UNIVERSITA’DEGLI STUDI DELL'INSUBRIA
DIPARTIMENTO DI SCIENZE TEORICHE E APPLICATE CORSO DI LAUREA MAGISTRALE IN INFORMATICA

Progetto Data Mining – Hate speech and offensive language recognition.

Di Russo Mattia, 725516
Boccati Eric, 724338

Indice

Indice delle figure

1

Introduzione

Lo scopo del progetto è quello di affrontare un problema di Natural Language Processing, applicando tecniche di sentiment analysis per distinguere in maniera automatica frasi offensive (offensive sentences) da frasi che generano odio (hate sentences). Un problema preliminare che notiamo è quello di dare una definizione formale di hate sentence. In generale possiamo dire che è un tipo di linguaggio che viene indirizzato verso gruppi sociali svantaggiati e che potrebbe promuovere violenza o disordini sociali (razzismo, sessismo, omofobia, ecc.).

Questa differenza tra i due tipi di linguaggi è demarcata nelle legislature di alcuni stati, dove l'utilizzo di un linguaggio di odio viene punito dalla legge; queste regole si estendono anche i social network che hanno dovuto rispondere a questo problema istituendo policy che vietano l'utilizzo di frasi di odio, e rendendo necessaria l'ideazione di meccanismi di rilevamento automatico come quello che si è cercato di realizzare in questo progetto.

Analizzeremo nei capitoli successivi le tecniche di sentiment analysis utilizzate per ottenere il riconoscimento automatico di questi tre tipi di sentences:

- Analisi del dataset
- Preprocessing dei dati
- Feature extraction
- Training e test dei modelli

2

Analisi del Dataset

Il dataset utilizzato in questo progetto contiene un corpus di tweet estratti da Twitter, scegliendo i quelli che contenessero parole selezionate da un elenco di parole di odio, reperibile sul sito hatebase.org.

In seguito, un gruppo di utenti ha assegnato a ogni tweet una tra le seguenti label: Hate, Offensive, Neither. Ogni tweet è stato "votato" da tre o più utenti, e ad ogni tweet è stata assegnata la label definitiva utilizzando una decisione per maggioranza; i tweet per i quali non è stata trovata una maggioranza sono stati scartati dal dataset.

```
df = pd.read_csv('progetto base/data/labeled_data.csv')
df.tail()
```

	Unnamed: 0	count	hate_speech	offensive_language	neither	class	tweet
24778	25291	3	0	2	1	1	you's a muthaf***in lie “@LifeAsKing: @2...
24779	25292	3	0	1	2	2	you've gone and broke the wrong heart baby, an...
24780	25294	3	0	3	0	1	young buck wanna eat!!.. dat nigguh like I ain...
24781	25295	6	0	6	0	1	youu got wild bitches tellin you lies
24782	25296	3	0	0	3	2	~~Ruffled | Ntac Eileen Dahlia - Beautiful col...

Figura A - Import del dataset e tail() del DataFrame associato

La colonna **Unnamed:0** contiene l'id originale del tweet, che nel dataset importato ha un id minore: questo perché i dati per i quali non è stata trovata una maggioranza di voti sono stati scartati.

La colonna **count** contiene il numero di quante persone hanno votato un determinato tweet, mentre **hate_speech**, **offensive_language** e **neither** rappresentano il numero di voti ricevuti dal tweet per ogni classe. L'attributo **class** contiene la label attribuita al tweet per maggioranza.

Infine, la colonna **tweet** contiene il messaggio del tweet stesso.

E' possibile notare in Figura B una larga discrepanza nel numero di tweet assegnati a ogni label: solo il 5% dei tweet sono codificati come "Hate", il 16% sono "Neither", mentre i restanti sono "Offensive".

```
countLabel = np.zeros(3)

for label in df['class']:
    countLabel[label] = countLabel[label] + 1

for index, count in enumerate(countLabel) :
    print('Percentuale di label', index , ': ', 100*(count/df['class'].size))

df['class'].hist()
```

```
Percentuale di label 0 :  5.770084332001776
Percentuale di label 1 :  77.43211072105879
Percentuale di label 2 :  16.797804946939436

<matplotlib.axes._subplots.AxesSubplot at 0x1cbe1e3ccf8>
```

