

Parte 1: Análisis exploratorio

Esta primera parte del trabajo práctico se centra en explorar el dataset e intentar entender el target que se buscará predecir.

Enlace al repositorio: <https://github.com/patricioibar/datos-tp3>

```
In [ ]: import random
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.ensemble import RandomForestClassifier
from sklearn.model_selection import train_test_split
from sklearn.metrics import f1_score
from wordcloud import WordCloud
import re

# suprimir future warnings
import warnings
warnings.simplefilter(action='ignore', category=FutureWarning)

SEED = 4237
np.random.seed(SEED)
random.seed(SEED)
CUSTOM_PALLETE = ['#00A7E1", "#FFA630"]
```

```
In [101...]: import pandas as pd
train_path = 'data/train.csv'
df = pd.read_csv(train_path)
random_sample = random.sample(range(len(df)), 10)
df.take(random_sample)
```

	id	keyword	location	text	target
299	440	apocalypse	NaN	The latest from @BryanSinger reveals #Storm is...	1
7458	10673	wounds	Alex/Mika/Leo 18 he/she/they	@ego_resolution im glad. My gay can heal 1000 ...	0
4160	5909	harm	England	@VileLunar I trickshot with a regular controll...	0
350	502	army	NaN	17.Beyonce Is my pick for http://t.co/thoYhrHk...	0
4468	6354	hostages	NaN	No #news of #hostages in #Libya\n\nhttp://t.co...	1
1704	2459	collided	Peterborough, On	#Newswatch: 2 vehicles collided at Lock and La...	1
4825	6870	mass%20murder	NaN	@guardian Has Japan ever truly come to terms w...	1
2835	4079	displaced	Oakland, CA	Historic flooding across Asia leaves hundreds ...	1
7512	10745	wreckage	khanna	Wreckage 'Conclusively Confirmed' as From MH37...	1
947	1372	blown%20up	Grimsby, England	My dog's just blown his kennel up Ù Bloody...	0

Se puede observar que el contenido del dataset es variado.

- En el campo `keyword` se pueden ver palabras clave posiblemente relacionadas con un desastre extraídas del tweet.
- El campo `location` se puede suponer que es la ubicación mencionada en el perfil del usuario que escribió el tweet. Es importante destacar que ese campo puede ser nulo o un string ingresado por el usuario, por lo que no solo puede contener ubicaciones con diferentes formatos y "granularidades" (por ejemplo podemos observar 'Grimsby, England', además de 'England'), sino que lo que contiene no necesariamente es una ubicación real (por ejemplo podemos encontrar edad, pronombres y lo que parecen ser nombres en la fila de id 10673).
- El campo `text` es el tweet en sí, como texto plano. Contiene menciones, hashtags, enlaces y otros caracteres unicode que se perdieron en alguna codificación. Todos los tweets que revisé están en inglés, por lo que supongo como mínimo que es el idioma predominante en el set de datos.

A continuación se pueden observar métricas del dataset.

```
In [102...]: print('Shape:', df.shape)
target_sum = df['target'].sum()
print(f"Cantidad de tweets \"Desastre\": {target_sum}")
print(f"Cantidad de tweets \"No Desastre\": {df['target'].count() - target_sum}")
df.describe()
```

Shape: (7613, 5)
Cantidad de tweets "Desastre": 3271
Cantidad de tweets "No Desastre": 4342

Out[102...]

	id	target
count	7613.000000	7613.00000
mean	5441.934848	0.42966
std	3137.116090	0.49506
min	1.000000	0.00000
25%	2734.000000	0.00000
50%	5408.000000	0.00000
75%	8146.000000	1.00000
max	10873.000000	1.00000

Lo más destacable es que las clases de tweets (Desastres y No Desastres) están relativamente balanceadas en el set de datos (~3000 y ~4000 respectivamente).

Tomé la hipótesis que el set está suficientemente balanceado, por lo que opté por no utilizar técnicas de oversampling ni undersampling que, realizadas de forma errónea, podrían dificultar el entrenamiento.

A continuación realizo una limpieza y preprocesamiento básico de texto, además de calcular las features que se mostrarán en las visualizaciones.

Cabe destacar que estas no son todas las features que usa el modelo, pero me parecieron las más interesantes e intuitivas de graficar.

In [103...]

```
def clean_text(s):
    if pd.isna(s):
        return ''
    s = str(s)
    s = s.lower()
    s = re.sub(r'http\S+', ' ', s)
    s = re.sub(r'www\S+', ' ', s)
    s = re.sub(r'[^w\s#@]', ' ', s)
    s = re.sub(r'[\s_]+', ' ', s).strip()
    return s

df['text_clean'] = df['text'].apply(clean_text)
df['keyword'] = df['keyword'].fillna('no_keyword_contained')
df['location'] = df['location'].fillna('no_location_contained')
df['text_len'] = df['text_clean'].apply(lambda s: sum(len(w) for w in s.split()))
df['word_count'] = df['text_clean'].apply(lambda s: len(s.split()))
df['mean_word_len'] = df.apply(lambda row: row['text_len'] / row['word_count'] if row['word_count'] > 0 else 0, axis=1)
df['num_hashtags'] = df['text'].apply(lambda s: 0 if pd.isna(s) else s.count('#'))
df['num_mentions'] = df['text'].apply(lambda s: 0 if pd.isna(s) else s.count('@'))
df['has_url'] = df['text'].apply(lambda s: 0 if pd.isna(s) else (1 if 'http' in s or 'www.' in s else 0))

disaster_terms = df['keyword'].dropna().unique().tolist()
def count_terms(s, terms=disaster_terms):
    s = s.lower()
    cnt = 0
    for t in terms:
        if t in s:
            cnt += 1
    return cnt
df['disaster_terms_count'] = df['text_clean'].apply(count_terms)

df['all_caps_count'] = df['text'].apply(lambda s: sum(1 for w in str(s).split() if w.isupper())).fillna(0)

df[['text_len', 'word_count', 'mean_word_len', 'num_hashtags', 'num_mentions', 'has_url', 'disaster_terms_count', 'all_caps_count']]
```

Out[103...]

	count	mean	std	min	25%	50%	75%	max
text_len	7613.0	70.222645	26.389220	5.0	50.000000	72.00	91.000000	125.0
word_count	7613.0	14.664259	6.125010	1.0	10.000000	15.00	19.000000	34.0
mean_word_len	7613.0	4.968473	1.064258	2.0	4.235294	4.85	5.526316	13.0
num_hashtags	7613.0	0.446999	1.099841	0.0	0.000000	0.00	0.000000	13.0
num_mentions	7613.0	0.362406	0.720097	0.0	0.000000	0.00	1.000000	8.0
has_url	7613.0	0.521608	0.499566	0.0	0.000000	1.00	1.000000	1.0
disaster_terms_count	7613.0	1.374097	0.866296	0.0	1.000000	1.00	2.000000	7.0
all_caps_count	7613.0	0.933929	2.010867	0.0	0.000000	0.00	1.000000	25.0

Visualizaciones

1. Distribución de longitudes del tweet según target

Para comenzar realicé un boxplot para visualizar la distribución de las longitudes de los tweets.

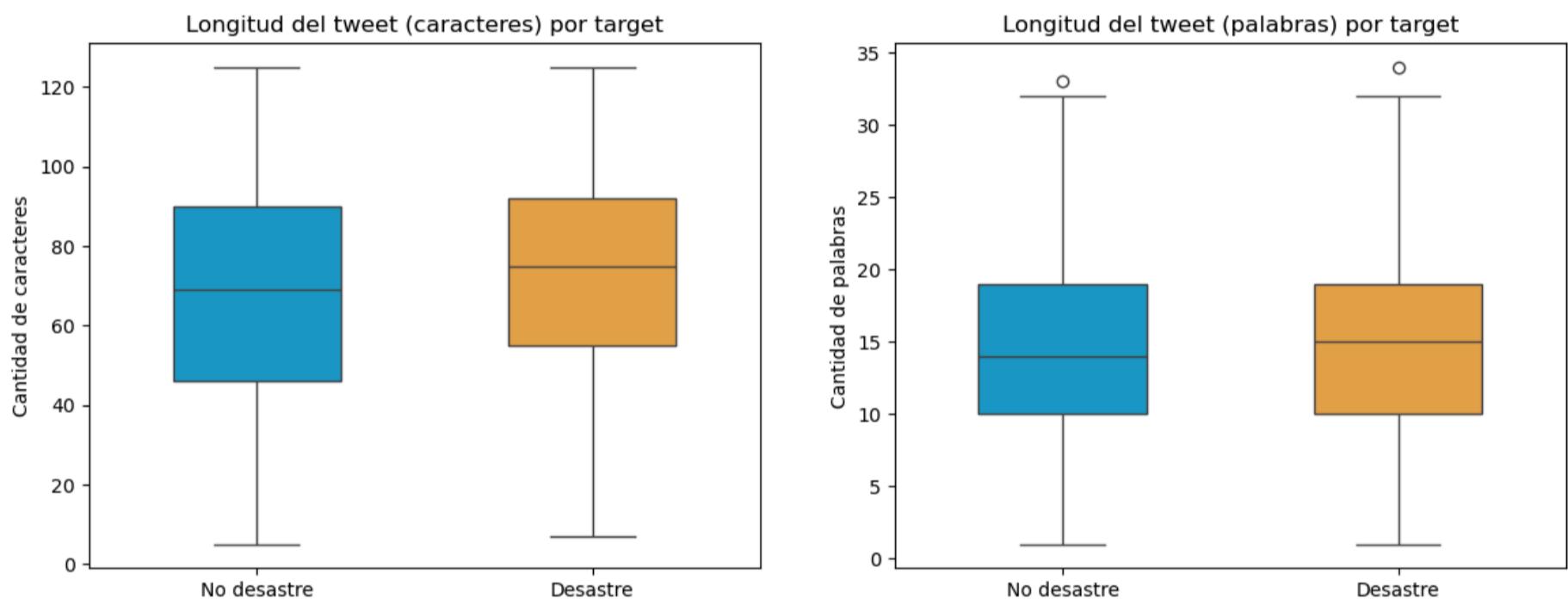
Se diferencian dos gráficos, uno que usa la longitud de caracteres del tweet y otro la longitud de palabras.

Se puede observar que en líneas generales las distribuciones son muy similares.

Sin embargo aún así se puede notar que la longitud en caracteres de los tweets de la clase Desastres es ligeramente mayor, teniendo una caja más chica (i.e. menos varianza en la longitud) y una mediana más alta.

Observar esto me dió una idea: ya que los tweets de Desastres tienen más o menos la misma longitud en palabras pero una longitud un poco mayor en caracteres, estos quizás tienden a utilizar palabras más altas. Intento visualizar esa idea en el siguiente gráfico.

```
In [104...]  
fig, axes = plt.subplots(1,2, figsize=(14,5))  
sns.boxplot(x='target', y='text_len', data=df, palette=CUSTOM_PALLETE, width=0.5, ax=axes[0])  
axes[0].set_title('Longitud del tweet (caracteres) por target')  
axes[0].set_xlabel('')  
axes[0].set_xticks([0,1], ['No desastre', 'Desastre'])  
axes[0].set_ylabel('Cantidad de caracteres')  
  
sns.boxplot(x='target', y='word_count', data=df, palette=CUSTOM_PALLETE, width=0.5, ax=axes[1])  
axes[1].set_title('Longitud del tweet (palabras) por target')  
axes[1].set_xlabel('')  
axes[1].set_xticks([0,1], ['No desastre', 'Desastre'])  
axes[1].set_ylabel('Cantidad de palabras')  
plt.show()
```

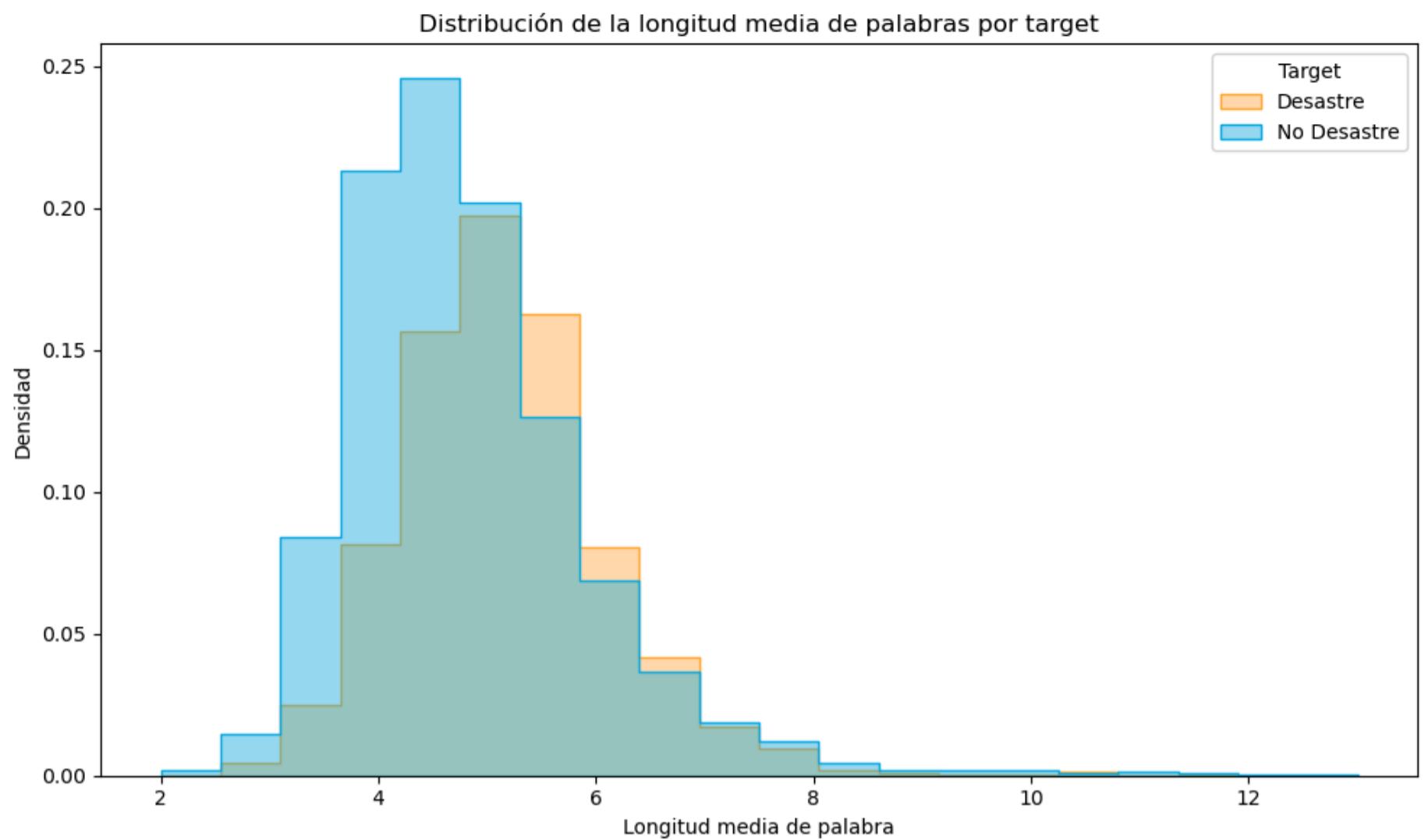


Siguiendo la observación de los gráficos anteriores, en este gráfico se visualiza la longitud media de las palabras por target.

Como podría esperarse, se observan distribuciones nuevamente muy similares, pero los tweets de Desastres tienden a tener una longitud media de palabras un poco más alta.

Si bien no es para nada una métrica concluyente, esto podría ayudar a los modelos a distinguir las dos categorías de tweets.

```
In [105...]  
plt.figure(figsize=(10,6))  
sns.histplot(  
    data=df,  
    x='mean_word_len',  
    hue='target',  
    bins=20,  
    palette=CUSTOM_PALLETE,  
    alpha=0.4,  
    element='step',  
    stat='density',  
)  
plt.title('Distribución de la longitud media de palabras por target')  
plt.xlabel('Longitud media de palabra')  
plt.ylabel('Densidad')  
plt.legend(title='Target', labels=['Desastre', 'No Desastre'])  
plt.tight_layout()  
plt.show()
```



2. Proporción de tweets según si la ubicación es mencionada en el tweet.

El propósito de esta visualización es intentar encontrar una relación entre el campo `location` y el campo `text`.

Ya sea en una noticia o un tweet de un usuario corriente, uno podría imaginarse que al hablar de un Desastre real una persona podría tender más a mencionar su ubicación. De esta idea surge la visualización mostrada a continuación.

Podemos ver en el heatmap que no necesariamente es el caso.

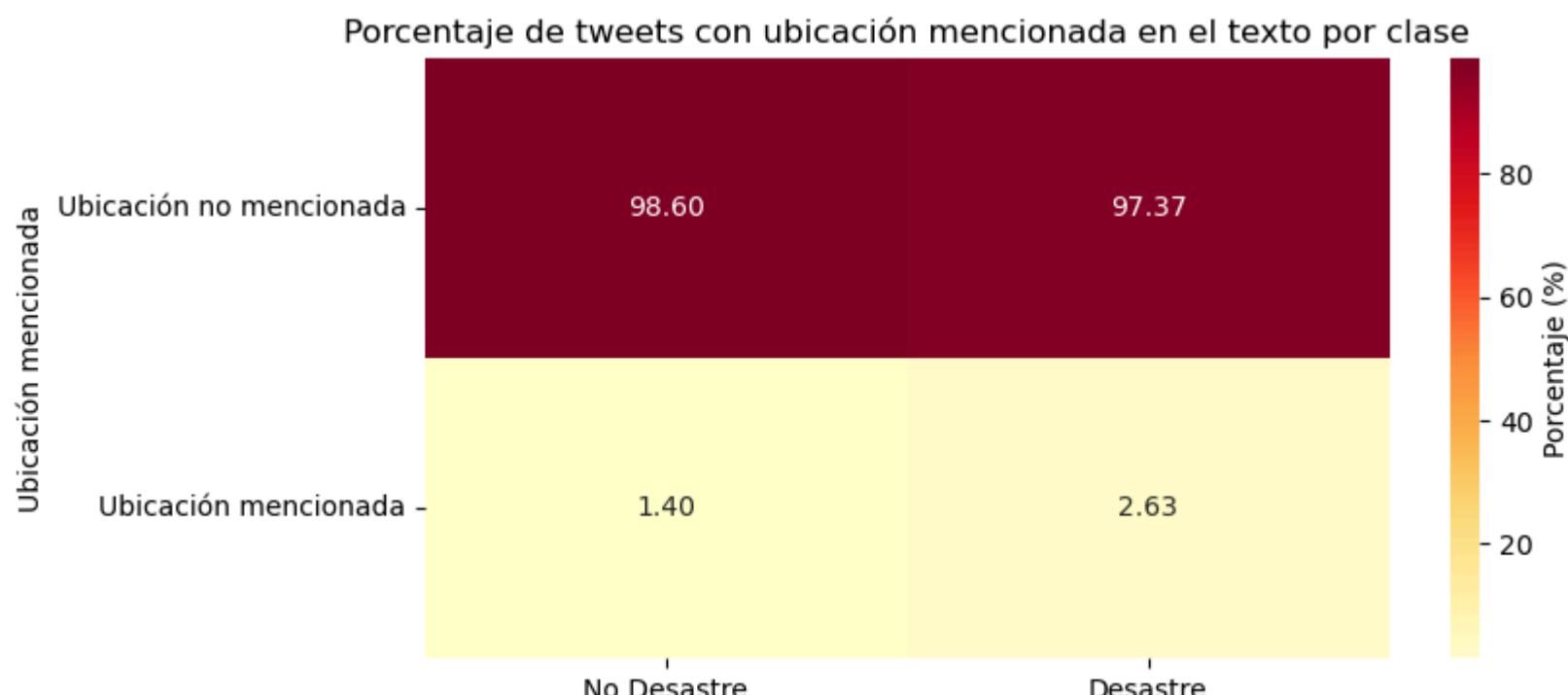
En la primera fila, podemos ver claramente que la gran mayoría de tweets de ambas clases no mencionan la ubicación en el tweet.

