Report Web Analytics e Analisi Testuale

“YouTube Mining”

-ANALISI DELLE TRACCE AUDIO PER LO STUDIO DEL FENOMENO DELL'OVER POLITICALLY CORRECT-

A CURA DI: MARCO COSSU *e* CARLO A. SERRA

MATRICOLE: 11/82/00211 *e* 11/82/00130

CORSO: DSBAI

A.A. 2020/2021

Report Web Analytics e Analisi Testuale

INDICE:

INTRODUZIONE

Al giorno d'oggi, qualsiasi articolo si legga, è possibile trovare il termine "politicamente corretto". Lo si usa per riferirsi a qualsiasi cosa, in maniera piuttosto indiscriminata.
Il politicamente corretto (o politically correct) è sempre la carta jolly da inserire nei commenti.
Generalmente, si tratta di un termine usato in maniera dispregiativa.
In questo periodo si discute molto della cosiddetta 'correttezza politica', che non si propone di discernere tra ciò che è vero e ciò che è falso, ma tra ciò che si può dire e ciò che non si può dire.
Ma cos'è questa '*correttezza politica*'?
Si definisce *politicamente corretta* qualsiasi espressione, manifestazione o linea di pensiero che tenga conto del rispetto e della sensibilità di tutti, ma in particolare delle categorie socialmente deboli.
La falsa intelligenza delle macchine, unita ad una devastante dose di *politically correct* da parte di chi le programma, ha raggiunto l'apice. Un esempio clamoroso è *Agdamator*, il canale YouTube più seguito al mondo dagli appassionati di scacchi, sport che non è assolutamente conosciuto come sport violento o che discrimina il colore della pelle. Probabilmente la paura di essere non *politically correct* dei gestori di YouTube ha fatto in modo che gli algoritmi di controllo dei video siano talmente precisi da essere totalmente incapaci di discriminare in base ai contesti.
Ed è da questo punto che ha origine la nostra tesi.
Purtroppo il gioco degli scacchi, il gioco più intelligente per definizione, ha due colori di pedine: nero e bianco. E già qui si apre una parentesi razzista mica da poco. La polemica nasce dal fatto che per le regole del politicamente corretto potrebbero chiamarli bianchi e diversamente bianchi, e figuriamoci se il bianco muove per primo.
In più spesso la Regina viene sacrificata per salvare il Re, disparità di trattamento sessista. In più si parla di attacco, di cattura, di minaccia. Insomma: dagli scacchi al terrorismo il passo è breve.
E qui, l'algoritmo *politically correct* di YouTube ha pensato: chiudiamo tutto.
Questo progetto nasce, quindi, proprio dall'interesse maturato verso questa tematica e verso tutte le implicazioni che sta avendo sulla società moderna, spingendo sempre più verso un'amplificazione del fenomeno, definito oggi "over politically correct".
Vista la tendenza sempre più crescente di chiusura di canali YouTube e Twitch trattanti tematiche legate al gioco degli scacchi, a causa di presunto razzismo, abbiamo deciso di verificare in prima istanza tramite Sentiment Analysis e, successivamente, con una Cluster Analysis, se effettivamente ci potessero essere delle evidenze concrete di possibile propaganda razzista o semplicemente sia stata una cattiva gestione del problema da parte di YouTube e dei suoi algoritmi.
Nell'ipotesi in cui ci trovassimo di fronte a risultati con predominanza di sentiment negativo, allora ci sarebbero delle evidenze a supporto della tesi del possibile linguaggio razzista nei video della categoria "scacchi", tenendo bene a mente che comunque si tratta di calcoli effettuati da algoritmi automatizzati, privi di ragione e capacità di interpretazione semantica in base al contesto.

CAP.1 – Estrazione Tracce Audio da Video YouTube

1.1 Download Video da URL

La nostra base di partenza è stata lavorare sul codice delle lezioni e adattarlo alle nostre esigenze. Tendiamo a preferire uno stile di programmazione basato sull'utilizzo di funzioni piuttosto che una ad oggetti.

Il primo problema che abbiamo dovuto affrontare è stato quello di comprendere come poter ottenere una descrizione di una partita di scacchi.
Per ovviare a ciò, abbiamo dovuto cercare un video che contenesse un commento (una sorta di telecronaca) della partita in questione.

Una volta selezionato il video adatto alle nostre esigenze, abbiamo creato la funzione **extract_video_from_url()**, così da effettuare il download del video dall'url: (https://www.youtube.com/watch?v=XUVIewfjGr8&t=118s&ab_channel=GMHuschenbeth) indicato nella cartella "Videos", definita nella variabile "*path*".

```
def extract_video_from_url():
    url = "https://www.youtube.com/watch?v=XUVIewfjGr8&ab_channel=GMHuschenbeth"
    path = "Videos"
    ydl_opts = {}
    os.chdir(path)
    with youtube_dl.YoutubeDL(ydl_opts) as ydl:
        ydl.download([url])
```

1.2 Estrazione Traccia Audio da Video

Successivamente, si è proceduto con l'estrazione dell'audio commento, dal video precedentemente scaricato con l'utilizzo della funzione **extract_audio_from_video(),** utilizzando la libreria **moviepy** che permette proprio di estrapolare i file audio in formato *.wav*, unico formato utilizzabile per implementare la **Speech Recognition**.

Per richiamare le funzionalità di **moviepy**, indicata come "*mp*", deve essere specificato il percorso assoluto in cui è reperibile il file video, a causa di un problema occorso negli ultimi test.