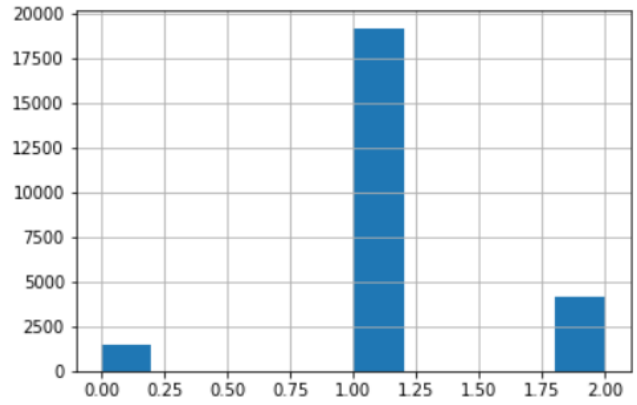


Figura B - Distribuzione delle classi

Poiché questa distribuzione non omogenea potrebbe ridurre notevolmente il recall, abbiamo scelto di utilizzare tecniche di sampling per rendere il dataset più equilibrato; per fare ciò, iniziamo applicando undersampling (ovvero eliminando dal dataset elementi che fanno parte della classe più popolata) sui tweet con classe offensive_language: creiamo un nuovo DataFrame che conterrà circa la metà dei tweet catalogati come offensivi e tutti i tweet delle altre classi. Questa riduzione nel numero dei dati dovrebbe portare alla creazione di un modello che ha una precisione generale più bassa, ma che riconosce in maniera più efficace le occorrenze delle due classi in minoranza. Cerchiamo quindi di trovare un equilibrio tra il recall (percentuale di istanze da riconoscere – nel nostro caso di tipo hate – che sono classificate come tali) e la precisione (percentuale di classificazioni esatte). Migliorare il primo a discapito

della seconda, è generalmente una tecnica utile per riconoscere istanze della classe in minoranza.

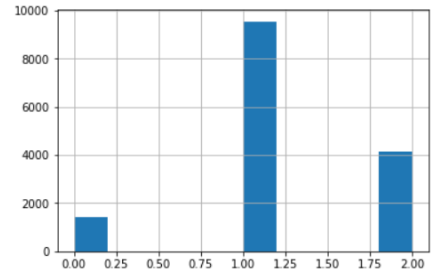
Come anticipato, abbiamo eliminato *circa* la metà dei tweet offensivi. E' stato realizzato ciò rimuovendo tutti i messaggi di classe 1 (offensive_language) di indice pari. E' stato preferito questo approccio piuttosto che eliminarli in maniera random per rendere il risultato di questo lavoro deterministico, e non dipendente da quali specifici tweet sono stati esclusi in ogni esecuzione differente.

```
columns = ['Unnamed: 0', 'count', 'hate_speech', 'offensive_language', 'neither', 'class', 'tweet']

df1 = pd.DataFrame(columns=columns)
df_offensive = pd.DataFrame(columns=columns)

for i in df.index :
    if df['class'][i] == 1 :
        if (i % 2 == 0) :
            df1 = df1.append(df.iloc[i], ignore_index=True)
        else :
            df_offensive = df_offensive.append(df.iloc[i], ignore_index=True)
    else :
        df1 = df1.append(df.iloc[i], ignore_index=True)
```

Figura C - Implementazione undersampling

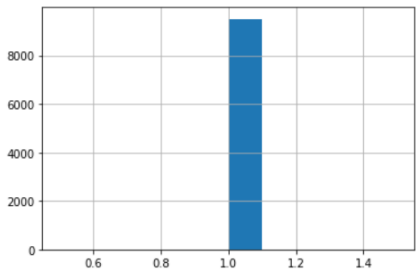
```
df1['class'].hist()
```

```
<matplotlib.axes._subplots.AxesSubplot at 0x1cbe3019e48>
```




```
df_offensive['class'].hist()
```

```
<matplotlib.axes._subplots.AxesSubplot at 0x1cbe23b6828>
```



Dal grafico della distribuzione delle classi in df1, notiamo un minore sbilanciamento – seppur consistente – nella ripartizione delle varie label. Abbiamo deciso quindi di utilizzare anche una tecnica di oversampling chiamata SMOTE (Synthetic Minority Oversampling Technique), che non crea duplicati, ma genera degli esemplari che hanno caratteristiche simili agli elementi delle classi in minoranza.

SMOTE realizza ciò scegliendo dei record, e modificandone una feature alla volta con un valore casuale, compatibile con quelli dei neighbors di tale record. Poiché stiamo lavorando con un dataset di tweet, quindi testi, questa operazione può essere svolta solo dopo le operazioni di preprocessing e feature extraction. Inoltre, da alcune

ricerche su SMOTE, è risultato che è meglio applicare questo metodo dopo aver separato il dataset in test e training data per evitare overfitting.

```
Before OverSampling, counts of label '0': 1292
Before OverSampling, counts of label '1': 8580
Before OverSampling, counts of label '2': 3745

After OverSampling, the shape of train_X: (25740, 3694)
After OverSampling, the shape of train_y: (25740,)

After OverSampling, counts of label '0': 8580
After OverSampling, counts of label '1': 8580
After OverSampling, counts of label '2': 8580
```

Figura D - Risultato oversampling

3

Preprocessing dei Dati

Osserviamo in questo capitolo il preprocessing dei dati: tecniche preliminari che ci consentono di "ripulire" i tweet da tutti gli elementi che creano rumore e che non sono utili ai fini della sentiment analysis. Ne sono un esempio i segni di punteggiatura, le citazioni (ad es. @mayasolovely), ma anche tutte quelle parole che sono molto frequenti nel testo, ma non danno un significato specifico alla singola frase come ad esempio le congiunzioni, i pronomi ecc. . Questo gruppo di parole prende il nome di **stop words**, e saranno rimosse da ogni tweet.

