ALMA MATER STUDIORUM – UNIVERSITÀ DI BOLOGNA
CAMPUS DI BOLOGNA

Scuola di Scienze
Corso di Laurea in Informatica per il Management

TOPIC ANALYSIS DELLA LETTERATURA SCIENTIFICA SUL TEMA COMPUTER CHESS CON METODI DI TEXT MINING NON SUPERVISIONATI

Relatore
Prof. Angelo Di Iorio

Co-relatori
Prof. Gianluca Moro
Prof. Paolo Ciancarini

Presentata da
Andrea Borghesi

Seconda Sessione di Laurea
Anno Accademico 2020 – 2021

PAROLE CHIAVE

Natural Language Processing

Machine Learning

Unsupervised Classification

Semantic Similarity Search

Python

Ai miei cari nonni.

Introduzione

Nel mondo moderno in cui viviamo, la mole di dati disponibile rappresenta una fonte preziosa di informazioni, che può essere impiegata per eseguire varie task di **machine learning**. Molti di questi dati sono in formato testuale e vengono denominati **dati non strutturati**: tramite tecniche di **Natural Language Processing** (**NLP**) è possibile processarli e convertirli in una rappresentazione numerica, trasformandoli nella stessa forma dei *dati strutturati*, ovvero tutti quei dati appartenenti a un dominio ben definito e utilizzabili per addestrare i vari modelli di machine learning.
Lo studio del testo a livello lessicale e di *sentiment analysis* è fondamentale per comprendere l'evoluzione degli argomenti, delle tecniche e delle soluzioni proposte negli anni e presenti in una vasta quantità di documenti. Basta pensare a tutte le informazioni in formato testuale ricavabili dal web, dai documenti digitali e da quelli fisici (i quali possono essere scannerizzati e memorizzati su un dispositivo digitale).

Il caso di studio di questa tesi si basa su un dataset di articoli raccolti dagli anni 50 a oggi, che trattano l'evoluzione del rapporto tra il mondo degli scacchi e quello dell'intelligenza artificiale. Tali documenti sono stati suddivisi in quattro ere storiche, individuate a seguito di uno studio svolto da un esperto. Con la nostra analisi si è cercato di osservare se questa suddivisione fosse ricavabile anche dai modelli di machine learning, basandosi solamente sul contenuto degli articoli.

Un problema che è probabile si verifichi in situazioni reali e, soprattutto, nel possedere dataset di grandi dimensioni con istanze in formato testuale, rappresenta la difficoltà nell'etichettare questi dati: i modelli di machine learning supervisionati per funzionare hanno bisogno di *istanze etichettate*, ovvero istanze a cui è stato assegnato a priori un label, una classe o una categoria di appartenenza. Questa associazione può non essere disponibile a priori e può essere molto complicato e *time consuming* effettuare un'etichettatura se si posseggono milioni di dati (non è un procedimento scalabile).
I dati senza label possono essere processati da **modelli non supervisionati**: un esempio sono i modelli di **clustering** (o di **classificazione non super-**

visionata), i quali cercano delle correlazioni basandosi solamente sui dati a disposizione.

Il problema che si è voluto affrontare in questa sede riguardava le due problematiche sopra citate: una *classificazione non supervisionata* di *dati non strutturati*. Le tecniche studiate e analizzate riguardano quindi il mondo del NLP, gli algoritmi di clustering e i modelli per individuare la presenza di topic all'interno della collezione di documenti.

Inoltre, un altro grande problema, che è molto plausibile incontrare in situazioni reali, è dato dalla distribuzione delle istanze nel dataset: si parla di **dataset non bilanciato** quando il numero di istanze appartenenti a ogni classe differisce significativamente, ovvero quando la quantità di istanze di una classe è in rapporto 4 o più volte maggiore rispetto a quella di un'altra classe. Questo fattore causa diversi problemi ai modelli di machine learning, i quali potrebbero produrre dei risultati inattesi o inaffidabili, ovvero determinati dallo sbilanciamento del dataset e non da correlazioni valide e giustificabili tra le istanze.

In particolare, la nostra distribuzione dei documenti nelle ere storiche era molto sbilanciata: il rapporto tra il numero di documenti nella prima era storica (1950-1977) e nella terza (1997-2017) era superiore addirittura di un ordine di grandezza. Per questo motivo sono state affrontate, studiate e applicate anche delle **tecniche di bilanciamento per istanze di formato testuale**.

Riassumendo, lo studio svolto in questa tesi è focalizzato sulla manipolazione e sul processamento di dati testuali, ottenuti da articoli aventi come argomento il parallelismo tra computer chess e AI e suddivisi in 4 classi con un rapporto fortemente sbilanciato, per addestrare dei modelli di classificazione non supervisionata al fine di verificare la suddetta suddivisione.

I risultati ottenuti mostreranno le difficoltà nel lavorare con un dataset così poco bilanciato e come il lavoro svolto sul preprocessing dei dati e sulla progettazione di diversi modelli legati al mondo del NLP (tf-idf, Word Embeddings, LDA) possa portare all'individuazione di correlazioni presenti all'interno dei testi di articoli diversi.

La tesi è suddivisa nei seguenti capitoli, che descrivono i passi dalla lettura dei file alla progettazione dei modelli di classificazione.

- **Capitolo 1 - Tecniche di ML e Deep Learning per analisi di testi.** Spiegazione e descrizione ad alto livello dei metodi necessari per svolgere

un valido preprocessing dei dati testuali e analisi delle contromisure da adottare con un dataset non bilanciato. Allo stesso modo sono state introdotte le teorie sui modelli LDA, sui transformers utilizzati e sui metodi di classificazione non supervisionata.

- **Capitolo 2 - Analisi del caso di studio.**
  Descrizione approfondita del dataset del caso di studio sugli articoli sugli scacchi e analisi della suddivisione in ere, anche tramite l'impiego di test statistici come il test Chiquadro.

- **Capitolo 3 - Applicazione della teoria con Python.**
  Elenco dei test effettuati con il rispettivo codice utilizzato e analisi dei risultati ottenuti.

- **Capitolo 4 - Conclusioni e sviluppi futuri.**
  Considerazioni ricavate a seguito degli esperimenti statistici e di machine learning e conclusioni sul problema relativo alla classificazione non supervisionata di dati non strutturati presi da un dataset non bilanciato.

Indice

Elenco delle figure

Capitolo 1

Tecniche di ML e Deep Learning per analisi di testi

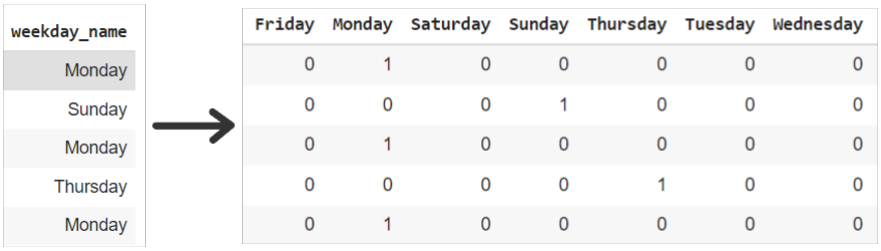
In questo capitolo verranno presentati tutti i modelli e le tecniche di machine learning e NLP (*Natural Language Processing*) applicati per questo caso di studio. Verrà presentato lo stato dell'arte di ogni argomento trattato e il motivo che ha portato alla selezione degli stessi. Nel Capitolo 3 verrà applicata questa teoria attraverso il linguaggio Python e le apposite librerie.

1.1 Lavorare con dati destrutturati

Nell'era dei Big Data, le informazioni che possiamo ricavare dai dati possono avere applicazione in numerosi tasks. Molti dati possono essere processati subito dopo essere stati raccolti: ovvero tutti quelli che sono registrati come numeri e hanno un dominio ben chiaro come quello dei numeri reali. Pensiamo a tutti quei valori a cui associamo un'unità di misura, come i kg, i metri, i gradi Celsius, i kWh per misurare il consumo energetico... tutte informazioni che i modelli di intelligenza artificiale possono comprendere e sfruttare.
Un'altra tipologia è quella delle variabili categoriche: ovvero quelle il cui valore è compreso in un dominio specifico, ma non necessariamente numerico. Un esempio sono i giorni della settimana, i mesi in un anno, un elenco di nomi di nazioni, una preferenza rappresentata da un numero intero compreso tra 1 e 5 e così via. Questo tipo di informazioni possono essere utilizzate facilemente dopo avere effettuato un preprocessing basilare come il *One-Hot Encoding* [1] : un procedimento dove le variabili categoriche sono convertite in una forma che può essere compresa dagli algoritmi di machine learning. Per esempio, i giorni

[1]Per saperne di più sulla codifica One-Hot:
https://www.kaggle.com/dansbecker/using-categorical-data-with-one-hot-encoding.

della settimana possono essere rappresentati da un vettore di 7 dimensioni dove si ha 1 nell'indice del numero del giorno specifico e 0 in tutti gli altri slot.



weekday_name
Monday
Sunday
Monday
Thursday
Monday

Friday	Monday	Saturday	Sunday	Thursday	Tuesday	Wednesday
0	1	0	0	0	0	0
0	0	0	1	0	0	0
0	1	0	0	0	0	0
0	0	0	0	1	0	0
0	1	0	0	0	0	0

Figura 1.1: Esempio di One-Hot Encoding per i giorni della settimana.
Fonte: https://towardsdatascience.com/stop-one-hot-encoding-your-time-based-features-24c699face2f

Tutti i dati descritti fino ad ora vengono denominati **dati strutturati**. Tuttavia, non possono essere ignorate tutte le informazioni che possiamo ottenere da dati di formato testuale, denominati **dati non strutturati**.
Quest'ultimi sono molto più facili da raccogliere, basti pensare a tutti i testi che possiamo ottenere dal web: blog, recensioni su siti vari, post su Facebook, tweet su Tweeter... tutte informazioni che possono risultare fondamentali nel raggiungimento di risultati ottimali per molte task di machine learning.
Non bisogna ignorare anche tutti i testi ricavabili da documenti fisici, che è proprio il caso discusso da questa tesi. I documenti, soprattutto quelli più datati, possono essere scannerizzati e letti da altri modelli di intelligenza artificiale, come quelli OCR (*Optical Character Recognition*).
Riassumendo, i modi per raccogliere dati testuali sono molteplici, ma non sono facilmente utilizzabili come quelli strutturati. Infatti, risulta fondamentale un accurato lavoro di preprocessing per convertirli in un formato (sempre numerico) sfruttabile dai modelli di AI.

1.1.1 Preprocessing dei dati

La fase di preprocessing dei dati può influenzare profondamente l'esito e l'accuratezza dei risultati di un modello di machine learning. Il testo è caratterizzato da delle regole linguistiche specifiche della lingua di appartenenza. A volte alcune frasi possono risultare ambigue e possono essere difficili da interpretare. Per esempio, nel caso in cui una parola abbia un significato diverso in base a due contesti differenti (si parla di polisemia). Un altro problema è quello di riconoscere parole con lo stesso significato, ovvero i sinonimi, che essendo lessicalmente diversi, se non vengono trattati intelligentemente possono

assumere un significato diverso per un modello di NLP. Altre volte invece, il testo può presentare degli errori: come frasi scritte male con errori verbali o di spelling. Oppure, come nel nostro caso, alcuni file mal scannerizzati, o comunque in bassa definizione, possono produrre degli errori nella lettura automatica da parte dei modelli di AI con conseguenti imprecisazioni nelle analisi successive.
Quello che si può fare è lavorare per migliorare la qualità del testo da processare impiegando tecniche come le seguenti:

- **Casefolding**. Il casefolding consiste nel convertire tutte le parole in minuscolo (o in maiuscolo), in modo che non venga fatta differenza, per esempio, tra una parola all'inizio di una frase con la prima lettera maiuscola e la stessa parola all'interno di un'altra frase.

- **Lemmatizzazione**. Le parole possono avere coniugazioni diverse: i sostantivi possono differire sintatticamente dal singolare al plurale e i verbi possono avere forme diverse in base al tempo verbale. Normalmente si lavora quindi con termini in forma flessa e può essere utile uniformarli ignorando la coniugazione. La lemmatizzazione consiste nel sostituire ogni parola con il suo **lemma**: la sua forma base presente nel dizionario. I nomi verranno quindi trasformati tutti al singolare, i verbi all'infinito... Per applicare questa tecnica è necessario avere una conoscenza approfondita della lingua di riferimento e devono essere impiegati algoritmi complessi.

- **Stemming**. Lo stemming consiste nel sostituire ogni parola con la sua **radice morfologica**: per esempio rimuovendo o sostituendo i suffissi. Questa tecnica è un'alternativa più semplice e efficiente della lemmatizzazione e unifica i termini simili potenzialemente correlati (ovvero parole con la stessa radice). Dall'altro lato però, può comportare una perdita di precisione, l'utilizzo delle radice rende in alcuni casi complicato giustificare e interpretare i risultati ottenuti e, infine, la sua buona efficacia può essere influenzata dalla lingua a cui viene applicata.

- **POS Tagging**. Le *part of speech* (POS) sono le classi grammaticali come nomi, aggettivi, verbi... Il POS Tagging altro non è che l'analisi grammaticale del testo: come la distinzione dei sostantivi in base al singolare o plurale o dei verbi in base al tempo. Questo permette di filtrare i termini di un testo in base al loro utilizzo nelle frasi.[2]

[2]La lista delle classi grammaticali riconosciute dagli algoritmi automatici come quelli forniti dalla libreria NLTK in Python si possono trovare al seguente link: https://www.cis.uni-muenchen.de/ schmid/tools/TreeTagger/data/Penn-Treebank-Tagset.pdf

- **Stopwords**. Alcuni termini non portano alcun beneficio ai modelli di machine learning: se una parola è molto frequente indipendentemente dall'argomento trattato, allora sarà di poco valore e potrà essere scartata. Un esempio sono gli articoli e le preposizioni. L'insieme di queste parole sono riconosciute come stopword e sono spesso rimosse a priori dal testo. Per compiere ciò, vengono utilizzate delle liste di parole predefinite specifiche del linguaggio impiegato.

- **Regex**. Una ***Reg****ural* ***Ex****pression* è una sequenza di caratteri che specifica un pattern di ricerca. Si tratta di uno strumento molto potente per individuare delle parti di testo specifiche, con lo scopo di poterle modificare o sostituire. Può essere impiegata per rimuovere i numeri e la punteggiatura da un documento.

- **WordNet**. WordNet è uno dei più famosi database lessicali della lingua inglese, con più di 150mila termini. Offre quindi anche un dizionario contenente tutte le parole esistenti e che può essere sfruttato per filtrare gli errori di spelling e le letture imprecise che causano parole inesistenti. Inoltre, raccoglie anche le principali relazioni semantiche tra le parole come l'iponimia, la meronimia, l'antonimia...

Per applicare le tecniche sopra riportate bisogna effettuare un'operazione chiamata **segmentazione**: consiste nello scomporre il testo in più parti. La *word tokenization* è una delle più comuni e trasforma un testo in una lista delle parole che lo compongono.
Una funzione che applica questo procedimento viene comunemente chiamata *tokenizer* e viene utilizzata da altre tecniche come la *tf-idf*.
Infine, una **Bag of Words** (**BoW**) è un elenco delle parole distinte contenute in un documento e il relativo numero di occorrenze di ognuna di esse al suo interno. Le tecniche di lemmatizzazione e di stemming permettono di ridurre il numero di questi termini.

1.1.2 TF-IDF

Un insieme di documenti può essere rappresentato in uno spazio vettoriale chiamato **Vector Space Model**: si tratta, tipicamente, di una matrice che ha in ogni riga un documento, in ogni colonna un termine e in ogni cella l'occorrenza di quel termine in quello specifico documento.
Quello appena descritto è un procedimento molto semplice, ma che da poche indicazioni sull'importanza delle parole in un documento: non distingue in maniera efficace le parole che lo rappresentano meglio dalle altre. Esistono a tale scopo degli schemi di **term weighting**, ossia tecniche che permettono di

pesare un termine all'interno di un documento in base alla sua importanza. Una di queste tecniche è la TF-IDF dove:

- **tf** sta per *term frequency* e indica l'importanza di un termine all'interno del documento che lo contiene basandosi sul numero di occorrenze in esso;
- **idf** sta per *inverse document frequency* e indica l'importanza del termine all'interno dell'intera collezione di documenti. Questo valore risulta alto per i termini che compaiono in un numero esiguo di documenti, mentre risulta basso per le parole contenute in un grande numero di documenti (come per le stopword).

Il tf-idf è quindi composto rispettivamente da un *fattore locale* e da un *fattore globale*. In termini matematici, le formule utilizzate sono le seguenti:

$$TF(d,t) = \frac{N_{t,d}}{N_d} \qquad IDF(t) = \log\left(\frac{D}{D_t}\right)$$

Dove:

- $N_{t,d}$ è il numero di occorrenze del termine t nel documento d
- N_d è il numero di termini nel documento d
- D è il numero totale di documenti
- D_t è il numero di documenti che contengono t

La formula complessiva è la seguente:

$$tf.idf(t,d) = TF \cdot IDF$$

Empiricamente si è visto che la crescita logaritmica dell'importanza dei termini rispetto al numero di frequenza funziona meglio.
Queste formule sono le più utilizzate nella letteratura, tuttavia sono state proposte numerose variazioni. Alcune di queste vanno a variare le formule dei due fattori, mentre altre vanno a modificare solo la formula finale. Inoltre, il metodo tf.idf è un metodo non supervisionato e anche in questa circostanza sono stati proposti altri metodi supervisionati [1] che sfruttano anche la classe o la categoria di appartenenza del documento utilizzato nelle formule.

Riassumendo, applicando il tf.idf a un corpo di documenti otterremo un vector space model che i modelli di machine learning riusciranno a interpretare.

1.1.3 LSA

La matrice prodotta tramite le tecniche come il tf.idf ha un numero di righe pari al numero di documenti e un numero di colonne pari al numero di termini distinti. Nella maggior parte dei casi reali il numero di parole di un intero elenco di documenti può essere molto elevato. Applicare del preprocessing, come la lemmatizzazione o la rimozione delle stopword, può diminuire il numero di termini unici complessivi. Tuttavia, ciò potrebbe non essere sufficiente e i termini potrebbero essere in numero ancora troppo elevato: questo potrebbe aumentare notevolmente i tempi di addestramento e di validazione dei modelli di machine learning.

