Sexism Identification in Social networks (EXIST 2021)

Trayectoria en el proyecto y exposición de resultados finales

Andrea García Pastor
Elizaveta Gilyarovskaya
Grado en Ciencia de Datos
19 de mayo 2021
Lenguaje Natural y Recuperación de la Información

Abstract. En la siguiente memoria se va a exponer la trayectoria seguida en el proyecto realizado como resultado de participar en la competición Exist 2021. Se han implementado modelos como SVM, BERT, Ensemble models... para clasificar los tweets en sexistas y no sexistas y para determinar el nivel de sexismo en algunos de ellos.
Aconsejamos fuertemente leer el notebook adjuntado con todo el código y comentarios necesarios para entender los pasos seguidos para llegar hasta el modelo final utilizado para predecir el conjunto test, cuyo resultado hemos enviado a la competición.

Índice de contenidos

1. Introducción

Al inscribirnos en el concurso como equipo " Andrea_Lisa", se nos proporcionó dos conjuntos de datos con tweets: uno de training y otro de test. Las muestras del dataset de training venían determinadas por una identificación, el idioma (español o inglés), la fuente (en este caso Twitter), el texto, la tarea1 y la tarea2. Respecto al test, solamente contenía el texto junto a su identificación. Detallaremos más el primero de ellos, cuyos atributos más importantes han sido: *Task1* y *Task2*.

La Task1 es una variable categórica que clasifica los tweets según sean sexistas o no lo sean. La tarea2 hace exactamente lo mismo pero de forma más específica, ya que los tweets aparecen clasificados según el tipo de sexismo: ideológico y desigualdad; estereotipos y dominación; objetificación; violencia sexual y misoginia y violencia no sexual. Y no sexista en el caso de que no sean sexistas.

Nuestro objetivo consiste en encontrar un modelo que consiga clasificar los tweets según la tarea 1 y la tarea 2 con el menor error posible.

2. Preproceso

En un primer momento decidimos entrenar un clasificador únicamente para los tweets ingleses, aunque al final obtenemos un modelo funcional para los dos idiomas. Por tanto filtramos por la variable 'language' aquellas muestras que estaban en inglés.

Debido al tiempo reducido, esta decisión viene motivada por el hecho de querer aprender a entrenar un clasificador para el Natural Language Processing independientemente del idioma. Es decir, nos interesaba dedicar más tiempo a probar diferentes estrategias y modelos de Natural Language Processing que dedicar tiempo a repetir el mismo modelo para dos idiomas distintos.

El preprocesamiento de los tweets va a consistir en quitar los signos de puntuación, los dígitos, las urls, las menciones, los hashtags, los emoticonos y las stopwords. Las funciones implementadas utilizan distintas librerías, tales como *expresiones regulares, nltk, string*, entre otras. Podemos observar el resultado para un tweet en concreto.

```
Tweet real: @Banyosssss My philosophy: He has a funny mustache what else does a man need
Tweet preprocesado: philosophy funny mustache else man need
```

Imagen 2.1 Resultado del preproceso

3. Extracción de características

A continuación, probaremos diferentes métodos para recodificar o transformar el input para futuros modelos. En otras palabras, el texto procesado de longitud variable se convertirá en vectores de características numéricas de tamaño fijo. Lo haremos mediante cuatro estrategias: Bolsa de palabras (Bag-of-Words), N-gramas de palabras, N-gramas de caracteres, Word embeddings y Codificación TF-IDF.

No entramos en detalle en las 3 primeras, ya que al final se ha optado por utilizar TF-IDF como método de codificación del texto preprocesado.

3.1 Codificación TF-IDF

Expresa cuán relevante es una palabra para un texto en un corpus. De esta forma, el valor de un término aumenta proporcionalmente al número de veces que aparece en el texto y es compensado por su frecuencia en el corpus.

Hemos decidido utilizarla con TfidfVectorizer, pues nos permite aplicarla directamente a un tweet entero, consiguiendo vectores de la misma longitud. El beneficio es que nos ahorramos aplicar una codificación palabra por palabra y luego tener que juntarlas todas en vectores que pueden salir muy poco representativos y de longitud variable.

Al igual que el atributo 'texto', las variables *Task1* y *Task2* deben ser convertidas a tipo numérico. En este caso, se puede realizar de forma manual asignando un número {0, ..., n}, siendo n el número de valores de cada variable.