Un'altra tecnica importante è quella di **tokenizzazione** che consiste nel dividere ogni frase in un array di singole parole – o tokens – poiché questa struttura è utile per le fasi successive.

Infine, l'ultima tecnica per quanto riguarda il preprocessing è lo **stemming** che consiste nel rimuovere il suffisso da ogni parola, in modo tale da ottenere parole uguali da parole simili che hanno lo stesso significato.

```
df1.head()['tweet']
```

```
0    !!! RT @mayasolovely: As a woman you shouldn't...
1    !!!!! RT @mleew17: boy dats cold...tyga dwn ba...
2    !!!!!!!!!!!!! RT @ShenikaRoberts: The shit you...
3    !!!!!!!!!!!!!!!!!"@T_Madison_x: The shit just...
4    !!!!&#8220;@selfiequeenbri: cause I'm tired of...
Name: tweet, dtype: object
```

Figura E - Esempio di tweets

3.1 Rimozione punteggiatura, numeri e caratteri speciali

Come anticipato, questi tipi di simboli non danno nessun valore aggiunto per la sentiment analysis, anzi causano rumore all'interno dei testi e quindi vanno rimossi. Rimuoviamo anche gli url e le citazioni ad altri utenti, che non sono altresì rilevanti.

Per fare ciò è sono state utilizzate delle espressioni regolari; i pattern specificati all'interno del testo vengono riconosciuti ed eliminati.

```
def remove_punct_num_spec(tweets):

    cleaned_tweets = []

    for i, tweet in enumerate(tweets) :
        tmp = re.sub(r'\w+:\/{2}[\d\w-]+(\.[\d\w-]+)*(?:(?:\/[^\s/]*))*', '', tweet, flags=re.MULTILINE)
        tmp = re.sub(r'@[\w]*', '', tmp)
        tmp = re.sub(r"&amp", '', tmp)
        tmp = re.sub(r"[^A-Za-z\s']+", '', tmp)
        tmp = re.sub(r"\n", '', tmp)
        cleaned_tweets.append(tmp)

    return cleaned_tweets
```

Figura F - Funzione di ripulitura testo da url e caratteri speciali

Il risultato di questa operazione è una lista di tweet ripulita da tutti gli elementi inutili ai fini della sentiment analysis.

```
[" RT  As a woman you shouldn't complain about cleaning up your house  as a man you should always take the trash out",
 ' RT  boy dats coldtyga dwn bad for cuffin dat hoe in the st place',
 ' RT  Dawg RT  You ever fuck a bitch and she start to cry You be confused as shit',
 ' The shit just blows meclaim you so faithful and down for somebody but still fucking with hoes ',
 " cause I'm tired of you big bitches coming for us skinny girls"]
```

3.2 Tokenizing e rimozione Stop Words

Lo step successivo è quello di suddividere il testo in token. Per fare ciò abbiamo utilizzato la libreria nltk (Natural Language Toolkit), che fornisce molti strumenti per quanto riguarda il processing di frasi in linguaggio naturale. Per tokenizzare i tweet abbiamo utilizzato il modulo word_tokenize di nltk: tramite la funzione tokenize, ogni tweet viene trasformato in un array di tokens.

```python
def tweets_tokenize (tweets, tokenizer, stop_words) :

    tokenized_tweets = []
    new_sentence = []

    for tweet in tweets :
        text = tokenizer.tokenize(tweet)
        for w in text:
            if w.lower() not in stop_words:
                new_sentence.append(w.lower())

        tokenized_tweets.append(new_sentence)
        new_sentence = []

    return tokenized_tweets
```

```
[['rt', 'woman', 'complain', 'cleaning', 'house', 'man', 'always', 'take', 'trash'], ['rt', 'shit', 'hear', 'might', 'true', 'm
ight', 'faker', 'bitch', 'told', 'ya'], ['shit', 'blows', 'meclaim', 'faithful', 'somebody', 'still', 'fucking', 'hoes'], ['hob
bies', 'include', 'fighting', 'mariambitch'], ['murda', 'gang', 'bitch', 'gang', 'land']]
```

