

Generare il testo perfetto

All'interno del Notebook progetto ci occupiamo di analizzare la **text generation** coinvolgendo l'utilizzo di tecniche e algoritmi dediti a questo scopo. La prima fase di questo processo prevederà l'analisi del dataset scelto.

Membri del gruppo:

- **Corbo Erika** (5086072)
- **Luongo Giovanni** (5038815)

```
In [4]: # Stopwords da ignorare per il preprocessing dei testi
sw = ["lj", 'bop', 'jack', 'boogety', 'hmm', 'yay', 'aah', 'woo', 'nana', 'ohh', 'doo']
```

```
In [5]: # Librerie utilizzate
import io
import re
import string
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
plt.style.use('seaborn-v0_8') # Per rendere i plot più belli
import seaborn as sns
sns.set(context='talk', style='ticks')
import random
import pickle
import tensorflow
import keras
from collections import Counter

# Librerie necessarie da sklearn
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.cluster import KMeans
from sklearn.decomposition import PCA
from sklearn.feature_extraction.text import CountVectorizer
from sklearn.decomposition import LatentDirichletAllocation

# Natural Language toolkit
import nltk
from nltk.corpus import stopwords
from nltk.tokenize import word_tokenize
from nltk.tokenize import RegexpTokenizer
from nltk import FreqDist
from nltk import ngrams

# Word2vec
from gensim.models import Word2Vec
%matplotlib inline
```

```
In [ ]: nltk.download('punkt')
nltk.download('stopwords')
nltk.download('averaged_perceptron_tagger')
nltk.download('stopwords')
```

1. Analisi dei dati

```
In [47]: filename = "billboard_top_100.csv"
df = pd.read_csv(filename)
# Numero di canzoni nel dataset
print("Numero di canzoni: ", df.shape)
```

Numero di canzoni: (6879, 5)

```
In [48]: # Per avere un df contenente solo canzoni degli anni [2010-2020]:
df = df.loc[(df['Hot100 Ranking Year'] >= 2010) & (df['Hot100 Ranking Year'] < 2020)]
print("Numero di canzoni: ", df.shape)
```

Numero di canzoni: (995, 5)

1.1. Pulizia dei dati

Prima di cominciare con le nostre analisi, dobbiamo rendere i testi delle canzoni manipolabili. Il dataset contiene i testi delle canzoni in un formato poco comodo per l'analisi.

```
In [49]: df.head()
```

```
Out[49]:
```

	Song	Artist Names	Hot100 Ranking Year	Hot100 Rank	Lyrics
5584	TiK ToK	['kesha']	2010	1	['wake', 'up', 'in', 'the', 'mornin', 'feelin'...
5585	Need You Now	['lady a']	2010	2	['picture', 'perfect', 'memories', 'scattered'...
5586	Hey, Soul Sister	['train']	2010	3	['hey', 'hey', 'hey', 'your', 'lipstick', 'sta...
5587	California Gurls	['katy perry', 'snoop dogg']	2010	4	['greetings', 'loved', 'ones', 'lets', 'take',...
5588	OMG (feat. will.i.am)	['usher', 'will i am']	2010	5	['oh', 'my', 'gosh', 'baby', 'let', 'me', 'I',...

Come prima cosa, modifichiamo il formato dei testi per renderli liste di parole. In seguito ci occuperemo di trasformarli in stringhe continue, ma all'inizio ci è più comodo tenerli come liste.

```
In [50]: tokenizer = RegexpTokenizer(r'\w+')
df['Lyrics'] = df['Lyrics'].apply(lambda x: tokenizer.tokenize(x.lower())) if np.all
df.head()
```

Out[50]:

	Song	Artist Names	Hot100 Ranking Year	Hot100 Rank	Lyrics
5584	TiK ToK	['kesha']	2010	1	[wake, up, in, the, mornin, feelin, like, p, d...
5585	Need You Now	['lady a']	2010	2	[picture, perfect, memories, scattered, all, a...
5586	Hey, Soul Sister	['train']	2010	3	[hey, hey, hey, your, lipstick, stains, on, th...
5587	California Gurls	['katy perry', 'snoop dogg']	2010	4	[greetings, loved, ones, lets, take, a, journe...
5588	OMG (feat. will.i.am)	['usher', 'will i am']	2010	5	[oh, my, gosh, baby, let, me, l, i, did, it, a...

```
In [51]: # Output dei testi senza ulteriori informazioni su un file .txt
joined_lyrics = df['Lyrics'].apply(lambda x: ' '.join(map(str, x))).tolist()

# File txt contenente solo le lyrics senza altri metadati
file = open('lyrics.txt', 'w')
for i in range(len(joined_lyrics)):
    file.write(joined_lyrics[i]+"\n\n")

file.close()
lyrics = pd.DataFrame({'Lyrics': io.open('lyrics.txt', 'r', encoding='ascii', error
```

In [52]: lyrics.head()

Out[52]:

Lyrics

```
0  wake up in the mornin feelin like p diddy hey ...
1  picture perfect memories scattered all around ...
2  hey hey hey your lipstick stains on the front ...
3  greetings loved ones lets take a journey i kno...
4  oh my gosh baby let me l i did it again so ima...
```

1.2. Numero di parole

Possiamo adesso facilmente contare il numero di parole per ogni canzone e alla fine sommarle per ottenere il numero complessivo di parole. Per prima cosa, aggiungiamo delle colonne che ci serviranno da contatori.

```
In [53]: lyrics['Number of characters'] = lyrics.Lyrics.str.len()
lyrics['Number of words'] = lyrics.Lyrics.str.split().str.len()
lyrics['Number of unique words'] = lyrics.Lyrics.apply(lambda x: len(set(x.split()))
```

In [54]: lyrics.head()

Out[54]:

	Lyrics	Number of characters	Number of words	Number of unique words
0	wake up in the mornin feelin like p diddy hey ...	2176	456	146
1	picture perfect memories scattered all around ...	979	217	78
2	hey hey hey your lipstick stains on the front ...	1371	288	112
3	greetings loved ones lets take a journey i kno...	2119	409	168
4	oh my gosh baby let me l i did it again so ima...	2125	526	112

