Anexo 1

Proyecto 1: Tokenization, Stimming and Lemmatization

Mg. Luis Felipe Bustamante Narváez

Tokenization

In [...
```
# Instalamos skitlearn
!pip install -U scikit-learn
```

In [...
```
import pandas as pd
```

In [...
```
# Creamos el dataframe para mostrar la base de datos
df = pd.read_csv('df_total.csv', encoding='UTF-8')
```

In [...
```
df
```

In [...
```
#Mostramos la estructura de los datos
df['news'][2]
```

In [...
```
# Separamos los datos en variable independiente y variable dependiente
# variable independiente
X = df['news']
#variable dependiente ( A predecir)
Y = df['Type']
```

In [...
```
#Mostramos la variable dependiente
df['Type'].value_counts()
```

In [...
```
from sklearn.model_selection import train_test_split
```

In [...
```
# Asignamos los datos de entrenamiento (80%) y prueba (20%)
X_train, X_test, Y_train, Y_test = train_test_split(X, Y, test_size=0.2)
```

In [...
```
#Mostramos los datos de entrenamiento  (1217datos*0.8 = 973)
X_train
```

In [...
```
#Mostramos los datos de prueba (1217datos*0.2 = 244)
X_test
```

In [...
```
#Mostramos los datos a predecir de entrenamiento  (1217datos*0.8 = 973)
Y_train
```

In [...
```
#Mostramos los datos a predecir de prueba  (1217datos*0.2 = 244)
Y_test
```

In [...
```
# Vectorizamos los datos
from sklearn.feature_extraction.text import CountVectorizer
vectorizer = CountVectorizer()
```

In [...
```
# vectorizamos los datos de entramiento
X_train_transformed = vectorizer.fit_transform(X_train)
# vectorizamos los datos de prueba
X_test_transformed = vectorizer.transform(X_test)
```

In [...
```
#Mostramos los datos transformados entrenamiento (de forma nativa)
X_train_transformed
```

In [...
```
#Mostramos los datos transformados entrenamiento
X_train_transformed_dense = X_train_transformed.toarray()
X_train_transformed_dense
```

In [...
```
#Mostramos los datos transformados prueba (de forma nativa)
X_test_transformed
```

In [...
```
#Mostramos los datos transformados prueba
X_test_transformed_dense = X_test_transformed.toarray()
X_test_transformed_dense
```

In [...
```
from sklearn.naive_bayes import MultinomialNB
from sklearn import metrics
```

In [...
```
# Creamos la variable para el modelo
model = MultinomialNB()
# Entrenamos el modelo con los datos de entrenamiento vectorizados y los valores reales dependientes
model.fit(X_train_transformed, Y_train)
#Creamos la predicción con los datos de prueba vectorizados
Y_predict = model.predict(X_test_transformed)
#Revisamos el porcentaje de predicción
accuracy = metrics.accuracy_score(Y_test, Y_predict)
print(accuracy)
```

Stimming

In [...
```
import nltk
from nltk.tokenize import word_tokenize
from nltk.corpus import stopwords
from nltk.stem import SnowballStemmer
```

In [...
```
nltk.download('punkt')
nltk.download('stopwords')
```

In [...
```
stemmer = SnowballStemmer('spanish')
```

In [...
```
# Función para tokenizar y stimmar
def tokenize_and_stem(text):
    tokens = word_tokenize(text.lower())
    stems = [stemmer.stem(token) for token in tokens if token.isalpha()]
    return ' '.join(stems)

# Otra manera de mostrar esta función
"""
def tokenize_and_stem(text):
    # Tokenizar el texto y convertirlo a minúsculas
    tokens = word_tokenize(text.lower())

    # Aplicar stemming con un bucle for
    stems = []
    for token in tokens:
        if token.isalpha():  # Filtrar solo palabras (sin números ni símbolos)
            stem = stemmer.stem(token)  # Aplicar stemming
            stems.append(stem)  # Agregar a la lista

    # Unir los stems en un solo string y retornarlo
    return ' '.join(stems)
"""
```

In [...
```
# Agregamos una nueva columna al dataframe para mostrar los tokens
df['news_stemmer'] = df['news'].apply(tokenize_and_stem)
```

In [...
```
df
```

In [...
```
# Separamos los datos en variables de entrada y etiquetas
Xs = df['news_stemmer']
Ys = df['Type']
# Creamos el entrenamiento
Xs_train, Xs_test, Ys_train, Ys_test = train_test_split(Xs, Ys, test_size=0.2)
vectorizerS = CountVectorizer()
# Transformamos los documentos en vectores
Xs_train_transformed = vectorizerS.fit_transform(Xs_train)
Xs_test_transformed = vectorizerS.transform(Xs_test)
# Creamos el modelo
modelS = MultinomialNB()
# Entrenamos el modelo
modelS.fit(Xs_train_transformed, Ys_train)
# Medimos el rendimiento del modelo
Ys_predict = modelS.predict(Xs_test_transformed)
# Mostramos su eficiencia
acurracyS = metrics.accuracy_score(Ys_test, Ys_predict)
print(acurracyS)
```

Comparamos las dimensiones del texto original y los stimming

In [...
```
origin = X_train_transformed.shape
stimming = Xs_train_transformed.shape

# Opcional --- creamos un DataFrame para mejorar la visual y repasar
dfCompare = pd.DataFrame({
    'DataSet': ['X_train_transformed', 'Xs_train_transformed'],
    'Vectores': [origin[0], stimming[0]],
    'Dimensiones': [origin[1], stimming[1]]
})
dfCompare
```

Lemmatization

In [...
```
import spacy
```

In [...
```
nlp = spacy.load('es_core_news_sm')
```

In [...
```
# función de lemmatization
def lemmatize_text(text):
    doc = nlp(text.lower())
    lemmas = [token.lemma_ for token in doc if token.is_alpha]
    return ' '.join(lemmas)
```