Figura G - Tokenizzazione dei tweet

In questa fase abbiamo scelto anche di rimuovere le stop words, per risparmiare un ciclo for su tutto il dataset. Le stop words non danno del significato aggiunto alle frase, al contrario rischiano di creare confusione all'interno del dataset. L'elenco di stop words è stato importato sempre dalla libreria nltk, e abbiamo aggiunto a tale elenco un'altra stop word, ovvero "rt", che è molto presente all'interno dei tweet e corrisponde alla parola "retweet", che non è rilevante ai fini della sentiment analysis.

```python
print(stop_words)
```

```
['i', 'me', 'my', 'myself', 'we', 'our', 'ours', 'ourselves', 'you', "you're", "you've", "you'll", "you'd", 'your', 'yours', 'y
ourself', 'yourselves', 'he', 'him', 'his', 'himself', 'she', "she's", 'her', 'hers', 'herself', 'it', "it's", 'its', 'itself',
'they', 'them', 'their', 'theirs', 'themselves', 'what', 'which', 'who', 'whom', 'this', 'that', "that'll", 'these', 'those',
'am', 'is', 'are', 'was', 'were', 'be', 'been', 'being', 'have', 'has', 'had', 'having', 'do', 'does', 'did', 'doing', 'a', 'a
n', 'the', 'and', 'but', 'if', 'or', 'because', 'as', 'until', 'while', 'of', 'at', 'by', 'for', 'with', 'about', 'against', 'b
etween', 'into', 'through', 'during', 'before', 'after', 'above', 'below', 'to', 'from', 'up', 'down', 'in', 'out', 'on', 'of
f', 'over', 'under', 'again', 'further', 'then', 'once', 'here', 'there', 'when', 'where', 'why', 'how', 'all', 'any', 'both',
'each', 'few', 'more', 'most', 'other', 'some', 'such', 'no', 'nor', 'not', 'only', 'own', 'same', 'so', 'than', 'too', 'very',
's', 't', 'can', 'will', 'just', 'don', "don't", 'should', "should've", 'now', 'd', 'll', 'm', 'o', 're', 've', 'y', 'ain', 'ar
en', "aren't", 'couldn', "couldn't", 'didn', "didn't", 'doesn', "doesn't", 'hadn', "hadn't", 'hasn', "hasn't", 'haven', "have
n't", 'isn', "isn't", 'ma', 'mightn', "mightn't", 'mustn', "mustn't", 'needn', "needn't", 'shan', "shan't", 'shouldn', "should
n't", 'wasn', "wasn't", 'weren', "weren't", 'won', "won't", 'wouldn', "wouldn't", 'rt']
```

Figura H - Stop words

In seguito i due grafici delle parole maggiormente presenti all'interno del corpus di tweet. Il primo prende in considerazione tutte le parole, il secondo esclude le stop words. Possiamo notare dal primo grafico che 9 delle 10 parole maggiormente diffuse, sono stop words.

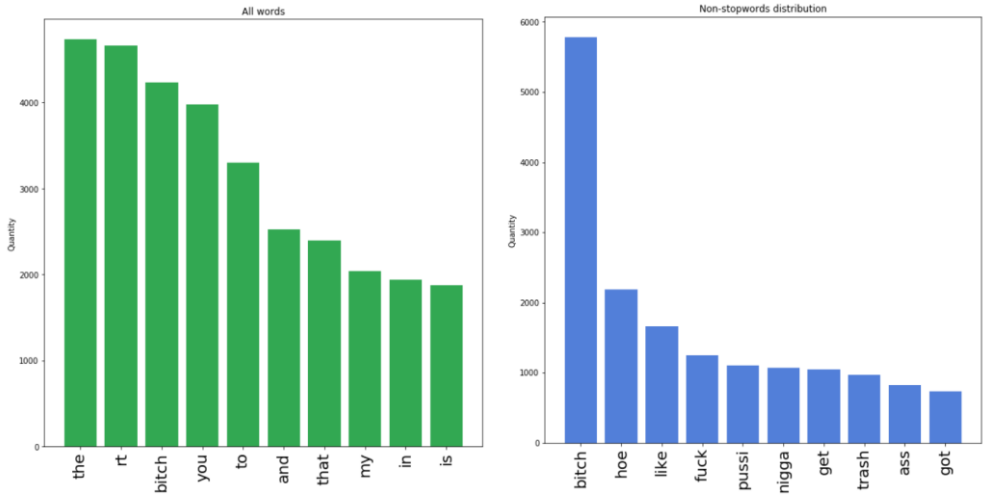


Figura I - Parole più frequenti includendo ed escludendo le stop words

3.3 Stemming

L'ultima fase di questo preprocessing consiste nello stemming, ovvero la pratica di ridurre tutte le parole alla loro radice, rimuovendone quindi il suffisso. Ad esempio, parole come "played", "playing", "playin'" vengono ridotte tutte alla loro radice "play". Anche in questo caso abbiamo utilizzato la libreria nltk attraverso il modulo SnowballStemmer. Esistono altre versioni di stemmer, come ad esempio il PorterStemmer che lavora in maniera molto simile allo Snowball (gli stessi realizzatori di Porter hanno ammesso che lo Snowball può essere considerato come una sua evoluzione), e il LancasterStemmer che abbiamo scelto di non utilizzare perché troppo "aggressivo".

```
def stem (tokenized_tweets, stemmer) :
    tokenized_stemmed = []

    for tweet in tokenized_tweets:
        tokenized_stemmed.append([stemmer.stem(word) for word in tweet])

    return tokenized_stemmed
```

```
[['rt', 'woman', 'complain', 'clean', 'hous', 'man', 'alway', 'take', 'trash'], ['rt', 'shit', 'hear', 'might', 'true', 'migh
t', 'faker', 'bitch', 'told', 'ya'], ['shit', 'blow', 'meclaim', 'faith', 'somebodi', 'still', 'fuck', 'hoe'], ['hobbi', 'inclu
d', 'fight', 'mariambitch'], ['murda', 'gang', 'bitch', 'gang', 'land']]
```

Figura J - Stemming dei token

Abbiamo utilizzato un processo di stemming e non lemmatizzazione, poiché quest'ultimo metodo, sebbene produca sempre parole esistenti nel vocabolario, è più lento rispetto allo stemmer che non si preoccupa di generare parole di senso compiuto.