Con la chiamata alla funzione **clip.audio.write_audiofile()** andiamo a creare il file audio.

```
def extract_audio_from_video():
    clip = mp.VideoFileClip(r"C:\Users\marco\PycharmProjects\WAAT-2021\WAAT-2021\Videos\Carlsen vs Caruana
    clip.audio.write_audiofile(r"Videos/estratto.wav")
```

1.3 Slicing Audio

Dopo aver ottenuto il file "**estratto.wav**" abbiamo effettuato un primo test della Speech Recognition, ottenendo un warning relativo all'impossibilità di poter applicare la funzione per la trascrizione dell'audio, questo a causa dell'eccessiva dimensione del file.

A tal proposito, è stata presa la decisione di suddividerlo in diversi file di dimensioni minori, salvarli nel percorso *"Videos/Audios"* ed esportarli col prefisso "**pezzo**", a cui segue il numero progressivo di file nel formato *.wav*.

Utilizziamo **slicing_audio()** all'interno della quale vengono inizializzate diverse variabili (*counter*, *interval, overlap, start, end, flag*) utili per dare le istruzioni per implementare lo slicing.
"Counter" viene usata per indicare il numero del file, "*interval"* indica la lunghezza (in millisecondi) del segmento audio, "*overlap*" è una variabile che indica la sovrapposizione tra due segmenti di audio consecutivi (per evitare che alla fine l'audio venga troncato, perdendo parte dell'informazione).
Infine, "*start*" e "*end*" indicano gli estremi del segmento.

```
def slicing_audio():
    os.makedirs('Videos/Audios')
    audio = AudioSegment.from_wav("estratto.wav")
    n = len(audio)
    counter, interval, overlap, start, end, flag = 1, 40 * 1000, 1.5 * 1000, 0, 0, 0
    for i in range(0, n, interval):
        if i == 0:
            start = 0
            end = interval
        else:
            start = end - overlap
            end = start + interval
        if end >= n:
            end = n
            flag = 1
        chunk = audio[start:end]
        filename = 'Videos/Audios/pezzo' + str(counter) + '.wav'
        chunk.export(filename, format="wav")
        print("Processing chunk " + str(counter) + ". Start = " + str(start) + " end = " + str(end))
        counter = counter + 1
```

CAP.2 – Conversione Speech2Text

2.1 Operazioni Preliminari

Con questa funzione abbiamo preparato il terreno per poter applicare la Speech Recognition all'interno di una sola cartella.
La funzione **get_file_paths()** restituisce una lista delle cartelle con tutti i file che vi sono all'interno, **os.walk()** corre all'interno della cartella estraendo tutti i nomi dei file, disposti in una struttura ad albero. Fornisce 3 tuple (*dirpath, dirnames, filenames*) da richiamare nel main in occasione della successiva funzione per la trascrizione.

os.path.join() unisce uno, o più componenti, del path in maniera intelligente. Restituisce una concatenazione del percorso e di ogni suo membro, con un separatore che divide le parti che producono un risultato. Se l'elemento finale fosse una cartella vuota l'ultimo termine restituito sarebbe un separatore.

```
def get_file_paths(dirname):
    file_paths = []
    for root, directories, files in os.walk(dirname):
        for filename in files:
            filepath = os.path.join(root,filename)
            file_paths.append(filepath)
    return file_paths
```

2.2 Speech Recognition Multipla

Tra i diversi strumenti di Speech recognition esistenti (Google-cloud-speech, pocketsphinx, ecc.) la scelta è ricaduta sul motore "**recognize_google**", indicato nella variabile *r_types* (più utilizzata), il quale viene applicato a tutti i file presenti nella cartella "*Videos/Audios*".
Nel caso in cui il riconoscimento non andasse a buon fine, verrebbe mostrato un messaggio di errore: *"Speech Recognition could not understand audio"*; il risultato, poi, è stato salvato in una stringa denominata "**result**" per poi aggiungere i termini ottenuti nella lista"**results**".

```
def recog_multiple(file):
    r = sr.Recognizer()
    r_types = ['recognize_google']
    results = []
    for r_type in r_types:
        result = ''
        with sr.AudioFile(file) as source:
            audio = r.record(source)
            try:
                result = str(getattr(r, r_type)(audio))
            except sr.UnknownValueError:
                result = 'Speech Recognition could not understand audio'
            except sr.RequestError as e:
                result = 'Speech Recognition could not understand audio; {0}'.format(e)
        results.append(result.split(' '))
    return results
```

Tutte le funzioni vengono richiamate nel ***__main__***, questo per poter poi salvare il testo nel file "***recognized.csv***" che costituisce l'oggetto su cui vengono memorizzati i dati da richiamare per creare un dataframe, utilizzando la libreria **Pandas**.

2.3 Remove Stopwords

Questa funzione utilizza come parametro la lista "*text*", contenente il testo organizzato in liste di parole. L'oggetto "*translator*" viene creato per sostituire i simboli di punteggiatura in spazi vuoti. "*Stopwords*", invece, utilizza dei metodi della libreria **nltk** per avere un unico oggetto contenente le stopwords. Infine, tramite diverse *list comprehension,* viene estratta ogni occorrenza e salvata in una lista in modo da poter eliminare i caratteri non alfanumerici, le stopwords ed i termini che non hanno sinonimi, il tutto utilizzando *synsets*, presente nella libreria ***nltk.corpus***.

La funzione restituirà la lista "*output*" contenente il testo appena processato.

```
def remove_stopwords(text):
    translator = str.maketrans(",", " ", string.punctuation)
    stopwords = nltk.corpus.stopwords.words('english')
    output = [i for i in text if i not in stopwords]
    output = [w for w in output if w.isalpha()]
    output = [word.translate(translator).lower() for word in output]
    output = [w for w in output if w and w not in stopwords]
    output = [w for w in output if wn.synsets(w)]
    return output
```

2.4 Creazione Data Frame Pandas

La struttura del file da richiamare è costituita da varie righe composte da parole separate da una virgola. Questo risulta utile in quanto, inserendo i delimitatori nella funzione di lettura **csv.reader()**, possiamo iterare all'interno del file.