La *Latent Semantic Analysis* (**LSA**) [2] è una tecnica che trasforma la matrice termini-documenti facendo emergere delle associazioni semantiche latenti: ovvero correlazioni rilevanti come tra due sinonimi. Viene effettuato il mapping della matrice in uno spazio vettoriale di dimensioni inferiori, che approssima quello originale ignorando alcune informazioni poco rilevanti (come l'eventuale rumore). Nello spazio risultante, i termini semanticamente simili occuperanno posizioni limitrofe.
Per effettuare questa operazione, viene effettuata la **fattorizzazione SVD**: una tecnica molto utilizzata per ridurre il rango di una matrice. Indichiamo la matrice documenti-termini con C, la fattorizzazione SVD effettua una scomposizione in tre diverse matrici:

$$\underset{MxN}{C} = \underset{MxM}{U} \cdot \underset{MxN}{\Sigma} \cdot \underset{NxN}{V^T}$$

dove U è la matrice documenti-documenti in cui ogni cella rappresenta la similarità tra due documenti, mentre V è la matrice termini-termini dove ogni cella contiene la similarità tra due termini. Con Σ viene indicata la matrice diagonale degli autovalori: essi vengono chiamati anche **valori singolari** e sono posizionati in ordine decrescente. La fattorizzazione della matrice dipende dal numero k di valori singolari che si intende selezionare: con un k troppo piccolo si rischia di perdere informazioni importanti, mentre con un k troppo grande aumenta il rischio di incorporare informazioni fuorvianti, ovvero del rumore.
In questo modo possiamo ridurre il rango della matrice iniziale a un valore k da noi prescelto. Le k nuove dimensioni ottenute avranno un'importanza uguale al corrispettivo valore singolare. Ogni documento sarà rappresentato da un vettore di k dimensioni che riassumono il valore dei pesi dei termini che lo compongono.
La LSA permette di individuare delle associazioni come la sinonimia tra le parole: questo può portare all'individuazione più accurata degli argomenti trattati nei testi e nella classificazione dei documenti.

Uno dei limiti maggiori del LSA è il non riconoscimento della polisemia: si verifica quando una parola assume significati diversi in base al contesto.

1.1.4 Word Embeddings: Word2Vec

Riassumendo i paragrafi precedenti, il testo è una forma di dato non strutturato che per essere processato deve essere convertito in dati strutturati al costo di una probabile perdita di informazioni.
La teoria legata al *Word Embeddings* cerca di proporre una soluzione alternativa: si considerano due parole come semanticamente correlate se il loro contesto è simile. In altri termini, due parole che hanno un contesto simile dovrebbero essere rappresentate da due vettori simili: questi vengono chiamati word embeddings (WE) e vengono calcolati con un algoritmo non supervisionato.
Tutte le parole potranno essere rappresentate in una spazio vettoriale di n dimensioni. Anche in questo caso, come con LSA, le dimensioni ottenute sono sensibilmente ridotte rispetto alla rappresentazione dei documenti con tf-idf.

L'algoritmo di apprendimento utilizza delle tecniche ispirate alle reti neurali e ai processi non supervisionati: si tratta di una funzione che effettua il mapping di una parola nello spazio vettoriale specificato.
Esistono quindi diverse tecniche per ottenere delle word embedding: attraverso le **reti neurali** addestrate su task di NLP oppure con metodi statistici come **Word2Vec** e **Gloves** [3]. In quest'analisi verrà messo a focus il funzionamento e l'impiego di Word2Vec.
Un modello di WE addestrato ci indicherà per ogni parola del nostro dominio le parole che, con maggior probabilità, si troveranno in sua prossimità all'interno di una frase o di un testo (dipende dai parametri passati al modello).

Il modello **Word2Vec** [3] (W2V) [4] è basato sui modelli di reti neurali che convertono un elenco di documenti non etichettati in un insieme di vettori rappresentativi delle parole e delle loro associazioni semantiche. Questo permette di eseguire delle operazioni vettoriali tra i termini. Per esempio, immaginando ogni parola come un vettore delle medesime dimensioni si ha:

$$King - Man + Woman = Queen$$

La somma dei vettori della parte sinistra dell'equazione produrranno un vettore molto simile a quello associato alla parola "Queen".
Possiamo inoltre misurare la similarità semantica tra due parole con la formula della *similarità coseno*: misura il coseno dell'angolo tra due vettori e restituisce

[3] Introdotto da Tomas Mikolov nel 2013.

un valore tra -1 e 1, dove 1 è il valore di massima similarità. Questo metodo risulta più efficace rispetto alla distanza euclidea che non tiene conto della scala dei valori delle dimensioni dei vettori. Nella similarità coseno conta solo l'angolo formato tra essi.

Esistono due diverse tecniche di apprendimento per i modelli W2V:

- **CBOW model**. Il *Continuos Bag-of-Words model* predice una parola in base al contesto: vengono combinati i valori delle parole circostanti per determinare la parola "nel mezzo". Uno degli iperparametri di questi modelli è proprio la finestra (*window*): ovvero il numero di termini posizionati prima e dopo la parola da definire. Una finestra di 10 termini prenderà in considerazione le 5 parole precedenti e le 5 successive.
- **Skip-Gram model**. Questo modello effettua il procedimento inverso rispetto a quello precedente: predice le parole circostanti a uno specifico termine.

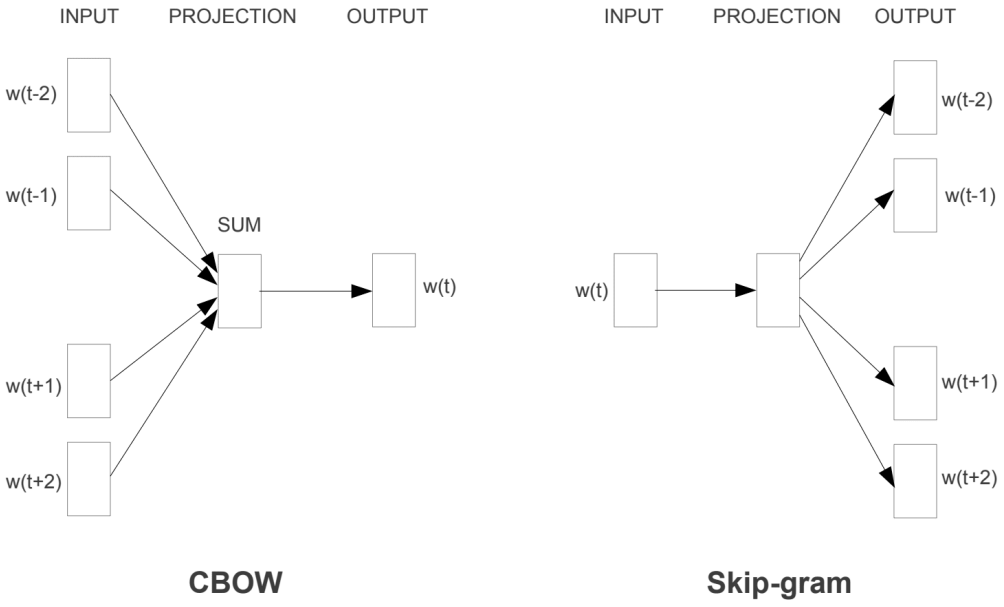


Figura 1.2: Word2Vec models.
Fonte: T. Mikolov, 2013, "Efficient Estimation of Word Representations in Vector Space"

Nella pratica, questi modelli vengono utilizzati per ottenere le parole più simili ai termini chiave (*keyword*)[4] o per calcolare la rappresentazione di un

[4]Vedi paragrafo 3.3.

intero documento effettuando il *pooling* delle parole che lo compongono. Per esempio, si parte da una matrice che ha nelle righe i termini di un documento e possiede tante colonne quante sono le dimensioni dei vettori ottenuti addestrando un modello W2V. Una semplice possibilità è quella di effettuare la media di ogni colonna, in modo da ottenere un unico vettore, delle stesse dimensioni di ogni termine, rappresentativo del documento. Questa tecnica e altre più complesse vengono utilizzate con lo stesso scopo nei transformers. Ripetendo il procedimento per ogni documento, si otterrà un vettore per ogni documento, rendendoli confrontabili tra loro con metodi come la similarità coseno.

1.1.5 Tecniche di bilanciamento di dataset testuali

I modelli di classificazione di machine learning vengono addestrati per predire la classe, categoria o *label* di appartenenza di un'istanza di dati. Se le classi sono più di due, allora si parla di *classificazione multi-label*.
È molto comune trovare un numero di istanze significativamente maggiore per una classe rispetto alle altre. Un esempio può essere un modello di classificazione di aerei in ritardo: i dati su aerei in orario saranno in rapporto molto superiori rispetto ai dati degli aerei che sono atterrati con un ritardo. Quando si verificano queste condizioni si parla di **dataset sbilanciato**. La soluzione più ovvia è quella di aumentare il numero di istanze della classe meno rappresentata, tuttavia questo è spesso complicato e potrebbe richiedere un consumo notevole di tempo.
Esistono due approcci classici applicabili in queste situazioni:

- *oversampling*: produrre delle istanze fittizie della classe meno rappresentata basandosi su quelle reali, creandone quindi delle nuove che siano "verosimili";

- *undersampling*: diminuire il numero di istanze della classe più rappresentata con il rischio di perdere informazioni importanti.

Applicare queste tecniche per dataset di dati strutturati risulta particolarmente semplice. Tuttavia, per dati non strutturati, come quelli testuali, bisogna compiere qualche passo in più. Infatti, applicando questi procedimenti sul Vector Space Model non si ottengono buoni risultati: verrebbero generate nuove istanze basandosi sui valori delle dimensioni, mentre sarebbe più efficace generare direttamente nuove istanze testuali.
In questo caso, si parla di **data augmentation** e si intende la creazione di nuove istanze testuali basandosi sui testi reali. Esistono diverse tecniche applicabili per ottenere questo risultato: una delle più semplici, applicata in

questa tesi, consiste nel dividere un documento in frasi o paragrafi, e mischiarli in maniera randomica. In questo modo si ottengono nuovi testi che mantengono il contesto in cui le parole vengono utilizzate.
Un'altra possibilità più complessa consiste nel sostituire i termini di un testo con dei sinonimi, sfruttando il vocabolario di WordNet o la lista delle parole più simili ricavibili con i modelli di word embeddings.

Figura 1.3: WordNet example.

Figura 1.4: Word2Vec example.

Fonte: https://amitness.com/2020/05/data-augmentation-for-nlp/

Con questi metodi, possiamo ottenere un dataset più bilanciato e i modelli, nella maggior parte dei casi, produrranno dei risultati più accurati e più affidabili.

1.2 Individuare i topics con LDA

La **latent Dirichlet allocation** [5] (LDA) è un modello probabilistico generativo nato con lo scopo di classificare dei documenti individuando dei topics "astratti" all'interno della Bag-of-Words.
L'intuizione si basa sul fatto che i documenti siano rappresentati da un insieme misto di topics latenti, dove ogni topic è definito da un elenco di parole. La bontà di un topic si può misurare attraverso la *topic coherence*: un punteggio calcolato in base alla similarità tra le parole più rappresentative del topic. Questo valore permette di distinguere i topic semanticamente interpretabili da quelli ottenuti solamente dai metodi statistici.

La similarità di un documento a ogni topic è calcolato in base alla combinazione lineare tra le parole in comune tra le coppie topic-document pesate per i coefficienti specifici di quel topic per ogni termine. In questo modo potremmo classificare i documenti in base agli argomenti con i quali hanno

maggior similarità.

Esempio di topic prodotto da un modello LDA addestrato su paper con argomento gli scacchi:

```
0.015*"player" + 0.014*"data" + 0.012*"rating" + 0.010*"table" +
0.008*"analysis" + 0.008*"tournament" + 0.007*"engine" +
0.006*"different" + 0.006*"average" + 0.006*"opening"
```

Ogni addendo è ottenuto dal prodotto tra una parola del topic e il coefficiente a essa associato.

1.3 Transformer per testi lunghi: Longformer e BigBird

I transformer hanno rivoluzionato il mondo del NLP venendo impiegati per numerose tasks, come *summarization*, *question answering*, *text generation* e *text classification*.
Data la vastità e la complessità dell'argomento, che il sottoscritto non ha trattato in nessun corso universitario, si è deciso in questa tesi di approfondire solamente il discorso riguardante all'*embedding* dei termini di un documento, con la finalità di eseguire dei modelli di classificazione non supervisionata.

Con i transformer è possibile ottenere un risultato molto simile all'impiego dei modelli di Word Embedding: si tratta di processare dei documenti per effettuare il mapping delle parole in vettori con il fine di renderli interpretabili dai modelli di machine learning.
Nell'ambito del Deep Learning, l'avvento dei transformer ha portato grandi progressi, superando lo stato dell'arte ottenuto con i modelli di Word Embeddings classici (come *Glove*).
Il modello Transformer è stato introdotto per la prima volta nel 2017 da Vaswani [6]. L'architettura proposta da questo lavoro è stata poi utilizzata nel 2018 da Devlin per la creazione di **BERT** [7] (*Bidirectional Encoder Representations from Transformers*): un *pre-trained language model* che riuscì a sua volta a battere lo stato dell'arte raggiunto fino a quel momento.
Nonostante sia stato progettato solo tre anni fa, BERT è già stato ampiamente superato da altri modelli di transformer, come con l'uscita di **RoBERTa** [8] nel 2019.

Dopo avere effettuato l'embedding dei termini, si ottiene una matrice con

una riga per ogni termine e 768 colonne (numero di dimensioni in cui una parola viene trasformata dopo avere effettuato l'embedding). Per ricavare un singolo vettore da questa matrice, si possono applicare diversi metodi tra cui delle *strategie di pooling*: come prendere il valore massimo di ogni colonna (*max pooling strategy*) o il valore medio (*mean pooling strategy*). Altri metodi implicano l'utilizzo del CLS *Classification Token*: un token posto all'inizio di ogni sequenza di token, che può essere impiegato nelle task di classificazione dato che aggrega il valore della sequenza stessa.
La scelta tra una strategia di pooling o del token CLS dipende dalla task da soddisfare e dal caso di studio. In questa tesi verranno testate entrambe le tecniche.

Uno dei limiti di BERT (e di molti altri transformer) è la lunghezza della sequenza, limitata a 512 token per non incorrere in alti costi computazionali. Per eseguire l'embedding anche su testi di lunghezza superiore, sono stati implementati nuovi transformer tra cui il **LongFormer** [9] e il **BigBird** [10], proposti rispettivamente nell'aprile e nel luglio 2020. Essi si basano su degli algoritmi che mantengono il costo computazionale accettabile anche per sequenze di migliaia di token. Nello specifico il limite di token è stato spostato da 512 a 4096 token.
I due transformer citati sono tra i più efficienti per queste task e hanno portato un miglioramento delle performance e dei risultati rispetto allo stato dell'arte [5]. Si può notare nella Figura 1.5 come il modello BigBird riesca a ottenere un ottimo score e allo stesso tempo a limitare le risorse necessarie per l'esecuzione.

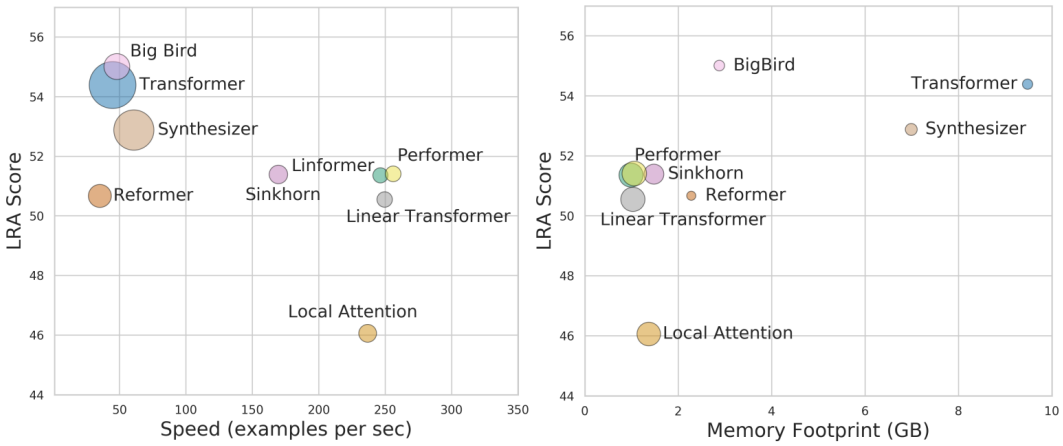


Figura 1.5: Trade-off tra performance e risorse utilizzate (tempo a sinistra e memoria a destra).
Fonte: LONG RANGE ARENA: A BENCHMARK FOR EFFICIENT TRANSFORMERS [11].

[5] In accordo con i test effetuati dal progetto *Long-Range Arena* [11]

1.4 Modelli di classificazione non supervisionata (clustering)

Se si utilizza un dataset con dati etichettati, ovvero se ogni istanza è associata a una classe o a un label, allora possono essere applicati dei modelli di machine learning supervisionati. Questi sfruttano le associazioni ricavabili tra il label e gli altri parametri dell'istanza per predire o classificare le classi di istanze che non hanno mai visto.
In alcune situazioni reali, risulta però molto complicato possedere un dataset di notevoli dimensioni con istanze tutte etichettate. Questo rende impossibile l'impiego di metodi supervisionati. Quindi, non avendo a disposizione i label, bisogna ricavare delle correlazioni o similarità tra le istanze del dataset per riuscire, come nel nostro caso di studio, a classificarle correttamente. In altre parole, si cerca di raggruppare tutte le istanze simili tra loro in quelli che vengono denominati *cluster*.
Questo procedimento di **classificazione non supervisionata**, o **clustering**, cerca di adempiere a questo scopo sviluppando dei modelli di machine learning appositi, che non abbiano bisogno di label.

