Gestione delle playlist di Spotify

Componenti del gruppo

- Emanuele Fontana, [MAT. 758344], e.fontana7@studenti.uniba.it

Link GitHub: progetto

A.A. 2023-2024

Indice

Capitolo 0) Introduzione

L'obiettivo di questo progetto è quello di riorganizzare le playlist di Spotify tenendo conto delle caratteristiche (features) delle canzoni. Successivamente il sistema dovrà essere in grado di suggerire all'utente in quale delle nuove playlist dovrà essere inserita una canzone che fino a quel momento non è mai stata inserita dentro alcuna playlist.

Requisiti funzionali

Il progetto è stato realizzato in Python in quanto è un linguaggio che offre a disposizione molte librerie che permetto di trattare i dati in modo facile e intuitivo. Versione Python: 3.11

IDE utilizzato: PyCharm

Librerie utilizzate:

- **matplotlib**: visualizzazione dei grafici (curve di apprendimento, grafici a barre, grafici a torta)
- **networkx**: visualizzazione di grafi (usato per osservare la struttura della rete bayesiana)
- **numpy**: libreria per gestire array
- **pandas**: importazione dei dataset .csv
- **pgmpy**: creazione della rete bayesiana
- **scikit_learn**: libreria utilizzata per apprendimento automatico
- **spotipy**: mette a disposizione tutte le API per interagire con spotify
- **tqdm**: barre di avanzamento
- **pyswip**: utilizzo di Prolog all'intenro dell'ambiente python
- **imblearn:** libreria utilizzata per oversampling tramite SMOTE

Installazione e avvio

Aprire il progetto con l'IDE preferito. Nel codice sono presenti alcune istruzioni commentante in quanto possono essere eseguite solo una volta (esempio: l'installazione dei requisiti). Avviare il programma partendo dal file main.py

Capitolo 1) Creazione del dataset

Il dataset è stato generato mediante i dati che è possibile richiedere a Spotify. Nello specifico tra tutte le info a disposizione vi era un file JSON contenente tutte le informazioni relative alle playlist da me create. Questo file JSON è stato poi utilizzato

per creare il dataset di partenza **playlist_tracks.csv** usando le API di Spotify messe a disposizione dalla libreria *spotipy*. È stato creato un oggetto della classe *Spotify*.

```
sp = spotipy.Spotify(
    auth_manager=SpotifyOAuth(scope=scope, client_id=client_id, client_secret=client_secret, redirect_uri=redirect_uri))
```

utilizzato per effettuare le seguenti chiamate:

- sp.audio_features(track_uri) -> permette di ottenere le features di una canzone dato il suo URI
- sp.track(track_uri) -> permette di ottenere informazioni relativamente alla canzone dato il suo URI

Per quanto riguarda le features utilizzate in questo progetto ho deciso di utilizzare le stesse che Spotify usa per descrivere le sue canzoni nella documentazione:

- **Danceability** -> La *danceability* descrive quanto una traccia sia adatta per ballare in base a una combinazione di elementi musicali, tra cui tempo, stabilità del ritmo, forza del ritmo e regolarità generale
- **Energy** -> L'energia è una misura da 0,0 a 1,0 e rappresenta una misura percettiva di intensità e attività. In genere, le tracce energiche sembrano veloci, forti e rumorose. Le caratteristiche percettive che contribuiscono a questo attributo includono la gamma dinamica, il volume percepito, il timbro, la velocità di insorgenza e l'entropia generale
- **Key** -> La tonalità in cui si trova la traccia. I numeri interi vengono mappati sulle altezze utilizzando la notazione standard Pitch Class. Se non è stata rilevata alcuna chiave, il valore è -1. Intervallo: -1 - 11
- **Loudness** -> Il volume complessivo di una traccia in decibel (dB). I valori del volume vengono calcolati in media sull'intera traccia e sono utili per confrontare il volume relativo delle tracce. Il volume è la qualità di un suono che è il principale correlato psicologico della forza fisica (ampiezza). I valori variano tipicamente tra -60 e 0 db.
- **Speechiness** -> La *speechiness* rileva la presenza di parole pronunciate in una traccia. Più la registrazione è esclusivamente vocale (ad esempio talk show, audiolibro, poesia), più il valore dell'attributo si avvicina a 1.0. Valori superiori a 0,66 descrivono brani probabilmente costituiti interamente da parole pronunciate. I valori compresi tra 0,33 e 0,66 descrivono tracce che possono contenere sia musica che parlato, in sezioni o sovrapposti, inclusi casi come la musica rap. I valori inferiori a 0,33 rappresentano molto probabilmente musica e altri brani non simili al parlato.

- **Acousticness** -> Una misura di confidenza da 0,0 a 1,0 che indica se la traccia è acustica. 1.0 rappresenta un'elevata certezza che la traccia sia acustica.
- **Instrumentalness** -> Prevede se una traccia non contiene parti vocali. I suoni "Ooh" e "aah" sono trattati come strumentali in questo contesto. Più il valore di strumentalità è vicino a 1.0, maggiore è la probabilità che la traccia non contenga contenuto vocale. I valori superiori a 0,5 intendono rappresentare brani strumentali, ma la sicurezza aumenta quando il valore si avvicina a 1,0.
- **Liveness** -> Rileva la presenza di un pubblico nella registrazione. Valori di vivacità più elevati rappresentano una maggiore probabilità che la traccia sia stata eseguita dal vivo. Un valore superiore a 0,8 fornisce una forte probabilità che la traccia sia live.
- **Valence** -> Una misura da 0,0 a 1,0 che descrive la positività musicale trasmessa da una traccia. Le tracce con alta valenza suonano più positive (ad esempio felice, allegro, euforico), mentre le tracce con bassa valenza suonano più negative (ad esempio triste, depresso, arrabbiato).
- **Tempo** -> Il tempo complessivo stimato di una traccia in battiti al minuto (BPM). Nella terminologia musicale, il tempo è la velocità o il ritmo di un dato brano e deriva direttamente dalla durata media del beat.
- **PlaylistName** (ottenuta dal file JSON) -> Nome della playlist in cui ho inserito la canzone

Preprocessing del dataset

Il preprocessing del dataset è stato utilizzato per la creazione del dataset realmente utilizzato successivamente, memorizzato nel file **newDataset.csv**. Il preprocessing si è dimostrato fondamentale per i seguenti motivi:

1) Rimozione della *playlistName* come feature. Quest'ultima è stata utilizzata solo per visualizzare come le canzoni sono distribuite attualmente nelle mie playlist**. Poiché l'obiettivo è riorganizzare le playlist secondo le sole feature delle canzoni mantenere in memoria il playlist name poteva compromettere l'algoritmo di clustering (oltre a non essere utile per gli scopi del progetto)
2) L'algoritmo di clustering utilizzato, ovvero il *KMeans*, può essere influenzato da scale diverse di valori numerici. Poiché l'algoritmo utilizza la distanza euclidea tra i punti dati per assegnarli a cluster, le caratteristiche con scale molto diverse possono avere un impatto significativo sui risultati
3) Senza preprocessing l'addestramento della rete Bayesiana portava a un eccessivo uso di memoria per il mio computer, facendo interrompere il

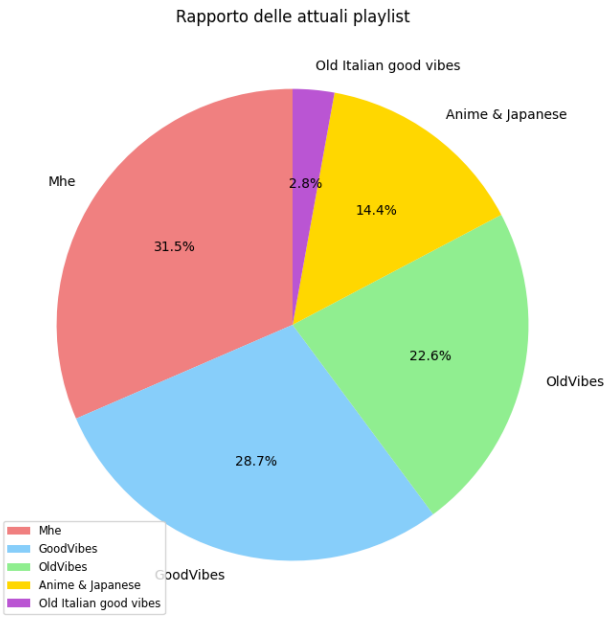
programma:

```
numpy.core._exceptions._ArrayMemoryError: Unable to allocate 83.9 GiB
```

Questo problema è stai poi risolto eseguendo la discretizzazione dei valori.

