

BIG DATA

Documentazione progetto "Authorship attribution"

Progetto d'Esame

Giacomo Gatto - VR478050

Luigi Hu - VR474722

2022-2023

Indice

1	Introduzione al progetto: Authorship attribution	2
2	Spark	3
2.1	RDD - Resilient Distributed Dataset	3
3	Machine Learning	5
4	Dataset	6
5	Approccio basato sulle parole più frequenti usate dall'autore	8
5.1	Variabili di supporto	8
5.2	Funzione clearText()	9
5.3	Funzione wordCount()	10
5.4	Salvataggio risultati della funzione wordCount()	11
5.5	Training	13
5.6	Testing	16
5.6.1	Metodo 1: Basato sulle parole in comune	19
5.6.2	Metodo 2: Usando la similarità del coseno	20
5.6.3	Metodo 3: Usando il coefficiente di similarità di Jaccard	23
5.7	Testing degli altri 2 indicatori	25
6	Approccio basato sull'utilizzo di algoritmi di Machine Learning	27
6.1	SVM	27
6.2	Random Forest	28
6.3	Classificatore di Bayes	29
6.4	Training	29
6.5	Testing	32
7	Risultati	34
8	Conclusioni	36

1 Introduzione al progetto: Authorship attribution

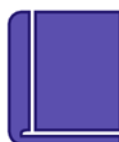
L'obiettivo del progetto assegnato consiste nel riconoscere l'autore di un libro anonimo dato il suo testo. In generale questo problema è stato studiato molto nell'ambito dei Big Data e, nel tempo, si sono individuate parecchie modalità diverse per risolverlo.

In questo progetto si cerca di implementare delle soluzioni efficaci, per la risoluzione di tale problema, attraverso l'utilizzo delle nozioni apprese durante il corso. Si è partiti con la creazione di un semplice dataset, composto da 100 libri (10 per ogni autore), che verrà poi diviso in 2 parti: una parte per il training e una per il testing. Il training consiste nell'estrarre le caratteristiche stilistiche dei vari autori, mentre il testing consiste nell'utilizzare tali caratteristiche per permettere di effettuare l'identificazione dell'autore dei testi anonimi e verificare se tali risultati siano corretti o meno.

Per fare ciò, si è usufruito di Apache Spark. Si tratta di un framework che permette di manipolare grandi quantità di dati (nel nostro caso sono le parole dei vari libri) in modo più semplice e veloce rispetto alle altre tecnologie disponibili. A livello implementativo, il linguaggio di programmazione scelto è Python.

Si sono utilizzati diversi tipi di classificatori che verranno spiegati nel dettaglio nelle sezioni successive. Alcuni di loro sono basati sull'utilizzo delle parole più frequenti di un autore, altri sono basati sul Machine Learning.

Unknown Documents



Who is the author?

Author 1



Author 2



Author 3



Author 4



2 Spark

Spark è un framework che permette l'elaborazione di dati distribuiti in tempo reale. E' stato creato in quanto l'approccio Map-Reduce di Hadoop aveva alcune limitazioni:

- Il paradigma Map-Reduce risulta essere molto rigido per elaborazioni più complesse.
- Se ci sono più fasi, ogni volta che si arriva alla conclusione di una fase bisogna copiarne i dati su hdfs. Il fatto di doverli memorizzare rallenta molto il processo.

Con Spark si prevede l'utilizzo di una memoria più veloce per evitare il collo di bottiglia della velocità di lettura/scrittura su hdfs. Quindi prima si caricano da hdfs i dati da analizzare in memoria ram (più veloce) e poi si eseguono le varie iterazioni. In questo contesto, però, bisogna gestire la volatilità della memoria ram. Un'applicazione Spark ha 3 parti fondamentali:

- Driver: definisce e coordina il lavoro da eseguire,
- Cluster Manager: assegna risorse e gestisce il cluster,
- Esecutori o nodi: eseguono il lavoro di elaborazione dati.

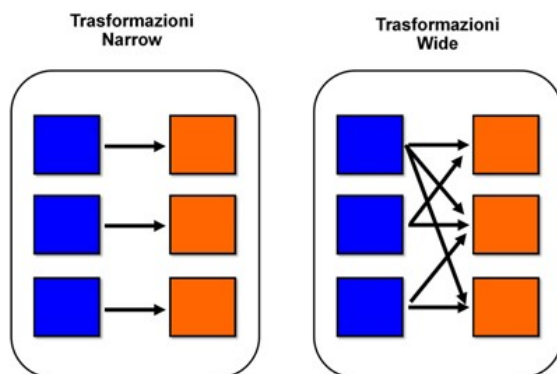
Ovviamente i dati si trovano sulla ram, non su memorie persistenti, e le elaborazioni avvengono su sistemi distribuiti. Per rappresentare questi dati si usano gli RDD (Resilient Distributed Dataset).

2.1 RDD - Resilient Distributed Dataset

Gli RDD sono collezioni distribuite di dati, partizionati su più macchine di un cluster, che vengono archiviati nella memoria principale. Gli RDD vengono manipolati attraverso trasformazioni e azioni.

- **Trasformazioni:** sono operazioni che partendo da un RDD creano un RDD modificato. Si dividono in:
 - **Trasformazioni narrow:** la trasformazione fa sì che il risultato dipenda unicamente da un solo RDD. Alcuni esempi sono: map, flatMap, filter, sample, union..

- **Trasformazioni wide:** un blocco di un RDD non dipende solo da un altro blocco, ma da altri blocchi dell’RDD. In caso di guasto per ricostruire l’RDD perduto ho bisogno dell’intero RDD di partenza. Di solito nella catena di elaborazione si fanno dei checkpoint in ram. Alcuni esempi sono: `reduceByKey`, `groupByKey`...
- **Azioni:** partendo da un RDD ne ottengono una cosa diversa. Ne sono esempi: `reduce`, `collect`, `take`, `count` ecc. . . .



3 Machine Learning

Il Machine Learning è un'area dell'intelligenza artificiale che si occupa di creare modelli che permettono alla macchina di apprendere e migliorare dalle esperienze passate. Questo processo tipicamente comporta questi passaggi:

1. **Raccolta dati:** quantità e qualità dei dati sono importanti per il successo del modello,
2. **Selezione del modello:** scegliere l'algoritmo più adatto per il tipo di applicazione,
3. **Training del modello:** utilizzo dei dati di training per riconoscere pattern o relazioni nei dati,
4. **Valutazione del modello:** valutare l'efficacia dei dati usando i dati di test,
5. **Implementazione:** una volta che il modello è stato allenato e validato può essere implementato in un'applicazione.

Nell'ambito del riconoscimento dei testi anonimi, possiamo usare il machine learning per effettuare il training su testi noti di autori diversi, ottenendo così un modello che potrà essere usato per identificare l'autore di appartenenza dei testi anonimi. Alcuni esempi di algoritmi che possono essere usati in questo contesto sono:

- **SVM:** La Support Vector Machine cerca di trovare il miglior iperpiano di separazione tra classi di dati in uno spazio multidimensionale, massimizzando la distanza tra i punti più vicini di ciascuna classe (support vectors),
- **Random Forest:** Crea un insieme di alberi decisionali e combina le loro previsioni per migliorare la classificazione,
- **Classificatori Bayesiani:** Classificatore semplice ed efficace basato sul teorema di Bayes.

Nell'ambito del progetto sono stati scelti proprio questi 3 algoritmi. Nelle sezioni successive verranno messi a confronto i vari risultati ottenuti.

