Clasificador de tweets empleando una máquina de soporte vectorial

Alberto Saborido Patiño

Índice

1. Introducción **2**

2. Construcción de la base de datos y el corpus **3**

2.1. Conexión 3

2.2. Extracción de los datos 4

3. Vectorización textual y análisis Tf-idf **7**

4. LLamamos a la máquina de soporte vectorial **7**

5. Análisis de sentimiento **10**

1. Introducción

Vamos a construir un clasificador automático de textos que, tras entrenarlo, sepa discernir la profesión o tipo de cuenta de quien escribió un determinado tweet.

Para ello, construiremos una base de datos constituida por una colección de tweets de ciertas personalidades e instituciones famosas. El objetivo será emplearla para entrenar un modelo de aprendizaje estadístico, que en este caso será una máquina de soporte vectorial. Obviamente, para que el modelo planteado funcione correctamente debemos incluir en la base de datos de entrenamiento ejemplos de todos los oficios/tipos de cuenta que sea posible encontrar en la muestra de test.

Para ejemplificar el método, hemos escogido las siguientes profesiones/tipos de cuenta para formar nuestra base de datos. Para cada profesión/tipo de cuenta nos hemos valido de los tweets de dos cuentas diferentes, que han sido, respectivamente:

Polticos (para el entrenamiento) :

@GOPChairwoman | https://twitter.com/GOPChairwoman

@JoeBiden | https://twitter.com/JoeBiden

Deportistas (para el entrenamiento) :

@KingJames | https://twitter.com/KingJames

@Simone_Biles | https://twitter.com/Simone_Biles

Periodicos (para el entrenamiento) :

@nytimes | https://twitter.com/nytimes

@BBCNews | https://twitter.com/BBCNews

Por otro lado, hemos descargado los tweets de otras cuentas diferentes para formar nuestra muestra de test. Hemos decidido hacerlo de este modo para probar que el algoritmo es capaz de diferenciar el tipo de cuenta independientemente de la persona en concreto, es decir, el modelo no ha sido entrenado con ningún tweet escrito por las personas/cuentas que sometemos al test. Simplemente estas ultimas encajan en un perfil determinado debido a su naturaleza. Los usuarios que hemos escogido para realizar nuestro test son los siguientes:

Usuarios para el test :

Politica: @KamalaHarris | https://twitter.com/KamalaHarris

Deportista: @LewisHamilton | https://twitter.com/LewisHamilton

Periódico: @WSJ | https://twitter.com/WSJ

2. Construcción de la base de datos y el corpus

La red social Twitter hace posible la extracción de tweets a través de su API (*Application Programming Interfaces*) . Trás crear una aplicación de desarrollo (https://developer.twitter.com/apps), solicitamos permiso para adquirir las claves que nos dan los accesos necesarios para llevar acabo la tarea.