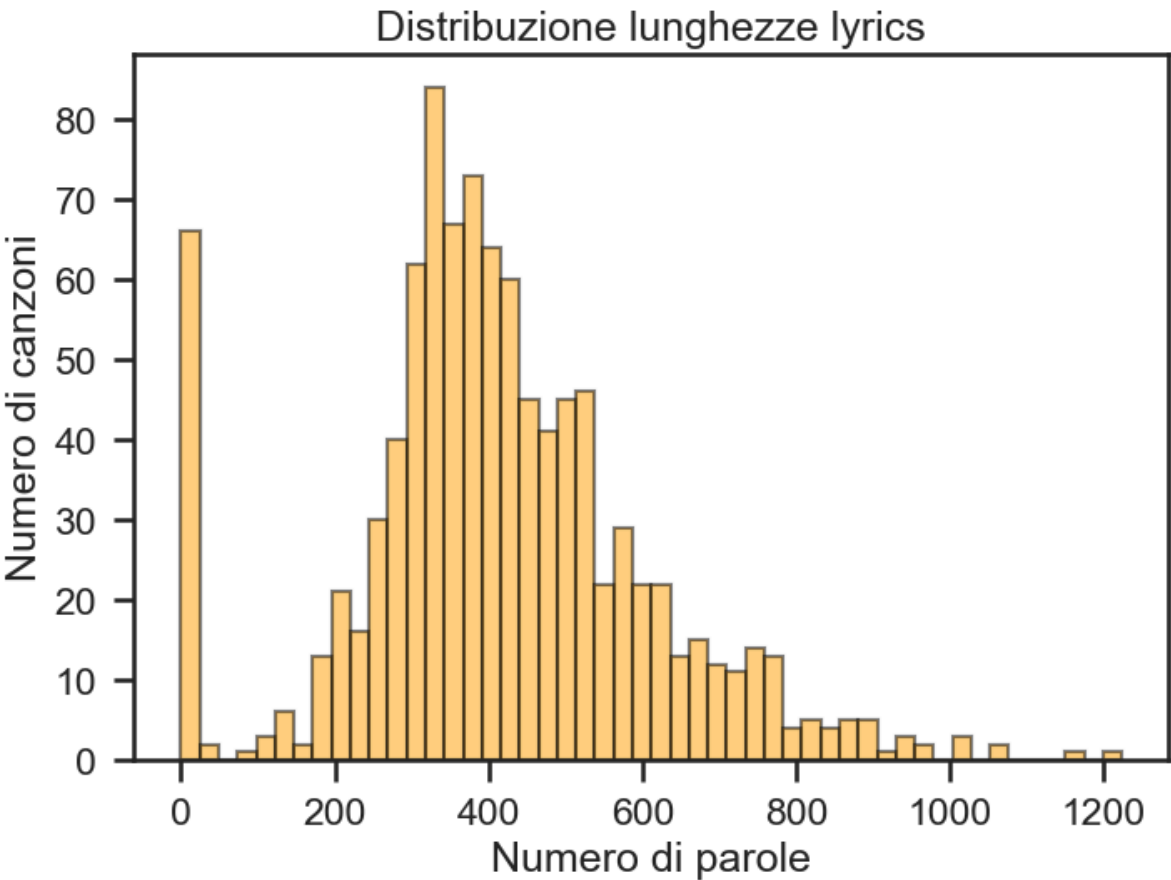
1.2.1. Numero medio di parole

Mostriamo la distribuzione delle lunghezze dei testi.

```
In [55]: plt.hist(lyrics['Number of words'], bins=50, color="orange", edgecolor='black', alp
plt.title('Distribuzione lunghezze lyrics')
plt.xlabel('Numero di parole')
plt.ylabel('Numero di canzoni')

print("Valore medio: ", int(lyrics['Number of words'].mean()))
```

Valore medio: 412



Possiamo notare che ci sono molte canzoni con 0 parole. Queste canzoni spesso sono instrumental, altre volte sono errori nel dataset. In generale, sono in numero minore rispetto alle canzoni che dispongono di testo e non influenzano i risultati a cui vogliamo arrivare.

```
In [56]: print("Numero di canzoni senza testo: ", (lyrics['Number of words'] == 0).sum())
```

Numero di canzoni senza testo: 65

1.3. Analisi delle parole

Passiamo ad analizzare nel concreto le parole presenti nel dataset.

```
In [57]: all_words = [word for word in df['Lyrics'].explode().tolist() if not isinstance(word, str)]
print("Numero di parole valide nel dataset: ", len(all_words))
```

Numero di parole valide nel dataset: 411121

Se vogliamo analizzare più a fondo i testi delle canzoni anche dal punto di vista della semantica, dobbiamo eseguire ulteriore pulizia dei dati.

1.3.1. Parole più frequenti senza preprocessing

Se analizzassimo i testi delle canzoni così come sono, le parole più frequenti che troveremmo sarebbero **stopword**. Le stopwords sono parole comuni nei testi che non hanno significato semantico. Sono articoli, preposizioni, congiunzioni. Sono presenti in qualsiasi tipo di testo e non ci sono utili per le nostre analisi.

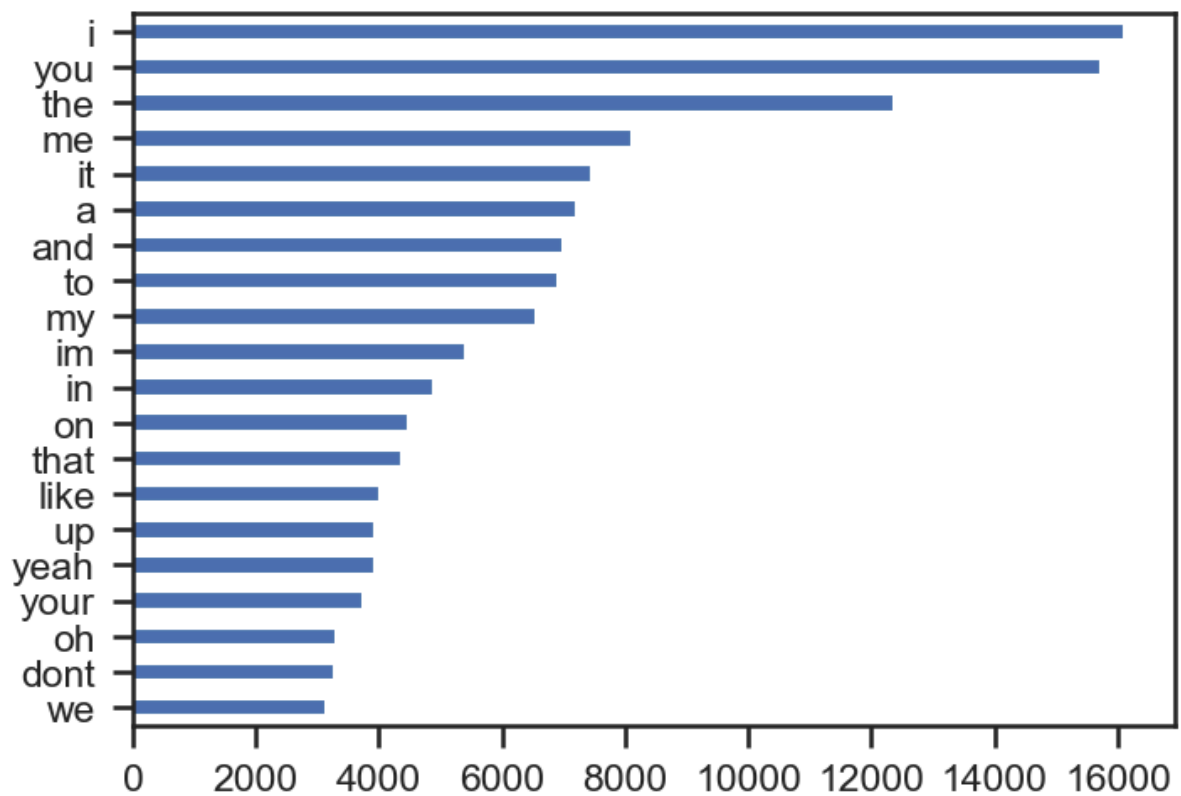
Mostriamo di seguito che le 15 parole più frequenti nei testi non pre-processati sono infatti stopwords:

```
In [58]: word_count = Counter(all_words)
most_common_words = word_count.most_common(15)
print("Parole più comuni:", most_common_words)
```

Parole più comuni: [('i', 16124), ('you', 15726), ('the', 12379), ('me', 8104), ('it', 7447), ('a', 7218), ('and', 7002), ('to', 6903), ('my', 6545), ('im', 5401), ('in', 4881), ('on', 4466), ('that', 4357), ('like', 4013), ('up', 3922)]

```
In [59]: pd.Series(' '.join(lyrics.Lyrics).lower().split()).value_counts()[:20][::-1].plot(kind='bar')
```

