2020

Detección de incoherencias en una evaluación por pares

DESAFIOS DE PROGRAMACION

ANTONIO BRI PÉREZ

Contenido

1. Nivel básico

1.1. Preprocesado básico de palabras

Necesitamos preprocesar el texto para deshacernos de información que no nos interesa a la hora de realizar una predicción como por ejemplo, comas, mayúsculas y minúsculas así como extraer las raíces de las palabras.

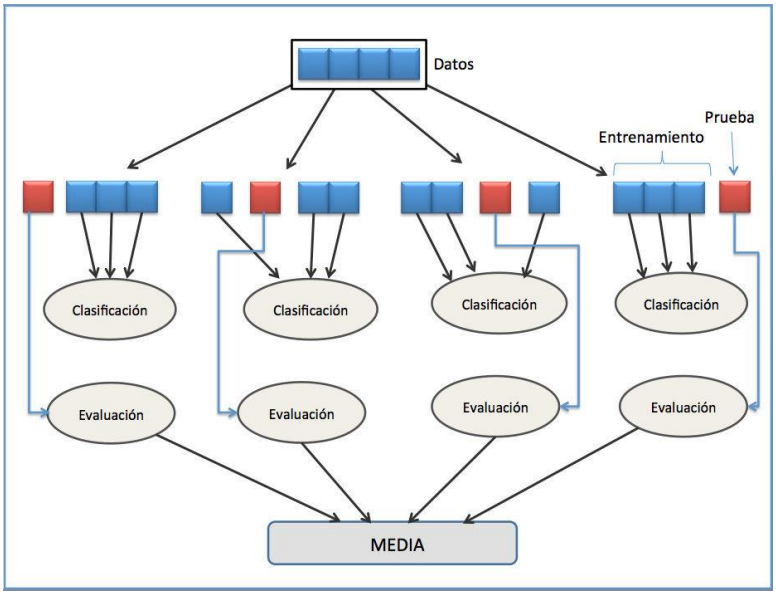
```
1 import nltk
2 nltk.download('stopwords')
3
4 from nltk.stem.snowball import SnowballStemmer
5 from nltk.tokenize import RegexpTokenizer
6
7 tokenizer = RegexpTokenizer(r'\w+')
8
9 stemmer = SnowballStemmer("spanish",ignore_stopwords=True)
10
11 preprocessed_feedback = []
12 for i in data.feedback:
13   tokens = [stemmer.stem(word) for word in tokenizer.tokenize(i.lower())]
14   preprocessed_feedback.append(np.array(' '.join(tokens)))
15
16 data['feedback prep'] = preprocessed_feedback
17 data['feedback prep'] = data['feedback prep'].astype('str')
18 data.head()
```

En este caso, importamos las librerías apropiadas y en la línea 7 nos declaramos un tokenizer para separar las frases por espacios en blanco. A continuación, con SnowBalllStemmer nos creamos un stemmer para quedarnos solo con la raíz de las palabras. Seguidamente, en el bucle for recorremos todas las valoraciones escritas y nos guardamos en los tokens todas las raíces de las palabras y además las convertimos a minúsculas. Con esto, tendríamos un preprocesado muy básico de las palabras. Aquí un ejemplo del resultado:

feedback	feedback prep
Faltan datos para completar la página	falt dat para complet la pagin
Poco original y trabajo muy básico.	poco original y trabaj muy basic
Correcto	correct

1.2. Validación cruzada 10-CV

La validación cruzada es una técnica utilizada para evaluar los resultados de un modelo y garantizar que son independientes de la partición entre datos de entrenamiento y datos de prueba. Consiste en repetir y calcular la media aritmética obtenida de las medidas de evaluación sobre diferentes particiones. Esta imagen ilustra muy bien que es un cross Validation con 4 splits:



En nuestro caso, tendríamos 10 splits

Para realizar la validación cruzada haremos uso de la función KFold del paquete sklearn.model_selection.

```
from sklearn.model_selection import KFold
n = int(len(new_data)*0.9)


#Resultados del primer clasificador
cvscores = []
cv = KFold(n_splits = 10, random_state=42, shuffle=True)
for train_index, test_index in cv.split(X):
  X_train, X_test = X[train_index], X[test_index]
  y_train, y_test = y[train_index], y[test_index]

```

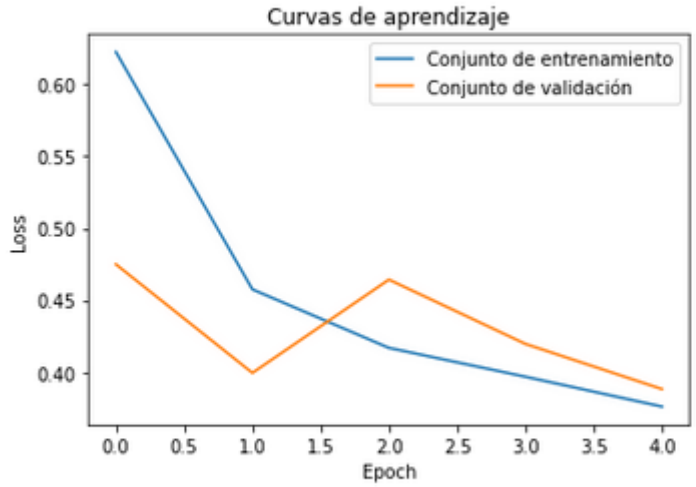
Creamos un objeto cv y le asignamos la función KFold en la cual hacemos 10 splits, fijamos el random_state a un valor constante y activamos el flag shuffle. A continuación, construimos nuestros conjuntos de entrenamiento y de tests. Una vez hemos hecho el Split, tendremos que

entrenar a nuestro modelo:

```
#Resultados del primer clasificador
cvscores = []
cv = KFold(n_splits = 10, random_state=42, shuffle=True)
for train_index, test_index in cv.split(X):
  X_train, X_test = X[train_index], X[test_index]
  y_train, y_test = y[train_index], y[test_index]

  input_shape = X_train.shape[1:]
  model = cnn_model1(input_shape)


  history = model.fit(X_train, y_train, epochs = 5, validation_split=0.1, batch_size = 32)
  scores = model.evaluate(X_test, y_test, verbose=1)

  print("%s: %.2f%%" % (model.metrics_names[1], scores[1] * 100))
  cvscores.append(scores[1] * 100)


# Para verificar que el modelo entrena correctamente crearemos un conjunto de validación del 10% de lo
print('Mostramos las curvas de aprendizaje')
plot_learning_curves(history)


print("%.2f%% (+/-%.2f%%)" % (np.mean(cvscores), np.std(cvscores)))
ResultadosPrimerClasificador = cvscores

```

Y finalmente, realizamos las predicciones con los valores de test y visualizamos los resultados:

```
mean_squared_error: 32.75%
Mostramos las curvas de aprendizaje
```




```
49.10% (+/-17.93%)
```