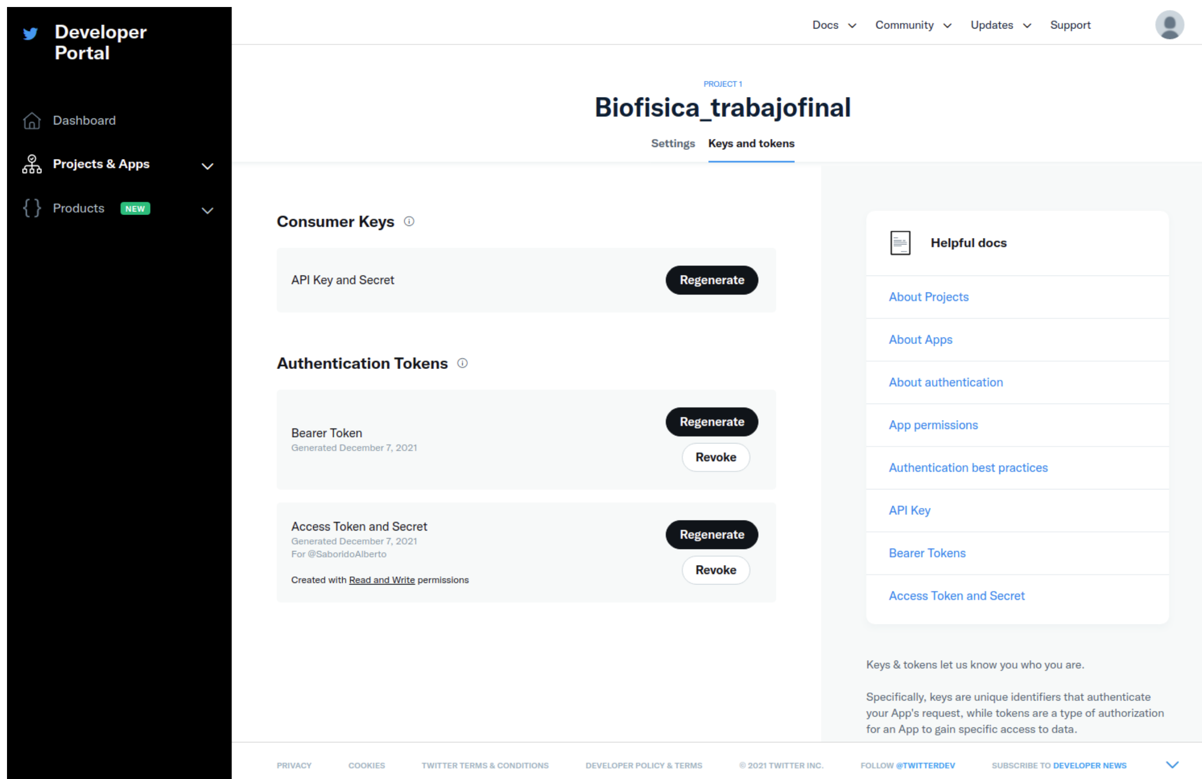

Figura 1: Captura de pantalla de la web que nos permite adquirir los permisos

Para descargar los datos nos valemos de la librería python-twitter, que nos permite conectarnos a la base de datos de Twitter a través de su API y nos proporciona diversas funcionalidades que nos serán útiles.

2.1. Conexión

Realizamos la conexión introduciendo nuestros credenciales y comprobamos que estamos conectados.

[ ]:
```
con = tw.Api(consumer_key='UaY1mg6m6O5IilZmQpAmPAaMJ',
  consumer_secret='vhnXuvQEWnvb21CY1W6a27bZFBmqfLsyFIvU8BsBCeqhXU4YVy',
  access_token_key='1338662510149431296-6GNT7JQN6ACCdm7DLxlR1eIVbQSHy3',
  access_token_secret='dfULab0chuwqRXgKNlq3ERq2Ta8lBOCPGJ9BckUYIq7h8')
```

[ ]:
```
if (con.VerifyCredentials()):
        print("Conected")
```

2.2. Extracción de los datos

Como pretendemos recuperar 6 perfiles distintos abstraeremos la funcionalidad de recuperar los tweets en un método que llamaremos crawler.

En los tweets, es muy común adjuntar contenido externo. Este contenido en Twitter viene referenciado a través de una URL. Como las URLs carecen de información, las eliminaremos. Además en Twitter son comunes ciertos símbolos que pueden influir en el análisis textual: las almohadillas (#) (Hashtags) y las arrobas (@) para mencionar otras cuentas. También eliminaremos estos símbolos. Se trata de hacer una limpieza simple pero muy efectiva para obtener solo la información textual de cada uno de los tweets. Esta limpieza la programamos en la función denominada tweetClean.

Función Crawler

Este método es el responsable de recuperar la información de los perfiles seleccionados. Debido a las limitaciones de la API de Twitter es necesario hacer un número de iteraciones acotadas. En nuestro caso planteamos 30 iteraciones de 200 tweets cada una, es decir, de cada perfil recuperaríamos, en principio, 6000 tweets. Como veremos esto termina por no ser así, muy probablemente debido a las limitaciones que impone Twitter respecto a la tasa de descarga de datos. Para cada tweet guardamos la siguiente información:

- Una etiqueta propia que fijamos nosotros (numeración).
- Autor del tweet.
- Profesión/tipo de cuenta del autor.
- Fecha de creación.
- Texto (procesado) del tweet.

Con esta información generamos diccionarios que almacenamos en un fichero con formato JSON. Tendremos así un fichero JSON por cuenta.