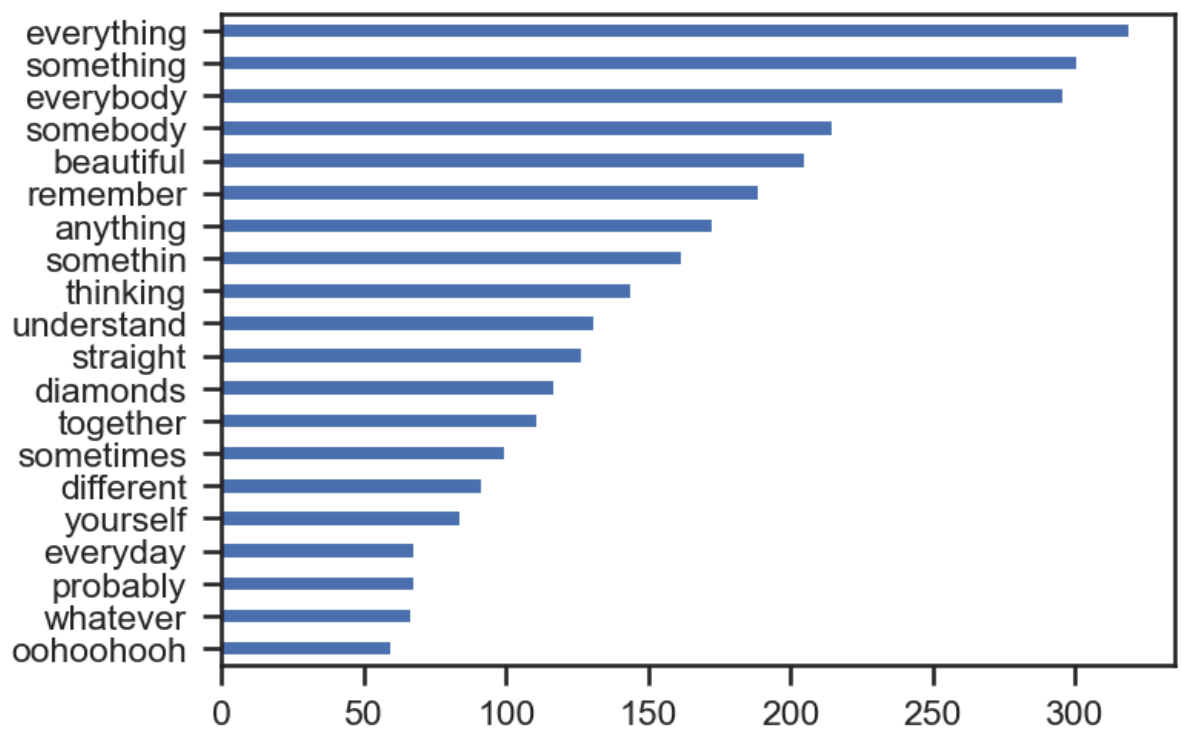
Out[59]: <Axes: >



Vediamo anche le parole lunghe (più di 8 caratteri) più presenti.

```
In [60]: # top Long words
pd.Series([w for w in ' '.join(lyrics.Lyrics).lower().split() if len(w) > 7]).value
```

```
Out[60]: <Axes: >
```



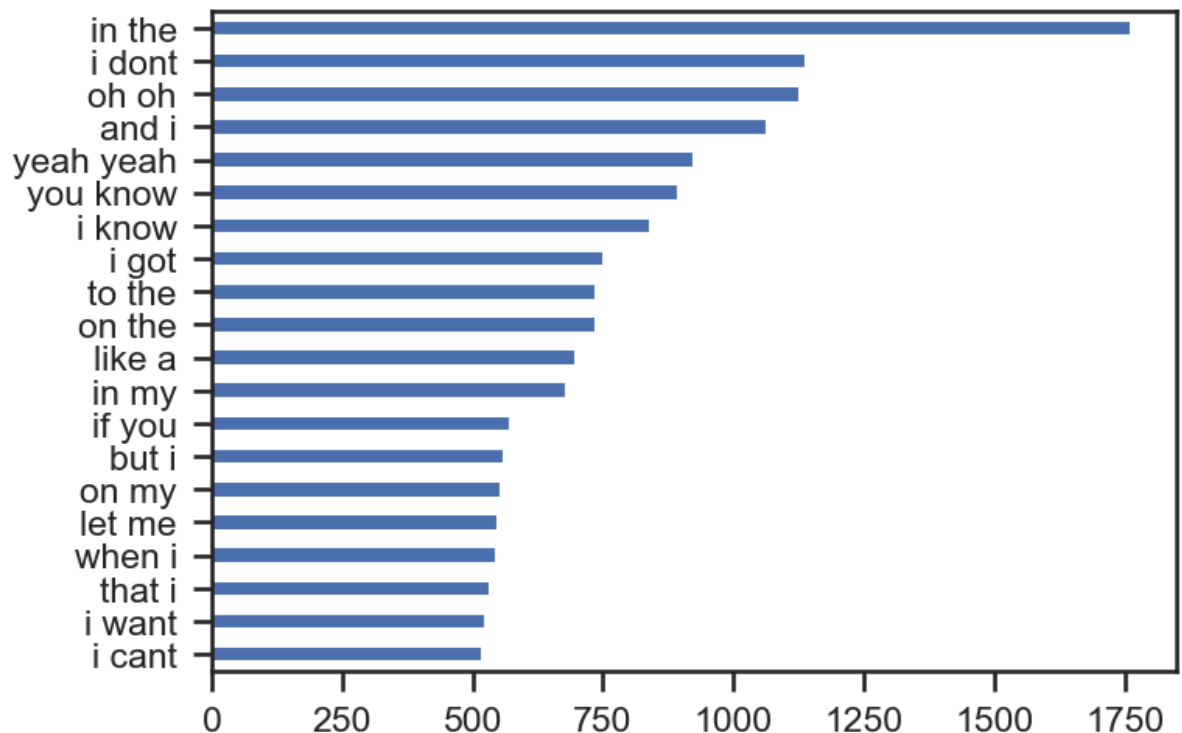
1.3.2. Gli n-gram più frequenti

Mostriamo i gruppi di parole (n-gram) che compaiono più frequentemente insieme.

```
In [61]: def get_ngrams_from_series(series, n=2):
lines = ' '.join(series).lower().split('\n')
lgrams = [ngrams(l.split(), n) for l in lines]
grams = [[' '.join(g) for g in list(lg)] for lg in lgrams]
return [item for sublist in grams for item in sublist]
```

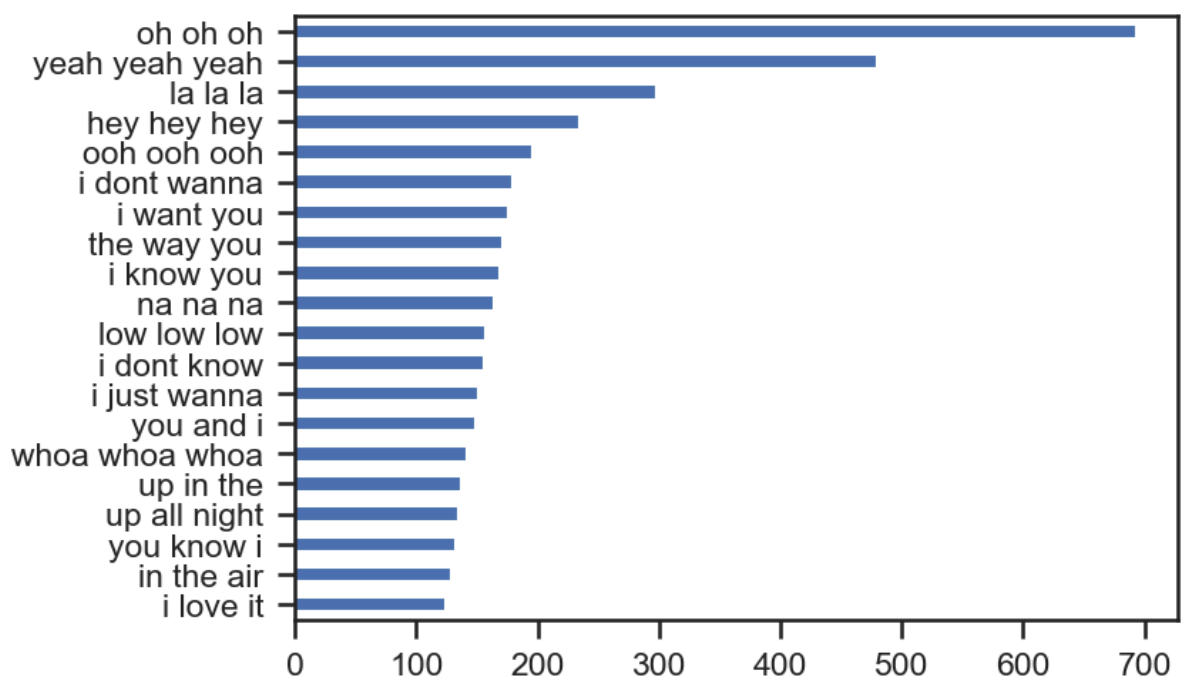
```
In [62]: # Top bi-grams
pd.Series(get_ngrams_from_series(lyrics.Lyrics, 2)).value_counts()[:20][::-1].plot()
```