4

Feature extraction

Uno dei passi più importanti è quello di feature extraction dal corpus di tweet, che consiste nel convertire un testo in un insieme di valori numerici che lo rappresentano.

Un approccio per fare ciò è la **term frequency-inverse document frequency (TF–IDF)**, che attribuisce un peso ad ogni parola in base a quanto spesso la parola ricorre all'interno del nostro corpus di tweet. TF-IDF è il prodotto tra la **term frequency** e la **inverse document frequency**:

- **tf:** è la frequenza di una parola all'interno di un singolo tweet. Si ottiene dividendo il numero di volte che la parola i occorre nel documento j, diviso la lunghezza del documento j.

$$\mathrm{tf_{i,j}} = \frac{n_{i,j}}{|d_j|}$$

- **idf:** indica la quantità di informazione fornita da una parola, in base a quanto è comune tra tutti i tweet. E' il rapporto tra il numero di documenti (tweet) e il numero di volte che la parola i appare tra tutti i documenti.

$$\mathrm{idf_i} = \log \frac{|D|}{|\{d : i \in d\}|}$$

4.1 TfidfVectorizer

Abbiamo utilizzato la classe TfidfVectorizer del modulo sklearn, che combina le funzionalità fornite da

- **CountVectorizer**: prende un array di documenti (tweets) e costruisce un modello bag-of-words, ossia un dizionario che associa ad ogni parola il numero di volte che questa parola occorre in tutti i documenti.
- **TfidfTransformer**: prende le frequenze generate da CountVectorizer come input, e le trasforma in tf-idf.

```
vectorizer = TfidfVectorizer(
    tokenizer=tokenize,
    preprocessor=remove_punct_num_spec_singleTweet,
    ngram_range=(1, 3),
    max_features=10000
)

X = vectorizer.fit_transform(df1.tweet)
pd.DataFrame(X.toarray(), columns=vectorizer.get_feature_names())[523:527]
```

	aap	ab	abil	abl	abo	abort	absolut	absurd	abt	abu	...	zebra	zero	zich	zijn	zimmerman	zip	zoe	zombi	zone	zoo
523	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	...	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.00000	0.0
524	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	...	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.00000	0.0
525	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	...	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.29024	0.0
526	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	...	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.00000	0.0

Figura K - Matrice tf-idf

Instanziamo un oggetto **TfidfVectorizer** settando innanzitutto i parametri **tokenizer** e **preprocessor**, ai quali assegnamo rispettivamente le funzioni di preprocessing e tokenizing, leggermente modificate rispetto a quelle viste in precedenza. Queste funzioni operano esattamente allo stesso modo, ma prendono come input un singolo tweet invece che una lista.

Il parametro **ngram_range** indica appunto il range di n-grams che si vuole utilizzare. Un n-gram è una sotto sequenza di un documento composta da n parole. Nel nostro caso verranno presi in considerazione unigram, bigram e trigram. Infine abbiamo deciso, tramite il parametro **max_features**, di limitare il numero di features utilizzate per la creazione della matrice a 10000. Questa scelta è stata influenzata dal fatto che un numero più alto di feature portava spesso a un errore di tipo Memory Error durante l'esecuzione del metodo vectorizer.fit_transform(). Questo valore può essere incrementato se il programma è eseguito su macchine con una RAM

più capiente; la nostra intenzione in ogni caso era quella di escludere le features meno rilevanti specificando il parametro max_features.

4.2 POS tag vectorization

Un altro metodo di feature extraction da un testo è quello del POS tagging (Part Of Speech tagging). Tramite questa tecnica si estrae un array di tags a partire da un documento. Questi tags sono relativi a quale parte del discorso (part of speech) appartiene ogni parola, ad esempio sostantivo, verbo, aggettivo, ecc.

```
def POS_tag (tweets) :
    tweet_tags = []
    for t in tweets:
        tokens = basic_tokenize(remove_punct_num_spec_singleTweet(t))
        tags = nltk.pos_tag(tokens)
        tag_list = [x[1] for x in tags]
        tag_str = " ".join(tag_list)
        tweet_tags.append(tag_str)

    return tweet_tags

POS_tags = POS_tag(df1.tweet)
POS_tags[:10]
```

```
['NN VB NN NN NN RB VBP NN',
 'NN RB VBD JJ NN NN VBD NN',
 'NN VBD JJ NNS VBG PRP JJ NNS',
 'MD VB JJ VB RB NNS',
 'NNS VBP NNS NN NN VBD NN IN NN',
 'NN NN NN NN NN',
 'NNS VBD NNS RB VBP JJ',
 'JJ NNS NN IN',
 'NN NN NN',
 'NN NN']
```