[ ]:
```
def tweetClean(tuit):
        tuit = re.sub('(www|https)[^\s]+', ' ', tuit)
        tuit = re.sub('#', ' ', tuit)
        tuit = re.sub('@', ' ', tuit)
        return tuit

def crawler(userAccount,job):
    timeLine=[]
    id = 0
```

```
    for i in range(0,30):
        time.sleep(1.)
        #primera iteracion, array vacio
        if not timeLine:
            rawTweets = con.GetUserTimeline(screen_name=userAccount,␣
↪count=200, trim_user=True)
            print(len(rawTweets))
            lastId = rawTweets[-1].id
        else:
            rawTweets = con.GetUserTimeline(screen_name=userAccount,␣
↪count=200, trim_user=True, max_id=lastId)
            print(len(rawTweets))
            lastId = rawTweets[-1].id


    for rawtweet in rawTweets:
        tweet = {}
        id += 1
        tweet['id'] = id
        tweet['author'] = userAccount
        tweet['date'] = rawtweet.created_at
        tweet['job'] = job
        tweet['text'] = tweetClean(rawtweet.text)

        timeLine.append(tweet)



    print(f"Se han extraído {len(timeLine)} tweets de la cuenta␣
↪{userAccount}")
    with open('timeLine_%s.json' % userAccount, mode="w",␣
↪encoding="utf8") as file:
        json.dump(timeLine,file)
```

Una vez definidas las funciones, las llamamos enviándoles los usuarios y correspondiente profesión/tipo de cuenta que nos interesan.

[ ]:
```
users = {'politics':['@GOPChairwoman','@JoeBiden','@KamalaHarris'],
        'sports':['@KingJames','@Simone_Biles','@LewisHamilton'],
        'news':['@nytimes','BBCNews','@WSJ']
        }



for x in users:
    for y in users[x]:
        crawler(y,x)
```

Leemos ahora la información textual que nos interesa para el análisis y almacenamos en

una misma lista los tweets que provengan de cuentas del mismo tipo. Es lo que se suele denominar el corpus de nuestro análisis (es decir, nos olvidamos en este momento de toda la información que hemos guardado que nos se corresponda con los tweets ya procesados). Para ello definimos la función buildCorpus.

Definimos también otra función que cree las muestras de entrenamiento y test. Para construir la Y, asignamos a cada profesión/tipo de cuenta un número entero diferente (0 para los políticos, 1 para los deportistas y 2 para los periódicos).

[ ]:
```
def buildCorpus(Users):
    corpus = []
    for user in Users:
        with open('timeLine_%s.json' % user, mode="r", encoding="utf8")␣
↪as file:
            documents = json.load(file)
        for doc in documents:
            corpus.append(doc['text'])
        file.close()

    return corpus

def trainTestSplit(corpusa,corpusb,corpusc, testa,testb,testc, ␣
↪clasea,claseb,clasec):
    x_train = corpusa + corpusb+ corpusc
    y_train = len(corpusa) * [clasea] + len(corpusb) * [claseb] +␣
↪len(corpusc) * [clasec]

    x_test = testa+testb+testc
    y_test = len(testa) * [clasea] + len(testb) * [claseb] + len(testc)␣
↪*   [clasec]
    return (x_train, y_train, x_test, y_test)
```

Llamamos a estas funciones con nuestros datos para obtener las muestras de entrenamiento y test.

[ ]:
```
politicos = buildCorpus(['@GOPChairwoman','@JoeBiden'])

deportistas = buildCorpus(['@KingJames','@Simone_Biles'])

periodicos = buildCorpus(['@nytimes','@BBCNews'])

test_pol = buildCorpus(['@KamalaHarris'])
test_dep = buildCorpus(['@LewisHamilton'])
test_per = buildCorpus(['@WSJ'])

(xtraincorpus, ytraincorpus, xtestcorpus, ytestcorpus) =␣
↪trainTestSplit(politicos,deportistas,periodicos,␣
↪test_pol,test_dep,test_per ,0,1,2)
```

3. Vectorización textual y análisis Tf-idf

En este punto tenemos todo lo necesario para inicial el análisis textual. El primer paso es vectorizar cada una de las colecciones de tweets de manera que cada elemento tenga un análisis Tf-idf.

La vectorización no es más que la conversión del texto a una representación numérica. El Tf-idf es una medida numérica que expresa cuán relevante es una palabra para un documento en una colección. Esta medida se utiliza a menudo como un factor de ponderación en la recuperación de información y la minería de texto. El valor tf-idf aumenta proporcionalmente al número de veces que una palabra aparece en el documento, pero es compensada por la frecuencia de la palabra en la colección de documentos, lo que permite manejar el hecho de que algunas palabras son generalmente más comunes que otras.

Este procesado del texto nos lo facilita la librería de scikit-learn feature_extraction.text mediante su función TfidfVectorizer (https://scikit-learn.org/stable/modules/generated/sklearn.feature_extraction.text.TfidfVectorizer.html).