En la segunda, podemos ver que aproximadamente el 2.6% de tweets sobre Desastres mencionan la ubicación del usuario en contra de un 1.4% para los tweets de No Desastres. Si bien la diferencia es cercana a un 1%, dado que los porcentajes son tan bajos esa diferencia puede resultar significativa.

Aún así, este dato evidentemente sigue sin ser concluyente e incluso podría resultar confuso en ciertos contextos para el modelo.

```
In [106...]: df['location'] = df['location'].apply(clean_text)
df['location_mentioned'] = df.apply(lambda row: 1 if row['location'].lower() in row['text_clean'] else 0, axis=1)
df[['location', 'location_mentioned']].value_counts()
location_mentioned_by_target = pd.crosstab(df['location_mentioned'], df['target'], normalize='columns') * 100
location_mentioned_by_target.index = ['Ubicación no mencionada', 'Ubicación mencionada']
location_mentioned_by_target.columns = ['No Desastre', 'Desastre']

plt.figure(figsize=(8, 4))
sns.heatmap(location_mentioned_by_target, annot=True, fmt=".2f", cmap='YlOrRd', cbar_kws={'label': 'Porcentaje (%)'})
plt.title('Porcentaje de tweets con ubicación mencionada en el texto por clase')
plt.ylabel('Ubicación mencionada')
plt.yticks(rotation=0)
plt.show()
```



3. Proporción de tweets con URL / hashtags / mentions según target

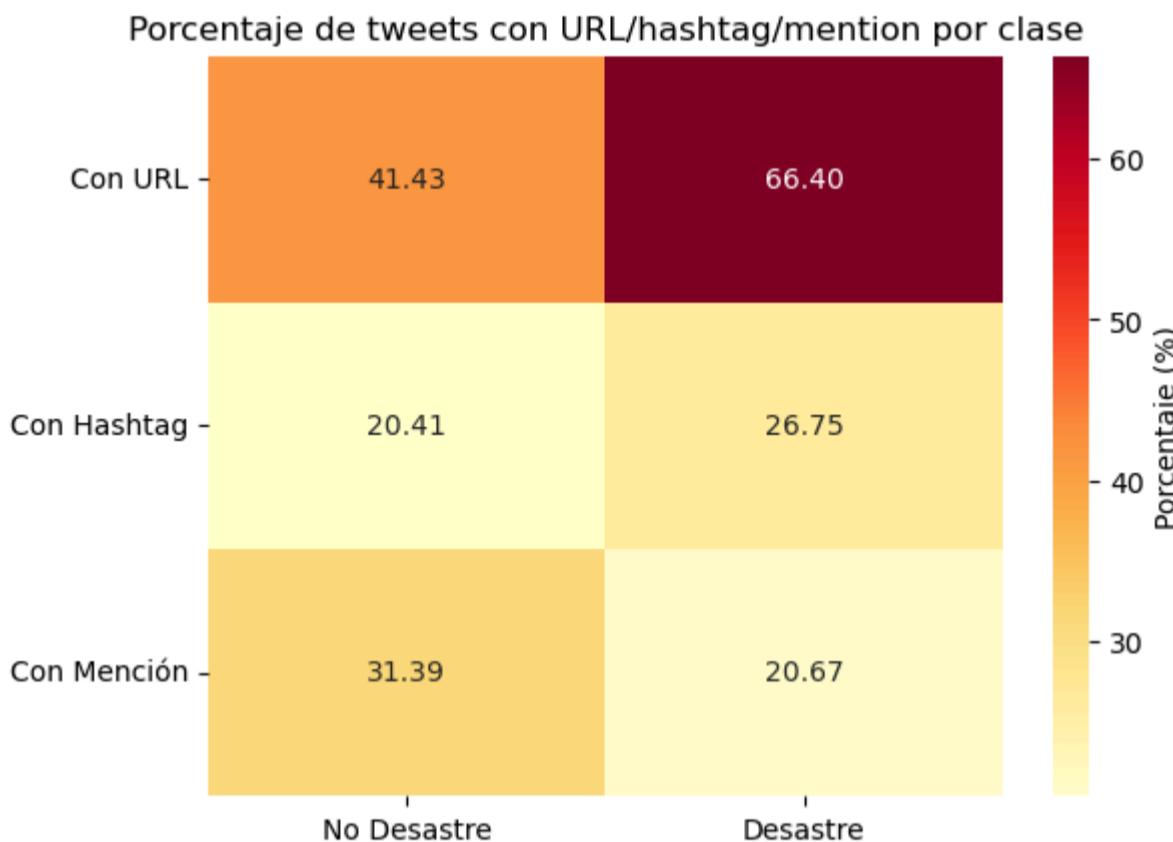
Las siguientes visualizaciones intentan ver la relación entre el target con la presencia y cantidad de menciones, hashtags y URLs contenidas en el tweet.

Primeramente, podemos observar ahora si porcentajes más variados en el heatmap.

Se puede identificar una diferencia marcada en la primera fila, la cual habla de la presencia de URL. Con una diferencia de un 25%, los tweets sobre desastres parecen tender considerablemente más a contener enlaces.

Si bien en las otras filas no hay diferencias tan marcadas (~5% y ~10%), esos datos podrían contribuir a la diferenciación de las clases.

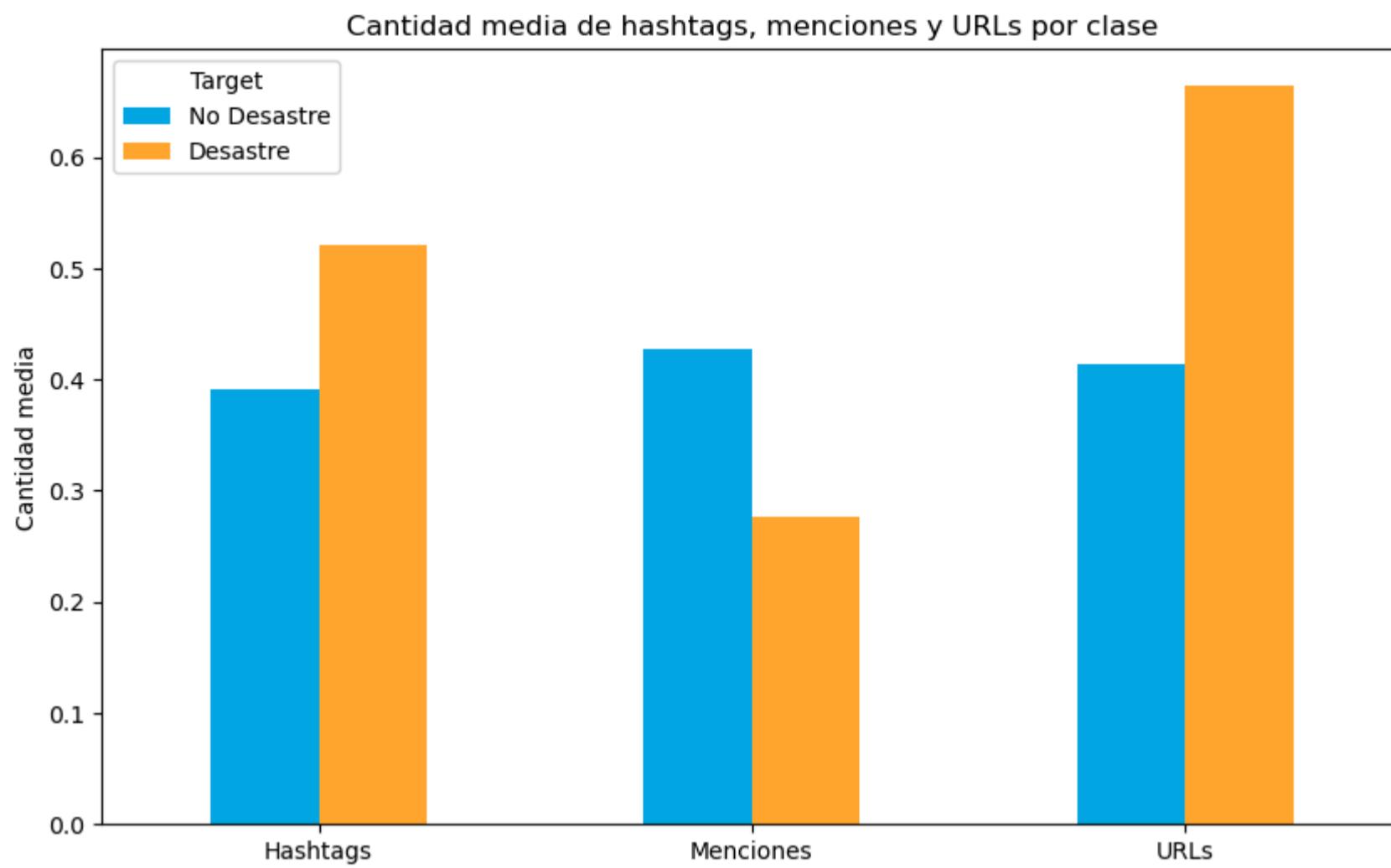
```
In [107...]  
data = pd.DataFrame({  
    'Con URL': df.groupby('target')['has_url'].mean()*100,  
    'Con Hashtag': (df['num_hashtags']>0).groupby(df['target']).mean()*100,  
    'Con Mención': (df['num_mentions']>0).groupby(df['target']).mean()*100  
})  
data = data.T  
data.columns = ['No Desastre', 'Desastre']  
sns.heatmap(data, annot=True, fmt=".2f", cmap='YlOrRd', cbar_kws={'label': 'Porcentaje (%)'})  
plt.title('Porcentaje de tweets con URL/hashtag/mention por clase')  
plt.yticks(rotation=0)  
plt.show()
```



En el siguiente gráfico de barras no se visualiza la proporción de los tweets que contienen o no contienen los elementos mencionados, sino la media de la cantidad de elementos utilizados.

Nuevamente la diferencia más notoria está en las URLs, sin embargo aún puede notarse cierta diferenciación en hashtags y menciones.

```
In [108...]  
df.groupby('target')[['num_hashtags', 'num_mentions', 'has_url']].mean()  
data = df.groupby('target')[['num_hashtags', 'num_mentions', 'has_url']].mean()  
data = data.T  
data.columns = ['No Desastre', 'Desastre']  
  
data.plot(kind='bar', figsize=(10, 6), color=CUSTOM_PALETTE)  
plt.title('Cantidad media de hashtags, menciones y URLs por clase')  
plt.ylabel('Cantidad media')  
plt.xticks(rotation=0, ticks=range(len(data.index)), labels=['Hashtags', 'Menciones', 'URLs'])  
plt.legend(title='Target')  
plt.show()
```



4. Cantidad de tweets según cantidad de términos relacionados a desastres y cantidad de palabras en mayúsculas por target

A continuación se observan dos gráficos construidos en el mismo formato pero con datos diferentes.

El primero muestra cantidad de tweets divididos según la cantidad de "términos de desastres" (obtenidos a partir del campo keyword) y por target.

Lo más relevante a destacar es que la gran mayoría de tweets sobre No Desastres del dataset contienen exactamente una palabra de desastre, mientras que los tweets sobre Desastres son más variados en la cantidad de términos que usan, siendo 1 2 y 3 las cantidades más comunes. Otro dato interesante a destacar es que no todos los tweets contienen una keyword contenida. Hice la verificación y hay varios (aunque minoría) tweets con valores no nulos en el campo `keyword` que aún así no hacen mención de ningún término en el texto. En esos casos se puede cuestionar el origen del campo `keyword`, si ese término no está en el tweet ¿de dónde se sacó esa información?

El segundo gráfico como mencioné tiene el mismo formato, las cantidades mostradas son de palabras completamente en mayúsculas del tweet. La cantidad de tweets se muestra en escala logarítmica para facilitar la visualización. También se muestran solamente hasta tweets con 15 palabras en mayúsculas, hay algunos con más de 15 (hasta 25) pero eran muy pocos (entre 0 y 10) y consideré que empobrecían la visualización.

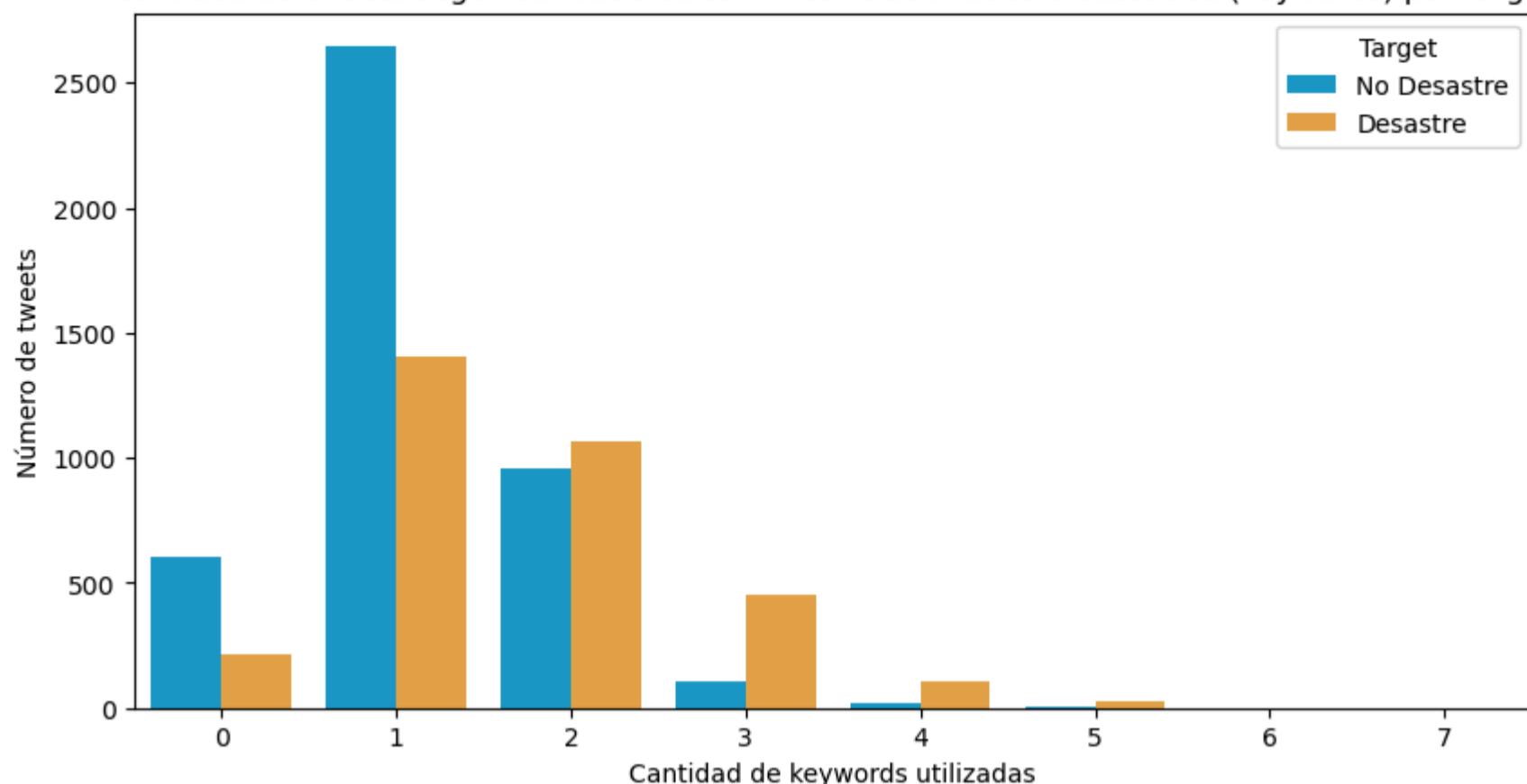
La idea detrás de esta visualización es que quizás noticias o tweets que expresan pánico en tweets sobre Desastres quizás harían más uso de palabras completamente en mayúsculas.

Sin embargo esa no fue la realidad. Se observan tendencias muy similares en cuanto a la utilización de palabras completamente en mayúsculas para ambas clases.

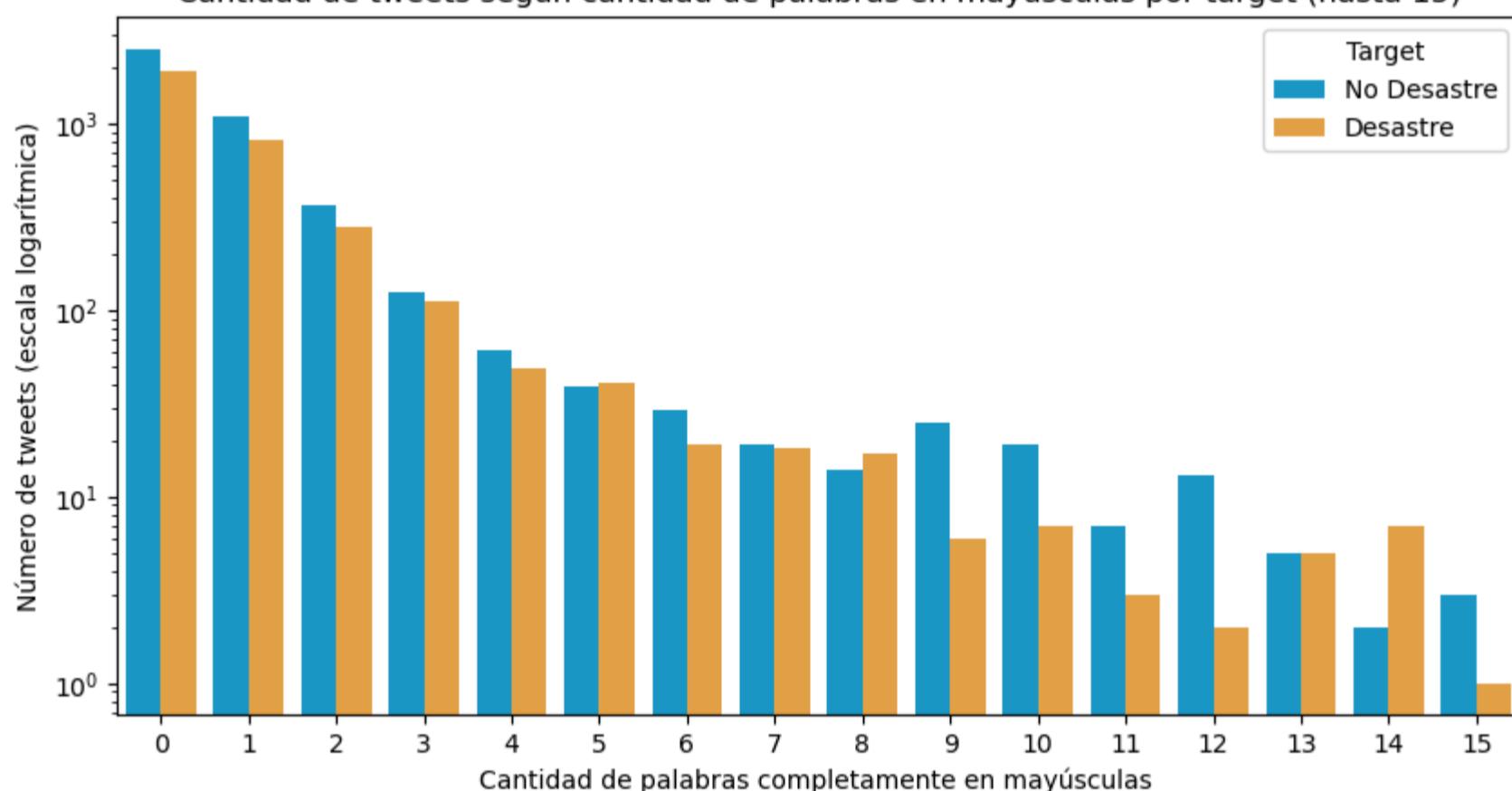
```
In [109...]
plt.figure(figsize=(10,5))
data = df.groupby(['disaster_terms_count','target']).size().reset_index(name='count')
data['target'] = data['target'].apply(lambda x: "No Desastre" if x == 0 else "Desastre")
sns.barplot(hue='target', x='disaster_terms_count', y='count', data=data, palette=CUSTOM_PALETTE)
plt.title('Cantidad de tweets según cantidad de términos relacionados a desastres (keywords) por target')
plt.xlabel('Cantidad de keywords utilizadas')
plt.ylabel('Número de tweets')
plt.legend(title="Target")
plt.show()

plt.figure(figsize=(10,5))
data = df.groupby(['all_caps_count','target']).size().reset_index(name='count')
data = data[data['all_caps_count'] <= 15]
data['target'] = data['target'].apply(lambda x: "No Desastre" if x == 0 else "Desastre")
sns.barplot(hue='target', x='all_caps_count', y='count', data=data, palette=CUSTOM_PALETTE)
plt.title('Cantidad de tweets según cantidad de palabras en mayúsculas por target (hasta 15)')
plt.xlabel('Cantidad de palabras completamente en mayúsculas ')
plt.ylabel('Número de tweets (escala logarítmica)')
plt.yscale('log')
plt.legend(title="Target")
plt.show()
```