4. Construcción de clasificador SVM

4.1 Task 1

Previamente a la construcción de modelos, comprobar si se trataba de una tarea de clasificación de clases desbalanceadas. En este caso, consideramos que la clase 'sexist' y 'not-sexist' no lo estaban.

```
Counter({'non-sexist': 1800, 'sexist': 1636})
```

Imagen 4.1 Clases balanceadas

Entrenamos un modelo SVM probando diferentes kernels y valores de "C" para ver cuál es el más óptimo. Comparamos los resultados de los modelos con diferentes parámetros con una función personalizada.

```
Mejor modelo SVC(C=1, break_ties=False, cache_size=200, class_weight=None, coef0=0.0,
    decision_function_shape='ovr', degree=3, gamma='scale', kernel='linear',
    max_iter=-1, probability=False, random_state=None, shrinking=True,
    tol=0.001, verbose=False) con {'kernel': 'linear'}
El error obtenido del mejor modelo es: 0.2753232795098338
Mejor resultado (accuracy): 0.7246767204901662
```

Imagen 4.2 Resultados de la función de comparación Task1

El modelo SVM con el mejor accuracy (0.7246) lo da aquél con un kernel lineal, C=1 tras hacer validación cruzada con fold=5.

4.2 Task 2

El proceso para la tarea 2 es similar a la anterior. Aplicando la misma función para encontrar un modelo de SVM que mejor prediga los niveles de sexismo y, tras la validación cruzada, obtuvimos los siguientes resultados.

```
Mejor modelo SVC(C=1, break_ties=False, cache_size=200, class_weight=None, coef0=0.0,
    decision_function_shape='ovr', degree=3, gamma='scale', kernel='linear',
    max_iter=-1, probability=False, random_state=None, shrinking=True,
    tol=0.001, verbose=False) con {'kernel': 'linear'}
El error obtenido del mejor modelo es: 0.39580413662367564
Mejor resultado (accuracy): 0.6041958633763244
```

Imagen 4.3 Resultados de la función de comparación Task2

Como podemos comprobar, a parte del procedimiento, el mejor modelo (accuracy=0.6041) es el mismo: kernel lineal, C=1 y validación cruzada con fold=5.

Destacar que no se ha añadido ningún parámetro de "class_weight" ni se ha hecho ningún preproceso de over o undersampling. Por otro lado, también habíamos probado con perceptrón multicapa y regresión logística. Sin embargo, ambos modelos en ambas tareas daban un accuracy mucho menor y la red neuronal, además, tardaba demasiado en entrenarse.

5. BERT

El segundo modelo entrenado, y finalmente el utilizado para el concurso en ambas tareas, ha sido Bert. A diferencia de SVM, no necesitamos preprocesar los tweets, ya que Bert tiene su propio tokenizador. Por esta razón, utilizaremos los datos de training originales sin ninguna modificación.

Por otro lado, utilizando el modelo preentrenado Bert_Multilingual_cased, la librería Transformers de hugging face con Pytorch, nos permite realizar las tareas con los textos en ambos idiomas. Es decir, a diferencia de SVM, el modelo soporta texto tanto en inglés

como en español. Lo único que no varía es la conversión de las variables target en variables numéricas.
Destacar que utilizamos el siguiente tutorial como guía para conseguir implementarlo: https://www.youtube.com/watch?v=mvh7DV84mr4

```
RANDOM_SEED = 58
MAX_LEN = 160
BATCH_SIZE = 16
NCLASSES = 6
```

Imagen 5.1 Parámetros iniciales

Respecto al código, se decidió asignar el parámetro dropout a 0.1 en lugar de su valor por defecto, lo que nos ha ayudado a mejorar de forma significativa el accuracy. El dropout es una técnica (método de regularización) en la que se ignoran neuronas seleccionadas al azar durante el entrenamiento, es decir, se "abandonan" aleatoriamente. Esto significa que su contribución a la activación de las neuronas posteriores se elimina temporalmente en el paso hacia delante y no se aplica ninguna actualización de peso a la neurona en el paso hacia atrás. El efecto es que la red se vuelve menos sensible a los pesos específicos de las neuronas. A su vez, da lugar a una red que es capaz de generalizar mejor y es menos probable que sobreajuste los datos de entrenamiento.

```
class BERTSentimentClassifier(nn.Module):

  def __init__(self, n_classes):
    super(BERTSentimentClassifier, self).__init__()
    self.bert = BertModel.from_pretrained(PRE_TRAINED_MODEL_NAME)
    self.drop = nn.Dropout(p=0.1)
    self.linear = nn.Linear(self.bert.config.hidden_size, n_classes)

  def forward(self, input_ids, attention_mask):
    _, cls_output = self.bert(
        return_dict=False,
        input_ids = input_ids,
        attention_mask = attention_mask
    )
    drop_output = self.drop(cls_output)
    output = self.linear(drop_output)
    return output
```

Imagen 5.2 Modificación Dropout

También hemos decidido modificar el parámetro learning rate (lr) a *1e-5* en lugar de dejar el valor por defecto, que era más pequeño. El cambio ha provocado cambios positivamente significativos en el accuracy, ya que el modelo en el entrenamiento daba pasos más grandes. La tasa de aprendizaje (lr) es un hiper parámetro que controla cuánto cambiar el modelo en respuesta al error estimado cada vez que se actualizan los pesos del modelo (en cada *epoch*).