Per i punti 2) e 3) è stato utilizzato un oggetto di tipo *MinMaxScaler* per normalizzare i valori delle feature nel dataset portandoli tutti quanti in un intervallo compreso tra 0 e 1. In particolare, la normalizzazione Min-Max si ottiene sottraendo al valore di una determinata feature il valore minimo di tale feature e dividendo per la differenza tra il massimo e il minimo della determinata feature. Per la successiva discretizzazione è stato usato un oggetto di tipo *KBinsDiscretizer.*

**



Capitolo 2) Apprendimento non supervisionato

L'apprendimento non supervisionato è una branca dell'apprendimento automatico in cui l'agente viene addestrato su un insieme di dati senza etichette. Esistono due principali tipi di apprendimento non supervisionato:

- Clustering -> L'obiettivo è raggruppare gli elementi del dataset in base a delle somiglianze

- Riduzione della dimensionalità -> L'obiettivo è ridurre il numero di feature utilizzate mantenendo però le informazioni più significative. Un esempio è la *PCA* (*Principal Component Analysis*)

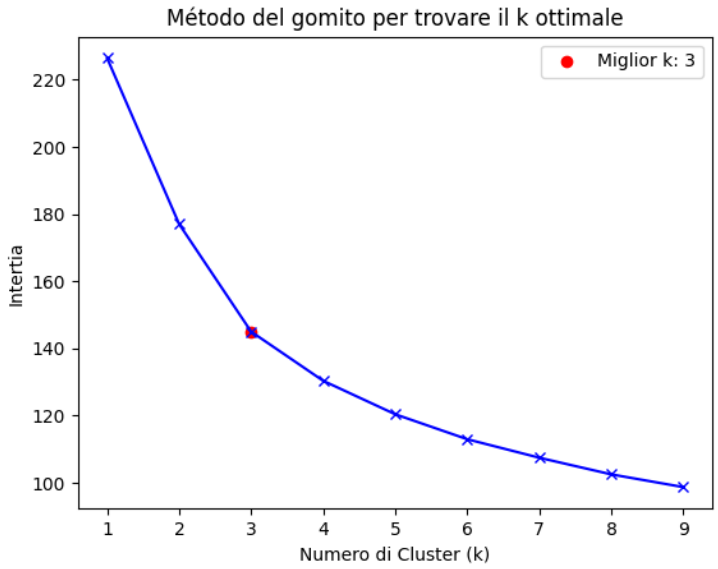
Nel mio caso l'apprendimento non supervisionato è stato utilizzato con lo scopo di effettuare clustering, per raggruppare le canzoni in dei gruppi che formeranno le mie nuove playlist. Esistono due tipologie di apprendimento di clustering: **hard clustering** e **soft clustering**. Il primo associa ogni esempio a un cluster specifico, il secondo associa a ogni esempio la probabilità di appartenenza a ogni cluster.

Per la realizzazione di questa parte è stata usata la libreria *sklearn* per il clustering, la libreria *matplotlib* per la visualizzazione di grafici e la libreria *kneed* per individuare il gomito.

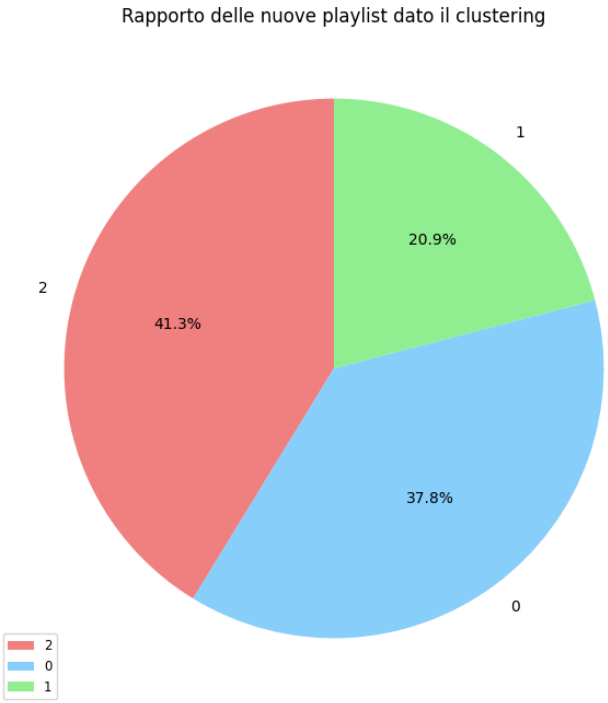
Nel mio caso ho deciso di utilizzare l'algoritmo di hard clustering *KMeans*. Uno dei problemi principali del *KMeans* è trovare il numero ideale di cluster da fornire in input all'algoritmo. Per risolvere questo problema ho seguito una strategia nota come "curva del gomito". La curva del gomito mostra la variazione dell'*inertia* al variare del numero di cluster, dove l'*inertia* rappresenta la somma delle distanze quadrate tra ogni punto dati e il centro del cluster assegnato. L'algoritmo di prevede dunque di eseguire il *KMeans* per diversi valori di numero di cluster, calcolare l'*inertia* per ognuno di essi, plottare su un grafico la curva e poi identificare il gomito, ovvero il punto in cui la diminuzione dell'*inertia* diventa meno significativa.

```
def regolaGomito(dataSet):
    intertia = []
    maxK=10
    for i in range(1, maxK):
        kmeans = KMeans(n_clusters=i, n_init=5, init='random')
        kmeans.fit(dataSet)
        intertia.append(kmeans.inertia_)
```

Per quanto riguarda le scelte progettuali ho deciso di eseguire l'algoritmo con 10 valori di cluster, che variano da 1 a 10. Per quanto riguarda il parametro *n_init* questo indica le "RANDOM_RESTART". In particolar modo per ogni numero di cluster **i** verrà eseguito l'algoritmo per 5 volte e verrà restituito il clustering migliore. Per quanto riguarda il parametro *init='random'* significa che i centroidi vengono inizializzati in maniera del tutto casuale.



Nel mio caso possiamo vedere come il numero ottimale di cluster è *k = 3*, dunque successivamente è stato eseguito l'algoritmo *KMeans* con 3 cluster. Il risultato del clustering è il seguente:



Ogni cluster viene identificato da un indice. Come possiamo vedere la distribuzione delle canzoni nei vari cluster non è perfettamente bilanciata. Ciò potrebbe influire su alcuni algoritmi di apprendimento supervisionato. Nel seguito verrà trattata la questione relativa allo sbilanciamento delle classi e alla sua possibile soluzione.

Capitolo 3) Apprendimento supervisionato

L'apprendimento supervisionato è una branca dell'apprendimento automatico in cui l'agente viene addestrato su un insieme di dati con etichette. Si divide in

- Classificazione: Le etichette rappresentano un dominio di valori discreto (quindi un insieme finito di valori) che la feature di output può assumere. Un caso particolare è la classificazione booleana, in cui le classi sono **true** e **false**.
- Regressione: L'etichetta può essere un qualsiasi valore numerico, quindi il dominio della feature di output è continuo.

L'obiettivo principale è far sì che l'algoritmo impari una relazione tra gli input e gli output in modo che, una volta addestrato, possa fare previsioni accurate su nuovi dati mai visti prima.

Ci sono diverse fasi da affrontare:

1) Scelta degli iper-parametri
2) Fase di addestramento
3) Fase di test
4) Valutazione delle prestazioni

Per questa parte di progetto sono state utilizzate le librerie: *matplotlib* e *numpy* per la visualizzazione di grafici, *sklearn* per l'apprendimento supervisionato, *pandas* e *imblearn.over_sampling* per l'oversampling del dataset.

Nel mio caso l'apprendimento supervisionato è stato utilizzato con scopo di classificazione, dove la classi sono gli indici dei cluster prima ottenuti mediante apprendimento non supervisionato.

Per questo progetto ho deciso di utilizzare tre modelli di apprendimento automatico:

- DecisionTree -> Classificatore strutturato ad albero in cui le foglie rappresentano le classi di appartenenza (o le probabilità di appartenenza a tali classi) mentre la radice e i nodi interni rappresentano delle condizioni sulle feature di input. A seconda se tali condizioni sono rispettate o meno, verrà seguito un percorso piuttosto che un altro e alla fine si arriverà alla classe di appartenenza
- RandomForest -> Classificatore che si ottiene creando tanti DecisionTree. Il valore in output si ottiene mediando sulle predizioni di ogni albero appartenente alla foresta (tecnica di bagging)

- Regressione Logistica Multinomiale (Funzione lineare) -> A differenza della classica regressione lineare, che è utilizzata per predire valori continui (regressione), la regressione logistica multinomiale viene utilizzata per calcolare le probabilità di appartenenza di un esempio a ogni classe. Per ogni classe abbiamo una funzione lineare di cui apprendere i pesi mediante tecnica di discesa di gradiente. In ogni funzione vi è un peso per ogni feature più un termine noto w_0 (a cui associamo una feature immaginaria X_0 con valore costante 1). Il risultato viene "schiacciato" da una funzione nota come **softmax**, la quale trasforma un vettore di valori reali in una distribuzione di probabilità su più classi. Il vantaggio è dunque poter apprendere n-1 funzioni (date n le classi) e ottenere la probabilità dell'ultima classe come 1- ∑ delle altre probabilità apprese. Il risultato è la categoria con la probabilità maggiore

Fase 1) Scelta degli iper-parametri

Gli iper-parametri sono i parametri di un modello di apprendimento automatico, i quali non vengono appresi durante la fase di addestramento come i normali parametri del modello (es. i pesi di una funzione lineare) ma devono essere necessariamente fissati prima che il modello possa cominciare l'addestramento. La loro scelta influisce sulle prestazioni e sulla complessità del modello. Uno dei compiti più complessi è proprio la scelta degli iper-parametri per i vari modelli.