El resultado del vectorizador es una matriz dispersa[1] N * D, donde N es el conjunto de tweets de training y D es el número de palabras tras la vectorización.

[ ]:
```
vectorizer = TfidfVectorizer(stop_words='english')

X_TRAIN = vectorizer.fit_transform(xtraincorpus)
X_TEST = vectorizer.transform(xtestcorpus)
```

Las llamadas *stop words* son palabras que carecen de significado como artículos, pronombres, preposiciones, etc. Estas son filtradas antes de realizar el procesamiento de los datos, por eso lo indicamos al llamar a la función.

4. LLamamos a la máquina de soporte vectorial

Las máquinas de soporte vectorial (SVM) son algoritmos que permiten llevar a cabo problemas de clasificación (como es nuestro caso) y regresión. La SVM construye un hiperplano o conjunto de hiperplanos en un espacio de dimensionalidad muy alta (o incluso infinita). Una buena separación entre las clases permitirá una clasificación correcta.

El algoritmo posee dos hiperparámetros fundamentales. Vamos a explicar brevemente su significado:

Idealmente, el modelo basado en SVM debería producir un hiperplano que separe completamente los datos de la muestra estudiada en dos categorías. Sin embargo, una separación perfecta no siempre es posible y, si lo es, el resultado del modelo no puede ser generalizado para otros datos. Esto se conoce como sobreajuste (*overfitting*).

Con el fin de permitir cierta flexibilidad, las SVM manejan un **parámetro C** que controla

[1] Una matriz dispersa es una matriz de gran tamaño en la que la mayor parte de sus elementos son cero.

la compensación entre errores de entrenamiento y los márgenes rígidos, creando así un margen blando (soft margin) que permita algunos errores en la clasificación a la vez que los penaliza.

La manera más simple de realizar la separación es mediante una línea recta, un plano recto o un hiperplano N-dimensional.

Desafortunadamente las muestras a estudiar no se suelen presentar en casos idílicos de dos dimensiones, sino que generalmente tratamos con más de dos variables predictoras, curvas no lineales de separación, casos donde los conjuntos de datos no pueden ser completamente separados o clasificaciones en más de dos categorías.

Debido a las limitaciones computacionales de las máquinas de aprendizaje lineal, éstas no pueden ser utilizadas en la mayoría de las aplicaciones del mundo real. La representación por medio de funciones Kernel ofrece una solución a este problema, proyectando la información a un espacio de características de mayor dimensión el cual aumenta la capacidad computacional de la máquinas de aprendizaje lineal. Es decir, mapearemos el espacio de entradas a un nuevo espacio de características de mayor dimensionalidad donde es más sencillo computacionalmente realizar la separación.

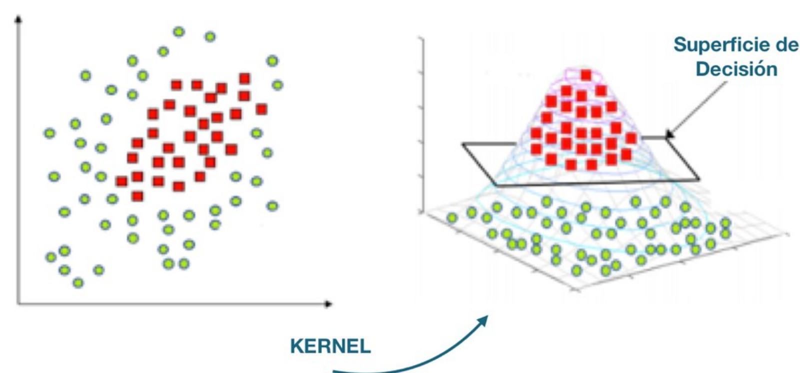

Figura 2: Ejemplo gráfico de la función que desempeña el kernel.
Imagen tomada de https://aprendeia.com/kernel-maquinas-vectores-de-soporte-clasificacion-regresion/

Existen diferentes funciones kernel que se pueden emplear. Realizaremos pruebas de entrenamiento empleando diferentes funciones kernel y valores de C distintos, para posteriormente compararlas y ver cual es la configuración óptima.

Definimos una función denominada evalModels donde llamamos a la máquina de soporte vectorial iterando sobre los diferentes valores de C y kernel. En ella incluimos también la creación de una matriz de confusión, que nos muestra de manera visual los resultados del test en cada caso.