Cantidad de tweets según cantidad de términos relacionados a desastres (keywords) por target



Cantidad de tweets según cantidad de palabras en mayúsculas por target (hasta 15)



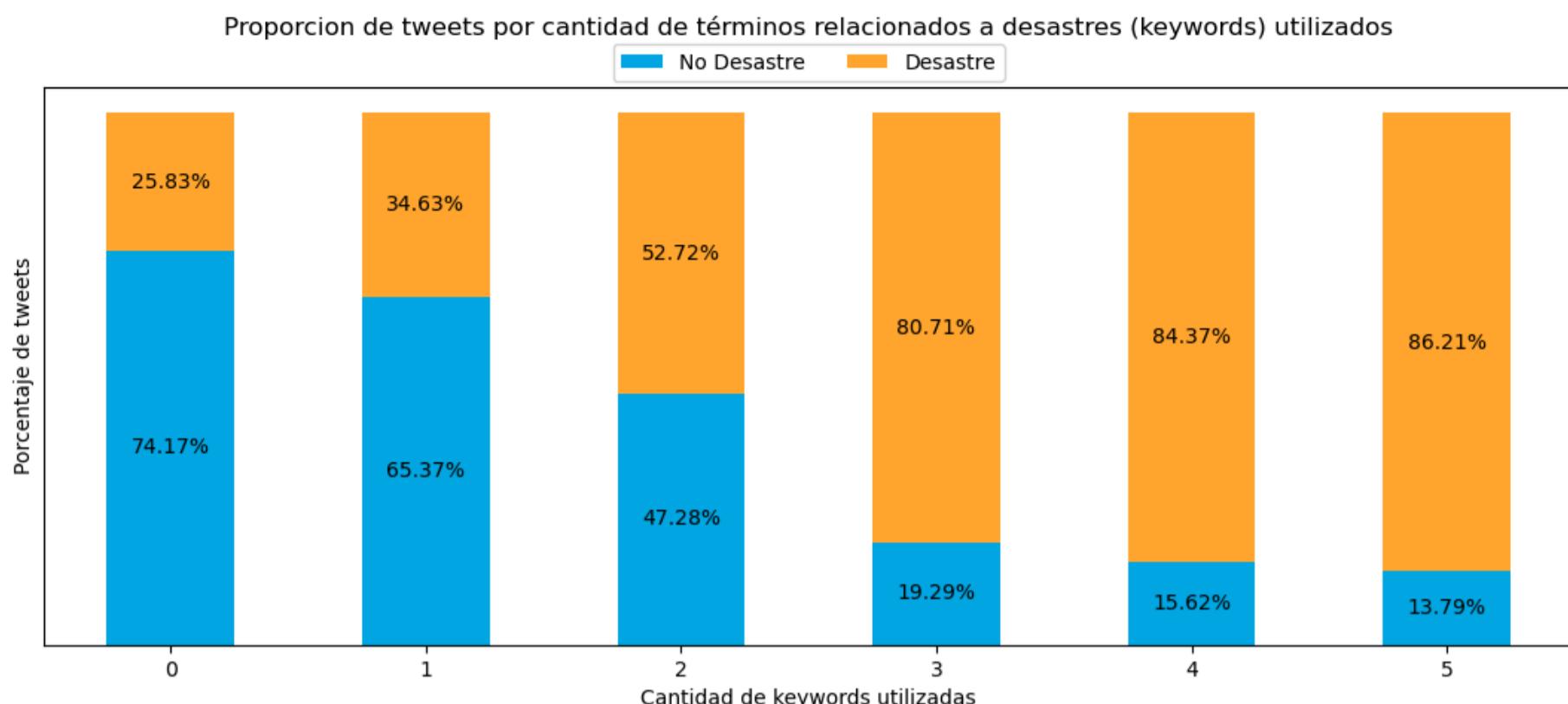
Por último muestro un grafico de barras apiladas para tener otra vista del primer gráfico de la celda anterior. No se incluyen las cantidades mayores a 5 porque representan muy pocos tweets.

Como se mencionó anteriormente, puede observarse que es mayor la proporción de tweets sobre Desastres cuando la cantidad de términos mencionados es mayor.

```
In [110...]: data = df.groupby(['disaster_terms_count','target'])["id"].nunique().unstack()
data = data[data.index <= 5]
data.columns = ['No Desastre', 'Desastre']
data.fillna(0, inplace=True)
fig, ax = plt.subplots(figsize=(12,6))
data.div(data.sum(axis=1), axis=0)\n    .plot(kind='bar', stacked=True, figsize=(12,6), color=CUSTOM_PALLETE, ax=ax)

for c in ax.containers:
    labels = [str(round(v.get_height()*100, 2)) + "%" if v.get_height() > 0 else '' for v in c]
    ax.bar_label(c,
                 label_type='center',
                 labels = labels,
                 size = 10)

plt.title('Proporcion de tweets por cantidad de términos relacionados a desastres (keywords) utilizados', pad=25)
plt.tight_layout(pad=5)
plt.legend(ncols=2, loc='upper center', bbox_to_anchor=(0.5, 1.095))
plt.xlabel('Cantidad de keywords utilizadas')
plt.ylabel('Porcentaje de tweets')
plt.yticks(ticks=[])
plt.xticks(rotation=0)
plt.show()
```



5. Distribución de sentimiento del tweet por target

Para esta visualización utilizo la librería de código abierto `vaderSentiment`, la cual proporciona una herramienta de análisis de sentimientos en textos, especializada en redes sociales.

```
In [111...]: from vaderSentiment.vaderSentiment import SentimentIntensityAnalyzer
analyzer = SentimentIntensityAnalyzer()

def vader_scores(text):
    if not isinstance(text, str) or text.strip() == "":
        return {'neg': 0.0, 'neu': 0.0, 'pos': 0.0, 'compound': 0.0}
    return analyzer.polarity_scores(text)

scores = [vader_scores(t) for t in df['text'].astype(str).tolist()]
scores_df = pd.DataFrame(scores)

df = pd.concat([df.reset_index(drop=True), scores_df.reset_index(drop=True)], axis=1)
```

A continuación podemos visualizar las distribuciones de las puntuaciones que proporciona `vaderSentiment`, según target.

Los primeros tres gráficos muestran los puntajes particulares (valores entre 0 y 1) para sentimiento negativo, neutro y positivo. El último gráfico muestra un puntaje "compuesto" que va desde -1 hasta 1, donde -1 representaría sentimiento completamente negativo, 0 sentimiento completamente neutro y 1 sentimiento completamente positivo.

Podemos ver distribuciones que aunque tienen diferencias son bastante similares. Para ambos targets la mayoría de tweets tienen puntajes que indican sentimientos neutros.

En el gráfico de puntaje compuesto se observa que los tweets sobre Desastres tienen una distribución más centrada en los sentimientos negativos, lo cual puede resultar para la distinción entre las clases para los modelos de machine learning.

Se puede considerar que estas 4 columnas contienen información bastante redundante.

Para los modelos finales resultó mejor utilizar solo la columna de puntaje compuesto. Mi hipótesis es que esto es así porque resume muy bien las otras tres columnas.

```
In [112...]: data=df[['target', 'neg', 'neu', 'pos', 'compound']]
data['target'] = data['target'].map(lambda x: "No Desastre" if x == 0 else "Desastre")

cols = ['neg', 'neu', 'pos', 'compound']
colnames = {
    'neg': 'Negativo',
    'neu': 'Neutro',
    'pos': 'Positivo',
    'compound': 'Compuesto'
}

fig, axes = plt.subplots(nrows=1, ncols=len(cols), figsize=(18, 5))

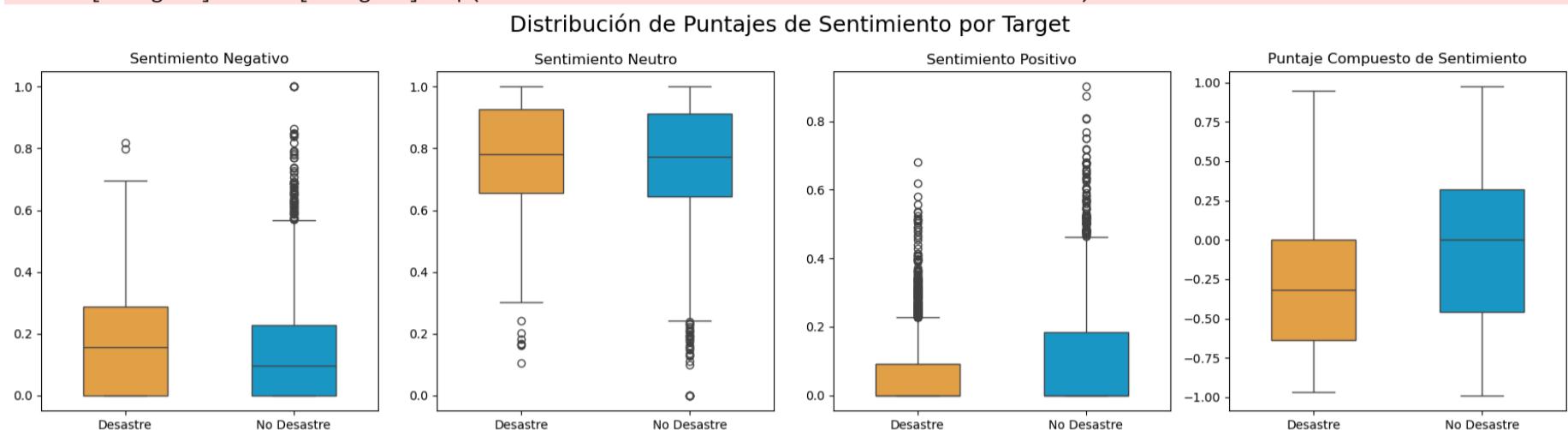
for i, col in enumerate(cols):
    sns.boxplot(data=data, x='target', y=col, ax=axes[i], palette=CUSTOM_PALLETE[::-1], width=0.5)
    axes[i].set_ylabel('')
    axes[i].set_xlabel('')
    if col != 'compound':
        axes[i].set_title(f"Sentimiento {colnames[col]}")
    else:
        axes[i].set_title("Puntaje Compuesto de Sentimiento")
fig.suptitle("Distribución de Puntajes de Sentimiento por Target", fontsize=18)
```

```
plt.tight_layout()  
plt.show()
```

C:\Users\Patricio\AppData\Local\Temp\ipykernel_8112\3128206617.py:2: SettingWithCopyWarning:
A value is trying to be set on a copy of a slice from a DataFrame.
Try using .loc[row_indexer,col_indexer] = value instead

See the caveats in the documentation: https://pandas.pydata.org/pandas-docs/stable/user_guide/indexing.html#returning-a-view-versus-a-copy

```
data['target'] = data['target'].map(lambda x: "No Desastre" if x == 0 else "Desastre")
```



6. Nubes de palabras para cada target

En esta última sección se realizan visualizaciones cualitativas, donde se intentó ilustrar la incidencia de las palabras utilizadas para distinguir las clases de los tweets.

Primero se muestran dos nubes de palabras, donde el tamaño de la palabra es proporcional a la cantidad de veces que aparece entre todos los tweets de esa clase. Antes de calcular las nubes de palabras utilice la librería `nltk` para remover las stopwords de la columna de texto formateado.

Analizando las dos nubes de palabras se pueden identificar varias palabras en común que aparecen bastante en ambas clases, por ejemplo 'amp' o 'new'.

También pueden identificarse algunas palabras generales como 'one' o 'day' que aparecen con tamaños considerablemente diferentes en ambas nubes.

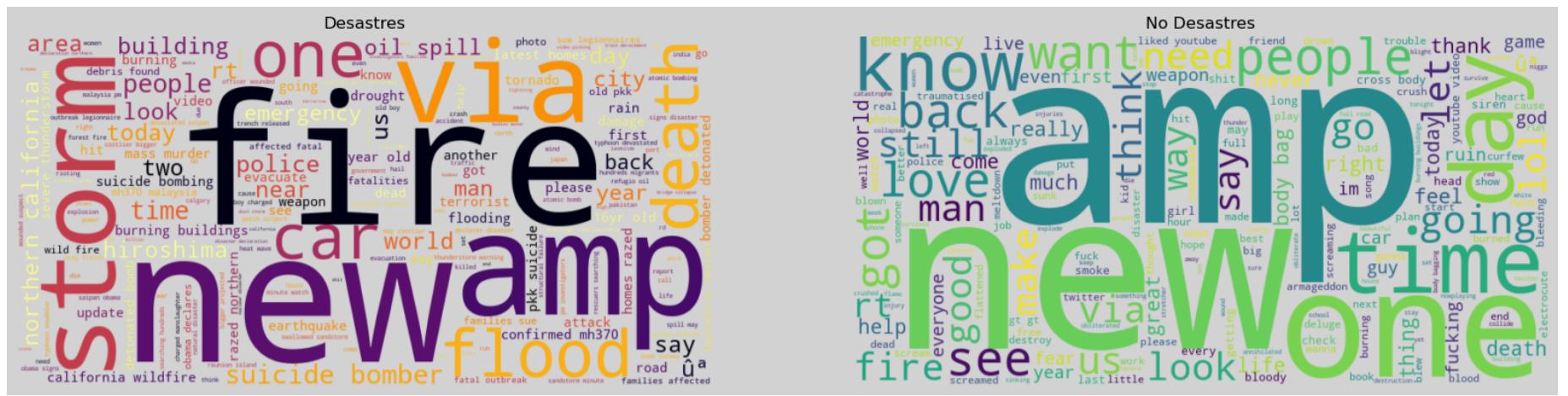
Por último también es importante destacar que hay muchas palabras que aparecen con gran tamaño en una clase y con uno muy pequeño o directamente sin aparecer en la otra clase. Por ejemplo en la nube de Desastres se puede encontrar con tamaño considerable las palabras 'fire' 'death' 'flood' y 'hiroshima', sin embargo estas aparecen bastante pequeñas o no aparecen en la nube de No Desastres.

Los patrones recién descritos son muy simples pero nos permiten ver diferencias entre ambas clases.

Estos patrones y muchos más (igual y más complejos) pueden ser aprendidos de manera automática por los modelos entrenados en este trabajo práctico. Es por esto que es importante encontrar una manera inteligente de representar el contenido del texto.

```
In [113]:  
def generate_word_cloud(colormap, words):  
    return WordCloud(  
        width=800, height=400, background_color='lightgray', random_state=SEED,  
        min_word_length=2, colormap=colormap,  
    ).generate(words)  
  
import nltk  
nltk.download('stopwords')  
from nltk.corpus import stopwords  
stop_words = set(stopwords.words('english'))  
df['text_clean_nostop'] = df['text_clean'].apply(  
    lambda s: ' '.join([word for word in s.split() if word not in stop_words])  
)  
  
disaster_words = ' '.join(df[df['target']==1]['text_clean_nostop'].tolist())  
not_disaster_words = ' '.join(df[df['target']==0]['text_clean_nostop'].tolist())  
disaster_wc = generate_word_cloud("inferno", disaster_words)  
not_disaster_wc = generate_word_cloud("viridis", not_disaster_words)  
fig, axes = plt.subplots(1,2, figsize=(20,8))  
axes[0].imshow(disaster_wc, interpolation='bilinear')  
axes[0].axis('off')  
axes[0].set_title('Desastres')  
axes[1].imshow(not_disaster_wc, interpolation='bilinear')  
axes[1].axis('off')  
axes[1].set_title('No Desastres')  
fig.patch.set_facecolor('lightgray')  
plt.show()
```

```
[nltk_data] Downloading package stopwords to  
[nltk_data]     C:\Users\Patricio\AppData\Roaming\nltk_data...  
[nltk_data]     Package stopwords is already up-to-date!
```



Análisis de palabras importantes usando TF-IDF + RandomForest

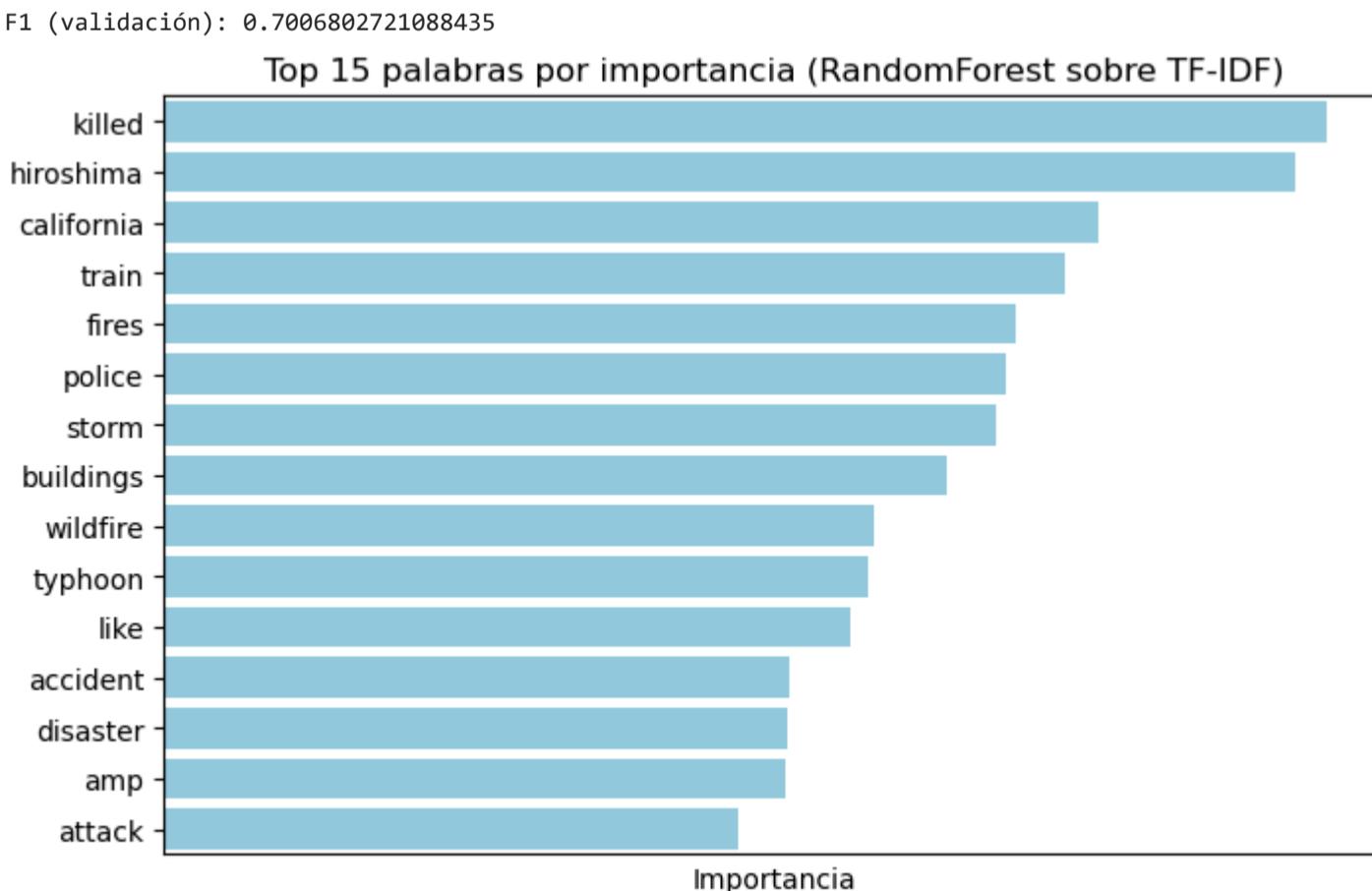
Para hacer otro pequeño análisis de la importancia de la columna text, se me ocurrió entrenar un RandomForestClassifier utilizando únicamente esa feature. La idea es hacer un análisis rápido, por lo que no incluí búsqueda de hiperparámetros ni un procesamiento muy complejo

Vectorizando los textos utilizando simplemente TF-IDF se obtuvo un puntaje F1 considerable, ~0.7. Con un procesamiento muy sencillo del texto, el modelo es capaz de encontrar patrones que permiten diferenciar relativamente bien las clases.

