GUIÓN DE PRÁCTICAS DE REDES NEURONALES ARTIFICIALES (TSCAO): Parte II

MªCarmen Pegalajar Jiménez

1/03/2021

Dpto Ciencias de la Computación e Inteligencia Artificial
Universidad de Granada

2 PREDICCIÓN DE LA CALIDAD DEL VINO: REGRESIÓN

A continuación, vamos a intentar predecir la calidad del vino, volveremos a utilizar los datos asociados a los vinos que vimos en la parte I del guión. En este caso, vamos a asumir que *quality* es una variable continua: la tarea no es una tarea de clasificación binaria sino una tarea de regresión ordinal. Es un tipo de regresión que se utiliza para predecir una variable ordinal: el valor *quality* existe en una escala arbitraria donde el orden relativo entre los diferentes valores de *quality* es significativo. En esta escala, la escala de calidad 0-10 para "muy malo" a "muy bueno" es un ejemplo.

Hay que tener en cuenta que también podría verse este tipo de problema como un problema de clasificación como vimos en la parte anterior, y considerar las etiquetas de calidad como etiquetas de clase fija.

En cualquier caso, esta configuración significa que sus etiquetas objetivo estarán en la columna *quality* en sus DataFrames *red* y *white* para este segundo problema de la práctica. Esto requerirá algo de preprocesamiento adicional.

Preprocesamiento de Datos

Dado que la variable *quality* se va a convertir en la clase objetivo, ahora se necesitará aislar las etiquetas de calidad del resto del conjunto de datos. Por tanto, se pondrá *wines.quality* en una variable diferente y se pondrán los datos de los vinos, con excepción de la columna *quality* en una variable *X*.

```
# Import pandas
import pandas as pd

# Read in white wine data
white = pd.read_csv('/Users/mcarmer
R/winequality-white.csv', sep=';')

# Read in red wine data
red = pd.read_csv('/Users/mcarmenpe
inequality-red.csv', sep=';')
```

```
# Add `type` column to `red` with value 1
red['type'] = 1

# Add `type` column to `white` with value 0
white['type'] = 0

# Append `white` to `red`
wines = red.append(white, ignore_index=True)
```

Seleccionamos los datos asociados a la variable X dejando fuera la variable *quality:*

```
X=wines.iloc[:,0:12]
```

```
X = wines.drop('quality', axis=1)
```

La variable objetivo *quality* la asociamos a la variable *Y*. Puesto que existen valores muy diferentes en las variables asociadas, al igual que hicimos en el problema anterior normalizaremos sus valores, utilizaremos de nuevo *StandardScaler*.

```
# Isolate target labels
#Y = wines.quality
Y=np.ravel(wines.quality)
```

```
In [22]:
```

```
# Import `StandardScaler` from `sklearn.preprocessing`
from sklearn.preprocessing import StandardScaler

# Scale
X_train=StandardScaler().fit_transform(X)
```

A continuación, se puede dividir los datos en un conjunto de entrenamiento y test, pero en este caso no seguiremos este enfoque. En esta segunda parte de la práctica, utilizaremos la validación *k-fold*, que requiere que divida los datos en *K* particiones. Por lo general, *K* se establece en 4 o 5. A continuación, se crearán instancias de modelos idénticos y se entrenará a cada uno en una partición, al tiempo que evalúa las particiones restantes. La puntuación de la validación para el modelo es entonces un promedio de las *K* puntuaciones de validación obtenidas.

Aunque ahora la usaremos para regresión, la arquitectura podría ser muy parecida, con dos capas de tipo '*Dense'*. No olvides que la primera capa es tu capa de entrada. Deberemos pasarle la forma que tienen los datos de entrada. En este caso, verás que va a utilizar *input_dim* para pasar las dimensiones de los datos de entrada que son 12 a la capa *Dense*. No hay que olvidar que también ahora estamos contando con la columna *Type*.

Volveremos a utilizar la función de activación *relu* y no utilizaremos sesgos. El número de neuronas ocultas que utilizaremos inicialmente será de 62. Terminaremos la configuración de la red con una capa de salida de una sola unidad *Dense(1)* . Esta es una configuración típica para hacer regresión en la que solo hay que predecir un único valor.

A continuación, compilaremos el modelo para ajustarlo a los datos pero utilizando una validación del tipo k-fold.