Finalmente, visualizamos los resultados de nuestra red

Visualización de resultados

Tenemos que crear un DataFrame con el texto del feedback, valores reales, los predichos y su diferencia para apreciar las diferencias.

```
1
2 X_test_new = new_data[len(X_train):][['feedback','value']].copy()
3 X_test_new['value pred'] = y_pred.ravel()
4 X_test_new.round(1)
```

	feedback	value	value pred
9412	Ningún problema con la imagen	3.0	2.8
899	Todo está bien. Además, me ha parecido muy bie...	3.0	2.1
3751	Perfecto todo	3.0	2.1
6047		2.0	1.6
4021	No aparece ninguna licencia en el power point,...	1.0	2.8
...	...	...	...
111		3.0	2.8
942	No indica el tipo de licencia en la presentación	1.7	2.8
5664	Aburrida y sin originalidad	3.0	2.8
96	ok	3.0	2.8
7763	No se ha citado ni indicado correctamente, la ...	1.0	1.6

950 rows × 3 columns

La verdad es que la red tarda unos 10 minutos en hacer todas las vueltas de la validación cruzada. En concreto, se demora mucho en cada una de las épocas y desconozco porque sucede esto. Por eso, tan solo la entreno 5 épocas por cada Split de la validación cruzada.

1.3. Aplicación de un par de redes neuronales (bag of words)

Para el bag of words emplearemos el mismo modelo de red que en el apartado anterior pero llevaremos a cabo el concepto de bag of words. El bag of words. El bag of words es un método que se utiliza para representar documentos ignorando el orden de las palabras. Por lo tanto, este método permite un modelado de las palabras basado en diccionarios donde cada bolsa contiene unas cuantas palabras del diccionario.

El primer paso de todos será realizar un Preprocesado de las palabras. Para ello, emplearemos el siguiente script:

```
for sen in range(0, len(data.feedback)):
    # Remove all the special characters
    document = re.sub(r'\W', ' ', str(data.feedback[sen]))

    # Substituting multiple spaces with single space
    document = re.sub(r'\s+', ' ', document, flags=re.I)


    # Converting to Lowercase
    document = document.lower()

    # Lemmatization
    document = document.split()

    document = [stemmer.lemmatize(word) for word in document]
    document = ' '.join(document)

    documents.append(document)
```

En este script utilizamos la librería de expresiones regulares “re” para realizar el Preprocesado. Comenzamos por suprimir todos los caracteres especiales. A continuación, sustituimos los posibles múltiples espacios en blanco por un solo espacio en blanco ya que tener varios espacios en blanco supondría ruido en la muestra. Seguidamente, transformamos todas las palabras a mayúsculas y nos quedamos solo con su lexema. Por ejemplo, en la palabra original, nos quedaríamos con “orig”.

Una vez hemos Preprocesado el texto tendremos que convertirlo a números empleando la ténica de bolsa de palabras o Bag of Words.

```
from sklearn.feature_extraction.text import CountVectorizer

vectorizer = CountVectorizer(max_features=1500, min_df=5, max_df=0.7, stop_words=stopwords.words('spanish'))
X = vectorizer.fit_transform(documents).toarray()
```

En el fragmento de arriba utilizamos la clase CountVectorizer. EL primer parámetro indica el número máximo de características ya que cuando convertimos palabras a números empleando

la ténica de bolsa de palabras todas las palabras únicas son convertidas en “características”. Esto significa que queremos usar las 1500 palabras más comunes como características para entrenar nuestro clasificador.

Los dos siguientes parámetros indican el mínimo número de bloques que deben contener cierta característica y el máximo respectivamente. Finalmente, eliminamos las “stop words” que no son más que palabras que no aportan valor a la muestra

```
from sklearn.feature_extraction.text import TfidfTransformer
from sklearn.feature_extraction.text import TfidfVectorizer

tfidfconverter = TfidfVectorizer(max_features=1500, min_df=5, max_df=0.7, stop_words=stopwords.words('spanish'))
X = tfidfconverter.fit_transform(documents).toarray()

```

En el script de arriba convertimos la bolsa de palabras a valores TFIDF que es una medida numérica que expresa cuán relevante es una palabra en un documento en una colección.

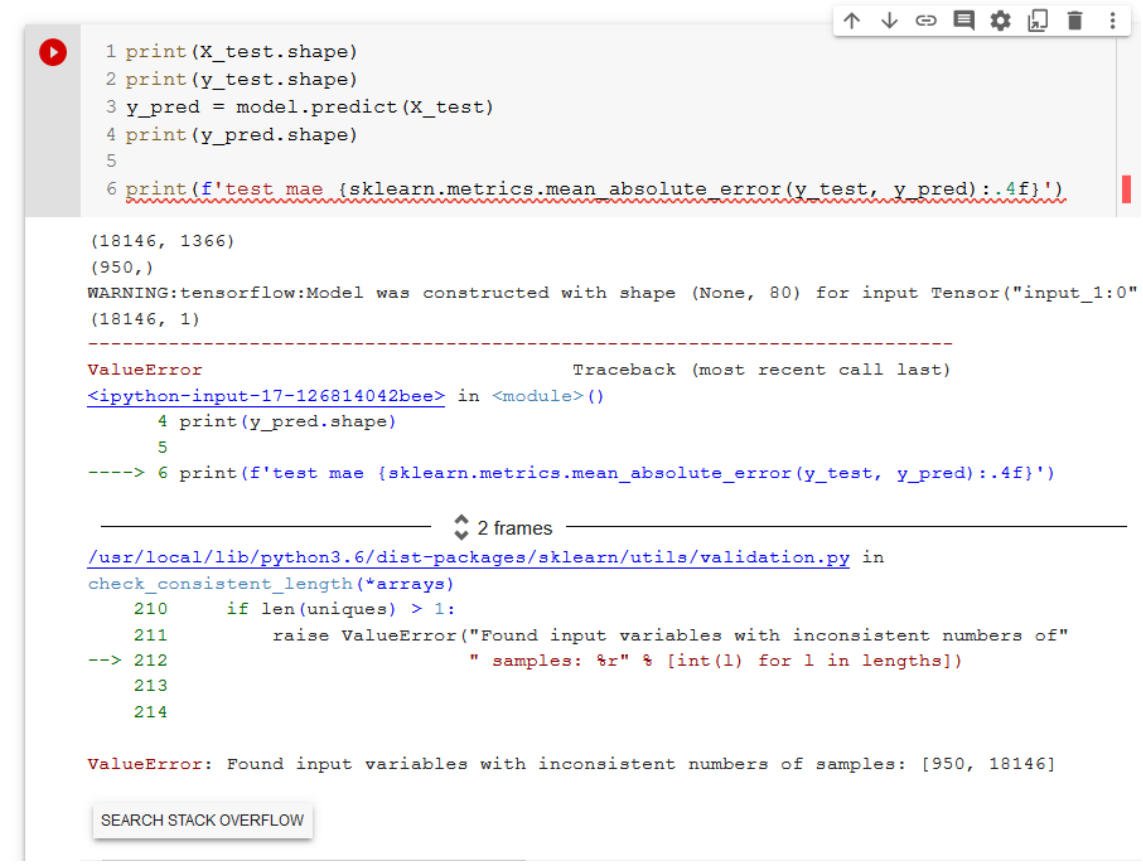
```

from sklearn.model_selection import train_test_split

X_train, X_test = X[:n], X[n:]
y_train, y_test = y[:n], y[n:]
print(np.shape(X))
input_shape = X.shape[1:]


model2 = keras.Input(shape=input_shape)
x = layers.Embedding(max_features, embedding_dims, input_length=maxlen)(model2)
x = layers.Dropout(0.3)(x)
x = layers.LSTM(units=64, return_sequences=True)(x)
x = layers.LSTM(units=64)(x)
outputs = layers.Dense(1, activation='linear')(x)
model2 = keras.Model(model2, outputs, name='model')

model2.compile(optimizer = 'adam', loss = 'mean_absolute_error')
model2.fit(X_train, y_train, epochs = 1, validation_split=0.1, batch_size = 32)