Uno dei tentativi testati è stato quello di creare un data frame, utilizzando le potenzialità della libreria *Pandas*, così da poter sfruttare il tipo di struttura dei data frame e ottenere valori utili per applicare l'analisi dei Sentiment.

Questo è ciò che effettivamente fa la funzione **Get_DF()**, a cui abbiamo passato come argomento la lista contenente il testo su cui lavorare.

```
def Get_DF():
    with open('Videos/Audios/recognized.csv', 'r') as file:
        reader = csv.reader(file, delimiter=',')
        df = pd.DataFrame([word for word in enumerate(reader)], columns=["Colonna1", "Testo"])
    del df['Colonna1']
    df = df.iloc[1:]
    df["Testo"] = df["Testo"].apply(remove_stopwords)
    return df
```

Al data frame ottenuto applichiamo, infine, la funzione ***remove_stopwords***, definita precedentemente. In questo modo, effettuiamo la pulizia del testo per tutto il data frame. Nel main andremo a richiamare la funzione *Get_DF()*, a cui applicheremo la Sentiment Analysis.

```
if __name__ == "__main__":
    os.makedirs('Videos')
    extract_video_from_url()
    extract_audio_from_video()
    slicing_audio()
    DIRNAME = r'Videos/Audios'
    OUTPUTFILE = r'Videos/Audios/recognized.csv'
    files = get_file_paths(DIRNAME)
    for file in files:
        (filepath, ext) = os.path.splitext(file)
        file_name = os.path.basename(file)
        if ext == '.wav':
            a = recog_multiple(file)
            with open('Videos/Audios/recognized.csv', 'a', newline='') as file:
                writer = csv.writer(file, quoting=csv.QUOTE_NONE)
                writer.writerows(a)
    df = Get_DF()
```

CAP.3 - Sentiment Analysis

3.1 Vader Sentiment

Per eseguire la Sentiment Analysis abbiamo utilizzato, in principio nel *main*, un sentiment analyzer "pre-trainato" chiamato *VADER (ValenceAwareDictionary And sEntiment Reasoner)* presente nella libreria **NLTK,** ed un secondo sentiment analyzer per poi confrontare i risultati.

L'input che deve riceve tale funzione è una stringa, usando il metodo delle stringhe *join,* sulla lista precedentemente pulita nella funzione "*remove_stopwords*", applicata al Data Frame, in modo da poterlo utilizzare.

Siamo riusciti, così, ad eseguire l'analisi mediante l'utilizzo di poche righe di codice rispetto alla struttura iniziale. Tali risultati vengono poi salvati in varie colonne create appositamente nel Data Frame, denominate *"Negative, Neutral* e *Positive"* in cui vengono salvate le percentuali di ogni tipo di Sentiment, viene quindi attribuito ad ogni riga un sentiment sulla base della percentuale maggiore presente colonna *"Vader Sentiment"*.

```
for i in range(1, (df.shape[0] + 1)):
    sentence = [",".join(df["Testo"][i]).replace(",", " ")]
    sentences.append(str(sentence))
    q = array(sentence)
    q = vect.transform(q)
    analyzer.polarity_scores(sentence)
    df["Negative"][i] = (analyzer.polarity_scores(sentence)["neg"])
    df["Neutral"][i] = (analyzer.polarity_scores(sentence)["neu"])
    df["Positive"][i] = (analyzer.polarity_scores(sentence)["pos"])
    if df["Positive"][i] > df["Neutral"][i] and df["Positive"][i] > df["Negative"][i]:
        df["Vader Sentiment"][i] = "Positive"
    elif df["Neutral"][i] > df["Positive"][i] and df["Neutral"][i] > df["Negative"][i]:
        df["Vader Sentiment"][i] = "Neutral"
    else:
        df["Vader Sentiment"][i] = "Negative"
    df["Sentiment"][i] = str(model.predict(q))
    with ExcelWriter("Dataframe.xlsx") as writer:
        df.to_excel(writer)
```

3.2 Altro algoritmo Sentiment & Confronto dei risultati

Il Secondo Analyzer utilizzato, mediante *vettorizzazione* di ogni riga, ha fornito direttamente una classificazione divisa tra "*Positivo o Negativo*" del sentiment della riga, ed è stato salvato nella colonna "*Sentiment*" del nostro Data Frame.

Per una rappresentazione più comprensibile, l'intero Data Frame è stato esportato in formato "*.xlsx*" in modo da poter essere aperto in un file che visualizza fogli di calcolo come Microsoft Excel, o simili, attraverso il metodo *ExcelWriter*, richiamando la funzione ***to_excel(writer)***.