Out[62]: <Axes: >



```
In [63]: # Top tri-grams
pd.Series(get_ngrams_from_series(lyrics.Lyrics, 3)).value_counts()[:20][::-1].plot()
```

Out[63]: <Axes: >



1.3.3. Parole più frequenti con preprocessing

Rimuoviamo le stopwords e diamo un'occhiata alle parole più comuni che hanno anche un'importanza semantica.

```
In [64]: # Per prima cosa, trasformiamo le Lyrics, che precedentemente abbiamo reso liste di
df['Lyrics'] = df['Lyrics'].apply(lambda x: ' '.join(map(str, x)))
df.head()
```

```
Out[64]:
```

	Song	Artist Names	Hot100 Ranking Year	Hot100 Rank	Lyrics
5584	TiK ToK	['kesha']	2010	1	wake up in the mornin feelin like p diddy hey ...
5585	Need You Now	['lady a']	2010	2	picture perfect memories scattered all around ...
5586	Hey, Soul Sister	['train']	2010	3	hey hey hey your lipstick stains on the front ...
5587	California Gurls	['katy perry', 'snoop dogg']	2010	4	greetings loved ones lets take a journey i kno...
5588	OMG (feat. will.i.am)	['usher', 'will i am']	2010	5	oh my gosh baby let me l i did it again so ima...

```
In [65]: # Funzione che pulisce il testo preso in input. Rimuove caratteri speciali, numeri,
# tutto in lowercase.
def preprocess(text: str) -> str:
    text = re.sub("[^A-Za-z]+", " ", text) # rimuoviamo numeri e caratteri speciali
    # rimuoviamo le stopwords
    # 1. creiamo i token
    tokens = nltk.word_tokenize(text)
    # 2. se il token è una stopwords, viene rimosso.
    tokens = [w.lower().strip() for w in tokens if not w.lower() in sw]
    return tokens

df["Lyrics"] = df['Lyrics'].apply(lambda x: preprocess(x))
```

Adesso abbiamo a disposizione testi senza stopwords, solo con parole che portano significato.

```
In [66]: df.head()
```


Out[66]:

	Song	Artist Names	Hot100 Ranking Year	Hot100 Rank	Lyrics
5584	TiK ToK	['kesha']	2010	1	[wake, mornin, feelin, diddy, girl, grab, glas...
5585	Need You Now	['lady a']	2010	2	[picture, perfect, memories, scattered, floor,...
5586	Hey, Soul Sister	['train']	2010	3	[lipstick, stains, lobe, left, brains, knew, f...
5587	California Gurls	['katy perry', 'snoop dogg']	2010	4	[loved, journey, place, grass, greener, warm, ...
5588	OMG (feat. will.i.am)	['usher', 'will i am']	2010	5	[gosh, baby, beat, rock, baby, love, ways, lov...

A questo punto ha senso guardare e analizzare le parole più frequenti.

```
In [67]: all_words = [word for text in df['Lyrics'] for word in text]
word_freq = Counter(all_words)
most_common_words = word_freq.most_common(10)
print("\nParole più frequenti:")
for word, freq in most_common_words:
    print(f"{word}: {freq}")
```

Parole più frequenti:

```
love: 2646
baby: 2281
girl: 1250
time: 1021
feel: 935
night: 860
good: 733
life: 730
money: 641
heart: 637
```

1.3.4. Parole più importanti

Mostriamo i **termini più importanti** contenuti nel nostro dataset facendo uso dell'algoritmo **TF-IDF**.

Il TF-IDF (term frequency-inverse document frequency) è un **algoritmo di estrazione di informazioni testuali da un documento** (o una collezione di documenti). La funzione permette di misurare **l'importanza** di ogni termine rispetto ad un documento o ad una collezione di documenti.

L'algoritmo **funziona nel modo seguente**: consideriamo N documenti e un termine T. Se T è presente in tutti gli N documenti, T è un termine comune. Se il termine T non è frequentemente presente all'interno dei documenti, ma comunque non in pochi, allora vuol dire che il termine ha un'importanza maggiore.

I termini più importanti sono termini che compaiono spesso nei documenti in cui sono presenti, ma che in generale sono poco frequenti.

L'algoritmo assegna un valore ai termini coinvolgendo due metriche:

- **Term frequency (TF)**: frequenza del termine in un documento. Tale valore è determinato dal rapporto tra il numero di occorrenze del termine e il numero delle parole nel documento. In sostanza, questo valore cresce se un termine si trova tante volte in uno stesso documento.
- **Inverse document frequency (IDF)**: questo valore cresce se il termine compare in pochi documenti tra gli N documenti. In sostanza, un termine ha un IDF alto se compare in pochi documenti.

Il valore TF-IDF associato a un termine è il prodotto di questi due valori.

Riassunto, l'algoritmo TF-IDF associa **maggiore importanza ai termini che compaiono in pochi documenti ma nei documenti in cui compaiono sono molto frequenti** e possiamo utilizzarlo per trovare le parole più importanti nei nostri testi di canzoni.

```
In [68]: joined_lyrics = df['Lyrics'].apply(lambda x: ' '.join(map(str, x)))

tfidf_vectorizer = TfidfVectorizer()
X_tfidf = tfidf_vectorizer.fit_transform(joined_lyrics)
feature_names = tfidf_vectorizer.get_feature_names_out()

# Recuperare le parole con i punteggi TF-IDF più alti
max_tfidf_per_doc = X_tfidf.max(axis=0).toarray().flatten()
sorted_items = sorted(zip(max_tfidf_per_doc, feature_names), reverse=True)
most_important_words = sorted_items[:3]
print("\nParole più importanti (TF-IDF più alto):")
for score, word in most_important_words:
    print(f"{word}: {score}")
```

```
Parole più importanti (TF-IDF più alto):
low: 0.9956578177630779
rumor: 0.9940216403121889
work: 0.9900447861384912
```

1.3.5. Parole correlate a parole frequenti e importanti

In questa sezione vediamo quali parole accompagnano più spesso le parole che abbiamo precedentemente individuato come più frequenti e importanti.

Troviamo queste connessioni utilizzando **Word2vec**, una rete neurale capace di trasformare insiemi di parole in rappresentazioni vettoriali. Questi vettori contengono informazioni sul significato delle parole in base alle parole che le accompagnano. La dimensione del vettore rappresenta la complessità dell'embedding e la **quantità di informazioni semantiche catturate**. Dimensioni più elevate catturano relazioni più sfumate, ma richiedono anche più dati e calcoli.

Word2vec prende in input un corpus di testi e restituisce un insieme di vettori che rappresentano la **distribuzione semantica** delle parole nel testo. Per ogni parola contenuta nel corpus, viene costruito un vettore in modo da rappresentarla come un punto nello spazio multidimensionale.

La rete neurale è capace di **riconoscere le relazioni tra le parole**. In particolare, parole che sembrano apparire in contesti simili, sono mappate a vettori vicini. Una volta che il modello è stato trainato, gli embedding delle parole che condividono un contesto simile nel corpus

(i.e. parole semanticamente simili) sono vicine nello spazio vettoriale. Di conseguenza, le parole dissimili sono localizzate lontane l'una dall'altra.

