UNIVERSITÀ DEGLI STUDI DI CAGLIARI

DATA SCIENCE, BUSINESS ANALYTICS E INNOVAZIONE

Web Analytics e analisi testuale

Report finale progetto

Michael Bellu, 11/82/00190

Alessandro Saiu, 11/82/00213

Indice

0. Introduzione

Il progetto ha lo scopo di estrarre dei dati testuali dalle testate economiche "The Economist" e "Financial Times", in particolare gli articoli che sono stati pubblicati dalle stesse nel periodo che va da marzo 2022 a ottobre 2021.
Si procede quindi a operare sugli stessi sia una sentiment analysis, così da determinare la percentuale di articoli positivi e negativi che sono stati pubblicati in un dato mese, sia una topic modelling che restituisca i principali temi trattati. L'output finale del progetto è un dataframe che contenga per ogni mese il valore dell'inflazione, il valore dell'aggregato monetario
M2, i risultati della sentiment analysis e del topic modelling il tutto riferito ai valori aggregati per i paesi dell' EU e agli articoli della sezione Europa. Per quanto riguarda la Sentiment Analysis ci aspettiamo che il sentiment in generale sia molto negativo, data la situazione attuale europea, che da una parte si trova a dover ancora ad avere a che fare con la pandemia Covid-19, mentre dall'altra sente la minaccia di una nuova guerra mondiale. Per quanto riguarda il topic model, ci aspettiamo che gli argomenti dei primi mesi (ottobre-dicembre) riguardino principalmente la questione Covid, le elezioni in Francia e discussioni ancora accese sulla Brexit. Per quanto riguarda la seconda parte ci aspettiamo che il topic della guerra in Ucraina sia totalmente predominante correlato agli aumenti del costo delle risorse energetiche.
Andiamo ora a vedere passo per passo come si sviluppa il codice del nostro progetto.

1. Web Scraping

1.1 Crawler

Il primo passo per la realizzazione del progetto è stato ottenere i dati testuali sui quali effettuare le analisi, in particolare articoli di giornale presi dalle testate online del “The Economist” e del “Financial times” rilasciati nella sola sezione Europa, si è quindi deciso di realizzare come prima cosa un crawler che raccogliesse i collegamenti ai singoli articoli contenuti nella sezione europea dei suddetti siti e che lo facesse per ogni pagina della stessa sezione.

```python
visitate = set()


class Crawler:

    def __init__(self, giornale):
        self._url = ''
        self.giornale = giornale
        self.pagina_ini = int(input(f'\nInserire la pagina di partenza per il {self.giornale}: '))
        self.pagina_fin = int(input(f'\nInserire la pagina finale per il {self.giornale}: '))

    # Definire una proprietà dell'attributo permette di modificare automaticamente l' URL di partenza in accordo alla
    # pagina dalla quale si decide di iniziare l'estrazione
    @property
    def url(self):
        return self._url

    @url.setter
    def url(self, indirizzo):
        self._url = indirizzo+str(self.pagina_ini)

    def ottieni_html(self):
        try:
            if self.url not in visitate:
                visitate.add(self.url)
                print('richiesta ok')
                return requests.get(self.url).content
        except Exception as e:
            return print(e)
```

Si è proceduto quindi a creare una classe crawler che prende come input l’URL della testata al momento della creazione dell’istanza, naturalmente poiché l’analisi deve tenere conto anche del fattore tempo doveva essere possibile selezionare la pagina da cui iniziare l’estrazione dei link e quella finale, infatti nel momento in cui si volessero recuperare articoli rilasciati in periodi precedenti quello attuale l’estrazione deve poter essere effettuata a partire da pagine successive alla prima.

Le proprietà dell'attributo url definite tramite il decoratore permettono di aggiornare automaticamente l'URL del giornale inserendo nello stesso la pagina di partenza selezionata dall'utente.
Il metodo iniziale, dopo aver verificato che la pagina non sia già stata analizzata tramite un set di nome "visitate" passa la stessa alla libreria requests.

```
# Il crawler è comune a entrambe le testate, dopo aver verificato per l'estrazione di quale testata è stato lanciato
# si individuano i selettori CSS all'inteno dei quali sono contenuti i link per i diversi articoli
def estrai_link(self):
    try:
        if self.url.find('https://www.economist.com') != -1:
            soup = BeautifulSoup(self.ottieni_html(), 'html.parser')
            pagina = soup.find(class_='layout-section-collection ds-layout-grid')
            titoli = pagina.find_all('a', class_="headline-link")
            temp_href_economist = [titolo.get('href') for titolo in titoli if titolo.get('href') not in visitate]
            print('estrazione ok')
            return temp_href_economist
        elif self.url.find('https://www.ft.com') != -1:
            soup = BeautifulSoup(self.ottieni_html(), 'html.parser')
            pagina = soup.find_all(class_='o-teaser__heading')
            lst_pagina = list(pagina)
            lst_stringhe = [str(lista) for lista in lst_pagina]
            temp_href_financial = []
            for _ in lst_stringhe:
                content_split = _.split('"')
                for item in content_split:
                    if item.startswith('/content'):
                        temp_href_financial.append(item)
            print('estrazione ok')
            return temp_href_financial
        else:
            print('URL non riconosciuta')
    except Exception as e:
        return print(e)
```