1.4.1 K-Means

Il K-Means[12] è uno dei modelli di clustering più conosciuti e semplici da utilizzare. Si basa sul concetto di **centroide**: un punto nello spazio situato in una posizione media rispetto a tutti gli altri punti del cluster a cui appartiene. È quindi un punto trovato dall'algoritmo e non ha corrispondenza nel dataset.
Il parametro fondamentale di questo algoritmo è il numero di cluster: questo deve essere scelto a priori e viene indicato con K.
Per prima cosa vengono presi un numero k di centroidi in maniera randomica (l'importante è che questi siano distinti e siano distanti tra loro).
Il passo successivo consiste nel calcolare la distanza di ogni punto del dataset da ogni centroide, per poi associare ogni istanza a quello più vicino.
Viene poi ricalcolata la posizione di ogni centroide basandosi sulla media delle coordinate dei punti associati al cluster e si ripete l'algoritmo.
Il procedimente ricorsivo termina quando non cambiano più le istanze associate a ogni centroide o le coordinate di quest'ultimo.

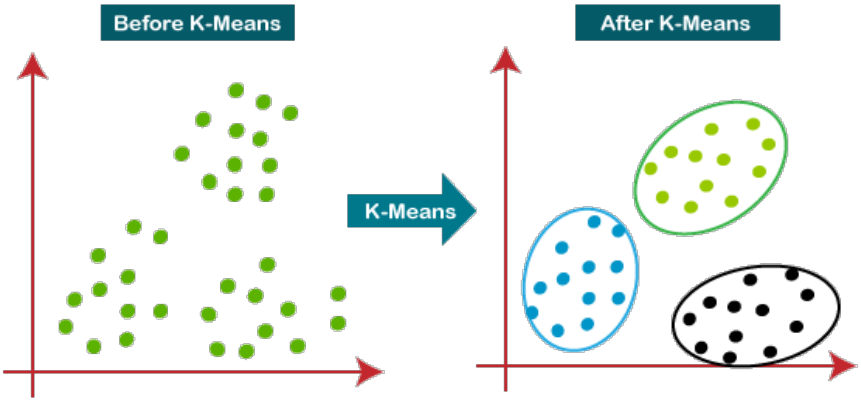


Figura 1.6: Esempio di clusterizzazione con k-means.
Fonte: https://www.javatpoint.com/k-means-clustering-algorithm-in-machine-learning

Quindi, l'algoritmo ricerca le coordinate ottimali per i centroidi e dipende perciò dal numero di essi, ovvero dal numero di cluster specificato come parametro. È importante scegliere un valore opportuno. Per questo scopo, esistono alcune tecniche per trovare il numero di cluster ottimale.
Il metodo più utilizzato è l'**elbow method** (metodo del gomito) che si basa sul concetto di WCSS (*Within Cluster Sum of Squares*), che definisce la variazione totale all'interno di un cluster. In altre parole, il WCSS è uguale alla somma delle distanze al quadrato dei punti di un cluster. È sufficiente eseguire l'algoritmo K-Means per un range di K prestabilito e calcolare il WCSS di ogni risultato. Infine, bisogna disegnare tali risultati in un piano cartesiano, in cui l'asse delle ascisse rappresenta il numero di cluster e l'asse delle ordinate rappresenta il WCSS.

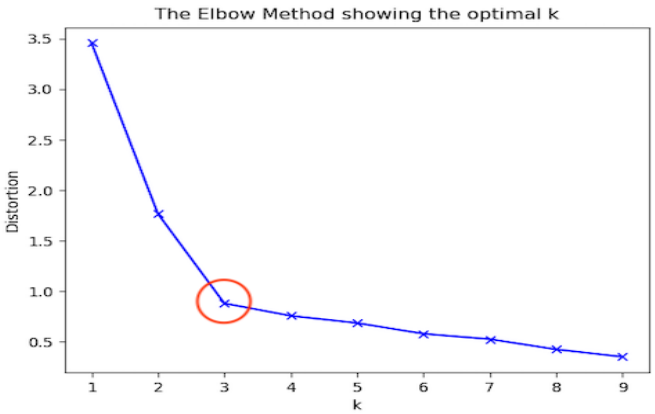


Figura 1.7: Plot dei valori di WCSS.
Fonte: https://www.developersmaggioli.it/blog/apprendimento-non-supervisionato-clustering-k-means/

Guardando il grafico, bisogna scegliere il numero di cluster in cui è presente l'angolo "a gomito" della curva.

1.4.2 DBSCAN

L'algoritmo K-Means rientra nella tipologia di clustering denominata con *centroid-based method* e divide il dataset in gruppi come mostrato nel paragrafo precedente. Esistono anche altri metodi di clustering, tra cui i **Density-Based Clustering**. Un algoritmo appartenente a tale gruppo si basa sull'idea di connettere le istanze in aree ad "alta intensità" in cluster.

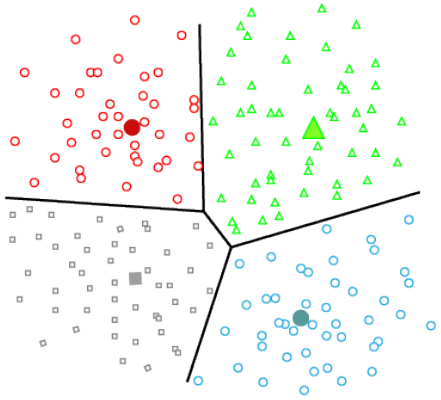


Figura 1.8: Centroid-Based method.

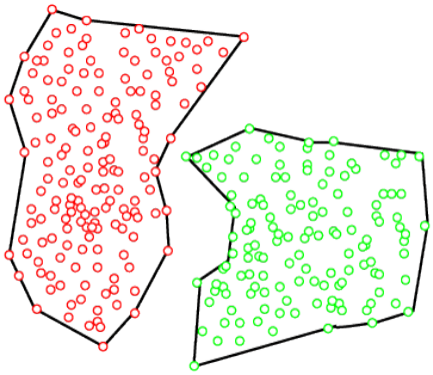


Figura 1.9: Density-Based method.

Fonte: https://www.javatpoint.com/clustering-in-machine-learning

Uno dei vantaggi di questi metodi è la capacità di individuare cluster con forme più complesse, come si può notare in figura.

Il **DBSCAN** (*Density-Based Spatial Clustering of Applications with Noise*) è un esempio di density-based model che riesce a separare le aree di alta densità da quelle di bassa densità. Il numero di cluster non può essere deciso in anticipo: dipende dalle aree che vengono trovate dall'algoritmo. Ciò che influenza tale numero sono i parametri che si possono fornire all'algoritmo, come il numero minimo di istanze necessarie per formare un cluster (*minimum cluster size*), ovvero un *core point*. Un altro parametro (chiamato *distance threshold epsilon*) serve per specificare la distanza massima che due istanze possono avere per essere considerate appartenenti allo stesso cluster: un numero troppo grande di questo valore può portare a un solo unico grande cluster, mentre un numero troppo basso potrebbe produrre un'elevata quantità di istanze classificate come rumore, ovvero come istanze a cui non è stato associato nessun cluster.

1.4.3 HDBSCAN

Mentre l'algoritmo DBSCAN ha bisogno dei due parametri sopracitati, HDBSCAN [13] è un'implementazione di DBSCAN che utlizza diversi valori di epsilon e ha bisogno di un solo parametro: il *minimum cluster size*.

Questo modello parte con l'obiettivo di trovare i *clustering core point* più robusti e resistenti possibili contro il rumore. L'idea alla base è quella di trasformare lo spazio dei dati basandosi sulle stime delle densità nelle varie aree dei dati, selezionare quelle con le maggiori densità e combinare i punti che non vengono ritenuti del rumore.
Il primo punto può essere eseguito in diversi modi, uno dei più comuni è il calcolo della *core distance*: la distanza di un punto dai suoi K punti più vicini. In aree di alta densità, ovvero dove possiamo riconoscere molti punti vicini, tale distanza sarà inferiore.

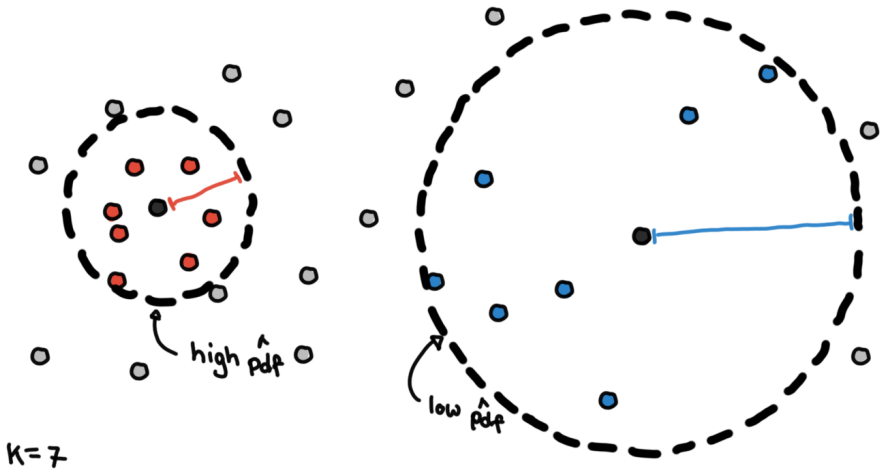


Figura 1.10: Esempio di calcolo della *core distance* con k=7.
Fonte: https://towardsdatascience.com/a-gentle-introduction-to-hdbscan-and-density-based-clustering-5fd79329c1e8

Nell'immagine i punti rossi rappresentano una zona ad alta densità. A questo punto si può scegliere un threshold che ci permetta di considerare queste aree come cluster e il rimanente come rumore. Questo approccio è però molto simile a DBSCAN.
Scegliere un unico threshold può portare a dei risultati insoddisfacenti nel caso esistano cluster con densità diverse: un valore troppo alto potrebbe produrre troppo rumore (come nel caso in cui un cluster abbia un'area di alta densità al centro e un'area meno densa ai margini), mentre un valore troppo basso potrebbe unire due cluster diversi spazialmente vicini.
HDBSCAN costruisce una gerarchia memorizzando quali aree di densità sono

più vicine a fondersi tra loro, ovvero quali picchi di densità debbano essere considerati appartenenti allo stesso cluster. Per compiere questa scelta l'algoritmo utilizza delle euristiche basate sulla robustezza delle aree di densità: si contano i loro punti circostanti per vedere se la somma di essi è maggiore della somma dei punti delle due aree, in tal caso conviene unire i due cluster, in caso contrario è opportuno mantenerli separati.

1.4.4 Classificazione con LDA

Nella sezione 1.2 abbiamo descritto il funzionamento dell'algoritmo LDA. In questo paragrafo viene descritto in breve come è possibile sfruttare i topic per raggruppare i documenti in cluster.
Ogni documento viene analizzato come un insieme di uno o più topic. Addestrando il modello LDA per n topics, otterremo una matrice documenti-topics dove ogni cella contiene la similarità (più precisamente la *coherence topic*) tra un documento e un topic.

```
         coherence                topic words
Topic 0 :  0.00015 => performance, memory, skill, study, rating
Topic 1 :  0.03199 => le, world, like, computer, player
Topic 2 :  0.86470 => search, position, value, tree, program
Topic 3 :  0.10314 => learning, system, player, board, problem
```

Nell'esempio sopra, si può vedere un modello LDA addestrato con 4 topic e il punteggio di ognuno di questi per un'istanza del dataset. Il "Topic 2" è nettamente il più coerente con questo documento. Nell'ultima colonna sono mostrate le prime 5 parole più rilevanti per il topic.

Possiamo associare ogni istanza del dataset con il topic con il punteggio più alto, in questo modo i topic saranno considerati come dei cluster che classificano i documenti in base al loro topic più rilevante.
Questo potrebbe causare dei topic senza alcuna istanza associata. Ciò può accadere nel caso ci siano degli argomenti più rilevanti e diffusi nel corpus di documenti o nel caso in cui il numero di topic da individuare sia molto alto.
Il numero di topic ideale dipende dal caso di studio, quindi è difficile stabilirlo a priori.

1.5 Rappresentazione dei dati con t-SNE

Abbiamo visto come con LSA può risultare utile e funzionale ridurre le dimensioni della matrice documenti-termini. La riduzione della dimensionalità può avere anche un altro scopo come quello di ridurre le dimensioni dei vettori a 2 o 3, in modo che possano essere graficate e confrontabili anche da noi umani. Applicando queste tecniche si cerca di comprimere tutte le informazioni contenute nelle colonne del dataset in poche dimensioni. È quindi probabile una perdita di informazioni: l'obiettivo è riuscire a scartare per prime le informazioni poco rilevanti.
Le tecniche più comuni per questo scopo, sono PCA e t-SNE, mentre per ridurre la dimensionalità di matrici da utilizzare per addestrare dei modelli di machine learning in tempi più rapidi, è fortemente consigliato l'utilizzo di tecniche come la fattorizzazione SVD (vedi paragrafo 1.1.3), la LSA o LDA.

La *Principal Component Analysis* (**PCA**) effettua una trasformazione non supervisionata: un mapping lineare cercando di mantenere le informazioni ricavabili dai pattern basati sulle correlazioni tra le features del dataset. Quindi l'algoritmo PCA cerca di mantenere la struttura globale dei dati.

La *t-distributed stochastic neighbourhood embedding* (**t-SNE**) [14] effettua una trasformazione non lineare e non supervisionata. L'algoritmo è matematicamente complesso, ma si fonda su un'idea semplice: mantenere i punti vicini nello spazio ad alta dimensionalità, anche nello spazio a 2 o 3 dimensioni. Risulta quindi particolarmente utile e efficace per visualizzare graficamente i dati.
É ritenuta una delle tecniche migliori per la riduzione della dimensionalità e superiore al PCA. Quest'ultimo, infatti, soffre molto la presenza di outliers e non utilizza iperparametri. L'algoritmo t-SNE può essere invece influenzato dall'assegnazione di parametri come il *learning rate* e il *number of steps*.

In questa tesi, per la riduzione della dimensionalità delle matrici impiegate dai modelli di machine learning è stata applicata la tecnica LSA, mentre per rappresentare il database o i topic ottenuti tramite LDA è stato utilizzato l'algoritmo t-SNE.

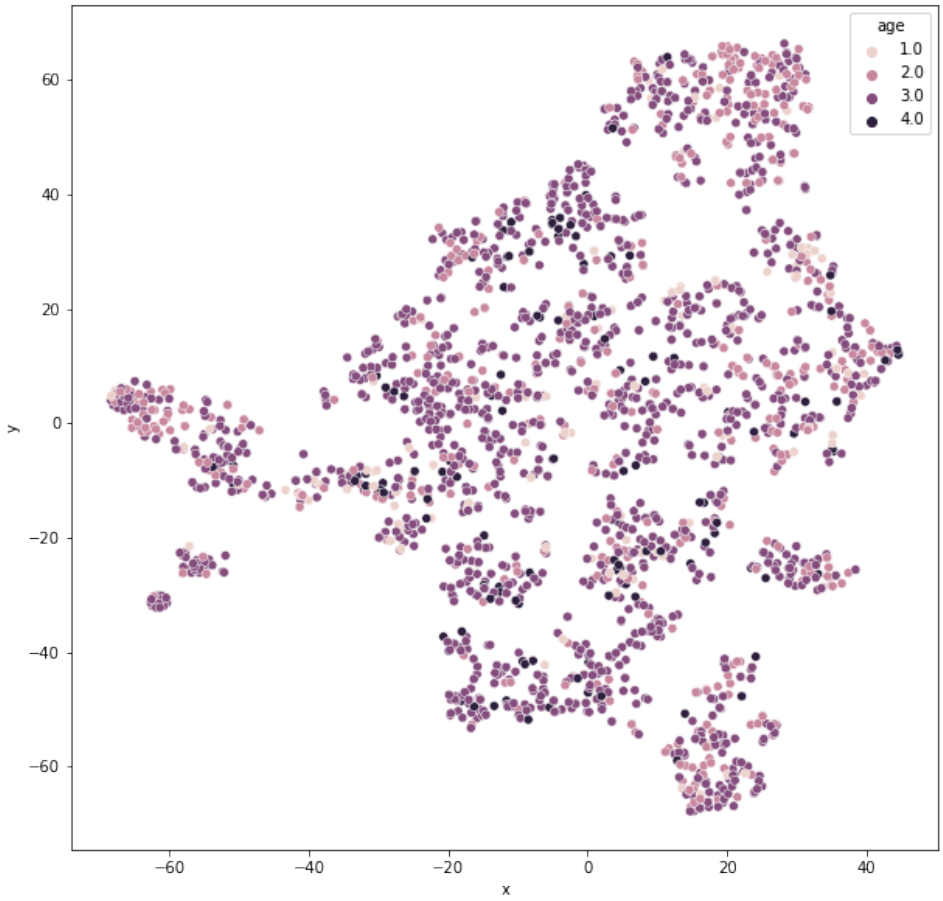


Figura 1.11: Grafico ottenuto con t-SNE del dataset di questo caso di studio.

Capitolo 2

Analisi del caso di studio

Il caso di studio su cui sono state applicate tutte le tecniche e teorie spiegate nel precedente capitolo riguarda una raccolta di articoli e documenti sugli scacchi scritti tra il 1950 e il 2021. In particolare, gli argomenti trattati in questi paper riguardano il parallelismo tra il mondo degli scacchi e del *computer chess* e quello dell'intelligenza artificiale.

> *The History of Computer Chess is a good proxy for understanding the evolution of Artificial Intelligence methods and technologies.*

Sono state poi individuate 4 ere storiche [**?**] che derivano dal lavoro e dallo studio del prof. Paolo Ciancarini, il quale, da esperto della materia, ha individuato gli anni in cui l'intelligenza artificiale ha portato i più grandi cambiamenti nel gioco degli scacchi.

L'obiettivo è quello di osservare se esiste una corrispondenza tra le classi proposte e il contenuto dei file: processando e classificando gli articoli del dataset a disposizione con modelli e tecniche non supervisionati.

Per valutare la qualità del dataset sono stati impiegati anche test statistici come il test chiquadro (vedi paragrafo 2.1) e dei modelli di classificazione supervisionata (vedi paragrafo 3.4.5) per confrontarne i risultati.

Le fasi storiche individuate sono le seguenti:

- **First Season 1950-1977**: Il mondo dei computer e quello degli scacchi si incontrano grazie al lavoro di **Claude Shannon** e **Alan Turing**. In particolare, Shannon individua due architetture software per il gioco degli scacchi: **Type-A** e **Type-B**. I programmi del primo tipo si basano su degli algoritmi greedy che esplorano la struttura dati denominata **game-tree**. I programmi Type-B sono invece considerati più intelligenti,

poiché esplorano solo una parte delle possibili combinazioni in base ad alcune euristiche che, in certi casi, tentano di imitare il comportamento umano.