In [...
```
# Agregamos una nueva columna al dataframe para mostrar los lemmas
df['news_lemma'] = df['news'].apply(lemmatize_text)
```

In [...
```
df['news_lemma'][5]
```

In [...
```
# Separamos los datos en variables de entrada y etiquetas
Xl = df['news_lemma']
Yl = df['Type']
# Creamos el entrenamiento
Xl_train, Xl_test, Yl_train, Yl_test = train_test_split(Xl, Yl, test_size=0.2)
vectorizerL = CountVectorizer()
```

```
# Transformamos los documentos en vectores
Xl_train_transformed = vectorizerL.fit_transform(Xl_train)
Xl_test_transformed = vectorizerL.transform(Xl_test)
# Creamos el modelo
modelL = MultinomialNB()
# Entrenamos el modelo
modelL.fit(Xl_train_transformed, Yl_train)
# Medimos el rendimiento del modelo
Yl_predict = modelL.predict(Xl_test_transformed)
# Mostramos su eficiencia
acurracyL = metrics.accuracy_score(Yl_test, Yl_predict)
print(acurracyL)
```

Comparamos las dimensiones del texto original y los stimming

In […

```
origin = X_train_transformed.shape
stimming = Xs_train_transformed.shape
lemmatizing = Xl_train_transformed.shape

# Opcional --- creamos un DataFrame para mejorar la visual y repasar
dfCompare = pd.DataFrame({
    'DataSet': ['X_train_transformed', 'Xs_train_transformed', 'Xl_train_transformed'],
    'Vectores': [origin[0], stimming[0], lemmatizing[0]],
    'Dimensiones': [origin[1], stimming[1], lemmatizing[1]]
})
dfCompare
```

Conclusiones

Se puede tomar la decisión de seleccionar el mejor modelo, aunque los datos son pocos, se puede observar una buena predicción, aunque en el texto se ve mejor la lemmatización, este requiere un gasto computacional más alto, que no se ve reflejado, para este ejemplo, en la eficacia del modelo.

Mg. Luis Felipe Bustamante Narváez

Anexo 2

Proyecto 2: Aplicación de TF-IDF

Mg. Luis Felipe Bustamante Narváez

Importamos los datos para su manipulación

In [...
```
import pandas as pd
df = pd.read_csv('movie_metadata.csv', encoding='UTF-8')
df.head()
```

Out[...

	color	director_name	num_critic_for_reviews	duration	director_facebook_likes	actor_3_facebook_likes	actor_2_name	actor_1_facebook_likes	gross	genres	...	num_user_for_revi
0	Color	James Cameron	723.0	178.0	0.0	855.0	Joel David Moore	1000.0	760505847.0	Action|Adventure|Fantasy|Sci-Fi	...	30
1	Color	Gore Verbinski	302.0	169.0	563.0	1000.0	Orlando Bloom	40000.0	309404152.0	Action|Adventure|Fantasy	...	12
2	Color	Sam Mendes	602.0	148.0	0.0	161.0	Rory Kinnear	11000.0	200074175.0	Action|Adventure|Thriller	...	9
3	Color	Christopher Nolan	813.0	164.0	22000.0	23000.0	Christian Bale	27000.0	448130642.0	Action|Thriller	...	27
4	NaN	Doug Walker	NaN	NaN	131.0	NaN	Rob Walker	131.0	NaN	Documentary	...	N

5 rows × 28 columns

In [...
```
# añadimos la columna id al principio de la tabla
df.insert(0, 'id', range(1, len(df)+ 1))
df
```

In [...
```
# Mostramos las columnas que no se ven para poder seleccionarlas
df.iloc[:,10:20].head()
```

In [...
```
# De acuerdo a esto, necesitamos las columnas id, genres, movie_title y plot_keywords
#verificamos su ubicación
df.columns
```

In [...
```
# Necesitamos las columnas: 0, 10,12, 17
cols_mantener = [0, 10, 12, 17]
df_new = df.iloc[:,cols_mantener]
df_new
```

In [...
```
# cambiamos los nombres de las columnas para poder utilizarlos de mejor manera
df_new = df_new.rename(columns={df_new.columns[1]:'genero',
                                df_new.columns[2]:'titulo',
                                df_new.columns[3]:'plot_keywords'})
df_new
```

In [...
```
# necesitamos reordenar las columnas para el manejo de los datos
col_mover = df_new.pop('titulo')
df_new.insert(1, 'titulo', col_mover)
df_new
```

In [...
```
# eliminamos los espacios en blanco al principio y al final de cada texto
for i in range(1, len(df_new.columns)):
    df_new.iloc[:,i] = df_new.iloc[:,i].str.strip()
df_new
```

In [...
```
# reemplazamos el caracter | por espacio vacío
for i in range(2, len(df_new.columns)):
    df_new.iloc[:,i] = df_new.iloc[:,i].str.replace('|', ' ')
df_new.head()
```

Out[…

	id	titulo	genero	plot_keywords	texto
0	1	Avatar	Action Adventure Fantasy Sci-Fi	avatar future marine native paraplegic	Action Adventure Fantasy Sci-Fi avatar future ...
1	2	Pirates of the Caribbean: At World's End	Action Adventure Fantasy	goddess marriage ceremony marriage proposal pi...	Action Adventure Fantasy goddess marriage cere...
2	3	Spectre	Action Adventure Thriller	bomb espionage sequel spy terrorist	Action Adventure Thriller bomb espionage seque...
3	4	The Dark Knight Rises	Action Thriller	deception imprisonment lawlessness police offi...	Action Thriller deception imprisonment lawless...
4	5	Star Wars: Episode VII - The Force Awakens	Documentary	NaN	Documentary nan