```

Finalmente, ya solo queda repartir los conjuntos de test y de entrenamiento, construir nuestro modelo, compilarlo y entrenarlo.

```
[nltk_data] Downloading package wordnet to /root/nltk_data...
[nltk_data]   Package wordnet is already up-to-date!
(26696, 894)
(26696, 1366)
241/241 [==============================] - 430s 2s/step - loss: 0.5684 - val_loss: 0.4815
<tensorflow.python.keras.callbacks.History at 0x7fa3da62eba8>
```

Predicciones

A la hora de realizar las predicciones no he conseguido hacer que funcione ya que de alguna manera el número de predicciones no se corresponde con el número de entradas. No he conseguido solucionarlo:



1.4. Evaluación de las redes por actividad (1 o 2)

En la muestra de datos existen 2 actividades. La actividad 1 y la actividad 2. Al tratarse de actividades diferentes e independientes lo más lógico es entrenar modelos de redes neuronales independientes para cada uno. Para ello, lo primero será convertir los datos de entrenamiento a la forma correcta. Esto ya viene en el ejemplo para la actividad 1 pero necesitamos la actividad 2 por separado.

Para elegir la actividad que queremos tan solo tendremos que cambiar la variable activity por un 2:

```
from tensorflow.keras import preprocessing

# Seleccionar la actividad 1 y sus secciones, ya que se evaluan por
activity = 2
new_data = data[(data['activity']==activity) & data['section'].isin
tokenizer = preprocessing.text.Tokenizer()
tokenizer.fit_on_texts(new_data['feedback prep'])
max_features = 1 + len(tokenizer.word_index)
maxlen = 80

X = preprocessing.sequence.pad_sequences(tokenizer.texts_to_sequenc

y = new_data['value'].values

embedding_dims = 10

```

De esta forma nos aseguramos de que en data['activity'] estamos cargando la actividad 2.

A continuación, entrenamos el modelo durante 10 épocas y empleamos también la validación cruzada igual que antes. Y visualizamos los resultados.

```
print(X.shape)
X_test_new = new_data[12046:][['feedback','value', 'activity']].copy()
X_test_new['value pred'] = y_pred.ravel()
X_test_new.round(1)
```

```
(13384, 80)
```

	feedback	value	activity	value pred
16541	El fondo de arriba me parece demasiado oscuro,...	2.0	2	3.0
16491	nada que objetar	3.0	2	2.5
21725	La imagen se corresponde bastante bien y el te...	2.5	2	3.0
15002	Ninguna sugerencia	3.0	2	3.0
16643	Todo bien.	3.0	2	3.0
...	...	...	...	...
15377	Actividades muy completas y originales.	3.0	2	3.0
21602	Está bien.	3.0	2	3.0
17730	falta enlace, mejorar diseño y contenidos	2.5	2	2.2
15725	Falta una imágen	2.5	2	3.0
19966	ninguna	3.0	2	2.7

1338 rows × 4 columns

Hemos tenido que adaptar también el nuevo set de datos de prueba de la X al nuevo tamaño que en este caso es 12046. Ahora podemos ver también como en la columna 'activity' solo se trata de la actividad 2. Vamos a realizar la misma operación para la actividad 1. Para ello solo tendremos que cambiar el valor de la variable activity e inicializarlo a 1. A continuación, entrenamos nuestra red durante 10 épocas, igual que antes y visualizamos los resultados:

Inicializamos la actividad a 1:

```
from tensorflow.keras import preprocessing

# Seleccionar la actividad 1 y sus secciones, ya que se evaluan por separado. La
activity = 1
new_data = data[(data['activity']==activity) & data['section'].isin(['1','2','3'
tokenizer = preprocessing.text.Tokenizer()
tokenizer.fit_on_texts(new_data['feedback prep'])
max_features = 1 + len(tokenizer.word_index)
maxlen = 80

X = preprocessing.sequence.pad_sequences(tokenizer.texts_to_sequences(new_data['

y = new_data['value'].values

embedding_dims = 10

```

Compilamos, entrenamos y visualizamos los resultados:

Predicciones con el test

Las predicciones se realizan entre 0 y 3 que son los valores mínimos y máximo establecidos para cada valorar cada ítem.

```
import sklearn

y_pred = np.clip(model.predict(X_test), 0, 3)

print(f'test mae {sklearn.metrics.mean_absolute_error(y_test, y_pred):.4f}')
```

```
test mae 0.2975
```

```
1
2 X_test_new = new_data[n:][['feedback','value', 'activity']].copy()
3 X_test_new['value pred'] = y_pred.ravel()
4 X_test_new.round(1)
```

(9500, 80)

	feedback	value	activity	value pred
9412	Ningún problema con la imagen	3.0	1	3.0
899	Todo está bien. Además, me ha parecido muy bie...	3.0	1	1.5
3751	Perfecto todo	3.0	1	2.7
6047		2.0	1	2.1
4021	No aparece ninguna licencia en el power point,...	1.0	1	3.0
...	...	...	...	...
111		3.0	1	3.0
942	No indica el tipo de licencia en la presentación	1.7	1	2.8
5664	Aburrida y sin originalidad	3.0	1	2.7
96	ok	3.0	1	3.0
7763	No se ha citado ni indicado correctamente, la ...	1.0	1	2.0

950 rows × 4 columns

Como vemos, ahora solo estamos tratando con la actividad 1.

Predicciones con el test

Las predicciones se realizan entre 0 y 3 que son los valores mínimos y máximo establecidos para cada valorar cada ítem.

```
[99] 1 import sklearn
     2
     3 y_pred = np.clip(model.predict(X_test), 0, 3)
     4
     5 print(f'test mae {sklearn.metrics.mean_absolute_error(y_test, y_pred):.4f}')
```

```
test mae 0.2118
```

1.5. Estudios comparativos usando test estadísticos (Wilcoxon)

Para realizar el estudio de Wilcoxon necesitaremos crear un segundo modelo. El test de Wilcoxon es importante en aprendizaje automático ya que es muy común verificar que dos experimentos distintos son equivalentes, o bien uno es mejor que otro. Lo primero será diseñar la red neuronal.