il secondo metodo distingue fra i link di una testata e dell'altra, infatti i siti sono strutturati in maniera diversa e richiedono l'uso di selettori css specifici, dopodiché utilizza la libreria bs4 per estrarre i link e inserirli in una lista.

```
# per poter ciclare fra le diverse pagine per le quali si vuole esplorare i collegamenti è necessario aggiornare
# l'URL cosi che si aggiorni il numero della pagina
def ciclo_pagine(self):
    try:
        href = []
        while self.pagina_ini <= self.pagina_fin:
            print(f"selezionata pagina: {self.pagina_ini}")
            href.append(self.estrai_link())
            self.pagina_ini += 1
            print(self.url)
            if self.url.find('https://www.economist.com') != -1:
                self.url = self.url.replace(str(self.pagina_ini - 1), str(self.pagina_ini))
            elif self.url.find('https://www.ft.com') != -1:
                self.url = self.url.replace(str(self.pagina_ini - 1), str(self.pagina_ini))
        return href
    except Exception as e:
        return print(e)
```

si arriva quindi all'ultimo metodo che racchiude i precedenti in quanto si occupa di aggiornare l'attributo url per poter esplorare tutte le pagine selezionate dall'utente e che per ogni pagina richiama i metodi precedenti dai quali ottiene come output la

lista di collegamenti della specifica pagina a loro volta inseriti nella lista finale href che verrà poi impiegata negli estrattori degli articoli veri e propri.

1.2 Scraper articoli del The Economist

Si è poi proceduto a creare due classi di estrattori specifiche per le testate, partiamo con l'esposizione di quella per il The Economist.
La classe prende in ingresso la lista restituita dal crawler contenente tante liste quante le pagine selezionate a loro volta contenenti i collegamenti per gli articoli.
Considerando come l'estrattore lavori con il collegamento agli articoli e la struttura annidata dell'output del crawler era necessario che la classe fosse iterabile, ecco perché si è deciso di utilizzare il metodo built-in "__iter__" .
Il metodo successivo utilizza la libreria urllib per ottenere un collegamento funzionante dagli href presi dal crawler e li passa alla libreria requests.

```
# questa classe serve per estrarre il contenuto di un articolo dall' Economist, gli attributi di istanza sono
# definiti in modo che sia possibile ciclare all'interno della lista di collegamenti fornita dal crawler
class Estrattore:
    def __init__(self, href=[]):
        self.href = href

    def __iter__(self):
        return self.href

    def __next__(self):
        self._indice += 1
        if self._indice >= len(self.href):
            self._indice = -1
            raise StopIteration
        else:
            return self.href[self._indice]

    i = 0

    # Questo metodo crea una url completa partendo dalla lista di url parziali generati dal crawler e gli applica
    # direttamente la funzione request
    def creazione_richieste(self):
        url_base = 'https://www.economist.com'
        try:
            print('selezionata pagina')
            url_completa = [urllib.parse.urljoin(url_base, articolo) for articolo in self.href[self.i]]
            for _ in url_completa:
                print(_)
            lista_url = [requests.get(x) for x in url_completa]
            print('creato URL per 12 articoli')
            return lista_url
        except Exception as e:
            return print(e)
```

l'ultimo metodo della classe si occuperà di utilizzare bs4 per selezionare tramite selettori css il corpo di testo degli articoli ed inserirlo come una stringa all'interno di un' altra lista, là dove non sia stato possibile utilizzare BeautifulSoup per l'utilizzo

massiccio di javascript si è deciso di optare per la libreria requests-html che carica i contenuti javascript su chronium.

```
    def creazione_soup(self, lista_url=[]):
        try:
            risultato = []
            for pagina in lista_url:
                for articolo in pagina:
                    soup = BeautifulSoup(articolo.content, 'html.parser')
                    if soup.find('div', class_="ds-layout-grid ds-layout-grid--edged layout-article-body"):
                        contenuto = soup.find('div', class_="ds-layout-grid ds-layout-grid--edged layout-article-body")
                        for a in contenuto.select('aside'):
                            a.decompose()
                        for b in contenuto.find(class_="layout-article-links layout-article-promo"):
                            b.decompose()
                        for c in contenuto.find_all('p', class_="article__footnote"):
                            c.decompose()
                        print('articolo estratto')
                        risultato.append(contenuto.text)

                    # Alcuni articoli fanno un utilizzo pesante dei javascript e sono difficilmente estraibili con
                    # Beautiful soup ho quindi creato una sessione su cronium
                    elif soup.select('div', class_='article-text ds-container svelte-o3pylb'):
                        try:
                            contenuto = []
                            temp_url = articolo.url
                            sessione = HTMLSession()
                            r = sessione.get(temp_url)
                            r.html.render(sleep=1, timeout=120)
                            for paragrafo in r.html.find('.article-text.ds-container'):
                                contenuto.append(paragrafo.text)
                            definitivo = ''.join(contenuto)
                            print('articolo estratto')
                            risultato.append(definitivo)
                        except Exception as e:
                            print(e)
                            continue
                    else:
                        print('formato html non riconosciuto')
            return risultato
```

L'ultima fase per estrarre gli articoli del The Economist vede la creazione dell'istanza inserendo come input ciò che viene restituito dal crawler, l'iterazione all'interno della già citata lista di liste e l'inserimento dell'output in un dataframe di pandas successivamente trasformato in un file markdown così da poter effettuare delle prove controllate.

```
lista_url_def = []

Economist = Estrattore(Economist.ciclo_pagine())

for lista in Economist.href:
    lista_url_def.append(Economist.creazione_richieste())
    Economist.i += 1

articoli_TE = Economist.creazione_soup(lista_url_def)

te_df = pd.DataFrame(articoli_TE)
```