Gli elementi fondamentali per Word2vec sono:

- **Continuous Bag of Words (CBOW)**: modello per trainare i word embedding che predice una parola dato il contesto.
- **Skip Gram**: in contrapposizione al precedente, modello che predice il contesto data una parola.

Inoltre Word2vec è molto efficiente e veloce poiché fa uso del **negative sampling**, ossia nel training della rete neurale ignora gli elementi non utili alla predizione, i quali sono caratterizzati da pesi negativi.

```
In [69]: # trainiamo il modello
model = Word2Vec(sentences=df['Lyrics'].tolist())
```

```
In [70]: # Analisi delle parole correlate a quelle più frequenti
print("Parole correlate alle parole più frequenti:")
for word in most_common_words:
    similar_words = model.wv.most_similar(positive=[word], topn=3)
    print(f"\nParole correlate con '{word}': ")
    for similar_word in similar_words:
        print(f"{similar_word}")
```

Parole correlate alle parole più frequenti:

Parole correlate con '('love', 2646)':
 ('behave', 0.9780300855636597)
 ('long', 0.9286976456642151)
 ('hopeless', 0.9122680425643921)

Parole correlate con '('baby', 2281)':
 ('adorn', 0.9390478730201721)
 ('whistle', 0.9029958844184875)
 ('showing', 0.878018856048584)

Parole correlate con '('girl', 1250)':
 ('body', 0.9213047027587891)
 ('discovering', 0.8560878038406372)
 ('earned', 0.807239294052124)

Parole correlate con '('time', 1021)':
 ('mind', 0.9568067193031311)
 ('granted', 0.9406144618988037)
 ('matters', 0.9375675916671753)

Parole correlate con '('feel', 935)':
 ('coming', 0.9273994565010071)
 ('matters', 0.9018546938896179)
 ('babe', 0.8929051160812378)

Parole correlate con '('night', 860)':
 ('lucky', 0.9427092671394348)
 ('stay', 0.8856745958328247)
 ('llove', 0.8697975277900696)

Parole correlate con '('good', 733)':
 ('granted', 0.9259580373764038)
 ('feeling', 0.9162207245826721)
 ('time', 0.9161772131919861)

Parole correlate con '('life', 730)':
 ('feeling', 0.9210057854652405)
 ('youngblood', 0.9152274131774902)
 ('matters', 0.9150844216346741)

Parole correlate con '('money', 641)':
 ('biz', 0.8933870196342468)
 ('mob', 0.8817222118377686)
 ('signs', 0.8628443479537964)

Parole correlate con '('heart', 637)':
 ('break', 0.9682520031929016)
 ('chains', 0.9440596103668213)
 ('ache', 0.9435321092605591)

```
In [72]: # Analisi delle parole correlate a quelle più importanti
print("Parole correlate alle parole più importanti:")
for word in most_important_words:
    similar_words = model.wv.most_similar(positive=[word[1]], topn=3)
    print(f"\nParole correlate con '{word}': ")
    for similar_word in similar_words:
        print(f"{similar_word}")
```

Parole correlate alle parole più importanti:

Parole correlate con '(0.9956578177630779, 'low')':
 ('repping', 0.9814177751541138)
 ('representing', 0.9810830354690552)
 ('starts', 0.9662962555885315)

Parole correlate con '(0.9940216403121889, 'rumor')':
 ('core', 0.8100433945655823)
 ('dyou', 0.8097468018531799)
 ('week', 0.7851085662841797)

Parole correlate con '(0.9900447861384912, 'work')':
 ('miracle', 0.9391815066337585)
 ('mountains', 0.9174107313156128)
 ('earned', 0.9166923761367798)

1.4. Analisi delle canzoni

Passiamo dall'analizzare i termini singolarmente, ad analizzare i testi nella loro complessità.

1.4.1. Temi più frequenti

In questa sezione analizziamo i temi più frequenti tra i testi. Questo tipo di analisi è più specifica rispetto alla sola analisi dei termini, poiché coinvolge molte più variabili.

In NLP, l'**allocazione Dirichlet latente (LDA)** è un modello probabilistico capace di **estrarre i temi a partire da documenti testuali**. Un documento è considerato come un **insieme di argomenti**, e ogni argomento è caratterizzato da una specifica distribuzione di parole.

Il modello analizza un testo e ogni parola in esso viene associata a un tema specifico. Se le parole di un determinato tema ricorrono maggiormente all'interno di un documento, è molto probabile che questo documento appartenga a tale tema.

Poiché LDA è un modello che effettua **clustering non-supervisionato**, non sono fornite etichette per i cluster, ma il significato di ogni cluster deve essere interpretato a partire dai suoi rappresentanti.

```
In [73]: joined_lyrics = df['Lyrics'].apply(lambda x: ' '.join(map(str, x)))

# countvectorizer trasforma una collezione di documenti in una matrice
count_vectorizer = CountVectorizer(max_df=0.95, min_df=2, stop_words='english')
X_counts = count_vectorizer.fit_transform(joined_lyrics)

# applichiamo LDA
num_topics = 5
lda = LatentDirichletAllocation(n_components=num_topics, random_state=42)
lda.fit(X_counts)

words = count_vectorizer.get_feature_names_out()
```

```
In [74]: def display_topics(model, feature_names, num_top_words):
    for topic_idx, topic in enumerate(model.components_):
        print(f"Topic {topic_idx}:")
        print(" ".join([feature_names[i] for i in topic.argsort()[::-num_top_words :
                                -1]])

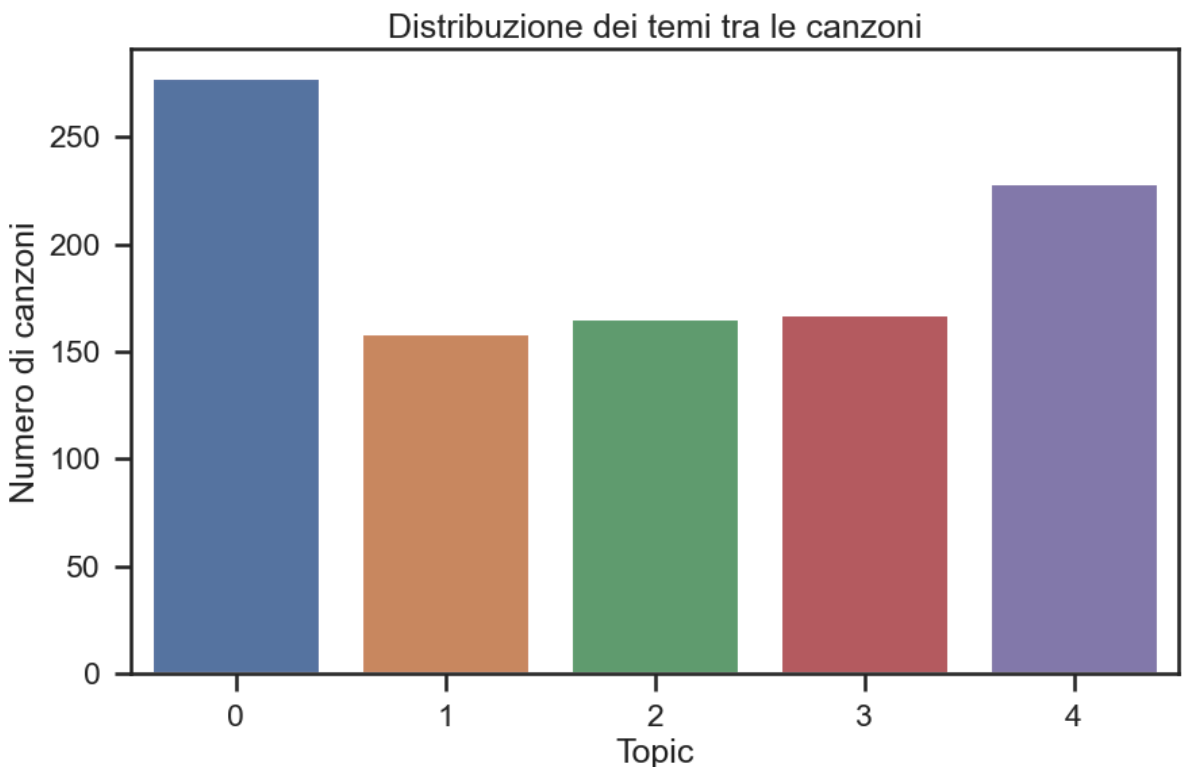
    num_top_words = 5
```

```
display_topics(lda, words, num_top_words)

# aggiungiamo i risultati del topic modeling al df originale
topics = lda.transform(X_counts)
df['Topic'] = topics.argmax(axis=1)

plt.figure(figsize=(10, 6))
sns.countplot(x='Topic', data=df)
plt.xlabel('Topic')
plt.ylabel('Numero di canzoni')
plt.title('Distribuzione dei temi tra le canzoni')
plt.show()
```