	A	B	C	D	E	F	G	H
1		**Testo**	**Negative**	**Neutral**	**Positive**	**Vader Sentiment**	**Sentiment**	
2	**1**	['another', 'body', 'welcome', 'cla	0	0,673	0,327	Neutral	['pos']	
3	**2**	['position', 'abuse', 'removes', 'bl	0,108	0,464	0,428	Neutral	['pos']	
4	**3**	['pond', 'egg', 'still', 'keep', 'advar	0,042	0,541	0,417	Neutral	['pos']	
5	**4**	['put', 'takes', 'takes', 'fruit', 'get',	0	0,652	0,348	Neutral	['neg']	
6	**5**	['dangerous', 'imagine', 'tough', '	0,124	0,567	0,309	Neutral	['pos']	
7	**6**	['still', 'almost', 'feels', 'like', 'bloc	0,079	0,595	0,325	Neutral	['pos']	
8	**7**	['like', 'needs', 'create', 'play', 'wc	0	0,475	0,525	Positive	['pos']	
9	**8**	['play', 'push', 'parts', 'queen', 'pı	0,059	0,499	0,442	Neutral	['neg']	
10	**9**	['snipe', 'battery', 'defend', 'nazi'	0,124	0,648	0,227	Neutral	['neg']	
11	**10**	['pistol', 'loader', 'tricky', 'someti	0,141	0,56	0,299	Neutral	['neg']	
12	**11**	['hurts', 'get', 'time', 'clock', 'gooc	0,155	0,462	0,384	Neutral	['neg']	
13	**12**	['question', 'howard', 'deals', 'pe	0,105	0,584	0,31	Neutral	['pos']	
14	**13**	['yes', 'get', 'points', 'get', 'even',	0,06	0,759	0,182	Neutral	['neg']	
15	**14**	['board', 'inn', 'drive', 'surprising'	0,17	0,633	0,197	Neutral	['pos']	
16	**15**	['appointment', 'push', 'forward'	0,137	0,541	0,322	Neutral	['pos']	
17	**16**	['roman', 'become', 'actor', 'broo	0,11	0,729	0,161	Neutral	['neg']	
18	**17**	['hope', 'enjoyed', 'game', 'maste	0,024	0,365	0,611	Positive	['neg']	
19	**18**	['play', 'sin', 'hear', 'bishop', 'bish	0,075	0,774	0,151	Neutral	['pos']	
20	**19**	['goes', 'listen', 'grab', 'space', 'th	0,17	0,685	0,144	Neutral	['neg']	
21	**20**	['put', 'pressure', 'paws', 'positio	0,05	0,698	0,252	Neutral	['pos']	
22	**21**	['advanced', 'white', 'black', 'side	0,18	0,501	0,319	Neutral	['neg']	
23	**22**	['nice', 'game', 'plate', 'course', 'd	0,048	0,771	0,181	Neutral	['pos']	
24	**23**	['like', 'pond', 'taking', 'poem', 'cc	0,193	0,607	0,2	Neutral	['pos']	
25	**24**	['game', 'open', 'file', 'supports', '	0	0,579	0,421	Neutral	['pos']	
26								

Come si può notare dall'immagine qui sopra, utilizzando il tool Vader la maggior parte delle righe viene classificata come "*Neutral*" e solamente due come "*Positive*". Le percentuali di Sentiment negativo rilevato sono estremamente basse, ma comunque presenti, probabilmente dovute a singole parole mal recepite dalla funzione di Speech Recognition (un esempio potrebbe essere nella riga 9 la parola "*nazi*", assolutamente di senso negativo, ma difficilmente utilizzabile nel contesto di una partita del gioco degli scacchi).

La ratio del secondo metodo per capire il Sentiment utilizzato sembra attribuire un peso maggiore alle parole di senso negativo, guardando il confronto con le percentuali fornite da Vader o, semplicemente, fornisce una diversa classificazione delle frasi dovendo selezionare una tra due sole modalità (Positiva o Negativa), al contrario di quanto succede con Vader.

Abbiamo iniziato la nostra analisi lavorando sui primi 4 “pezzi” di audio, che hanno costituito il nostro *Training set*. Successivamente, abbiamo verificato la fondatezza dei risultati ottenuti e, infine, applicato la rappresentazione al Data Frame per intero.

3.4 Topic Modeling

Questo è un tipo di modello che viene utilizzato per scoprire quali sono gli argomenti (in inglese, *topic*) che vengono più utilizzati in una raccolta di documenti, nel nostro caso in un insieme di frasi. Lavora prendendo i gruppi di parole che appaiono più spesso in un testo.
Questi saranno i topic estratti, analizza le statistiche delle parole di ciascun testo.

```
# TOPIC MODELING
def display_topics(model, feature_names, no_top_words):
    for topic_idx, topic in enumerate(model.components_):
        print("Topic %d:" % (topic_idx))
        print(" ".join([feature_names[i] for i in topic.argsort()[:-no_top_words - 1:-1]]))
```

La funzione viene chiamata ***display_topics*** e, attraverso un *ciclo for*, restituisce il numero del topic e l’insieme di parole che ne fanno parte.
Nel nostro caso, nel *main*, viene impostato un numero massimo di 10 topic, i più rilevanti.

```
# TOPIC MODELING
no_topics = 10
no_features = 1000
tf_vectorizer = CountVectorizer(max_df=0.95, min_df=2, max_features=no_features,
                                stop_words='english')
tf = tf_vectorizer.fit_transform(sentences)
tf_feature_names = tf_vectorizer.get_feature_names()
lda = LatentDirichletAllocation(n_components=no_topics, max_iter=5,
                                learning_method='online', learning_offset=50.,
                                random_state=0).fit(tf)
display_topics(lda, tf_feature_names, 5)
```

La funzione ***CountVectorizer***, contenuta nella libreria *Scikit-Learn*, converte una raccolta di documenti di testo in un vettore di numeri su cui si possono effettuare calcoli statistici e matematici che non sarebbe possibile eseguire su un testo. Viene applicata a tutte le frasi che sono contenute nella lista *sentences,* per poi estrarre il nome delle prime 5 feature dal modello e mostrarle come output, nell’IDE di Python.

Nell'immagine in basso si evidenziano i primi 10 topic estratti, ognuno di essi contiene una sequenza di parole selezionate per rilevanza, si può notare che l'argomento comune a tutte è sicuramente quello di una partita di scacchi (*es. check, king, queen, bishop*).

```
Topic 0:
checks king check board course
Topic 1:
open checks abuse lot impact
Topic 2:
person imagine protect comes convincing
Topic 3:
queen night taken playing board
Topic 4:
game black play look want
Topic 5:
queen fashion time check yes
Topic 6:
bishop check queen nice pond
Topic 7:
position taking win like really
Topic 8:
hope like know video check
Topic 9:
threatening come board question protect
```

CAP.4 - Algoritmi di Classificazione

4.1 Bernoulli

La funzione **random** permette di legarsi a un generatore di numeri casuali, così che la casualità non influisca sulle varie prove di configurazione del training.