- **Second Season 1977-1997**: Questa era è determinata dall'invenzione e dall'evoluzione di hardware speciali, tra cui le macchine capaci di eseguire computazioni parallele. Questa stagione si conclude con il celebre incontro tra **Kasparov** e il software **Deep-Blue**.
- **Third Season 1997-2017**: In questa fase i progressi svolti in ambito software hanno portato a programmi eseguibili anche su computer portatili e a soluzioni di analisi del game-tree sempre più efficienti.
- **Fourth Season 2017-**: La fase storica corrente include i programmi basati sul **deep learning**. Ha inizio nel 2017 quando **Google** rilascia **Alpha Zero**: un software addestrato a partire da una conoscenza nulla, che impara giocando contro se stesso fino a vincere contro **Stockfish 8**, un programma *game-tree based* molto potente.

Per valutare l'accuratezza dei modelli e delle tecniche implementate sono stati etichettati tutti quei file che avevano l'anno di pubblicazione all'interno del nome. Tale informazione ci ha permesso di suddividere i file attribuendogli automaticamente le ere storiche di appartenenza.

I paper sono stati raccolti e memorizzati digitalmente in diversi formati: pdf, testuali, immagini e altri meno rilevanti. Alcuni di questi sono stati ottenuti scannerizzando i documenti fisici.

	Count	Size (MB)
pdf	2221	1775.32
docs	56	42.97
webpages	19	0.60
images	2	0.13
other_formats	43	22.71

Figura 2.1: Contenuto della cartella del caso di studio.

Quindi, il primo problema da risolvere è stato quello di leggere tutti questi file e salvarne il contenuto in memoria come testo (vedi paragrafo 3.1).

2.1 Verifica indipendenza tra paper e le ere storiche con il test statististico Chiquadro

Il test chiquadro è particolarmente utile per risolvere il problema di selezione delle variabili (*features*) di un dataset: avere delle variabili fortemente correlate può causare problemi ai modelli di machine learning supervisionati e non supervisionati, è opportuno quindi mantere solo le feature di maggiore utilità.

Il test chiquadro χ^2 è un **metodo statistico** di verifica d'ipotesi e può essere applicato per diversi scopi. Nel nostro caso, *vogliamo controllare se le informazioni contenute in un paper sono correlate all'era storica di appartenenza* (come dovrebbe essere) oppure se sono indipendenti. In altre parole, vogliamo capire se i paper sono rappresentativi delle ere storiche da noi utilizzate.

Secondo questo metodo, due eventi A e B sono indipendenti se

$$P(AB) = P(A)P(B) \quad \textit{oppure} \quad P(A|B) = P(A) \wedge P(B|A) = P(B)$$

In altre parole, se i due eventi non si influenzano.
In questo test, l'ipotesi nulla è proprio l'indipendenza tra i due eventi: se essa viene rigettata, significa che i due eventi sono correlati.
In particolare, vogliamo controllare la correlazione tra un documento e la propria era storica. Questo procedimento è molto simile a quello di controllare se una *query*, ovvero una serie di termini o parole chiave, appartiene a una determinata classe.

Una query tratta da un corpo di documenti può essere ridotta a un vettore e trattata proprio come un nuovo testo. Quindi, il primo passo da compiere è convertire il documento in una Bag-of-Words e applicare una tecnica come il tf-idf.
Dopo avere portato la query nello stesso spazio dimensionale dei documenti, possiamo calcolare la similarità coseno con ognuno di essi. Prendiamo poi gli n documenti più simili alla query, dove n è il numero di istanze appartenenti alla classe confrontata, e ricaviamo i seguenti valori:

- *true positive* (TP): documenti nel sottoinsieme appartenenti alla classe;
- *false positive* (FP): documenti nel sottoinsieme non appartenenti alla classe;
- *true negative* (TN): documenti non nel sottoinsieme e non appartenenti alla classe;

- *false negative* (FN): documenti non nel sottoinsieme e appartenenti alla classe.

Questi valori vengono combinati in una matrice chiamata *confusion matrix* (matrice di confusione).

		Real labels	
		Negative	Positive
Predicted Labels	Negative	TN	FN
	Positive	FP	TP

Tabella 2.1: Esempio di matrice di confusione per test chiquadro.

Questa matrice può essere utilizzata per effettuare il test chiquadro con la seguente formula:

$$\chi_c^2 = \sum_i \frac{(p_i - r_i)^2}{r_i}$$

dove p_i è il *predicted label* e r_i è il *real label* dell'i-esimo documento della lista. Mentre c indica il numero di *gradi di libertà*, ovvero il numero massimo di variabili all'interno dello spazio dimensionale che possono variare in maniera indipendente.

Nel nostro caso di studio, questa teoria è stata messa in pratica con la libreria `Scipy` di Python.
Dopo aver costruito la matrice di confusione come sopra, la si è passata come parametro alla funzione `scipy.stats.chi2_contingency`[1], la quale effettua un test di indipendenza sulla tabella e restituisce, nel seguente ordine, questi valori:

– `chi2` : il valore del test chiquadro; maggiore è il suo valore assoluto, più sarà probabile che l'ipotesi nulla sia rigettata.

– `p` : il p-value del test, indica la percentuale di probabilità che i due eventi siano indipendenti. Quindi, per valori bassi l'ipotesi nulla sarà rigettata.

– `dof` : gradi di libertà utilizzati.

[1] https://docs.scipy.org/doc/scipy/reference/generated/scipy.stats.chi2_contingency.html

– `expected` : i valori attesi ottenuti dal test chiquadro; si basa sulle somme marginali della tabella.

Il test è stato ripetuto per tutte le istanze del dataset per un numero di componenti della matrice LSA da 10 a 300 con un passo paria a 10: a ogni iterazione, veniva preso un documento di un'era specifica, applicato il tf-idf e la LSA per la riduzione della dimensionalità, calcolate le similirità coseno e infine presi gli n documenti più simili. I valori ottenuti sono stati mostrati nei grafici nella Figura 2.2, dove nella prima colonna viene mostrato il valore del p-value e nella seconda colonna la percentuale di volte in cui l'ipotesi nulla è stata accettata (confermando l'ipotesi di indipendenza).

I risultati ottenuti indicano che la nostra ipotesi nulla, ovvero l'indipendenza tra un paper e la propria era storica, è stata accettata più della metà delle volte con una probabilità fissata al 95%. In altri termini, *molto spesso non vi era correlazione tra un paper e la suddivisione in ere, a indicare che una buona parte di questi non possono considerarsi rappresentativi della fase storica di appartenenza.*

Questo test ha rivelato che il nostro dataset non si ben dispone alla classificazione se viene mantenuta questa suddivisione in ere. Una prima possibile spiegazione potrebbe essere il numero troppo basso di paper disponibili e la suddivisione sbilanciata in ere storiche (vedi paragrafo 2.3.).

2.2 Intervallo di confidenza

Il calcolo dell'intervallo di accuratezza di un modello di classificazione è molto utile per valutare l'affidabilità del risultato: ci suggerisce con che probabilità l'accuratezza in un caso reale si avvicinerà alla nostra stima.
Per prima cosa bisogna scegliere la probabilità che si desidera ottenere dell'intervallo di accuratezza. Da essa possiamo ricavare il valore di $\alpha \in [0, 1]$ attraverso la formula `prob = 1 -` α .
Supponendo di volere una probabilità del 95%, il valore di alpha è pari a 0.05 e possiamo ottenere anche il valore del quantile $Z_{\frac{\alpha}{2}}$ che in questo caso è pari a 1.96.
La formula per ottenere l'intervallo di confidenza avrà la seguente forma:

$$P\left(-Z_{\frac{\alpha}{2}} < \frac{acc - x}{\sqrt{\frac{x(x-1)}{N}}} < Z_{\frac{\alpha}{2}} \right) = 1 - \alpha$$

dove N è il numero di istanze, acc è l'accuratezza stimata del nostro modello e x è la variabile incognita da ricavare dall'equazione per ottenere l'intervallo di

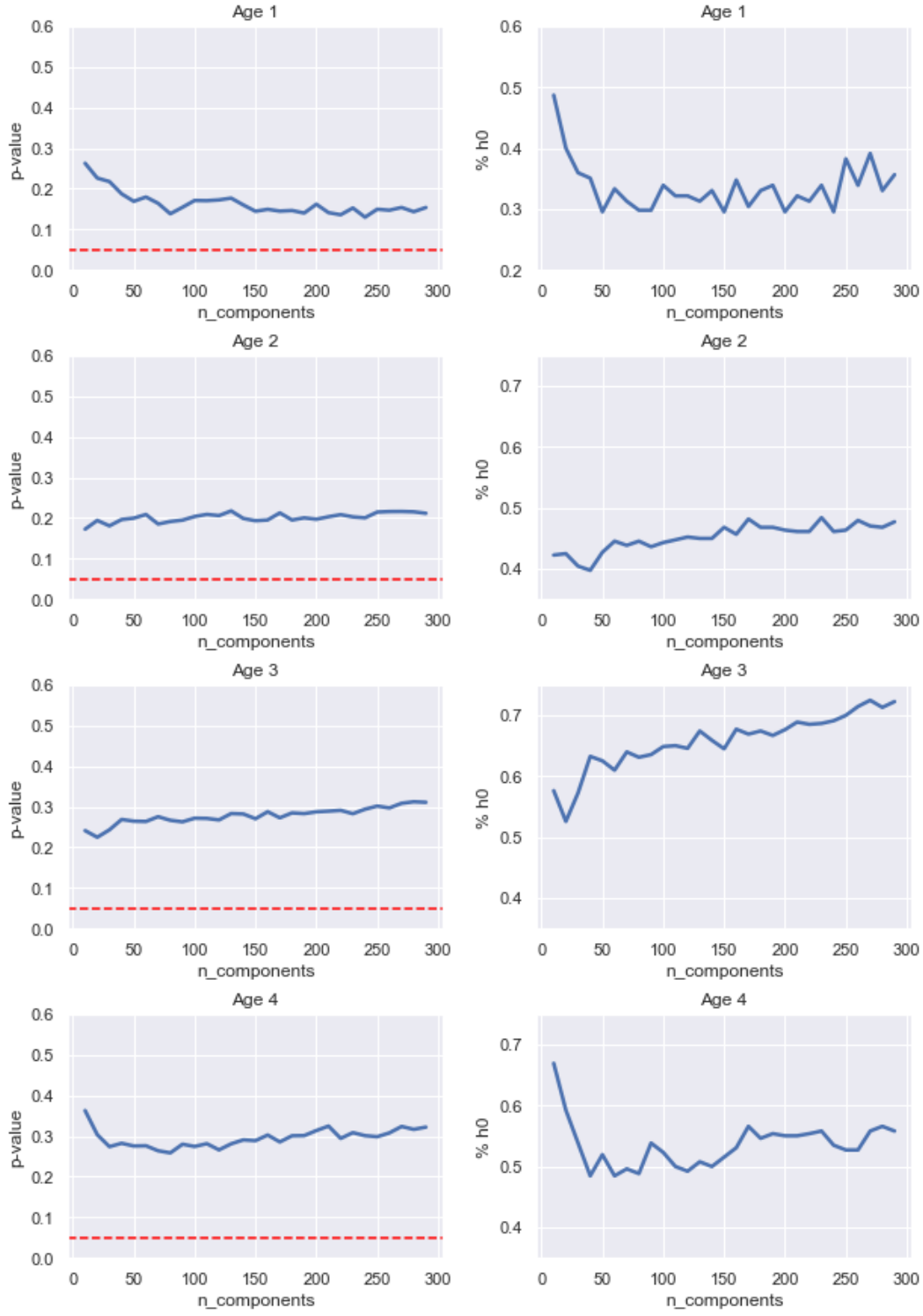


Figura 2.2: Risultati del test chiquadro effettuato sul nostro dataset.

confidenza.
La formula finale per ricavare il limite inferiore e superiore dell'intervallo è la seguente:

$$p = \frac{2 \cdot N \cdot acc + Z^2_{\frac{\alpha}{2}} \pm Z_{\frac{\alpha}{2}} \cdot \sqrt{Z^2_{\frac{\alpha}{2}} + 4 \cdot N \cdot acc - 4 \cdot N \cdot acc^2}}{2(N + Z^2_{\frac{\alpha}{2}})}$$

Supponendo di ottenere un'accuratezza del 60%, i paper utilizzati per validare i modelli, ovvero i paper che sono stati etichettati con successo, sono in totale 2054 (valore assegnato a N). Con questi dati possiamo ricavare x e ottenere l'intervallo di confidenza qui sotto riportato.

$$58\% < \mathbf{60\%} < 62\%$$

In conclusione, l'accuratezza che otterremmo dai nostri modelli sarà abbastanza affidabile con un margine di errore del 2%.

2.3 Analisi esplorativa

In questa sezione effettueremo un'analisi della distribuzione ottenuta suddividendo i paper in ere storiche.
I grafici sono stati realizzati con le librerie `matplotlib` e `seaborn` di Python utilizzando il dataframe costruito sui dati dei paper[2].

Come primo passo grafichiamo la distribuzione delle ere storiche tra i documenti.

[2]Vedi paragrafo 3.2

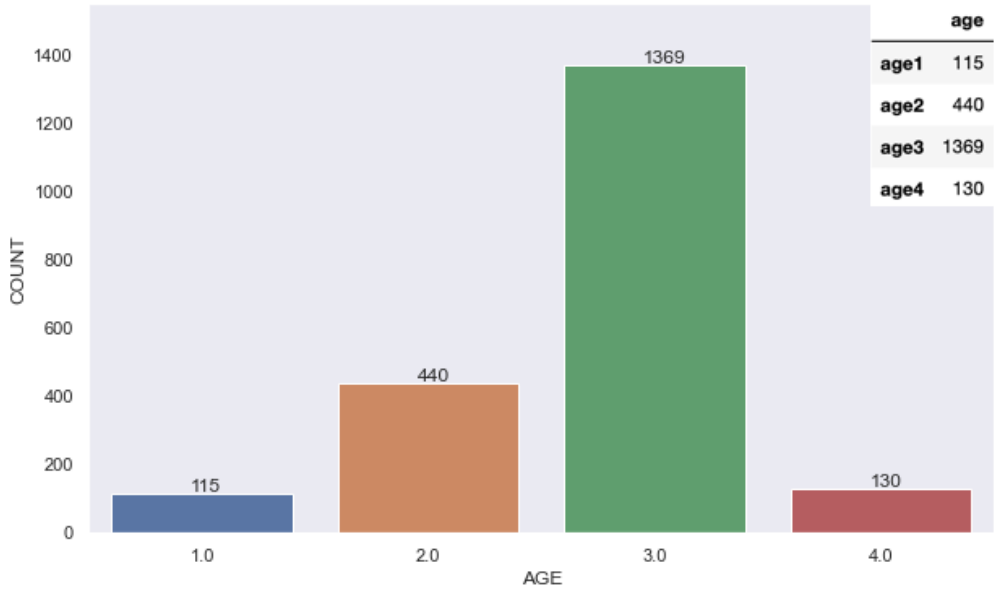


Figura 2.3: Distribuzione ere storiche tra i paper.

Si nota subito uno sbilanciamento del dataset in favore delle terza era storica (quella tra il 1977 e il 2017). Questo è facilmente motivabile dal fatto che è un periodo molto lungo in cui sono stati prodotti diversi paper, soprattutto ottenibili in formato digitale. Al contrario, nella prima era (1950-1977) è molto più complicato ottenere paper in formato digitale o testuale da scannerizzare. Inoltre, dato che gli argomenti trattati nei paper sono relativi al mondo dell'AI, è normale che il numero di documenti aumenta con il progresso di quest'ultima. Anche la quarta ed ultima era, essendo molto corta (circa 4 anni) e non essendosi ancora conclusa, è presumibile abbia un numero ridotto di paper.

La distribuzione di paper negli anni si può osservare meglio nella Figura 2.3: si nota la differenza dei paper raccolti negli anni successivi al 2000. In alcuni anni essi superano il centinaio, come l'intero numero di paper della prima o quarta era a disposizione.

Nella Figura 2.4 si può osservare come in alcuni anni i paper raccolti nella prima era sono limitati a uno o due. Inoltre, questi paper sono risultati come i più complicati da leggere con tecniche automatiche: a volta a causa di documenti fisici scannerizzati e memorizzati in bassa definizione.

Nella figura 2.5 è rappresentata la distribuzione dei paper negli anni della quarta era, in questo caso il basso numero totale di essi è dovuto al numero troppo esiguo di anni.

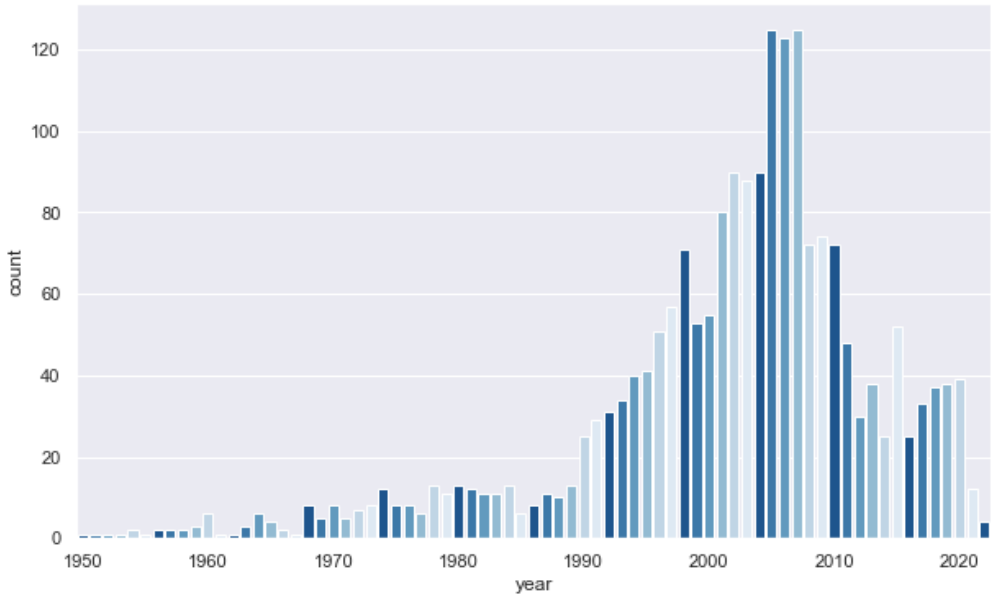


Figura 2.4: Distribuzione dei paper per ogni anno dal 1950 al 2021.