Figura L - POS tagging

La funzione POS_tag prende come input un corpus di documenti (tweets) e restituisce un array contenente i POS tags relativi a ogni documento. Per assegnare a ogni parola il suo corrispettivo tag abbiamo utilizzato la funzione pos_tag della libreria nltk che prende come input un array di token: è stato necessario creare una versione modificata del tokenizer che evitasse di applicare lo stemming alle parole da taggare,

poiché dopo aver eseguito lo stemming è impossibile riconoscere ad esempio un nome da un verbo.

Possiamo utilizzare TfidfVectorizer per estrarre le frequenze dei tag relative ad ogni tweet. Evitiamo quindi di calcolare la inverse document frequency in quanto questa informazione non è utile se applicata sui POS tags, le informazioni di questi ultimi sono utili all'interno del contesto di un singolo tweet.

```
pos_vect = TfidfVectorizer(
    tokenizer=None,
    lowercase=False,
    preprocessor=None,
    ngram_range=(1, 3),
    use_idf=False,
    smooth_idf=False,
    )

pos = pos_vect.fit_transform(pd.Series(POS_tags))
pd.DataFrame(pos.toarray(), columns=pos_vect.get_feature_names())[27:30]
```

	CC	CC CD	CC CD JJS	CC CD NNS	CC IN	CC IN NN	CC JJ	CC JJ JJ	CC JJ NN	CC JJ NNS	...	WRB NN VB	WRB NNS	WRB NNS IN	WRB NNS RB
27	0.000000	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	...	0.0	0.0	0.0	0.0
28	0.000000	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	...	0.0	0.0	0.0	0.0
29	0.204124	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	...	0.0	0.0	0.0	0.0

Figura M - Matrice tf-idf dei POS tag

4.3 Altre features

E' stata realizzata infine una funzione che estrae altre features deducibili direttamente dal testo, ad esempio la lunghezza di ogni tweet, il numero di caratteri del testo preprocessato e non, il numero di sillabe, e altre informazioni di questo tipo. Abbiamo inoltre rilevato se il tweet è un retweet cercandone all'interno la parola 'RT' (all'interno del tweet non preprocessato).

Abbiamo utilizzato inoltre un sentiment analyzer del modulo VaderSentiment per aggiungere questo tipo di feature alla nostra matrice.

```
sentiment1 = sentiment_analyzer.polarity_scores("hi dear, I'm very happy")
sentiment2 = sentiment_analyzer.polarity_scores('you should be killed')
print(sentiment1)
print(sentiment2)
```

```
{'neg': 0.0, 'neu': 0.313, 'pos': 0.687, 'compound': 0.7645}
{'neg': 0.6, 'neu': 0.4, 'pos': 0.0, 'compound': -0.6705}
```

Figura N - Sentiment Analyzer

Infine abbiamo utilizzato come features anche due parametri che fanno parte dei test di leggibilità Flesch–Kincaid. Ne esistono due tipi di test:

- FLE (Flesch Reading Ease): test che indica quanto è comprensibile un testo. Un alto punteggio a questo test indica una elevata leggibilità, viceversa un basso punteggio corrisponde ad una scarsa leggibilità. Il punteggio è calcolato nel seguente modo:

$$206.835 - 1.015\left(\frac{\text{total words}}{\text{total sentences}}\right) - 84.6\left(\frac{\text{total syllables}}{\text{total words}}\right)$$

- FK Grade Level: test che usa gli stessi parametri del FLE per calcolare la leggibilità, ma ne è inversamente proporzionale: in questo caso un basso punteggio corrisponde ad un'elevata leggibilità.

$$0.39\left(\frac{\text{total words}}{\text{total sentences}}\right) + 11.8\left(\frac{\text{total syllables}}{\text{total words}}\right) - 15.59$$

Mostriamo in Figura O la funzione che estrae queste features da ogni tweet.

```python
def other_features(tweet):

    sentiment_analyzer = VS()
    sentiment = sentiment_analyzer.polarity_scores(tweet)
    words = remove_punct_num_spec_singleTweet(tweet)

    syllables = textstat.syllable_count(words)
    num_chars = sum(len(w) for w in words)
    num_chars_total = len(tweet)
    num_terms = len(tweet.split())
    num_words = len(words.split())
    avg_syl = round(float((syllables+0.001))/float(num_words+0.001),4)
    num_unique_terms = len(set(words.split()))

    FKRA = round(float(0.39 * float(num_words)/1.0) + float(11.8 * avg_syl) - 15.59,1)
    FRE = round(206.835 - 1.015*(float(num_words)/1.0) - (84.6*float(avg_syl)),2)

    retweet = 0
    if "rt" in words:
        retweet = 1

    features = [FKRA, FRE,syllables, avg_syl, num_chars, num_chars_total,
                num_terms, num_words, num_unique_terms, sentiment['neg'],
                sentiment['pos'], sentiment['neu'], sentiment['compound'], retweet]
    return features

other_features(df1.tweet[0])
```

```
[9.1, 69.14, 30, 1.3636, 114, 140, 25, 22, 20, 0.0, 0.12, 0.88, 0.4563, 0]
```