In […

```
# Unimos las columnas genero y plot_keywords
df_new['texto'] = df_new[['genero', 'plot_keywords']].apply(lambda row: ' '.join(row.values.astype(str)), axis=1)
df_new
```

In […

```
# observamos con un ejemplo como quedó la columna texto
df_new['texto'].iloc[0]
```

TF-IDF

In […

```
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.metrics.pairwise import cosine_similarity, euclidean_distances
```

In […

```
# vectorizamos poniendo un número máximo de tokens, entre más palabras haya, debe ser mayor este dato
tfIdf = TfidfVectorizer(max_features=3000)
```

In […

```
# Matriz de vectores (total de registros, total de dimensiones creadas)
X = tfIdf.fit_transform(df_new['texto'])
X
```

In […

```
# Creamos una serie para mapear las etiquetas de los nombres de las películas
movies = pd.Series(df_new.index, index=df_new['titulo'])
movies
```

In […

```
# Podemos buscar el índice de cada película
indice = movies['A Plague So Pleasant']
indice
```

In […

```
# consultamos
consulta = X[indice]
consulta
```

In […

```
# Observamos el vector
consulta2 = consulta.toarray()
consulta2
```

Out[…

```
array([[0., 0., 0., ..., 0., 0., 0.]], shape=(1, 3000))
```

In […

```
# observemos las columnas del vector donde no hay datos en cero, donde hay tokens
print(consulta)
```

In […

```
# calculamos la similitud de los vectores (el valor de prueba con el resto de los datos)
simil = cosine_similarity(consulta2, X)
simil
```

Out[…

```
array([[0.        , 0.        , 0.05807863, ..., 1.        , 0.64349593,
        0.        ]], shape=(1, 5043))
```

In […

```
# Lo anterior muestra el arreglo de similitudes, veamos un ejemplo de similitud entre
# la película 0 y nuestra consulta, y luego con la película 2 y nuestra consulta
comparar = 1763
a = simil[0][comparar]  #simil[consulta (siempre es 0), peli a comparar]
print(f'''La similitud entre la película
"{df_new.iloc[comparar, 1]}" y nuestra película consultada,
"{df_new.iloc[indice, 1]}", es de {a}''')
```

```
La similitud entre la película
"Identity" y nuestra película consultada,
"A Plague So Pleasant", es de 0.039206668970362867
```

In […

```
# calcular la similitud de manera más profesional
simil2 = simil.flatten()
```

In […

```
# ejemplo
comparar = 1763
```

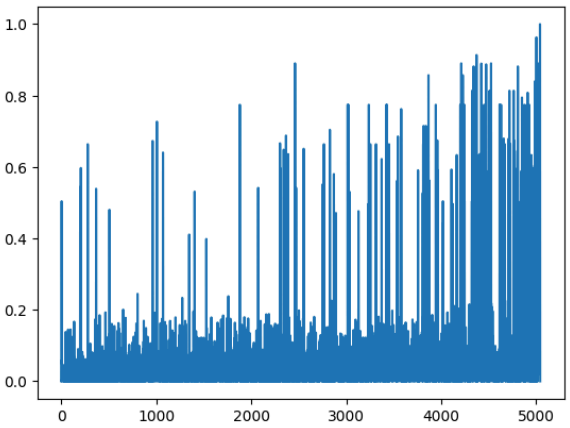
```
b = simil2[comparar]  #simil[consulta (siempre es 0), peli a comparar]
print(f'''La similitud entre la película
"{df_new.iloc[comparar, 1]}" y nuestra película consultada,
"{df_new.iloc[indice, 1]}", es de {b}''')
# Nótese que de esta manera solo se pone el valor de la película a comparar
```

Gráficos de similitud

In [...
```
!pip install matplotlib
```

In [...
```
import matplotlib.pyplot as plt
```

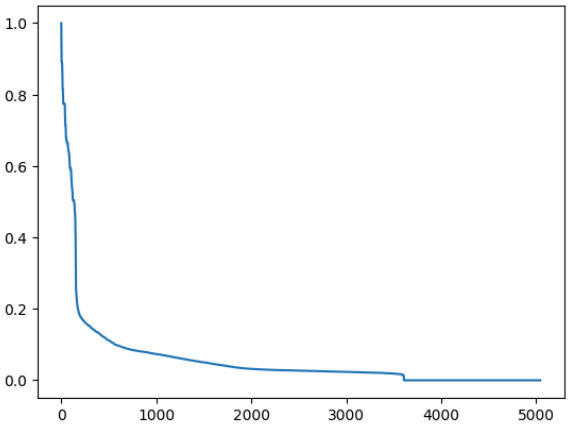
In [...
```
# Recordemos que usamos simil2 ya que esta está vectorizada con el flatten
plt.plot(simil2)
plt.show()
```



In [...
```
# ordenemos la similitud
similOrden = (-simil2).argsort()
similOrden
```

In [...
```
# graficamos ordenadamente de mayor a menor similitud
plt.plot(simil2[similOrden])
plt.show()

#Esto nos muestra una gráfia logarítmica generada por IDF
```



Recomendación de películas

In [...
```
#Ordenamos la similitud omitiendo la película de la consulta
#creando un top 10 de recomendaciones
recomendacion = (-simil2).argsort()[1:11]
```

In [...
```
# Creamos la lista del top10 de recomendados
df_new['titulo'].iloc[recomendacion]
```

Out[…
```
5002                 Run, Hide, Die
4372    A Beginner's Guide to Snuff
4211                      Antibirth
4525          The Vatican Exorcisms
2461                    Restoration
5028                    Tin Can Man
4422                      Unsullied
4473                  Forget Me Not
4341     The Curse of Downers Grove
4807                         Fabled
Name: titulo, dtype: object
```

Conclusiones

Al consultar una película cualquiera, como en el ejemplo, "A Plague So Pleasant", de acuerdo con el género y las plot_keywords de la tabla, podemos recomendar un top 10 de películas que tienen similitud con esta, algo básico, pero similar a lo que usan las aplicaciones de streaming como NetFlix, Disney, entre otras.