X e **Y** sono le colonne del nostro data frame, a cui abbiamo associato rispettivamente la lista delle parole *sentences* e la sezione del dataframe con la classificazione della riga in Positivo e negativo.

Abbiamo chiamato la funzione ***CountVectorizer***, assegnandola alla variabile *"vect"*, che ci permette di *vettorizzare* le singole frasi. In questo caso, abbiamo impostato l'opzione ***ngram_range(1,2)*** per considerare anche le parole composte da due termini contigui (per considerarne solamente una basta mettere (1,1) in modo che l'apprendimento sia più veloce).

Vettorizzata la variabile **X**, importiamo il metodo ***train_test_split*** da scikit-learn per separare il dataset in training e test set, quest'ultimo composto dal 30% delle osservazioni totali il quale ci servirà per provare l'affidabilità del modello.

A questo punto abbiamo scelto un algoritmo di apprendimento tra i classificatori disponibili.

Optiamo per l'algoritmo di *naive bayes* ***BernoulliNB***.
Impostato il modello con l'algoritmo *(model = BernoulliNB())*, lo abbiamo addestrato sui dati *training (X_train, y_train)* con il metodo ***fit().***

Al termine dell'addestramento abbiamo usato il metodo ***predict()*** per ottenere i risultati sui *test (X_test)*.

```
# # MODELLO BERNOULLIANO
print("MODELLO BERNOULLIANO")
random.seed(10)
X, y = sentences, df['Sentiment']
vect = CountVectorizer(ngram_range=(1, 2))
X = vect.fit_transform(X)
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.33)
model = BernoulliNB()
model.fit(X_train, y_train)
p_test = model.predict(X_test)
print("Accuratezza del modello: ", accuracy_score(y_test, p_test))
print("Tasso di errata classificazione: ", 1 - (accuracy_score(y_test, p_test)))
print("\n")
```

Abbiamo importato da ***scikit learn*** un metodo per valutare l'accuratezza del modello. Uno dei più semplici è ***accuracy_score()***.

In seguito, è stato utilizzato per confrontare le etichette corrette dei dati test (y_test), con le risposte elaborate dal programma (p_test), e abbiamo stampato il risultato.

Per completezza, abbiamo calcolato e stampato anche il *Tasso di Errata Classificazione*, calcolato come il complemento dell'*accuracy score*.

Il modello ha classificato i dati test con un'accuratezza del 75% e un errore pari al 25%.

```
MODELLO BERNOULLIANO
Accuratezza del modello:  0.75
Tasso di errata classificazione:  0.25
```

Ciò significa che ha classificato correttamente più di 7 dati su 10.
Usando altri algoritmi, il risultato potrebbe essere diverso. Perciò, abbiamo deciso di addestrare anche una Random Forest.

4.2 Random Forest

Abbiamo creato il nostro classificatore "*classifier*", utilizzando la funzione per implementare l'algoritmo **Random Forest**.

Il parametro *n_estimators* indica il numero di *stimatori*, o alberi nella foresta: 100 nell'esempio. Successivamente, abbiamo adattato il modello al training set utilizzando **fit (X_train, y_train)** e abbiamo eseguito la previsione sul test set usando **predict (X_test)**.

```
# RANDOM FOREST
print("RANDOM FOREST")
classifier = RandomForestClassifier(n_estimators=100, random_state=0)
classifier.fit(X_train, y_train)
y_pred = classifier.predict(X_test)
print("Matrice di confusione: \n", confusion_matrix(y_test, y_pred))
print("Report di classificazione: \n", classification_report(y_test, y_pred))
print("Accuratezza del modello Random Forest: ", accuracy_score(y_test, y_pred))
print("Tasso di errata classificazione del modello Random Forest: ", 1 - (accuracy_score(y_test, y_pred)))
print("\n")
```

Il modello viene ora valutato usando la **matrice di confusione**.

Il fondamento di una matrice di confusione è il numero di previsioni corrette (veri positivi) e non corrette (falsi positivi) riassunte in termini di classe.
Otterremo la matrice di confusione sotto forma di oggetto array.
La dimensione di questa matrice è 2x2 perchè abbiamo un modello di classificazione basato su due classi, positivo e negativo.
I valori diagonali rappresentano previsioni accurate, mentre gli elementi non diagonali sono previsioni imprecise.

```
RANDOM FOREST
Matrice di confusione:
 [[2 0]
 [6 0]]
Report di classificazione:
              precision    recall  f1-score   support

     ['neg']       0.25      1.00      0.40         2
     ['pos']       0.00      0.00      0.00         6

    accuracy                           0.25         8
   macro avg       0.12      0.50      0.20         8
weighted avg       0.06      0.25      0.10         8

Accuratezza del modello Random Forest:  0.25
Tasso di errata classificazione del modello Random Forest:  0.75
```

Il secondo indice di valutazione è il **report di classificazione**: riassumiamo i risultati ottenuti dal modello, la precisione, il richiamo, l'f1 score e il supporto.