Luego, se muestra un gráfico de la importancia que el modelo le dió a cada palabra. Podemos notar importancias bastante diferentes, las palabras más importantes son 'killed' e 'hiroshima'. También es interesante notar que no todas las palabras de mayor importancia son necesariamente relacionadas a desastres (e.g. 'california' y 'like').

En la parte 4 se incuyen gráficos visualizando los tweets como vectores (reduciendo sus dimensiones), donde se pueden identificar agrupamientos que se corresponden a si los tweets son desastres o no.

```
In [114...]  
y = df['target'].values  
X_train, X_val, y_train, y_val = train_test_split(df['text_clean'], y, test_size=0.20, random_state=SEED)  
vectorizer = TfidfVectorizer(max_features=400, stop_words='english')  
vectorizer.fit(X_train)  
X_tfidf = vectorizer.transform(X_train)  
  
rf = RandomForestClassifier(n_estimators=20, random_state=SEED, n_jobs=-1)  
rf.fit(X_tfidf, y_train)  
val_tfidf = vectorizer.transform(X_val)  
y_pred = rf.predict(val_tfidf)  
print('F1 (validación):', f1_score(y_val, y_pred))  
  
importances = rf.feature_importances_  
indices = np.argsort(importances)[::-1][:15]  
feat_names = np.array(vectorizer.get_feature_names_out())  
top_words = feat_names[indices]  
top_importances = importances[indices]  
  
plt.figure(figsize=(8,5))  
sns.barplot(x=top_importances, y=top_words, color='skyblue')  
plt.title('Top 15 palabras por importancia (RandomForest sobre TF-IDF)')  
plt.xlabel('Importancia')  
plt.xticks(ticks=[])  
plt.show()
```



Parte 2: Baseline (Regresión Logística).

En esta parte se implementa un modelo de regresión logística como baseline para el problema de clasificación de tweets relacionados a desastres naturales. Se utilizan características numéricas, categóricas y de texto procesadas previamente.

En esta parte me centro en conseguir un modelo rápido y sencillo de interpretar, dejando modelos más complejos para la parte 3.

El paso a paso del proceso de entrenamiento está más detallado en la parte 3. Como este fue el primer modelo que entrené tuve bastante prueba y error, por lo que terminé arreglando el proceso general luego de terminar la parte 3 y obtuve un pipeline similar al de los modelos más complejos.

Respuestas a las preguntas del enunciado:

- ¿Cuál es el mejor score de validación obtenido? (¿Cómo conviene obtener el dataset para validar?)\

El mejor score de validación obtenido fue de **0.73800** en Cross Validation y de **0.75867** en el set de validación (20% del train set original).

Considero el segundo score como más representativo del desempeño real del modelo, ya que en CV se entrena y evalúa en múltiples folds, lo que puede llevar a un sobreajuste a los datos de entrenamiento.

- Al predecir con este modelo para la competencia, ¿Cuál es el score obtenido? (guardar el csv con predicciones para entregarlo después)

El score obtenido en la competencia fue de **0.76769**.

- ¿Qué features son los más importantes para predecir con el mejor modelo? Graficar

Esta pregunta está respondida en la sección [Importancia de features](#) más abajo.

```
In [1]: import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
import re
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.model_selection import train_test_split, StratifiedKFold, GridSearchCV
from sklearn.preprocessing import TargetEncoder
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import f1_score, classification_report
from vaderSentiment.vaderSentiment import SentimentIntensityAnalyzer

SEED = 42
np.random.seed(SEED)

In [2]: train_path = 'data/train.csv'
df = pd.read_csv(train_path)
print('train shape:', df.shape)

train shape: (7613, 5)
```

Preparación datos para entrenamiento y validación

A continuación defino las características mostradas en la parte 1.

```
In [3]: def clean_text(s):
    if pd.isna(s):
        return ''
    s = str(s)
    s = s.lower()
    s = re.sub(r'http\S+', ' ', s)
    s = re.sub(r'www\S+', ' ', s)
    s = re.sub(r'[\^w\s#@]', ' ', s)
    s = re.sub(r'[\s_]+', ' ', s).strip()
    return s

df['text_clean'] = df['text'].apply(clean_text)
df['keyword'] = df['keyword'].fillna('no_keyword_contained')
df['location'] = df['location'].apply(clean_text)
df['location'] = df['location'].fillna('no_location_contained')

df['word_count'] = df['text_clean'].apply(lambda s: len(s.split()))
df['text_len'] = df['text_clean'].apply(lambda s: sum(len(w) for w in s.split()))
df['mean_word_len'] = df.apply(lambda row: row['text_len'] / row['word_count'] if row['word_count'] > 0 else 0, axis=1)
df['num_hashtags'] = df['text'].apply(lambda s: 0 if pd.isna(s) else s.count('#'))
df['num_mentions'] = df['text'].apply(lambda s: 0 if pd.isna(s) else s.count('@'))
df['has_url'] = df['text'].apply(lambda s: 0 if pd.isna(s) else (1 if 'http' in s or 'www.' in s else 0))
df['has_hashtag'] = df['num_hashtags'].apply(lambda x: 1 if x > 0 else 0)
df['has_mention'] = df['num_mentions'].apply(lambda x: 1 if x > 0 else 0)
df['location_mentioned'] = df.apply(lambda row: 1 if row['location'].lower() in row['text_clean'] else 0, axis=1)

disaster_terms = df['keyword'].dropna().unique().tolist()
def count_terms(s, terms=disaster_terms):
    s = s.lower()
    cnt = 0
```

```

    for t in terms:
        if t in s:
            cnt += 1
    return cnt
df['disaster_terms_count'] = df['text_clean'].apply(count_terms)
df['all_caps_count'] = df['text'].apply(lambda s: sum(1 for w in str(s).split() if w.isupper())).fillna(0)

analyzer = SentimentIntensityAnalyzer()

def vader_scores(text):
    if not isinstance(text, str) or text.strip() == "":
        return {'neg': 0.0, 'neu': 0.0, 'pos': 0.0, 'compound': 0.0}
    return analyzer.polarity_scores(text)

scores = [vader_scores(t) for t in df['text'].astype(str).tolist()]
scores_df = pd.DataFrame(scores)

df = pd.concat([df.reset_index(drop=True), scores_df.reset_index(drop=True)], axis=1)

```

```

In [4]: num_features = [
    'text_len', 'word_count', 'mean_word_len',
    'num_hashtags', 'num_mentions', 'disaster_terms_count',
    'all_caps_count',
    'neg', 'neu', 'pos', 'compound'
]
cat_features = ['location', 'keyword']
bool_features = ['has_url', 'has_hashtag', 'has_mention', 'location_mentioned']
emb_features = ['text_clean']
X = df[num_features + cat_features + bool_features + emb_features]
y = df['target'].astype(int).values

X_train, X_val, y_train, y_val = train_test_split(
    X, y, stratify=y, test_size=0.2, random_state=SEED
)
print('Train shape:', X_train.shape, 'Val shape:', X_val.shape)

```

Train shape: (6090, 18) Val shape: (1523, 18)

Encoding + búsqueda de hiperparámetros con GridSearchCV

```

In [5]: from sklearn.compose import ColumnTransformer
from sklearn.pipeline import Pipeline

preprocessor = ColumnTransformer(
    transformers=[
        # ('nombre', transformador, columnas_a_aplicar)

        ('emb', TfidfVectorizer(max_features=100, stop_words='english'), 'text_clean'),
        ('kyw', TargetEncoder(random_state=SEED), ['keyword']),
        ('loc', TargetEncoder(random_state=SEED), ['location']),
        ('other', 'passthrough', num_features + bool_features),
    ],
    remainder='drop'
)

base_line_pipeline = Pipeline(
    steps=[
        ('preprocessor', preprocessor),
        ('baseline', LogisticRegression(max_iter=1000, class_weight='balanced', random_state=SEED))
    ]
)

```

```

In [ ]: hparams = {
    'baseline__C': [0.01, 0.05, 0.1, 0.25, 0.5, 0.75, 1.0, 1.25, 1.5],
    'baseline__penalty': ['l1', 'l2', None],
}

cv = StratifiedKFold(n_splits=5, shuffle=True, random_state=SEED)
grid = GridSearchCV(base_line_pipeline, hparams, scoring='f1', cv=cv, n_jobs=1, verbose=0)

grid.fit(X_train, y_train)

```

```

In [7]: best_bl = grid.best_estimator_
vectorizer = best_bl.named_steps['preprocessor'].named_transformers_['emb']
model = best_bl.named_steps['baseline']
print('Mejores hiper parametros:', grid.best_params_)
print('Mejor F1 en CV:', grid.best_score_)

y_val_pred = best_bl.predict(X_val)
print('F1 en validación:', f1_score(y_val, y_val_pred))

```

Mejores hiper parametros: {'baseline__C': 0.25, 'baseline__penalty': 'l2'}
Mejor F1 en CV: 0.7380042354347581
F1 en validación: 0.7586726998491704

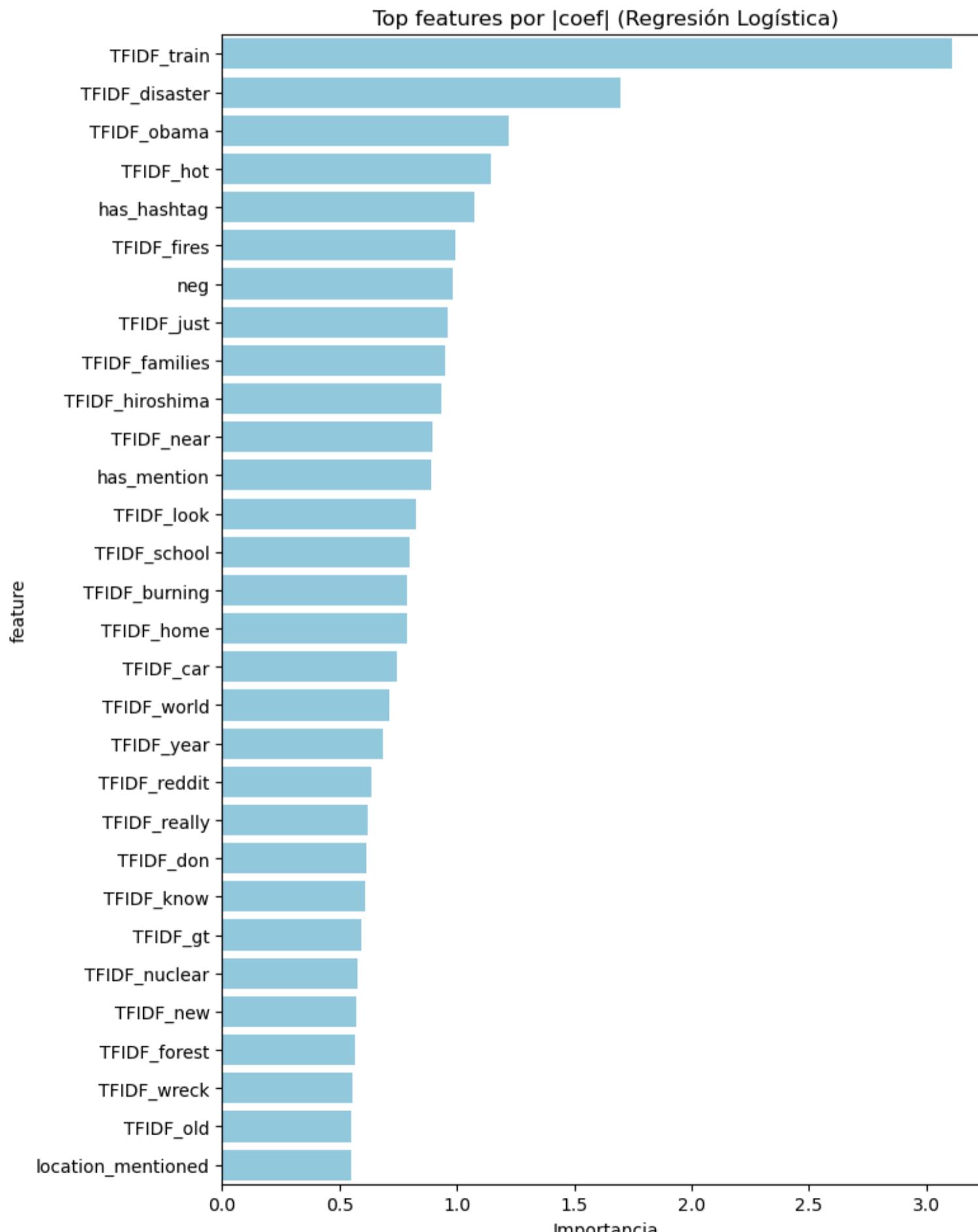
Importancia de features

Tomo los coeficientes del modelo entrenado (`coef_`) para analizar la importancia de las features.

Puede verse que la mayoría de las features más importantes son las relacionadas al texto (TF-IDF), aunque también pueden identificarse algunas features booleanas y numéricas relevantes.

```
In [8]: tfidf_col_names = [f"TFIDF_{c}" for c in vectorizer.get_feature_names_out()]
feature_names_tab = num_features + ['location_enc', 'keyword_enc'] + bool_features + tfidf_col_names
# Coefs del mejor clf (coef_ corresponde a todas las features incl. embeddings)
coefs = model.coef_[0]
# Tomamos coeficientes relativos a las features tabulares (al inicio)
n_tab = len(feature_names_tab)
tab_coefs = coefs[:n_tab]
imp_df = pd.DataFrame({'feature': feature_names_tab, 'coef_abs': np.abs(tab_coefs), 'coef': tab_coefs})
imp_df = imp_df.sort_values('coef_abs', ascending=False).head(30)

plt.figure(figsize=(8,10))
sns.barplot(x='coef_abs', y='feature', data=imp_df, color='skyblue')
plt.title('Top features por |coef| (Regresión Logística)')
plt.xlabel('Importancia')
plt.tight_layout()
plt.show()
```



Predicción sobre el set de Test

```
In [9]: test_path = 'data/test.csv'
df_test = pd.read_csv(test_path)

In [10]: df_test['text_clean'] = df_test['text'].apply(clean_text)
df_test['keyword'] = df_test['keyword'].fillna('no_keyword_contained')
df_test['location'] = df_test['location'].apply(clean_text)
df_test['location'] = df_test['location'].fillna('no_location_contained')

df_test['word_count'] = df_test['text_clean'].apply(lambda s: len(s.split()))
df_test['text_len'] = df_test['text_clean'].apply(lambda s: sum(len(w) for w in s.split()))
df_test['mean_word_len'] = df_test.apply(lambda row: row['text_len'] / row['word_count'] if row['word_count'] > 0 else 0, axis=1)
df_test['num_hashtags'] = df_test['text'].apply(lambda s: 0 if pd.isna(s) else s.count('#'))
df_test['num_mentions'] = df_test['text'].apply(lambda s: 0 if pd.isna(s) else s.count('@'))
df_test['has_url'] = df_test['text'].apply(lambda s: 0 if pd.isna(s) else (1 if 'http' in s or 'www.' in s else 0))
df_test['has_hashtag'] = df_test['num_hashtags'].apply(lambda x: 1 if x > 0 else 0)
df_test['has_mention'] = df_test['num_mentions'].apply(lambda x: 1 if x > 0 else 0)
df_test['location_mentioned'] = df_test.apply(lambda row: 1 if row['location'].lower() in row['text_clean'] else 0, axis=1)

disaster_terms = df_test['keyword'].dropna().unique().tolist()

df_test['disaster_terms_count'] = df_test['text_clean'].apply(count_terms)
df_test['all_caps_count'] = df_test['text'].apply(lambda s: sum(1 for w in str(s).split() if w.isupper())).fillna(0)

scores = [vader_scores(t) for t in df_test['text'].astype(str).tolist()]
scores_df_test = pd.DataFrame(scores)

df_test = pd.concat([df_test.reset_index(drop=True), scores_df_test.reset_index(drop=True)], axis=1)
```

```
In [11]: test_pred = best_bl.predict(df_test)
```

```
In [12]: submission = pd.DataFrame({'id': df_test['id'], 'target': test_pred})
submission.to_csv('baseline_submission.csv', index=False)
```

Parte 3: Machine Learning

Para esta parte del trabajo práctico elegí entrenar principalmente dos modelos: KNN y una red neuronal.

Además, de forma adicional entrené un modelo de LightGBM para probar hacer un ensamblado con la red neuronal.

En las secciones respectivas a cada modelo doy detalles de cómo fue el proceso de entrenamiento, selección de features y demás.

Pero antes de pasar con los modelos, incluyo una sección de Feature Engineering común a ambos modelos.

Feature engineering común a ambos modelos

En esta sección creo features que son utilizadas por todos los modelos entrenados.

Primero importo las funciones y objetos que necesito para esta sección y además cargo en memoria el set de entrenamiento.

Las demás importaciones las realizaré medida que los vaya utilizando.

```
In [1]:  
import os  
import numpy as np  
import pandas as pd  
import tensorflow as tf  
import re  
import nltk  
nltk.download('stopwords')  
from nltk.corpus import stopwords  
  
SEED = 42  
os.environ['PYTHONHASHSEED'] = str(SEED)  
np.random.seed(SEED)  
tf.random.set_seed(SEED)  
tf.config.experimental.enable_op_determinism()  
stop_words = set(stopwords.words('english'))  
  
train_path = 'data/train.csv'  
df = pd.read_csv(train_path)
```

```
[nltk_data] Downloading package stopwords to  
[nltk_data]     C:\Users\Patricio\AppData\Roaming\nltk_data...  
[nltk_data]     Package stopwords is already up-to-date!
```

A continuación creo las features básicas exploradas en la parte 1.

```
In [2]: # Limpieza y procesamiento básico de texto  
def clean_text(s):  
    if pd.isna(s):  
        return ''  
    s = str(s)  
    s = s.lower()  
    s = re.sub(r'http\S+', ' ', s)  
    s = re.sub(r'www\S+', ' ', s)  
    s = re.sub(r'^\w\s#@', ' ', s)  
    s = re.sub(r'[\s_]+', ' ', s).strip()  
    filtered_words = [word for word in s.split() if word.lower() not in stop_words and not word.startswith('@')]  
    filtered_words = [(w[1:] if w.startswith('#') and len(w) > 1 else w) for w in filtered_words]  
    filtered_words = [w for w in filtered_words if w]  
    cleaned_text = " ".join(filtered_words)  
    return cleaned_text  
  
df['text_clean'] = df['text'].apply(clean_text)  
df['keyword'] = df['keyword'].fillna('no_keyword_contained')  
df['location'] = df['location'].apply(clean_text)  
df['location'] = df['location'].fillna('no_location_contained')  
  
# Creación de features numéricas adicionales  
df['word_count'] = df['text_clean'].apply(lambda s: len(s.split()))  
df['text_len'] = df['text_clean'].apply(lambda s: sum(len(w) for w in s.split()))  
df['mean_word_len'] = df.apply(lambda row: row['text_len'] / row['word_count'] if row['word_count'] > 0 else 0, axis=1)  
df['num_hashtags'] = df['text'].apply(lambda s: 0 if pd.isna(s) else s.count('#'))  
df['num_mentions'] = df['text'].apply(lambda s: 0 if pd.isna(s) else s.count('@'))  
df['num_exclamations'] = df['text'].apply(lambda s: 0 if pd.isna(s) else s.count('!'))  
df['num_questions'] = df['text'].apply(lambda s: 0 if pd.isna(s) else s.count('?'))  
df['num_smilies'] = df['text'].apply(lambda s: 0 if pd.isna(s) else len(re.findall(r'[:;=8][\-\o\*\']?[\\]\[(\\dDpP/:\\)\{@\\|\\\]', s)))  
df['num_repeated_chars'] = df['text'].apply(lambda s: 0 if pd.isna(s) else len(re.findall(r'(.)\1{2,}', s)))  
df['num_capital_words'] = df['text'].apply(lambda s: sum(1 for w in str(s).split() if w.isupper())).fillna(0)  
df['numeric_chars_count'] = df['text'].apply(lambda s: 0 if pd.isna(s) else len(re.findall(r'\d', s)))  
df['stopword_ratio'] = df.apply(lambda row: sum(1 for w in str(row['text']).split() if w.lower() in stop_words) / row['word_count'])  
df['unique_word_ratio'] = df.apply(lambda row: len(set([w.strip() for w in row['text_clean'].split()])) / row['word_count'] if row['word_count'] > 0 else 0)  
  
# Creación de features booleanas adicionales  
df['has_url'] = df['text'].apply(lambda s: 0 if pd.isna(s) else (1 if 'http' in s or 'www.' in s else 0))  
df['has_hashtag'] = df['num_hashtags'].apply(lambda x: 1 if x > 0 else 0)  
df['has_mention'] = df['num_mentions'].apply(lambda x: 1 if x > 0 else 0)  
df['location_mentioned'] = df.apply(lambda row: 1 if row['location'].lower() in row['text_clean'] else 0, axis=1)  
  
disaster_terms = df['keyword'].dropna().unique().tolist()
```

```

def count_terms(s, terms=disaster_terms):
    s = s.lower()
    cnt = 0
    for t in terms:
        if t in s:
            cnt += 1
    return cnt

# Feature engineering adicional
df['disaster_terms_count'] = df['text_clean'].apply(count_terms)

```

Incluyo también columnas de análisis de sentimiento de los tweets

```

In [3]: from vaderSentiment.vaderSentiment import SentimentIntensityAnalyzer
analyzer = SentimentIntensityAnalyzer()

def vader_scores(text):
    if not isinstance(text, str) or text.strip() == "":
        return {'neg': 0.0, 'neu': 0.0, 'pos': 0.0, 'compound': 0.0}
    return analyzer.polarity_scores(text)

scores = [vader_scores(t) for t in df['text'].astype(str).tolist()]
scores_df = pd.DataFrame(scores)

df = pd.concat([df.reset_index(drop=True), scores_df.reset_index(drop=True)], axis=1)

```

Y por último la feature de mayor importancia para todos los modelos, los embeddings.