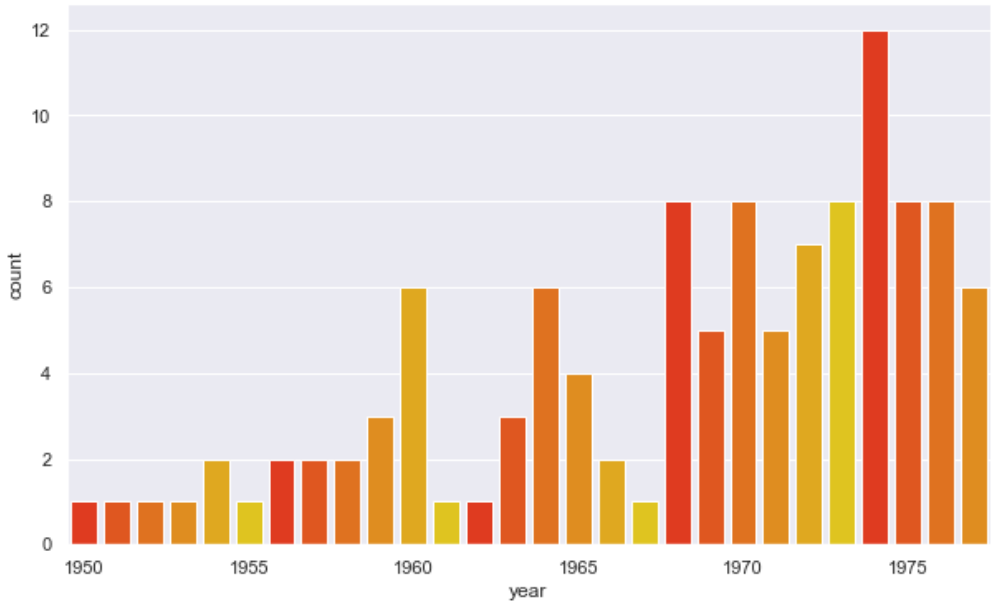


Figura 2.5: Distribuzione dei paper per ogni anno della prima era, dal 1950 al 1977.

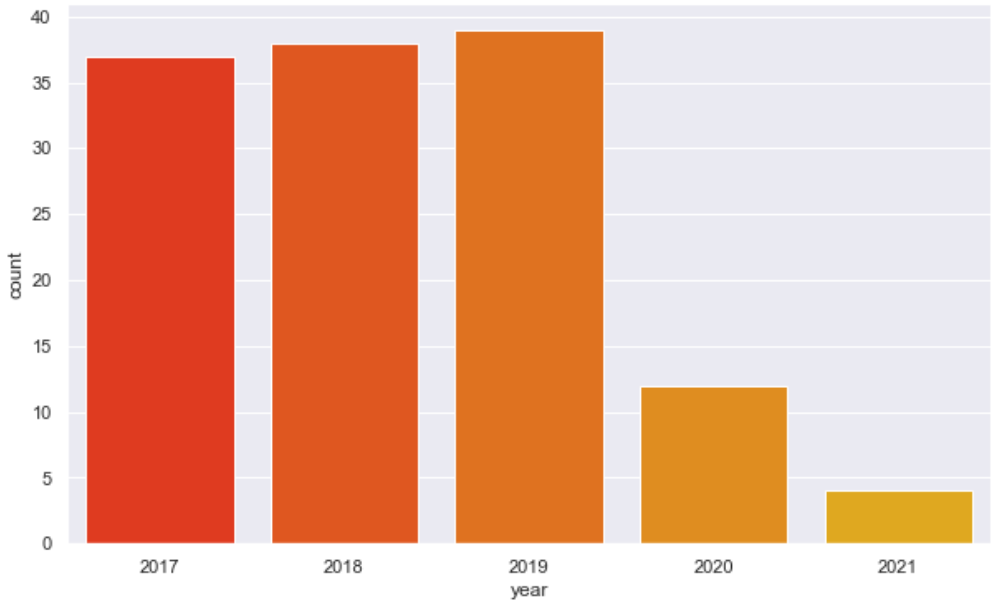


Figura 2.6: Distribuzione dei paper per ogni anno della quarta era iniziata nel 2017.

In conclusione, vi è un evidente problema di **sbilanciamento delle classi** attribuite ai paper attraverso la divisione in fasi storiche. Per questo motivo, sono stati anche svolti dei test con i modelli di classificazione non supervisionata dopo aver applicato le tecniche di bilanciamento descritte nel primo capitolo[3].

[3]Vedi paragrafo 1.1.5

Capitolo 3

Applicazione della teoria con Python

In questo capitolo verranno applicate al nostro caso di studio tutte le teorie introdotte nel Capitolo 1 .
Riassumendo quanto detto nel precedente capitolo,

> *la nostra analisi vuole provare o confutare l'ipotesi che i paper sugli scacchi siano classificabili, tramite modelli di machine learning, nelle ere storiche di appartenenza*[1].

Prima di arrivare alle tecniche di machine learning, si è dovuto analizzare la directory contenente tutti i paper per poterli leggere e memorizzare su disco uno alla volta. Gli articoli in questione sono per la maggioranza in formato pdf, ma sono presenti anche in altri formati, come mostrato all'inizio del capitolo 2. Nella prima sezione ci si è per cui focalizzati nel salvataggio del contenuto dei documenti in memoria sotto forma di *stringhe*.

Il passo successivo è stato quello di costruire un DataFrame apposito con la libreria `Pandas` di Python, in modo da poter richiamare facilmente i testi dei paper ogniqualvolta fosse necessario. Inoltre, nel DataFrame sono stati memorizzati anche gli anni di tutti i paper, laddove questa informazione fosse ottenibile all'interno del nome: questo ha permesso di calcolare le metriche per quantificare la bontà di ogni classificazione.

Nella sezione 3.3 sono state combinate le tecniche di Word Embeddings e t-SNE per avere una rappresentazione delle classi tramite delle keyword rappresentative.

[1] Vedi inizio Capitolo 2.

Infine, nell'ultima sezione vengono mostrati i codici e i risultati ottenuti da tutti i modelli.

3.1 Lettura dei file

In questa sezione viene analizzata la directory messa a disposizione come caso di studio: i file verranno differenziati in base al formato, in modo da poterli studiare ed elaborare in maniera opportuna.
Il passo successivo è quello di impiegare delle librerie specifiche per i formati individuati al fine di leggere il testo contenuto nei documenti.

3.1.1 Analisi della directory

I file raccolti hanno una dimensione complessiva di poco inferiore a 2GB (Figura 3.1). Si noti che la maggior parte dei file è in formato pdf: questo ci aiuterà a scegliere il package migliore per effettuare la lettura come descritto nel prossimo paragrafo.

I formati non riconosciuti sono per la maggior parte file eseguibili ed essendo in quantità esigua li ignoreremo nelle nostre analisi successive. Per lo stesso motivo sono stati ignorate le immagini e i file HTML/HTM.

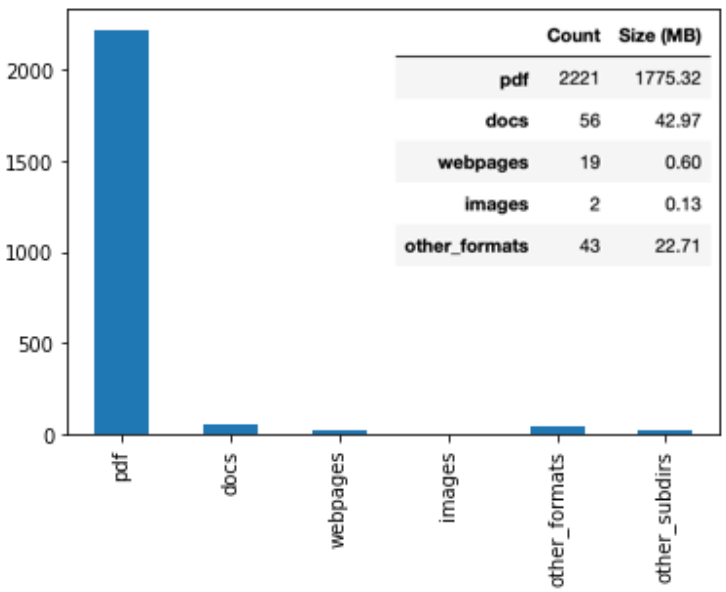


Figura 3.1: Distribuzione dei formati nei file.

3.1.2 Lettura formato pdf

Data la grande quantità di pdf, ci si è inizialmente concentrati sui modelli per la lettura di quest'ultimi. Il package che è stato utilizzato è chiamato `pdftotext` ed è un wrapper di un'altra libreria di più basso livello: `Poppler`[2].

Poppler is a PDF rendering library based on the xpdf-3.0 code base.

La selezione della libreria da utilizzare si è basata su vari test valutati in base alla correttezza nella lettura del testo e nella velocità di esecuzione. In particolare, l'utilizzo del package `pdftotext` è risultato nettamente più rapido rispetto all'utilizzo di un OCR (*Optical Character Recognition*) come `PyTesseract`: sul computer in cui il codice è stato eseguito, i tempi di lettura per duemila pdf sono stati inferiori ai 2 minuti impiegando `pdftotext`, mentre con `PyTesseract` i tempi stimati superavano le 24 ore.
Ecco il codice per leggere i file pdf sfruttando le path raccolte nel paragrafo precedente e memorizzate in un dizionario.

```
import pdftotext

pdf_page_counter = 0
errors = []
for path in dir_content['pdf']:
    # gestiamo l'eccezione per non bloccare il ciclo
    try:
        # apro il file in modalità lettura
        with open(path, "rb") as f:
            # estraggo il testo dal pdf
            pdf = pdftotext.PDF(f)
            pdf_page_counter += len(pdf)
            # concateno tutte le pagine
            paper['pdf'][path] = "  ".join(pdf)
    except:
        errors.append(path)
```

Figura 3.2: Codice per leggere i file pdf.

Nessun file ha provocato un errore durante l'esecuzione. In totale le pagine analizzate sono state circa 35 mila.

Successivamente, si è analizzata la lunghezza dei risultati prodotti dalla lettura per individuare quali file hanno prodotto un testo corto (inferiore ai 300 caratteri), il quale è un sintomo di lettura "sporca" del file. In questo modo sono state raccolte circa 180 path di file non letti correttamente, circa il 10% sul

[2]Sito web ufficiale: https://poppler.freedesktop.org/

totale.
Inoltre, un'ulteriore analisi ha mostrato che questi file appartenevano soprattutto alla prima era storica degli scacchi (antecedente al 1977), dove il numero di paper raccolti era nettamente inferiore (probabilmente a causa della difficoltà di reperire articoli e paper di quegli anni).

Per non perdere quindi tutti questi dati preziosi si è deciso di impiegare un modello OCR nella speranza che questo riesca a eseguire una lettura più accurata con successo. Per questo compito è stato adottato il già citato `PyTesseract`: un wrapper per il *Google's Tesseract-OCR Engine*. Quest'ultimo può essere impiegato nella lettura di immagini eseguendolo da terminale. Tuttavia, `PyTesseract` forniva delle prestazioni molto simili e si è deciso di impiegarlo per comodità e per continuare a sfruttare Python come linguaggio di programmazione.
Per utilizzare questo package, è necessario trasformare i pdf in immagini, procedimento che aumenta i tempi di esecuzione e che può essere svolto tramite la libreria `pdf2image`. Inoltre, ulteriori test hanno mostrato un'accuratezza e una velocità superiore nella lettura di immagini in cui è stato applicato un filtro che modifica la colorazione sulla scala dei grigi. Le librerie `OpenCV` e `Numpy` sono state impiegato per eseguire quanto detto.

Il codice sottostante riporta una funzione che impiega tutte le librerie appena citate e restituisce il contenuto del pdf passatole come parametro.

```python
import cv2
import numpy as np
import pytesseract
from pdf2image import convert_from_path

# Funzione per leggere file con PyTesseract
def read_file_with_pytesseract(path):
    # converto il pdf in immagine
    images = convert_from_path(path)
    # istanzio un testo vuoto
    text = ''
    for img in images:
        # applico il filtro all'immagine
        data = cv2.cvtColor(np.asarray(img), cv2.COLOR_BGR2GRAY)
        # concateno ogni pagina al testo già letto
        text += ' ' + pytesseract.image_to_string(data)
    return text
```

Figura 3.3: Codice per leggere un pdf con PyTesseract.

La lettura di 178 paper con questa funzione ha portato a un'attesa di ben 90 minuti.

Ripetendo il test della lunghezza minima di 300 caratteri, è risultato che solo 18 volte la lettura ha fallito, permettendo di recuperare il contenuto di 160 file.

Passiamo ora ad analizzare i file diversi dal formato pdf per selezionare quali di questi leggere.

3.1.3 Lettura di altri formati

Per leggere i formati diversi dal pdf è stato impiegato il package `textract`: questa libreria non fa altro che eseguire delle altre librerie specifiche per il formato di file passato in rassegna. Può essere considerato quindi un wrapper di alto livello. Essendo i file testuali facili da leggere, si è deciso di impiegarla per comodità dato che i tempi di lettura sono molto bassi (non si ottiene quindi grande vantaggio ad applicare i package specifici di basso livello per il formato del documento). I 56 file testuali sono stati letti in soli 7 secondi, con una percentuale di successo pari al 93%.

```
import textract
# ciclo le path dei documenti di tipo testuale
for path in dir_content['docs']:
    try:
        paper['docs'][path] = textract.process(path)
    except:
        print(path)
```

Figura 3.4: Codice per leggere un file con `textract`.

A questo punto abbiamo letto tutti i file dei formati che ci interessavano: passiamo dunque alla creazione del DataFrame e al preprocessing dei dati.

3.2 Costruzione Dataframe

In questa sezione viene creato e salvato il dataframe con le istanze e le colonne che verranno utilizzate nei modelli di classificazione dei prossimi capitoli.

Di seguito è riportato il codice per la creazione del DataFrame con la libreria `Pandas` e per aggiungere le colonne relative all'era storica e all'anno (Figura 3.5). Come già detto nell'introduzione di questo studio, i file in possesso non erano etichettati, ma molti di questi avevano l'anno all'interno del nome. È stato scelto di sfruttare questa informazione laddove fosse disponibile per calcolare le metriche di validazione delle classificazioni effettuate nei prossimi capitoli.

```
df = pd.DataFrame(
    all_papers.values(),
    index=all_papers.keys(),
    columns=['Content']
)
```

```
df['year'] = np.zeros( df['Content'].size ) * np.nan
df['age'] = np.zeros( df['Content'].size )
```

```
for i in df.index:
    try:
        year = int(i.split('/')[-1][0:2])
        year = year+1900 if year>21 else year+2000
        df['year'][i] = year
        df['age'][i] = 4 if year >= 2017 else \
                       3 if year >= 1997 else \
                       2 if year >= 1977 else \
                       1
    except:
        continue
```

Figura 3.5: Codice applicato per la costruzione del DataFrame

```
df.head()
```

	Content	year	age
articoli/articoli/05ChowErikssonFan.pdf	Chess Tab...	2005.0	3.0
articoli/articoli/60Reider.pdf	Reider, Norman, CHESS, OEDIPUS AND THE MATER D...	1960.0	1.0
articoli/articoli/95OnofrijCuratola.pdf	t " ~'7.¢ ,;ilELSEVIER ...	1995.0	2.0
articoli/articoli/07Guid.pdf	Factors Affectin...	2007.0	3.0
articoli/articoli/92MazurBooth.pdf	Testosterone and Chess Competition Allan Mazu...	1992.0	2.0

Figura 3.6: Prime 5 righe del dataframe.

Il DataFrame generato ha 2273 righe di cui 206 senza anno. Analizzando questi 206 file si è tentato di etichettarne alcune manualmente. Le righe rimanenti verranno scartate nei modelli di classificazione per poter effettuare il calcolo dell'accuratezza dei modelli.

3.2.1 Preprocessing del testo

Il preprocessing effettuato sui testi dei documenti può essere riassunto nei seguenti steps:

- Eliminazione righe con valori NaN nel campo `year`
- Decoding del testo memorizzato con il formato ASCII
- Rimozione bibliografia o referenze
- Creazione di un tokenizer che sfrutti le tecniche descritte nel primo capitolo[3]

Per primo, eliminiamo tutte le righe non nulle (Figura 3.7) per mantenere solo i paper etichettati che utilizzeremo per validare i nostri modelli.
Durante le analisi effettuate, si sono riscontrate delle problematiche relative alla conversione di testo in formato ASCII, in particolare in quei file dove le librerie di lettura (`pdftotext`, `PyTesseract` e `textract`) non sono riuscite ad effettuare una lettura pulita del documento. Essendo i dati in formato ASCII non utilizzabili in alcune funzioni e modelli, è risultato comodo effettuare una decodifica a priori. Tale passaggio fallisce nel caso in cui lo si effettui su una semplice stringa: per questa ragione il codice è stato inglobato in un `try except`.

```
df.year.isna().sum(), df.year.size
```

```
(201, 2255)
```

```
df.dropna(inplace=True)
df.year.size
```

```
2054
```

Figura 3.7: Eliminazione valori NaN dal DataFrame.

```
def decode_ascii_content(text):
    try:
        return text.decode('utf-8')
    except:
        return text
df['Content'] = df['Content'].apply(lambda x: decode_ascii_content(x))
```

Figura 3.8: Decodifica ASCII.

[3]Vedi paragrafo 1.1.1.