Precision: su 8 previsioni negative ("neg"), l'algoritmo di random forest ha individuato il 25% (2/8) delle istanze come negative.
Per le 0 previsioni positive ("pos") ovviamente il risultato è 0.
La *Precision* può essere vista come una misura di *esattezza* o fedeltà, ed è definita come il numero di documenti attinenti recuperati da una ricerca diviso il numero totale di documenti recuperati dalla stessa ricerca

Recall: per tutte le istanze che erano effettivamente positive, lo 0% (0/6) è stata classificata correttamente per i Positivi, mentre tale percentuale raggiunge il 100% (2/2) se si tratta dei Negativi. La *recall* è una misura di *completezza*, ed è definita come il numero di documenti attinenti recuperati da una ricerca diviso il numero totale di documenti attinenti esistenti (che dovrebbe essere stato recuperato).

Nell'information retrieval, *precision* e *recall* sono definite in termini di insieme di documenti recuperati (lista di documenti restituiti da un motore di ricerca rispetto ad una query) e un insieme di documenti attinenti (lista di tutti i documenti che sono attinenti per l'argomento cercato).

F1-score: è una media armonica ponderata delle due precedenti metriche.

Il **grado di accuratezza (accuracy)**: indica la frazione dei campioni predetti correttamente. Ciò ci ha permesso di concludere che solo quasi **3 volte su 10** il modello è in grado di predire correttamente.

Cap.5 - Cluster Analysis

5.1 K-MEANS

Il primo metodo di clusterizzazione scelto è stato il ***K-Means***, richiamato mediante la funzione ***cluster_analysis()*** *e,* osservando il grafico riguardante il numero di cluster (Elbow), viene scelto un valore **k=7**.

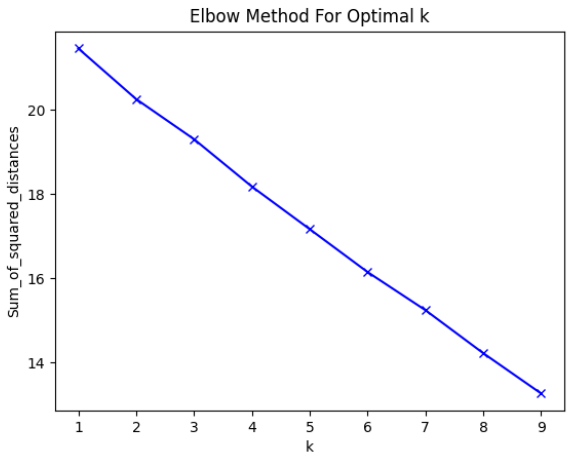
```
def cluster_analysis(sentences):
    tfidfVectorizer = TfidfVectorizer()
    X = tfidfVectorizer.fit_transform(sentences)
    Sum_of_squared_distances = []
    K = range(1, 10)
    for k in K:
        km = KMeans(n_clusters=k, max_iter=200, n_init=10)
        km = km.fit(X)
        Sum_of_squared_distances.append(km.inertia_)
    plt.plot(K, Sum_of_squared_distances, 'bx-')
    plt.xlabel('k')
    plt.ylabel('Sum_of_squared_distances')
    plt.title('Elbow Method For Optimal k')
    plt.show()
    true_k = 7  # abbiamo scelto 7 in quanto risulta essere l'unico gomito visibile
    model = KMeans(n_clusters=true_k, init='k-means++', max_iter=200, n_init=10)
    model.fit(X)
    labels = model.labels_
    cl = pd.DataFrame(list(zip(df.index, labels)), columns=['N° Frasi', 'cluster'])
    result = {'cluster': labels, 'testo': sentences}
    result = pd.DataFrame(result)
    for k in range(0, true_k):
        s = result[result.cluster == k]
        text = s['testo'].str.cat(sep=' ')
        text = text.lower()
        text = ' '.join([word for word in text.split()])
        wordcloud = WordCloud(max_font_size=75, max_words=100, background_color="black").generate(text)
        print('Cluster: {}'.format(k))
        print('N° Frasi')
        titles = cl[cl.cluster == k]['N° Frasi']
        print(titles.to_string(index=False))
        plt.figure()
        plt.imshow(wordcloud, interpolation="bilinear")
        plt.axis("off")
        plt.show()
```

L'argomento fornito alla funzione è la lista *sentences*, contenente il testo di ogni riga del data frame; gli elementi di questa lista vengono vettorizzati mediante la funzione ***tfidfVectorizer.fit_transform()***. questo risulta essere il modello più comunemente usato. L'acronimo sta per *"Term Frequency - Inverse Document Frequency" ed* opera in modo da assegnare un punteggio ad ogni parola all'interno del documento.

Le due parti di questo algoritmo sono così definite:

- **Term Frequency**: calcola quanto spesso una parola appare in un determinato documento;
- **Inverse Document Frequency**: diminuisce il punteggio assegnato ad una parola che compare spesso tra i vari documenti;

Nella funzione, prima di tutto, è stato disegnato un grafico che mostra la relazione tra i k-possibili cluster e la somma delle distanze al quadrato, definita come distanza di ciascuna osservazione dal suo centroide.



Risulta indicato come miglior k il valore 7, in modo da utilizzarlo durante la ricerca dei cluster e delle frasi che lo compongono.
Per rendere più semplice l'analisi è stato creato un nuovo data frame denominato **cl**, contenente due colonne:
il numero di frase ed il cluster a cui essa viene ricondotta.

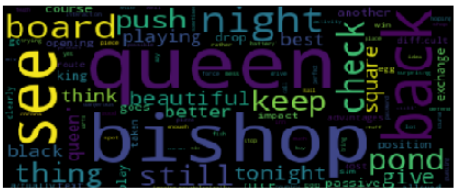
Infine, la creazione di un terzo data frame chiamato *result*, ha permesso la realizzazione di una ***WordCloud*** per cluster e la stampa dei risultati dell'analisi. Essa è una rappresentazione visiva di parole (o tag) più comunemente utilizzate in un testo, in questo caso per testo si intende l'insieme delle righe del data frame che fanno parte di un cluster.