Si bien no está explícito en este notebook, puedo decir con certeza que son los más importantes ya que al probar retirarlos el puntaje F1 de todos los modelos disminuye drásticamente (por lo menos 0.3 puntos).

Se probó utilizar diferentes embeddings. Entre ellos Word2Vec (GoogleNews-vectors-negative300.bin), GloVe (glove.twitter.27B.zip), Fasttext (cc.en.300.bin) y demás configuraciones integrando pesos con TF-IDF.

En la versión final de todos los modelos utilice Sentence Transformers (también conocidos como SBERT).

Esto se debe a que los demás embeddings que probé están ideados para codificar palabras, por lo que debía ingeniar y probar manualmente diferentes mecanismos para transformar esos vectores de palabras en un único vector para el tweet.

El propósito de SBERT es codificar directamente oraciones ("sentences"), por lo que consideré que llegaría a mejores resultados utilizándolo.

Si bien no incluyo ninguna prueba de resultados de los otros embeddings en este notebook, cabe mencionar que los resultados obtenidos fueron bastante similares, aunque ligeramente mejores utilizando SBERT.

```

In [4]: def clean_text_minimal(s):
    s = str(s).lower()
    s = re.sub(r'http\S+', ' ', s)
    return s.strip()
df['text_emb'] = df['text'].apply(clean_text_minimal)

from sentence_transformers import SentenceTransformer
model_name = "all-MiniLM-L6-v2"
sbert = SentenceTransformer(model_name)
df_sbert = sbert.encode(df['text_emb'].astype(str).tolist(), convert_to_numpy=True)
sbert_columns = [f'sbert_{i}' for i in range(df_sbert.shape[1])]
df = pd.concat([df.reset_index(drop=True), pd.DataFrame(df_sbert, columns=sbert_columns)], axis=1)

```

```

c:\Users\Patricio\miniconda3\envs\ml_env\lib\site-packages\tqdm\auto.py:21: TqdmWarning: IProgress not found. Please update jupyter and ipywidgets. See https://ipywidgets.readthedocs.io/en/stable/user_install.html
  from .autonotebook import tqdm as notebook_tqdm
WARNING:tensorflow:From c:\Users\Patricio\miniconda3\envs\ml_env\lib\site-packages\tf_keras\src\losses.py:2976: The name tf.losses.sparse_softmax_cross_entropy is deprecated. Please use tf.compat.v1.losses.sparse_softmax_cross_entropy instead.

```

Notar que no utilice la columna `text_clean` previamente calculada, ya que en ella se removieron las stopwords.

Como SBERT (y la mayoría de los embeddings) son modelos contextuales, es decir, infieren el significado de las palabras a partir del contexto, dependen fuertemente de la estructura del texto para entender el significado. Como muchas stopwords pueden ser muy relevantes para entender el sentido de los tweets ("not", "but", etc.) es crucial conservarlas para realizar los embeddings.

¿cómo conviene elegir los datos de validación respecto de los de train?

Aquí divido el set original en entrenamiento y validación.

La división debe hacerse de manera tal que el set de validación sea estadísticamente idéntico al set de test ya que este último es lo más parecido que tenemos al "mundo real".

Esta división se hace de forma aleatoria (con una seed para garantizar reproducibilidad) y estratificada utilizando el target. Esto garantiza que se mantenga la misma proporción de clases en train y validación. Como a priori no tenemos los targets del set de test, la manera más fiel que tenemos de representar la "realidad" en el test de validación es basándonos en el set de entrenamiento.

En todos los modelos, utilizo `X_train` para búsqueda de hiperparámetros utilizando CrossValidation.

Luego, entreno el modelo final con los hiperparámetros encontrados utilizando la totalidad de `X_train` y validando con `X_val` para obtener el puntaje F1 solicitado en el enunciado.

```
In [5]: from sklearn.model_selection import train_test_split
X = df
y = df['target'].astype(int).values

X_train, X_val, y_train, y_val = train_test_split(
    X, y, stratify=y, test_size=0.2, random_state=SEED
)
```

Features descartadas

Aprovecho el final de esta sección para nombrar algunas features o estrategias que probé y acabé descartando en el proceso porque no dieron buenos resultados.

- **Embeddings sobre categorías:** Para intentar obtener información útil de las features categóricas, se me ocurrió reaizar embeddings sobre las mismas. Probé utilizar TF-IDF, Word2Vec y SBERT (pues hay varios keywords compuestos por dos palabras), pero obtuve resultados iguales o peores que utilizando las columnas con simples encodings (one hot y mean encoding). Decidí descartar esta estrategia porque además de no mejorar los resultados me pareció que compejizaba el modelo innecesariamente.
- **Clustering de categorías:** Para reducir la cardinalidad de las feature `keyword`, se me ocurrió que podría agrupar palabras "similares" bajo una misma categoría. Hice esto utilizando los embeddings mencionados previamente y agrupándolos con un algoritmo de clustering. Supuse que de esta formaaría dar una vista más genérica al modelo y quizás podría resultar beneficiosa. Utilicé HDBSCAN con diferentes parámetros, y los resultados tuvieron sentido.
Por ejemplo se encontraban juntas las keywords que tienen que ver con "destrucción" ('destroy', 'destroyed', 'destruction'), a diferencia de las relacionadas al fuego ('fire', 'burned', 'burning', 'burning buildings'). Luego, dependiendo de los parámetros elegidos también podían encontrarse asociaciones más o menos generales, por ejemplo, con el valor por defecto de `min_cluster_size` no se agrupaban las palabras que tenían que ver con explosiones, bombas o detonaciones, sin embargo aumentando el valor sí que se encontraban en un mismo cluster.
Dejando de lado los resultados interesantes de cómo se formaban los clusters, el utilizar esta feature tampoco dio mejores resultados que usando las `keywords` con mean encoding. Creo que puede deberse a que esto hacía muy general la información proporcionada, lo cual dificulta la distinción de tweets sobre desastres reales.
Un ejemplo simple que se me ocurre es que alguien podría usar una expresión como "es la bomba" que claramente no habla de un desastre, podría agruparse en el mismo cluster que un tweet que usó "explosión" ("exploded"), u otras palabras considerablemente más específicas y que probablemente sean utilizadas en tweets que sí son desastres.
- **Reducción de Dimensiones:** Intenté utilizar los algoritmos PCA y UMAP para reducir las dimensiones de las features.
Probé utilizarlos para reducir las dimensiones de los embeddings obtenidos de SBERT, y obtuve resultados muy similares que utilizando todas las dimensiones. Podría argumentarse que habría sido buena idea dejar una reducción de dimensiones para que el modelo no tenga tantos datos de entrada y por la posible reducción en el overfitting. Sin embargo, decidí descartar esta estrategia ya que aumentaba considerablemente el tiempo de cómputo y agregaba nuevos hiperparámetros a buscar sin mejorar notablemente los resultados.
También probé reducir las dimensiones del set de entrenamiento completo lo cual dio resultados mucho peores. Considero que puede deberse a que es más difícil reducir dimensiones cuando la información que proporciona es tan diferente, por lo que la información resultante era muy general y no muy útil para realizar las predicciones deseadas.

Primer modelo: KNN

En la siguiente celda se definen las columnas que utiliza el modelo.

Puede verse que hay varias comentadas, esto se debe a que se obtuvieron mejores resultados sin incluir esas columnas.

Supongo que incluir estas columnas no suponían una mejora en el modelo por diversas razones:

Algunas features contienen información redundante:

- `neg`, `neu` y `pos` indican el sentimiento del tweet, pero compound las resume.
- `has_url`, `has_hashtag` y `has_mention` sólo indican si sus respectivas columnas numéricas son mayores a cero.

Las demás features no son redundantes, pero la información que contienen puede resultar tan baja que es indistinguible del ruido que puede haber en el entrenamiento.

Aún así, la diferencia en el puntaje F1 final del modelo varía aproximadamente en ±0.01 puntos, por lo que realmente podría simplemente tratarse de ruido a la hora de entrenar.

Por último, cabe destacar que todo lo que expliqué ahora son suposiciones sin sustento más allá de los resultados que obtuve, por lo que para llegar a puntos más concluyentes debería hacerse un análisis más profundo.

```
In [6]: num_features = [
    'text_len', 'word_count', 'mean_word_len',
    'num_hashtags', 'num_mentions', 'disaster_terms_count',
    'neg', 'pos',
    # 'neu',
    # 'compound',
    'num_exclamations', 'num_questions', 'num_smilies',
    # 'num_repeated_chars', 'num_capital_words',
    # 'numeric_chars_count', 'stopword_ratio', 'unique_word_ratio'
]
bool_features = [
    'has_url', 'has_hashtag', 'has_mention',
    'location_mentioned'
]
```

En la siguiente celda se definen los pre procesamientos que se realizan a cada columna durante cada fold de Cross Validation (y que también deben hacerse luego para predecir datos).

Al estar utilizando únicamente objetos de `sklearn`, este proceso se facilita mucho utilizando un pipeline. Se garantiza que en cada fold de Cross Validation se hará fit únicamente con el set de entrenamiento, previniendo fuga de datos, y luego se transformarán correctamente también los datos de validación.

Utilizo OneHotEncoder para `location` limitando el máximo de categorías a utilizar. El límite de categorías a utilizar puede buscarse como hiperparámetro, pero lo dejé fijo en 20 porque es un valor que me dio buenos resultados y acelera bastante el proceso de búsqueda con GridSearch.

Esto es útil por varios motivos. Como el campo `location` es un texto ingresado a mano por los usuarios, en realidad puede contener cualquier cosa (por ejemplo 'hollywoodland', 'Twitter Lockout in progress', etc.). Como en general esos textos suelen ser únicos o poco frecuentes, convenientemente se agrupan bajo una misma categoría. Por otro lado, también se reduce considerablemente la cantidad de columnas generadas por el encoding.

Para trabajo futuro, sería conveniente hacer una reducción más inteligente de esta columna. Esto podría hacerse encontrando textos diferentes que puedan referir a la misma ubicación ('L.A.' = 'Los Angeles' = 'Los Angeles, USA'), o agrupándolo en ubicaciones más grandes ('Missisipi' -> 'USA'; 'hollywood' -> 'USA').

Para la columna de `keyword` hago un mean encoding usando el `TargetEncoder` de `sklearn.preprocessing`. Probé hacer una reducción manual de la cantidad de categorías diferentes en esta columna pero obtuve resultados peores.

Como KNN es un modelo que se basa completamente en calcular distancia, es necesario escalar las features para que tengan dimensiones similares.

Si no escala las features, las features con valores más grandes dominaría por completo el cálculo de distancia. En cambio, al escalar las features todas tienen valores de una magnitud similar y contribuyen de forma más proporcional al cálculo de la distancia.

Encontré mejores resultados usando el `RobustScaler` de `sklearn.preprocessing` que usando el `StandardScaler`. Supongo que esto se debe a que el primero toma en consideración los outliers de forma más robusta, porque esa es la principal diferencia entre ambos mecanismos.

```
In [7]: from sklearn.preprocessing import RobustScaler, StandardScaler, TargetEncoder, OneHotEncoder
from sklearn.pipeline import Pipeline
from sklearn.neighbors import KNeighborsClassifier
from sklearn.compose import ColumnTransformer

preprocessor = ColumnTransformer(
    transformers=[
        # ('nombre', transformador, columnas_a_aplicar)

        ('num', RobustScaler(), num_features),

        ('bool', 'passthrough', bool_features),

        ('loc_ohe', OneHotEncoder(handle_unknown='ignore', sparse_output=False, max_categories=20), ['location']),

        ('kyw_target', TargetEncoder(random_state=SEED), ['keyword']),

        ('sbert', RobustScaler(), sbert_columns),
    ],
    remainder='drop'
)

full_pipeline = Pipeline(steps=[
    ('preprocessor', preprocessor),
    ('model', KNeighborsClassifier())
])
```

Luego de definir el pipeline, pasamos a la búsqueda de hiperparámetros y evaluación en validación del modelo.

Defino primero el espacio de hiperparámetros a buscar en `param_grid`. Aquí en realidad fui cambiando los parámetros que probé. Comencé utilizando rangos más grandes ([5,10,20,50,100,200]) y fui dando granularidad más específica según iba obteniendo resultados. En el estado final de la celda, dejé algunos parámetros que sé no darán buenos resultados para ilustrar la búsqueda. Puede notarse que hay varios valores de `n_neighbors` consecutivos entre 35 y 40, esto se debe a que en "rondas" anteriores, los mejores hiperparámetros estaban entre 30 y 40 por lo que supuse que el mejor valor estaría entre esos números.

Me parece interesante destacar una tendencia que noté durante las iteraciones: al incluir más features para que utilice el modelo, el mejor valor de `n_neighbors` tiende a ser más alto.

Si bien esto podría ser una simple casualidad, mi hipótesis es que puede deberse a ese "ruido" provocado por incluir demasiadas features. Ese ruido genera que el modelo necesite buscar en más vecinos cercanos para obtener "suavizar" el ruido y obtener una decisión más precisa.

Luego de obtener los mejores hiperparámetros, se pasa a reentrenar el modelo con todo el set de entrenamiento (sin Cross Validation) y a obtener el valor final del puntaje f1 de validación.

```
In [8]: from sklearn.model_selection import GridSearchCV, StratifiedKFold
from sklearn.metrics import f1_score

param_grid = {
    'model_n_neighbors': [5, 10, 20, 30, 35, 37, 38, 39, 40, 42, 50, 60, 100],
    'model_weights': ['uniform', 'distance'],
```

```

'model__metric': ['euclidean', 'manhattan', 'cosine', 'minkowski'],
# 'preprocessor__loc_ohe_max_categories': [10, 20, 50, 100, 200],
}

cv = StratifiedKFold(n_splits=5, shuffle=True, random_state=SEED)
grid = GridSearchCV(full_pipeline, param_grid, scoring='f1', cv=cv, n_jobs=-1, verbose=1)

grid.fit(X_train, y_train)

print("Mejores parámetros:", grid.best_params_)

```

Fitting 5 folds for each of 104 candidates, totalling 520 fits
Mejores parámetros: {'model__metric': 'cosine', 'model__n_neighbors': 38, 'model__weights': 'distance'}

```

In [9]: best_params = grid.best_params_
best_knn = Pipeline(steps=[
    ('preprocessor', preprocessor),
    ('model', KNeighborsClassifier(**{k.replace('model__', ''): v for k, v in grid.best_params_.items() if k.startswith('model_')})])
])

best_knn.set_params(**best_params)
best_knn.fit(X_train, y_train)

y_val_pred = best_knn.predict(X_val)
print("F1 en validación:", f1_score(y_val, y_val_pred))

```

F1 en validación: 0.8018433179723502

Como se observa, el puntaje obtenido es apenas mayor a 0.8.

Este valor se obtuvo luego de refinar al máximo las features seleccionadas y el preprocessamiento de las mismas y considero que ese es el límite alcanzable utilizando estas herramientas.

Para obtener valores más altos y modelos que hagan mejores predicciones, se deberían usar herramientas más complejas y que permitan un mejor análisis de la intención detrás de los tweets.

Segundo modelo: LightGBM

Como segundo modelo utilice LightGBM. La estrategia empleada es muy similar a la utilizada en KNN, con pequeñas diferencias.

A diferencia que en KNN, aquí no excluye ninguna de las features creadas. Esto es así porque el modelo busca automáticamente las más útiles. Se puede configurar la proporción de features tomadas en cuenta con el hiperparámetro `colsample_bytree`.

```

In [10]: from lightgbm import LGBMClassifier

metadata_features = [
    'text_len', 'word_count', 'mean_word_len',
    'num_hashtags', 'num_mentions', 'disaster_terms_count',
    'compound',
    'num_exclamations', 'num_questions', 'num_smilies',
    'num_repeated_chars', 'num_capital_words',
    'has_url', 'has_hashtag', 'has_mention',
    'location_mentioned'
]

preprocessor_lgbm = ColumnTransformer([
    ('sbert', 'passthrough', sbert_columns),
    ('keyword_enc', TargetEncoder(random_state=SEED), ['keyword']),
    ('location_enc', OneHotEncoder(handle_unknown='ignore', sparse_output=False, max_categories=50), ['location']),
    ('meta_scaled', 'passthrough', metadata_features)
], remainder='drop')

```

La otra diferencia con la estrategia usada en KNN es la forma de buscar los hiperparámetros.

Esta vez utilice otra librería para hacerlo, `optuna`. Este framework promete hacer más óptima la búsqueda de hiperparámetros, tanto con optimizaciones internas como agregando mecanismos que pueden ser adoptados por el usuario.

En la siguiente celda defino una función "objetivo" la cual `optuna` intentará maximizar buscando hiperparámetros dentro del rango definido. Esta función objetivo devuelve la media de todos los puntajes F1 obtenidos en los folds de cross validation, por lo que es lo que se buscará maximizar.