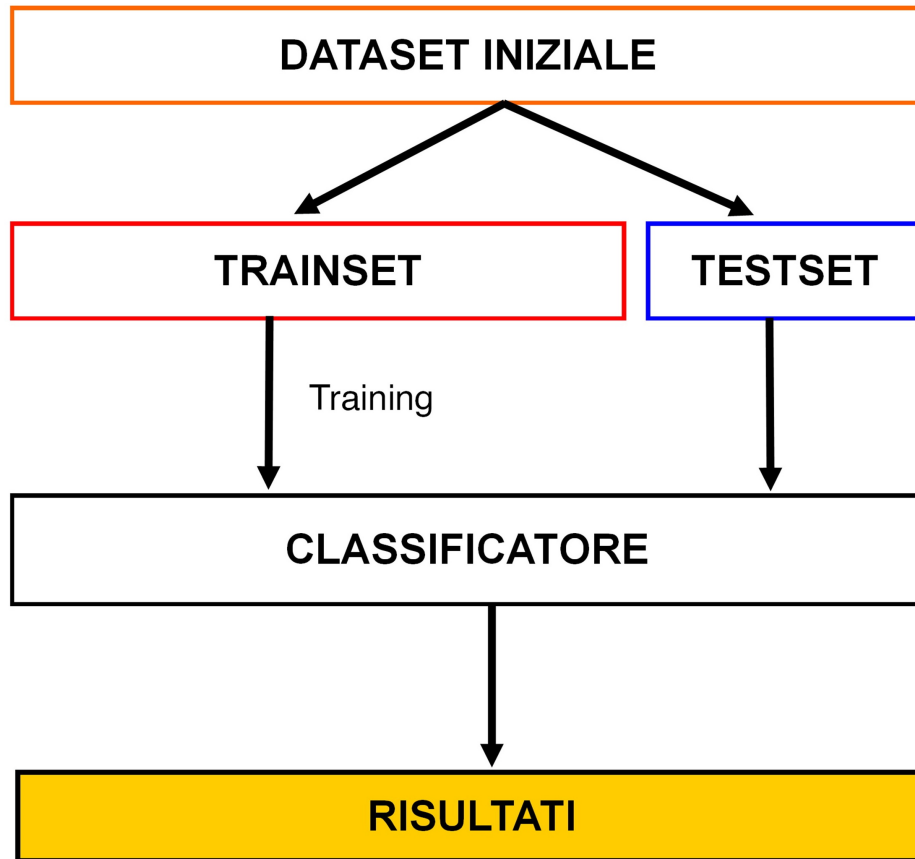
4 Dataset

Il dataset utilizzato è stato creato prendendo i testi da www.gutenberg.org. Cos'è gutenber? Project Gutenberg è una libreria online di eBook gratuiti che vanta nella propria collezione oltre 100.000 ebook di oltre 100 autori diversi. Si è scelto di prendere i testi dei seguenti 10 autori:

- Bret Harte
- Charles Dickens
- Daniel Defoe
- Edith Wharton
- Jane Austen
- Joseph Conrad
- Lewis Carroll
- Louisa May Alcott
- Voltaire
- William Shakespeare

Per ogni autore si sono presi 10 testi che andranno a formare il dataset del progetto, composto da 100 testi totali.

Il dataset verrà poi diviso, con una suddivisione 70-30, in 2 parti: una per il training (training-set) e l'altra per il testing (testing-set). Il training-set sarà composto da 70 testi totali (7 per ogni autore) mentre il testing-set sarà composto da 30 testi totali (3 per ogni autore). Dal training-set verrà creato, dopo una fase di training, un classificatore il quale userà i testi del testing-set per ottenere i risultati del progetto.



5 Approccio basato sulle parole più frequenti usate dall'autore

Uno degli approcci, al problema descritto in precedenza, consiste nell'estrarre le caratteristiche stilistiche dei vari autori per poter ottenere un indicatore che li rappresenti e, successivamente, usare questi indicatori per cercare di identificare la paternità dei testi anonimi. Si è scelto di rappresentare gli autori attraverso l'utilizzo di un vettore composto da un insieme di parole da lui più utilizzate. Ci sono molti modi per implementare questo concetto ed è importante fare alcune considerazioni:

- Si possono scegliere diverse tipologie di parole per rappresentare un autore,
- Non tutte le parole hanno la stessa importanza,
- E' importante considerare la frequenza relativa e non quella assoluta delle parole.

I testi degli autori da analizzare verranno elaborati con Spark in quanto rappresentano una grande quantità di dati.

5.1 Variabili di supporto

Nel progetto sono state usate alcune variabili come **function_words** e **punctuations_char** per supportare le operazioni di pulizia del testo.

Function_words è una lista che contiene parole che svolgono principalmente una funzione grammaticale o strutturale all'interno di una frase. Si tratta quindi di parole largamente impiegate e che forniscono poco significato lessicale. Alcuni esempi di function words sono:

- Articoli: a, an, the, ecc..
- Preposizioni: in, on, at, by, with, for, ecc. . .
- Pronomi: I, you, he, she, it, ecc. . .
- Congiunzioni: and, but, or, so, for, ecc..
- Avverbi di luogo
- Avverbi di tempo

La lista contiene in tutto 512 parole inglesi in quanto i testi analizzati sono in lingua inglese.

Punctuations_char, invece, rappresenta una lista che contiene caratteri di punteggiatura/speciali che, a loro volta, forniscono poco significato lessicale. Qualche esempio di tali caratteri sono: il punto, la virgola, il trattino ecc.

Altre variabili di supporto per il funzionamento del codice:

- **sc**: rappresenta lo spark context e serve per interagire con Spark,
- **my_dir**: è il path del dataset utilizzato,
- **readFromSavedFile**: variabile booleana per indicare se si vuole ricreare gli rdd (False) o se si vuole leggerli da quelli già salvati (True),
- **authors**: rappresenta la lista degli autori,
- **n_authors**: indica il numero degli autori,
- **booksName**: si tratta di un vettore che contiene, per ogni autore, la lista dei nomi dei suoi libri presenti nel dataset.

5.2 Funzione clearText()

Si tratta di una funzione che effettua una pulizia preliminare del testo:

- **textFile()**: legge il testo dal file path dato in input e lo trasforma in un RDD,
- **filter()**: toglie le righe vuote,
- Serie di **map()**: separa i caratteri di punteggiatura/speciali dal resto delle parole.

Alla fine viene ritornato l’RDD risultante.

```
1 #Function to clear the text given in input
2
3 def clearText(file_path):
4     return sc.textFile(file_path).filter(bool) \
5         .map(lambda w: w.replace(".", " . ")) \
6         .map(lambda w: w.replace(",", " , ")) \
7         .map(lambda w: w.replace(";", " ; ")) \
8         .map(lambda w: w.replace(":", " : ")) \
9         .map(lambda w: w.replace("!", " ! ")) \
10        .map(lambda w: w.replace("?", " ? ")) \
```

```

11      .map(lambda w: w.replace("'", " ' ")) \
12      .map(lambda w: w.replace(" '", "' ")) \
13      .map(lambda w: w.replace("''", "' ' ")) \
14      .map(lambda w: w.replace(" '", "' ")) \
15      .map(lambda w: w.replace(" '", "' ")) \
16      .map(lambda w: w.replace("-", " - ")) \
17      .map(lambda w: w.replace("_", " _ ")) \
18      .map(lambda w: w.replace("{", " { ")) \
19      .map(lambda w: w.replace("}", " } ")) \
20      .map(lambda w: w.replace("[", " [ ")) \
21      .map(lambda w: w.replace("]", " ] ")) \
22      .map(lambda w: w.replace("(", " ( ")) \
23      .map(lambda w: w.replace(")", " ) ")) \
24      .map(lambda w: w.replace("*", " * "))

```

5.3 Funzione wordCount()

La funzione `wordCount()` viene usata per effettuare il conteggio delle parole presenti in un testo dato in input. Esso prende il testo, passato in input come path, ed esegue una serie di trasformazioni quali:

- **clearText()**: funzione di pulizia del testo citata precedentemente,
- **flatMap()**: ogni riga viene trasformata in un insieme di parole che andranno a rappresentare gli oggetti della nuova RDD restituita,
- i 2 **map()**: il primo trasforma tutte le parole in minuscolo, il secondo mappa ogni parola in una tupla nel formato seguente: (parola, 1),
- **reduceByKey()**: raggruppa le tuple per chiave (in questo caso la parola) e ne somma i valori associati,
- **filter()**: viene tolta la tupla con chiave "" (ovvero una parola vuota),
- **sortBy()**: esegue l'ordinamento decrescente delle tuple per valore (ovvero dalle parole più frequenti a quelle meno)

Viene poi calcolato e aggiunto il numero totale delle parole sfruttando l'uso di:

- **collect()**: trasforma un RDD in una lista,
- **parallelize()**: trasforma una lista in un RDD.

```

1 #WordCount returning a list of words
2
3 def wordCount(file_path):
4     words = clearText(file_path) \
5         .flatMap(lambda line: line.split(" ")) \
6         .map(lambda w: w.lower()) \
7         .map(lambda w: (w, 1)) \
8         .reduceByKey(lambda v1, v2: v1 + v2) \
9         .filter(lambda w: w[0] != "") \
10        .sortBy(lambda w: -w[1])
11
12     result = words.collect()
13
14     # Calculate total words of the book given in input
15     total_words = 0
16     for t in result:
17         if t[0] not in punctuations_char:
18             total_words += t[1]
19
20     # Add the total word to the list of words
21     result.append(('total_words', total_words))
22
23     return sc.parallelize(result)

```

5.4 Salvataggio risultati della funzione wordCount()

Si è deciso di salvare su file system i risultati della funzione wordCount() in quanto il suo calcolo è molto oneroso. In questo modo, il calcolo viene effettuato solo la prima volta, permettendo un'esecuzione più veloce del codice le volte successive.