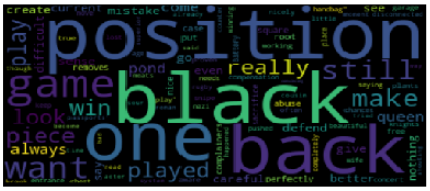
```
Cluster: 0
N° Frasi
 5
18
22
Cluster: 1
N° Frasi
 2
20
24
Cluster: 2
N° Frasi
 4
 7
 8
 9
15
16
19
21
23
```

```
Cluster: 3
N° Frasi
 3
13
17
Cluster: 4
N° Frasi
 6
14
Cluster: 5
N° Frasi
10
11
Cluster: 6
N° Frasi
 1
12
```

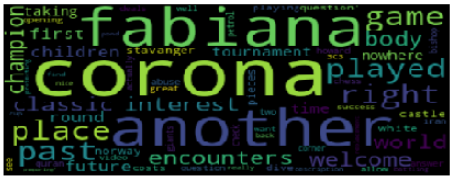
In genere, questa lista è presentata in ordine alfabetico, con la peculiare caratteristica di attribuire un font di dimensioni più grandi alle parole più importanti.
Si tratta quindi di una lista pesata.
Le immagini a lato mostrano i 7 cluster, e le rispettive frasi che vi appartengono.

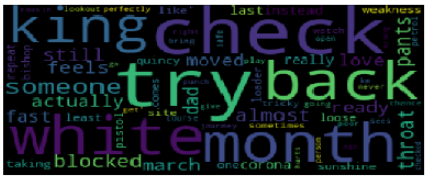

roughly
wants
paws
putpressure
equal
positions
well
far
openings

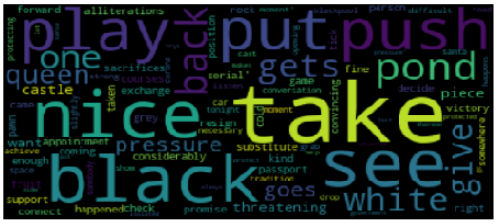

board
push
night
queen
see
bishop
back
pond
thing
still
tonight
keep
beautiful
check


position
black
one
back
game
win
make
really
still
want
played
piece
always


video
question
check
know
queen
hope
hurts
stop
time
take
yes
age
elegant
sofas
game
surgery
guys

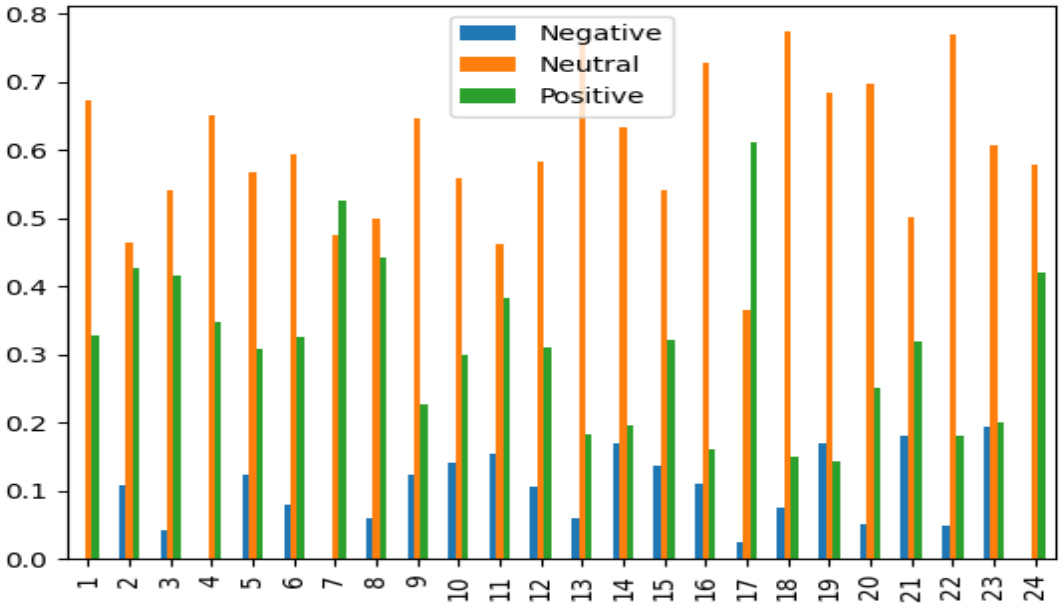

fabiana
corona
another
played
game
body
right
place
past
encounters
welcome
champion
tournament
nowhere


king
check
tryback
white
month
moved
someone
actually
fast
blocked
march
ready
almost
throat
pants


play
put
push
one
queen
back
gets
pond
castle
nice
take
see
black
pressure
white
give
goes
threatening