1.2 Scraper articoli del Financial Times

Per estrarre gli articoli dal Financial Times si è deciso di utilizzare lo stesso approccio anche se si è rivelato molto più complesso, la parte iniziale del codice è uguale ma ben presto abbiamo capito come non fosse possibile estrarre buona parte dei contenuti a causa del paywall, ma avendo accesso agli stessi dal nostro browser personale abbiamo deciso di utilizzare selenium e geckodriver per aprire il sito caricando i dati contenuti nel nostro profilo personale.

```
# IMPORTANTE: molti degli articoli del FT richiedono una sottoscrizione e non è possibile estrarli con bs4
# per far funzionare questo programma è quindi necessario scaricare geckodriver e copiarlo nella cartella di python
# dopodiché bisognerà caricare il profilo personale di firefox, si noti che il path scritto qua andrà sostituito con
# quello della macchina dove sta girando il programma, il motivo è che si suppone che nel profilo siano salvate le
# credenziali di accesso e si sia installata l'estensione "i don't care about cookies" che rimuove automaticamente
# i pop-up per l'accettazione dei cookies, in linea puramente teorica in assenza di credenziali si potrebbero
# visionare gli articoli anche installando l'estensione "Bypass Paywalls Clean"
profile_path = r'C:\Users\utente\AppData\Roaming\Mozilla\Firefox\Profiles\nt4lo6kh.default-release'
ffOptions = Options()
ffOptions.add_argument("-profile")
ffOptions.add_argument(profile_path)
driver = webdriver.Firefox(options=ffOptions)

# Questo metodo crea una url completa partendo dalla lista di url parziali generati dal crawler e gli applica
# direttamente la funzione request
def creazione_richieste(self):
    url_base = 'https://www.ft.com'
    try:
        print('\nselezionata pagina')
        url_completa = [urllib.parse.urljoin(url_base, articolo) for articolo in self.href[self.i]]
        for item in url_completa:
            print(item)
        lista_url = [requests.get(x) for x in url_completa]
        print('\ncreato URL per 26 articoli')
        return lista_url
    except Exception as e:
        return print(e)
```

Questo approccio ha dunque affiancato quello classico con BeautifulSoup, per il resto il codice è fondamentalmente speculare a quello del The Economist compresa la trasformazione in un dataframe di pandas e successivamente in un file markdown.

1.3 WordCloud

Abbiamo pensato potesse essere utile poter effettuare delle rappresentazioni grafiche per agevolare il processo di analisi e nel contesto di un'analisi testuale il pensiero non poteva non correre al word cloud, abbiamo quindi deciso di implementare una classe che applicasse l'omonima libreria ad un input che poteva essere o direttamente una stringa di testo, nell'eventualità che avessimo deciso di passare direttamente un articolo, possibilità che poi è stata sfruttata nel contesto della topic analysis, o che tale stringa la ottenesse mettendo insieme le linee di un testo, opzione invece rivelatasi utile per un impiego più canonico su tutti gli articoli di un determinato mese.

```
class Cloud:

    def __init__(self, testo):
        self.testo = testo

    # Questo metodo restituisce un testo un minimo ripulito
    def tokenize(self):
        tokens = nltk.word_tokenize(self.testo)
        token_words = [w for w in tokens if w.isalpha()]
        parole_significative = [w for w in token_words if not w in stop]
        joined_words = (" ".join(parole_significative))
        return joined_words

    # Partendo dal testo ripulito si utilizza la libreria wordcloud
    def nuvola(self):
        wordcloud = WordCloud(
            width=3000,
            height=2000,
            background_color='white').generate(self.tokenize())
        fig = plt.figure(
            figsize=(40, 30),
            facecolor='k',
            edgecolor='k')
        plt.imshow(wordcloud, interpolation='bilinear')
        plt.axis('off')
        plt.tight_layout(pad=0)
        plt.show()
        nome = str(input('\nInserire il nome del file: '))
        wordcloud.to_file(nome + '.png')
```

2 Pre-Processing

La parte di pre-processamento dei dati testuali è fondamentale per andare a fare un'analisi significativa. La rimozione dei segni di interpunzione, le stopwords, la stemmatizzazione e la lemmatizzazione sono delle tecniche che permettono, per l'appunto, di migliorare le prestazioni del modello che andiamo ad utilizzare, poiché eliminano tutto quel rumore all'interno del testo, che può essere fuorviante per il modello stesso. In questo senso siamo andati a creare una classe chiamata 'Pre Processing', che avrà diversi metodi che possono essere richiamati:

```
1   import re
2   from nltk.tokenize import word_tokenize
3   from nltk.corpus import stopwords
4   from nltk.stem import PorterStemmer
5   from nltk.stem import WordNetLemmatizer
6
7
8   class PreProcessing(object):
9
10      def __init__(self, text):
11          self.text = text
```

Abbiamo anche importato le librerie che ci serviranno successivamente.