▾ WILCOXON RANKED TEST

Tercer modelo

```
input_shape = X.shape[1:]


model3 = keras.Input(shape=input_shape)
x = layers.Embedding(max_features, embedding_dims, input_length=maxlen)(model3)
x = layers.Dropout(0.2)(x)
x = layers.GRU(units=256, return_sequences=True)(x)
x = layers.GRU(units=128)(x)
x = layers.Dense(256)(x)
outputs = layers.Dense(1, activation='linear')(x)
model3 = keras.Model(model3, outputs, name='model3')

model3.compile(optimizer = 'adam', loss = 'mean_absolute_error', metrics=['mean_squared_error'])

# Para verificar que el modelo entrena correctamente crearemos un conjunto de validación del 10% de los datos con el parámetro ()
model3.fit(X_train, y_train, epochs = 15, validation_split=0.1, batch_size = 32)
scores3 = model3.evaluate(X_test, y_test, verbose=1)
y_pred = np.clip(model3.predict(X_test), 0, 3)
print(f'test mae {sklearn.metrics.mean_absolute_error(y_test, y_pred):.4f}')
```

Una vez tenemos la red neuronal la comparamos con el primer modelo que recordemos nos había arrojado estos valores

Una vez tenemos el modelo 3 definido tendremos que entrenarlo para el conjunto de datos que tenemos:

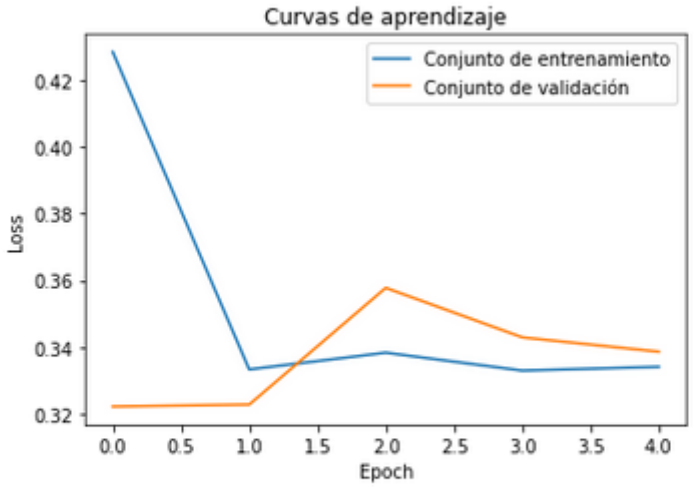
Tercer modelo

```
import tensorflow as tf



#Resultados del tercer clasificador
cvscores3 = []

cv = KFold(n_splits = 10, random_state=42, shuffle=True)
for train_index, test_index in cv.split(X):
  X_train, X_test = X[train_index], X[test_index]
  y_train, y_test = y[train_index], y[test_index]

  input_shape = X.shape[1:]

  model3 = cnn_model3(input_shape)
  history = model3.fit(X_train, y_train, epochs = 5, validation_split=0.1, batch_size = 32)
  scores3 = model3.evaluate(X_test, y_test, verbose=1)

  print("%s: %.2f%%" % (model3.metrics_names[1], scores3[1] * 100))
  cvscores3.append(scores3[1] * 100)

print('Mostramos las curvas de aprendizaje')
plot_learning_curves(history)

print("%.2f%% (+/-%.2f%%)" % (np.mean(cvscores3), np.std(cvscores3)))
ResultadosTercerClasificador = cvscores3

print("%.2f%% (+/-%.2f%%)" % (np.mean(cvscores3), np.std(cvscores3)))
ResultadosTercerClasificador = cvscores3
y_pred = np.clip(model3.predict(X_test), 0, 3)
print(f'test mae {sklearn.metrics.mean_absolute_error(y_test, y_pred):.4f}')



```

La forma de entrenarlo es totalmente la misma que la del primer modelo

El modelo 3 nos arroja los siguientes resultados tras ser entrenado

```
Mostramos las curvas de aprendizaje
```




```
19.84% (+/-4.32%)
19.84% (+/-4.32%)
test mae 0.2991
```

```
[26]  1 from scipy.stats import wilcoxon
      2 import warnings
      3 warnings.filterwarnings('ignore')
      4
      5
      6 wilcox_V, p_value =  wilcoxon(ResultadosPrimerClasificador, ResultadosTercerClasificador, alternative='greater', zero_method='wilcox', correction=False)
      7
      8 print('Resultado completo del test de Wilcoxon')
      9 print(f'Wilcox V: {wilcox_V}, p-value: {p_value:.2f}')

    Resultado completo del test de Wilcoxon
    Wilcox V: 55.0, p-value: 0.00
```

Como vemos, la probabilidad de que el primer clasificador sea peor que el tercero es del 100%, esto es obvio ya que el segundo clasificador obtiene casi la mitad del error absoluto que el clasificador 1.

2. Nivel medio

2.1. Arquitecturas de redes adicionales a la básica y ajustes combinados

Redes GRU. Las redes GRU (Gated Recurrent Units) son un mecanismo de compuerta que se comporta como una LSTM pero con una compuerta de olvido. Tiene menos parámetros que una LSTM ya que carece de una salida. Las GRU's han demostrada ofrecer un mejor desempeño en algunos datasets más pequeños y menos frecuentes.

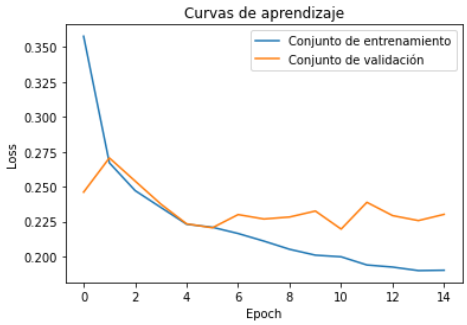
Tercer modelo

```
input_shape = X.shape[1:]



model3 = keras.Input(shape=input_shape)
x = layers.Embedding(max_features, embedding_dims, input_length=maxlen)(model3)
x = layers.Dropout(0.2)(x)
x = layers.GRU(units=256, return_sequences=True)(x)
x = layers.GRU(units=128)(x)
x = layers.Dense(256)(x)
x = layers.Dense(1)(x)
outputs = layers.Dense(1, activation='linear')(x)
model3 = keras.Model(model3, outputs, name='model3')

model3.compile(optimizer = 'adam', loss = 'mean_absolute_error', metrics=['mean_squared_error'])

# Para verificar que el modelo entrena correctamente crearemos un conjunto de validación del 10% de los datos con el parámetro ()
model3.fit(X_train, y_train, epochs = 15, validation_split=0.1, batch_size = 32)
scores3 = model3.evaluate(X_test, y_test, verbose=1)
y_pred = np.clip(model3.predict(X_test), 0, 3)
print(f'test mae {sklearn.metrics.mean_absolute_error(y_test, y_pred):.4f}')



```

En el ejemplo de arriba podemos ver como se añadirían algunas capas GRU en vez de las LSTM que hemos estado utilizando. Vamos a compararlo con las LSTM.

