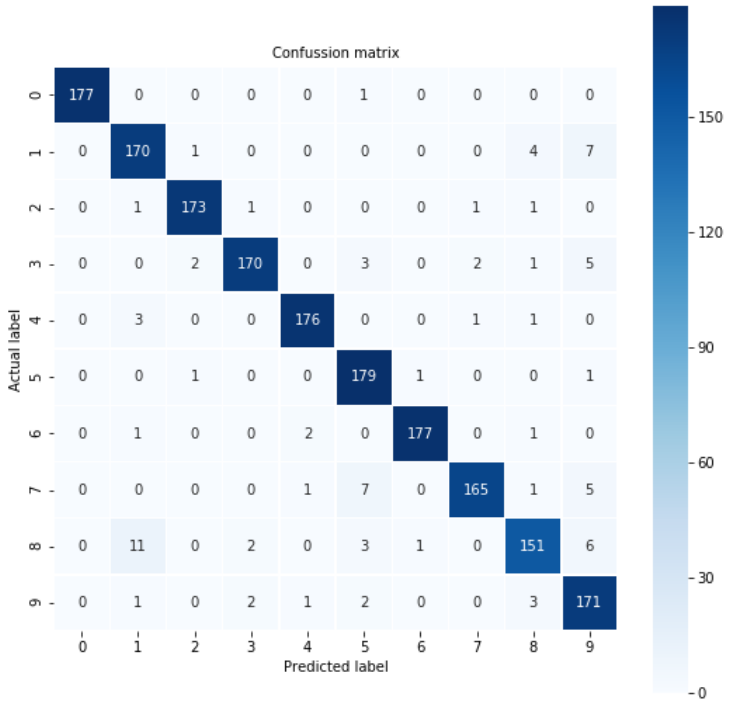

Figura 1.4: Matriz de confusión

Viendo esta matriz de confusión, podemos ver qué errores se han cometido. Podemos ver que el error más repetido es confundir ochos con unos. A continuación se muestran algunos dígitos que han sido mal clasificados.

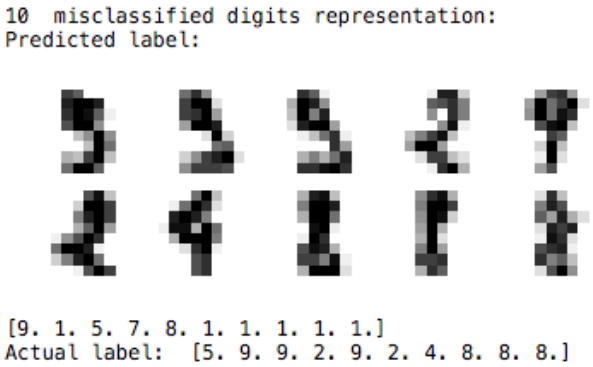

Figura 1.5: Algunos dígitos mal clasificados

Como puede verse, es un error razonable. A simple vista muchas veces no es fácil distinguir unos de otros, de no ser porque tenemos la verdadera etiqueta escrita debajo. Por ejemplo, a simple vista, los tres primeros números parecen iguales, pero se trata de dos números diferentes.

1.9. Estimación del error E_{out} del modelo

Como puede verse en la salida del programa (1.3), $E_{out} = 0{,}04897$, calculado como $1-$ *aciertos* en (7).

```
    #error con datos de test
    print('Eout: {:.5f}'.format(1.0-accuracy))
```

Listing 7: Código cálculo E_{out}

1.10. Discusión y justificación de la calidad del modelo

Me parece un buen modelo, ya que nuestro porcentaje de aciertos es muy alto. Además, hay que tener en cuenta que se ha entrenado el modelo con dígitos escritos por personas con caligrafías diferentes, y por tanto es normal que existan errores, y más cuando ni a simple vista pueden distinguirse en muchas ocasiones, como pudo verse en (1.5).

2. Problema de regresión

2.1. Compresión del problema

Este problema es un problema de regresión. Se nos proporciona una base de datos obtenidos por la NASA con datos obtenidos de pruebas aerodinámicas y acústicas de secciones de palas aerodinámicas realizadas en un túnel de viento.

Los datos de entrada serán los siguientes:

- Frecuencia, en Hercios.
- Ángulo de ataque, en grados.
- Longitud de cuerda, en metros.
- Velocidad de flujo libre, en metros por segundo.
- Grosor de desplazamiento del lado de aspiración, en metros.

Tendremos una única salida: el nivel de presión sonora, en decibelios. Tenemos un total de 1503 instancias.