Mg. Luis Felipe Bustamante Narváez

Anexo 3

Proyecto 3: Aplicación de Words Embedding

Mg. Luis Felipe Bustamante Narváez

Analogías

In [...]
```
!pip install gensim
```

In [...]
```
import gensim
```

In [...]
```
#Cargamos el archivo de la db en un muestreo de vectores
vectores = gensim.models.KeyedVectors.load_word2vec_format('SBW-vectors-300-min5.txt')
```

In [...]
```
# Definimos la función que analizará la similitud
def analogics(v1, v2, v3):
    simil = vectores.most_similar(positive=[v1, v3], negative=[v2]) #buscamos el negativo
    print(f'{v1} es a {v2}, como {simil[0][0]} es a {v3}')
```

In [...]
```
# Llamamos la función con un ejemplo
analogics('rey', 'hombre', 'mujer')
```

```
rey es a hombre, como reina es a mujer
```

In [...]
```
analogics('Colombia', 'colombiano', 'venezolano')
```

```
Colombia es a colombiano, como Venezuela es a venezolano
```

In [...]
```
analogics('pan', 'trigo', 'leche')
```

```
pan es a trigo, como yogur es a leche
```

In [...]
```
analogics('vaca', 'leche', 'huevo')
```

```
vaca es a leche, como gallina es a huevo
```

In [...]
```
analogics('lápiz', 'borrador', 'corrector')
```

```
lápiz es a borrador, como pincel es a corrector
```

In [...]
```
analogics('chimenea', 'humo', 'agua')
```

```
chimenea es a humo, como desagüe es a agua
```

In [...]
```
# Obsevemos que no siempre es preciso
analogics('carne', 'carnívoro', 'vegetariano')
```

```
carne es a carnívoro, como carnes es a vegetariano
```

Valores cercanos

In [...
```python
#definimos la función de los cercanos
def cercanos(v):
    vecinos = vectores.most_similar(positive=[v])
    print(f'Vecinos de {v}:\n')
    for word, score in vecinos:
        print(f'{word}')
```

In [...
```python
cercanos('rey')
```

```
Vecinos de rey:

monarca
reyes
reino
príncipe
reina
Harthacnut
Ragnald
Sverkersson
regente
Hardeknut
```

In [...
```python
cercanos('ves')
```

```
Vecinos de ves:

sabes
piensas
dices
tú
verás
vas
sientes
estás
tienes
crees
```

In [...
```python
cercanos('vez')
```

```
Vecinos de vez:

ocasión
cuando
nuevamente
ya
tamporada
rememoraría
luego
Súperfinal
que
cosa
```

In [...
```python
cercanos('vegetariano')
```

```
Vecinos de vegetariano:

vegetariana
vegano
vegetarianismo
vegetarianos
vegana
abstemio
veganismo
crudiveganismo
veganos
crudivorismo
```

Conclusiones

Con la base de datos consumida, se puede observar el comportamiento de las words embedding, cómo hallar similitudes entre palabras asociadas a un contexto y cómo visualizar las palabras que constituyen una clasificación asociada a una palabra específica.

Mg. Luis Felipe Bustamante Narváez

Anexo 4

Proyecto 4: Creación de Words Embedding Nivel 1

Mg. Luis Felipe Bustamante Narváez

Word2Vec

Es un modelo que se utiliza para aprender representaciones vectoriales de palabras. Estas representaciones pueden capturar muchas propiedades lingüisticas de las palabras, como su significado semántico, gramatical y hasta contextual.

Para este primer ejemplo, usaremos un texto corto llamado mundiales que tiene alrededor de 50.000 caracteres.

Librerías

In [...
```
!pip install pypdf2
```

In [...
```
import string
from gensim.models import Word2Vec
import PyPDF2
```

Cargamos el documento

In [...
```
with open('Entrenamiento_Word2Vec/mundiales.txt',
          'r', encoding='utf-8') as file:
    documento = file.read()
```

In [...
```
documento
```

In [...
```
len(documento)
```

Out[...
```
48155
```

Procesamiento de datos

El objetivo del procesamiento es convertir el documento en una lista de frases, y cada frase en una lista de palabras, eliminando signos de puntuación y convirtiendo todo a minúsculas.

In [...
```
# Dividimos el documento en frases usando la coma como separador
frases = documento.split(',')
len(frases)
```

Out[...
```
533
```

In [...
```
# Mostramos un ejemplo
frases[0]
```

Out[…] 'Capítulo 1: Historia del Mundial de Fútbol\nLos primeros pasos del Mundial\nLa historia del Mundial de Fútbol se remonta a principios del siglo XX'

In […]
```
# Mostramos un ejemplo
frases[500]
```

Out[…] ' **Adidas**'

In […]
```
# Limpiamos las frases
frases_limpias = []
for frase in frases:
    #Eliminamos la puntuación y dividimos por espacios
    tokens = frase.translate(str.maketrans('','',string.punctuation)).split()
    #print(tokens)  #para mostrar qué ha hecho hasta aquí
    #Convertimos a minúsculas
    tokens = [word.lower() for word in tokens]
    #print(tokens)  #para mostrar qué ha hecho hasta aquí
    if tokens:
        frases_limpias.append(tokens)
```

In […]
```
# Mostramos los resultados
frases_limpias[500]
```

Out[…] ['adidas']

Entrenamiento del modelo Word2Vec

In […]
```
model = Word2Vec(sentences=frases_limpias,
                 vector_size=500,
                 window=5,
                 min_count=1,
                 workers=4)
```

Explicación:

- sentences: Es la lista de palabras que vamos a vectorizar
- vector_size: Es el tamaño de dimensiones que le daremos al vector
- window: Son la cantidad de palabras por encima y por debajo que le darán contexto
- min_count: La aparición mínima de una palabra para tenerla en cuenta en el entrenamiento
- workers: Cantidad de núcleo de procesador que vamos a invertir en el entrenamiento

Pruebas

In […]
```
# Verificamos el vector para alguna palabra
vector = model.wv['mundial']
vector
```

In […]
```
# Mostramos las palabras cercanas
palabras_cercanas = model.wv.most_similar('jugador', topn=10)
palabras_cercanas
# Es probable que la similitud falle por tener tan pocas palabras en el texto
```

Out[…
```
[('historia', 0.9089663624763489),
 ('su', 0.9083009362220764),
 ('con', 0.9079578518867493),
 ('y', 0.9078695774078369),
 ('brasil', 0.9078560471534729),
 ('en', 0.9078254699707031),
 ('más', 0.9077353477478027),
 ('del', 0.9077184200286865),
 ('francia', 0.9076839685440063),
 ('juego', 0.9075537919998169)]
```

Guardar modelo

In […
```
model.save('Entrenamiento_Word2Vec/mundiales.model')
```

Cargar el modelo

In […
```
modelo_cargado = Word2Vec.load('Entrenamiento_Word2Vec/mundiales.model')
```

In […
```
# Probamos con el modelo caragado
palabras_cercanas2 = modelo_cargado.wv.most_similar('jugador', topn=10)
palabras_cercanas2
```

Out[…
```
[('historia', 0.9089663624763489),
 ('su', 0.9083009362220764),
 ('con', 0.9079578518867493),
 ('y', 0.9078695774078369),
 ('brasil', 0.9078560471534729),
 ('en', 0.9078254699707031),
 ('más', 0.9077353477478027),
 ('del', 0.9077184200286865),
 ('francia', 0.9076839685440063),
 ('juego', 0.9075537919998169)]
```

Guardar Embedido

Existen dos maneras, usando .txt sin binarios, y usando .bin con binarios.

In […
```
model.wv.save_word2vec_format('Entrenamiento_Word2Vec/munidiales_emb.txt', binary=False)
model.wv.save_word2vec_format('Entrenamiento_Word2Vec/munidiales_emb.bin', binary=True)
```

Cargar Embedidos

Si se carga el .txt, se usa sin binarios; si se carga el .bin, se usa con binarios

In […
```
from gensim.models import KeyedVectors
embedding_cargado_txt = KeyedVectors.load_word2vec_format(
    'Entrenamiento_Word2Vec/munidiales_emb.txt', binary=False)
```

In […
```
embedding_cargado_bin = KeyedVectors.load_word2vec_format(
    'Entrenamiento_Word2Vec/munidiales_emb.bin', binary=True)
```

In […
```
# Probamos
```

```
embedding_cargado_txt
```

Out[…] <gensim.models.keyedvectors.KeyedVectors at 0x21302849a30>

In […]
```
# Probamos
embedding_cargado_bin
```

Out[…] <gensim.models.keyedvectors.KeyedVectors at 0x213062ccaa0>

Analogías

In […]
```
def analogics(v1, v2, v3):
    simil = embedding_cargado_bin.most_similar(positive=[v1,v3],
                                               negative=[v2]
                                              )
    print(f'{v1} es a {v2}, como {simil[0][0]} es a {v3}')
```

In […]
```
analogics('jugador', 'fútbol', 'historia')
```

```
jugador es a fútbol, como brasil es a historia
```

Conclusiones

El texto mundiales tiene cerca de 50.000 caracteres, lo que implica una base de datos pequeña para entrenar un modelo. De cierta forma, el modelo se ajusta en algunos casos puntuales, pero suele mostrar demasiadas stopwords, que tendríamos que manipular para mejorar la predicción de analogías. Veremos con un texto más grande, como se generaría la predicción, por ejemplo el libro "Cien años de soledad" de Gabriel García Márquez.

Mg. Luis Felipe Bustamante Narváez

Anexo 5

Proyecto 5: Creación de Words Embedding Nivel 2

Mg. Luis Felipe Bustamante Narváez

Word2Vec

Es un modelo que se utiliza para aprender representaciones vectoriales de palabras. Estas representaciones pueden capturar muchas propiedades lingüísticas de las palabras, como su significado semántico, gramatical y hasta contextual.

Para este segundo ejemplo, usaremos un texto más extenso llamado "Cien años de soledad" del escritor colombiano Gabriel García Márquez, que tiene alrededor de 139.318 palabras y 823735 caracteres.

Librerías

In [...
```
!pip install pypdf2
```

```
Requirement already satisfied: pypdf2 in c:\users\luis_\anaconda3\envs\notebook\lib\site-packages (3.0.1)
```

In [...
```
import string
from gensim.models import Word2Vec
import PyPDF2
```

Cargamos el documento

In [...
```
def extraer_texto_desde_pdf(ruta_archivo):
    with open(ruta_archivo, 'rb') as archivo:
        lector = PyPDF2.PdfReader(archivo)
        texto = ''
        for pagina in range(len(lector.pages)):
            texto += lector.pages[pagina].extract_text()
        return texto
```

In [...
```
documento = extraer_texto_desde_pdf('Entrenamiento_Word2Vec/cien_anos_de_soledad.pdf')
```

In [...
```
len(documento)
```

Out[...
```
823735
```

Procesamiento de datos

El objetivo del procesamiento es convertir el documento en una lista de frases, y cada frase en una lista de palabras, eliminando signos de puntuación y convirtiendo todo a minúsculas.