Per prima cosa, si controlla la variabile **readFromSavedFile** per verificare se effettuare o meno la funzione **wordCount()** per poi, eventualmente, salvare il risultato su file system tramite il metodo **saveAsTextFile()**.

```

1 #Saving word count RDD for every book for all authors
2
3 if not readFromSavedFile:
4
5     for i in range(n_authors):
6         # Obtain the path of author directory
7         tmp_filename = os.path.join(my_dir, authors[i])
8         print(f"Word count of {authors[i]}’s books.")
9
10        for j in range(len(booksName[i])):
11            # Obtain the path of books given the path of an
12            author

```

```

12         input_filename = os.path.join(tmp_filename,
booksName[i][j])
13
14         print(f"Start - Word count of book: {booksName[i][
j]}\n")
15         wordsOfABook = wordCount(input_filename)
16         print(f"End - Word count of book: {booksName[i][j
]}\n")
17
18         print(f"Start - Saving word count of book: {
booksName[i][j]}\n")
19         wordsOfABook.saveAsTextFile("./RDD/Books/
SingleBooks/" + authors[i] + "/" + booksName[i][j])
20         print(f"End - Saving word count of book: {
booksName[i][j]}\n")

```

In seguito si procede alla lettura da file system dei risultati della funzione **wordCount()**, sfruttando il metodo **textFile()**. Tali risultati verranno poi salvati nella variabile **wordsCountAuthorsBooks**.

```

1
2 #Read saved word count RDD for every book for all authors
3 wordsCountAuthorsBooks = [[] for _ in range(n_authors)]
4
5 for i in range(n_authors):
6     print(f"Reading books for: {authors[i]}")
7     for j in range(len(booksName[i])):
8         # Getting path of books' name RDD
9         tmp = "./RDD/Books/SingleBooks/" + authors[i] + "/" +
booksName[i][j] + "/*"
10
11         print(f"Start - Reading words count of book: {
booksName[i][j]}\n")
12         result = sc.textFile(tmp).collect()
13         # Converting list of string into list of tuple
14         result = stringToTuple(result)
15         print(f"End - Reading words count of book: {booksName[
i][j]}\n")
16
17         wordsCountAuthorsBooks[i].append(result)

```

Si noti che si è usata una funzione di supporto **stringToTuple()** che converte stringhe in tuple. Tale passaggio è necessario in quanto il metodo **textFile()** legge le tuple salvate come se fossero delle stringhe.

```

1
2 #Given a list of string convert it to a list of tuple
3 def stringToTuple(wordCount):
4
5     result = []

```

```

6   for s in wordCount:
7       #Remove useless characters in the string
8       res = s.strip("(").strip(")").split(", ")
9       tmp = len(res[0])
10      #Take only the string inside the ''
11      result.append((str(res[0][1:tmp-1]), float(res[1])))
12
13  return result

```

5.5 Training

Come citato in precedenza, si è scelto di usare una lista delle parole più utilizzate come caratteristica stilistica dei vari autori. Tale lista di parole può essere scelta in diversi modi. In questo progetto si è scelto di usare 3 tipologie di lista di parole:

1. Solo parole con un significato lessicale. Si escludono quindi le **function_words** e le **punctuations_chars**,
2. Solo parole che svolgono principalmente una funzione grammaticale (**function_words**),
3. Solo punteggiature e caratteri speciali (**punctuation_chars**).

Anche per i risultati del training, si è deciso di salvare i dati su file system per lo stesso motivo spiegato nel paragrafo 5.4.

Per effettuare i vari training si è proceduto nel seguente modo:

- Viene definita la variabile **training_index** che contiene gli indici dei libri che andranno a formare il training set. Tali indici sono 7 e saranno gli stessi per tutti gli autori,
- Mediante la variabile **readFromSavedFile** viene controllato se è necessario eseguire questo blocco di codice,
- Per ogni autore viene creato un RDD vuoto (`empty_rdd`) che verrà poi riempito con gli RDD dei suoi libri presenti nella variabile **training_index**,
- Al nuovo RDD generato verranno svolte alcune trasformazioni:
 - **reduceByKey()**: si raggruppa per chiave e i relativi valori associati vengono sommati,

- **2 filter()**: la parola non deve essere presente né in **function_words** né in **punctuations_char**,
 - **sortBy()**: si esegue un ordinamento decrescente sui valori. In questo modo ci saranno come prime parole quelle più frequenti.
- Sapendo che per ciascun libro era stata aggiunta, con la funzione **word_count()**, la chiave **total_words**, tale chiave si troverà al primo posto, nel nuovo RDD, in quanto indica il numero totale delle parole presenti nell’RDD. Essa viene salvata nella rispettiva variabile e verrà poi utilizzata per la normalizzazione,
 - Tramite un **map()** si esegue la normalizzazione della frequenza delle parole,
 - Vengono poi prese e salvate su file system solamente le prime 200 parole dell’RDD.

```

1
2 #Calculating Word Count of books in training set for every
   authors
3 training_index = [0, 1, 2, 3, 4, 5, 6]
4
5 if not readFromSavedFile:
6     for a in range(n_authors):
7
8         print(f"Author: {authors[a]}")
9         empty_rdd = sc.emptyRDD()
10        #Take only the books in training_index
11        for b in training_index:
12            empty_rdd = empty_rdd.union(sc.parallelize(
wordsCountAuthorsBooks[a][b]))
13
14        #Word Count of all books in training_index of the
author
15        print("Start - Word Count of all books in
training_index")
16        # Filter to remove function words and punctuations
char
17        result = empty_rdd.reduceByKey(lambda v1, v2: v1 + v2)
\
18            .filter(lambda w: w[0] not in function_words)
\
19            .filter(lambda w: w[0] not in
punctuations_char) \
20            .sortBy(lambda w: -w[1])

```

```

21     print("End - Word Count of all books in training_index
22 ")
23     #Take total words of the training set
24     total_words = result.take(1)[0][1]
25     print(f"Total words: {total_words}")
26
27     # Normalization
28     print("Start - Normalization of word's frequency")
29     result = result.map(lambda x: (x[0], x[1]/total_words)
30 )
31     print("End - Normalization of word's frequency")
32
33     # Only save top 200 words
34     result = sc.parallelize(result.take(200))
35     print("Start - Saving RDD Training set")
36     result.saveAsTextFile(f"./RDD/Books/TrainingResult/{
authors[a]}")
    print("End - Saving RDD Training set\n")

```

Il codice appena mostrato è relativo al training della prima tipologia di lista di parole [1]. Per quanto riguarda il training delle altre 2 tipologie, è sufficiente effettuare alcune sostituzioni al codice:

- Per la seconda tipologia di lista di parole [2] bisogna sostituire i 2 **filter()** con un unico **filter()**, il quale prenderà solo le parole presenti in **function_words** oltre alla parola **total_words** che servirà per la normalizzazione.

```

1 result = empty_rdd.reduceByKey(lambda v1, v2: v1 + v2) \
2     .filter(lambda w: (w[0] in function_words) or (w[0]
3     == "total_words")) \
    .sortBy(lambda w: -w[1])