Se usará el modelo de **Regresión lineal Ridge**.

2.2. Preprocesado de los datos

Comprobamos que los datos tienen las dimensiones deseadas y separamos en *inputs* y *output*, teniendo en cuenta que las primeras 6 columnas corresponden a los datos de entrada y la última a la salida.

```
#Funcion para cargar los datos de los archivos
def regression_loadData(file):
    print('Loading␣data␣from␣file␣', file, '...')
    #inicializar listas de atributos y output
    data_input = []
    data_output = []
    #abrir el archivo
    with open(file, 'r') as cvsfile:
        #ponemos como delimitador la tabulacion
        csvreader = csv.reader(cvsfile, delimiter='\t')
        #para cada fila
        for row in csvreader:
            #guardamos las primeras seis columnas como atributos
            inp = np.array(row[:5])
            data_input.append(inp)
            #guardar la ultima columna como output
            outp = row[5]
            data_output.append(outp)
    #convertir a array
    data_input = np.array(data_input, np.float64)
    data_output = np.array(data_output, np.float64)
    print(data_input.shape)
    print(data_output.shape)
    print('Done.')
```

```
    return data_input, data_output
```

Listing 8: Código procesar datos

```
Task 2. Regression problem

Loading data from file  datos/airfoil/airfoil_self_noise.dat
...
(1503, 5)
(1503,)
Done.
```

Figura 2.1: Dimensiones de los datos.

Como parte del proceso de preprocesado, se escalan y se centran los datos para intentar conseguir un mejor ajuste.

```
#Funcion para estandarizar datos (media 0 y varianza 1)
def standarizeData(x):
    stand = preprocessing.StandardScaler()
    x_ = stand.fit_transform(x)
    return x_

def minMaxScaler(x):
    scaler = preprocessing.MinMaxScaler()
    x_ = scaler.fit_transform(x)
    return x_

def regressionPreprocessing(x):
    print('Preprocessing␣data...')
    x_ = minMaxScaler(x)
    x_ = standarizeData(x_)
    print('Done.')
    return x_
```

Listing 9: Código preprocesado

2.3. Selección de clases de funciones a usar

Como en el caso anterior, se usará la clase de funciones lineales.

2.4. Conjuntos de training, validación y test usados en su caso

En este caso, al no tener los datos separados en conjuntos de entrenamiento y test, tendrán que dividirse. Se escogerá la partición $80\,\% - 20\,\%$ con el código (10).

```
    X_train, X_test, y_train, y_test = train_test_split(
        x_preprocessed, y, test_size=0.2)
```

Listing 10: Separación de datos en training y test

Una vez hecho esto, aplicaremos validación cruzada para probar diferentes *alpha* en el modelo. Se ha vuelto a tomar $numsplits = 10$, y en este caso se ha usado la función $KFold$, al tratarse de un problema de regresión. El código utilizado puede verse en (11).

En este caso se utilizará para medir lo bueno que es el modelo el *error cuadrático medio*, en lugar de número de aciertos, al tratarse de un problema de regresión.

```
    alpha_s = range(0,10)
    #dividimos el conjunto en numsplits subconjuntos
    kf = KFold(n_splits=numsplits)
    kf.get_n_splits(x, y)
    bestalpha = 0
    bestmse = 1000
    for i in alpha_s:
        alpha = 10**(-i)
        mse_mean = 0
        mse = 0
        #iteramos por los conjuntos
        for train_index, test_index in kf.split(x, y):
            X_train_, X_test_ = x[train_index], x[test_index]
            y_train_, y_test_ = y[train_index], y[test_index]
            #probamos el modelo con el alpha escogido
            model = Ridge(alpha=alpha)
            model.fit(X_train_, y_train_)
            y_pred = model.predict(X_test_)
            mse += mean_squared_error(y_test_, y_pred)
        mse_mean = mse/numsplits
        #si el alpha nos ha dado un mejor error cuadratico medio
           que el alpha guardado
        if mse_mean < bestmse:
            bestmse = mse_mean
            #guardar el alpha como mejor alpha hasta el momento
            bestalpha = alpha
    return bestalpha, bestmse
```

Listing 11: Código validación cruzada para regresión

2.5. Necesidad de regularización

Al igual que en problema de clasificación, se aplicará regularización para obtener mejores resultados. En este caso, el parámetro *alpha* es la fuerza de la regularización y no su inversa, como en la regresión logística.