In [...
```
# Dividimos el documento en frases usando la coma como separador
frases = documento.split(',')
len(frases)
```

Out[...
```
8854
```

In [...
```
# Mostramos un ejemplo
frases[500]
```

Out[...
```
' las huestes de su padre tenían un aspecto de náufragos sin escapatoria'
```

In [...]
```
# Mostramos un ejemplo
frases[3000]
```

Out[...]
```
' sino porque lo único que me faltó fue darte de mamar.» Aureliano escapaba al alba y regresaba a la madrugada siguiente'
```

In [...]
```
# Limpiamos las frases
frases_limpias = []
for frase in frases:
    #Eliminamos la puntuación y dividimos por espacios
    tokens = frase.translate(str.maketrans('','',string.punctuation)).split()
    #print(tokens)  #para mostrar qué ha hecho hasta aquí
    #Convertimos a minúsculas
    tokens = [word.lower() for word in tokens]
    #print(tokens)  #para mostrar qué ha hecho hasta aquí
    if tokens:
        frases_limpias.append(tokens)
```

In [...]
```
# Mostramos los resultados
frases_limpias[500]
```

Out[...]
```
['las',
 'huestes',
 'de',
 'su',
 'padre',
 'tenían',
 'un',
 'aspecto',
 'de',
 'náufragos',
 'sin',
 'escapatoria']
```

Entrenamiento del modelo Word2Vec

In [...]
```
model = Word2Vec(sentences=frases_limpias,
                 vector_size=500,
                 window=5,
                 min_count=1,
                 workers=4)
```

Explicación:

- sentences: Es la lista de palabras que vamos a vectorizar
- vector_size: Es el tamaño de dimensiones que le daremos al vector
- window: Son la cantidad de palabras por encima y por debajo que le darán contexto
- min_count: La aparición mínima de una palabra para tenerla en cuenta en el entrenamiento
- workers: Cantidad de núcleo de procesador que vamos a invertir en el entrenamiento

Pruebas

In [...]
```
# Verificamos el vector para alguna palabra
vector = model.wv['padre']
vector
```

In [...]
```
# Mostramos las palabras cercanas
palabras_cercanas = model.wv.most_similar('buendía', topn=10)
palabras_cercanas
# Es probable que la similitud falle por tener tan pocas palabras en el texto
```

Out[…
```
[('segundo', 0.9980278611183167),
 ('josé', 0.9935610890388489),
 ('buendia', 0.9925903081893921),
 ('tío', 0.9853704571723938),
 ('aureliano', 0.9842246770858765),
 ('promovió', 0.9841890335083008),
 ('pequeño', 0.9830081462860107),
 ('participó', 0.9812502264976501),
 ('informaciones', 0.9800763726234436),
 ('arcadio', 0.9795325398445129)]
```

Guardar modelo

In […
```
model.save('Entrenamiento_Word2Vec/cien_anos_de_soledad.model')
```

Cargar el modelo

In […
```
modelo_cargado = Word2Vec.load('Entrenamiento_Word2Vec/cien_anos_de_soledad.model')
```

In […
```
# Probamos con el modelo caragado
palabras_cercanas2 = modelo_cargado.wv.most_similar('buendía', topn=10)
palabras_cercanas2
```

Out[…
```
[('segundo', 0.9980278611183167),
 ('josé', 0.9935610890388489),
 ('buendia', 0.9925903081893921),
 ('tío', 0.9853704571723938),
 ('aureliano', 0.9842246770858765),
 ('promovió', 0.9841890335083008),
 ('pequeño', 0.9830081462860107),
 ('participó', 0.9812502264976501),
 ('informaciones', 0.9800763726234436),
 ('arcadio', 0.9795325398445129)]
```

Guardar Embedido

Existen dos maneras, usando .txt sin binarios, y usando .bin con binarios.

In […
```
model.wv.save_word2vec_format('Entrenamiento_Word2Vec/cien_anos_de_soledad_emb.txt', binary=False)
model.wv.save_word2vec_format('Entrenamiento_Word2Vec/cien_anos_de_soledad_emb.bin', binary=True)
```

Cargar Embedidos

Si se carga el .txt, se usa sin binarios; si se carga el .bin, se usa con binarios

In […
```
from gensim.models import KeyedVectors
embedding_cargado_txt = KeyedVectors.load_word2vec_format(
    'Entrenamiento_Word2Vec/cien_anos_de_soledad_emb.txt', binary=False)
```

In […
```
embedding_cargado_bin = KeyedVectors.load_word2vec_format(
    'Entrenamiento_Word2Vec/cien_anos_de_soledad_emb.bin', binary=True)
```

In […
```
# Probamos
embedding_cargado_txt
```

Out[…
```
<gensim.models.keyedvectors.KeyedVectors at 0x1116c4d67b0>
```

In […
```
# Probamos
embedding_cargado_bin
```

Out[…
```
<gensim.models.keyedvectors.KeyedVectors at 0x1116c43a390>
```

Analogías

In [...
```
def analogics(v1, v2, v3):
    simil = embedding_cargado_bin.most_similar(positive=[v1,v3],
                                               negative=[v2]
                                              )
    print(f'{v1} es a {v2}, como {simil[0][0]} es a {v3}')
```

In [...
```
analogics('aureliano', 'buendía', 'iguarán')
```

```
aureliano es a buendía, como luz es a iguarán
```

Conclusiones

El texto Cien años de soledad, tiene 823735 caracteres, lo que implica una base de datos más grande para entrenar un modelo, pero sigue siendo pequeña para un modelo adecuado. De cierta forma, el modelo se ajusta en algunos casos puntuales, pero suele mostrar demasiadas stopwords en las similitudes, que tendríamos que manipular para mejorar la predicción de analogías. Veremos con un paquete de textos más grandes y variados, como se generaría la predicción.