Figura O - Altre features

4.4 Unire tutte le features

Concateniamo queste matrici di features, creando il nostro dataset definitivo. Ricordiamo che questo dataset non contiene i tweet che erano stati rimossi con underfitting.

	aap	aaron	ab	abil	abl	abo	abort	absolut	abt	abu	...	num_chars	num_chars_total	num_terms	num_words	num_unique_words	vader neg	vader pos	va
27	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	...	58.0	60.0	13.0	11.0	11.0	0.173	0.180	0.
28	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	...	63.0	65.0	13.0	11.0	10.0	0.384	0.000	0.
29	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	...	53.0	158.0	10.0	9.0	9.0	0.399	0.158	0.

3 rows × 13305 columns

Figura P - Matrice unione di tutte le features estratte

5

Training e test dei modelli

Per la scelta del modello ideale al nostro scopo sono stati presi in considerazione i seguenti classificatori:

- LogisticRegression

```
LR = LogisticRegression(solver='lbfgs', max_iter=100, multi_class="auto")
cv_LR = cross_validate(LR, X_train_res, y_train_res, cv=5, verbose=True)
```

- Bernoulli Naïve Bayes

```
BNB = BernoulliNB()
cv_BNB = cross_validate(BNB, X_train_res, y_train_res, cv=5, verbose=True)
```

- DecisionTree

```
DTC = DecisionTreeClassifier()
cv_DTC = cross_validate(DTC, X_train_res, y_train_res, cv=5, verbose=True)
```

- Support Vector Machine

```
SVC = svm.SVC(gamma='scale')
cv_SVC = cross_validate(SVC, X_train_res, y_train_res, cv=5, verbose=True)
```

I modelli sono stati addestrati mediante la funzione cross_validate, che divide il dataset in un numero di fold passato come parametro e successivamente esegue l'addestramento mediante cross validation.

I modelli sono stati valutati in base a due fattori:

- Tempo di fitting del modello

- Accuratezza delle predictions

Per quanto riguarda il tempo di fitting possiamo notare che il DecisionTreeClassifier (DTC) impiega un tempo decisamente lungo probabilmente a causa dell'elevato numero di features (ricordiamo che abbiamo preso in considerazione solo le 10000 features più rilevanti) da analizzare a ogni split dell'albero di decisione.

I modelli probabilistici LogisticRegression (LR) e BernoulliNaiveBayes (BNB) hanno impiegato un tempo decisamente minore, dovuto al minor numero di iterazioni presenti all'interno di questi algoritmi, soprattutto nel caso di BNB.

Infine, SupportVectorClassifier (SVC) non ha terminato la sua esecuzione in tempi ragionevoli, pertanto abbiamo deciso di non utilizzarlo ma comunque di analizzarne il risultato. Questo tipo di classificatore ha una complessità elevata, soprattutto in presenza di un ingente numero di features e dati da analizzare, quindi non è adatto alle nostre esigenze.

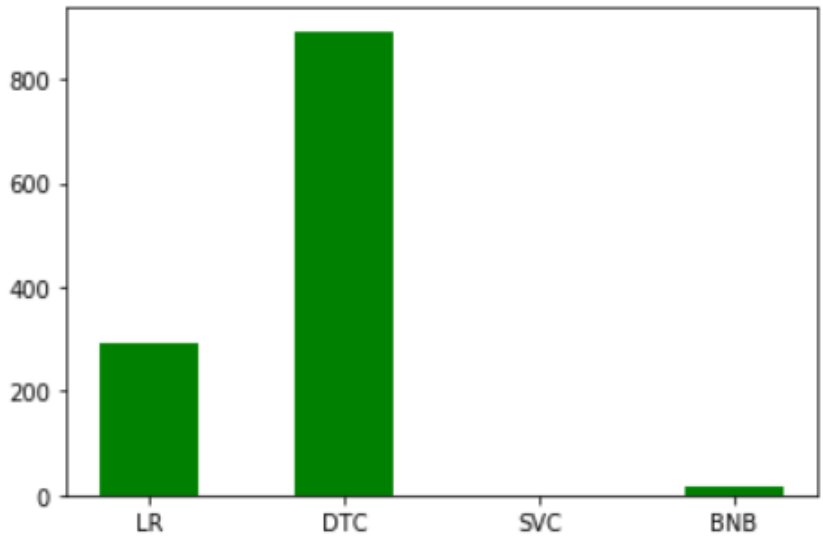


Figura Q - Analisi dei tempi di fitting (l'asse y rappresenta i secondi)

Analizziamo adesso i risultati per quanto riguarda l'accuratezza delle predictions. Dalle confusion matrix in Figura R, possiamo notare che BNB e LR presentano risultati simili; BNB è molto più preciso nel classificare i messaggi offensivi e neutri – che non è l'obiettivo di questo progetto – mentre LR riconosce in maniera leggermente più precisa i messaggi di odio, ma con una precisione generale minore.