En el caso de la GRY vemos como nos ofrece un error cuadrático medio de 0.1995. El error cuadrático medio mide el promedio de los errores al cuadrado. Es decir, la diferencia entre el estimador y lo que estima.

```
Mostramos las curvas de aprendizaje
```



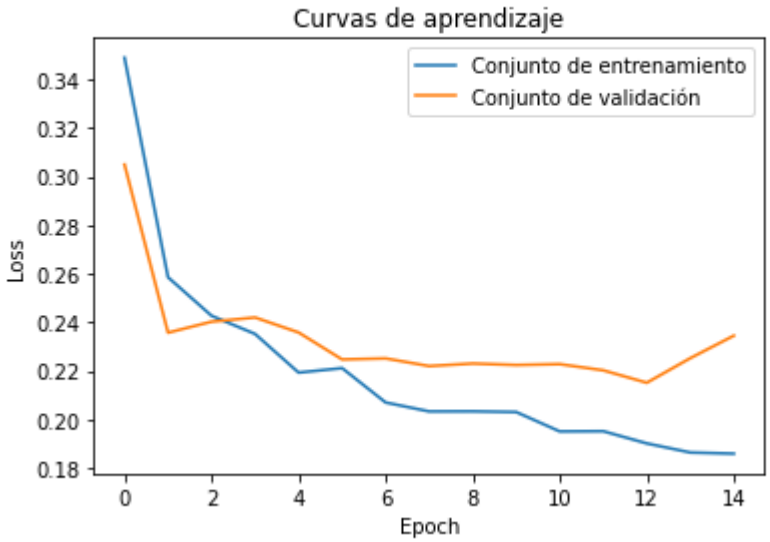

```
42/42 [==============================] - 0s 5ms/step - loss: 0.2133 - mean_squared_error: 0.1325
test mae 0.2131
```

```
#Resultados del tercer clasificador
cvscores3 = []

print("%s: %.2f%%" % (model3.metrics_names[1], scores3[1] * 100))
cvscores3.append(scores3[1] * 100)
print("%.2f%% (+/-%.2f%%)" % (np.mean(cvscores3), np.std(cvscores3)))
ResultadosTercerClasificador = cvscores3
```

```
mean_squared_error: 19.95%
19.95% (+/-0.00%)
```

En el caso de la LSTM nos ofrece lo siguiente:

```
339/339 [==============================] - 4s 12ms/step - los
Mostramos las curvas de aprendizaje
```




```python
scores = model.evaluate(X_test, y_test, verbose=1)
y_pred = np.clip(model.predict(X_test), 0, 3)
print(f'test mae {sklearn.metrics.mean_absolute_error(y_test, y_pred):.4f}')

#Resultados del primer clasificador
cvscores = []

print("%s: %.2f%%" % (model.metrics_names[1], scores[1] * 100))
cvscores.append(scores[1] * 100)
print("%.2f%% (+/-%.2f%%)" % (np.mean(cvscores), np.std(cvscores)))
ResultadosPrimerClasificador = cvscores
```

```
42/42 [==============================] - 0s 5ms/step - loss: 0.2157 - mean_squared_error: 0.1368
test mae 0.2157
mean_squared_error: 13.68%
13.68% (+/-0.00%)
```

Como podemos observar, el error cuadrático medio de la LSTM es inferior a la de la GRU. Además, parece que en el caso de aumentar las épocas la LSTM sería capaz de aprender más.

Vamos a ver también de aplicar las redes cuda CuDNNLSTM y CuDNNGRU ya que son versiones optimizadas para CUDA y deberían funcionar más rápido. Vamos a realizar la misma comparativa para ambas redes pero usando su versión optimizada para CUDA.

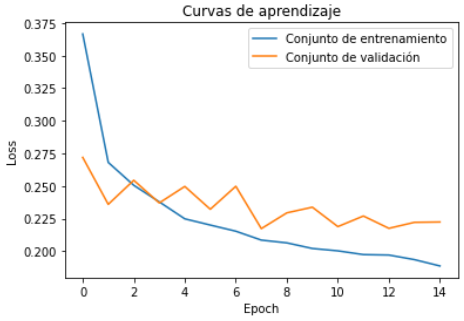
En el caso de la GRU la red quedaría de la siguiente forma:

Tercer modelo

```
def cnn_model3(input_shape):

  model3 = keras.Input(shape=input_shape)
  x = layers.Embedding(max_features, embedding_dims, input_length=maxlen)(model3)
  x = layers.Dropout(0.2)(x)
  x = tf.compat.v1.keras.layers.CuDNNGRU(units=256, return_sequences=True)(x)
  x = tf.compat.v1.keras.layers.CuDNNGRU(units=128)(x)
  x = layers.Dense(256)(x)
  x = layers.Dense(1)(x)
  outputs = layers.Dense(1, activation='linear')(x)
  model3 = keras.Model(model3, outputs, name='model3')

  model3.compile(optimizer = 'adam', loss = 'mean_absolute_error', metrics=['mean_squared_error'])

  return model3
```

A continuación, se procede a entrenar la red, la cual ahora entrena más rápido debido a su optimización para GPU. La red nos arroja el siguiente resultado:

```
Mostramos las curvas de aprendizaje
```




```
42/42 [==============================] - 4s 87ms/step - loss: 0.2116 - mean_squared_error: 0.1493
test mae 0.2035
```

```
#Resultados del tercer clasificador
cvscores3 = []

print("%s: %.2f%%" % (model3.metrics_names[1], scores3[1] * 100))
cvscores3.append(scores3[1] * 100)
print("%.2f%% (+/-%.2f%%)" % (np.mean(cvscores3), np.std(cvscores3)))
ResultadosTercerClasificador = cvscores3
```

```
mean_squared_error: 14.93%
14.93% (+/-0.00%)
```

En el caso de la LSTM la red quedaría de la siguiente forma:

```
from tensorflow import keras
from tensorflow.keras.models import Sequential
from tensorflow.keras import layers

input_shape = X.shape[1:]
model = keras.Input(shape=input_shape)
x = layers.Embedding(max_features, embedding_dims, input_length=maxlen)(model)
x = layers.Dropout(0.2)(x)
x = tf.compat.v1.keras.layers.CuDNNLSTM(units=256, return_sequences=True)(x)
x = tf.compat.v1.keras.layers.CuDNNLSTM(units=128)(x)
x = layers.Dense(256)(x)
x = layers.Dense(1)(x)
outputs = layers.Dense(1, activation='linear')(x)
model = keras.Model(model, outputs, name='model')



model.compile(optimizer = 'adam', loss = 'mean_absolute_error', metrics=['mean_squared_error'])