Mg. Luis Felipe Bustamante Narváez

Anexo 6

Proyecto 6: Conversión masiva de nombres de archivos

Mg. Luis Felipe Bustamante Narváez

Renombrar

Este ejercicio, permitirá cambiar el nombre de forma masiva a diferentes archivos para ejecutarlos de manera rápida en un análisis de vectorización para NPL, evitando errores por los nombres que traen estos archivos a la hora de descargarlos o de generarlos.

Librerías

In [...]
```
import os
```

Ruta de los archivos a cambiar su nombre

In [...]
```
pathOrigin = 'Entrenamiento_Word2Vec/textos'
pathFinal = 'Entrenamiento_Word2Vec/textos/textos'
```

Función para renombrar

In [...]
```
def changeNames(originPath, finalPath, prefix):
    files = os.listdir(originPath)
    for counter, fileName in enumerate(files):
        name, extension = os.path.splitext(fileName)
        newName = f'{prefix}_{counter}{extension}'
        try:
            os.rename(os.path.join(originPath, fileName), os.path.join(finalPath, newName))
            print("Cambio de nombres realizado.")
        except (err):
            print("Error al renombrar: ", err)
```

Llamamos la función

In [...]
```
prefix = 'texto'
changeNames(pathOrigin, pathFinal, prefix)
```

Conclusiones

Este código, permite renombrar listas largas de archivos para una mejor manipulación en entrenamientos de AI.

Mg. Luis Felipe Bustamante Narváez

Anexo 7

Proyecto 7: Creación de Words Embedding Nivel 3

Mg. Luis Felipe Bustamante Narváez

Word2Vec

Es un modelo que se utiliza para aprender representaciones vectoriales de palabras. Estas representaciones pueden capturar muchas propiedades lingüísticas de las palabras, como su significado semántico, gramatical y hasta contextual.

Para este tercer ejemplo, usaremos 100 textos aleatorios, desde 20.000 caracteres hasta más de 1.500.000 caracteres, los cuales suman para el entrenamiento un total de ---- caracteres.

Librerías

In [...
```
!pip install pypdf2
```

```
Requirement already satisfied: pypdf2 in c:\users\luis_\anaconda3\envs\notebook\lib\site-pack
ages (3.0.1)
```

In [...
```
!pip install tqdm
```

```
Requirement already satisfied: tqdm in c:\users\luis_\anaconda3\envs\notebook\lib\site-packag
es (4.67.1)
Requirement already satisfied: colorama in c:\users\luis_\anaconda3\envs\notebook\lib\site-pa
ckages (from tqdm) (0.4.6)
```

In [...
```
import string
from gensim.models import Word2Vec
import PyPDF2
import os
from tqdm import tqdm
```

Cargamos el documento

In [...
```
def extraer_texto_desde_pdf(ruta_archivo):
    with open(ruta_archivo, 'rb') as archivo:
        lector = PyPDF2.PdfReader(archivo)
        texto = ''
        for pagina in range(len(lector.pages)):
            texto += lector.pages[pagina].extract_text()
        return texto
```

In [...
```
ruta_carpeta = 'Entrenamiento_Word2Vec/textos'
```

Guardamos todos los textos en una lista

In [...
```
todos_los_textos = []
for archivo in tqdm(os.listdir(ruta_carpeta)):
    if archivo.endswith('.pdf'):
```

```
        ruta_completa = os.path.join(ruta_carpeta, archivo)
        try:
            documento = extraer_texto_desde_pdf(ruta_completa)
            todos_los_textos.append(documento)
        except Exception as e:
            print(f'Error al procesar {archivo}: {e}')
```

```
100%|██████████| 100/100 [12:46<00:00,  7.66s/it]
```

In [...
```
len(todos_los_textos)
```

Out[...
```
100
```

Procesamiento de datos

El objetivo del procesamiento es convertir el documento en una lista de frases, y cada frase en una lista de palabras, eliminando signos de puntuación y convirtiendo todo a minúsculas.

In [...
```
# Dividimos el documento en frases usando la coma como separador
frases_totales = []
caracteres = 0

for documento in todos_los_textos:
    caracteres = caracteres + len(documento)
    frases = documento.split(',')
    frases_totales.extend(frases)
```

In [...
```
# Mostramos el número de caracteres totales
print(f'Número de caracteres: {caracteres}')
```

```
Número de caracteres: 240745048
```

In [...
```
# Mostramos el número de oraciones totales
print(f'El número de frases totales es de: {len(frases_totales)}')
```

```
El número de frases totales es de: 1980300
```

In [...
```
# Mostramos un ejemplo
frases_totales[500]
```

Out[...
```
'\ninfluyendo en el desarrollo de la humanidad en múltiples aspectos. A medida que avanzamos en el\nconocimiento'
```

In [...
```
# Mostramos un ejemplo
frases_totales[3000]
```

Out[...
```
' nuevas perspectivas emergen'
```

In [...
```
# Limpiamos las frases
frases_limpias = []
for frase in frases_totales:
    #Eliminamos la puntuación y dividimos por espacios
    tokens = frase.translate(str.maketrans('','',string.punctuation)).split()
    #print(tokens)  #para mostrar qué ha hecho hasta aquí
    #Convertimos a minúsculas
    tokens = [word.lower() for word in tokens]
    #print(tokens)  #para mostrar qué ha hecho hasta aquí
    if tokens:
        frases_limpias.append(tokens)
```

In [...
```
# Mostramos los resultados
frases_limpias[500]
```

Out[…
```
['influyendo',
 'en',
 'el',
 'desarrollo',
 'de',
 'la',
 'humanidad',
 'en',
 'múltiples',
 'aspectos',
 'a',
 'medida',
 'que',
 'avanzamos',
 'en',
 'el',
 'conocimiento']
```

CPU disponibles en mi PC

En este apartado, observaremos la cantidad de núcleos de procesamiento tiene nuestro computador para el trabajo en NPL. Como este modelo requiere de más gasto computacional, es bueno identificar este dato, para ser eficientes en el entrenamiento, y evitar relentizar el equipo u otros procesos en paralelo.