2.1 Punctuation

Per punctuation intendiamo tutti quei simboli che sono inutili ai fini dell'analisi. In questo nostro specifico caso siamo andati ad utilizzare un metodo funzione che prende in ingresso una stringa di testo (nel nostro caso un articolo) e, attraverso due regex (a cui abbiamo passato il comando 'non prendere nulla se non le parole, i numeri e gli spazi vuoti' per la prima e poi 'non prendere i numeri' per la seconda), elimina sia i segni di interpunzione che i numeri e, infine, volge tutto il testo in lowercase.

```
12
13      # Pulisco il testo da tutti i segni di interpunzione attraverso una regex.
14          def cleaner(self):
15              text_cleaned = re.sub(rf"[^\w\s]", "", self.text)
16              text_cleaned2 = re.sub(rf"[\d]", "", text_cleaned).lower()
17              return text_cleaned2
18
```

2.2 Tokenization

Un altro metodo utilizzato è la tokenizzazione. Il metodo prende il testo pulito passato dal metodo precedente, e lo divide in tokens, ossia parole separate:

```
18
19      # Divido il testo in una lista di parole, o anche dette token
20          def tokenization(self):
21              words = word_tokenize(self.cleaner())
22              return words
```

L'output sarà dunque una lista di parole. Questa trasformazione ci permette di fare alcuni lavori sulle parole singole che non potremmo riuscire a fare avendo un testo.

2.3 Stopwords

Un altro passaggio fondamentale è l'eliminazione delle c.d. stopwords, ossia congiunzioni, avverbi, preposizioni, etc., che vanno anch'essi a creare del rumore nella nostra analisi. Il metodo inizializza una variabile 'stop_words', che contiene un tutte le stopwords in lingua inglese.

```
23
24      # Elimino tutte le stopwords, ossia preposizioni, congiunzioni, etc, che servono per legare i
25      # dare un significato alla nostra analisi
26          def remove_stopwords(self):
27              stop_words = set(stopwords.words("english"))
28              filtered_words = [word for word in self.tokenization() if word not in stop_words]
29              return filtered_words
30
```

Successivamente attraverso una list comprehension cicla all'interno della lista di tokens/parole precedentemente definita e, se la parola non si trova all'interno del set di stopwords, la inserisce all'interno di un'altra lista.

2.4 Stemming vs Lemming

Lo stemming e il lemming sono due metodologie che vengono utilizzate principalmente per accorpare le parole molto simili tra loro, che verosimilmente cercano di dirci la stessa cosa. La differenza tra i due è che lo stemming riporta le parole alla sua radice originaria, mentre il lemming al suo lemma originario. In generale lo stemming riesce a contrarre molte più parole e, di conseguenza, avremo una minore capacità di interpretazione, ma i risultati delle analisi saranno più accurati; al contrario il lemming ci premette di avere una maggiore interpretazione, ma dei risultati molto meno accurati.
Inizialmente nella classe abbiamo inserito entrambi i metodi e funzionano in modo analogo:

STEMMING:

Abbiamo utilizzato il PorterStemmer in quanto per quanto riguarda i termini inglesi è più ottimizzato rispetto agli altri.

```
30
31  # Lo stemming mi permette di portare le parole alla radice; uso Porter perché è perfor
32      def stemming_words(self):
33          stemmer = PorterStemmer()
34          stemmed_words = [stemmer.stem(word) for word in self.remove_stopwords()]
35          return stemmed_words
```

LEMMING:

Per quanto riguarda il dizionario abbiamo scelto WordNet arbitrariamente.

Per quanto riguarda il metodo il procedimento è analogo sia per lo stemming che per il lemming. Anche qui viene utilizzata una list comprehension, si cicla all'interno dell'output del metodo precedente, ogni parola viene passata alla funzione 'lemmatize' che esegue il lemming.

```
36
37      # Il lemming permette di riportare la parola al suo lemma originario. E' preferibile allo st
38      # avere un po' più di interpretazione nell'analisi.
39          def lemmatize_words(self):
40              lemmatizer = WordNetLemmatizer()
41              lemmatized_words = [lemmatizer.lemmatize(word) for word in self.remove_stopwords()]
42              return lemmatized_words
43
```

Infine la parola viene inserita in un'altra lista, che sarà l'output del metodo.

Questa classe così definita verrà utilizzata per pulire i nostri testi in seguito, prima di darli in pasto ai modelli.

3 Topic Model

Il secondo tipo di analisi che siamo andati ad svolgere per quanto riguarda i nostri articoli è un Topic Model. Abbiamo scelto questo tipo di modello perché vogliamo andare a sapere quali sono gli argomenti di cui si è discusso di più in un determinato periodo di tempo. In questo caso abbiamo applicato il modello ad ogni mese per 12 mesi. Agli articoli erano stati già estratti in precedenza. L'output del modello consisterà in 3 topic, ossia gli argomenti di cui si è parlato di più, e, per ogni topic, le 5 parole più significative.

Iniziamo con l'importazione delle librerie utili e con la definizione della classe:

```
1   from sklearn.feature_extraction.text import CountVectorizer
2   from sklearn.feature_extraction.text import TfidfTransformer
3   from sklearn.decomposition import LatentDirichletAllocation as LDA
4   from Tokenization_objects import PreProcessing
5
6
7   class ProcessText(object):
8
9       def __init__(self, list_of_text):
10          self.list_of_text = list_of_text
11
```

3.2 Advanced cleaning

Successivamente procediamo con la definizione di un metodo che vada a pulire anche gli articoli; in un primo momento viene richiamata la classe 'PreProcessing' precedentemente creata nel file 'Tokenization Objects' e applicandola a tutti gli articoli.

```
11
12      def text_cleaner(self):
13      # Questa parte mi permette di pulire gli articoli che gli passo, tokenizzarli, e, suc
14          articoli_lemmed = []
15          for articolo in self.list_of_text:
16              articoli_tokenized = PreProcessing(articolo).lemmatize_words()
17              articoli_lemmed.append(articoli_tokenized)
18
```

A questo punto con la variabile 'articoli lemmed' ho una lista di liste in cui ogni lista è un articolo tokenizzato. Abbiamo scelto di inserire ogni articolo tokenizzato in una lista poiché in questo modo gli articoli non vengono mescolati tra loro. Noi però abbiamo necessità di avere una lista di articoli interi e non di parole, in quanto la funzione 'CountVectorizer' prende in ingresso una lista in cui ogni argomento è un testo (in questo caso un articolo), e non una lista di tokens. Perciò abbiamo proceduto con la ricongiunzione di questi ultimi:

```
18
19          articoli_def = []
20          for token in articoli_lemmed:
21              articoli_cleaned = ''
22              for tok in token:
23                  articoli_cleaned += tok + ' '
24              articoli_def.append(articoli_cleaned[:-1])
25          return articoli_def
26
```

In questo modo abbiamo una lista di elementi, in cui ogni elemento è un articolo.