[ ]:
```
def evalModels(xTrain, yTrain, xTest, yTest, hyperparams, authors):
    results = []
    for hp in hyperparams:
        kernel = hp[0]
```

```
        C = hp[1]
        print(f"KERNEL: {kernel}, C: {C}")

        #Train y predict
        clf = svm.SVC(kernel = kernel, C = C, random_state = 23).
↪fit(xTrain, yTrain)
        yPred = clf.predict(xTest)


        correctos = (yPred == yTest)
        print(f"Porcentaje de acierto:␣
↪{accuracy_score(yTest,yPred)*100}%" )

        results.append({'kernel': kernel, 'C': C, 'resultado':␣
↪sum(correctos)/len(correctos)*100})

        np.set_printoptions(precision=2)

        titles_options = [("Confusion matrix, without normalization",␣
↪None),
                    ("Normalized confusion matrix", 'true')]


        for title, normalize in titles_options:

            disp = ConfusionMatrixDisplay.from_estimator(
                clf, xTest, yTest,
                display_labels=authors,
                cmap=plt.cm.Blues,
                normalize=normalize)
            disp.ax_.set_title(title)

        plt.show()

    results = pd.DataFrame.from_dict(results)
    best = results.sort_values(by='resultado', ascending=False).head(1)
    print("---------------------------------------------------")
    print("------------ Best SVM " + str(best['resultado'].values[0]) +␣
↪" -----------")
    print("                Kernel = " + best['kernel'].values[0] + " C␣
↪= " + str(best['C'].values[0]))
    print("---------------------------------------------------")
```

[ ]:
```
hiperparams = [("linear", 1.0), ("linear", 1.5), ("linear", 2.0),
               ("poly", 1.0), ("poly", 1.5), ("poly", 2.0),
               ("rbf", 1.0), ("rbf", 1.5), ("rbf", 2.0),
               ("sigmoid", 1.0), ("sigmoid", 1.5), ("sigmoid", 2.0)]
```

```
[ ]: evalModels(X_TRAIN, ytraincorpus, X_TEST, ytestcorpus, hiperparams,␣
     ↪['Politicos','Deportistas','Periodicos'])
```

El resultado obtenido es que el kernel más apropiado es rbf (*Radial Basis Function*) con un valor de C de 1,5. Cabe destacar que hemos realizado estas y otras pruebas para valores de C, llegando a la conclusión de que este tenía un muy pequeño impacto en la precisión del modelo. Lo que sí tiene un impacto importante es la elección del kernel.

Las matrices de confusión (normalizada y sin normalizar) que obtenemos para la configuración óptima de hiperparámetros se muestran en las siguientes figuras:

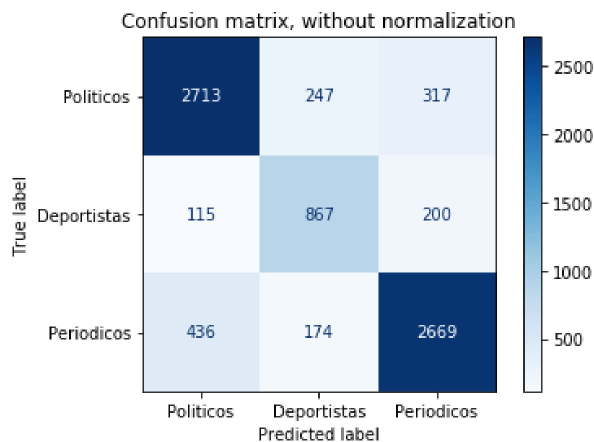

Figura 3: Matriz de confusión

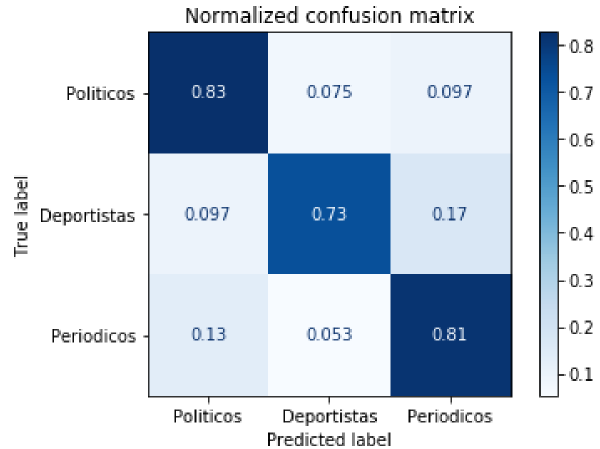

Figura 4: Matriz de confusión normalizada

Con esta configuración, el modelo nos proporciona un porcentaje de acierto del 80,8 %.