```

- Stesso discorso per la terza tipologia di lista di parole [3] dove il nuovo **filter()** prenderà solo le parole presenti in **punctuations_char** oltre alla parola **total_words**.

```

1 result = empty_rdd.reduceByKey(lambda v1, v2: v1 + v2) \
2     .filter(lambda w: (w[0] in punctuations_char) or (w
3     [0] == "total_words")) \
    .sortBy(lambda w: -w[1])

```

In seguito si procede alla lettura da file system dei risultati dei vari training i quali verranno salvati nelle seguenti variabili:

- **trainingSetWordCountAuthors** per i risultati del primo training,

- **trainingSetFunctionWordsAuthors** per i risultati del secondo training,
- **trainingSetPunctuationAuthors** per i risultati del terzo training.

```

1 trainingSetWordCountAuthors = []
2
3 #Reading RDD of authors training set result
4 for i in range(n_authors):
5     print(f"Author: {authors[i]}")
6
7     tmp = f"./RDD/Books/TrainingResult/{authors[i]}/*"
8
9     print(f"Start - Reading RDD Training set")
10    result = sc.textFile(tmp).collect()
11
12    # Converting list of string into list of tuple
13    result = stringToTuple(result)
14    print(f"End - Reading RDD Training set\n")
15
16    trainingSetWordCountAuthors.append(result)

```

Tali risultati verranno in seguito usati come indicatori per effettuare il testing e verificarne l'accuratezza.

5.6 Testing

Una volta che si sono effettuati tutti i training e ottenuto i vari indicatori, si procede al testing di tali indicatori per poter verificarne l'accuratezza.

Di seguito viene riportata la funzione che esegue il testing (essendo molto simili i codici dei testing degli indicatori ottenuti, si spiega nel dettaglio solo il testing del primo indicatore, ovvero **trainingSetWordCountAuthors**):

- Si definisce la variabile **testing.index** che contiene gli indici dei libri che andranno a formare il testing set. Tali indici sono 3 e saranno gli stessi per tutti gli autori,
- Si crea la variabile **testPredictions** in cui verranno salvati le predizioni della paternità dei libri del testing set.
- Per ogni libro di test di tutti gli autori:
 - Viene salvato, nella variabile **bookToTest**, il word count del libro da testare,
 - Si prendono le parole totali (**total_words**) di questo libro,

- Si dichiara la variabile **top100** che sarà una lista di 100 tuple (parola, frequenza) che verrà utilizzata per rappresentare il libro da testare,
- Per ogni tupla presente in **bookToTest**:
 - * Si verifica che la chiave (parola) non sia né in **function_words** né in **punctuations_char**,
 - * Viene effettuata la normalizzazione della frequenza che verrà salvata, insieme alla relativa parola, in una nuova tupla,
 - * Si aggiunge la nuova tupla alla lista **top100**,
 - * Si ripetono i passaggi finché la lista **top100** non ha 100 tuple.
- Dopodiché, a seconda del parametro **method** passato come input alla funzione, si esegue la predizione del libro. La variabile **method** indica il metodo che si è scelto di utilizzare per il testing:
 1. Conteggio delle parole in comune dei libri (**words_in_common**),
 2. Utilizzo della similarità del coseno (**cosine_similarity**),
 3. Utilizzo del coefficiente di similarità di Jaccard (**jaccard_similarity**).
- La predizione viene salvata nella variabile **testPredictions**
- Infine viene calcolata e stampata l'accuratezza del metodo scelto.

```

1 def testing(method):
2
3     # books of author to test
4     testing_index = [7, 8, 9]
5     # result of test
6     testPredictions = [[] for _ in range(n_authors)]
7
8     # give the predictions for every book in testing index of
9     # all authors
10    for a in range(n_authors):
11        for b in testing_index:
12            # word count of the book to test
13            bookToTest = wordsCountAuthorsBooks[a][b]
14            # take total words of the book
15            total_words = bookToTest[len(bookToTest)-1][1]
16            #take only top 100 words
17            top100 = []
18            count = 0
19            # Find the top100 words that are not in '
function_words' and not in 'punctuations_char' +
normalization
            for w in bookToTest:

```

```

20         if (w[0] not in function_words) and (w[0] not
in punctuations_char):
21             # Normalization
22             temp = (w[0], w[1]/total_words)
23             top100.append(temp)
24             count += 1
25             if count == 100:
26                 break
27
28         # predict the author of the book and save the
prediction using the given method
29         if method == 1:
30             testPredictions[a].append(
method1_words_in_common(top100))
31         elif method == 2:
32             testPredictions[a].append(
method2_cosine_similarity(top100, "word_count"))
33         elif method == 3:
34             testPredictions[a].append(
method3_jaccard_similarity(top100))
35
36     modelAccuracy = 0
37     # Calculate the accuracy of the model
38     for a in range(n_authors):
39         print(f"\nAuthor: {authors[a]}")
40         predictions = testPredictions[a]
41         print(predictions)
42
43         accuracy = 0
44         for p in predictions:
45             if method == 1:
46                 if a in p:
47                     accuracy += 1
48             else:
49                 if a == p:
50                     accuracy += 1
51
52         modelAccuracy += accuracy
53         accuracy = accuracy/len(predictions)*100
54         print(f"Accuracy to predict the author= {accuracy}%")
55
56     # Print the model accuracy
57     tot = sum(len(sublist) for sublist in testPredictions)
58     modelAccuracy = modelAccuracy/tot*100
59     print(f"\nModel Accuracy= {modelAccuracy}%")
60
61     return modelAccuracy/100

```

5.6.1 Metodo 1: Basato sulle parole in comune

Il primo metodo, utilizzato per il testing degli indicatori ottenuti dai training, è basato sul calcolo del numero delle parole in comune. Tale metodo è stato implementato nella funzione `method1_words_in_common()` dove si effettuano i seguenti passaggi:

- Si prende in input una lista (**top100**) che rappresenta il libro anonimo di cui si vuole predire la paternità,
- Per ogni autore **a**, si confronta il relativo indicatore che lo rappresenta (**trainingSetWordCountAuthors[a]**) con la lista **top100**, per poterne calcolare il numero di parole in comune attraverso l'uso della funzione ausiliare **count_common_words()**,
- Si salvano l'autore/gli autori che hanno il maggior numero di parole in comune,
- Si restituisce il risultato predetto.

```
1 # Predict the author of the book given in input
2 def method1_words_in_common(top100):
3     max_count = -1
4     author_index = []
5
6     #Check which authors have the greatest number of words in
7     #common compared to the book in input
8     for a in range(n_authors):
9         count = count_common_words(top100,
10         trainingSetWordCountAuthors[a])
11         if count == max_count:
12             author_index.append(a)
13
14         if count > max_count:
15             author_index = [a]
16             max_count = count
17
18     return author_index
```

La funzione ausiliare **count_common_words()**, citata sopra, esegue un semplice calcolo del numero delle parole in comune tra le 2 liste passate come parametro.

```
1 #Function to count words in common given two lists of tuple.
2 #Tuple=(int, 'word')
3 def count_common_words(list1, list2):
4     count_words = 0
```

```

4
5     list2top100 = list2[1:101]
6
7     #count words in common of the 2 lists given in input
8     for w in list1:
9         for w2 in list2top100:
10            if w[0] == w2[0]:
11                count_words += 1
12                continue
13
14     return count_words

```

Per visualizzare i risultati, che si ottengono dal testing attraverso l'utilizzo del metodo appena descritto, è sufficiente richiamare la funzione **testing()** passandogli come parametro il valore 1 o qualche variabile che abbia lo stesso valore.

```

1 method = 1
2 accuracy_wordsInCommon = testing(method)

```

5.6.2 Metodo 2: Usando la similarità del coseno

Il secondo metodo prevede l'utilizzo della similarità del coseno. E' una misura che si usa per valutare quanto due vettori siano simili fra di loro. Per calcolare la similarità del coseno fra due vettori, ad esempio A e B, si usa la seguente formula:

$$\cos(\theta) = \frac{A \cdot B}{\|A\| \|B\|} \quad (1)$$

Dove:

- $A \cdot B$ rappresenta il prodotto scalare tra i vettori A e B
- $\|A\| \|B\|$ rappresentano la lunghezza o norma dei vettori A e B rispettivamente.