DTC ha ottenuto i risultati migliori, che sono molto simili a quelli ottenuti dalla ricerca dalla quale questo progetto prende ispirazione, dove però il modello finale utilizzato è Logistic regression. La precisione verso le predizioni sulle classi neither e offensive è molto alta, attestandosi attorno al 90%, mentre quella verso le predizioni di messaggi di odio arriva al 59%.

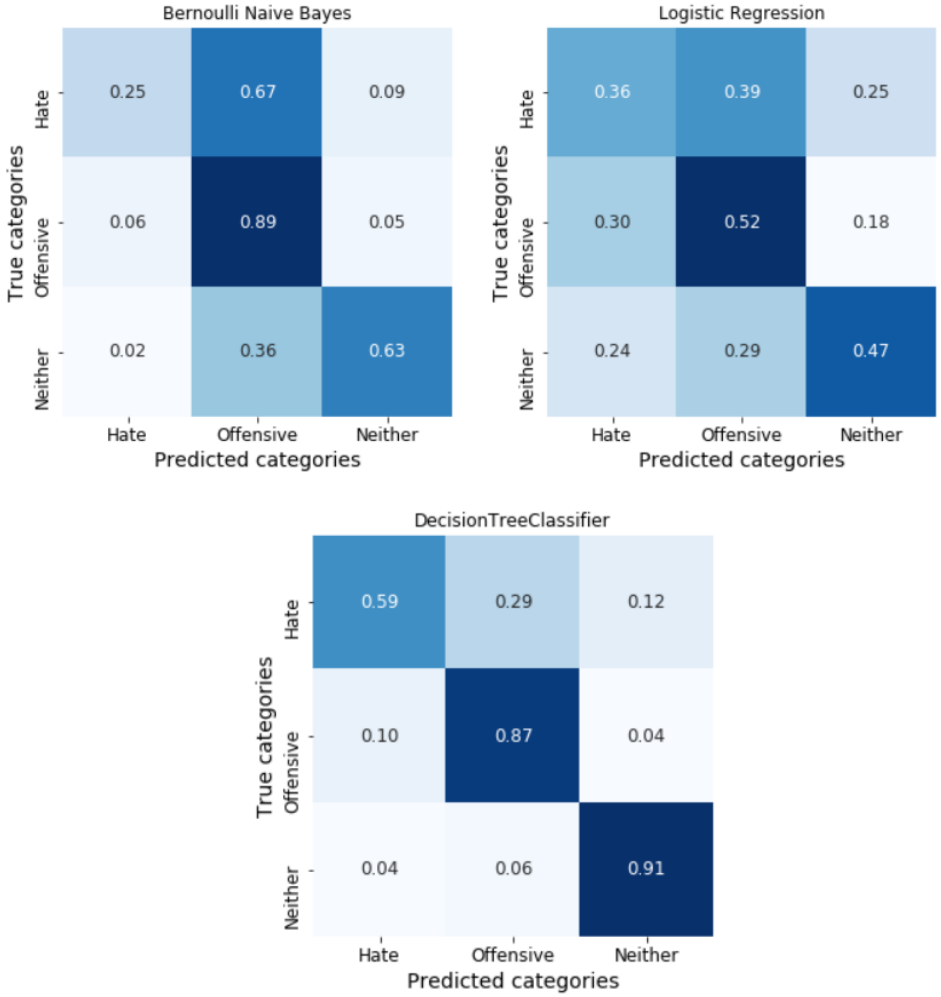


Figura R - confusion matrix di BernoulliNaiveBayes, LogisticRegression e DecisionTreeClassifier

In conclusione possiamo affermare che il modello realizzato con DecisionTreeClassifier è abbastanza affidabile, ma non essendo stato effettuato per motivi di hardware a disposizione alcun metodo di tuning degli hyperparameters, è legittimo pensare che anche gli altri due classificatori potessero esprimere migliori potenzialità. Questo è avvalorato dal fatto che, come anticipato, gli autori della ricerca sull' hate speech detection hanno ottenuto risultati simili (migliori) utilizzando LogistcRegression come classificatore.

Sarebbe inoltre interessante come sviluppo futuro utilizzare metodi di ensemble per far collaborare più modelli in modo tale da ottenere predictions ancora più precise soprattutto per quanto riguarda il rilevamento di istanze per quanto riguarda la classe Hate, che presenta le maggiori criticità venendo riconosciuta solo il 60% delle volte circa.