Analizzando i paper a nostra disposizione, si è notato che molti di questi contenevano una bibliografia o la sezione relativa alle referenze in cui venivano citati altri documenti. La mole di dati o parole presenti in queste sezioni può potenzialmente confondere i modelli di machine learning e rendere la previsione meno accurata. Per questo motivo si è deciso di rimuovere dal testo queste parti: cercando nello specifico le parole "*bibliography*" e "*reference*" ed eliminando tutto il testo successivo all'ultima occorrenza di una di queste.

```
contents = []
for doc in range(df.index.size):
    # Converto il testo in minuscolo
    text = df.iloc[doc].Content.lower()
    # Per ognuna delle tue parole guardo se è presente nel testo
    for w in ['bibliography', 'reference']:
        if w in text:
            res = re.finditer(r'\W*('+w+')\W*', text)
            # Ricavo l'indice dell'ultima occorrenza della parola
            # nel testo
            index = [m.start(0) for m in res][-1]
            # Taglio il testo
            text = text[0:index]
    contents.append(text)
# Sovrascrivo il contenuto dei papers nel DataFrame
df['Content'] = contents
```

Figura 3.9: Rimozione bibliografia o referenze ad altri paper.

Infine, rimane da applicare la tokenizzazione del testo e il preprocessing per ogni singola parola (Figura 3.10). In ordine, le operazioni eseguite sono le seguenti:

1. Conversione di tutto il testo in minuscolo.
2. Tokenizzazione del testo in parole, rimuovendo numeri e punteggiatura.
3. Rimozione stopwords.
4. Rimozione parole non presenti nel vocabolario[4].
5. Lemmatizzazione di ogni parola.

A questo punto il DataFrame contiene anche una colonna con il testo trasformato dal tokenizer nella Figura 3.10, pronto per essere impiegato dai modelli (come tf-idf).

[4]È stato utilizzato un vocabolario di termini inglesi scaricato da internet e prodotto dalla società *Infochimps* che contiene 370mila parole rispetto alle 240mila parole contenute nel dizionario di WordNet, in cui mancavano termini come "turing" e "software".

```
import requests
import json
```

```
url='https://github.com/dwyl/english-words/raw/master/words_dictionary.json'
res = requests.get(url)
english_vocab = list(res.json().keys())
```

```
stoplist = nltk.corpus.stopwords.words("english") # Scarico stopwords
```

```
def tokenizer(text):
    # Converto il testo in minuscolo.
    text = text.lower()
    # Dividi il documento in parole rimuovendo numeri e punteggiatura.
    tokenizer = RegexpTokenizer(r"[a-zA-Z]+")
    tokens = tokenizer.tokenize(text)
    # Rimuovo parole di lunghezza inferiore ai due caratteri.
    tokens = [token for token in tokens if len(token) > 2 ]
    # Rimuovo stopwords e parole non esistenti nel vocabolario.
    tokens = [token for token in tokens if token not in stoplist]
    tokens = [token for token in tokens if token in english_vocab]
    # Applico la lemmatizzazione a ogni parola del documento.
    lemmatizer = WordNetLemmatizer()
    tokens = [lemmatizer.lemmatize(token) for token in tokens]
    return tokens
```

Figura 3.10: Preprocessing delle singole parole in ogni testo.

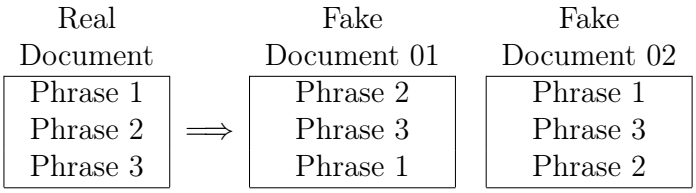
3.2.2 Bilanciamento delle classi

Durante la sezione relativa all'analisi esplorativa (vedi paragrafo 2.3), si è potuto osservare lo sbilanciamento delle classi. Come mostrato in tabella, la terza era storica ha un numero di istanze superiore di 10 volte rispetto il numero totale di documenti della prima e quarta era.

	Age 1	Age 2	Age 3	Age 4
Documents	115	440	1369	130

Per questo motivo, i test svolti sui modelli di machine learning sono stati effettuati prima con il dataset sbilanciato e poi anche con il dataset bilanciato attraverso le tecniche di *text augmentation* (vedi paragrafo 1.1.5).

In particolare, sono state applicate due tecniche diverse. La prima di queste consiste nel dividere il testo in frasi e creare nuove istanze mischiando in maniera randomica le frasi tra loro.

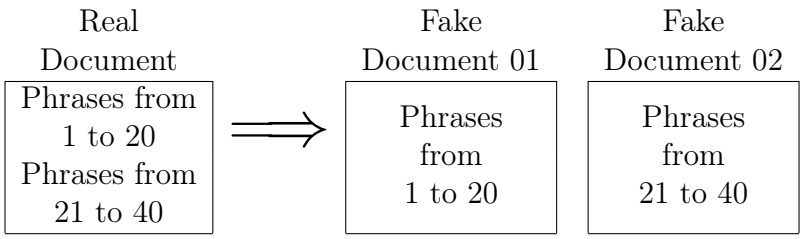



```python
def get_random_newdata(texts, numberToReach):
    """
    texts : lista documenti reali
    numberToReach : numero di documenti da raggiungere
    numberToReach = #documenti reali + #documenti fake
    """
    newtexts = []
    # a ogni iterazione prendiamo un indice randomico e creiamo
    # nuovi documenti aggiungendoli all'array "newtexts"
    while (len(newtexts)+len(texts)) < (numberToReach):
        i = random.randint(0, len(texts)-1)
        text = texts[i].split('.')
        random.shuffle(text)
        newtexts.append( " . ".join(text) )
    return newtexts
```

```python
newdata1 = get_random_newdata(df.loc[df.age == 1]['Content'], 1200)
newdata2 = get_random_newdata(df.loc[df.age == 2]['Content'], 1200)
newdata4 = get_random_newdata(df.loc[df.age == 4]['Content'], 1200)
```

Figura 3.11: Random Text Augmentation.

La seconda tecnica applicata consiste nel suddividere un paper in diverse parti, formate da un numero di frasi prestabilito: in questo caso, nel nuovo dataset bilanciato non avremo più i paper reali, ma solo quelli generati.



```python
def split_doc_content(doc, n=20):
    """
    doc : documento reale
    n : numero di frasi necessarie per creare un nuovo testo
    """
    tokenizer = nltk.data.load('tokenizers/punkt/english.pickle')
    sentences = tokenizer.tokenize(doc)
    new_docs = []
    for i in range(0, len(sentences), n):
        new_docs.append(" .".join(sentences[i: i+n]))
    return new_docs
```

Figura 3.12: Split Text Augmentation.

3.3 Word Embeddings e rappresentazione delle keyword delle classi con t-SNE

L'idea sviluppata in questa sezione consiste nell'addestrare un modello di Word Embeddings (WE) in modo da avere una rappresentazione vettoriale del dizionario di parole del corpus di documenti. In questo modo è possibile calcolare la similarità tra ogni coppia di termini.
Successivamente, possiamo eseguire una task supervisionata, che consiste nell'identificare delle *keywords* per ogni categoria, ottenere le parole più simili a tali termini tramite il modello di WE e avere una rappresentazione grafica che rispecchi la distribuzione delle parole più simili di ogni era storica.

Per applicare il modello di Word Embeddings è stato usata la libreria `gensim` di Python. Nello specifico la classe `Word2Vec`[5], della quale sono stati utilizzati i seguenti parametri:

- **vector_size** - Dimensionalità dei vettori rappresentativi delle parole: la dimensione ottimale dipende dal task da svolgere e dalle dimensioni del dizionario del corpus di documenti. Al riguardo è stata svolta un'analisi per individuare un valore minimo ottimale di questo parametro attraverso un metodo statistico[15].

- **window** - Stabilisce la massima distanza tra due parole all'interno di una frase o un testo per essere considerate correlate. Una finestra di grande dimensioni tende a rappresentare meglio le similarità tra i topic, mentre per bassi valori tende a produrre similarità più funzionali e sintattiche.

[5]https://radimrehurek.com/gensim/models/word2vec.html

- **sg** - Questo parametro può assumere due valori: 0 se si intende applicare l'algoritmo CBOW oppure 1 se invece si vuole applicare la tecnica skip-gram[6].
- **min_count** - Il modello ignora tutte le parole che hanno un'occorrenza inferiore a questo valore.
- **workers** - Specifica il numero di worker threads da utilizzare per l'addestramento del modello.

Infine, applicheremo la tecnica t-SNE (vedi paragrafo 1.5) per avere una rappresentazione grafica della disposizione delle parole simili alle keyword di ogni categoria in un piano cartesiano.

In questa situazione sono stati aggiunti alle lista di termini anche i bigram, dato che alcune parole chiave sono composte da due termini, come per "deep learning".

```
def add_bigram(words):
    words_with_bigram = []
    for i in range(len(words)-1):
        words_with_bigram.append(words[i])
        words_with_bigram.append(words[i] + ' ' + words[i+1])
    return words_with_bigram
```

Figura 3.13: Aggiunta dei bigram a una lista di termini.

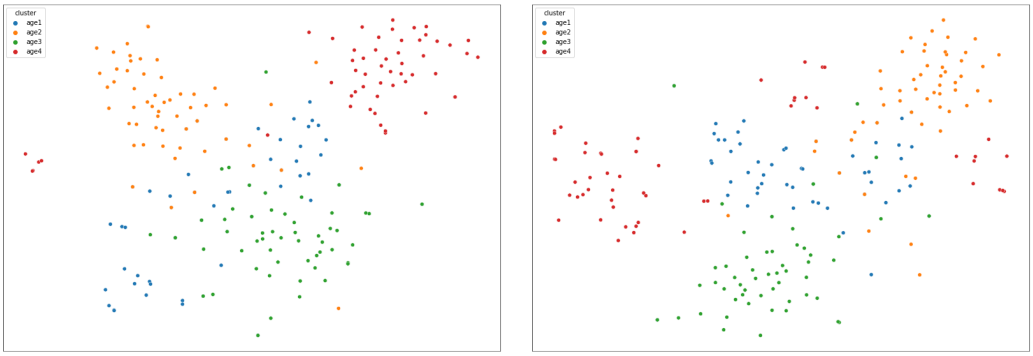
Le keywords sono state individuate dai riassunti delle fasi storiche proposte in questa analisi (vedi Capitolo 2).

	Age 1	Age 2	Age 3	Age 4
Keywords	game tree Shannon Turing hardware brute force type special hardware	evolution special hardware parallel massive Karsparov Deep Blue	software personal computer game tree	deep learning Alpha Zero Google Stockfish

Come si nota dai grafici nella Figura 3.14, le parole simili alle keyword individuate mostrano una più o meno netta distinzione tra le ere storiche: questo significa che le rappresentazioni vettoriali delle keyword sono differenti

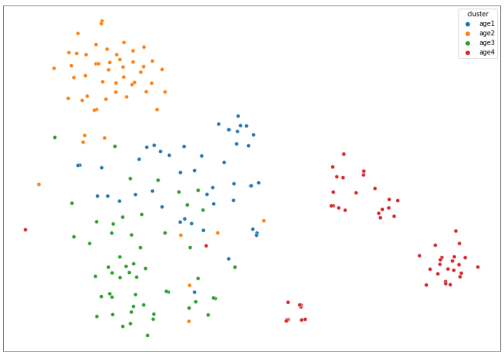
[6]Vedi paragrafo 1.1.4.

tra loro e, di conseguenza, anche dalle parole simili a quest'ultime.
Tuttavia, analizzando la distribuzione di queste parole all'interno dei documenti, si è notato per la maggiore che tali termini raffiguravano in maniera casuale tra le quattro ere storiche. Questo può voler dire che le parole, in particolare le keyword scelte, non sono utili per suddividere correttamente i paper.

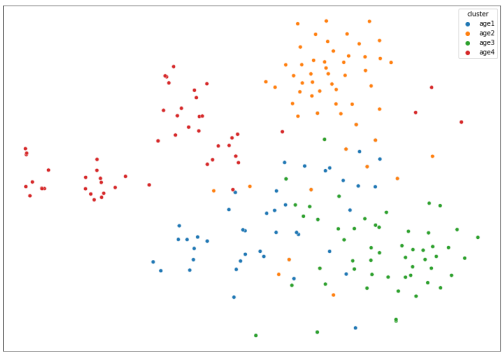


(a) `vector_size = 100` `window = 5`

(b) `vector_size = 100` `window = 10`



(c) `vector_size = 100` `window = 40`



(d) `vector_size = 400` `window = 10`

Figura 3.14: Grafici ottenuti con diversi valori di `vector_size` e `window`. Ogni colore rappresenta un'era storica: prima era (blu), seconda era (arancione), terza era (verde) e quarta era (rossa).

Dalla figura si può notare come all'aumentare del valore del parametro `window`, i termini tendano ad allontanarsi.

3.4 Modelli di classificazione non supervisionata

In questa sezione verranno utilizzati diversi modelli di classificazione non supervisionata, ognuno dei quali è stato addestrato più volte con dati differenti

per effettuare un'analisi più ampia sui risultati ottenibili.
Più precisamente, i modelli di clustering sono stati addestrati con le seguenti rappresentazioni dei documenti:

- tf-idf
- tf-idf + LSA
- Aggiunta di bigram alla BoW + tfidf + LSA
- tf-idf + LSA + clustering con le n parole di valore tf-idf maggiore
- Rappresentazione vettoriale delle parole e dei documenti tramite Word Embedding (Word2Vec)
- Rappresentazione vettoriale delle parole e dei documenti tramite Transformers (LongFormer e BigBird)

Inoltre, è stata applicata la tecnica LDA per ricercare un numero prestabilito di topic all'interno del corpus di documenti e per utilizzare quelli più rilevanti come cluster per classificare le istanze del dataset.

Alla classe `TfidfVectorizer` sono passati due parametri appositi per fare in modo che il modello accetti una lista di parole anziché un testo completo da tokenizzare. Questo risparmia l'esecuzione della tokenizzazione a ogni test eseguito: la lista di termini è stata calcolata una volta sola e memorizzata nel DataFrame (nello specifico nella colonna "bow").

Per la validazione dei modelli sono state calcolate le seguenti metriche:

- **Recall**: Percentuale di istanze di una classe classificate correttamente.
- **Precision**: Percentuale di istanze classificate correttamente di una classe in rapporto alla somma di tutte le istanze classificate di quella stessa classe.
- **F1-Measure**: Media pesata della precision e della recall che tende a dare maggior peso ai punteggi più bassi.

Le formule di queste metriche si basano sui valori ottenuti dalla matrice di confusione, ovvero dal numero di istanze classificate come *true positive* (TP), *false positive* (FP), *true negative* (TN) e *false negative* (FN).

$$recall = \frac{TP}{TP + FN}$$

$$precision = \frac{TP}{TP + FP}$$

$$f1\_measure = \frac{2 \cdot precision \cdot recall}{precision + recall}$$

3.4.1 Modelli di Clustering: K-Means, DBSCAN, HDBSCAN

Per applicare sequenzialmente le tecniche tfidf, LSA e i modelli di clustering è stata utilizzata la classe `Pipeline` della libreria `sklearn`: la quale permette di integrare più passaggi in un unico modello.

Per attribuire un cluster a un'era storica, ovvero a una classe del nostro dataset, sono stati applicati i seguenti metodi:

1. Ogni cluster è attribuito all'era storica più presente in valore assoluto in esso. Questo metodo tende ad attribuire la maggior parte dei cluster all'era storica con il numero maggiore di istanze, quindi in un dataset non bilanciato come il nostro può portare a risultati indesiderati. In ogni caso, questo metodo massimizza il valore della *precision*.

2. Ogni cluster è attribuito all'era storica in base al valore della *recall* maggiore di ognuna di esse. Questo metodo massimizza il valore della *recall*.

3. Ogni cluster è attribuito all'era storica la cui percentuale in esso contenuta si discosta maggiormente (in positivo) dalla percentuale delle stessa era in una distribuzione ottenuta da un modello randomico.

Quest'ultima ha portato a risultati più bilanciati e accurati nella maggior parte dei casi. Per il nostro caso di studio è stata utilizzata la funzione nella Figura 3.15.

```
def topic_age(cluster):
    # Calcolo classi predetta da modello random
    tot_rapp = [v/2054 for v in [115, 440, 1369, 130]]
    # Calcolo rapporto delle classi nel cluster
    obs_rapp = [v/sum(cluster) for v in cluster]
    # Calcolo differenza con modello random
    res = pd.Series(obs_rapp) - pd.Series(tot_rapp)
    res.index = [1, 2, 3, 4]
    # Restituisco il topic con la differenza positiva maggiore
    return res.idxmax()
```

Figura 3.15: Associazione di un cluster a una classe.

K-Means

I due parametri da indicare per questa pipeline sono il numero di cluster che il modello K-Means deve individuare e il numero di componenti della matrice LSA, ovvero lo spazio dimensionale nel quale trasformare i vettori dei documenti ottenuti da tfidf.

```
N_CLUSTERS=30
N_COMPONENTS = 20
model = Pipeline([
    ('vectorizer', TfidfVectorizer(tokenizer=lambda x: x, lowercase=False)),
    ('lsa', TruncatedSVD(n_components=N_COMPONENTS, random_state=0)),
    ('cluster', KMeans(n_clusters=N_CLUSTERS, random_state=0))
])
model.fit(df['bow'])
```

Figura 3.16: Esempio di Pipeline con LSA e KMeans.

```
pd.Series(model.named_steps['cluster'].labels_, index=df.index)[0:5]
```

```
articoli/articoli/05ChowErikssonFan.pdf    28
articoli/articoli/60Reider.pdf              2
articoli/articoli/95OnofrijCuratola.pdf    11
articoli/articoli/07Guid.pdf               29
articoli/articoli/92MazurBooth.pdf         11
dtype: int32
```

Figura 3.17: Cluster ottenuti dall'addestramento della Pipeline della Figura 3.16

```
pd.Series(model.predict(df.tfidf), index=df.index)[0:5]
```

```
articoli/articoli/05ChowErikssonFan.pdf     2
articoli/articoli/60Reider.pdf              2
articoli/articoli/95OnofrijCuratola.pdf     2
articoli/articoli/07Guid.pdf               29
articoli/articoli/92MazurBooth.pdf          2
dtype: int32
```

Figura 3.18: Cluster ottenuti dalla previsione dei 20 termini più rappresentativi in accordo con il loro valore tfidf.

Per impostare il numero ottimale di cluster viene spesso applicato il **metodo a gomito**[7] (*elbow method*): con il modello K-Means di `sklearn` possiamo memorizzare il valore dell'attributo `intertia_` per ogni diverso numero di cluster e, infine, graficare i dati ottenuti.