Topic 0:
love baby night girl time
Topic 1:
girl good turn time work
Topic 2:
money watch man bad bitches
Topic 3:
love low life girl feel
Topic 4:
baby love long time feel



2. Generazione dei testi

La generazione del testo 'perfetto' sarà affidata all'AI più utilizzata: **ChatGPT nella versione GPT-4o**. Per capire come funziona la generazione di testo abbiamo prodotto due esempi dimostrativi in cui svilupperemo:

- Una **rete neurale** addestrata sui testi del dataset che proverà a **prevedere**, dato un contesto, la parola più probabile.
- Una funzione in grado di generare testo utilizzando le **catene di Markov**.

2.1. Predizione della prossima parola

In questa sezione addestreremo un modello che proverà a predire le possibili parole successive ad una stringa data in input.

Per la realizzazione di questa rete, avremo bisogno di conoscere i concetti di:

- **RNN e LSTM**
- **One Hot Encoding**

2.1.1. RNN e LSTM

Una **Recurrent Neural Network (RNN)** è un insieme di reti neurali che si passano in catena un messaggio: l'obiettivo è quello di cercare di **ricordare informazioni**, simulando il comportamento del cervello umano.

Negli ultimi anni le RNN sono state utilizzate per risolvere diverse task, tra cui il riconoscimento vocale e il captioning delle immagini.

Un limite importante delle RNN è quello di non riuscire a 'ricordare' un grande numero di informazioni per lungo tempo: **all'aumentare dello spazio tra le dipendenze diminuisce l'efficacia delle RNN nel 'ricordarle'**.

Per ovviare a questo problema si utilizzano le **Long Short Term Memory(LSTM)**.

Ogni modulo LSTM contiene **4 strati che interagiscono tra loro**. Ogni strato ha il compito di **modificare lo stato del modulo**, in modo da decidere quanta informazione far passare in base alla sua importanza nel contesto.

2.1.2. One Hot Encoding

La tecnica di **One Hot Encoding** prevede di suddividere i dati in **array di una lunghezza predefinita**: nel nostro caso, un array sarà lungo come la lunghezza del testo della canzone ed ogni array identificherà una parola. Assegneremo il valore 1 ad un solo bit e questo rappresenterà la posizione della parola all'interno della frase (e.g. in una frase composta da 5 parole, la terza parola della frase sarà rappresentata dall'array "[0,0,1,0,0]"). Per rendere univoco il riferimento alla parola, questa avrà assegnato un indice. Introduciamo anche un **vocabolario che identifica** le parole possibili all'interno della frase.

Esempio:

Vocabolario: [I, you, love]

I = 1, you = 2, love = 3

"I love you " = [1,3,2]

love = [0,1,0] = 1 all'interno della frase

Grazie a questa tecnica otteniamo una **matrice che identifica la frase**: in questo modo sarà più semplice calcolare prodotti puntuali, importanti per il calcolo della probabilità del termine successivo.

2.1.3. Preparazione dei dati

```
In [28]: exdf = pd.read_csv(filename)
exdf = exdf.loc[(exdf['Hot100 Ranking Year'] >= 2010) & (exdf['Hot100 Ranking Year']
```

```
exdf.head()
```

Out[28]:

	Song	Artist Names	Hot100 Ranking Year	Hot100 Rank	Lyrics
5584	TiK ToK	['kesha']	2010	1	['wake', 'up', 'in', 'the', 'mornin', 'feelin'...
5585	Need You Now	['lady a']	2010	2	['picture', 'perfect', 'memories', 'scattered'...
5586	Hey, Soul Sister	['train']	2010	3	['hey', 'hey', 'hey', 'your', 'lipstick', 'sta...
5587	California Gurls	['katy perry', 'snoop dogg']	2010	4	['greetings', 'loved', 'ones', 'lets', 'take',...
5588	OMG (feat. will.i.am)	['usher', 'will i am']	2010	5	['oh', 'my', 'gosh', 'baby', 'let', 'me', 'I',...

```
In [29]: tokenizer = RegexpTokenizer(r'\w+')
exdf['Lyrics'] = exdf['Lyrics'].apply(lambda x: tokenizer.tokenize(x.lower()) if np
```

```
In [30]: exdf['Lyrics'] = exdf['Lyrics'].apply(lambda x: ' '.join(map(str, x)))
exdf.head()
```

Out[30]:

	Song	Artist Names	Hot100 Ranking Year	Hot100 Rank	Lyrics
5584	TiK ToK	['kesha']	2010	1	wake up in the mornin feelin like p diddy hey ...
5585	Need You Now	['lady a']	2010	2	picture perfect memories scattered all around ...
5586	Hey, Soul Sister	['train']	2010	3	hey hey hey your lipstick stains on the front ...
5587	California Gurls	['katy perry', 'snoop dogg']	2010	4	greetings loved ones lets take a journey i kno...
5588	OMG (feat. will.i.am)	['usher', 'will i am']	2010	5	oh my gosh baby let me l i did it again so ima...

```
In [31]: text = list(exdf['Lyrics'])
joined_text = " ".join(text)
```

```
In [32]: tokenizer = RegexpTokenizer(r"\w+")
tokens = tokenizer.tokenize(joined_text)
```

```
In [33]: unique_tokens=np.unique(tokens)
unique_tokens_index = {token: idx for idx, token in enumerate(unique_tokens)}
```



```
In [34]: n_words = 5
input_words = []
next_words = []

for i in range(len(tokens) - n_words):
    input_words.append(tokens[i:i + n_words])
    next_words.append(tokens[i + n_words])

X=np.zeros((len(input_words), n_words, len(unique_tokens)), dtype=bool)
Y=np.zeros((len(next_words), len(unique_tokens)), dtype= bool)
```

```
In [35]: #one hot encoding
for i, words in enumerate(input_words):
    for j, word in enumerate(words):
        X[i, j, unique_tokens_index[word]]=1
    Y[i, unique_tokens_index[next_words[i]]] = 1
```

2.1.4. Creazione del modello

```
In [36]: #Inizializzazione del modello
model = keras.models.Sequential()
#Aggiunta di strati
model.add(keras.layers.LSTM(128, input_shape=(n_words, len(unique_tokens)),return_sequences=True))
model.add(keras.layers.LSTM(128))
model.add(keras.layers.Dense(len(unique_tokens), activation='softmax')) #Strato di output
#Compilazione del modello per il training
opt = keras.optimizers.Adamax(learning_rate=0.01)
model.compile(loss='categorical_crossentropy', optimizer=opt,metrics=["accuracy"])

#Riassunto del modello
model.summary()
```

C:\Users\erika\anaconda3\Lib\site-packages\keras\src\layers\rnn\rnn.py:204: UserWarning: Do not pass an `input_shape`/`input_dim` argument to a layer. When using Sequential models, prefer using an `Input(shape)` object as the first layer in the model instead.

super().__init__(**kwargs)

Model: "sequential"