El parámetro de max_len está acotado a la longitud de los tokens obtenidos después de aplicarle Tokenizador del mismo modelo preentrenado a los tweets.

```
EPOCHS = 5
optimizer = AdamW(model.parameters(), lr=1e-5, correct_bias=False)
total_steps = len(train_data_loader) * EPOCHS
scheduler = get_linear_schedule_with_warmup(
    optimizer,
    num_warmup_steps = 0,
    num_training_steps = total_steps
)
loss_fn = nn.CrossEntropyLoss().to(device)
```

Imagen 5.3 Modificación Learning Rate

Realizando una partición train-test 0.8-0.2, y utilizando 5 epochs, los resultados obtenidos son mayores que con el modelo SVM para ambas tareas.

```
Entrenamiento: Loss: 0.11089324775872059, accuracy: 0.9650655021834061
Validación: Loss: 0.905277336232884, accuracy: 0.7863372093023255
```

Imagen 5.4 Resultados Task1

```
Entrenamiento: Loss: 0.5169852087781217, accuracy: 0.8342590933524457
Validación: Loss: 1.1962187757546252, accuracy: 0.6131805157593123
```

Imagen 5.5 Resultados Task2

6. StackingClassifier

Por último, vamos a probar la estrategia de Stacking para ver si mejoramos los resultados de Bert. La idea de Stacking implica el entrenamiento de un modelo para combinar las predicciones de otros modelos.

Se comienza entrenando varios modelos aprendices sobre los datos de entrenamiento (modelos de primer nivel), y se termina entrenando un modelo final (modelo de segundo nivel o stacking model) sobre los datos originales, considerando como características adicionales las predicciones de los primeros. Este modelo final es frecuentemente una regresión logística.

El rendimiento de este método de ensamblado aumenta cuanto más diversos sean los modelos de primer nivel, ya que cada uno de estos modelos explicará una parte diferente de la varianza de los datos originales.

Destacar que para el entrenamiento hemos tenido que pasar los tweets y las targets procesados de igual manera que en el modelo de SVM.

```
svm = SVC()
lr = LogisticRegression()
cart = DecisionTreeClassifier()
rf = RandomForestClassifier()
mlp = MLPClassifier(random_state=1, max_iter=300)
knn = KNeighborsClassifier()

estimators = [('svm', svm),('lr', lr),('cart', cart),('rf', rf),('mlp',mlp),('knn',knn)]

sclf = StackingClassifier(estimators=estimators, final_estimator=lr)
```

Imagen 6.1 Código de Stacking

A parte de obtener el accuracy para este nuevo método, también obtenemos los mejores resultados para cada uno de los modelos de primer nivel. Hay que destacar que el proceso se ha realizado por separado para las dos tareas.

En la imagen se observa que ninguno de los clasificadores supera al modelo preentrenado de Bert. Por esta razón se decidió entregar los runs con Bert.

```
Accuracy: 0.699359 [SVC]
Accuracy: 0.699651 [Logistic Regression]
Accuracy: 0.679574 [Decision Tree]
Accuracy: 0.712166 [Random Forest]
Accuracy: 0.648137 [MLPClassifier]
Accuracy: 0.628634 [KNN]
Accuracy: 0.726716 [StackingClassifier]
```

Imagen 6.2 Resultados para Task1

```
Accuracy: 0.561990 [SVC]
Accuracy: 0.569850 [Logistic Regression]
Accuracy: 0.550641 [Decision Tree]
Accuracy: 0.601572 [Random Forest]
Accuracy: 0.557625 [MLPClassifier]
Accuracy: 0.555586 [KNN]
```

Imagen 6.3 Resultados para Task2

Vamos a evaluar las predicciones de task2 con la métrica f1-macro, ya que es más robusta que el accuracy en este tipo de clasificaciones.

```
F1 score: 0.252657 [SVC]
F1 score: 0.286128 [Logistic Regression]
F1 score: 0.416182 [Decision Tree]
F1 score: 0.386311 [Random Forest]
F1 score: 0.375267 [MLPClassifier]
F1 score: 0.391629 [KNN]
```

Imagen 6.3 Resultados para Task2

Ya sea comparando el accuracy como el f1, el mejor modelo es el preentrenado de Bert. Por tanto, nos quedamos con BERT MULTILINGUAL para predecir ambas tareas.

7. Resultados finales

Una vez entregados los runs para cada tarea, los resultados obtenidos han sido los siguientes. Todo el código comentado del proyecto está en el siguiente google collab: https://colab.research.google.com/drive/1io-DFOuqa3x85erhHKoL9Y7t8FxlHCMc?usp=sharing

Task	Accuracy	F1	Posición
1	0.7186	0.7182	#34
2	0.6129	0.5204	#28