La librería permite añadir más optimizaciones a la búsqueda de hiperparámetros, pero no conseguí hacerlo de manera reproducible. Puede verse la adopción de una de estas estrategias en el [anexo](#)

```

In [11]: import optuna
from sklearn.model_selection import cross_val_score

def objective(trial):
    # A. Sugerir Hiperparámetros
    # Optuna elige valores inteligentes dentro de estos rangos
    param = {
        'n_estimators': trial.suggest_int('n_estimators', 100, 1500),
        'num_leaves': trial.suggest_int('num_leaves', 20, 150),
        'max_depth': trial.suggest_int('max_depth', 3, 15),
        'learning_rate': trial.suggest_float('learning_rate', 0.005, 0.2, log=True),
        'subsample': trial.suggest_float('subsample', 0.5, 1.0),
    }

```

```

'colsample_bytree': trial.suggest_float('colsample_bytree', 0.5, 1.0),
'reg_alpha': trial.suggest_float('reg_alpha', 1e-8, 10.0, log=True),
'reg_lambda': trial.suggest_float('reg_lambda', 1e-8, 10.0, log=True),
'min_child_samples': trial.suggest_int('min_child_samples', 5, 100),

'random_state': SEED,
'is_unbalance': True,
'n_jobs': 4,
'verbose': -1,
'force_col_wise': True
}

model = LGBMClassifier(**param)

pipeline = Pipeline([
    ('preprocessor', preprocessor_lgbm),
    ('model', model)
])

cv = StratifiedKFold(n_splits=5, shuffle=True, random_state=SEED)

scores = cross_val_score(pipeline, X_train, y_train, cv=cv, scoring='f1', n_jobs=1)

return scores.mean()

```

La librería permite también configurar finamente el tiempo empleado en la búsqueda. Esta vez utiliceo n_trials para garantizar la reproducibilidad, pero también puede usarse un timeout.

```
In [ ]: optuna_sampler = optuna.samplers.TPESampler(seed=SEED)
lgbm_study = optuna.create_study(direction='maximize', sampler=optuna_sampler)
lgbm_study.optimize(objective, n_trials=10, n_jobs=1)

print("Mejores Hiperparámetros:")
print(lgbm_study.best_params)
```

Para entrenar la versión final del modelo utiliceo estos parámetros hardcodeados que encontré en una búsqueda anterior, pero perdí al volver a correr la celda anterior con parámetros diferentes.

```
In [13]: # best_params = Lgbm_study.best_params
best_params = {'n_estimators': 940, 'num_leaves': 50, 'max_depth': 10, 'learning_rate': 0.018865211249690032, 'subsample': 0.84806}
best_params.update({
    'verbose': -1,
    # necesarios para asegurar la reproducibilidad:
    'random_state': SEED,
    'is_unbalance': True,
    'n_jobs': 1,
    'force_col_wise': True
})

final_lgbm = LGBMClassifier(**best_params)

final_pipeline = Pipeline([
    ('preprocessor', preprocessor_lgbm),
    ('model', final_lgbm)
])

final_pipeline.fit(X_train, y_train)

preds_lgbm_optuna = final_pipeline.predict(X_val)

f1_final_optuna = f1_score(y_val, preds_lgbm_optuna)
print(f"F1 Score Final en X_val: {f1_final_optuna:.5f}")

F1 Score Final en X_val: 0.80671
c:\Users\Patricio\miniconda3\envs\ml_env\lib\site-packages\sklearn\utils\validation.py:2749: UserWarning: X does not have valid feature names, but LGBMClassifier was fitted with feature names
  warnings.warn()
```

Para el mejor modelo de ambos, ¿cuál es el score en la competencia?

Ya que obtuve un puntaje muy similar entre ambos modelos, decidí utilizar el de KNN para predecir los datos de test.

Esto se debe a que KNN es un modelo más simple por lo que, siguiendo el principio de la Navaja de Ockham, podría considerarse un mejor modelo.

```
In [10]: test_path = 'data/test.csv'
test_df = pd.read_csv(test_path)

test_df['text_clean'] = test_df['text'].apply(clean_text)
test_df['keyword'] = test_df['keyword'].fillna('no_keyword_contained')
test_df['location'] = test_df['location'].apply(clean_text)
test_df['location'] = test_df['location'].fillna('no_location_contained')
test_df['word_count'] = test_df['text_clean'].apply(lambda s: len(s.split()))
test_df['text_len'] = test_df['text_clean'].apply(lambda s: sum(len(w) for w in s.split()))
test_df['mean_word_len'] = test_df.apply(lambda row: row['text_len'] / row['word_count'] if row['word_count'] > 0 else 0, axis=1)
test_df['num_hashtags'] = test_df['text'].apply(lambda s: 0 if pd.isna(s) else s.count('#'))
test_df['num_mentions'] = test_df['text'].apply(lambda s: 0 if pd.isna(s) else s.count('@'))
```

```

test_df['num_exclamations'] = test_df['text'].apply(lambda s: 0 if pd.isna(s) else s.count('!'))
test_df['num_questions'] = test_df['text'].apply(lambda s: 0 if pd.isna(s) else s.count('?'))
test_df['num_smilies'] = test_df['text'].apply(lambda s: 0 if pd.isna(s) else len(re.findall(r'[:;=8][\o\*\\]?[\\])\\([\\dDpP:/\\}\\]{', s)))
test_df['num_repeated_chars'] = test_df['text'].apply(lambda s: 0 if pd.isna(s) else len(re.findall(r'(.)\\1{2,}', s)))
test_df['num_capital_words'] = test_df['text'].apply(lambda s: sum(1 for w in str(s).split() if w.isupper())).fillna(0)
test_df['numeric_chars_count'] = test_df['text'].apply(lambda s: 0 if pd.isna(s) else len(re.findall(r'\d', s)))
test_df['stopword_ratio'] = test_df.apply(lambda row: sum(1 for w in str(row['text']).split() if w.lower() in stop_words) / row['word_count'])
test_df['unique_word_ratio'] = test_df.apply(lambda row: len(set([w.strip() for w in row['text_clean'].split()]))) / row['word_count']
test_df['has_url'] = test_df['text'].apply(lambda s: 0 if pd.isna(s) else (1 if 'http' in s or 'www.' in s else 0))
test_df['has_hashtag'] = test_df['num_hashtags'].apply(lambda x: 1 if x > 0 else 0)
test_df['has_mention'] = test_df['num_mentions'].apply(lambda x: 1 if x > 0 else 0)
test_df['location_mentioned'] = test_df.apply(lambda row: 1 if row['location'].lower() in row['text_clean'] else 0, axis=1)
test_df['disaster_terms_count'] = test_df['text_clean'].apply(count_terms)

scores = [vader_scores(t) for t in test_df['text'].astype(str).tolist()]
scores_test_df = pd.DataFrame(scores)
test_df = pd.concat([test_df.reset_index(drop=True), scores_test_df.reset_index(drop=True)], axis=1)

test_df['text_emb'] = test_df['text'].apply(clean_text_minimal)
test_df_sbert = sbert.encode(test_df['text_emb'].astype(str).tolist(), convert_to_numpy=True)
sbert_columns = [f'sbert_{i}' for i in range(test_df_sbert.shape[1])]
test_df = pd.concat([test_df.reset_index(drop=True), pd.DataFrame(test_df_sbert, columns=sbert_columns)], axis=1)

```

```
In [11]: knn_test_preds = best_knn.predict(test_df)
knn_result = pd.DataFrame({
    'id': test_df['id'],
    'target': knn_test_preds,
})
knn_result.to_csv('knn_test_predictions.csv', index=False)
```

Subiendo el archivo resultante a la competencia, obtuve un puntaje F1 de 0.81826.

Es un poco mayor al obtenido en validación (0.80184), lo cual puede deberse a que el set de test es estadísticamente diferente al de validación, o simplemente a variaciones normales en el desempeño del modelo.

Anexo

Me resultó muy difícil obtener puntajes mayores a 0.8 para esta parte.

Obtener 0.79 fue sencillo, de hecho, como se verá en la parte 4, eso puede alcanzarse fácilmente utilizando simplemente KNN y embeddings con SBERT.

Pero pasar ese umbral y conseguir un modelo superador resultó extremadamente costoso. Intenté varios enfoques distintos, diferentes modelos, diferentes features agregadas, etc.

Además de las features adicionales mencionadas al [final de la sección Feature engineering](#), probé algunas estrategias que describo en esta sección.

Modelo Extra: Red Neuronal

Originalmente este iba a ser mi segundo modelo, pero no logré obtener una puntuación mayor a 0.8 de manera consistente.

Intenté utilizar diferentes enfoques, entre ellos diferentes ramas de input de datos, usar una layer LSTM, convoluciones, usar embeddings preentrenados y refinarlos con los datos de entrenamiento, etc. Dejo adjunta la última versión que estuve intentando mejorar, que es una que tiene 3 ramas de input de datos.

```
In [ ]: num_features = [
    'text_len', 'word_count', 'mean_word_len',
    'num_hashtags', 'num_mentions', 'disaster_terms_count',
    # 'neg', 'neu', 'pos',
    'compound',
    'num_exclamations', 'num_questions', 'num_smilies',
    'num_repeated_chars', 'num_capital_words',
    # 'numeric_chars_count', 'stopword_ratio', 'unique_word_ratio'
]
bool_features = [
    # 'has_url', 'has_hashtag', 'has_mention',
    # 'location_mentioned'
]
```

A continuación se encuentra la definición en sí de la red neuronal.

Primeramente se puede observar que se reciben hiperparámetros y las dimensiones de los datos de entrada.

Se pueden diferenciar distintos tipos de capas, entre ellas:

- **Input** define datos de entrada. Debe especificarse la dimensión de la entrada, el tipo de dato y un nombre para poder referenciar los datos que se envíen luego al modelo.
- **Dense** es una capa común, densamente conectada. Puede definirse cantidad de neuronas, función de activación, si usar o no un bias, entre otros. Notar también que se especifica un inicializador para los parámetros para intentar asegurar la reproducibilidad.
- **Dropout** es una capa de regularización. Simplemente desactiva algunas conexiones entre neuronas con cierta probabilidad. Esto hace que el entrenamiento sea más robusto y que las predicciones no dependan únicamente de un conjunto específico de features o neuronas. Aquí también se explicita la semilla para intentar asegurar la reproducibilidad.

- **Batch Normalization** esta es una capa que no vimos en clase y me pareció interesante. Normaliza los datos del batch entrante para pasar datos con una distribución normal a la siguiente capa. Esto facilita el entrenamiento de las diferentes capas, ya que hace que los datos recibidos no tengan escalas dependientes de la capa anterior.
- Para utilizar estas capas la práctica más utilizada es desactivar la activación en la capa densa previa y utilizar la función de activación luego de la normalización. Desconozco exactamente por qué se hace así, pero parece ser la forma en que suele ser implementado y resultó dar buenos resultados.

También se puede notar que la red está definida con tres ramas que reciben diferentes datos de entrada. Esto permite al modelo dar un trato diferente a los diferentes tipos de datos.

Por un lado se reciben los embeddings de SBERT. Esta rama tiene dos capas densas intercaladas con capas de Dropout para regularización. Es notable el declive del resultado cuando no se incluyen las capas de Dropout.

En otra rama se recibe el input categórico, en donde se recibe tanto la categoría `keyword` con mean encoding como `location` con one hot. La estructura de capas empleada aquí es la misma que para el input de SBERT, pero con sus propio hiperparámetro para la cantidad de neuronas. Finalmente, la rama de input numérico recibe las demás features, tanto numéricas como booleanas (aunque como vimos antes no se incluyen variables booleanas en este modelo). Aquí adoptamos por primera vez el patrón para usar Batch Normalization.

Probé diferentes distribuciones para las ramas (por ejemplo, `keyword` que es mean encoding pasarlo junto a las numéricas y `location` que es one hot junto a las booleanas) pero esta fue la que dio ligeramente mejores resultados.

Luego todas las ramas de input se combinan en un último tramo en el que se reduce la cantidad de neuronas por capa hasta llegar a una última neurona de output con una función de activación sigmoidea, la cual dará el resultado de la predicción.

```
In [ ]: from tensorflow.keras.metrics import F1Score
from tensorflow.keras.optimizers import RMSprop
from tensorflow.keras import initializers
import keras
from keras import layers

def build_neural_network(sbert_dim, cat_dim, num_dim, hparams={}):
    SBERT_UNITS = hparams.get('sbert_units', sbert_dim)
    CAT_UNITS = hparams.get('cat_units', cat_dim)
    NUM_UNITS = hparams.get('num_units', num_dim)
    FINAL_UNITS = hparams.get('final_units', 64)
    DROPOUT_RATE = hparams.get('dropout_rate', 0.5)
    LEARNING_RATE = hparams.get('learning_rate', 0.001)

    init = initializers.GlorotUniform(seed=SEED)

    # INPUT SBERT (EMBEDDING)
    input_sbert = keras.Input(shape=(sbert_dim,), dtype="float32", name="input_sbert")
    x_sbert = layers.Dense(SBERT_UNITS, activation='relu', kernel_initializer=init)(input_sbert)
    x_sbert = layers.Dropout(DROPOUT_RATE, seed=SEED)(x_sbert)
    x_sbert = layers.Dense(SBERT_UNITS//2, activation='relu', kernel_initializer=init)(x_sbert)
    x_sbert = layers.Dropout(DROPOUT_RATE, seed=SEED)(x_sbert)

    # INPUT CATEGÓRICO
    input_cat = keras.Input(shape=(cat_dim,), dtype="float32", name="input_cat")
    x_cat = layers.Dense(CAT_UNITS, activation='relu', kernel_initializer=init)(input_cat)
    x_cat = layers.Dropout(DROPOUT_RATE, seed=SEED)(x_cat)
    x_cat = layers.Dense(CAT_UNITS//2, activation='relu', kernel_initializer=init)(x_cat)
    x_cat = layers.Dropout(DROPOUT_RATE, seed=SEED)(x_cat)

    # INPUT NUMÉRICO Y BOOLEANO
    input_num = keras.Input(shape=(num_dim,), dtype="float32", name="input_num")
    x_num = layers.Dense(NUM_UNITS, use_bias=False, kernel_initializer=init)(input_num)
    x_num = layers.BatchNormalization()(x_num)
    x_num = layers.Activation('relu')(x_num)
    x_num = layers.Dropout(DROPOUT_RATE, seed=SEED)(x_num)

    # COMBINACIÓN DE RAMAS Y OUTPUT
    merged = layers.concatenate([x_sbert, x_cat, x_num])
    final_output = layers.Dense(FINAL_UNITS, use_bias=False, kernel_initializer=init)(merged)
    final_output = layers.BatchNormalization()(final_output)
    final_output = layers.Activation('relu')(final_output)
    final_output = layers.Dropout(DROPOUT_RATE, seed=SEED)(final_output)
    final_output = layers.Dense(FINAL_UNITS//2, use_bias=False, kernel_initializer=init)(final_output)
    final_output = layers.Activation('relu')(final_output)
    # final_output = layers.Dropout(DROPOUT_RATE, seed=SEED, kernel_initializer=init)(final_output)
    preds = layers.Dense(1, activation='sigmoid')(final_output)

    model = keras.Model(inputs=[input_sbert, input_cat, input_num], outputs=preds)

    model.compile(
        loss="binary_crossentropy",
        optimizer=RMSprop(learning_rate=LEARNING_RATE),
        metrics=[ "AUC", F1Score(threshold=0.5, name="f1_score") ]
    )
    return model
```

Utilizo una función de poda de `optuna` (`TFKerasPruningCallback`), que permite cortar prematuramente el entrenamiento de un modelo si detecta que no supondrá una mejora al mejor ya encontrado. Esto puede resultar muy útil a la hora de buscar hiperparámetros, pero viene con la contra de tener que implementar un poco más a mano el entrenamiento de cross validation, ya que no es completamente compatible.

En la función objetivo entreno la red neuronal usando cross validation y el set de entrenamiento dividido previamente, dejando el set de validación para la comparación con otros modelos.

Notar que en la función `train_neural_network_model` definí "manualmente" el preprocesamiento de datos y el entrenamiento de los diferentes folds del cross validation, cosa que en el modelo anterior se podía hacer de manera "automática" con un Pipeline y otros objetos de `sklearn`. En este proceso tuve que ser sumamente cuidadoso de no cometer data leaks, ya que suponen un gran problema para el entrenamiento del modelo, dando acceso a datos de validación durante el entrenamiento y permitiéndole "hacer trampa".

También utilice un `EarlyStopper` callback para evitar overfitting. El mismo detiene el entrenamiento si las métricas de validación dejan de mejorar a pesar que las métricas de entrenamiento lo sigan haciendo.

Para entrenar la red neuronal, configuré la métrica AUC-ROC como métrica a verificar por el `EarlyStopper` y el callback de poda de `optuna`. Si bien no hay tanta diferencia con entrenar usando F1, decidí entrenar la red utilizando la curva AUC ya que parece ser lo más estándar y utilizado ampliamente.

Finalmente, es importante devolver los encoders y scalers que se utilizaron durante el entrenamiento ya que son los que se deben utilizar para dar nuevos inputs a predecir para el modelo.

```
In [ ]: from sklearn.model_selection import KFold
from tensorflow.keras.callbacks import EarlyStopping
from optuna.integration import TFKerasPruningCallback

def objective(trial):
    hparams = {
        'learning_rate': trial.suggest_float('learning_rate', 1e-5, 1e-1, log=True),
        'sbert_units': trial.suggest_int('sbert_units', 256, 720, step=8),
        'cat_units': trial.suggest_int('cat_units', 4, 86, step=2),
        'num_units': trial.suggest_int('num_units', 4, 86, step=2),
        'final_units': trial.suggest_int('final_units', 16, 256, step=4),
        'dropout_rate': trial.suggest_float('dropout_rate', 0.1, 0.6),
    }

    kfold = KFold(n_splits=5, shuffle=True, random_state=SEED)
    fold_f1_scores = []
    for fold, (train_idx, val_idx) in enumerate(kfold.split(X_train, y_train)):

        model, x_val_formatted, y_val_formatted, _, _, _ = \
            train_neural_network_model(hparams, X_train.iloc[train_idx], X_train.iloc[val_idx], y_train[train_idx], y_train[val_idx])

        # Evaluar...
        scores = model.evaluate(
            x=x_val_formatted,
            y=y_val_formatted,
            verbose=0
        )

        val_f1 = scores[2] # indice 2 = 'f1_score'
        fold_f1_scores.append(val_f1)

    # 1c. Devuelve el score promedio de este "trial"
    mean_f1 = np.mean(fold_f1_scores)

    # Reporta si el trial fue podado
    if trial.should_prune():
        raise optuna.exceptions.TrialPruned()

    return mean_f1

def train_neural_network_model(hparams, x_train, x_val, y_train, y_val, trial=None):
    y_train_fold = y_train.reshape(-1, 1)
    y_val_fold = y_val.reshape(-1, 1)

    # embedding input
    X_sbert_train = x_train[sbert_columns]
    X_sbert_val = x_val[sbert_columns]

    oh_encoder = OneHotEncoder(handle_unknown='ignore', max_categories=20, sparse_output=False)
    mean_encoder = TargetEncoder(random_state=SEED)

    # categorical input
    # onehot para Location + target encoding para keywords
    X_cat_train = np.hstack([
        mean_encoder.fit_transform(x_train[['keyword']]), y_train_fold),
        oh_encoder.fit_transform(x_train[['location']])))
    ])
    X_cat_val = np.hstack([
        mean_encoder.transform(x_val[['keyword']])),
        oh_encoder.transform(x_val[['location']])))
    ])

    numeric_transformer = StandardScaler()
    X_num_train = numeric_transformer.fit_transform(np.hstack([x_train[num_features], x_train[bool_features]])))
    X_num_val = numeric_transformer.transform(np.hstack([x_val[num_features], x_val[bool_features]])))

    model = build_neural_network(
        sbert_dim=X_sbert_train.shape[1],
        cat_dim=X_cat_train.shape[1],
        num_dim=X_num_train.shape[1],
```

```

        hparams=hparams
    )

    early_stopper = EarlyStopping(monitor="AUC", mode="max", patience=5, restore_best_weights=True)

    if trial is not None:
        # poda de Optuna
        pruning_callback = TFKerasPruningCallback(trial, "AUC")

    validation_data = (
        {'input_sbert': X_sbtert_val, 'input_cat': X_cat_val, 'input_num': X_num_val},
        y_val_fold
    )

    model.fit(
        x={'input_sbtert': X_sbtert_train, 'input_cat': X_cat_train, 'input_num': X_num_train},
        y=y_train_fold,
        batch_size=128,
        epochs=30,
        validation_data=validation_data,
        callbacks=[early_stopper, pruning_callback] if trial is not None else [early_stopper],
        verbose=0
    )

    return model, validation_data[0], validation_data[1], mean_encoder, oh_encoder, numeric_transformer

```

En la siguiente celda comienzo la búsqueda de hiperparámetros. Esta búsqueda es muy personalizable y permite decidir fácilmente cuánto tiempo invertir en la misma.

```
In [ ]: import optuna.samplers

optuna_sampler = optuna.samplers.TPESampler(seed=SEED)
nn_study = optuna.create_study(direction="maximize", sampler=optuna_sampler)

nn_study.optimize(objective, timeout=120)

print(f"Mejor F1 Score: {nn_study.best_value:.4f}")
print("Mejores Hiperparámetros:")
print(nn_study.best_params)
```

Un detalle importante de este modelo es que, a pesar de mis mayores esfuerzos por establecer una semilla y utilizar todas las funciones que encontré, no logré hacer que sea reproducible. Cada vez que vuelvo a entrenar el modelo con los mismos parámetros obtengo un resultado diferente de score F1.