Per la scelta degli iper-parametri ho utilizzato una tecnica di *K-Fold Cross Validation* (CV).

Nella *K-Fold CV* il dataset viene diviso in *k* fold (insiemi disgiunti) e il modello viene addestrato *k* volte. Per ogni iterazione 1 fold viene usato per il testing mentre gli altri *k-1* fold vengono utilizzati per il training. In questo modo è possibile testare e addestrare il modello su dati diversi per comprendere "la bontà" del modello.

La strategia che ho deciso di applicare per ricercare gli iper-parametri dei miei modelli è la *GridSearch* con *Cross Validation*. In questo approccio vengono definite le griglie dei valori possibili per gli iper-parametri e si esplorano tutte le combinazioni possibili alla ricerca della miglior combinazione possibile.

IPER-PARAMETRI RICERCATI e UTILIZZATI

DecisionTree

- Criterion -> Misura la qualità dello spit effettuato sui nodi. Può assumere i seguenti valori:

    - gini -> Misura la “purezza” della divisione dei dati, più precisamente quanto spesso un elemento viene classificato in modo sbagliato
    - entropy -> Misura la quantità di disordine nei dati. Minimizzare l’entropia significa massimizzare l’informazione guadagnata durante
    - log_loss -> perdita logaritmica. Indicata quando l’output corrisponde a una probabilità piuttosto che a un valore di classe.
- Splitter -> Indica la strategia da utilizzare per il criterio di split. In questo caso ho usato il valore di default **best** che indica il miglior criterio di split possibile, e dunque non l’ho ricercato ma l’ho direttamente utilizzato.
- Max_depth -> Indica l’altezza massima dell’albero
- Min_samples_split -> Il numero minimo di esempi necessari affinché possa essere inserito un criterio di split. Se il numero è minore viene innestata una foglia
- min_samples_leaf -> Il numero minimo di esempi per poter creare una foglia

```
DecisionTreeHyperparameters = {
    'DecisionTree__criterion': ['gini', 'entropy','log_loss'],
    'DecisionTree__max_depth': [None, 5, 10],
    'DecisionTree__min_samples_split': [2, 5, 10, 20],
    'DecisionTree__min_samples_leaf': [1, 2, 5, 10, 20],
    'DecisionTree__splitter': ['best']}
```

RandomForest

Per quanto riguarda la creazione dei singoli alberi ho utilizzato gli stessi criteri del DecisionTree (meno splitter che non è un iperparametro per questo modello). Per quanto riguarda i criteri relativi alla foresta abbiamo:

- n_estimators: il numero di alberi nella foresta

```
RandomForestHyperparameters = {
    'RandomForest__criterion': ['gini', 'entropy','log_loss'],
    'RandomForest__n_estimators': [10, 20,50],
    'RandomForest__max_depth': [None, 5, 10],
    'RandomForest__min_samples_split': [2, 5, 10, 20],
    'RandomForest__min_samples_leaf': [1, 2, 5, 10, 20]}
```

LogisticRegression (Multinominal)

- Penalty -> Stabilisce la penalizzazione applicata al modello, per ridurne la complessità e cercare di evitare *overfitting*
    - l1 -> Viene applicata la norma 1 come termine di penalizzazione (ovvero al somma dei valori assoluti dei pesi). Viene anche detta regolarizzazione lasso. Tende a ridurre a 0 i pesi delle feature non importanti. In questo caso non è stata utilizzata in quanto la *lbfgs* non la supporta

```
ValueError: Solver lbfgs supports only 'l2' or 'none' penalties, got l1 penalty.
```

    - l2 -> Viene applicata la norma 2 come termine di penalizzazione (ovvero la somma dei quadrati dei pesi). Favorisce pesi più bilanciati, utile per prevenire *overfitting*.
- C -> Specifica quanto "forte" è la regolarizzazione. Più il valore di C è basso e più forte è la regolarizzazione. Valori di C piccoli prevengono *overfitting* ma rendono il modello più rigido. Valori alti rendono il modello più flessibile ma sono soggetti a *overfitting*
- Max_iter -> Indica il numero di iterazioni utilizzate
- Solver -> Indica l'algoritmo da utilizzare per apprendere i pesi
    - lbfgs -> mantiene solo alcune informazioni delle derivate parziali, utilizzando dunque poca memoria. È adatto per problemi con numero di feature elevato, soprattutto quando la memoria a disposizione non è molta
    - liblinear -> Basato su algoritmi di programmazione lineare

```
LogisticRegressionHyperparameters = {
    'LogisticRegression__C': [0.001, 0.01, 0.1, 1, 10, 100],
    'LogisticRegression__penalty': ['l2'],
    'LogisticRegression__solver': ['liblinear', 'lbfgs'],
    'LogisticRegression__max_iter': [100000,150000]}
```

Nel mio caso ogni modello è stato addestrato con un valore di *K = 5 (5 fold, 5 iterazioni)*

```
gridSearchCV_dtc = GridSearchCV(Pipeline([('DecisionTree', dtc)]), DecisionTreeHyperparameters, cv=5)
gridSearchCV_rfc = GridSearchCV(Pipeline([('RandomForest', rfc)]), RandomForestHyperparameters, cv=5)
gridSearchCV_reg = GridSearchCV(Pipeline([('LogisticRegression', reg)]), LogisticRegressionHyperparameters, cv=5)
```

Nel mio caso i parametri che sono stati restituiti sono:

Modello	Parametro	Valore
Decision Tree	criterion	entropy
Decision Tree	max_depth	10
Decision Tree	min_samples_split	10
Decision Tree	min_samples_leaf	2
Random Forest	n_estimators	20
Random Forest	max_depth	5
Random Forest	min_samples_split	10
Random Forest	min_samples_leaf	2
Random Forest	criterion	log_loss
Logistic Regression	C	100
Logistic Regression	penalty	l2
Logistic Regression	solver	lbfgs
Logistic Regression	max_iter	100000

Fasi 2) e 3) Fase di addestramento e test

Per la fase di addestramento e di test i modelli sono stati addestrati utilizzando una *K-Fold Cross Validation* con *K = 5*

```
cv = RepeatedKFold(n_splits=5, n_repeats=5)
scoring_metrics = ['accuracy', 'precision_macro', 'recall_macro', 'f1_macro']
results_dtc = {}
results_rfc = {}
results_reg = {}
for metric in scoring_metrics:
    scores_dtc = cross_val_score(dtc, X, y, scoring=metric, cv=cv)
    scores_rfc = cross_val_score(rfc, X, y, scoring=metric, cv=cv)
    scores_reg = cross_val_score(reg, X, y, scoring=metric, cv=cv)
    results_dtc[metric] = scores_dtc
    results_rfc[metric] = scores_rfc
    results_reg[metric] = scores_reg
```

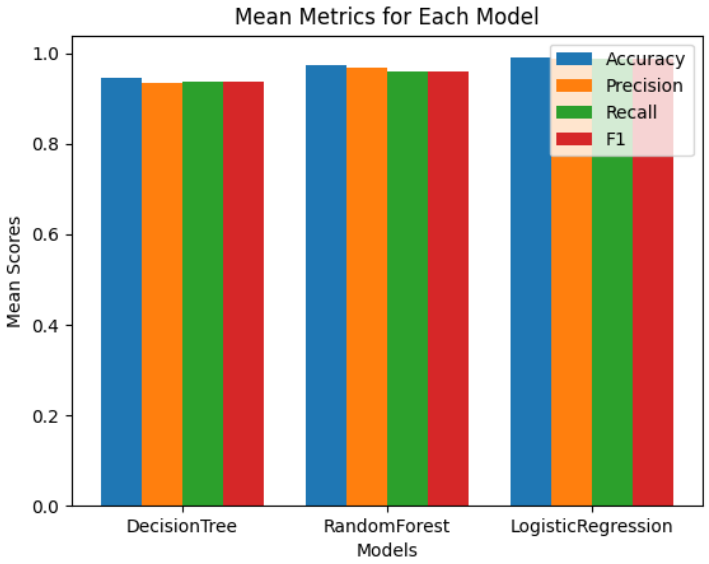
Fase 4) Valutazione delle prestazioni

La K-Fold Cross Validation, come già detto precedentemente, permette di addestrare il modello diverse volte utilizzando ogni volta esempi di training e di test diversi, in modo tale da ottenere delle valutazione più “veritiere” rispetto alle reali performance del sistema.