Usamos la función *compile()* para compilar el modelo y fit() para ajustarlo a los datos. Para hacer esto tenemos que definir los parámetros *optimizer* para el que escogemos *rsmprop* (algoritmo de entrenamiento más conocido) y *loss* para el que escogeremos *mse* como función de error (función de pérdida). También, tenemos que tener en cuenta el argumento *metrics* para juzgar el rendimiento del modelo. Para problemas de regresión es habitual tomar el error absoluto medio (*MAE*).

```
# Import `Sequential` from `keras.models`
from keras import Sequential

# Import `Dense` from `keras.layers`
from keras.layers import Dense

import numpy as np
from sklearn.model_selection import StratifiedKFold

seed = 7
np.random.seed(seed)

kfold = StratifiedKFold(n_splits=5, shuffle=True, random_state=seed)
for train, test in kfold.split(X, Y):
    model = Sequential()
    model.add(Dense(64, input_dim=12, activation='relu'))
    model.add(Dense(1))
    model.compile(optimizer='rmsprop', loss='mse', metrics=['mae'])
    model.fit(X_train[train], Y[train], epochs=10,verbose=0)
    y_pred=model.predict(X_train[test])
mse_value, mae_value=model.evaluate(X_train[test], Y[test], verbose=0)
print(mse_value)
```

Otra medida que también podemos usar la medida R2. Aquí, debemos obtener una puntuación cercana a 1.0, siendo 1.0 la mejor puntuación. Sin embargo, la puntuación también puede ser negativa.

```
from sklearn.metrics import r2_score

r2_score(Y[test],y_pred)
```

Recoged los resultados anteriores en una tabla donde se indiquen los parámetros, resultados y métricas estudiados hasta el momento. Realizar estas ejecuciones para un número mayor de iteraciones del algoritmo (por ejemplo 20, 30, 40, 50…) ver qué ocurre. Comentar conclusiones

Ajuste Fino del Modelo de Regresión

No siempre será tan fácil como en el primer ejemplo de clasificación que hemos visto en la parte I, ahora se trata de aumentar el número de capas y jugar con el número de neuronas en cada capa.

a) **Agregar más capas.** Probemos a añadir más capas. Realizar la ejecución con diferentes números de capas y construir una tabla con los resultados. Mantened por el momento el mismo número de neuronas, ya que veremos el resultado de cambiar el número de las neuronas posteriormente. Construir una tabla con los parámetros y métricas utilizadas. Comentad los resultados y sacar conclusiones.

```
import numpy as np
from sklearn.model_selection import StratifiedKFold

seed = 7
np.random.seed(seed)

kfold = StratifiedKFold(n_splits=5, shuffle=True, random_state=seed)
for train, test in kfold.split(X, Y):
    model = Sequential()
    model.add(Dense(64, input_dim=12, activation='relu'))
    model.add(Dense(64, activation='relu'))
    model.add(Dense(1))
    model.compile(optimizer='rmsprop', loss='mse', metrics=['mae'])
    model.fit(X_train[train], Y[train], epochs=10,verbose=0)
    y_pred=model.predict(X_train[test])
mse_value, mae_value=model.evaluate(X_train[test], Y[test], verbose=0)
print(mse_value)
```

b) **Agregar más unidades ocultas.** Partiendo de la configuración de una única capa y posteriormente de varias capas, probad con diferente número de neuronas. Construir una tabla que recoja tanto los parámetros utilizados, como las métricas y resultados. Comentad los resultados y sacar conclusiones.

```
# Import `Sequential` from `keras.models`
from keras import Sequential

# Import `Dense` from `keras.layers`
from keras.layers import Dense

import numpy as np
from sklearn.model_selection import StratifiedKFold

seed = 7
np.random.seed(seed)

kfold = StratifiedKFold(n_splits=5, shuffle=True, random_state=seed)
for train, test in kfold.split(X, Y):
    model = Sequential()
    model.add(Dense(128, input_dim=12, activation='relu'))
    model.add(Dense(1))
    model.compile(optimizer='rmsprop', loss='mse', metrics=['mae'])
    model.fit(X_train[train], Y[train], epochs=10,verbose=0)
    y_pred=model.predict(X_train[test])
mse_value, mae_value=model.evaluate(X_train[test], Y[test], verbose=0)
print(mse_value)
```

Puede ocurrir que los resultados que hemos obtenido a primera vista sean bastante malos, ahora tendremos que ajustar mejor el modelo.

IMPORTANTE: Recordad que hay que tener en cuenta que las redes tengan el menor número de neuronas, por varios motivos, ¿Cuáles son estos motivos?, ¿Cuándo se considera que una red sobrepasa el tamaño adecuado y deja de ser conveniente?

Algunos experimentos más: parámetros de optimización

Además de agregar capas y jugar con las unidades ocultas, también puedes intentar ajustar (algunos de) los parámetros del algoritmo de optimización que proporciona la función *compile()*. Hasta ahora, siempre le hemos pasado al argumento *optimizer* el parámetro *rmsprop*.