Più il valore della similarità del coseno è vicino a 1, maggiore è la similarità tra i vettori.

Per l'implementazione si è utilizzata la funzione **cosine_similarity()** di Scikit-learn o abbreviato sklearn. Quest'ultima è una libreria open-source che fornisce molti strumenti e algoritmi relativi all'apprendimento automatico ampiamente utilizzata in questo settore. La funzione **cosine_similarity()** prende in input due array che hanno le seguenti caratteristiche:

- Hanno la stessa dimensione,

- Ogni i -esima posizione dei due array (a_i e b_i) fa riferimento allo stesso attributo.

In questa implementazione:

- I due array rappresentano le frequenze delle parole,
- Ogni i -esima posizione dei due array fa riferimento alla stessa parola.

Il relativo codice è stato implementato nella funzione **method2_cosine_similarity()** il quale effettua i seguenti passaggi:

- Si prende in input una lista (**list1**), che rappresenta il libro anonimo, e **feature** che specifica quale indicatore usare,
- Per ogni autore:
 - Viene creata una lista **wordsForCosineSimilarity** che dovrà contenere l'unione delle parole presenti in **list1** e **list2**,
 - In base alla **feature** data in input, viene scelto quale indicatore usare per la creazione della lista **list2**,
 - Vengono aggiunti a **wordsForCosineSimilarity** le parole di **list1** e le parole di **list2** senza, però, aggiungere eventuali parole duplicate,
 - Sfruttando **wordsForCosineSimilarity**, si creano 2 liste, **wordsValuesList1** e **wordsValuesList2**, che rappresentano rispettivamente le frequenze delle parole di **wordsForCosineSimilarity** in **list1** e in **list2**,
 - Come citato in precedenza, la funzione **cosine_similarity()** prende in input due array. Per questo motivo le due liste appena create necessitano di essere convertiti in due array: **frequency_array1** e **frequency_array2**,
 - Tali array vengono passati alla funzione **cosine_similarity()** per poterne calcolare la similarità del coseno e il risultato verrà, poi, salvato nella variabile **cosine**,
 - L'autore predetto sarà quello con il valore della similarità del coseno più alta,
- Viene restituito il risultato predetto.

```

1 # Predict the author of the book given in input
2 def method2_cosine_similarity(list1, features):
3
4     # Save only the words of list1
5     list1Words = [tup[0] for tup in list1]
6     max_cosine = -1
7     author_index = -1
8
9     #Check which authors have the greatest cosine similarity
10    for a in range(n_authors):
11        # List of words for cosine similarity (Union words of
12        # 2 lists)
13        wordsForCosineSimilarity = []
14        for w in list1Words:
15            wordsForCosineSimilarity.append(w)
16
17        # Check the feature requested
18        if features == "word_count":
19            #Take top 100 of training set wordcount
20            list2 = trainingSetWordCountAuthors[a][1:101]
21        elif features == "function-words":
22            #Take tuples of training set FunctionWords
23            list2 = trainingSetFunctionWordsAuthors[a][1:]
24        elif features == "punctuation":
25            #Take tuples of training set Punctuation
26            list2 = trainingSetPunctuationAuthors[a][1:]
27        else:
28            print(f"Feature: {features} NOT AVAILABLE")
29            list2 = []
30
31        # Complete the list of words for cosine similarity
32        list2Words = [tup[0] for tup in list2]
33        for w in list2Words:
34            if w not in wordsForCosineSimilarity:
35                wordsForCosineSimilarity.append(w)
36
37        # Take the values of the words present in list1
38        wordsValuesList1 = [0 for _ in range(len(
39        wordsForCosineSimilarity))]
40        for i, w in enumerate(wordsForCosineSimilarity):
41            if w in list1Words:
42                wordsValuesList1[i] = list1[list1Words.index(w)
43                ][1]
44
45        # Take the values of the words present in list2
46        wordsValuesList2 = [0 for _ in range(len(
47        wordsForCosineSimilarity))]
48        for i, w in enumerate(wordsForCosineSimilarity):
49            if w in list2Words:

```

```

46         wordsValuesList2[i] = list2[list2Words.index(w
    )][1]
47
48     # Create the frequency arrays
49     frequency_array1 = np.array(wordsValuesList1).reshape
    (1, -1)
50     frequency_array2 = np.array(wordsValuesList2).reshape
    (1, -1)
51     # Calculate cosine similarity
52     cosine = cosine_similarity(frequency_array1,
    frequency_array2)
53     # Update max_cosine and author_index if necessary
54     if cosine > max_cosine:
55         author_index = a
56         max_cosine = cosine
57
58     return author_index

```

Per visualizzare i risultati che si ottengono dal testing attraverso il calcolo della similarità del coseno, è sufficiente richiamare la funzione **testing()** passandogli come parametro il valore 2 o qualche variabile che abbia lo stesso valore.

```

1 method = 2
2 accuracy_cosineSimilarity = testing(method)

```

5.6.3 Metodo 3: Usando il coefficiente di similarità di Jaccard

Il terzo metodo utilizza il coefficiente di similarità di Jaccard o Jaccard similarity. Essa è una metrica che misura la similarità fra due insiemi. La formula per calcolare tale coefficiente, dati due insiemi A e B, è la seguente:

$$J(A, B) = \frac{|A \cap B|}{|A \cup B|} \quad (2)$$

Dove:

- $|A \cap B|$ è la cardinalità dell'intersezione tra gli insiemi A e B, cioè gli elementi che sono presenti in entrambi gli insiemi,
- $|A \cup B|$ è la cardinalità dell'unione tra gli insiemi A e B, ovvero il numero totale di elementi unici presenti in entrambi gli insiemi.

Questa formula restituisce un valore che è compreso fra 0 e 1, dove 0 significa che non è presente nessuna similarità tra i due insiemi, mentre 1 significa che i due insiemi sono uguali. Questa misura è adatta per insiemi non ordinati.

Tale metodo è stato implementato nella funzione `method3_jaccard_similarity()` dove vengono eseguiti i seguenti passaggi:

- Si prende in input una lista (**list1top100**) che rappresenta il libro anonimo di cui si vuole predire la paternità,
- Vengono salvate in **list1top100Words** solo le parole presenti nella lista **list1top100**,
- Per ogni autore **a**:
 - Viene salvato in **list2top100** il relativo indicatore che lo rappresenta (**trainingSetWordCountAuthors[a]**),
 - In **list2top100Words** vengono salvate solo le parole presenti nella lista **list2top100**,
 - Viene calcolato la dimensione dell'intersezione e la dimensione dell'unione fra **list1top100Words** e **list2top100Words**. I risultati vengono salvati rispettivamente nelle variabili **intersection** e **union**,
 - Si calcola il coefficiente di similarità di Jaccard attraverso il rapporto tra **intersection** e **union**,
 - L'autore predetto sarà quello con il coefficiente di similarità di Jaccard più alta,
- Si restituisce il risultato predetto.

```
1 # Predict the author of the book given in input
2 def method3_jaccard_similarity(list1top100):
3
4     # Save only the words of list1
5     list1top100Words = set([tup[0] for tup in list1top100])
6     max_jaccard = -1
7     author_index = -1
8
9     #Check which authors have the greatest Jaccard similarity
10    for a in range(n_authors):
11
12        #Take top 100 of training set wordcount
13        list2top100 = trainingSetWordCountAuthors[a][1:101]
14        list2top100Words = set([tup[0] for tup in list2top100
15                                ])
16
17        # Calculate jaccard similarity
```

```

17     intersection = len(list1top100Words.intersection(
18         list2top100Words))
19     union = len(list1top100Words.union(list2top100Words))
20     jaccard_similarity = intersection / union
21
22     # Update max_jaccard and author_index if necessary
23     if jaccard_similarity > max_jaccard:
24         author_index = a
25         max_jaccard = jaccard_similarity
26
27     return author_index

```

Per visualizzare i risultati che si ottengono dal testing attraverso il calcolo del coefficiente di similarità di Jaccard, è sufficiente richiamare la funzione **testing()** passandogli come parametro il valore 3 o qualche variabile che abbia lo stesso valore.

```

1 method = 3
2 accuracy_jaccardSimilarity = testing(method)