```

Nuevamente, entrenamos el modelo de nuevo el cual entrena más rápido debido a la optimización y mostramos los resultados:

Modelo 1

```
def cnn_model1(input_shape):
  model = keras.Input(shape=input_shape)
  x = layers.Embedding(max_features, embedding_dims, input_length=maxlen)(model)
  x = layers.Dropout(0.2)(x)
  x = tf.compat.v1.keras.layers.CuDNNLSTM(units=256, return_sequences=True)(x)
  x = tf.compat.v1.keras.layers.CuDNNLSTM(units=128)(x)
  x = layers.Dense(256)(x)
  x = layers.Dense(1)(x)
  outputs = layers.Dense(1, activation='linear')(x)
  model = keras.Model(model, outputs, name='model')
  model.compile(optimizer = 'RMSprop', loss = 'mean_absolute_error', metrics=['mean_squared_error'])

  return model


```

```
scores = model.evaluate(X_test, y_test, verbose=1)
y_pred = np.clip(model.predict(X_test), 0, 3)
print(f'test mae {sklearn.metrics.mean_absolute_error(y_test, y_pred):.4f}')

#Resultados del primer clasificador
cvscores = []

print("%s: %.2f%%" % (model.metrics_names[1], scores[1] * 100))
cvscores.append(scores[1] * 100)
print("%.2f%% (+/-%.2f%%)" % (np.mean(cvscores), np.std(cvscores)))
ResultadosPrimerClasificador = cvscores
```

```
42/42 [==============================] - 4s 96ms/step - loss: 0.1947 - mean_squared_error: 0.1459
test mae 0.1945
mean_squared_error: 14.59%
14.59% (+/-0.00%)
```

En este caso la LSTM parece obtener un mejor resultado en el error medio cuadrático.

2.2. Preprocesado avanzado

Aparte del Preprocesado básico del texto que hemos realizado en el apartado básico existen algunas herramientas adicionales a la hora de preprocesar el texto. Una de ellas es conocida por "Stopword removal". Las *stop words* son un conjunto de palabras comúnmente usadas en un cierto idioma. Por ejemplo: las preposiciones, demostrativos, conjunciones etc…. Algunos ejemplos concretos: y, ante, bajo, eso, ellos…

Estos conjuntos de palabras representan información pobre y ruido en la muestra que puede ser omitido. Aunque en ciertas ocasiones demuestra no ser algo crítico a la hora de evaluar los resultados de nuestro modelo, ayuda a mantener un tamaño contenido de las características y que el modelo no crezca en exceso. Vamos a ver que es muy fácil de hacer:

```
from sklearn.feature_extraction.text import CountVectorizer

from sklearn.feature_extraction.text import TfidfTransformer
from sklearn.feature_extraction.text import TfidfVectorizer

tfidfconverter = TfidfVectorizer(max_features=1500, min_df=5, max_df=0.7, stop_words=stopwords.words('spanish'))
X = tfidfconverter.fit_transform(documents).toarray()
```

Aprovechando la implementación del bag of words hicimos un nuevo Preprocesado a partir del del elemento data.feedback. En el método TfidVectorizer existe un parámetro que es stop_words que sirve para eliminar esta clase de palabras de los datos.

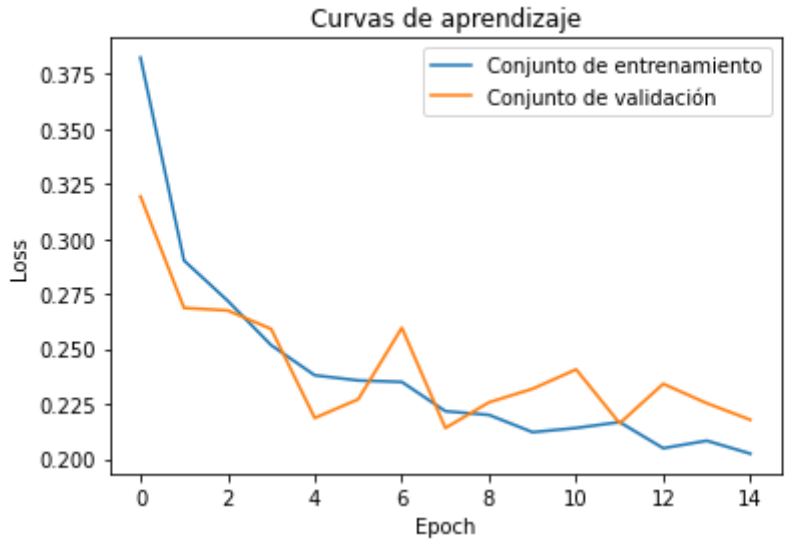
2.3. Ajuste de algoritmo de optimización (mejores resultados con menos épocas)

A continuación, vamos a trabajar con el algoritmo de optimización sobre el modelo de red LSTM optimizado para CUDA. Siguiendo con el *modus operandi* del apartado homólogo del desafío 1, vamos a realizar diferentes pruebas con algoritmos de optimización conocidos, vamos a representar su gráfica de aprendizaje y finalmente, los clasificaremos por orden de mejor a peor en una tabla. Los algoritmos que vamos a estudiar son:

- Adam
- Adadelta
- Adagrad
- SGD (stochastic gradient descent)
- RMSprop

Comenzando por Adam tenemos el resultado que hemos expresado justamente en el primer apartado de la parte media.

```
Mostramos las curvas de aprendizaje
```

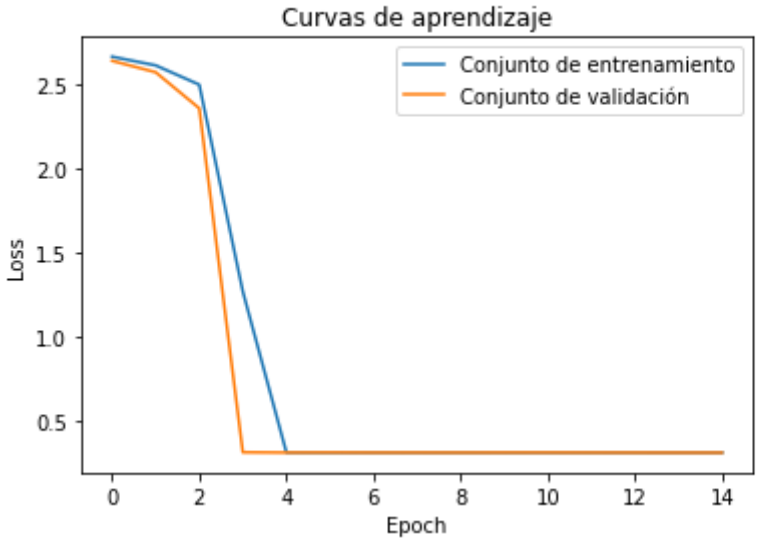


Para 15 épocas nos ofrece un error absoluto medio del 0.1945.