Podemos probar la importación de *RMSprop* desde *keras.models* y ajustar la tasa de aprendizaje *lr*. También puede cambiar los valores predeterminados que se han establecido para los otros parámetros *RMSprop()*, pero esto no es muy recomendable.

```
from keras.optimizers import RMSprop


rmsprop=RMSprop(lr=0.0001)
seed = 7
np.random.seed(seed)

kfold = StratifiedKFold(n_splits=5, shuffle=True, random_state=seed)
for train, test in kfold.split(X, Y):
    model = Sequential()
    model.add(Dense(128, input_dim=12, activation='relu'))
    model.add(Dense(1))
    model.compile(optimizer='rmsprop', loss='mse', metrics=['mae'])
    model.fit(X_train[train], Y[train], epochs=10,verbose=0)
    y_pred=model.predict(X_train[test])
mse_value, mae_value=model.evaluate(X_train[test], Y[test], verbose=0)
print(mse_value)
```

Realiza los cambios que comentamos y, además, prueba experimentar con otros algoritmos de optimización, como el Descenso de gradiente estocástico (SGD). ¿Qué efecto notas?

Vuelve a recoger tus resultados en una tabla junto con los parámetros utilizados y métricas. Comenta resultados y saca conclusiones.

```
from keras.optimizers import SGD, RMSprop


sgd=SGD(lr=0.1)
seed = 7
np.random.seed(seed)

kfold = StratifiedKFold(n_splits=5, shuffle=True, random_state=seed)
for train, test in kfold.split(X, Y):
    model = Sequential()
    model.add(Dense(128, input_dim=12, activation='relu'))
    model.add(Dense(1))
    model.compile(optimizer=sgd, loss='mse', metrics=['mae'])
    model.fit(X_train[train], Y[train], epochs=10,verbose=0)
    y_pred=model.predict(X_train[test])
mse_value, mae_value=model.evaluate(X_train[test], Y[test], verbose=0)
print(mse_value)
```

Finalmente, De todas las que has probado, ¿Cuál sería para ti la mejor configuración del modelo que se adapta al problema? ¿Por qué?

Escoger otro problema de regresión y explorar el diseño que mejor se adapte al problema

3. CLASIFICACIÓN DEL RIESGO DE ABANDONO DE LOS CLIENTES DE UN BANCO

El conjunto de datos con el que vamos a trabajar ahora contiene información sobre los usuarios de un banco. Queremos predecir si los clientes van a dejar de usar los servicios de dicho banco o no. El conjunto de datos consta de 10000 observaciones y 14 variables.

La siguiente figura indica cómo cargar el conjunto de Datos:

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

In [2]:
```
import matplotlib.pyplot as plt
```

In [3]:
```
import pandas as pd
```

In [4]:
```
dataset = pd.read_csv('D:\mcarmen2019-2010\PYTHON\Churn_Modelling.csv')
```

In [5]:
```
dataset.head()
```

Out[5]:

	RowNumber	CustomerId	Surname	CreditScore	Geography	Gender	Age	Tenure	Balanc
0	1	15634602	Hargrave	619	France	Female	42	2	0.(
1	2	15647311	Hill	608	Spain	Female	41	1	83807.8
2	3	15619304	Onio	502	France	Female	42	8	159660.8
3	4	15701354	Boni	699	France	Female	39	1	0.(
4	5	15737888	Mitchell	850	Spain	Female	43	2	125510.8

Creamos una matriz con las variables de entrada y otra matriz con la variable de salida (objetivo, columna 14). Excluiremos la columna 1 y 2 que son '*row_number*' y '*customerid*' ya que no nos aportan información útil para el análisis.

In [6]:

```
X = dataset.iloc[:, 3:13].values
```

In [7]:

```
X[0:4]
```

Out[7]:

```
array([[619, 'France', 'Female', 42, 2, 0.0, 1, 1, 1, 101348.88],
       [608, 'Spain', 'Female', 41, 1, 83807.86, 1, 0, 1, 112542.58],
       [502, 'France', 'Female', 42, 8, 159660.8, 3, 1, 0, 113931.57],
       [699, 'France', 'Female', 39, 1, 0.0, 2, 0, 0, 93826.63]],
      dtype=object)
```

In [8]:

```
y = dataset.iloc[:, 13].values
```

Vamos a hacer el análisis más sencillo si codificamos las variables no numéricas. *Country* contiene los valores: 'France, Spain, Germany' y *Gender:* 'Male, Female'. La manera de codificarlo será convertir estas palabras a valores numéricos. Para esto usaremos la función *LabelEncoder*, de la librería 'ScikitLearn', que al darle una cadena de texto nos devuelve valores entre 0 y n_clases-1.