```

5.7 Testing degli altri 2 indicatori

Il testing degli altri 2 indicatori, ovvero **trainingSetFunctionWordsAuthors** e **trainingSetPunctuationAuthors**, è molto simile al testing spiegato sopra ma con qualche piccola differenza:

- In base all'indicatore scelto, vengono cambiati i filtri al libro da testare,
- Non vengono più prese solo le prime 100 parole,
- Viene effettuato il testing solo col secondo metodo, ovvero attraverso l'uso della similarità del coseno.

```

1 # Testing function for 'function_words' or 'punctuations_char
2 def testing2(features):
3     # books of author to test
4     testing_index = [7, 8, 9]
5     # result of test
6     testPredictions = [[] for _ in range(n_authors)]
7
8     # give the predictions for every book in testing index of
9     # all authors
10    for a in range(n_authors):
11        for b in testing_index:
12            # word count of the book to test
13            bookToTest = wordsCountAuthorsBooks[a][b]

```

```

13         # take total words of the book
14         total_words = bookToTest[len(bookToTest)-1][1]
15         # take words to test
16         test = []
17         if features == "function-words":
18             # Find the words that are in 'function_words'
19             for w in bookToTest:
20                 if w[0] in function_words:
21                     # Normalization
22                     temp = (w[0], w[1]/total_words)
23                     test.append(temp)
24         elif features == "punctuation":
25             # Find the words that are in '
punctuations_char'
26             for w in bookToTest:
27                 if w[0] in punctuations_char:
28                     # Normalization
29                     temp = (w[0], w[1]/total_words)
30                     test.append(temp)
31
32         # predict the author of the book and save the
prediction
33         testPredictions[a].append(
method2_cosine_similarity(test, features))

```

6 Approccio basato sull'utilizzo di algoritmi di Machine Learning

Un'altra soluzione per riconoscere gli autori, partendo da testi anonimi, è sicuramente mediante l'uso di algoritmi di Machine Learning. Come citato in precedenza, il machine learning è suddiviso in varie fasi:

- **Raccolta di dati:** Si è utilizzato il dataset dei libri precedentemente spiegato,
- **Selezione del modello:** Sono stati scelti SVM, Random Forest e il Classificatore Bayesiano come algoritmi di implementazione,
- **Training del modello:** Innanzitutto si è utilizzato Spark per operazioni di pulizia del testo per poi effettuare il training tramite la libreria sklearn,
- **Valutazione:** Si confrontano i risultati predetti con i risultati effettivi,
- **Implementazione:** Non si è implementato la soluzione in un'applicazione separata.

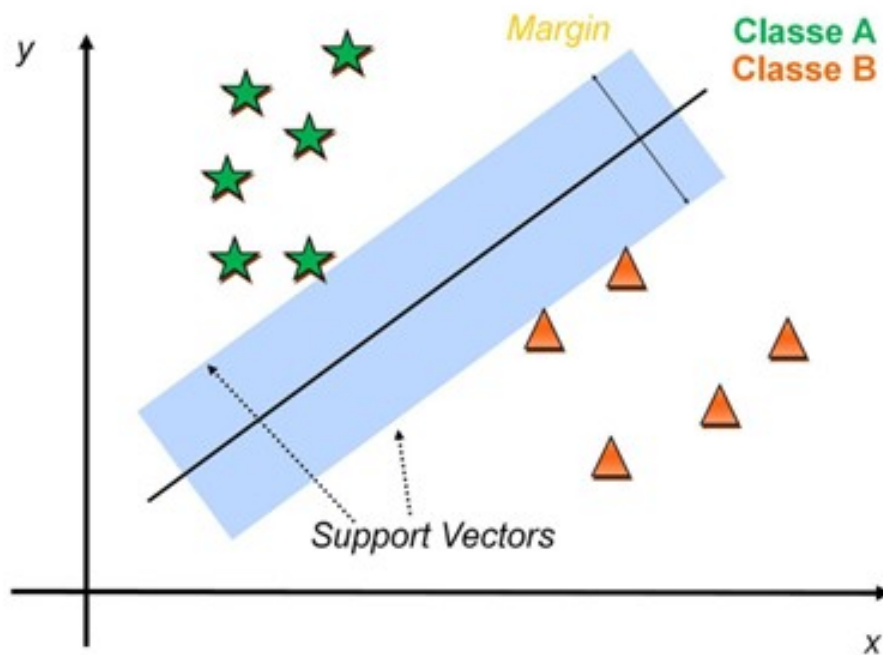
6.1 SVM

SVM o Support Vector Machine è un algoritmo di apprendimento supervisionato che si utilizza per la classificazione.

L'obiettivo di questo algoritmo è quello di trovare un iperpiano ottimale che separa al meglio le classi (nel nostro caso gli autori), ovvero deve trovare la separazione delle classi che massimizza il margine tra le classi stesse. Il margine rappresenta la distanza minima dalla retta di separazione ai punti delle classi che si ottiene mediante l'utilizzo dei vettori di supporto. Tali vettori rappresentano i valori di una classe più vicini alla retta di separazione, sostanzialmente sono i valori classificabili con maggiore difficoltà.

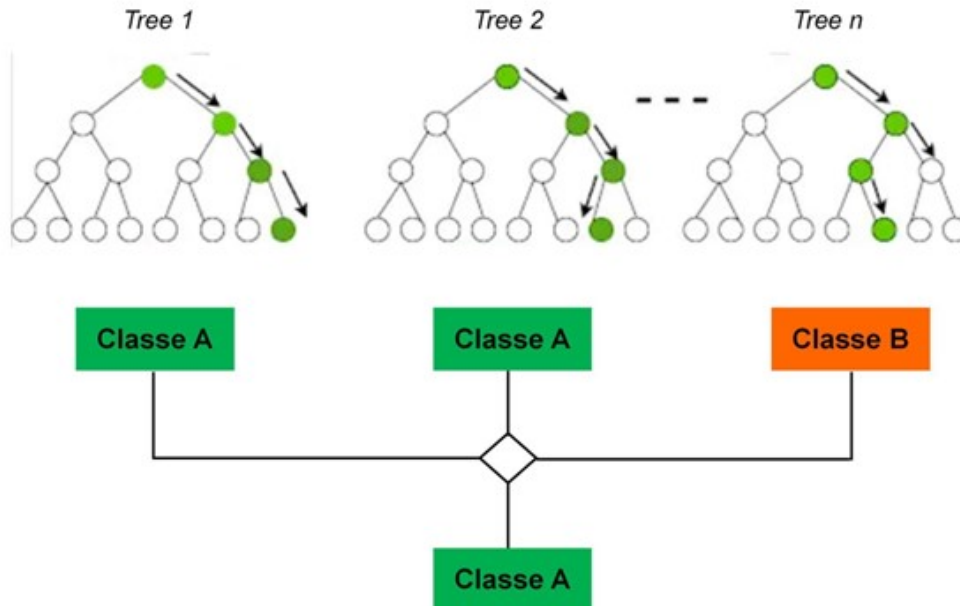
Questo aspetto differenzia la SVM da altri algoritmi di classificazione perché mentre una regressione logistica impara a classificare prendendo come riferimento gli esempi più rappresentativi di una classe, la SVM cerca gli esempi più difficili e considera soltanto quelli per fare la classificazione. Maggiore è il margine e maggiore sarà la distanza fra le classi e quindi ci sarà una migliore generalizzazione. Per il training di un modello SVM bisogna scegliere alcuni parametri come il tipo di kernel da utilizzare (lineare, polinomiale ecc...).

Questo algoritmo si rivela potente soprattutto quando si hanno dati di dimensioni moderate.



6.2 Random Forest

Random Forest è un algoritmo di apprendimento automatico usato per i problemi di classificazione e consiste nel costruire diversi alberi decisionali per poi combinarne le previsioni. Ciascun albero presente all'interno di Random Forest viene addestrato partendo da un sottoinsieme casuale dei dati presenti nel training set. Il risultato finale che viene restituito non è altro che la classe restituita dal maggior numero di alberi.