Per la valutazione delle performance del sistema ho deciso di utilizzare le seguenti metriche:

- Accuracy -> L'accuracy è una misura generale della correttezza del modello e rappresenta la frazione di previsioni corrette rispetto al totale delle previsioni.
- Precision_Macro -> La precision macro è la media delle precisioni calcolate per ogni classe. La precisione per ogni classe è il numero di istanze di classe **c** classificate nella classe **c** / numero di istanze classificate nella classe **c**
- Recall_Macro -> La recall macro è la media delle recall calcolate per ogni classe. La recall per ogni classe è il numero di istanze di classe **c** classificate nella classe **c** / numero di istanze nella classe **c**
- F1_Macro -> La F1 macro è la media delle F1 calcolate per ogni classe. La F1 calcolata per ogni classe è la media armonica tra precision e recall (dunque è alta solo se entrambi i valori sono alti) ed è calcolata come segue: 2*Precision*Recall/(Precision+Recall)

RISULTATI



Possiamo notare che in generale i valori sono molto buoni per ogni metrica in ogni modello. Questo potrebbe rappresentare un segnale di *overfitting*. Per verificare la

presenza di *overfitting* bisogna prendere in considerazione altri fattori, come la varianza, la deviazione standard e le curve di apprendimento.

L'*overfitting* è un problema molto noto in apprendimento automatico. Si verifica quando un modello si sovra-adatta ai dati di addestramento, imparando correlazioni spurie presenti nel dataset e non riuscendo a generalizzare bene (e dunque il modello non sarà in grado di predire correttamente i valori per esempi mai visti che non presentano queste correlazioni).

Varianza e deviazione standard delle curve di apprendimento

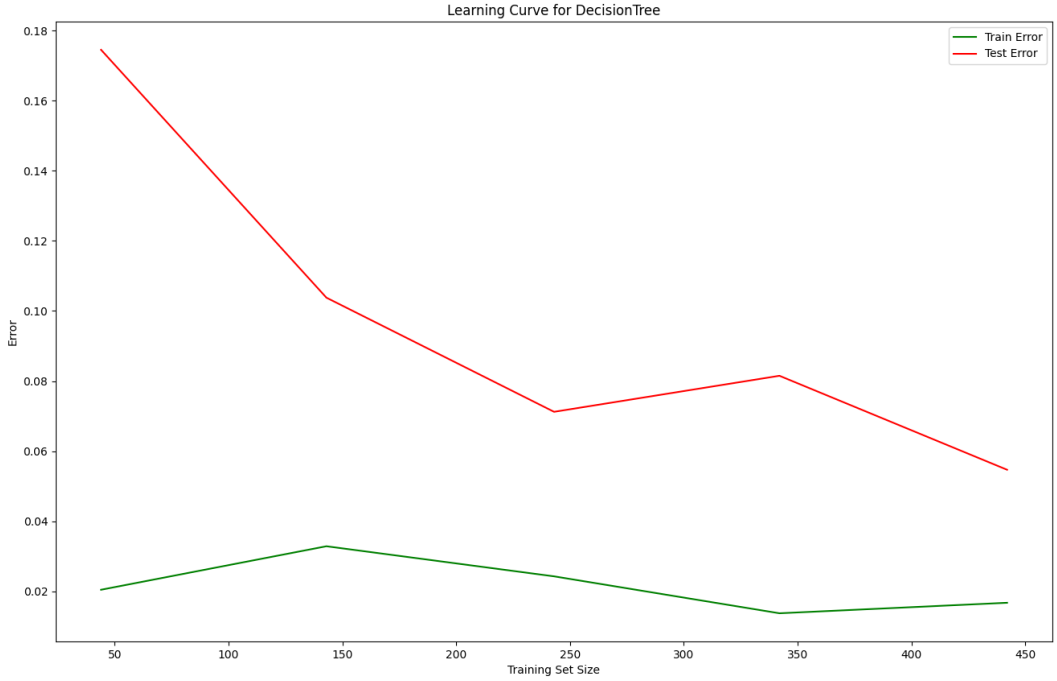
Varianza -> È la media dei quadrati delle differenze tra ciascun dato e la media

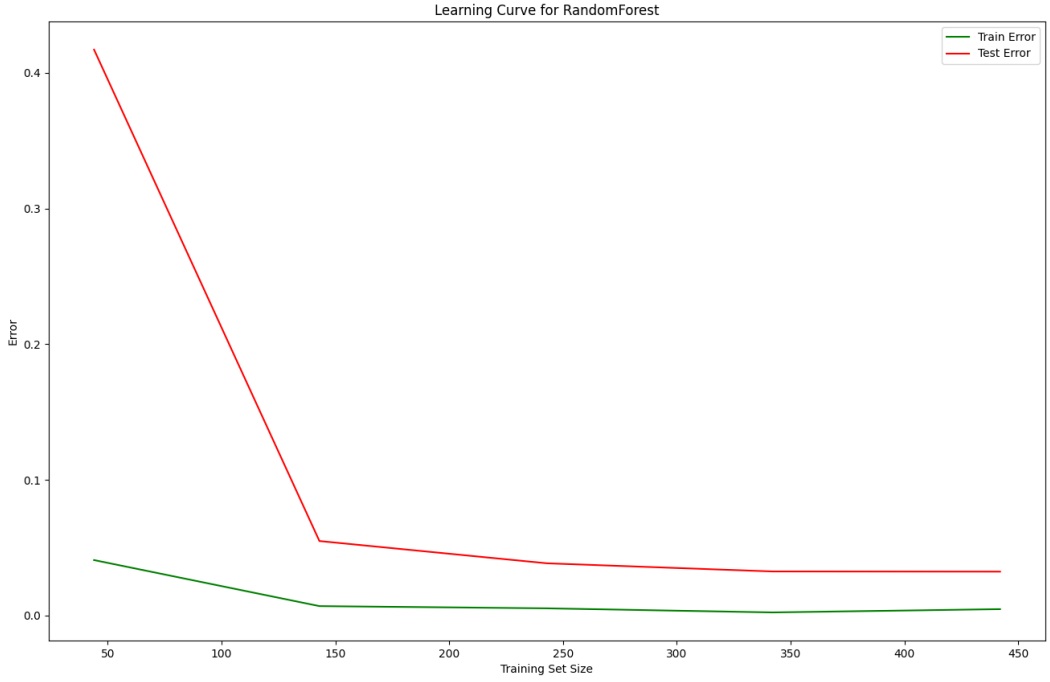
Deviazione Standard -> Radice quadrata della varianza

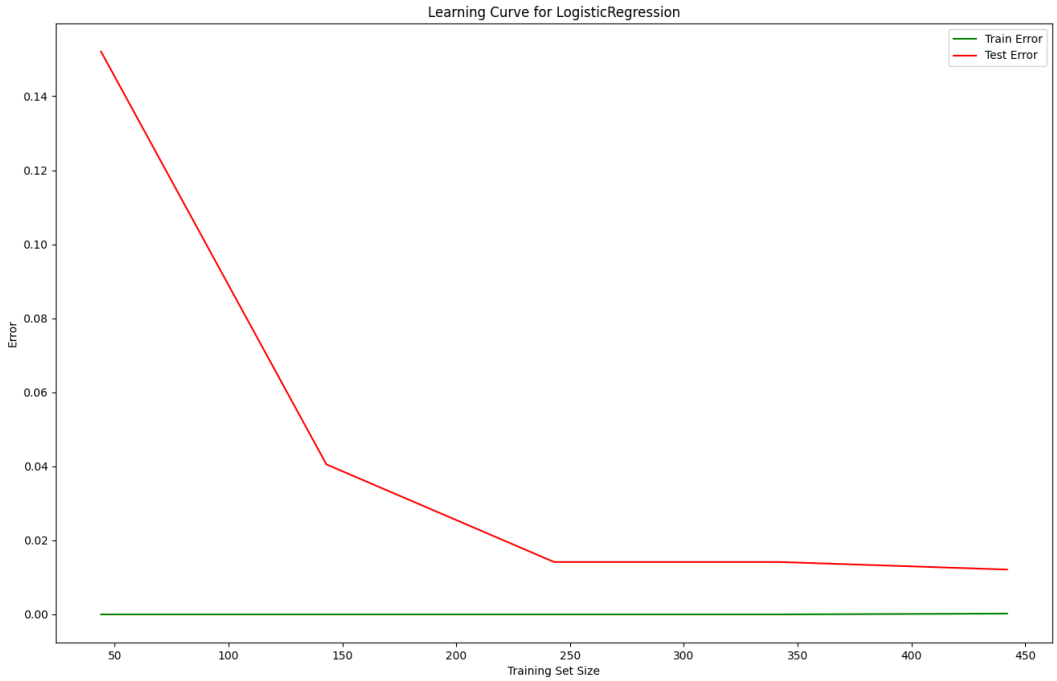
Nel contesto dell'apprendimento automatico misurano entrambe la dispersione dell'errore. Alti valori suggeriscono che il modello è sensibile alle variazioni dei dati in input, e dunque suggeriscono una possibile presenza di *overfitting*.