5.2 Rappresentazioni Grafiche

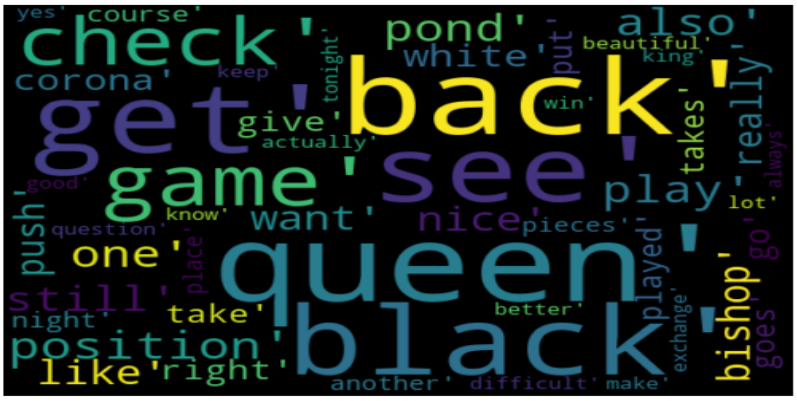
Questa rappresentazione grafica è stata creata utilizzando la libreria ***matplotlib***, chiamando la funzione **df.plot.bar()** che traccia il *Bar Plot sui dati forniti dal data frame df.*
Questo tipo di grafico mette in relazione il numero della riga con le frequenze delle 3 diverse polarità calcolate dal *tool Vader*: *negative, positive e neutral*. Così facendo, avremo una prima idea della tendenza del testo, cercando evidenze sulla presenza o meno di intenzionalità *razziste*.
Si può notare che, in varie frasi, vi è la totale assenza di sentiment negativi, con un'elevatissima presenza di neutralità e una rilevante presenza di positività, dall'analisi con questo tool risulta abbastanza evidente che non vi siano prove di intenzioni razziste nel video analizzato.




```
# CLUSTER ANALYSIS K-MEANS
cluster_analysis(sentences
                 )
# CREAZIONE BAR PLOT FREQUENZE SENTIMENT
plt.figure()
df.plot.bar()
plt.show()

# WORDCLOUD DEL TESTO TOTALE
allWords = ' '.join([str(word) for word in df['Testo']])
wordCloud = WordCloud(width=600, height=400, random_state=21, max_font_size=110, max_words=50).generate(allWords)
plt.imshow(wordCloud, interpolation="bilinear")
plt.axis('off')
plt.show()
```

Il grafico successivo è una WorldCloud, analoga alle precedenti, ma il testo da cui è generata comprende tutte le frasi presenti nel video oggetto del nostro lavoro. Anche da un'analisi sommaria, appare chiaro che non vi siano evidenze di razzismo nel senso totale del testo analizzato; anzi, risulta evidente che il macro-argomento comune che pervade tutto il documento sia una partita a scacchi, rilevata attraverso vari termini, tra cui: *queen, black, white, game, check, bishop, play.*



5.3 Alcune precisazioni

Nella sezione MAIN richiamiamo le diverse funzioni del progetto, volte all'ottenimento, la preparazione e l'elaborazione dei dati.

La prima funzionalità che viene richiamata è ***os.makedirs()*** che serve a creare la cartella *Videos* in cui andremo a conservare i file, Video e Audio. Le variabili DIRNAME e OUTPUTFILE servono, rispettivamente, a specificare il relative path in cui si trovano i file audio e il file in formato *.csv* in cui salvare la trascrizione di questi ultimi.

Le funzioni ***extract_video_from_url()***, ***extract_audio_from_video()***, ***slicing_audio()*** servono, come già preannunciano i nomi, a preparare il file video originale e i derivanti segmenti audio per poter applicare la Speech Recognition, il cui risultato verrà poi scritto, attraverso ***writer.writerows()***, nel file *recognized.csv*. Il ciclo for ci permette di iterare all'interno della cartella Audios e di andare a operare solamente sui file in formato *".wav"* che si trovano al suo interno, in modo da evitare errori nel caso vi si trovassero altri tipi di file.

Successivamente viene creato il data frame mediante pandas, con la funzione ***Get_DF()***, esso viene poi modificato per salvare i risultati derivanti dal tool Vader Sentiment, nelle colonne *"Neutral", "Negative" e "Positive"* vengono salvate le varie percentuali rilevate, nella colonna *"Vader Sentiment"* è inserito il sentiment prevalente. Un'ulteriore colonna, denominata *"Sentiment"*, è utilizzata dalla predizione del sentiment basata sul *modello Bernoulliano*, in essa viene indicato semplicemente il sentiment rilevato nella frase corrispondente. Tutti questi risultati vengono successivamente salvati nel file "*Dataframe.xlsx*" mediante la funzione ***df.to_excel()***.

Le stringhe seguenti servono ad impostare i valori per poter applicare la *Topic Modeling, il modello Bernoulliano, la Random Forest, il K-Means clustering* e stamparne i relativi risultati per poter valutare tutti modelli.

Le ultime righe richiamano la funzione ***df.plot.bar()*** per ottenere il grafico che mette in relazione i vari sentiment che compongono le varie frasi e la ***WordCloud*** totale di tutto il documento.

CONCLUSIONI

Per concludere, abbiamo visto che è possibile analizzare dati non strutturati, come audio estratti da un video, per effettuare analisi statistiche finalizzate a sostenere determinate tesi, ma anche a confutarle.

In particolare, con il nostro progetto, abbiamo confermato la tesi di fondo: non vi sono chiare evidenze circa intenzionalità razziste nelle parole di uno spiker che commenta partite di scacchi.

Quindi, siamo di fronte a una classica situazione di over politically correct.

E i dati ci fanno da testimoni, ci danno la prova tangibile e inconfutabile.

Infatti, già da principio, era strano pensare che ci fosse qualcuno con fini di propaganda razzista dietro video di questo tipo. Il dubbio nasceva da diversi termini, fraintendibili da un'intelligenza artificiale, che innescavano il blocco dei canali YouTube a causa di algoritmi basati su discriminanti davvero troppo stringenti (si pensi a quanto vengano ripetute negli scacchi le parole "black" e "white"). Tutto ciò ha portato a un down del sistema.

La buona notizia è che la tecnologia, ad oggi, sta operando per evitare che si ripetano errori di questo tipo, sfruttando sistemi sempre più complessi che vanno ad analizzare nello specifico i contesti in cui determinati termini vengono utilizzando, evitando disservizi ma al contempo sempre combattendo quei canali in cui effettivamente ci sono contenuti con oggetto l'odio razziale.