Aún así, los máximos resultados que obtuve no superaron 0.81 de puntaje F1 en validación.

```
In [ ]: nn_x_train, nn_x_val, nn_y_train, nn_y_val = train_test_split(
    X_train, y_train, test_size=0.2, random_state=SEED
)
best_nn, _, _, mean_encoder, oh_encoder, numeric_transformer = \
    train_neural_network_model(
        nn_study.best_params,
        nn_x_train,
        nn_x_val,
        nn_y_train,
        nn_y_val
    )

# Validación final
scores = best_nn.evaluate(
    x={
        'input_sbtert': X_val[sbtert_columns],
        'input_cat': np.hstack([
            mean_encoder.transform(X_val[['keyword']]),
            oh_encoder.transform(X_val[['location']])
        ]),
        'input_num': numeric_transformer.transform(
            np.hstack([X_val[num_features], X_val[bool_features]])
        )
    },
    y=y_val.reshape(-1, 1),
    verbose=0
)

f1_nn = scores[2] # indice 2 = 'f1_score'
print(f"Mejor F1 en validación: {f1_nn:.4f}")
```

```
c:\Users\Patricio\miniconda3\envs\ml_env\lib\site-packages\sklearn\preprocessing\_label.py:110: DataConversionWarning: A column-vector y was passed when a 1d array was expected. Please change the shape of y to (n_samples, ), for example using ravel().
  y = column_or_1d(y, warn=True)
Mejor F1 en validación: 0.7846
```

Blending

Otra estrategia que intenté utilizar fue combinar las predicciones de dos modelos.

Combinando la red neuronal con LightGBM se obtienen ligeramente mejores puntajes F1. Si bien esto podría ser ruido, también es razonable pensar que los modelos se complementan bien: la red neuronal podría estar, por ejemplo, haciendo predicciones más certeras basándose en los embeddings mientras que LGBM podría ser mejor analizando los metadatos. De esta forma se podrían obtener resultados superadores, pero aún así no conseguí que sean mucho mejores.

```
In [ ]: preds_lgbm = final_pipeline.predict_proba(X_val)[:, 1]
preds_nn = best_nn.predict(x={
    'input_sbert': X_val[sbert_columns],
    'input_cat': np.hstack([
        mean_encoder.transform(X_val[['keyword']]),
        oh_encoder.transform(X_val[['location']])
    ]),
    'input_num': numeric_transformer.transform(np.hstack([
        X_val[num_features],
        X_val[bool_features]
    ]))
})
best_f1 = 0
best_weight = 0

for nn_weight in np.arange(0.1, 1.0, 0.1):
    lgbm_weight = 1.0 - nn_weight

    # (weight_nn * preds_nn) + (weight_Lgbm * preds_Lgbm)
    blend_preds = (preds_nn.reshape(-1) * nn_weight) + (preds_lgbm * lgbm_weight)

    final_preds = (blend_preds > 0.5).astype(int)
    current_f1 = f1_score(y_val, final_preds)

    print(f"Peso NN: {nn_weight:.1f} | Peso LGBM: {lgbm_weight:.1f} | F1: {current_f1:.4f}")

    if current_f1 > best_f1:
        best_f1 = current_f1
        best_weight = nn_weight

print("\n--- Mejor Resultado ---")
print(f"Mejor F1 solo LGBM: {f1_final_optuna:.4f}")
print(f"Mejor F1 solo NN: {f1_nn:.4f}")
print(f"Mejor F1 de Blending: {best_f1:.4f}")
print(f"Mejor Peso (para NN): {best_weight:.1f}")
print(f"Mejor Peso (para LGBM): {1.0-best_weight:.1f}")

--- Buscando el Peso Óptimo ---
1/48 ━━━━━━━━ 4s 98ms/step
c:\Users\Patricio\miniconda3\envs\ml_env\lib\site-packages\sklearn\utils\validation.py:2749: UserWarning: X does not have valid feature names, but LGBMClassifier was fitted with feature names
  warnings.warn(
48/48 ━━━━━━━━ 0s 3ms/step
Peso NN: 0.1 | Peso LGBM: 0.9 | F1: 0.7920
Peso NN: 0.2 | Peso LGBM: 0.8 | F1: 0.7975
Peso NN: 0.3 | Peso LGBM: 0.7 | F1: 0.7936
Peso NN: 0.4 | Peso LGBM: 0.6 | F1: 0.7920
Peso NN: 0.5 | Peso LGBM: 0.5 | F1: 0.7903
Peso NN: 0.6 | Peso LGBM: 0.4 | F1: 0.7885
Peso NN: 0.7 | Peso LGBM: 0.3 | F1: 0.7876
Peso NN: 0.8 | Peso LGBM: 0.2 | F1: 0.7867
Peso NN: 0.9 | Peso LGBM: 0.1 | F1: 0.7846

--- Mejor Resultado ---
Mejor F1 solo LGBM: 0.7914
Mejor F1 solo NN: 0.7846
Mejor F1 de Blending: 0.7975
Mejor Peso (para NN): 0.2
Mejor Peso (para LGBM): 0.8
```

Fine-tuning de un modelo preentrenado (distilbert)

Aunque por muy poco, este fue modelo que mejor puntaje obtuvo. Se trata de un modelo preentrenado de HuggingFace, al cual le hice fine-tuning utilizando el set de entrenamiento para que pueda predecir el target.

Aún utilizando un transformer complejo con fine-tuning, el puntaje obtenido es apenas 0.81. Esto habla de la complejidad del problema.

Seguramente haya que dar con algún dato clave en particular para lograr obtener mejores puntajes.

```
In [2]: import evaluate
from datasets import Dataset
from transformers import (
    AutoTokenizer,
    AutoModelForSequenceClassification,
    TrainingArguments,
    Trainer
)
MODEL_CHECKPOINT = "distilbert-base-uncased"
```

```
c:\Users\Patricio\miniconda3\envs\ml_env\lib\site-packages\tqdm\auto.py:21: TqdmWarning: IProgress not found. Please update jupyter and ipywidgets. See https://ipywidgets.readthedocs.io/en/stable/user_install.html
  from .autonotebook import tqdm as notebook_tqdm
WARNING:tensorflow:From c:\Users\Patricio\miniconda3\envs\ml_env\lib\site-packages\tf_keras\src\losses.py:2976: The name tf.losses.sparse_softmax_cross_entropy is deprecated. Please use tf.compat.v1.losses.sparse_softmax_cross_entropy instead.
```

```
In [3]: from sklearn.model_selection import train_test_split
```

```
df = pd.read_csv('data/train.csv')

df['text'] = df['text'].fillna("")
df = df.rename(columns={'target': 'label'})

df_train, df_val = train_test_split(
    df,
    test_size=0.2,
    random_state=SEED,
    stratify=df['label']
)

dataset_train = Dataset.from_pandas(df_train)
dataset_val = Dataset.from_pandas(df_val)
```

```
In [4]: tokenizer = AutoTokenizer.from_pretrained(MODEL_CHECKPOINT)
```

```
def tokenize_function(examples):
    return tokenizer(examples["text"], truncation=True, padding="max_length", max_length=128)

tokenized_train = dataset_train.map(tokenize_function, batched=True)
tokenized_val = dataset_val.map(tokenize_function, batched=True)

tokenized_train = tokenized_train.remove_columns(["id", "keyword", "location", "text", "__index_level_0__"])
tokenized_val = tokenized_val.remove_columns(["id", "keyword", "location", "text", "__index_level_0__"])
```

```
Map: 100%|██████████| 6090/6090 [00:00<00:00, 22105.35 examples/s]
Map: 100%|██████████| 1523/1523 [00:00<00:00, 17176.97 examples/s]
```

```
In [5]: labels = ["Not Disaster", "Disaster"]
id2label = {0: "Not Disaster", 1: "Disaster"}
label2id = {"Not Disaster": 0, "Disaster": 1}
```

```
model = AutoModelForSequenceClassification.from_pretrained(
    MODEL_CHECKPOINT,
    num_labels=2,
    id2label=id2label,
    label2id=label2id
)
```

```
Some weights of DistilBertForSequenceClassification were not initialized from the model checkpoint at distilbert-base-uncased and are newly initialized: ['classifier.bias', 'classifier.weight', 'pre_classifier.bias', 'pre_classifier.weight']
You should probably TRAIN this model on a down-stream task to be able to use it for predictions and inference.
```

```
In [6]: f1_metric = evaluate.load("f1")
accuracy_metric = evaluate.load("accuracy")
```

```
def compute_metrics(eval_pred):
    logits, labels = eval_pred

    predictions = np.argmax(logits, axis=-1)

    f1 = f1_metric.compute(predictions=predictions, references=labels)["f1"]
    acc = accuracy_metric.compute(predictions=predictions, references=labels)["accuracy"]

    return {
        "f1": f1,
        "accuracy": acc
    }
```

```
In [10]: training_args = TrainingArguments(
```

```
    output_dir='./results',
    eval_strategy="epoch",
    save_strategy="epoch",

    num_train_epochs=3,
    per_device_train_batch_size=16,
    per_device_eval_batch_size=16,

    learning_rate=2e-5,
    weight_decay=0.01,

    load_best_model_at_end=True,
    metric_for_best_model="f1",
    greater_is_better=True,
)

trainer = Trainer(
    model=model,
```

```
    args=training_args,
    train_dataset=tokenized_train,
    eval_dataset=tokenized_val,
    tokenizer=tokenizer,
    compute_metrics=compute_metrics,
)

trainer.train()
```

```
C:\Users\Patricio\AppData\Local\Temp\ipykernel_30620\2050177316.py:18: FutureWarning: `tokenizer` is deprecated and will be removed
in version 5.0.0 for `Trainer.__init__`. Use `processing_class` instead.
  trainer = Trainer(
c:\Users\Patricio\miniconda3\envs\ml_env\lib\site-packages\torch\utils\data\dataloader.py:668: UserWarning: 'pin_memory' argument i
s set as true but no accelerator is found, then device pinned memory won't be used.
  warnings.warn(warn_msg)
```

[1143/1143 43:39, Epoch 3/3]

Epoch	Training Loss	Validation Loss	F1	Accuracy
1	No log	0.366562	0.818033	0.854235
2	0.406200	0.420679	0.807202	0.831254
3	0.283200	0.431146	0.811866	0.841760

```
c:\Users\Patricio\miniconda3\envs\ml_env\lib\site-packages\torch\utils\data\dataloader.py:668: UserWarning: 'pin_memory' argument i
s set as true but no accelerator is found, then device pinned memory won't be used.
  warnings.warn(warn_msg)
c:\Users\Patricio\miniconda3\envs\ml_env\lib\site-packages\torch\utils\data\dataloader.py:668: UserWarning: 'pin_memory' argument i
s set as true but no accelerator is found, then device pinned memory won't be used.
  warnings.warn(warn_msg)
```

```
Out[10]: TrainOutput(global_step=1143, training_loss=0.33385882707197, metrics={'train_runtime': 2621.0148, 'train_samples_per_second': 6.9
71, 'train_steps_per_second': 0.436, 'total_flos': 605044843361280.0, 'train_loss': 0.33385882707197, 'epoch': 3.0})
```

```
In [11]: eval_results = trainer.evaluate()
```

```
print(f"Puntaje F1 en validación: {eval_results['eval_f1']:.4f}")
```

```
c:\Users\Patricio\miniconda3\envs\ml_env\lib\site-packages\torch\utils\data\dataloader.py:668: UserWarning: 'pin_memory' argument i
s set as true but no accelerator is found, then device pinned memory won't be used.
  warnings.warn(warn_msg)
```

[96/96 00:43]

Puntaje F1 en validación: 0.8180

Parte 4: Visualización de embeddings

Elegí realizar a consigna adicional de visualizar embeddings de campo `text` aplicando una técnica de reducción de dimensiones.

Incluyo visualizaciones del campo `text` vectorizado utilizando TF-IDF, Word2Vec preentrenado (GoogleNews-vectors-negative300.bin) y SBERT preentrenado (all-MiniLM-L6-v2).

La reducción de dimensiones probé hacerlas con t-SNE y UMAP. Dejé en el notebook la que dio resultados con visualizaciones más claras en cada caso. En mi caso t-SNE dio mejores visualizaciones.

```
In [1]: from sklearn.manifold import TSNE
import re
from umap import UMAP
import seaborn as sns
import matplotlib.pyplot as plt
import random
import numpy as np
SEED = 42
random.seed(SEED)
np.random.seed(SEED)
CUSTOM_PALLETE = ["#00A7E1", "#FFA630"]

import pandas as pd
train_path = 'data/train.csv'
df = pd.read_csv(train_path)
def clean_text(s):
    s = str(s).lower()
    s = re.sub(r'http\S+', ' ', s)
    return s.strip()
df['text_emb'] = df['text'].apply(clean_text)
```

TF-IDF

Para comenzar hago visualizaciones sobre una vectorización simple usando TF-IDF.

Es interesante notar la gran diferencia entre la reducción realizada por UMAP y por t-SNE.

En la visualización de UMAP no hay un patrón distinguible que permita diferenciar en principio las dos clases, los puntos están bastante mezclados y dispersos.

En cambio, en la visualización de t-SNE, si bien los datos siguen bastante mezclados se puede distinguir una clara "granularidad" mayor en esas mezclas. Se pueden distinguir claramente pequeños clusters donde se agrupan puntos de una misma clase. Un modelo de machine learning podría aprender a distinguir esas secciones donde se encuentran esas mayores concentraciones de los targets para hacer predicciones.

Son notables las superposiciones de puntos de diferente clase en los bordes de esas secciones, lo cual podría indicar que las predicciones sobre esos límites vayan a ser malas.

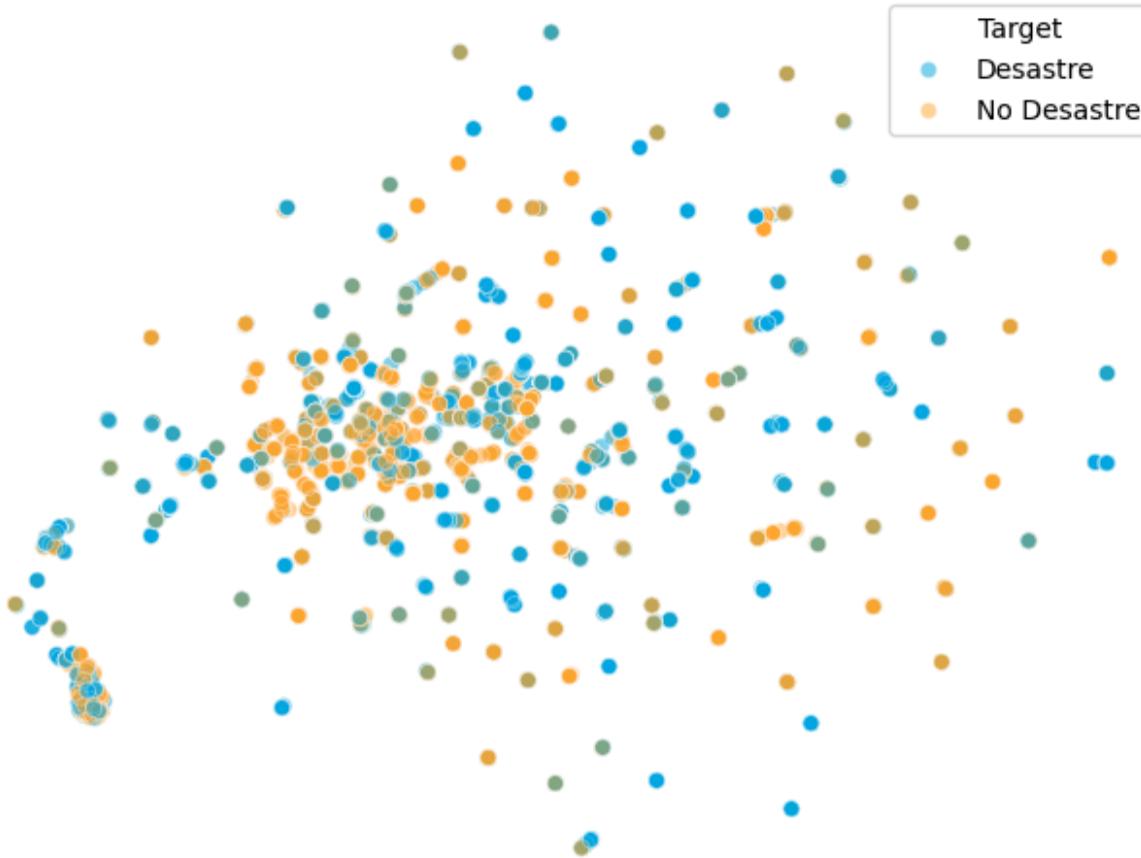
También es importante notar que las "formas" de las divisiones entre las secciones que contienen alta densidad de puntos pertenecientes a un mismo target son muy complejas, en otras palabras, puede resultar muy difícil para un modelo poder aprender de forma correcta esas divisiones.

```
In [2]: from sklearn.feature_extraction.text import TfidfVectorizer
vectorizer = TfidfVectorizer(max_features=400, stop_words='english')
dim_reduc = UMAP(n_components=2, random_state=SEED)
X_embedded = vectorizer.fit_transform(df['text_emb'])
X_2d = dim_reduc.fit_transform(X_embedded.toarray())

df_plot = pd.DataFrame({
    'x': X_2d[:,0],
    'y': X_2d[:,1],
    'target': ['Desastre' if t == 1 else 'No Desastre' for t in df.target.values]
})
plt.figure(figsize=(8,6))
sns.scatterplot(
    data=df_plot,
    x='x',
    y='y',
    hue='target',
    palette=CUSTOM_PALLETE,
    alpha=0.5
)
plt.title("Visualización de embeddings de tweets (UMAP y tSNE)")
plt.legend(title='Target')
plt.axis('off')
plt.show()
```

```
c:\Users\Patricio\miniconda3\envs\tp3_env\Lib\site-packages\umap\umap_.py:1952: UserWarning: n_jobs value 1 overridden to 1 by setting random_state. Use no seed for parallelism.
  warn(
```

Visualización de embeddings de tweets (UMAP y tSNE)



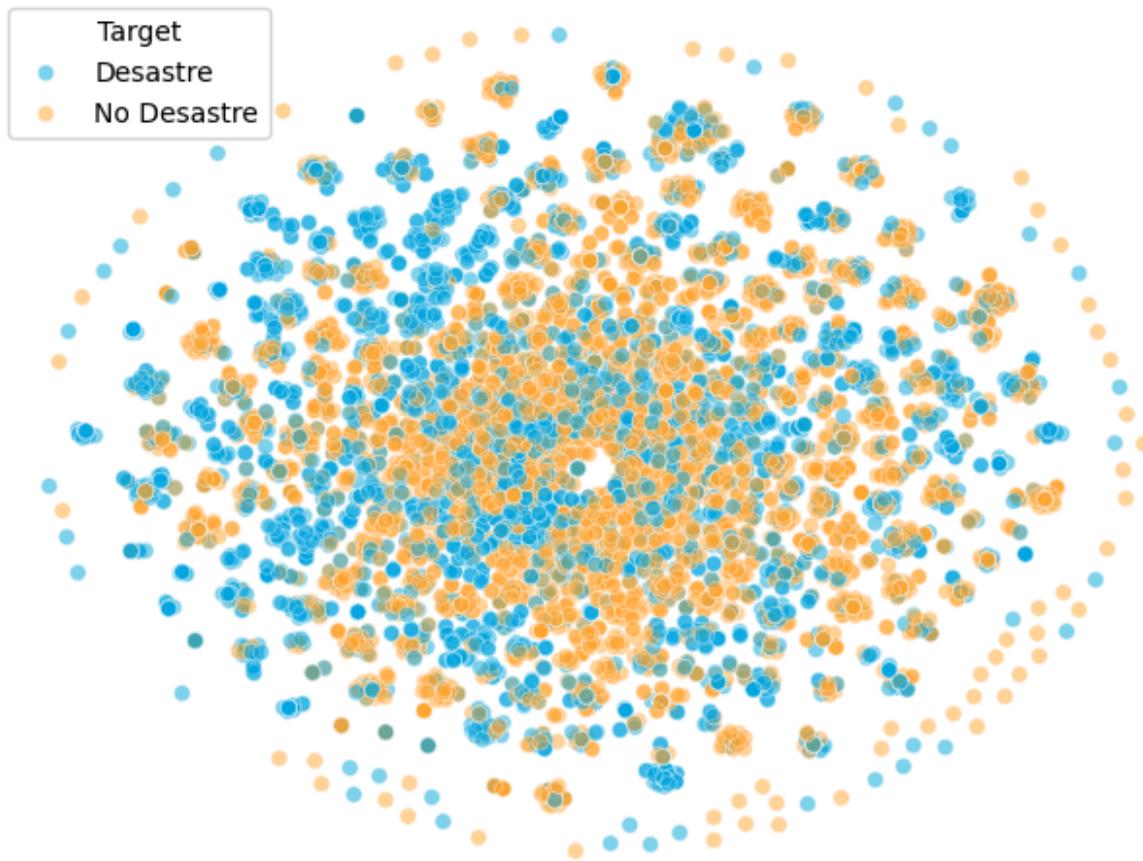
```
In [3]: dim_reduc = TSNE(n_components=2, random_state=SEED, perplexity=40)
X_2d = dim_reduc.fit_transform(X_embedded.toarray())

df_plot = pd.DataFrame({
    'x': X_2d[:,0],
    'y': X_2d[:,1],
    'target': ['Desastre' if t == 1 else 'No Desastre' for t in df.target.values]
})
plt.figure(figsize=(8,6))
sns.scatterplot(
    data=df_plot,
    x='x',
    y='y',
    hue='target',
    palette=CUSTOM_PALLETE,
    alpha=0.5
)
plt.title("Visualización de embeddings de tweets (TF-IDF y tSNE)")
plt.legend(title='Target')
plt.axis('off')
plt.show()
```

```
c:\Users\Patricio\miniconda3\envs\tp3_env\Lib\site-packages\threadpoolctl.py:1226: RuntimeWarning:
Found Intel OpenMP ('libiomp') and LLVM OpenMP ('libomp') loaded at
the same time. Both libraries are known to be incompatible and this
can cause random crashes or deadlocks on Linux when loaded in the
same Python program.
Using threadpoolctl may cause crashes or deadlocks. For more
information and possible workarounds, please see
    https://github.com/joblib/threadpoolctl/blob/master/multiple_openmp.md

warnings.warn(msg, RuntimeWarning)
```

Visualización de embeddings de tweets (TF-IDF y tSNE)



Word2Vec

Utilizo un modelo de Word2Vec preentrenado que produce embeddings de 300 dimensiones.