Definisco inoltre un nuovo metodo chiamato 'topic', in cui andrò a processare il testo e, successivamente, passarlo al modello.

3.3 Vectorization

Attraverso la vettorizzazione, ogni articolo viene trasformato in un vettore attraverso la funzione ‘CountVectorize()’. Successivamente viene trasformato in una matrice dei termini.

```
30
31      def topic(self):
32      # Vettorizzazione: trasformo gli articoli in vettori.
33          count_vect = CountVectorizer()
34          x_counts = count_vect.fit_transform(self.text_cleaner())
35
```

3.4 Tf-Idf

Il tf-idf o anche term frequency - inverse document frequency prende la matrice dei termini e verifica la frequenza dei termini all’interno del documento(in questo caso articolo), e la incrocia anche con gli altri documenti(articoli).

```
35
36          tfidf_transformer = TfidfTransformer()
37          x_tfidf = tfidf_transformer.fit_transform(x_counts)
38
```

Questo output verrà successivamente passato all nostro modello LDA.

3.5 LDA: Latent Dirichlet Allocation

Con l'LDA calcoliamo la probabilità che ogni articolo appartenga ad uno dei topic.
Inizialmente definiamo la ‘dimension’, ossia il numero dei topic. Successivamente richiamiamo la funzione LDA impostando come parametro il numero di topic che vogliamo ottenere. Passiamo dunque il risultato del tf-idf al modello LDA.

```
39
40          dimension = 3
41          lda = LDA(n_components=dimension)
42          lda_array = lda.fit_transform(x_tfidf)
43          print(lda_array)
44
```

L’output è stato inserito all’interno della variabile lda_array e facendo un print possiamo vedere graficamente:

```
[[0.0222221  0.9555558  0.0222221 ]
 [0.02045359 0.95909282 0.02045359]
 [0.01648888 0.96702223 0.01648888]
 [0.02115773 0.95768455 0.02115773]
 [0.02149946 0.95700108 0.02149946]
```

E’ un array in cui le colonne sono i topic e le righe gli articoli. Ciò che rappresenta è la probabilità che un articolo appartenga ad uno dei 3 topic. In questo caso c’è una probabilità molto elevata (tra il 95% e 96%) che gli articoli appartengano al secondo topic.

In questa ultima parte si definisce il numero di topic (components) e il numero delle features (il dizionario delle parole).
Successivamente si va a ciclare per il numero dei topic 'len(components)' e per ogni componente si vanno a prendere le parole più significative per ogni topic, in ordine di frequenza:

```
44
45          components = [lda.components_[i] for i in range(len(lda.components_))]
46          features = count_vect.get_feature_names()
47          important_words = [sorted(features, key=lambda x: components[j][features.index(x)], reverse=True)[:5] for j in
48                             range(len(components))]
49
50          return important_words
```

L'output sarà dunque una lista di liste in cui ogni lista è un topic e gli elementi sono le parole più importanti per quel topic:

```
[['mr', 'ukraine', 'russia', 'russian', 'eu'], ['gamers', 'tournament', 'valorant', 'esports', 'protein']
```

In questo caso ho tagliato un topic per questione di spazio, ma è chiaro che nel primo topi si stia parlando della situazione attuale tra russia e ucraina, mentre nel secondo si fa riferimento ad un torneo di un videogioco chiamato ‘valorant’.

3.6 Rappresentazione grafica

Per rappresentare graficamente i topic derivanti dalla precedente analisi, abbiamo definito una funzione ‘mostra_topic’. La funzionzione inizializza una lista e una stringa, apre il file (in questo caso riguarda gli articoli del mese di Marzo), inserisce gli articoli nella lista precedentemente inizializzata e infine richiama il metodo ‘topic’ precedentemente descritto e gli vengono passati gli articoli estratti. Il metodo topic viene richiamato per 10 volte e, per ogni iterazione, tutte le parole che escono fuori dai topic vengono inserite all’interno della stringa inizializzata all’inizio e gli viene applicata una WordCloud. L’output è una wordcloud dei topic più frequenti.

```
64  def mostra_topic():
65      # MARZO
66      print('\necco i topic per il mese di marzo: ')
67      articoli = []
68      bag = ''
69      with open('6. Marzo_22.md', encoding='utf8') as f:
70          for line in f:
71              articoli.append(line)
72      articoli_def = ProcessText(articoli)
73      print(articoli_def.topic())
74      for i in range(1, 10):
75          topic = articoli_def.topic()
76          for lista in topic:
77              tmp = ' '.join(lista)
78              bag = bag+' '+tmp
79      crea_cloud(bag)
80
```

4 Sentiment Analysis

Il primo tipo di analisi che siamo andati a svolgere sui nostri articoli è una Sentiment Analysis di tipo binomiale, ossia abbiamo cercato di capire se ogni singolo articolo fosse positivo o negativo. Abbiamo effettuato l’analisi per ogni singolo mese, in un arco temporale di 6 mesi. In questo modo possiamo andare a vedere la variazione della sentiment della popolazione rispetto alla situazione europea nell’arco dell’ultimo anno. Quello che ci aspettiamo è che in alcuni periodi dell’anno il sentiment sia più positivo, mentre in altre più negativo (per esempio la guerra russia-ucraina).