In [9]:

```
from sklearn.preprocessing import LabelEncoder, OneHotEncoder
```

In [10]:

```
labelencoder_X_1 = LabelEncoder()
```

In [11]:

```
X[:, 1] = labelencoder_X_1.fit_transform(X[:, 1])
```

In [12]:

```
labelencoder_X_2 = LabelEncoder()
```

In [13]:

```
X[:, 2] = labelencoder_X_2.fit_transform(X[:, 2])
```

Observamos que *Country* ahora toma valores del 0 al 2 mientras que *male* y f*emale* fueron reemplazados por 0 y 1.

Usaremos la función *train_test_split* de la librería *ScikitLearn* para dividir nuestros datos. Usaremos 80% para entrenar el modelo y 20% para validarlo.

In [14]:
```
from sklearn.model_selection import train_test_split
```

In [15]:
```
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size = 0.2)
```

Si observamos los datos detenidamente podemos apreciar que hay variables cuyos valores pueden ser muy variados, desde muy altos a muy pequeños por esta razón escalaremos los datos.

In [16]:
```
from sklearn.preprocessing import StandardScaler
```

In [17]:
```
sc = StandardScaler()
```

In [18]:
```
X_train = sc.fit_transform(X_train)
```

In [19]:
```
X_test = sc.transform(X_test)
```

Una vez escalados los datos, pasamos a construir la red neuronal. Importamos Keras, usamos el módulo *Sequential* para inicializar la red y el modelo *Dense* para añadir capas ocultas.

In [20]:
```
import keras
```

```
Using TensorFlow backend.
```

In [21]:
```
from keras.models import Sequential
```

Inicializamos la red con Sequential().

```
In [22]:
from keras.layers import Dense

In [23]:
classifier = Sequential()
```

Añadimos las capas usando la función *Dense*. Indicamos el número de nodos que queremos añadir con *output_dim, Init* es la inicialización del descenso de gradiente estocástico. Los pesos iniciales serán una variable aleatoria uniforme. *Input_dim* sólo es necesaria en la primera capa para que el modelo sepa la cantidad de variables que va a recibir, en nuestro caso 11. A partir de aquí las siguientes capas heredarán esta cualidad de la primera capa. La función de activación que utilizaremos será *relu* en las dos primeras capas *(*cuanto más cerca tenga su valor a 1, la neurona estará más activada y tendrá más interacción*)* y en la capa final hemos utilizado la función sigmoide ya que nuestro objetivo es clasificar.

Una vez que tenemos la configuración específica de la red, la siguiente tarea es compilarla, para eso utilizamos la función *Compile*. El primer argumento de esta función es *Optimizer* que indica el método para entrenar los pesos. *Adam* es un algoritmo que se basa en el cálculo del descenso del Gradiente Estocástico. El segundo parámetro es *loss*, este usará la función '*binary_crossentropy'* para clasificar en 2 categorías. Si tuviéramos más categorías utilizaríamos la función '*categorical_crossentropy*'. Para saber la bondad de nuestra red neuronal utilizaremos la métrica *accuracy*.

In [24]:

```
classifier.add(Dense(6, activation = 'relu', input_shape =(10,)))
```

In [25]:

```
classifier.add(Dense(6, activation = 'relu'))
```

In [26]:

```
classifier.add(Dense( 1, activation = 'sigmoid'))
```

In [27]:

```
classifier.compile(optimizer = 'adam', loss = 'binary_crossentropy', metrics = ['accuracy']
```

Usaremos la función *fit* para ajustar los pesos de la red. *Batch_size* para especificar el número de observaciones que necesita entrenar antes de actualizar los pesos. *Epoch* nos indica el número de iteraciones que realizaremos en el entrenamiento. La estimación de estos parámetros se tiene que hacer por ensayo-error, probando con diferentes valores

In [ ]:

```
classifier.fit(X_train, y_train,epochs=100, batch_size=1, verbose=1)
```

```
Epoch 1/100
8000/8000 [==============================] - 8s 1ms/step - loss: 0.4687 - ac
curacy: 0.7937
```

Para realizar la predicción sobre nuestro conjunto de test lo haremos mediante la siguiente expresión:

```
y_pred = classifier.predict(X_test)

y_pred = (y_pred > 0.5)
```

La predicción nos proporcionará la probabilidad de pertenecer a un grupo u otro, de tal manera que aquellos valores mayores que 0.5 serán 1 y el resto 0.

Creamos una matriz de confusión y vemos los resultados:

```
from sklearn.metrics import confusion_matrix

cm = confusion_matrix(y_test, y_pred)

cm
```

En base a todo lo aprendido, realizar una tabla que recoja los parámetros, métricas y resultados obtenidos de las diferentes ejecuciones que hayas realizado. Sacar conclusiones.