Como Word2Vec transforma palabras en vectores, y lo que necesitamos es transformar tweets (conjuntos de palabras) en vectores, hay que buscar alguna manera de combinar los vectores de palabras de un tweet para conseguir un vector único que represente el tweet.

Esto se puede hacer de diferentes formas, por ejemplo simplemente sumando los vectores de las palabras que conforman un tweet.

Otra forma posible que me pareció interesante y que dio un gráfico interesante fue haciendo un promedio ponderado, utilizando los valores de TF-IDF de las palabras como pesos. Esta es la estrategia que adopté para la visualización, pero el resultado no es muy diferente que haciendo un promedio balanceado.

Podemos visualizar en el gráfico que ahora los puntos de las clases se separan de una forma notoriamente más sencilla. Parecería haber una frontera difusa que permite distinguir un grupo de mucha concentración de puntos naranjas a la derecha y puntos azules a la izquierda. Aún teniendo esta división con una forma más simple, también podemos ver que los puntos siguen estando bastante mezclados en las zonas donde hay más puntos.

```
In [4]: from gensim.models import KeyedVectors
w2v_path = "GoogleNews-vectors-negative300.bin"
w2v = KeyedVectors.load_word2vec_format(w2v_path, binary=True)
```

```
In [5]: feature_names = vectorizer.get_feature_names_out()
idf = dict(zip(feature_names, vectorizer.idf_))
max_idf = max(idf.values()) if idf else 1.0
tfidf_dict = {w: idf.get(w, 0.0)/max_idf for w in idf}

use_tfidf_weights = True
def sentence_vector_avg(tokens, model, dim):
    if use_tfidf_weights:
        vec = np.zeros(dim, dtype=float)
        w_sum = 0.0
        for w in tokens:
            wkey = w if w in model else (w.lower() if w.lower() in model else None)
            if wkey is None:
                continue
            tfidf_w = tfidf_dict.get(w, 0.0)
            vec += model[wkey] * tfidf_w
            w_sum += tfidf_w
        if w_sum == 0:
            return np.zeros(dim, dtype=float)
        return vec / w_sum
    else:
        if not tokens:
            return np.zeros(dim, dtype=float)
        vecs = []
        for w in tokens:
            if w in model:
                vecs.append(model[w])
            elif w.lower() in model:
                vecs.append(model[w.lower()])
        if len(vecs) == 0:
            return np.zeros(dim, dtype=float)
        return np.mean(vecs, axis=0)

token_pattern = re.compile(r"\w+")
```

```

def tokenize(text):
    return token_pattern.findall(text)
texts = df['text_emb'].astype(str).tolist()
tokens_list = [tokenize(t) for t in texts]

vecs = []
labels = []
for i in range(len(tokens_list)):
    tokens = tokens_list[i]
    v = sentence_vector_avg(tokens, w2v, dim=w2v.vector_size)
    vecs.append(v)
    labels.append(int(df.iloc[i]['target']))

w2v_embeddings = np.vstack(vecs)

```

```
In [6]: dim_reduc = TSNE(n_components=2, random_state=SEED)
X_2d = dim_reduc.fit_transform(w2v_embeddings)
```

```
In [7]: plt.figure(figsize=(10,7))
sns.scatterplot(
    x=X_2d[:,0],
    y=X_2d[:,1],
    hue=labels,
    palette=CUSTOM_PALLETE,
    alpha=0.4
)
plt.title('Visualización con UMAP de Word2Vec preentrenado (GoogleNews-vectors-negative300)')
plt.legend(title='Target', labels=['Desastre', 'No Desastre'])
plt.axis('off')
plt.show()
```

Visualización con UMAP de Word2Vec preentrenado (GoogleNews-vectors-negative300)



SBERT

A continuación muestro una visualización utilizando el mismo embedding preentrenado de SentenceTransformers que utilicé para los modelos de la parte 2 y 3.

```
In [8]: from sentence_transformers import SentenceTransformer
texts = df['text_emb'].astype(str).tolist()
y = df['target'].astype(int).values

model_name = "all-MiniLM-L6-v2"
print("Cargando SentenceTransformer:", model_name)
sbert = SentenceTransformer(model_name)
```

WARNING:tensorflow:From c:\Users\Patricio\miniconda3\envs\tp3_env\Lib\site-packages\tf_keras\src\losses.py:2976: The name tf.losses.sparse_softmax_cross_entropy is deprecated. Please use tf.compat.v1.losses.sparse_softmax_cross_entropy instead.

Cargando SentenceTransformer: all-MiniLM-L6-v2

```
In [9]: sbert_embeddings = sbert.encode(texts, show_progress_bar=True, batch_size=64, convert_to_numpy=True)
```

Batches: 0% | 0/119 [00:00<?, ?it/s]

Se puede apreciar un resultado sorprendentemente similar al producido utilizando los vectores de Word2Vec.

La diferencia que noto es que en este gráfico se pueden notar clústeres más densos de puntos de Desastres, lo cual podría facilitar la decisión de un modelo de aprendizaje automático cuando un punto cayera en esos espacios.

También puede notarse que, a grandes rasgos, la frontera entre los dos grandes grupos de desastre/no desastre es más difusa y se mezclan más.

```
In [10]: reducer = TSNE(n_components=2, random_state=SEED, perplexity=30)
X_2d = reducer.fit_transform(sbert_embeddings)

df_plot = pd.DataFrame({
    'x': X_2d[:,0],
    'y': X_2d[:,1],
    'target': y
})
df_plot['target_name'] = df_plot['target'].map({0: 'No Desastre', 1: 'Desastre'})

plt.figure(figsize=(10,7))
ax = sns.scatterplot(data=df_plot, x='x', y='y', hue='target_name',
                      palette=CUSTOM_PALLETE[::-1], alpha=0.5, s=30)
ax.set_title("Visualización Sentence Transformers reducidos con t-SNE")
plt.legend(title='Target')
plt.axis('off')
plt.show()
```

Visualización Sentence Transformers reducidos con t-SNE



```
In [11]: dim_reduc = TSNE(n_components=3, random_state=SEED)
embeddings_3d = dim_reduc.fit_transform(sbert_embeddings)
```

Por último incluyo una visualización 3d de este último embedding.

```
In [12]: import plotly.express as px
y_str = df.target.map({0: 'No Desastre', 1: 'Desastre'}).astype(str)
df_plot = pd.DataFrame({
    'Dim1': embeddings_3d[:, 0],
    'Dim2': embeddings_3d[:, 1],
    'Dim3': embeddings_3d[:, 2],
    'Target': y_str,
    'Text': df['text_emb'].str[:50] + '...'
})
df_subset = df_plot.sample(frac=0.5, random_state=SEED)
# Gráfico interactivo
fig = px.scatter_3d(
    df_subset,
    x='Dim1', y='Dim2', z='Dim3',
    color='Target',
    hover_name='Text',
    hover_data={
        'Dim1': False,
        'Dim2': False,
        'Dim3': False,
        'Target': True,
        'Text': False
    },
    title='t-SNE 3D de Embeddings SBERT',
```

```

    color_discrete_map={'No Desastre': CUSTOM_PALLETE[0], 'Desastre': CUSTOM_PALLETE[1]}
)

fig.update_layout(showlegend=False)
fig.update_traces(marker=dict(size=3, opacity=0.7))
fig.update_layout(
    width=900,
    height=700,
    showlegend=False,
    scene=dict(
        xaxis=dict(visible=False),
        yaxis=dict(visible=False),
        zaxis=dict(visible=False)
    )
)

fig.show()

```

Para obtener una versión interactiva, puede correrse [este notebook](#)

A grandes rasgos, las características son las mismas que en las visualizaciones 2d anteriores.

¿Sería posible predecir el target con estos embeddings?

Para responder esta pregunta voy a entrenar un modelo de KNN utilizando únicamente los embeddings, reduciendo a diferentes cantidades de dimensiones y comparando resultados.

```
In [13]: from sklearn.model_selection import GridSearchCV, StratifiedKFold
from sklearn.metrics import f1_score, classification_report
from sklearn.preprocessing import RobustScaler
from sklearn.pipeline import Pipeline
from sklearn.neighbors import KNeighborsClassifier
from sklearn.model_selection import train_test_split

knn_pipeline = Pipeline(steps=[
    ('preprocessor', RobustScaler()),
    ('model', KNeighborsClassifier())
])
```

```
In [14]: def train_val_knn(X_train, X_val, y_train, y_val, preprocessor = RobustScaler()):
    param_grid = {
        'model_n_neighbors': [5, 10, 20, 30, 40, 50, 60, 70, 100, 125, 150, 200],
        'model_weights': ['uniform', 'distance'],
        'model_metric': ['euclidean', 'manhattan', 'cosine', 'minkowski'],
    }

    knn_pipeline = Pipeline(steps=[
        ('preprocessor', preprocessor),
        ('model', KNeighborsClassifier())
    ])

    cv = StratifiedKFold(n_splits=5, shuffle=True, random_state=SEED)
    grid = GridSearchCV(knn_pipeline, param_grid, scoring='f1', cv=cv, n_jobs=-1, verbose=0)

    grid.fit(X_train, y_train)

    best_knn = Pipeline(steps=[
        ('preprocessor', preprocessor),
        ('model', KNeighborsClassifier(**{k.replace('model__', ''): v for k, v in grid.best_params_.items()}))
    ])
    best_knn.fit(X_train, y_train)

    y_val_pred = best_knn.predict(X_val)
    return y_val_pred
```

```
In [15]: from sklearn.preprocessing import MaxAbsScaler

all_results = {}

X = df['text_emb']
y = df['target'].astype(int).values

X_train, X_val, y_train, y_val = train_test_split(
    X, y, stratify=y, test_size=0.2, random_state=SEED
)

vectorizer = TfidfVectorizer(max_features=100, stop_words='english')
X_train_tfidf = vectorizer.fit_transform(X_train)
X_val_tfidf = vectorizer.transform(X_val)
y_val_pred = train_val_knn(X_train_tfidf, X_val_tfidf, y_train, y_val, preprocessor=MaxAbsScaler())
score = f1_score(y_val, y_val_pred)
all_results['TF-IDF 100'] = score

vectorizer = TfidfVectorizer(max_features=500, stop_words='english')
X_train_tfidf = vectorizer.fit_transform(X_train)
```

```

X_val_tfidf = vectorizer.transform(X_val)
y_val_pred = train_val_knn(X_train_tfidf, X_val_tfidf, y_train, y_val, preprocessor=MaxAbsScaler())
score = f1_score(y_val, y_val_pred)
all_results['TF-IDF 500'] = score

vectorizer = TfidfVectorizer(max_features=1000, stop_words='english')
X_train_tfidf = vectorizer.fit_transform(X_train)
X_val_tfidf = vectorizer.transform(X_val)
y_val_pred = train_val_knn(X_train_tfidf, X_val_tfidf, y_train, y_val, preprocessor=MaxAbsScaler())
score = f1_score(y_val, y_val_pred)
all_results['TF-IDF 1000'] = score

```

```

In [16]: X = w2v_embeddings
y = df['target'].astype(int).values

X_train, X_val, y_train, y_val = train_test_split(
    X, y, stratify=y, test_size=0.2, random_state=SEED
)

y_val_pred = train_val_knn(X_train, X_val, y_train, y_val)
score = f1_score(y_val, y_val_pred)
all_results['WORD2VEC 300'] = score

dim_reduc = UMAP(n_components=100, random_state=SEED)
X_train_reduced = dim_reduc.fit_transform(X_train)
X_val_reduced = dim_reduc.transform(X_val)
y_val_pred = train_val_knn(X_train_reduced, X_val_reduced, y_train, y_val)
score = f1_score(y_val, y_val_pred)
all_results['WORD2VEC 100'] = score

dim_reduc = UMAP(n_components=50, random_state=SEED)
X_train_reduced = dim_reduc.fit_transform(X_train)
X_val_reduced = dim_reduc.transform(X_val)
y_val_pred = train_val_knn(X_train_reduced, X_val_reduced, y_train, y_val)
score = f1_score(y_val, y_val_pred)
all_results['WORD2VEC 50'] = score

```

```

c:\Users\Patricio\miniconda3\envs\tp3_env\Lib\site-packages\umap\umap_.py:1952: UserWarning:
n_jobs value 1 overridden to 1 by setting random_state. Use no seed for parallelism.

c:\Users\Patricio\miniconda3\envs\tp3_env\Lib\site-packages\scipy\sparse\_index.py:143: SparseEfficiencyWarning:
Changing the sparsity structure of a csr_matrix is expensive. lil_matrix is more efficient.

c:\Users\Patricio\miniconda3\envs\tp3_env\Lib\site-packages\umap\umap_.py:1952: UserWarning:
n_jobs value 1 overridden to 1 by setting random_state. Use no seed for parallelism.

c:\Users\Patricio\miniconda3\envs\tp3_env\Lib\site-packages\scipy\sparse\_index.py:143: SparseEfficiencyWarning:
Changing the sparsity structure of a csr_matrix is expensive. lil_matrix is more efficient.

```

```

In [17]: X = sbert_embeddings
y = df['target'].astype(int).values

X_train, X_val, y_train, y_val = train_test_split(
    X, y, stratify=y, test_size=0.2, random_state=SEED
)

original_dims = sbert_embeddings.shape[1]
y_val_pred = train_val_knn(X_train, X_val, y_train, y_val)
score = f1_score(y_val, y_val_pred)
all_results[f'SBERT {original_dims}'] = score

dim_reduc = UMAP(n_components=100, random_state=SEED)
X_train_reduced = dim_reduc.fit_transform(X_train)
X_val_reduced = dim_reduc.transform(X_val)
y_val_pred = train_val_knn(X_train_reduced, X_val_reduced, y_train, y_val)
score = f1_score(y_val, y_val_pred)
all_results['SBERT 100'] = score

dim_reduc = UMAP(n_components=50, random_state=SEED)
X_train_reduced = dim_reduc.fit_transform(X_train)
X_val_reduced = dim_reduc.transform(X_val)
y_val_pred = train_val_knn(X_train_reduced, X_val_reduced, y_train, y_val)
score = f1_score(y_val, y_val_pred)
all_results['SBERT 50'] = score

```

```
c:\Users\Patricio\miniconda3\envs\tp3_env\Lib\site-packages\umap\umap_.py:1952: UserWarning:  
n_jobs value 1 overridden to 1 by setting random_state. Use no seed for parallelism.  
c:\Users\Patricio\miniconda3\envs\tp3_env\Lib\site-packages\scipy\sparse\_index.py:143: SparseEfficiencyWarning:  
Changing the sparsity structure of a csr_matrix is expensive. lil_matrix is more efficient.  
c:\Users\Patricio\miniconda3\envs\tp3_env\Lib\site-packages\umap\umap_.py:1952: UserWarning:  
n_jobs value 1 overridden to 1 by setting random_state. Use no seed for parallelism.  
c:\Users\Patricio\miniconda3\envs\tp3_env\Lib\site-packages\scipy\sparse\_index.py:143: SparseEfficiencyWarning:  
Changing the sparsity structure of a csr_matrix is expensive. lil_matrix is more efficient.
```

```
In [18]: print("RESULTADOS:")  
  
print("\nMODELO\t\tPUNTAJE F1")  
for key, val in all_results.items():  
    print(f"{key}\t{val}")  
  
max_value = max(all_results.values())  
max_key = max(all_results, key=all_results.get)  
  
print(f"\nEl mejor resultado fue de {max_key} con un puntaje F1 en validación de {max_value:.4f}")
```

RESULTADOS:

MODELO	PUNTAJE F1
TF-IDF 100	0.5929283771532184
TF-IDF 500	0.7193877551020408
TF-IDF 1000	0.7338371116708649
WORD2VEC 300	0.723404255319149
WORD2VEC 100	0.7125307125307125
WORD2VEC 50	0.7135922330097088
SBERT 384	0.795088257866462
SBERT 100	0.7754172989377845
SBERT 50	0.7759146341463414

El mejor resultado fue de SBERT 384 con un puntaje F1 en validación de 0.7951

Con estos resultados podemos sacar conclusiones interesantes.

Para empezar, en respuesta a la pregunta de la consigna, sí es posible predecir el target con estos embeddings.

Es más, vemos que el mejor puntaje es casi 0.8, el mismo puntaje solicitado para los modelos de la parte 3 ¡y solo utilizando una feature! (embedding SBERT).

Podemos observar que todos los resultados son considerablemente similares.

Por un lado podemos observar que se pueden obtener resultados decentes utilizando únicamente TF-IDF, que es una técnica mucho más sencilla que los otros dos modelos utilizados. Dependiendo del uso podría ser conveniente sacrificar un poco de precisión del modelo a cambio de velocidad.

También es interesante ver cómo reducir las dimensiones en Word2Vec y en SBERT no comprometen demasiado la precisión del modelo. De hecho, es notable que la diferencia de 100 a 50 es mucho menor que de 300 a 100. Mi conclusión es que esto puede deberse a que la cantidad de información contenida con 50 dimensiones puede ser muy similar a la que se puede conseguir con 100, pero con las 300 se debe agregar información útil para discernir con más precisión casos borde.

Por último podemos notar que los mejores resultados se obtienen con SBERT. Esto es esperable porque está ideado justamente para poder representar de manera inteligente oraciones, que es justo lo que necesitamos para este problema