Iniziamo importando le librerie utili. Alcune sono inutilizzate in quanto quella parte del codice è stata commentata:

```
import nltk
import random
from nltk.corpus import stopwords
from nltk.classify.scikitlearn import SklearnClassifier
from sklearn.naive_bayes import MultinomialNB, BernoulliNB
from sklearn.linear_model import LogisticRegression, SGDClassifier
from sklearn.svm import SVC, LinearSVC, NuSVC
from nltk.classify import ClassifierI
from statistics import mode
from sklearn.model_selection import train_test_split
from Tokenization_objects import PreProcessing

```

Successivamente apriamo e puliamo i file due file che useremo per l'addestramento del nostro modello. I file in questione sono file di recensioni: uno con recensioni positive, l'altro con recensioni negative.

```

# Apertura dei file base con cui verrà successivamente addestrato il modello.
short_positive_raw = open('positive.txt', 'r').read()
short_negative_raw = open('negative.txt', 'r').read()

# Pulizia dei due file richiamando il metodo cleaner
short_positive_cleaned = PreProcessing(short_positive_raw).cleaner()
short_negative_cleaned = PreProcessing(short_negative_raw).cleaner()

```

Abbiamo utilizzato come addestramento delle recensioni, ma ci sono altri modi che possono essere utilizzati per addestrare i modelli (per esempio vengono scelti manualmente degli articoli già positivi e altri già negativi)

4.1 Featurization

Successivamente inizializziamo due liste che ci serviranno tra poco.
Inizializziamo inoltre un'altra lista che contiene i tag grammaticali e che ci permetterà di filtrare e successivamente prendere solo quei tipi: 'J' è aggettivi, 'V' per verbi e 'N' per i sostantivi:

```
22  # creazione di due liste
23  documents = []
24  all_words = []
25
26  # lista dei 'tag' delle parole che saranno estratte dai documenti.
27  allowed_word_types = ['J', 'V', 'N']
28
```

Successivamente questa parte del codice itera nel documento delle recensioni positive, inserisce nella lista 'documenti' una tupla in cui il primo valore è l'articolo e il secondo è la classificazione. Parallelamente le recensioni vengono tokenizzate, rimosse le stopwords, e lemmatizzate e inserite nella varaibile 'words'. In seguito viene chiamata la funzione 'pos_tag' dalla libreria nltk e gli viene passata la variabile 'words'. La funzione restituisce il tag per ogni parola. Infine si cicla nei tag e se il tag corrisponde ai tag consentiti (dichiarati precedentemente nella variabile 'allowed_word_types'), la parola viene inserita nella lista 'all_words', anch'essa inizializzata in precedenza. Stesso procedimento anche con le negative.

```
35  for p in short_positive_cleaned.split('\n'):
36      documents.append((p, 'pos'))
37      words = PreProcessing(p).lemmatize_words()
38      pos = nltk.pos_tag(words)
39      for w in pos:
40          if w[1][0] in allowed_word_types:
41              all_words.append(w[0])
42
43  # Qui succede la stessa cosa ma con il file delle recensioni negative.
44  for p in short_negative_cleaned.split('\n'):
45      documents.append((p, 'neg'))
46      words = PreProcessing(p).lemmatize_words()
47      pos = nltk.pos_tag(words)
48      for w in pos:
49          if w[1][0] in allowed_word_types:
50              all_words.append(w[0])
51
```

In seguito La lista 'all words' viene passata alla funzione FreqDist della libreria nltk, che genera la frequenza di ogni parola all'interno della lista. Può essere chiamato il metodo 'most_common', che restituisce in ordine decrescente una lista di tuple in cui il primo elemento è la parola e il secondo la frequenza della parola (quante volte la parola appare nella lista). Se invece viene passata una parola, restituisce la frequenza di quest'ultima.

```
57  all_words_fd = nltk.FreqDist(all_words)
58  print(all_words_fd.most_common(10))
59  print(all_words_fd['good'])
60
```

Dentro questa variabile viene inserita la lista di tutte le parole prese una sola volta:

```
61
62  # Dentro questa variabile viene inserita la lista di tutte le parole prese una
63  word_feature = list(all_words_fd.keys())
```

Poi definiamo un'altra funzione che prende in ingresso un documento, lo pulisce e vengono lemmatizzate le parole richiamando la classe PreProcessing. Successivamente viene inizializzato un dizionario in cui vengono inserite come chiavi le parole del della lista di tutte le parole e come valore un booleano (True se la parola è presente nel documento, False viceversa). La funzione restituisce questo dizionario.

```
67  def find_features(document):
68      wordz = PreProcessing(document)
69      wordz_def = wordz.lemmatize_words()
70      features = {}
71      for word in word_feature:
72          features[word] = (word in wordz_def)
73      return features
74
```

Infine iteriamo all'interno di 'documents' (che ricordiamo essere una lista di tuple), e, utilizzando una list comprehension, viene inserita all'interno della variabile 'featuresets' una tupla che ha come primo elemento l'output della funzione 'find_features' a cui viene passato l'articolo, mentre come secondo elemento il tag relativo all'articolo (pos o neg):

```
76  featuresets = [(find_features(article), tag) for (article, tag) in documents]
77  random.shuffle(featuresets)
```

Featuresets sarà a tutti gli effetti il nostro dataset che verrà utilizzato per l'analisi.