6.3 Classificatore di Bayes

Il classificatore di **Bayes** è un metodo di classificazione supervisionato che si basa appunto sul teorema di Bayes. Il teorema di Bayes calcola la probabilità di un evento A condizionata a un altro evento B

$$P(A|B) = \frac{P(B|A) \cdot P(A)}{P(B)} \quad (3)$$

E' quindi un algoritmo probabilistico. Si calcola la probabilità di ogni classe per un determinato oggetto osservando le sue caratteristiche per poi scegliere la classe con probabilità maggiore.

6.4 Training

Come accennato in precedenza, prima di eseguire il training si sono svolte alcune operazioni di pulizia del testo qui di seguito elencate:

- Viene creato una lista **authorsBooksForVectorizer** che conterrà, per ogni libro di ogni autore, il relativo testo dopo le operazioni di pulizia,

- Per ogni autore e per ogni libro:
 - Viene creato il path del libro,
 - Vengono effettuate le operazioni di pulizia tramite l'utilizzo di Spark e il risultato viene salvato nella variabile **text**. Tale processo prevede:
 - * `clearText()`: Funzione di pulizia del testo spiegata nelle sezioni precedenti,
 - * `3 map()`: Trasformano tutte le parole in minuscolo e, per ogni riga del testo, tolgono le parole presenti in **function_words** o in **punctuations_char**,
 - * `reduce()`: Vengono unite tutte le righe del testo, separate da uno spazio vuoto, in un'unica riga,
 - La variabile **text** viene, infine, aggiunta in **authorsBooksForVectorizer**

```

1 #Create authorsBooksForVectorizer array. For every author we
  store his books text as a string.
2
3 authorsBooksForVectorizer = [[] for _ in range(n_authors)]
4
5 for a in range(n_authors):
6     tmp_filename = os.path.join(my_dir, authors[a])
7     for l in range(len(booksName[a])):
8         #Create the path of the text of the book
9         input_filename = os.path.join(tmp_filename, booksName[
10            a][l])
11
12         #Use pyspark to simplify the text and reduce the rows
13         #in a single row string
14         text = clearText(input_filename) \
15             .map(lambda w: w.lower()) \
16             .map(lambda line: " ".join([word for word in line.
17                split() if word not in function_words])) \
18             .map(lambda line: " ".join([word for word in line.
19                split() if word not in punctuations_char])) \
20             .reduce(lambda x, y: x + " " + y)
21
22         #Save text in authorsBooksForVectorizer
23         authorsBooksForVectorizer[a].append(text)

```

In una seconda fase i dati vengono ulteriormente elaborati per essere usati dal modello. Infatti, **authorsBooksForVectorizer** viene prima convertito

in un numpy array e poi, attraverso l'utilizzo del metodo **flatten()**, trasformato in un array monodimensionale. Tale risultato viene infine salvato nella variabile **nested_array**. In **authors_label** vengono salvati, per ogni testo presente in **nested_array**, i relativi indici degli autori.

```
1 #Preparing data for the ML models
2
3 #We flat the variable authorsBooksForVectorizer
4 nested_array = np.array(authorsBooksForVectorizer).flatten()
5
6 #Creating author_labels
7 authors_label = []
8 for i in range(n_authors):
9     for _ in range(len(booksName[i])):
10         authors_label.append(i)
```

E' necessario, dopo questi passaggi, estrarre le feature dai testi. Si è deciso di utilizzare una classe della libreria di scikit-learn, ovvero **CountVectorizer()**, che permette di convertire testi in una rappresentazione numerica basata sulla frequenza delle parole. Il risultato di questo passaggio viene salvato nella variabile **X**. Infine, si converte **author_label** in un numpy array e il risultato verrà salvato nella variabile **y**.

```
1 #Feature extraction from the text string
2 vectorizer = CountVectorizer()
3 X = vectorizer.fit_transform(nested_array)
4
5 #Convert authors_label in np array
6 y = np.array(authors_label)
```

Ora c'è tutto il necessario per procedere con la fase di training. Tramite la funzione **train_test_split()**, si esegue la suddivisione (70-30) del dataset in train-set e test-set per ottenere le seguenti variabili:

- X_train
- X_test
- y_train
- y_test

```
1 #Split data in train and test
2 X_train, X_test, y_train, y_test = train_test_split(X, y,
3     test_size=0.3, random_state=42)
```

Il training viene eseguito dalla funzione **classifier_function()** il quale:

- In base al parametro **classifier_type** passato in input, sceglie il modello su cui eseguire il training. E' possibile scegliere di utilizzare SVC, RandomForest o Bayes e il modello scelto viene salvato nella variabile **classifier**,
- Viene, poi, eseguito il training vero e proprio attraverso il metodo **fit()** il quale necessita in input delle variabili **X_train** e **y_train** ottenuti in precedenza,
- Infine viene restituito il classificatore ottenuto.

```

1 #choose a classifier_type and train the data
2 def classifier_function(classifier_type):
3     if classifier_type == 'SVC':
4         classifier = LinearSVC()
5     if classifier_type == 'RandomForest':
6         classifier = RandomForestClassifier()
7     if classifier_type == 'Bayes':
8         classifier = MultinomialNB()
9
10    classifier.fit(X_train, y_train)
11
12    return classifier

```

Di conseguenza, in base al modello che si vuole utilizzare, questa funzione verrà richiamata in modo diverso:

```

1 #Linear SVM classifier
2 classifier_type = 'SVC'
3 classifier = classifier_function(classifier_type)
4
5
6 #Random Forest Classifier
7 classifier_type = 'RandomForest'
8 classifier = classifier_function(classifier_type)
9
10
11 #Bayes Classifier
12 classifier_type = 'Bayes'
13 classifier = classifier_function(classifier_type)

```

6.5 Testing

Il testing del classificatore ottenuto dalla funzione **classifier_function()** viene eseguito nella funzione **mltesting()** che esegue i seguenti passi:

- E' necessario il passaggio come parametri in input di **classifier**, che indica il classificatore da testare, e **classifier_type**, che indica il modello scelto per il classificatore,
- Vengono salvati, nella variabile **y_pred**, i risultati predetti dal classificatore sui dati di test (**X_test**),
- Viene calcolata l'accuratezza tramite la funzione **accuracy_score()**. A questa funzione, infatti, vengono passati sia gli autori predetti (**y_pred**) sia quelli reali (**y_test**),
- Si ottengono informazioni aggiuntive, oltre all'accuracy, tramite la funzione **classification_report()**,
- Viene creata e stampata il grafico della matrice di confusione,
- Alla fine viene ritornata l'accuratezza del classificatore passato in input.

```

1 #test the classifier
2 def mltesting(classifier, classifier_type):
3     y_pred = classifier.predict(X_test)
4     accuracy = accuracy_score(y_test, y_pred)
5     report = classification_report(y_test, y_pred)
6
7     print(f"Accuracy score of {classifier_type} : {accuracy}")
8     print(f"Classification Report: \n{report}")
9
10    # Calculate the confusion matrix
11    conf_matrix = confusion_matrix(y_test, y_pred)
12
13    # Create a heatmap of the confusion matrix
14    plt.figure(figsize=(8, 6))
15    sns.heatmap(conf_matrix, annot=True, fmt="d", cmap="Blues"
16    ,
17                xticklabels=label_mapping.keys(), yticklabels=
18                label_mapping.keys())
19    plt.xlabel("Valori Predetti")
20    plt.ylabel("Valori Veri")
21    plt.title("Matrice di Confusione")
22    plt.show()
23
24    return accuracy

```

Per effettuare il testing e calcolare l'accuratezza dei diversi modelli bisogna, quindi, richiamare la funzione appena descritta:

```

1 accuracy = mltesting(classifier, classifier_type)