Modello	Tipo di Errore	Deviazione Standard	Varianza
Decision Tree	Train	0.0036760354772108522	1.3513e-05
Decision Tree	Test	0.034823766461100594	0.0012127
RandomForest	Train	0.0032707765372852735	1.0698e-05
RandomForest	Test	0.032750466961771356	0.0010726
Logistic Regression	Train	0.0006787330316742056	4.6068e-07
Logistic Regression	Test	0.0183846177304 89	0.000338

Curve di apprendimento







Analisi dei risultati per ogni modello

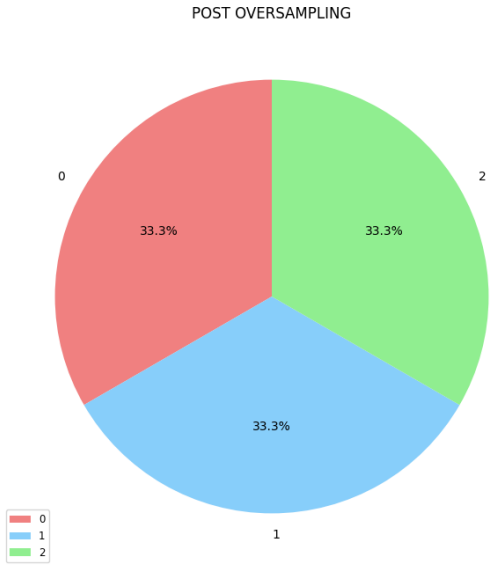
- **Decision Tree:** I valori di deviazione standard e varianza sono relativamente basse. Tuttavia, la varianza del test errore è molto più alta rispetto a quello del train error, ciò potrebbe suggerire *overfitting*. Osservando la curva di apprendimento è possibile notare un piccolo aumento dell'errore sul test dai 250 ai 350 esempi circa. Ciò potrebbe essere sintomo di *overfitting*. Tutto ciò suggerisce che il Decision Tree potrebbe soffrire di *overfitting.*
- **Random Forest:** Anche in questo caso i valori di deviazione standard e varianza solo relativamente bassi, e in particolar modo la differenza tra le due è ancora più marcato rispetto al Decision Tree. La curva parte con un errore molto più elevato ma questo diminuisce molto rapidamente avvicinandosi a quello di training. Tuttavia, la forte differenza tra le varianze suggerisce *overfitting.*
- **Logistic Regression:** I valori di varianza e deviazione sono molto bassi, e la distanza tra le due varianze in questo caso è molto ridotta. Troviamo in generale un comportamento nelle curve molto simile a quello della Random Forest. Non sembra esserci *overfitting.*

POSSIBILE SOLUZIONE

L'*overfitting* potrebbe dipendere da uno sbilanciamento delle classi. Una possibile soluzione è effettuare over-sampling. In questo caso ho individuato la classe con più esempi e ho generato esempi sintetici per le altre classi, facendo in modo che le classi

fossero più equilibrate. Per fare ciò ho utilizzato una tecnica nota come *SMOTE*. La tecnica *SMOTE* per ogni esempio nelle classi di minoranze identifica i suoi *k* vicini più prossimi e, mediante combinazione lineari tra questi ultimi e l'esempio stesso, genera nuovi dati sintetici.

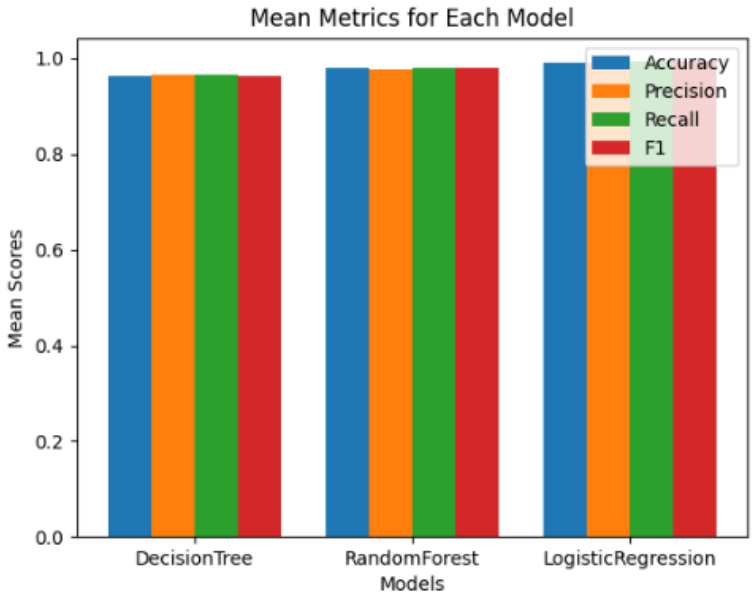
Vengono ora riportati i risultati:



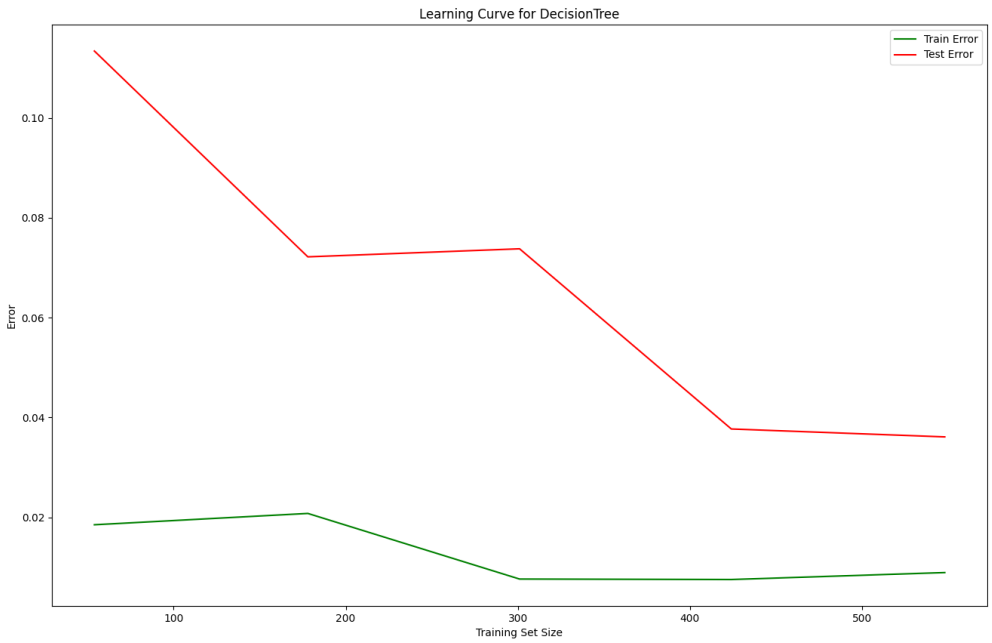
Bilanciamento delle classi dopo l'*oversampling*

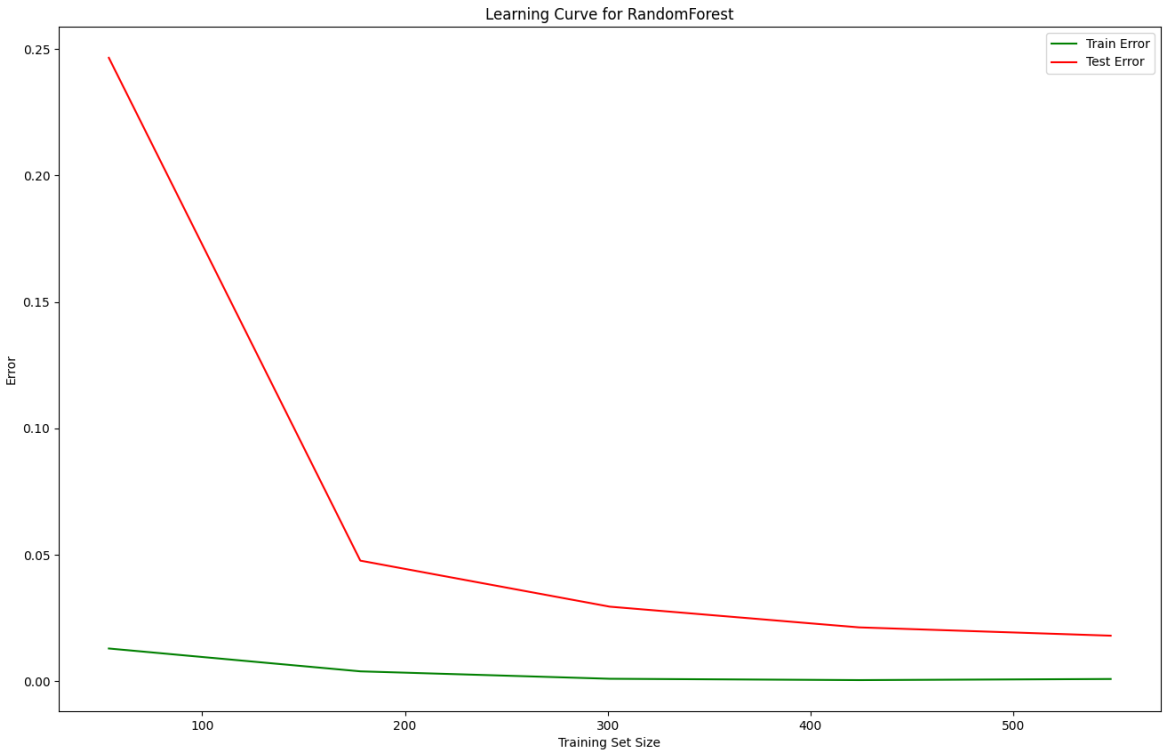
Modello	Parametro	Valore
Decision Tree	criterion	entropy
Decision Tree	max_depth	10
Decision Tree	min_samples_split	2
Decision Tree	min_samples_leaf	2
Random Forest	n_estimators	50
Random Forest	max_depth	10
Random Forest	min_samples_split	10
Random Forest	min_samples_leaf	1
Random Forest	criterion	entropy
Logistic Regression	C	0.1
Logistic Regression	penalty	l2
Logistic Regression	solver	lbfgs
Logistic Regression	max_iter	100000