Si bien es cierto que el resultado es muy bueno, debemos tener en cuenta que estamos trabajando con cuentas que responden a un patrón de comportamiento muy concreto (como periódicos o políticos). Haciendo más pruebas, hemos entrenado el modelo para diferenciar también cantantes. En ese caso, aunque el porcentaje de acierto seguía siendo muy alto, el número de confusiones entre deportistas y cantantes era considerable. Esto se debe a que son cuentas ligeramente más personales y que por lo tanto twitean con frecuencia cosas que no tienen que ver con su profesion (y por lo tanto no siempre emplean el léxico más habitual que le correspondería). En cuentas de tipo institucional como periódicos, no se encuentran tweets personales y por lo tanto la clasificación es más acertada en promedio.

5. Análisis de sentimiento

Vamos ahora a tratar de realizar un análisis de sentimiento de los tweets de cada uno de los individuos empleando la librería Vader (Valence Aware Dictionary and sEntiment Reasoner). Es una herramienta de análisis de sentimientos basada en reglas y léxico que está específicamente orientada a los sentimientos expresados en las redes sociales.

La librería nos ofrece 4 métricas acerca de la polaridad de una frase: Positive, Negative, Neutral y Compound. El procedimiento desarrollado consistirá en sacar la métrica Compound, que es la medida con la que podemos estimar la polaridad total de la frase, para cada tweet. El sentimiento global del tweet se extrae en base a la relación que mostramos en la siguiente imagen, extraída directamente de la documentación de la librería (https://github.com/cjhutto/vaderSentiment).

1. **positive sentiment**: `compound` score >= 0.05
2. **neutral sentiment**: (`compound` score > -0.05) and (`compound` score < 0.05)
3. **negative sentiment**: `compound` score <= -0.05

[ ]:
```
analyzer = SentimentIntensityAnalyzer()
```

[ ]:
```
def getCompoundAnalysis(user):
    positive = []
    negative = []
    neutral = []
    compound = []
    tuiters = buildCorpus([user])
    for tweet in tuiters:
        score = analyzer.polarity_scores(tweet)
        positive.append(score['pos'])
        negative.append(score['neg'])
        neutral.append(score['neu'])
        compound.append(score['compound'])
    return (round(np.mean(positive),4),round(np.
 ↪mean(negative),4),round(np.mean(neutral),4),round(np.
 ↪mean(compound),4))
```

[ ]:
```
users=['@GOPChairwoman','@JoeBiden','@KingJames',␣
 ↪'@Simone_Biles','@nytimes','@BBCNews','@KamalaHarris',␣
 ↪'@LewisHamilton','@WSJ']
print("(\t\tPOSITIVE NEGATIVE NEUTRAL COMPOUND \t)")
for u in users:
    x=u
    if(u in ['@WSJ']):
        x = u + "\t"
    print(x+"\t"+str(getCompoundAnalysis(u)))
```

	POSITIVE	NEGATIVE	NEUTRAL	COMPOUND
@GOPChairwoman	0.0948,	0.1007,	0.8039,	-0.0383
@JoeBiden	0.1146,	0.0619,	0.822,	0.1305
@KingJames	0.1684,	0.0627,	0.7661,	0.2824
@Simone_Biles	0.1998,	0.052,	0.7403,	0.3034
@nytimes	0.063,	0.0685,	0.8676,	-0.0199
@BBCNews	0.0702,	0.1258,	0.804,	-0.0964
@KamalaHarris	0.1213,	0.0609,	0.8177,	0.1591
@LewisHamilton	0.1805,	0.0439,	0.752,	0.3532
@WSJ	0.0698,	0.0599,	0.8703,	0.0207

Figura 5: Resultados del análisis de sentimiento que nos proporciona la librería Vader

En la tabla anterior mostramos, para cada cuenta, la media de todos los valores para cada métrica. Vemos como en general todos los deportistas tienen una amplia polaridad positiva en promedio. Como cabría esperar, tanto en el caso de Kamala Harris como Joe Biden (ambos actualmente en el gobierno) predomina el sentimiento positivo, mientras que en el caso de Ronna McDaniel (presidenta del RNC, en la oposición), el sentimiento es neutro. En el caso de los periódicos son predominantemente neutros, a excepción de la BBC, donde predomina el sentimiento negativo.