```

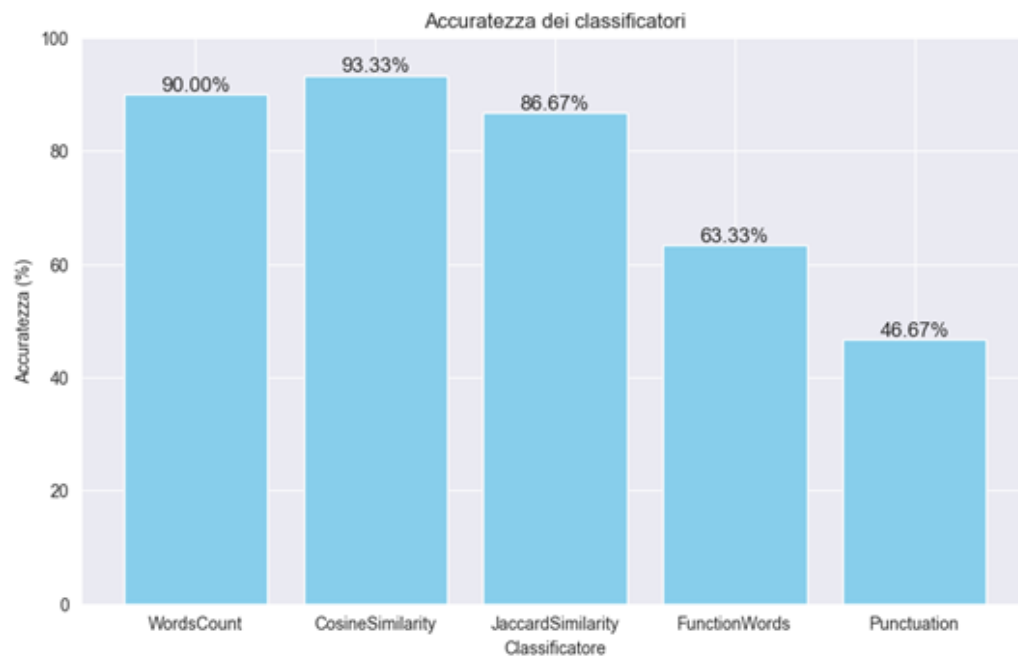
7 Risultati

I risultati ottenuti possono essere visualizzati, singolarmente, tramite le varie funzioni di test spiegate in precedenza in quanto, in esse, sono comprese azioni di calcolo e stampa dell'accuratezza del classificatore scelto.

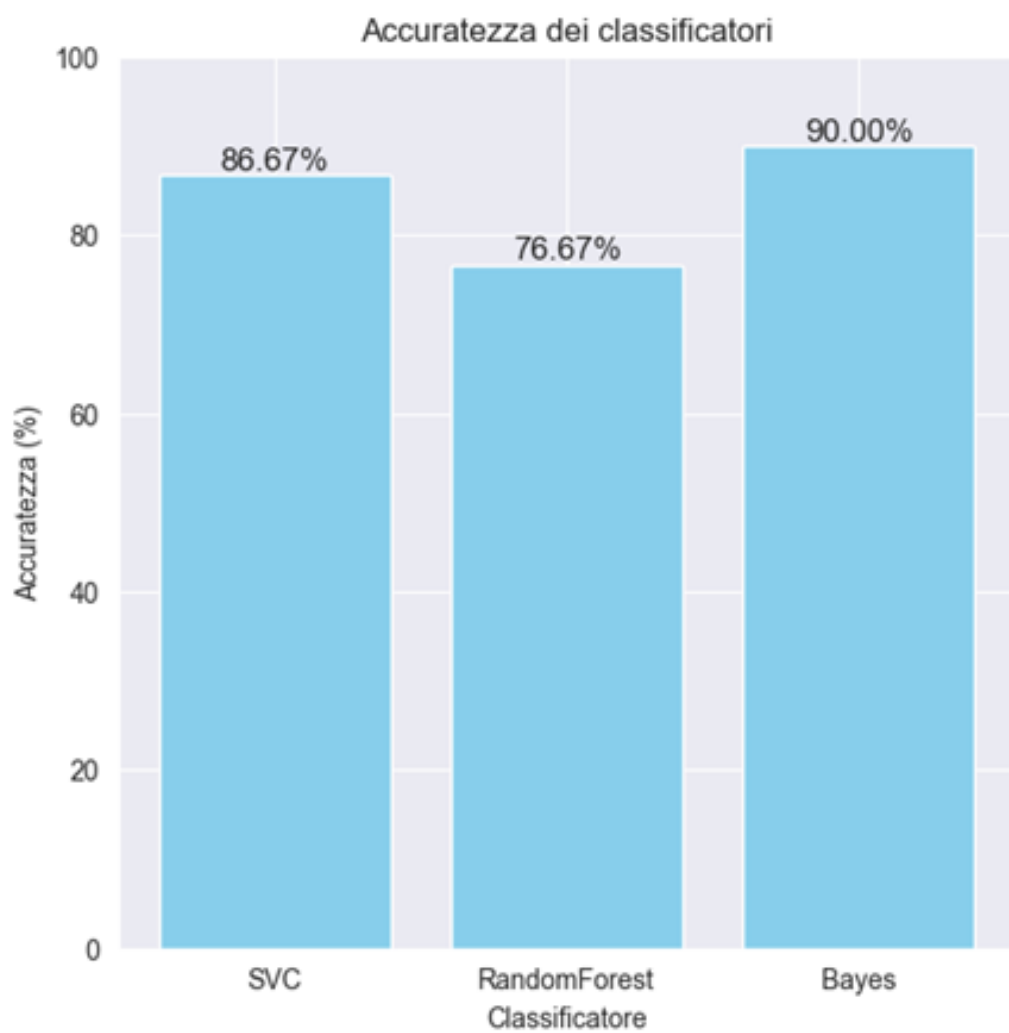
Per poter effettuare un controllo dell'accuratezza dei vari classificatori in maniera più semplice, si è utilizzata la libreria matplotlib che permette di stampare dei grafici in modo tale da facilitare la visualizzazione dei diversi risultati.

Di seguito sono riportate le performance dei vari classificatori:

- Approcci basati sulle parole più frequenti usate dall'autore



- Approcci basati sull'utilizzo di algoritmi di Machine Learning



Si può notare la presenza di classificatori più accurati rispetto ad altri. I due approcci hanno ottenuto risultati abbastanza simili fra loro, anche se sia il migliore che il peggior classificatore si trovano nel gruppo del primo approccio.

8 Conclusioni

Come già detto in precedenza i classificatori implementati funzionano mediamente abbastanza bene, dal migliore al peggiore troviamo:

Classificatore	Accuratezza
Cosine Similarity	93,33%
Bayes	90%
WordsCount	90%
SVC	86,67%
Jaccard Similarity	86,67%
Random Forest	76,67%
Function Words	63,33%
Punctuation	46,67%


I 2 modelli peggiori (con accuratezza 63.33% e 46.67%) fanno parte del primo approccio e sono quelli che prendono in considerazione o solo le **function_words** o solo i **punctuation_char**. Tale bassa accuratezza potrebbe essere dovuta al fatto che, visto che si riferiscono a parole/punteggiature che forniscono poco significato lessicale e che sono largamente utilizzati in qualunque testo scritto, essi non riescano a fornire abbastanza informazioni da poter caratterizzare un autore in maniera univoca.

Per il primo approccio si può, anche, notare che il metodo che usa la similarità del coseno performa leggermente meglio degli altri. Il motivo dietro ciò potrebbe essere che la similarità del coseno non prende in considerazione soltanto gli insiemi delle parole più frequenti (come invece accade con la similarità di Jaccard e al word count), ma viene data molta rilevanza anche alla frequenza di quanto queste parole appaiono nei vari testi oltre al valutarne la direzione e la distanza dei vettori.

I classificatori dell'approccio che usano algoritmi di Machine Learning, invece, offrono delle accuratezze abbastanza buone e molto simili tra loro con differenze dettate, principalmente, dalla tipologia del modello adottato. In particolare la meno performante risulta essere il classificatore che usa random forest (76,66%), il quale algoritmo potrebbe essere il meno adatto, tra quelli da noi usati, per risolvere il problema affrontato in questo progetto.

Note aggiuntive:

- E' importante sottolineare che i risultati riportati potrebbero subire delle variazioni a seconda del dataset utilizzato.

- Si noti che, nel caso si utilizzino dataset diversi da quello utilizzato in questo progetto, è necessario il ricalcolo degli RDD. Tale operazione potrebbe essere abbastanza onerosa in termini di tempo.
- L'implementazione del progetto può essere visionato nella seguente repository di GitHub:  [Link](#)
- Per l'esecuzione del codice è necessario l'installazione di python 3, pySpark oltre alle librerie importate nel progetto.