4.2 Subsetting

Andremo ora a preparare il dataset per in modo che possa essere passato ai modelli. Ci sono vari approcci al setting del set dei dati per l'addestramento e per il testaggio dei modelli. In questo caso abbiamo utilizzato direttamente l'approccio del training e del validation set, in cui viene diviso il dataset secondo una certa percentuale definita dall'analista. In questo caso scegliamo 50/50 in quanto andremo ad utilizzare i modelli per classificare dei dati che non sono recensioni, ma tweet e articoli. Se utilizzassimo un training set troppo grande rischieremmo un overfitting, ossia che il modello sia molto performante con i dati che gli abbiamo dato noi, ma poi se ne dovessimo passare altri nuovi, performerebbe in modo mediocre.

```
93  # Validation set approach
94  training, validation = train_test_split(featuresets, test_size=0.5, random_state=5)
95
```

4.3 Modelli

Abbiamo utilizzato diversi modelli per la classificazione e le prestazioni di questi ultimi si aggirano tutte intorno al 75%. Alla luce delle prestazioni, decidiamo di procedere utilizzando il modello di Naive-Bayes in quanto risulta essere il miglior compromesso tra precisione dell'analisi e sforzo computazionale.

Nella prima riga viene richiamato il classificatore e gli viene passato il training set in modo che venga addestrato con la funzione 'train'. Successivamente è possibile vedere la percentuale di accuratezza tramite la funzione 'accuracy', a cui questa volta viene passato il validation set. L'ultima riga permette di vedere le parole più informative per il modello.

```
102 ### Naive Bayes ###
103 NBclassifier = nltk.NaiveBayesClassifier.train(training)
104 print('Original_Naive_Bayes_accuracy:', (nltk.classify.accuracy(NBclassifier, validation))*100)
105 NBclassifier.show_most_informative_features(20)
```

Tutti gli altri modelli utilizzati sono stati commentati all'interno del codice, nel caso si vogliano utilizzare è necessario decommentare la parte del codice interessata.
I modelli in questione sono: Il multinomial Naive-Bayes e il Bernoulli Naive-Bayes, due varianti del modello Naive Bayes; la regressione logistica, lo Stochastic Gradient Descent, l' SVM, SVM lineare e il Nu SVM.

In seguito viene costruita una classe che serve per 'votare' quale sia il responso della sentiment (se positivo o negativo). Prende in ingresso da 1 a n

modelli/classificatori, e il metodo classify ritorna il sentiment, mentre il metodo confidence ritorna la sicurezza/accuratezza della decisione

```
168 class VoteClassifier(ClassifierI):
169
170     def __init__(self, *classifiers):
171         self._classifiers = classifiers
172
173     def classify(self, features):
174         votes = []
175         for c in self._classifiers:
176             v = c.classify(features)
177             votes.append(v)
178         return mode(votes)
179
180     def confidence(self, features):
181         votes = []
182         for c in self._classifiers:
183             v = c.classify(features)
184             votes.append(v)
185
186         choice_votes = votes.count(mode(votes))
187         conf = choice_votes / len(votes)
188         return conf
189
```

Qui viene richiamata la classe precedentemente descritta. Può prendere in input uno o più classificatori. Nel caso siano di più ritorna la moda della classificazione, dunque la decisione più frequente. Anche qui il fatto di utilizzare più di un classificatore è una scelta dell'analista: Per avere una maggiore interpretabilità è sempre meglio utilizzare un solo modello alla volta. Nel caso in cui interessi una maggiore precisione nella classificazione, e di conseguenza una maggiore capacità di predizione. Anche qui è sempre questione di trade-off

```
197 voted_classifier = VoteClassifier(NBclassifier)
```

Nel caso si vogliano aggiungere altri classificatori bisogna aggiungere una virgola e aggiungere il classificatore.

4.4 Sentiment

Infine questa funzione è ciò che verrà chiamato nel momento in cui si importa questo file/modulo. Alla funzione verrà passato del testo (che può essere un articolo o un

tweet), viene featurizzato, e ci restituisce non solo la sentiment del testo (quindi se è positivo o negativo); ma anche l'accuratezza con cui viene decretata la scelta. L'output viene restituito sotto forma di tupla.

```
204     def sentiment(text):
205         feats = find_features(text)
206         return voted_classifier.classify(feats), voted_classifier.confidence(feats)
```

Questo output viene richiamato all'interno di un nuovo file chiamato 'sentiment articles', a cui verranno effettivamente passati i file di testo.
Iniziamo come sempre importando le librerie e i moduli:

```
1   from topic_modeling_objects import ProcessText
2   import Sentiment_mod_objects as s
3   import matplotlib.pyplot as plt
4   import numpy as np
5
```

Viene definita la classe SentimentAnalysis che prende come input una lista di file che nel nostro caso specifico corrispondono ai mesi, tra gli atrributi vengono invece inizializzate alcune liste che serviranno come input per i metodi successivi:

```
class SentimentAnalysis:
    def __init__(self, lista):
        self.lista = lista
        self.articoli = []
        self.articoli_puliti = []
        self.articoli_sentiment = []

    positivi = pd.DataFrame({'% positivi': []})
    negativi = pd.DataFrame({'% negativi': []})
```