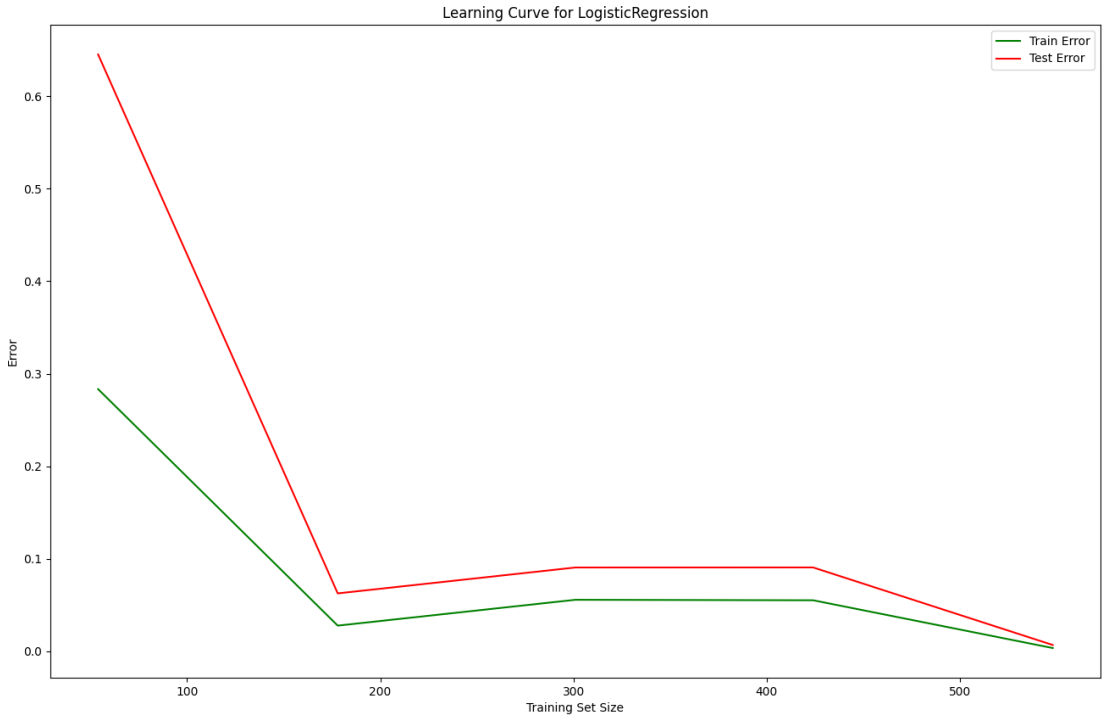
Possiamo notare dei cambiamenti rispetto a prima nella scelta degli iperparametri



Modello	Tipo di Errore	Deviazione Standard	Varianza
Decision Tree	Train	0.0020725942867884303	4.2956e-06
Decision Tree	Test	0.029135933565658266	0.0008489
RandomForest	Train	0.0016823986235935950	2.8305e-06
RandomForest	Test	0.015465542839437070	0.0002392
Logistic Regression	Train	0.0019653886157425370	3.8628e-06
Logistic Regression	Test	0.0080649240042151260	6.5043e-05




Learning Curve for RandomForest
Train Error
Test Error
Error
0.25
0.20
0.15
0.10
0.05
0.00
100
200
300
400
500
Training Set Size


Learning Curve for LogisticRegression
Train Error
Test Error
Error
0.6
0.5
0.4
0.3
0.2
0.1
0.0
100
200
300
400
500
Training Set Size

In generale possiamo notare come per tutti i modelli i valori di deviazione standard e varianza si sono ridotti, e soprattutto si è ridotta anche la distanza tra distanza tra le varianze di test e di error. Anche le curve di apprendimento suggeriscono un miglioramento in tutti i casi.

Conclusione: L'*oversampling* sembra aver risolto l'*overfitting* e migliorato i diversi modelli

Capitolo 4) Ragionamento probabilistico e Bayesian Network

Il ragionamento probabilistico è una forma di ragionamento che sfrutta la teoria della probabilità, in particolar modo dipendenza e indipendenza tra variabili e regola di Bayes. Nel ragionamento probabilistico si assegnano probabilità a ipotesi ed eventi e si utilizzano le probabilità a posteriori per i ragionamenti. Un'applicazione del ragionamento probabilistico sono le reti bayesiane. Queste vengono rappresentate mediante grafi orientati aciclici (DAG) dove ogni nodo del grafo rappresenta una variabile e gli archi indicano le dipendenze probabilistiche tra le variabili.

Per questa parte di progetto sono state utilizzate le librerie *networkx* e *matplotlib* per la visualizzazione di grafo e grafici e *pgmpy* per gestire la rete bayesiana.

PRIMO APPROCCIO (fallimentare): Valori continui

STRUTTURA RETE BAYESIANA

Per poter utilizzare una rete bayesiana vi è il bisogno di identificare le variabili e le dipendenze tra esse. Nel mio caso le variabili sono le features delle canzoni e l'indice del cluster. Il problema è dunque l'apprendimento della struttura.

Inizialmente ho provato utilizzare una tecnica nota come *HillClimbSearch* per apprendere la struttura della rete. Questo è un algoritmo di ricerca locale che cerca di massimizzare la funzione di verosimiglianza del dataset fornito. Ho però riscontato molti problemi con questa tecnica.

```
'''hc_k2=HillClimbSearch(dataSet)
k2_model=hc_k2.estimate(max_iter=4)'''
```

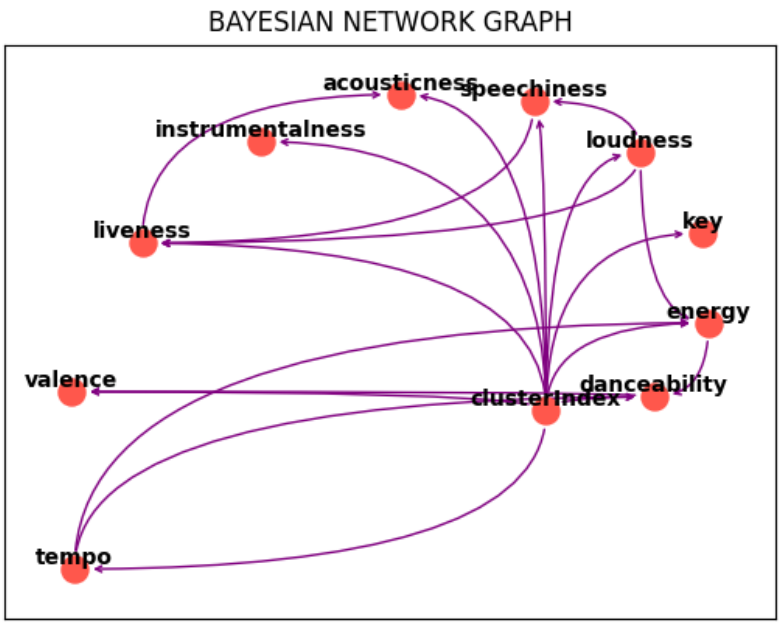
In particolar modo con un numero di iterazioni dell'algoritmo >=4 questo richiedeva troppa memoria (decine di GiB o anche TiB) oppure portava al crash del mio dispositivo. Ho provato allora a eseguire l'algoritmo per l'apprendimento su delle piattaforme online (come Google Colab) ma anche in questo caso le risorse non erano sufficienti.

```
7 frames
/usr/local/lib/python3.10/dist-packages/joblib/parallel.py in _return_or_raise(self)
    752         try:
    753             if self.status == TASK_ERROR:
--> 754                 raise self._result
    755             return self._result
    756         finally:

TerminatedWorkerError: A worker process managed by the executor was unexpectedly terminated. This could be caused by a segmentation fault while calling the function or by an excessive memory usage causing the Operating System to kill the worker.

The exit codes of the workers are {SIGKILL(-9)}
```

Ho deciso allora di fornire una struttura arbitraria alla rete bayesiana. In particolar modo ho reso dipendenti tutte le feature di input dalla feature di output e, sfruttando la descrizione delle metriche fornita dalla documentazione di Spotify, ho cercato di capire quali fossero le correlazioni tra le variabili.