Para el optimizador Adadelta:

```
18 model.compile(optimizer = 'ada', loss = 'mean_absolute_error', metrics=['mean_squared_error'])
19
```

```
Mostramos las curvas de aprendizaje
```



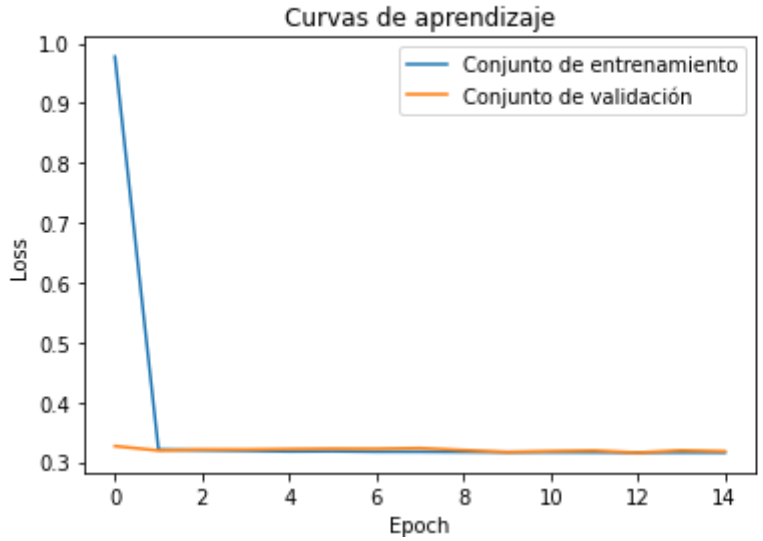
Se puede ver un descenso muy acusado del valor de perdida pero el entrenamiento se estanca a partir de la época 4 así que no tiene sentido aumentar épocas. Este optmizador arroja un error absoluto medio de:

```
1 scores = model.evaluate(X_test, y_test, verbose=1)
2 y_pred = np.clip(model.predict(X_test), 0, 3)
3 print(f'test mae {sklearn.metrics.mean_absolute_error(y_test, y_pred):.4f}')
4
5 #Resultados del primer clasificador
6 cvscores = []
7
8 print("%s: %.2f%%" % (model.metrics_names[1], scores[1] * 100))
9 cvscores.append(scores[1] * 100)
10 print("%.2f%% (+/-%.2f%%)" % (np.mean(cvscores), np.std(cvscores)))
11 ResultadosPrimerClasificador = cvscores
```

```
42/42 [==============================] - 0s 7ms/step - loss: 0.2722 - mean_squared_error: 0.2634
test mae 0.2721
mean_squared_error: 26.34%
26.34% (+/-0.00%)
```

Para el optimizador Adagrad:

Mostramos las curvas de aprendizaje

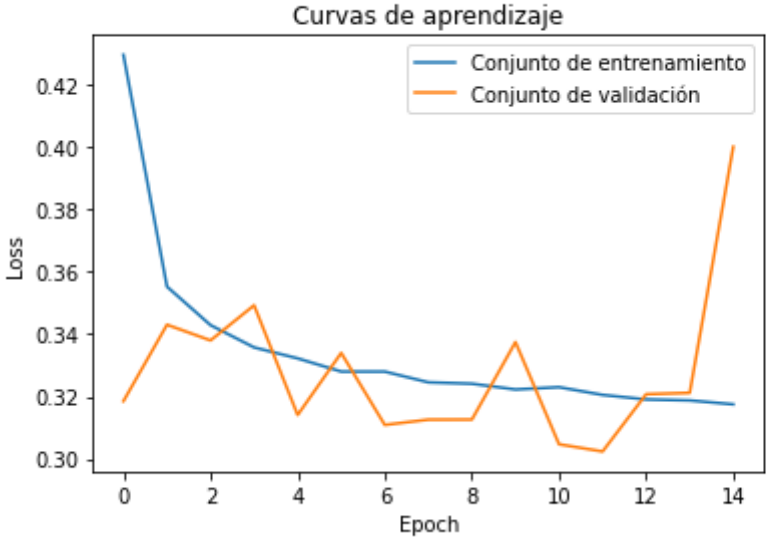



```
scores = model.evaluate(X_test, y_test, verbose=1)
y_pred = np.clip(model.predict(X_test), 0, 3)
print(f'test mae {sklearn.metrics.mean_absolute_error(y_test, y_pred):.4f}')

#Resultados del primer clasificador
cvscores = []

print("%s: %.2f%%" % (model.metrics_names[1], scores[1] * 100))
cvscores.append(scores[1] * 100)
print("%.2f%% (+/-%.2f%%)" % (np.mean(cvscores), np.std(cvscores)))
ResultadosPrimerClasificador = cvscores
```

```
42/42 [==============================] - 0s 7ms/step - loss: 0.2758 - mean_squared_error: 0.2746
test mae 0.2758
mean_squared_error: 27.46%
27.46% (+/-0.00%)
```

Para el optimizador SGD

Mostramos las curvas de aprendizaje

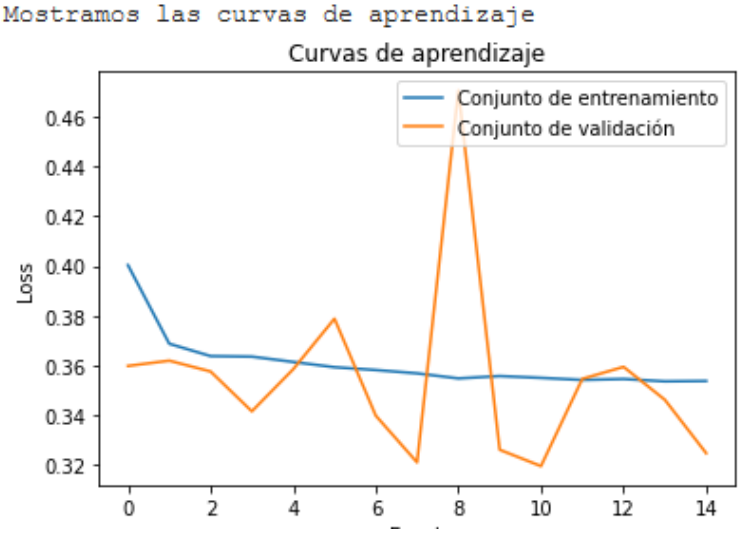


```
scores = model.evaluate(X_test, y_test, verbose=1)
y_pred = np.clip(model.predict(X_test), 0, 3)
print(f'test mae {sklearn.metrics.mean_absolute_error(y_test, y_pred):.4f}')

#Resultados del primer clasificador
cvscores = []

print("%s: %.2f%%" % (model.metrics_names[1], scores[1] * 100))
cvscores.append(scores[1] * 100)
print("%.2f%% (+/-%.2f%%)" % (np.mean(cvscores), np.std(cvscores)))
ResultadosPrimerClasificador = cvscores
```

```
42/42 [==============================] - 0s 7ms/step - loss: 0.3623 - mean_squared_error: 0.2988
test mae 0.2638
mean_squared_error: 29.88%
29.88% (+/-0.00%)
```