Se modifica *alpha* en el bucle exterior, y se selecciona el mejor para el ajuste del modelo. En este caso, el valor que mejor ajusta el modelo es $alpha = 1$, como podrá verse más adelante en la salida del programa.

2.6. Modelos a usar. Estimación de parámetros e hiperparámetros

Los hiperparámetros son *alpha*, del que ya se habló anteriormente y *numsplits*, que es el número de subconjuntos en los que se dividirá el conjunto de datos para hacer validación cruzada.

El modelo a usar será regresión lineal Ridge.

2.7. Selección y ajuste modelo final

```
def ridge_regressionProblem(file):
    #cargar datos del archivo file
    x, y = regression_loadData(file)
    input('\n---␣Tap␣enter␣to␣continue␣---\n')
```

```
    #preprocesar datos
    x_preprocessed = regressionPreprocessing(x)
    #separar en conjuntos train y test
    X_train, X_test, y_train, y_test = train_test_split(
        x_preprocessed, y, test_size=0.2)
    #aplicar cross validation para hallar mejor valor del parametro
    bestalpha, bestmse = ridge_regressionCV(X_train, y_train, 10)
    #ajustar modelo
    model = Ridge(bestalpha).fit(X_train, y_train)
    y_pred = model.predict(X_test)
    #hallar error cuadratico medio
    mse = mean_squared_error(y_pred, y_test)
    #imprimir resultados
    print('Best alpha: ', bestalpha)
    print('Mean squared error: %.2f' % mse)
    print('Eint: {:.5f}'.format(1- model.score(X_train, y_train)))
    print('Eout: {:.5f}'.format(1-model.score(X_test, y_test)))
    input('\n--- Tap enter to continue ---\n')
    #pintar predicciones
    printError(y_test, y_pred)
```

Listing 12: Código modelo final

Como puede verse aquí el número de aciertos es de aproximadamente la mitad, pero el error cuadrático medio no es demasiado alto en comparación con los valores que se alcanzan.

```
Preprocessing data...
Done.
Best alpha:  1
Mean squared error: 24.85
Eint: 0.47675
Eout: 0.51947
```

Figura 2.2: Salida del programa.

2.8. Idoneidad de la métrica usada en el ajuste

Se escoge como métrica error cuadrático medio. Se prefiere esta métrica al número de aciertos como en el problema de clasificación ya que al tratarse de regresión es mejor que todos los puntos se acerquen mucho al valor esperado en lugar de tener la mitad de aciertos y que la otra mitad estén muy lejos del valor esperado.

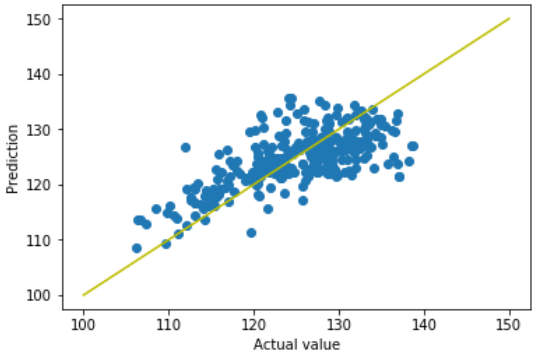

Figura 2.3: Representación de las predicciones de los datos.

El código empleado para dibujar las predicciones puede verse a continuación.

```
#Funcion que representa los datos predecidos
def printError(label, pred):
    #pintar predicciones
    plt.scatter(label, pred)
    #pintar recta para ver como de correcta es la prediccion
    plt.plot([100,150], [100,150], color = 'y')
    plt.xlabel('Actual value')
    plt.ylabel('Prediction')
    plt.show()
```

Listing 13: Código para pintar predicciones

2.9. Estimación del error E_{out} del modelo

Se calcula de misma forma que en el problema de clasificación (14). No se escribe el error obtenido ya que éste depende de la ejecución.

```
    print('Eout: {:.5f}'.format(1-model.score(X_test, y_test)))
```

Listing 14: Código cálculo E_{out}

2.10. Discusión y justificación de la calidad del modelo

No me parece un mal ajuste teniendo en cuenta que el error cuadrático medio no es muy alto en comparación con el rango en los que se mueven los valores, como puede verse en (2.3) a simple vista.

Sin embargo, hay que tener en cuenta la finalidad del ajuste. Si se necesitan resultados precisos, definitivamente se necesitan modelos más complejos para ajustar los datos. Aún así, me parece un buen ejemplo para demostrar que con modelos simples se puede conseguir un ajuste bastante aceptable.