Questi sono esempi di probabilità che la rete è riuscita ad apprendere:

```
CPD per la variabile 'clusterIndex':
+-----------------+----------+
| clusterIndex(0) | 0.182692 |
+-----------------+----------+
| clusterIndex(1) | 0.387019 |
+-----------------+----------+
| clusterIndex(2) | 0.430288 |
+-----------------+----------+
=======================================

CPD per la variabile 'danceability':
+-----------------------------------+-----+----------------------------+
| clusterIndex                      | ... | clusterIndex(2)            |
+-----------------------------------+-----+----------------------------+
| energy                            | ... | energy(1.0000000000000002) |
+-----------------------------------+-----+----------------------------+
| tempo                             | ... | tempo(1.0)                 |
+-----------------------------------+-----+----------------------------+
| danceability(0.0)                 | ... | 0.0034482758620689655      |
+-----------------------------------+-----+----------------------------+
| danceability(0.02145922746781115) | ... | 0.0034482758620689655      |
+-----------------------------------+-----+----------------------------+
```

Come possiamo vedere clusterIndex è una variabile indipendente, dunque il suo valore non dipende dalle altre variabili.

Danceability, invece, dipende da variabili quali *clusterIndex*, *energy* e *tempo*. In questo caso stiamo dicendo che P(danceability=0.02145922746781115 | clusterIndex=2, energy=1, tempo=1) = 0.0034482758620689755. Questo significa che, dati gli esempi nel dataset, si è appreso che, se *clusterIndex=2, energy=1 e tempo=1*, allora *danceability* ha circa lo 0.3% di probabilità di assumere quel valore.

La rete bayesiana permette di effettuare un semplice task di classificazione e molto altro:

- **Generazione di sample**: Vengono sfruttate le probabilità apprese dal dataset in fase di addestramento per generare un esempio sintetico credibile
- **Gestione di dati mancanti**: Le reti bayesiane sono in grado di gestire casi in cui per una variabile casuale non sia stato osservato alcun valore

ESEMPI

```
ESEMPIO RANDOMICO GENERATO --->   [[ 5.15722609e-01  6.98716738e-01  9.92930941e-01 -1.90913636e+01
   3.26577985e-02  3.01951708e-01  8.34214286e-01  2.51195929e-02
   4.61443114e-01  1.84752496e+02]]
PREDIZIONE DEL SAMPLE RANDOM
+-----------------+---------------------+
| clusterIndex    |   phi(clusterIndex) |
+=================+=====================+
| clusterIndex(0) |              1.0000 |
+-----------------+---------------------+
| clusterIndex(1) |              0.0000 |
+-----------------+---------------------+
| clusterIndex(2) |              0.0000 |
+-----------------+---------------------+
```

In questo caso, date le probabilità apprese dal dataset, è stato generato questo esempio randomico a cui poi ho fatto calcolare il valore del cluster di appartenenza.

```
ESEMPIO RANDOMICO SENZA ENERGY ---->  [ 5.15722609e-01  9.92930941e-01 -1.90913636e+01  3.26577985e-02
  3.01951708e-01  8.34214286e-01  2.51195929e-02  4.61443114e-01
  1.84752496e+02]
PREDIZIONE DEL SAMPLE RANDOM SENZA ENERGY
+-----------------+---------------------+
| clusterIndex    |   phi(clusterIndex) |
+=================+=====================+
| clusterIndex(0) |              1.0000 |
+-----------------+---------------------+
| clusterIndex(1) |              0.0000 |
+-----------------+---------------------+
| clusterIndex(2) |              0.0000 |
+-----------------+---------------------+
```

Possiamo notare come, dopo aver rimosso la feature “energy” (nell’ordine, la seconda) la rete bayesiana sia ancora in grado di effettuare calcoli di probabilità.

Problema delle reti bayesiane con valori continui

Le reti Bayesiane, rispetto agli altri modelli di apprendimento, presentano un problema nel caso in cui i valori delle feature sono continui e non discreti. Infatti, per creare le *CPD*, è richiesta l’osservazione di ogni singolo valore. Se ciò non avviene la rete bayesiana restituisce degli errori

Esempio:

```
PREDIZIONE DELLA CANZONE: you broke me first DI Tate McRae
Traceback (most recent call last):
  File "C:\Users\fonta\Desktop\ICON\ICON\Spotipy\main.py", line 86, in <module>
    prediciCluster(bayesianNetwork, esempio, "clusterIndex")
  File "C:\Users\fonta\Desktop\ICON\ICON\Spotipy\bayesianNetwork.py", line 78, in prediciCluster
    result = inference.query(variables=[differentialColumn], evidence=example)
             ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
  File "C:\Users\fonta\Desktop\ICON\ICON\ProvaGPT\Lib\site-packages\pgmpy\inference\ExactInference.py", line 338, in query
    indexer[index] = phi.get_state_no(
                     ^^^^^^^^^^^^^^^^^
  File "C:\Users\fonta\Desktop\ICON\ICON\ProvaGPT\Lib\site-packages\pgmpy\utils\state_name.py", line 72, in get_state_no
    return self.name_to_no[var][state_name]
           ~~~~~~~~~~~~~~~~~~~~^^^^^^^^^^^^
KeyError: 0.45694482515565654
```

In questo caso ho provato a predire il cluster per questa canzone che non appartiene al mio dataset e il programma ha generato un errore in quanto uno dei valori delle metriche non era mai stato osservato fino a quel momento. Questo è un problema tipico delle reti bayesiane, in quanto per costruire le *CPD* le reti necessitano di aver osservato tutti i valori possibili (nel mio caso per ogni metrica tutti i possibili valori compresi tra 0 e 1 generati dal *MinMaxScaler*).

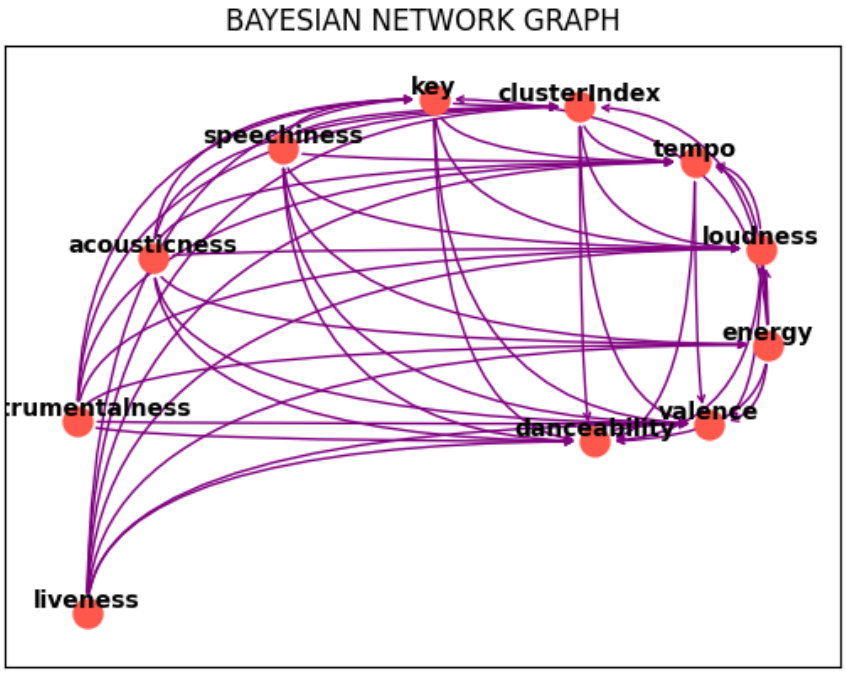
```
PREDIZIONE DELLA CANZONE: Stressed Out DI Twenty One Pilots
+------------------+----------------------+
| clusterIndex     |   phi(clusterIndex) |
+==================+======================+
| clusterIndex(0) |               1.0000 |
+------------------+----------------------+
| clusterIndex(1) |               0.0000 |
+------------------+----------------------+
| clusterIndex(2) |               0.0000 |
+------------------+----------------------+
```

In questo caso ho provato a predire il cluster per una canzone già appartenente al dataset (sulla quale dunque il modello è stato allenato) e infatti non risultano esserci problemi.

SECONDO APPROCCIO: Valori discreti

Per risolvere questo problema ho deciso di discretizzare i valori continui mediante *KBinsDiscretizer* (presente nella libreria *scikit_learn*) il quale permette di trasformare i valori continui in discreti.

L'utilizzo dei valori discreti, oltre a risolvere i problemi dei valori continui, mi ha permesso anche di risolvere i problemi relativamente all'utilizzo della memoria, che prima riscontravo.