[7] Vedi paragrafo 1.4.1.

```python
def get_kmeans_inertia(n_clusters, x_train):
    model = Pipeline([
        ('tfidf', TfidfVectorizer(tokenizer=lambda x: x, lowercase=False)),
        ('lsa', TruncatedSVD(n_components=20, random_state=0)),
        ('cluster', KMeans(n_clusters=n_clusters, random_state=0))
    ])
    model.fit(x_train)
    return model.named_steps['cluster'].inertia_
```

```python
wcss = []
index = []
for n in range(2, 80, 1):
    wcss.append(get_kmeans_inertia(n, df.bow))
    index.append(n)

wcss = pd.Series(wcss, index=index)
```

Figura 3.19: Calcolo WCSS per un numero di cluster tra 2 e 80.

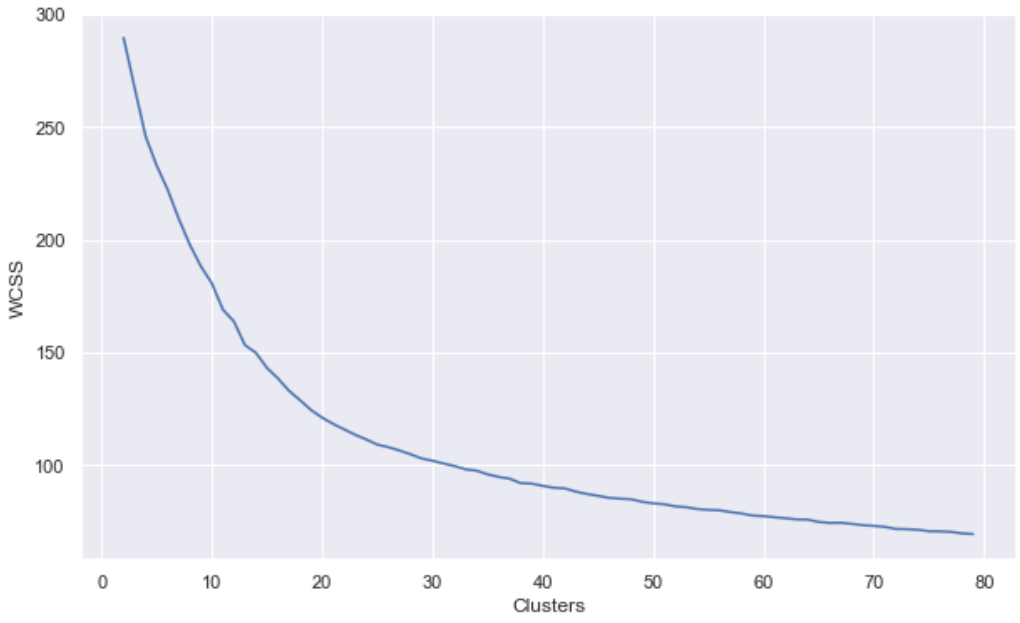


Figura 3.20: Grafico prodotto dalla Figura 3.19.

DBSCAN

Con il modello DBSCAN non possiamo scegliere il numero di cluster a priori, ma dobbiamo specificare alcuni parametri che saranno determinanti per

tale valore e il numero di istanze che vengono classificate come rumore.
I due parametri fondamentali[8] sono:

- `eps` (*distance threshold epsilon*) : il valore è inversamente proporzionale al numero di istanze riconosciute come rumore
- `min_samples` (*minimum cluster size*) : il valore è proporzionale al numero di istanze riconosciute come rumore

Inoltre, anche il numero di componenti della matrice LSA è risultato direttamente proporzionale al numero di rumore.

```
model = Pipeline([
    ('vectorizer', TfidfVectorizer(tokenizer=lambda x: x, lowercase=False)),
    ('lsa', TruncatedSVD(n_components=N_COMPONENTS, random_state=0)),
    ('cluster', DBSCAN(eps=0.2, min_samples=5, algorithm='auto'))
])
```

Figura 3.21: Esempio di Pipeline con LSA e DBSCAN.

DBSCAN inoltre fornisce la possibilità di impostare il parametro `metric`, che determina la formula che viene impiegata per calcolare la distanza tra le istanze. Di default viene utilizza la distanze euclidea, ma ponendo tale parametro uguale a `cosine`, verrà impiegata la similarità coseno, la quale è più opportuna quando si utilizzano dei dataset di documenti testuali.

Per visualizzare i cluster ottenuti può essere utilizzato lo stesso codice presente nella Figura 3.18. Tutte le istanze associate al valore -1 rappresentano quelle classificate come rumore: troppo distanti ai *cluster core* per essere attribuite a uno di essi.

HDBSCAN

Per utilizzare il modello HDBSCAN è necessario scaricare l'omonima libreria `hdbscan`, la quale è a sua volta implementata per funzionare anche in una pipeline della libreria `sklearn`.
Questa classe non ha bisogno dell'*epsilon* come DBSCAN, ma solo del numero minimo di istanze necessarie per formare un cluster (`min_cluster_size`). Inoltre, se si vuole usare il modello per predire i cluster di nuovi dati, allora bisogna anche specificare il parametro `prediction_data` uguale a `True`. Questo ci permette di svolgere dei test addestrando il modello su tutte le parole

[8] Per il significato dei parametri vedere il paragrafo 1.4.2.

del corpus di documenti e classificando le istanze tramite le sole n parole più rilevanti individuate con tf-idf.

```
from hdbscan import HDBSCAN

model = Pipeline([
    ('vectorizer', TfidfVectorizer(tokenizer=lambda x: x, lowercase=False)),
    ('lsa', TruncatedSVD(n_components=N_COMPONENTS, random_state=0)),
    ('cluster',
        HDBSCAN(
            min_cluster_size=4,
            prediction_data=True
        )
    )
])
```

Figura 3.22: Esempio di Pipeline con LSA e HDBSCAN.

Anche qui per visualizzare i cluster ottenuti può essere utilizzato lo stesso codice presente nella Figura 3.18.
Se si desidera invece addestrare un modello per predire dei nuovi dati bisogna svolgere ogni operazione, senza l'impiego di una Pipeline.

```
# Addestro modello tf-idf
tfidf = TfidfVectorizer(tokenizer=lambda x: x, lowercase=False)
vsm = tfidf.fit_transform(df.bow)
# Addestro modello LSA
lsa = TruncatedSVD(n_components=N_COMPONENTS, random_state=0)
lsa_data = lsa.fit_transform(vsm.todense())
# Converto la lista di termini di ogni documento
# con tf-idf e poi nello spazio LSA
docs_tfidf = tfidf.transform(df.tfidf)
docs_tfidf = lsa.transform(docs_tfidf)
```

```
results, _  = approximate_predict(model.named_steps['cluster'], docs_tfidf)
results = pd.Series(results, index=df.index)
```

Figura 3.23: Esempio di classificazione di nuovi dati con HDBSCAN.

3.4.2 LDA

Il modello LDA utilizzato è stato importato dalla libreria `gensim`, la quale offre due classi per questo scopo: `LdaModel` e `LdaMulticore`. Quest'ultima permette di utilizzare più core in parallelo in fase di addestramento del modello, abbattendo i tempi di esecuzione di circa il 60% con soli due core.
Per questo motivo, nella nostra analisi abbiamo utilizzato questa classe, la quale ha i seguenti parametri:

- `corpus`: Rappresentazione BoW dei documenti da analizzare.

- `id2word`: Dizionario dei termini di tutti i documenti ottenuto con la classe `Dictionary` sempre della libreria `gensim`.

- `num_topics`: Il numero di topic latenti da estrarre.

- `chunksize`: Numero di documenti utilizzati contemporaneamente in ogni *training chunk*, influenza la memoria necessari per l'addestramento.

- `passes`: Numero di iterazioni del corpus di documenti, equivalente delle *epoche* nelle reti neurali.

- `eval_every`: Calcola la *perplexity*[9] a ogni iterazione allungando in maniera sostanziale i tempi di addestramento.

- `iterations`: Il numero massimo di iterazioni per ogni documento, alcuni documenti potrebbero non convergere in un numero basso di iterazioni.

- `alpha`: Iperparametro che rappresenta la densità *document-topic*: con alti valori i documenti saranno formati da più topic, mentre con bassi alpha ne conterranno di meno.

- `beta`: Iperparametro che rappresenta la densità *topic-word*: è direttamente proporzionale al numero di parole che formano un topic.

Gli iperparametri `alpha` e `beta` sono i più complicati e dipendono dal tipo di distribuzione del modello LDA: simmetrica o asimmetrica. In quest'ultimo caso la loro definizione cambia leggermente, ma nella maggior parte dei casi avremo una distribuzione simmetrica. In generale: alti alfa producono documenti che contengono topic più simili e alti beta generano topic formati solo da parole molto simili tra loro.
Nella classe utilizzata il valore di `alpha` e `beta` di default è data dal valore del reciproco di `num_topics` e considerano una distribuzione simmetrica. Con queste impostazioni per il nostro caso di studio si sono ottenuti dei risultati leggermente migliori.

[9] La *perplexity* è una formula che indica quanto il modello è in grado di predire con successo dei dati che non ha mai visto.

```
from gensim.corpora import Dictionary
# Otteniamo il dizionario di termini dal corpus di documenti
dictionary = Dictionary(df['bow'])
# Filtriamo le parole che raffigurano in meno
# di 5 documenti (no_below) e che compaiono in più
# del 60% dell'intero corpus (no_above)
dictionary.filter_extremes(no_below=5, no_above=0.5)

# Creo la Bag-of-Words in base agli id presenti nel dizionario
corpus = [dictionary.doc2bow(doc) for doc in df['bow']]
```

```
from gensim.models import LdaModel, LdaMulticore

model = LdaMulticore(
    corpus=corpus,
    id2word=dictionary,
    chunksize=2054,
    iterations=400,
    num_topics=50,
    passes=40,
    eval_every=None,
    workers=4
)
```

Figura 3.24: Esempio di addestramento di un modello LDA.

Dopo avere addestrato il modello LDA possiamo ottenere una matrice documenti-topic dove in ogni cella è presente la similarità tra un documento e un topic: questo è facilitato dal metodo `get_document_topics` che fornisce il valore della similarità coseno di un documento con ognuno dei topic.

```
M_docs_topic = []
for i in range(len(df['bow'])):
    topics_sim = (
        model.get_document_topics(corpus[i],
                                  minimum_probability=0.0)
    )
    # Estraiamo il valore della similarità per ogni topic
    topics_vec = [topics_sim[i][1] for i in range(num_topics)]
    M_docs_topic.append(topics_vec)
```

Figura 3.25: Esempio di creazione matrice similarità documenti-topics.

Il passo successivo è associare a ogni documento il topic con la similarità maggiore, in modo da avere per ogni topic una collezione di istanze da cui ricavare il valore della classe di appartenenza. Possiamo trattare i topic allo stesso modo dei cluster ottenuti con i modelli precedenti e valutare se il contenuto dei topic è omogeneo oppure no con le dovute formule (recall, precision, f1-measure).

Per rendere il risultato ottenuto più affidabile è importante stabilire un valore minimo del contenuto dei topic, come è stato fatto con il parametro `min_cluster_size` nel modello HDBSCAN. Infatti, potrebbero essere individuati topic molto specifici a cui vengono associati solo 2 o 3 documenti, che in questo caso sarebbero da considerare del *rumore*. Per questo motivo, sono stati azzerati i topic contenti meno di 5 documenti.

Sono stati infine analizzati i topic rimasti per osservarne le dimensioni: il fine è di controllare se sono stati individuati pochi topic troppo grandi o tanti topic molto piccoli. I risultati migliori, sia per accuratezza che per questo ragionamento, sono stati ottenuti con un numero di `num_topics` compreso tra 80 e 120. Utilizzando questi valori abbiamo ottenuto in media tra i 10 e i 15 topic nulli e una buona distribuzione nei topic rimanenti.
Nell'ultimo paragrafo di questo capitolo analizzeremo meglio i risultati ottenuti e, attraverso test statistici, verificheremo che non siano stati frutto del caso.

3.4.3 Word Embeddings

Come mostrato nel paragrafo 3.3, possiamo addestrare un modello di Word Embeddings (WE) con la classe `Word2Vec` della libreria `Gensim` per ottenere una rappresentazione vettoriale di ogni termine appartenente al corpus di documenti.
In questo modo, possiamo trasformare la lista di parole appartenenti a un documento, in una lista di vettori, dove ogni parola è stata sostituita con il vettore ottenuto dal modello Word2Vec. Per ogni documento avremo quindi una matrice mxn dove m è il numero di termini nel documento e n è il numero di dimensioni di ognuno di essi (determinato dal modello WE). Infine per passare a una rappresentazione vettoriale è stato applicato un metodo chiamato *average pooling*: calcolando il valore medio di ogni colonna, in modo da avere per ogni documento un vecttore di n dimensioni.

Si ottiene quindi una matrice documenti-`vector_size`, dove quest'ultimo è il parametro passato al modello Word2Vec, utilizzabile per addestrare modelli di machine learning come quelli di classificazione non supervisionata appena visti.

```python
def doc_to_vect(words):
    # Ottengo lista dei termini conosciuti al modello
    w2v_vocab = model.wv.index_to_key
    # Converto i termini conosciuti in vettori
    m = [model.wv[word] for word in words if word in w2v_vocab]
    # Restituisco la media di ogni colonna della matrice
    return pd.DataFrame(m).mean(0)
```

Figura 3.26: Funzione per trasformare un documento in un vettore con un modello Word2Vect di Gensim.

3.4.4 Transformers

I modelli dei transformers utilizzati sono i due spiegati nel paragrafo **??**: **LongFormer** e **BigBird**. Questo ha permesso di utilizzare sequenze lunghe fino a 4096 token, proprietà molto utile per il nostro dataset dato che molti documenti eccedono i 512 token massimi in un modello classico come Bert. Tuttavia, alcuni documenti, dopo la tokenizzazione hanno comunque prodotto sequenze più lunghe di 4096, obbligandoci a implementare una funzione apposita che riduca il numero di token.
Il modo più semplice è quello di utilizzare il parametro `truncation` del tokenizer e porlo uguale a `True`: eliminando tutti i token successivi a quello in posizione 4096 (per essere precisi il token alla posizione 4096 viene sostituito con il *separation token*). Questa soluzione può potenzialmente produrre una perdita di informazioni rilevante, per questo si è deciso di selezionare e unire i primi 2000 token della sequenza con gli ultimi 2096: in questo modo si sono preservate parzialmente le informazioni presenti all'inizio del paper (titolo, autore, abstract) e alla fine di esso (riassunti, conclusioni).
Infine, dopo aver eseguito l'embedding della sequenza di token con il modello del transformer e ottenuta la matrice rappresentativa del documento, per ricavare un solo vettore da associare al documento si sono applicate diverse tecniche alternative tra loro:

- *average pooling*: Stesso procedimento svolto nel paragrafo precedente per i modelli Word Embeddings.
- *CLS token*: Prendere dal `last_hidden_state` il primo token di ogni sequenza, ovvero il classification token (CLS), il quale è rappresentativo della sequenza.
- *pooler output*: Richiamare l'attributo `pooler_output` della variabile `embedding` ottenuta dal modello (Vedi Figura), il quale restituisce un vettore ottenuto dal classification token dopo essere stato ulteriormente processato.

Per utilizzare i transformers in maniera efficienza è fortemente consigliato eseguirli su un dispositivo dotato di una GPU di dimensioni superiori ai 10 GB. Nel nostro caso di studio, il tempo necessario per eseguire il codice su GPU è stato del 97% inferiore rispetto alla stessa esecuzione su una CPU.
Per questo motivo è stata utilizzata la GPU fornita gratuitamente da Google sulla piattaforma Colab online.

Per testare il tipo di runtime in esecuzione può essere utilizzato il codice in Figura 3.27. Mentre per rientrare nei limiti di memoria della GPU può essere utile svuotare la cache della stessa per guadagnare dello spazio (Figura 3.30).

```
import torch
device = "cuda:0" if torch.cuda.is_available() else "cpu"
device
```

```
'cuda:0'
```

Figura 3.27: Assegna alla variabile `device` il tipo di GPU disponibile se presente, altrimenti la stringa "cpu".

```
import gc

def empty_cache_gpu():
  gc.collect()
  torch.cuda.empty_cache()
```

Figura 3.28: Funzione per svuotare la cache della GPU.

```
from transformers import LongformerModel, LongformerTokenizer
model = LongformerModel.from_pretrained('allenai/longformer-base-4096')
tokenizer = LongformerTokenizer.from_pretrained("allenai/longformer-base-4096")
```

Figura 3.29: Inizializzazione Longformer.

```
from transformers import BigBirdModel, BigBirdTokenizer
model = BigBirdModel.from_pretrained('google/bigbird-roberta-base')
tokenizer = BigBirdTokenizer.from_pretrained('google/bigbird-roberta-base')
```

Figura 3.30: Inizializzazione BigBird.

Dopo avere inizializzato uno dei due modelli di transformer, nella Figura 3.31 sono stati implementati i seguenti quattro passaggi:

1. Tokenizzazione del testo con il tokenizer specifico del transformer, nel nostro caso applichiamo automaticamente anche il padding e ritorniamo un tensore di `PyTorch`.

2. Ridurre le sequenze con più di 4096 token prendendo i primi 2000 e gli ultimi 2096.

3. Applicare l'embedding delle sequenze di token al transformer.

4. Ottenere un vettore del tipo `ndarray` (Numpy) dal risultato tramite la sequenza di CLS o il `pooler_output`.

```
def doc_to_vect(text, tokenizer, model):
  # Tokenizzazione del testo
  input_ids = tokenizer(text, padding="max_length", return_tensors='pt')
  input_ids = input_ids.input_ids
  # Controllo per sequenze di token lunghe più di 4096
  if len(input_ids[0])>4096:
      input_ids = input_ids[0]
      input_ids = np.concatenate((input_ids[0:2000], input_ids[-2096:]))
      input_ids = torch.tensor(input_ids)
      input_ids = torch.tensor([[v for v in input_ids]])
  # Embedding dei token con il transformer
  embedding = model(input_ids.to(device))
  return embedding
```

```
# CLS
embedding.last_hidden_state[:,0,:].cpu().detach().numpy()[0]
# POOLER_OUTPUT
embedding.pooler_output[0].detach().numpy()
```

Figura 3.31: Embedding di un testo con un transformer.