Para el optimizador RMSprop

```
Mostramos las curvas de aprendizaje
```




```
scores = model.evaluate(X_test, y_test, verbose=1)
y_pred = np.clip(model.predict(X_test), 0, 3)
print(f'test mae {sklearn.metrics.mean_absolute_error(y_test, y_pred):.4f}')

#Resultados del primer clasificador
cvscores = []

print("%s: %.2f%%" % (model.metrics_names[1], scores[1] * 100))
cvscores.append(scores[1] * 100)
print("%.2f%% (+/-%.2f%%)" % (np.mean(cvscores), np.std(cvscores)))
ResultadosPrimerClasificador = cvscores
```

```
42/42 [==============================] - 0s 8ms/step - loss: 0.2820 - mean_squared_error: 0.2744
test mae 0.2820
mean_squared_error: 27.44%
27.44% (+/-0.00%)
```

Posiciones finales

Algoritmo	Error absoluto medio
1º Adam	0.1945
2º SGD	0.2638
3º Adadelta	0.2721
4º Adagrad	0.2758
5º RMSprop	0.2820

Como se puede apreciar, el mejor algoritmo de optimización es el Adam en este caso. No es de extrañar ya que Adam es un de los algoritmos más utilizados en aprendizaje automático.

2.4. Notación funcional del API de Keras

```
from tensorflow import keras
from tensorflow.keras.models import Sequential
from tensorflow.keras import layers

input_shape = X.shape[1:]
model = keras.Input(shape=input_shape)
x = layers.Embedding(max_features, embedding_dims, input_length=maxlen)(model)
x = layers.Dropout(0.2)(x)
x = layers.LSTM(units=64, return_sequences=True)(x)
x = layers.LSTM(units=64)(x)
outputs = layers.Dense(1, activation='linear')(x)
model = keras.Model(model, outputs, name='model')



model.compile(optimizer = 'adam', loss = 'mean_absolute_error', metrics=['mean_squared_error'])

```

Modelo para actividad 2

```
input_shape = X.shape[1:]
model = keras.Input(shape=input_shape)
x = layers.Embedding(max_features, embedding_dims, input_length=maxlen)(model)
x = layers.Dropout(0.2)(x)
x = layers.LSTM(units=64, return_sequences=True)(x)
x = layers.LSTM(units=64)(x)
outputs = layers.Dense(1, activation='linear')(x)
model = keras.Model(model, outputs, name='model')



model.compile(optimizer = 'adam', loss = 'mean_absolute_error')
```

```
input_shape = X.shape[1:]


model3 = keras.Input(shape=input_shape)
x = layers.Embedding(max_features, embedding_dims, input_length=maxlen)(model3)
x = layers.Dropout(0.2)(x)
x = layers.GRU(units=256, return_sequences=True)(x)
x = layers.GRU(units=128)(x)
x = layers.Dense(256)(x)
x = layers.Dense(1)(x)
outputs = layers.Dense(1, activation='linear')(x)
model3 = keras.Model(model3, outputs, name='model3')

model3.compile(optimizer = 'adam', loss = 'mean_absolute_error', metrics=['mean_squared_error'])
```

2.5. Detección de palabras en las que se centra el modelo para explicar su predicción

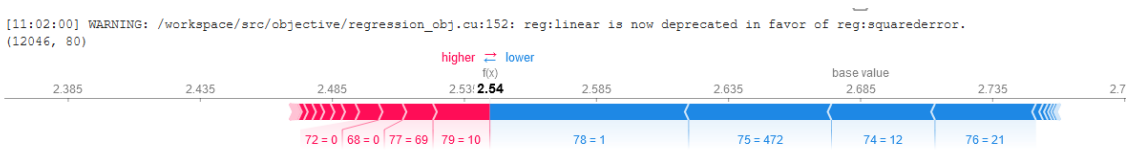
Para ver en que valores se centra el modelo podemos utilizar la librería SHAPLEY que no es más que un enfoque teórico para explicar el resultado de cualquier modelo de aprendizaje. Esta librería conecta la asignación óptima de créditos con explicaciones locales utilizando valores de teoría de juegos.

Lo primero será descargar shap y a continuación importar el modelo xboost y la liberería shaplety.

▸ SHAPLEY

```
!pip install shap
import xgboost
import shap

# load JS visualization code to notebook
shap.initjs()

from sklearn.model_selection import KFold
n = int(len(new_data)*0.9)

cv = KFold(n_splits = 10, random_state=42, shuffle=True)
for train_index, test_index in cv.split(X):
  X_train, X_test = X[train_index], X[test_index]
  y_train, y_test = y[train_index], y[test_index]

model4 = xgboost.XGBRegressor(3, 0.1, 100, 1)
model4.fit(X=X_train, y=y_train)
# explain the model's predictions using SHAP
# (same syntax works for LightGBM, CatBoost, scikit-learn and spark models)
explainer = shap.TreeExplainer(model4)
shap_values = explainer.shap_values(X)

print(X_train.shape)
X_train_df = pd.DataFrame(data=X_train)
# visualize the first prediction's explanation (use matplotlib=True to avoid Javascript)
shap.force_plot(explainer.expected_value, shap_values[0,:], X_train_df.iloc[0,:])
```

Hemos utilizado un regressor XGB ya que los modelos personalizados que hemos ido realizando a lo largo de la práctica no están soportados por la liberería SHAPLEY.



Aquí podemos ver que palabras utiliza el modelo para basarse en su resultado. A cada uno de los números de la zona superior le correspondería una palabra pero no he sabido hacer para que se presenten palabras en vez de números.

2.6. Comparativas entre los diferentes ajustes con verificación de resultados

Este apartado se completa con las explicaciones ofrecidas en apartados anteriores ya que se exponen gráficas y resultados mostrando diferentes comparativas.

3. Bibliografía

- Cursos de Verano de la Universidad de Alicante 'Rafael Altamira' - Introducción al Deep Learning
- https://es.wikipedia.org/wiki/Validaci%C3%B3n_cruzada
- https://stackabuse.com/text-classification-with-python-and-scikit-learn/
- https://machinelearningmastery.com/nonparametric-statistical-significance-tests-in-python/
- https://towardsdatascience.com/recurrent-neural-networks-by-example-in-python-ffd204f99470https://towardsdatascience.com/recurrent-neural-networks-by-example-in-python-ffd204f99470
- https://www.tensorflow.org/guide/keras/functional?hl=es-419
- https://www.analyticsvidhya.com/blog/2019/11/shapley-value-machine-learning-interpretability-game-theory/