In […
```python
def numero_de_cpus():
    return os.cpu_count()

print(f'Mi equipo tiene {numero_de_cpus()} CPU´s')
```

```
Mi equipo tiene 8 CPU´s
```

Entrenamiento del modelo Word2Vec

In […
```python
model = Word2Vec(sentences=frases_limpias,
                 vector_size=500,
                 window=5,
                 min_count=1,
                 workers=6)
```

Explicación:

- sentences: Es la lista de palabras que vamos a vectorizar
- vector_size: Es el tamaño de dimensiones que le daremos al vector
- window: Son la cantidad de palabras por encima y por debajo que le darán contexto
- min_count: La aparición mínima de una palabra para tenerla en cuenta en el entrenamiento
- workers: Cantidad de núcleo de procesador que vamos a invertir en el entrenamiento

Pruebas

In […
```python
# Verificamos el vector para alguna palabra
vector = model.wv['ciencia']
vector
```

In […
```python
# Mostramos las palabras cercanas
palabras_cercanas = model.wv.most_similar('ciencia', topn=10)
```

```
palabras_cercanas
# Es probable que la similitud falle por tener tan pocas palabras en el texto
```

Out[…
```
[('principios', 0.3798693120479584),
 ('tecnología', 0.36144816875457764),
 ('medicina', 0.2695100009441376),
 ('biografía', 0.2573043406009674),
 ('múltiples', 0.2501693665981293),
 ('múltiplesdimensiones', 0.2432234287261963),
 ('economía', 0.24100026488304138),
 ('relevantebiografía', 0.22121661901474),
 ('vidas', 0.20030616223812103),
 ('han', 0.19419683516025543)]
```

In […
```
# Mostramos las palabras cercanas
palabras_cercanas = model.wv.most_similar('sistemas', topn=10)
palabras_cercanas
# Es probable que la similitud falle por tener tan pocas palabras en el texto
```

Out[…
```
[('enseñanza', 0.8446722030639648),
 ('educación', 0.4766235649585724),
 ('crisis', 0.4423055946826935),
 ('financieras', 0.4304288625717163),
 ('cultural', 0.37052440643310547),
 ('videojuegos', 0.35479822754859924),
 ('actualidad', 0.29080677032470703),
 ('gastronomía', 0.25270408391952515),
 ('fundamentales', 0.2261238843202591),
 ('aportes', 0.2239352911710739)]
```

Guardar modelo

In […
```
model.save('Entrenamiento_Word2Vec/100textos.model')
```

Cargar el modelo

In […
```
modelo_cargado = Word2Vec.load('Entrenamiento_Word2Vec/100textos.model')
```

In […
```
# Probamos con el modelo caragado
palabras_cercanas2 = modelo_cargado.wv.most_similar('importancia', topn=10)
palabras_cercanas2
```

Out[…
```
[('emocional', 0.6839113831520081),
 ('industria', 0.29906395077705383),
 ('cultura', 0.2832416296005249),
 ('másimportantes', 0.24401310086250305),
 ('expertos', 0.2300969511270523),
 ('energía', 0.22232283651828766),
 ('géneros', 0.22123292088508606),
 ('importantes', 0.20623861253261566),
 ('nuclear', 0.19350115954875946),
 ('descubrimientos', 0.18687954545021057)]
```

Guardar Embedido

Existen dos maneras, usando .txt sin binarios, y usando .bin con binarios.

In [...
```
model.wv.save_word2vec_format('Entrenamiento_Word2Vec/100textos_emb.txt', binary=False)
model.wv.save_word2vec_format('Entrenamiento_Word2Vec/100textos_emb.bin', binary=True)
```

Cargar Embedidos

Si se carga el .txt, se usa sin binarios; si se carga el .bin, se usa con binarios

In [...
```
from gensim.models import KeyedVectors
embedding_cargado_txt = KeyedVectors.load_word2vec_format(
    'Entrenamiento_Word2Vec/100textos_emb.txt', binary=False)
```

In [...
```
embedding_cargado_bin = KeyedVectors.load_word2vec_format(
    'Entrenamiento_Word2Vec/100textos_emb.bin', binary=True)
```

In [...
```
# Probamos
embedding_cargado_txt
```

Out[...
```
<gensim.models.keyedvectors.KeyedVectors at 0x1d45dbc0a70>
```

In [...
```
# Probamos
embedding_cargado_bin
```

Out[...
```
<gensim.models.keyedvectors.KeyedVectors at 0x1d444aa0710>
```

Analogías

In [...
```
def analogics(v1, v2, v3):
    simil = embedding_cargado_bin.most_similar(positive=[v1,v3],
                                               negative=[v2]
                                              )
    print(f'{v1} es a {v2}, como {simil[0][0]} es a {v3}')
```

In [...
```
analogics('científico', 'ciencia', 'cultura')
```

```
científico es a ciencia, como pensamiento es a cultura
```

In [...
```
analogics('científico', 'ciencia', 'nuclear')
```

```
científico es a ciencia, como energía es a nuclear
```

In [...
```
analogics('científico', 'ciencia', 'imperio')
```

```
científico es a ciencia, como romano es a imperio
```

Conclusiones

Utilizando 100 textos con temáticas aleatorias, se puede observar que las predicciones en las analogías son mucho más reales; chatGPT3 utilizó en su primer entrenamiento 570G en textos, libros y artículos, nuestro entrenamiento utilizó tan solo 82.8M y aún así, encontramos mucha coherencia a la hora de probar la similitud. ¿Qué pasaría si usaramos por lo menos 1.000.000 de textos?

Mg. Luis Felipe Bustamante Narváez