Layer (type)	Output Shape	Param #
lstm (LSTM)	(None, 5, 128)	1,735,680
lstm_1 (LSTM)	(None, 128)	131,584
dense (Dense)	(None, 3261)	420,669

Total params: 2,287,933 (8.73 MB)

































Trainable params: 2,287,933 (8.73 MB)

Non-trainable params: 0 (0.00 B)

2.1.5. Training del modello

```
In [37]: history = model.fit(X, Y, batch_size=128, epochs=50, shuffle=True)
```

```

Epoch 1/50
325/325  22s 51ms/step - accuracy: 0.0334 - loss: 6.4515
Epoch 2/50
325/325  16s 50ms/step - accuracy: 0.0470 - loss: 5.8558
Epoch 3/50
325/325  17s 51ms/step - accuracy: 0.0807 - loss: 5.4831
Epoch 4/50
325/325  17s 51ms/step - accuracy: 0.1419 - loss: 4.9586
Epoch 5/50
325/325  16s 50ms/step - accuracy: 0.2039 - loss: 4.4392
Epoch 6/50
325/325  16s 50ms/step - accuracy: 0.2713 - loss: 3.9873
Epoch 7/50
325/325  17s 51ms/step - accuracy: 0.3428 - loss: 3.5737
Epoch 8/50
325/325  16s 50ms/step - accuracy: 0.4109 - loss: 3.1677
Epoch 9/50
325/325  16s 51ms/step - accuracy: 0.4630 - loss: 2.8429
Epoch 10/50
325/325  16s 50ms/step - accuracy: 0.5228 - loss: 2.5109
Epoch 11/50
325/325  17s 51ms/step - accuracy: 0.5677 - loss: 2.2540
Epoch 12/50
325/325  17s 52ms/step - accuracy: 0.6049 - loss: 2.0131
Epoch 13/50
325/325  17s 51ms/step - accuracy: 0.6468 - loss: 1.7897
Epoch 14/50
325/325  16s 51ms/step - accuracy: 0.6849 - loss: 1.5959
Epoch 15/50
325/325  17s 52ms/step - accuracy: 0.7163 - loss: 1.4234
Epoch 16/50
325/325  17s 51ms/step - accuracy: 0.7409 - loss: 1.2921
Epoch 17/50
325/325  17s 51ms/step - accuracy: 0.7723 - loss: 1.1425
Epoch 18/50
325/325  16s 51ms/step - accuracy: 0.8003 - loss: 1.0147
Epoch 19/50
325/325  17s 51ms/step - accuracy: 0.8256 - loss: 0.8870
Epoch 20/50
325/325  17s 51ms/step - accuracy: 0.8424 - loss: 0.8029
Epoch 21/50
325/325  17s 51ms/step - accuracy: 0.8635 - loss: 0.6898
Epoch 22/50
325/325  17s 51ms/step - accuracy: 0.8846 - loss: 0.5911
Epoch 23/50
325/325  17s 51ms/step - accuracy: 0.8995 - loss: 0.5259
Epoch 24/50
325/325  17s 54ms/step - accuracy: 0.9137 - loss: 0.4445
Epoch 25/50
325/325  17s 52ms/step - accuracy: 0.9241 - loss: 0.3965
Epoch 26/50
325/325  17s 51ms/step - accuracy: 0.9357 - loss: 0.3352
Epoch 27/50
325/325  17s 52ms/step - accuracy: 0.9384 - loss: 0.3097
Epoch 28/50
325/325  17s 51ms/step - accuracy: 0.9432 - loss: 0.2869
Epoch 29/50
325/325  17s 52ms/step - accuracy: 0.9568 - loss: 0.2101
Epoch 30/50
325/325  19s 57ms/step - accuracy: 0.9574 - loss: 0.1949
Epoch 31/50
325/325  20s 62ms/step - accuracy: 0.9623 - loss: 0.1689
Epoch 32/50
325/325  20s 62ms/step - accuracy: 0.9625 - loss: 0.1640

```

```

Epoch 33/50
325/325 ————— 20s 63ms/step - accuracy: 0.9628 - loss: 0.1573
Epoch 34/50
325/325 ————— 20s 62ms/step - accuracy: 0.9624 - loss: 0.1576
Epoch 35/50
325/325 ————— 20s 62ms/step - accuracy: 0.9628 - loss: 0.1545
Epoch 36/50
325/325 ————— 20s 62ms/step - accuracy: 0.9645 - loss: 0.1318
Epoch 37/50
325/325 ————— 20s 60ms/step - accuracy: 0.9653 - loss: 0.1322
Epoch 38/50
325/325 ————— 19s 59ms/step - accuracy: 0.9623 - loss: 0.1346
Epoch 39/50
325/325 ————— 20s 62ms/step - accuracy: 0.9617 - loss: 0.1394
Epoch 40/50
325/325 ————— 20s 63ms/step - accuracy: 0.9631 - loss: 0.1315
Epoch 41/50
325/325 ————— 19s 59ms/step - accuracy: 0.9638 - loss: 0.1283
Epoch 42/50
325/325 ————— 20s 62ms/step - accuracy: 0.9661 - loss: 0.1094
Epoch 43/50
325/325 ————— 21s 63ms/step - accuracy: 0.9644 - loss: 0.1151
Epoch 44/50
325/325 ————— 21s 63ms/step - accuracy: 0.9635 - loss: 0.1170
Epoch 45/50
325/325 ————— 21s 63ms/step - accuracy: 0.9635 - loss: 0.1158
Epoch 46/50
325/325 ————— 21s 63ms/step - accuracy: 0.9612 - loss: 0.1178
Epoch 47/50
325/325 ————— 21s 63ms/step - accuracy: 0.9658 - loss: 0.1103
Epoch 48/50
325/325 ————— 21s 63ms/step - accuracy: 0.9634 - loss: 0.1135
Epoch 49/50
325/325 ————— 21s 63ms/step - accuracy: 0.9674 - loss: 0.1046
Epoch 50/50
325/325 ————— 21s 63ms/step - accuracy: 0.9666 - loss: 0.1060

```

```

In [38]: #salvo il modello
         model.save("mymodel.keras")

```

```

In [40]: history_df = pd.DataFrame(history.history)

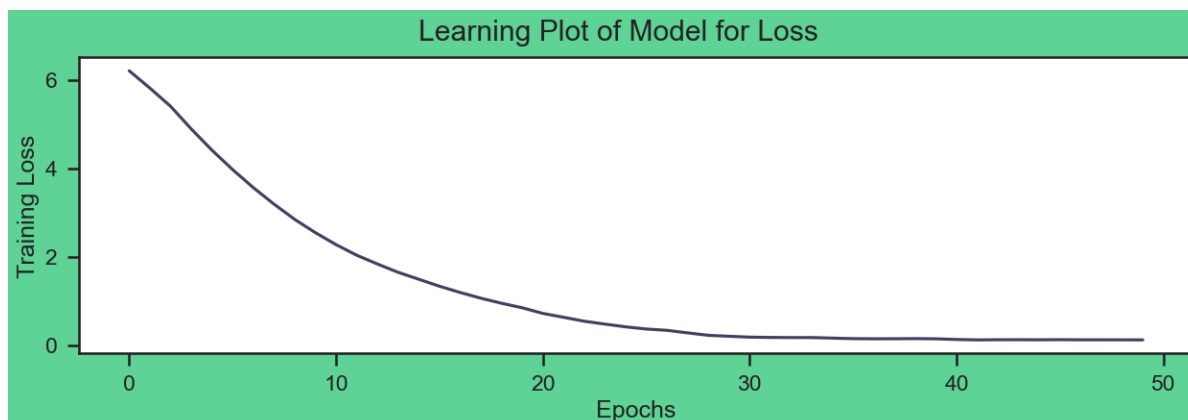
         fig = plt.figure(figsize=(15,4), facecolor="#63D297")
         fig.suptitle("Learning Plot of Model for Loss")
         pl=sns.lineplot(data=history_df["loss"],color="#444160")
         pl.set(ylabel = "Training Loss")
         pl.set(xlabel = "Epochs")