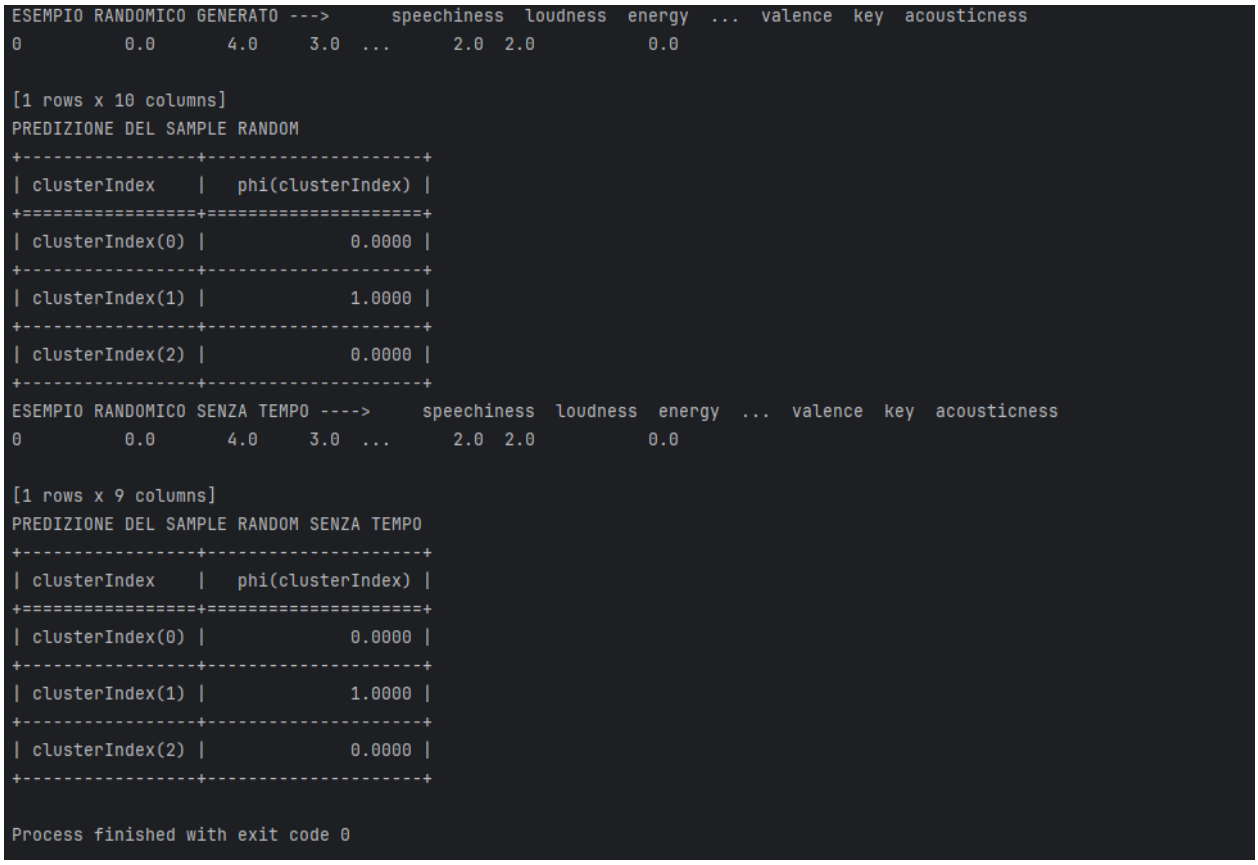
Adesso è stato reso possibile l'utilizzo del metodo *HillClimbSearch* per apprendere la struttura della rete. Possiamo vedere come la struttura è molto più complessa rispetto a quella da me ipotizzata.

```
CPD per la variabile 'danceability':
+------------------+-----+----------------------+
| acousticness     | ... | acousticness(4.0)    |
+------------------+-----+----------------------+
| clusterIndex     | ... | clusterIndex(2)      |
+------------------+-----+----------------------+
| energy           | ... | energy(4.0)          |
+------------------+-----+----------------------+
| instrumentalness | ... | instrumentalness(4.0) |
+------------------+-----+----------------------+
| key              | ... | key(4.0)             |
+------------------+-----+----------------------+
| liveness         | ... | liveness(4.0)        |
+------------------+-----+----------------------+
| loudness         | ... | loudness(4.0)        |
+------------------+-----+----------------------+
| speechiness      | ... | speechiness(4.0)     |
+------------------+-----+----------------------+
| tempo            | ... | tempo(4.0)           |
+------------------+-----+----------------------+
| danceability(0.0) | ... | 0.2                 |
+------------------+-----+----------------------+
| danceability(1.0) | ... | 0.2                 |
+------------------+-----+----------------------+
| danceability(2.0) | ... | 0.2                 |
+------------------+-----+----------------------+
| danceability(3.0) | ... | 0.2                 |
```

Ovviamente anche le *CPD* sono cambiate:

```
PREDIZIONE DELLA CANZONE: Sofia DI Clairo
+-----------------+----------------------+
| clusterIndex    |   phi(clusterIndex) |
+=================+======================+
| clusterIndex(0) |               0.3333 |
+-----------------+----------------------+
| clusterIndex(1) |               0.3333 |
+-----------------+----------------------+
| clusterIndex(2) |               0.3333 |
+-----------------+----------------------+
```

Adesso la rete è riuscita a predire il cluster su canzoni non appartenenti al dataset:



Ovviamente l'esempio randomico viene generato già discretizzato.

Capitolo 5) Ragionamento logico e Prolog

Il ragionamento logico si differisce da quello probabilistico in quanto in questo caso il ragionamento avviene usando la logica matematica. Viene costruita una cosiddetta *Knowledge Base*, contenente gli assiomi (ovvero affermazioni sempre vere). Gli assiomi si dividono a loro volta in regole e fatti. I fatti sono verità immutabili, una regola è una dichiarazione che afferma che qualcosa è vero in base ad altre cose che sono vere.

Ho utilizzato la libreria *pyswip* per il *Prolog*.

Prolog è un linguaggio di programmazione dichiarativo basato sul ragionamento logico. Nel mio caso sono stati definiti 2 tipi di fatti e 1 regola

- Fatto song: contiene le informazioni relative a una canzone
- Fatto clustered_song: contiene le informazioni relative a una canzone e al suo cluster di appartenenza

- Regola: canzoni_info(NomeCanzone, Autore, Cluster) :- (song(A1, B1, C1, D1, E1, F1, G1, H1, I1, L1, Autore, NomeCanzone), clustered_song(A2, B2, C2, D2, E2, F2, G2, H2, I2, L2, Cluster)), A1= A2, B1 = B2, C1 = C2, D1 = D2, E1 = E2, F1 = F2, G1 = G2, H1 = H2, I1 = I2, L1 = L2.

Questa regola può essere utilizzata per diversi scopi. Un esempio di applicazione da me fatto è stato quello di ricavare il cluster in cui ogni canzone è stato inserito:

```
Cluster:  1.0
Sunflower Rex Orange County
Let Me Down Slowly Alec Benjamin
you were good to me (bonus track) Jeremy Zucker
Best Part of Me (feat. YEBBA) Ed Sheeran
Let Her Go Passenger
Lucid Dreams Juice WRLD
July Noah Cyrus
prom dress mxmtoon
```

Questo è una piccola parte dell'output restituito.

Potrebbe essere usata anche per altri scopi (esempio: dato il nome di una canzone vedere chi è l'autore e capire in quale cluster è stata inserita).

Sviluppi futuri

Il progetto da un punto di vista funzionale potrebbe completare le funzionalità relative a Spotify per creare dunque sul profilo dell'utente le nuove playlist. Si potrebbe dare la possibilità all'utente di interfacciarsi con il sistema in modo tale da classificare e inserire automaticamente le canzoni nelle nuove playlist. Il tutto potrebbe essere eseguito con un'interfaccia grafica

Da un punto di vista progettuale si potrebbe applicare la *PCA* o altre tecniche di *feature selection* per provare a ridurre la dimensionalità del problema.

Riferimenti bibliografici

Ragionamento logico: D. Poole, A. Mackworth: Artificial Intelligence: Foundations of Computational Agents. 3/e. Cambridge University Press [Ch.5]

Prolog:] D. Poole, A. Mackworth: Artificial Intelligence: Foundations of Computational Agents. 3e, Cambridge University Press [Ch.15]

Apprendimento supervisionato: D. Poole, A. Mackworth: Artificial Intelligence: Foundations of Computational Agents. Cambridge University Press. 3rd Ed. [Ch.7]
Apprendimento non supervisionato: D. Poole, A. Mackworth: Artificial Intelligence: Foundations of Computational Agents. Cambridge University Press. 3rd Ed. [Ch.10]
Ragionamento probabilistico e reti bayesiane: D. Poole, A. Mackworth: Artificial Intelligence: Foundations of Computational Agents. Cambridge University Press. 3/e. [Ch.9]

Descrizione metriche del dataset: https://developer.spotify.com/documentation/web-api/reference/get-audio-features