Viene definito un metodo pulizia articoli a cui vengono passati gli articoli precedentemente estratti dai file e vengono puliti richimando il metodo 'text_cleaner' della classe 'ProcessText'

```
53
54      def pulizia_articoli(self):
55          self.articoli_puliti = ProcessText(self.articoli).text_cleaner()
56
```

Creiamo una funzione che prende in ingresso una lista di articoli, per ogni articolo crea una tupla che ha come primo elemento l'articolo e come secondo elemento una

tupla (richiamata dal modulo 'sentiment_mod_objects' e a cui viene passato ogni articolo) che ha come primo elemento il sentiment dell'articolo e come secondo elemento l'accuratezza del sentiment.

```
60
61      def sentiment(self):
62          for art in self.articoli_puliti:
63              self.articoli_sentiment.append((art, s.sentiment(art)))
64
```

In seguito viene definita un'altra funzione che prende in ingresso la lista di tuple restituita dalla funzione precedente; inizializza due contatori, itera all'interno della tupla andando a ricercare il risultato del sentiment per ogni articolo. A questo punto se l'articolo risulta positivo, aumenta il counter dei positivi, viceversa dei negativi. Infine restituisce i due valori.

```
18  def counter_pos_neg(tupla_di_tuple):
19      neg_count = 0
20      pos_count = 0
21      for art, sent in tupla_di_tuple:
22          if sent[0] == 'pos':
23              pos_count += 1
24          elif sent[0] == 'neg':
25              neg_count += 1
26
27      return pos_count, neg_count
```

L'ultimo metodo è quello principale che richiama i precedenti, prende la lista di mesi con cui si è creata l'istanza e applica ad ogni elemento sottostante i metodi precedenti, infine realizza un grafico a barre della percentuali di risultati con sentiment negativa e positiva per ogni mese e salva il risultato in un dataframe che servirà poi per l'inserimento finale.

```
def esegui_sentiment(self):
    for elemento in self.lista:
        print(f'Sto lavorando su {elemento}')
        with open(elemento, encoding='utf8') as f:
            for line in f:
                self.articoli.append(line)
            self.pulizia_articoli()
            self.sentiment()
            risultato = self.counter_pos_neg()
            perc_pos = round(np.divide(risultato[0], risultato[0] + risultato[1]) * 100)
            perc_neg = round(np.divide(risultato[1], risultato[0] + risultato[1]) * 100)
```

```
            # memorizzo nelle corrispondenti righe del dataframe tramite gli attributi di classe 'positivi' 'negativi'
            plt.bar('positive', perc_pos, label=f'positive:  {perc_pos}%')
            plt.bar('negative', perc_neg, label=f'negative: {perc_neg}%')
            plt.title(f'Sentiment per il file {elemento}')
            plt.xlabel('Sentiment')
            plt.ylabel('% of Articles')
            plt.legend(loc='upper right')
            plt.show()
            self.positivi.loc[len(self.positivi.index)] = perc_pos
            self.negativi.loc[len(self.negativi.index)] = perc_neg
    df_finale = pd.concat([temp, self.positivi, self.negativi], axis=1)
    df_finale.to_csv('DataFrame_progetto.csv')
```

5. Main.py

Il file main è il crocevia da cui chiamare tutti i moduli necessari all'occorrenza, prevede un minimale processo di interfaccia con l'eventuale utente che permette a quest'ultimo di svolgere le funzioni che più gli interessano nascondendo allo stesso tempo i processi sottostanti che sono contenuti in moduli esterni.

```
if __name__ == '__main__':
    try:
        selettore = int(input("selezionare l'attività da svolgere:\n"
                              "1. Estrarre gli articoli\n"
                              "2. Generare una wordcloud per gli articoli\n"
                              "3. Applicare il topic modelling agli articoli\n"
                              "4. Applicare la sentiment analysis agli articoli\n"
                              ">> "))
        if selettore == 1:
            import pandas as pd
            from ft_lettura import ft_df
            from economist_lettura import te_df

            crea_file()

        elif selettore == 2:
            from word_cloud import crea_cloud

            nome_file = str(input('Scegliere il file per il quale si vuole visionare la wordcloud: '))
            crea_cloud(nome_file + ".md")

        elif selettore == 3:
            from topic_modeling_objects import mostra_topic

            mostra_topic()

        elif selettore == 4:
            from sentiment_articles import Sentiment

            Sentiment.esegui_sentiment()

    except ValueError:
        print('selezionare con un numero')
```

6. Conclusioni

I risultati dell'analisi svolta sono in generale vicini a ciò che ci aspettavamo, ma con alcune imprecisioni e differenze. Per quanto riguarda la Sentiment Analysis, la sentiment è fortemente negativa per tutto il periodo analizzato. Ci aspettavamo che ad ottobre la percentuale di negatività fosse più bassa, andando a crescere fino a Marzo, in cui avrebbe toccato il picco massimo. C'è sicuramente del margine di miglioramento dei modelli, soprattutto per quanto riguarda l'addestramento di quest'ultimo: noi abbiamo utilizzato dei file di recensioni per l'addestramento, in quanto non è stato possibile reperire degli articoli già catalogati come positivi e negativi. L'implementazione consisterebbe nel classificare manualmente una parte degli articoli, in modo da poterli poi utilizzare come input per l'addestramento.
Per quanto riguarda il Topic Model, i topic individuati dal modello sono molto vicini a quelli da noi individuati, ma può capitare che alcune volte il modello estragga dei topic molto marginali. Infine, come previsto, il topic della guerra in Ucraina viene sempre estratto dal modello.