```

```

Out[40]: [Text(0.5, 0, 'Epochs')]

```




```

song = []
if startswith:
    song.append(self.gen_sentence(length=length, startswith=startswith))
    lines -= 1
for _ in range(lines):
    sent_len = random.randint(*length_range) if length_range else length
    song.append(self.gen_sentence(length=sent_len))
return '\n '.join(song)

```

```

In [24]: # random song
lyrics_generator = markov(corpus=' '.join(lyrics.Lyrics))
lyrics_generator.gen_song(lines=10, length_range=[5, 10])

```

```

Out[24]: 'alright cleaned a lot whats the name of\n when im feelin no you cant say were sat
isfied angie\n cause it took so long and i aint seen\n man after him waits here ma
belle amie apres tous\n swear they still are true i really\n we are miles apart if
you give\n hunger an endless sea all we do yeah do it\n music makes your hair and
a few other changes if you\n ladder is waiting for her locked deep inside im so ti
red of\n you cried the day throwing stones at'

```

```

In [25]: # song on topics
lyrics_generator.gen_song(lines=10, length_range=[5, 10], startswith='love')

```

```

Out[25]: 'peace love and i forgot to say i never promised\n by me im only tryin to get off
\n opened up the booze and the bears convoy off your trip\n over there ill take yo
u high love is fadin i\n something quick you gonna see inside you\n byebye id like
to ask is that\n and free my soul i wanna rock and roll band i\n that to the croco
dile rocking is something i could\n around me the indians said i aint got no\n thr
ow in my brain is squirmin like a river thats'

```

```

In [53]: lyrics_generator.gen_song(lines=10, length_range=[5, 10], startswith='vodka')

```

```

Out[53]: 'some vodka some chonky konk and watch tv and she\njust chill with me oh oh oh\ndo
wn then we all want a bag yeah\nworryin bout whoevers next i am with you\nthem dig
its yeah dont like the ceiling cant\na little when i think i care what you\nshe go
t it girl come here boy i mustve\nadmit it youve seen all this time im just sittin
g in\nknow somebody i can turn crumbs to bricks and i dont wanna\nthink youre repl
aying in my heart was a fairytale you were'

```

2.3. Generazione del testo con chat GPT

2.3.1. Cos'è ChatGPT?

Per generare il testo vero e proprio attraverso cui trarremo le nostre conclusioni è **ChatGPT** nella sua versione gpt-4o.

ChatGPT è un **trasformatore generativo pre-allenato** (Generative Pre-trained Transformer) sottoforma di Chat bot. Il suo compito è quello di prendere in input una domanda da parte di un utente per poter **generare una risposta in linguaggio naturale** ed informativa. La versione GPT-4o è in anche grado di elaborare diversi tipi di file, anche vocali.

Alla base di questo tool c'è un **LLM** (Large Language Model), motivo per cui è in grado di simulare il linguaggio naturale.

Introduciamo brevemente i concetti di trasformatore e di LLM.

2.3.2. Trasformatori

Nel contesto dell'intelligenza artificiale, un trasformatore è un tipo di modello di deep learning utilizzato principalmente per compiti di **elaborazione del linguaggio naturale (NLP)**.

Introdotta nel 2017 nel documento "**Attention is All You Need**", il trasformatore ha rivoluzionato il campo del NLP grazie alla sua capacità di gestire in modo efficace la **dipendenza a lungo termine nei dati sequenziali**.

La struttura di un trasformatore è di tipo **encoder-decoder**, a cui si aggiungono elementi (e.g. attention) per ottimizzare l'elaborazione dei dati.

Il compito dell'**encoder** è prendere una sequenza di input e trasformarla in una rappresentazione densa. È costituito da una serie di strati, ognuno dei quali contiene sotto-strati di attenzione e di feed-forward.

Il **decoder** prende la rappresentazione densa prodotta dall'encoder e genera la sequenza di output. Anche il decoder è costituito da una serie di strati simili a quelli dell'encoder, ma con meccanismi aggiuntivi per gestire l'input precedente nella sequenza di output.

Il meccanismo di **self-attention** permette al modello di pesare l'importanza di diverse parti della sequenza di input per ogni parte della stessa sequenza. Ogni elemento della sequenza può "prestare attenzione" a ogni altro elemento, calcolando dei pesi di attenzione.

Ogni strato dell'encoder e del decoder contiene una rete neurale **feed-forward** completamente connessa, che elabora la rappresentazione ottenuta dallo strato di self-attention.

2.3.3. LLM

Un **LLM (Large Language Model)** è un tipo di modello di intelligenza artificiale basato su architetture di deep learning, progettato per **comprendere e generare testo umano** in modo coerente e contestualmente rilevante.

Gli LLM sono **addestrati su vasti corpus di testo** e utilizzano modelli come i trasformatori per acquisire una profonda comprensione delle strutture linguistiche.

Gli LLM sono inizialmente addestrati su grandi quantità di testo (spesso l'intero corpus di Internet) con obiettivi di apprendimento non supervisionati, come la predizione della parola successiva (per modelli generativi) o la mascheratura di parole.

Gli LLM hanno un **numero estremamente elevato di parametri** (miliardi), che rappresentano la capacità del modello di apprendere e memorizzare informazioni. Più parametri ha un modello, maggiore è la sua capacità di rappresentare conoscenze complesse.

Molti LLM, come GPT, utilizzano solo la parte decoder di un trasformatore per la generazione del testo. Altri modelli, come **BERT** (Bidirectional Encoder Representations from Transformers), utilizzano solo l'encoder per compiti di comprensione del testo.

2.2.4. L'importanza del prompt

Per ottenere una risposta soddisfacente è indispensabile formulare bene il prompt (la domanda). Il prompt deve contenere tutte le indicazioni che permettano di generare la risposta che si sta cercando.

2.2.5. Valutazione del testo generato

```
In [24]: file = open("chatgpt_twentytens.txt", 'r')  
content = file.read()  
print(content)
```

//PROMPT 1:Hello! I am analyzing a dataset containing information about songs that were in the billboard top 100 between the years 2010 and 2020(year 2020 excluded). These information are: Song title, Artist, Lyrics, Ranking and Year). To confirm the results of my analysis, I need you to generate new lyrics for a new song that would rank for sure in this kind of ranking based only on the songs that were in this ranking in the years I told you (from 2010 until 2020 excluded). Can you do this? Remember to not consider any additional information: it's important to consider just the songs that are in the rankings and their lyrics to not compromise the result of my analysis. I just need the lyrics without any further information about its structure(such as verses, choruses...consider it but don't print out the structure while printing the song).

Sure, I can generate new lyrics inspired by the songs that were in the Billboard Top 100 between 2010 and 2019. Here are the lyrics:

Turn up the lights, let's go all night,
Feeling the beat, everything's alright,
You and me, we got that vibe,
Living our dream, side by side.

Lost in the rhythm, hearts on fire,
We keep on moving, never tire,
City lights shining bright,
Together we rise, into the night.

Love is our anthem, can't deny,
With you, I touch the sky,
Holding on, never letting go,
This is our moment, let it flow.

Dancing through life, no regrets,
Making memories we'll never forget,
Hands up, feeling free,
This is our story, you and me.

Chasing dreams, we won't look back,
On this journey, we stay on track,
Sing our song, loud and clear,
With you, I've got no fear.

In this world, just you and I,
With every beat, we reach the sky,
Forever young, hearts collide,
In this love, we'll always ride.

These lyrics capture the themes of love, freedom, and celebration that are common in many popular songs from 2010 to 2019.

Il contenuto del testo generato dall'LLM contiene effettivamente elementi estratti dalle analisi effettuate precedentemente nel notebook. Sono presenti sia alcuni dei temi individuati, sia alcuni dei termini più comuni.