Dopo avere eseguito questa funzione per ogni documento, otterremo una matrice mxn dove m è il numero di documenti e n è pari a 768, ovvero il numero di dimensioni ottenute dall'embedding con il transformer. Potremo infine utilizzare questa matrice per addestrare i modelli di clustering.

Transformer Fine-Tuning

Per ottenere dei risultati ottimali con i transformers è necessario addestrare il modello sul corpus di documenti, in modo che possa individuare le correlazioni come la polisemia di un termine.
Questo procedimento, chiamato **fine-tuning**, è spesso complesso e richiede un utilizzo ingente di GPU. Nello specifico, per il nostro caso di studio, per effettuare il fine-tuning sul transformer BigBird sono stati necessari 18GB di GPU. Essendo il nostro dataset di sole 2mila istanze, i tempi per l'addestramente sono stati accettabili (circa 15 minuti).

Per implementare questo procedimento in maniera non supervisionata, è stata impiegata la tecnica del *random masking*[16]: in poche parole, il transformer viene addestrato cercando di indovinare le parole in una frase a lui nascoste ("mascherate") in base al contesto.

Il codice per eseguire quanto detto è riportato nelle figure 3.32 e 3.33, per una spiegazione più dettagliata delle classi e dei parametri è possibile trovare tutto nella documentazione online prodotta dagli sviluppatori della libreria `transformers`: il sito web si trova all'indirizzo `https://huggingface.co/`.

```
# Importo librerie e scarico modelli necessari
from transformers import AutoTokenizer
from transformers import AutoModelForCausalLM
from transformers import DataCollatorForLanguageModeling
from transformers import TrainingArguments
from transformers import Trainer
import torch

model_checkpoint = "google/bigbird-roberta-base"

tokenizer = AutoTokenizer.from_pretrained(model_checkpoint)
model = AutoModelForCausalLM.from_pretrained(model_checkpoint)

# Utilizziamo la GPU
model.to('cuda:0')

# Tokenizzazione intero dataset
tokenized_datasets = [ tokenize_function(" ".join(bow))
                       for bow in df.bow ]
```

Figura 3.32: Fase 1 del procedimento di fine-tuning.

```
# Inizializzo data collector: crea le batch ed effettua
# il random masking
data_collator = DataCollatorForLanguageModeling(tokenizer=tokenizer)

# Inizializzo i parametri per il training
training_args = TrainingArguments(
    "test-fine-tuning",
    learning_rate=2e-5,
    weight_decay=0.01,
    per_device_train_batch_size=1,
    per_device_eval_batch_size=1
)

# Inizializzo Trainer utilizzando 3/4 del dataset per il training
trainer = Trainer(
    model=model,
    args=training_args,
    train_dataset=tokenized_datasets[0:1500],
    eval_dataset=tokenized_datasets[1500:],
    data_collator=data_collator
)

# Effettuo il training e salvo il modello
trainer.train()
trainer.save_model('model-fine-tuned')
```

Figura 3.33: Fase 2 del procedimento di fine-tuning.

3.4.5 Confronto con modello supervisionato

Abbiamo effettuato dei test anche con modelli supervisionati per confrontare i risultati con quelli dei modelli visti fino a questo punto. L'obiettivo è controllare se vi è una netta differenza nella classificazione dei paper del nostro caso di studio tra i due diversi modelli: quanto, in questo caso, avere dei dati etichettati possa portare benefici.

A questo scopo sono stati impiegati dei modelli di classificazione lineare e non lineare, nello specifico:

- **Regressione Logistica** [17]. Modello di classificazione lineare, restituisce la probabilità che ogni istanza appartenga a una delle classi.
- **XGBoost** [18]. Modello di classificazione non lineare che combina alberi creati in sequenza e addestrati in base agli errori residui degli alberi precedenti.

Sono state impiegate la classe `LogisticRegression` della libreria `sklearn` e la classe `XGBoostClassifier` della libreria `xgboost`.

```
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LogisticRegression
import xgboost as xgb

# Divido i dati in train set e validation set
X_train, X_val, y_train, y_val = train_test_split(X, y, test_size=1/5,
                                                  random_state=0, shuffle=True)

# Regressione Logistica
model = LogisticRegression(solver='saga', random_state=42,
                           C=0.1, multi_class='multinomial')

# XGBoost
model = xgb.XGBClassifier(max_depth=100,
                          random_state=42,
                          n_jobs=-1)

# Creo la pipeline con il modello
pipe = Pipeline([
    ('tfidf', TfidfVectorizer(tokenizer=lambda x: x, lowercase=False)),
    ('lsa', TruncatedSVD(n_components=12, random_state=0)),
    ('model', model)
])
pipe.fit(X_train, y_train)
y_pred = pipe.predict(X_val)
```

Figura 3.34: Esempio di utilizzo di un modello di Regressione Logistica e un XGBoost Classifier in una Pipeline.

I risultati ottenuti sono in linea con quelli dei modelli non supervisionati: lo sbilanciamento del dataset ha prodotto uno sbilanciamento delle previsioni a favore dell'era più rappresentata, anche se applicate tecniche di bilanciamento[10], come la *text augmentation* o l'utilizzo del parametro `class_weight` [11].

3.5 Risultati ottenuti

I modelli di classificazione non supervisionata implementati in questo capitolo sono stati addestrati e testati con diverse versioni del nostro dataset, che sono state ottenute con diverse tecniche di preprocessing. Riassumendo, sono stati prodotti i seguenti *training set*:

- Bag-of-Words prodotta dal preprocessing come descritto nel paragrafo 1.1.1.

[10] È fondamentale ricordare che l'addestramento può essere effettuato sul dataset bilanciato, ma la validazione deve essere svolta unicamente con il dataset originale non bilanciato, altrimenti il modello potrebbe restituire buoni risultati solo perché predice correttamente le istanze artificiali, che non sono di nostro interesse.

[11] Il parametro `class_weight` permette di dare un peso maggiore alle istanze delle classi meno rappresentate, in modo da bilanciare la fase di addestramento.

- Bag-of-Words contenente solo gli n termini più rilevanti ottenuti con il metodo tf-idf, dove n è stato testato con più valori (10, 20, 40...).
- Bag-of-Words ottenuta dal dataset dopo aver effettuato *text augmentation*, come descritto nel paragrafo 1.1.5.

Alle Bag-of-Words è stata applicata spesso la tecnica LSA per la riduzione della dimensionalità, la quale ha in media fornito risultati migliori rispetto al suo non utilizzo.

Il modello **K-Means** è stato fortemente influenzato dallo sbilanciamento del dataset e non è stato in grado di individuare dei cluster omogenei: la maggioranza di questi avevano al proprio interno una distribuzione delle istanze paragonabile a quella randomica.

I modelli di clusterizzazione *density-based*, ovvero **DBSCAN** e **HDBSCAN**, non sono risultati idonei per il nostro caso di studio: i risultati ottenuti possono essere riassunti come un *trade-off* tra il numero di istanze ritenute rumore e il numero di istanze appartenenti a un cluster di grandi dimensioni. Per esempio, in DBSCAN abbassando il parametro *epsilon*, aumentava il numero di istanze classificate come rumore, mentre alzando tale parametro le istanze finivano per essere raggruppate tutte in un unico grande cluster.

I risultati ottenuti con questi modelli di clustering confermano le "campanelle dall'allarme" del *test statistico chiquadro* (vedi paragrafo 2.1) e il problema relativo allo sbilanciamento del dataset: essendo la prima e quarta classe circa 12 volte più piccole della terza classe, i modelli sono sempre stati non bilanciati anche nei risultati.

Per questo motivo, anche la rappresentazione vettoriale dei documenti ottenuta tramite l'*embedding* dei transformer **LongFormer** e **BigBird**, non ha portato a risultati migliori con gli algoritmi di clustering. Ancora una volta, viene confermata l'ipotesi fornita dal test chiquadro che sosteneva che i paper non fossero rappresentativi delle ere storiche di appartenenza.
Il fine-tuning dei modelli si è rilevato in questa circostanza inefficace, probabilmente perché per modelli complessi come il BigBird, vi era bisogno di un numero di istanze nettamente maggiore (sono state impiegate solo 1500 istanze per il training).

Il modello **LDA** ha fornito i risultati migliori ricercando un numero di topic compreso tra 80 e 120: i cluster inviduati con questa tecnica sono risultati più omogenei. Inoltre, l'attribuzione di un topic in base alla classe in rapporto

più rappresentata rispetto allo stesso caso in un modello randomico ha portato a risultati migliori (vedi Figura 3.15).

	LDA topics = 80		LDA topics = 100		LDA topics = 120	
	f1-measure	success rate	f1-measure	success rate	f1-measure	success rate
Age 1	26.3	66	25	26	27.3	44.3
Age 2	46.5	52	48.8	60	50.6	58
Age 3	62.5	51.1	68	55.5	68.2	52
Age 4	27.3	51.5	29.7	47	26.1	42.3

È importante capire se i risultati ottenuti sono frutto del caso oppure sono prodotti realmente dal lavoro di preprocessing e dai modelli costruiti e impiegati nell'analisi del nostro caso di studio. Per ottenere questa risposta dobbiamo confrontare i nostri modelli con un modello randomico.
Dato che abbiamo effettuato una classificazione non supervisionata, non possiamo fornire al modello randomico il peso delle classi del dataset. Dunque la sua previsione sarà equibilanciata: ogni classe avrà il 25% di possibilità di essere predetta per ogni istanza.
Riassumendo, confronteremo l'accuratezza e l'*f1-measure* tra i due modelli e useremo questi valori per svolgere i seguenti test:

- **K-Statistic**. Misura il guadagna relativo di un modello rispetto a quello randomico.
- **Test d'Ipotesi** e **Test t-Student**. Sono due test statistici impiegati, in questo studio, per valutare l'indipendenza tra un modello e quello casuale, ovvero se i risultati ottenuti fossero solo frutto del caso.

Nella prossima tabella sono riportati i risultati ottenuti con la formula della *K-Statistics*. Come si può notare, sono in generale positivi: nella prima colonna si è ottenuto un punteggio superiore al 35% per ogni era.

	LDA topics = 80	LDA topics = 100	LDA topics = 120
Age 1	56.1	4.5	28
Age 2	37.8	48	45.4
Age 3	36.4	42.2	37.5
Age 4	36.3	30.3	24.2

Utilizziamo ora i risultati ottenuti con un numero di topics pari a 80 per effettuare il test d'ipotesi e il t-Student con una confidenza scelta del 90%.

Test t-Student

con	f1-measure	success rate
p-value	0.07	0.001
stats score	2.137	8.822

Test d'Ipotesi

intervallo con f1-measure	intervallo con success rate
0.2 - 0.25	0.29 - 0.35

Come si può osservare dai valori prodotti da entrambi i test, la differenza tra i nostri risultati e quelli ottenuti con classificazione casuale è statisticamente significativa con confidenza del 90%.

Capitolo 4

Conclusioni e sviluppi futuri

Questo studio ci porta alla conclusione che un dataset con le caratteristiche come quello a disposizione non è idoneo alla classificazione: il rapporto tra il numero di istanze di due classi era superiore all'ordine di grandezza e non si è riusciti a superare questo ostacolo.

I risultati ottenuti nell'analisi svolta in questa tesi sono stati coerenti tra loro: lo studio iniziale svolto sull'analisi dello sbilanciamento del dataset e con il test statistico chiquadro per verificare la correlazione o indipendenza tra i paper e la rispettiva era storica di appartenenza, non è stato smentito dai risultati ottenuti dai modelli di classificazione.

Tuttavia, il modello LDA ha fornito dei risultati accettabili e, soprattutto, abbiamo ottenuto un guadagno non irrilevante rispetto al modello randomico, a dimostrazione del fatto che il lavoro svolto per processare i dati testuali e progettare il modello LDA, ha portato all'individuazione di alcune correlazioni tra i documenti tramite l'identificazione dei topic trattati nei vari articoli. Inoltre, i test d'ipotesi svolti hanno dimostrato che tali risultati non sono stati frutto del caso.

Un possibile sviluppo futuro potrebbe essere il miglioramento del dataset tramite l'inserimento di nuove istanze appartenenti alle classi meno rappresentate. Questo passaggio può risultare molto complicato e lungo (se non impossibile, data la difficoltà di trovare articoli appartenenti agli anni 60'). Per questo motivo, si potrebbero analizzare tecniche di *text augmentation*, ovvero di bilanciamento di un dataset testuale, non affrontate in questa sede per motivi di tempo.

Un'altra strada potenziale potrebbe integrare un utilizzo più intenso dei modelli OCR: per individuare parti specifiche degli articoli (titolo, abstract, citazioni...) e ripetere i test basandosi solamente su queste informazioni. Quindi, un'analisi più incentrata sulla struttura degli articoli permetterebbe in modo più preciso di rimuovere parti poco rilevanti (come i *related works*) ed estrarre porzioni di testo o informazioni più rappresentative dell'articolo.

Anche altri approcci semi-supervisionati, come quello mostrato al paragrafo 3.3, potrebbero essere resi più efficaci da una migliore individuazione di keyword rappresentative delle classi o da delle loro descrizioni riassuntive più ricche di informazioni. In questa sede, ci si è voluti concentrare sui modelli non supervisionati e tale parte di lavoro è stata per cui limitata.

Infine, uno studio più incentrato sui transformers e sul *fine-tuning* di quest'ultimi, in aggiunta a un bilanciamento migliore del dataset, potrebbe portare a sviluppi interessanti data la complessità di questi modelli e i progressi che hanno e stanno portando nel mondo del NLP e del machine learning.

Ringraziamenti

I miei ringraziamenti vanno soprattutto al Prof. Angelo Di Iorio e al Prof. Gianluca Moro, i quali con la loro esperienza mi hanno fornito indicazioni e consigli fondamentali per il completamento di questa tesi.

Ringrazio anche il Prof. Paolo Ciancarini che ha reso disponibile la sua preziosa collezione di articoli.

Grazie anche ai miei genitori per tutto il sostegno su cui ho potuto contare in questi anni e per avermi dato la possibilità di studiare a Bologna.

Bibliografia

[1] Giacomo Domeniconi, Gianluca Moro, Roberto Pasolini, and Claudio Sartori. A study on term weighting for text categorization: A novel supervised variant of tf.idf. 07 2015. https://www.semanticscholar.org/paper/A-Study-on-Term-Weighting-for-Text-Categorization%3A-Domeniconi-Moro/830daecade28db50f114b84733132a5d6a7c60d0.

[2] Peter W. Foltz. Latent semantic analysis for text-based research. 28(2):197–202, 1996. Behavior Research Methods, Instruments, Computers. Volume 28.

[3] Jeffrey Pennington, Richard Socher, and Christopher D. Manning. Glove: Global vectors for word representation. In *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing, EMNLP 2014, October 25-29, 2014, Doha, Qatar, A meeting of SIGDAT, a Special Interest Group of the ACL*, pages 1532–1543, 2014.

[4] Tomas Mikolov, Wen-tau Yih, and Geoffrey Zweig. Linguistic regularities in continuous space word representations. In *Proceedings of the 2013 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, pages 746–751, Atlanta, Georgia, June 2013. Association for Computational Linguistics. https://aclanthology.org/N13-1090.

[5] David M. Blei, Andrew Y. Ng, and Michael I. Jordan. Latent dirichlet allocation. *J. Mach. Learn. Res.*, 3:993–1022, March 2003. Publisher: JMLR.org.

[6] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention is all you need, 2017. https://arxiv.org/abs/1706.03762.

[7] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding, 2018. https://arxiv.org/abs/1810.04805.

[8] Yinhan Liu, Myle Ott, Naman Goyal, Jingfei Du, Mandar Joshi, Danqi Chen, Omer Levy, Mike Lewis, Luke Zettlemoyer, and Veselin Stoyanov. Roberta: A robustly optimized bert pretraining approach, 2019. https://arxiv.org/abs/1907.11692.

[9] Iz Beltagy, Matthew E. Peters, and Arman Cohan. Longformer: The long-document transformer, 2020. https://arxiv.org/abs/2004.05150.

[10] Manzil Zaheer, Guru Guruganesh, Avinava Dubey, Joshua Ainslie, Chris Alberti, Santiago Ontanon, Philip Pham, Anirudh Ravula, Qifan Wang, Li Yang, and Amr Ahmed. Big bird: Transformers for longer sequences, 2020. https://arxiv.org/abs/2007.14062.

[11] Yi Tay, Mostafa Dehghani, Samira Abnar, Yikang Shen, Dara Bahri, Philip Pham, Jinfeng Rao, Liu Yang, Sebastian Ruder, and Donald Metzler. Long range arena: A benchmark for efficient transformers, 2020. https://arxiv.org/abs/2011.04006.

[12] Stuart Lloyd. Ieee transactions on information theory. pages 129–137, 1982. Publisher: IEEE Information Theory Society.

[13] Ricardo Campello, Davoud Moulavi, and Joerg Sander. Density-based clustering based on hierarchical density estimates. volume 7819, pages 160–172, 04 2013. https://link.springer.com/chapter/10.1007/978-3-642-37456-2_14.

[14] Laurens van der Maaten and Geoffrey Hinton. Viualizing data using t-sne. 9:2579–2605, 11 2008. Journal of Machine Learning Research, Volume 9.

[15] Kevin Patel and Pushpak Bhattacharyya. Towards lower bounds on number of dimensions for word embeddings. In *Proceedings of the Eighth International Joint Conference on Natural Language Processing (Volume 2: Short Papers)*, pages 31–36, Taipei, Taiwan, November 2017. Asian Federation of Natural Language Processing. https://aclanthology.org/I17-2006.

[16] Mengjie Zhao, Tao Lin, Fei Mi, Martin Jaggi, and Hinrich Schütze. Masking as an efficient alternative to finetuning for pretrained language models, 2020. https://arxiv.org/abs/2004.12406.

[17] J. S. Cramer. The origins of logistic regression. 2002. https://papers.tinbergen.nl/02119.pdf.

[18] Tianqi Chen and Carlos Guestrin. Xgboost. *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, Aug 2016. http://dx.doi.org/10.1145/2939672.2939785.