



Progetto ICON

DI-CA 2021/2022

Capotorto Lorena	l.capotorto@studenti.uniba.it	696947
Dimatteo Roberto	r.dimatteo6@studenti.uniba.it	707120

SOMMARIO

1	Introduzione.....	2
1.1	Dataset utilizzati e Librerie	2
2	Recommender System	3
2.1	Scelta delle metriche per Recommender	3
2.2	Realizzazione.....	6
2.3	Classificazione	8
3	Knowledge Base	13
3.1	Gestione delle Knowledge Base.....	13
3.1.1	Fatti	13
3.1.2	Regole	14
3.1.3	Liking Probability.....	15
3.2	Interazione con l'Utente	16
4	Ontologia.....	19
4.1	Protégé.....	19
4.2	Owlready2.....	23
5	Conclusioni.....	26

1 INTRODUZIONE

Steam è una piattaforma digitale della Valve Corporation che si occupa della distribuzione di una grande varietà di videogiochi, sia sviluppati dalla Valve stessa che da altre case produttrici. È una delle più grandi piattaforme di distribuzione digitale e al 2021 offre servizi ad oltre 69milioni di utenti giornalieri.

Visto questo uso elevato, e in quanto noi stessi giocatori che utilizzano spesso la piattaforma, abbiamo deciso di dedicarci, in questo caso di studio, alla creazione di un sistema che possa dare supporto ai giocatori nelle loro scelte e nella ricerca di nuovi giochi.

A tale scopo, abbiamo realizzato:

- un **recommender system** capace di consigliare nuovi giochi sulla base delle preferenze personali dell'utente, poiché tra le tante scelte di giochi disponibili è difficile sapere di preciso quella migliore;
- un **modello di classificazione** con **predizioni** sulla categoria dei ratings dei giochi da parte degli utenti online;
- una **knowledge base** che permette all'utente di interrogare il dominio di interesse tramite un'interazione basata sulla forma 'domanda-risposta', per determinare cosa sia vero e permettere all'utente di acquisire informazioni sui videogiochi;
- un'**ontologia**, che descrive il dominio d'interesse specificando il significato dei simboli nel sistema.

Una volta avviato il programma, l'utente interagisce con questo, potendo in qualsiasi momento scegliere l'opzione che più può accontentare i suoi bisogni del momento potendosi spostare velocemente tra il recommender system, la knowledge base e l'ontologia.

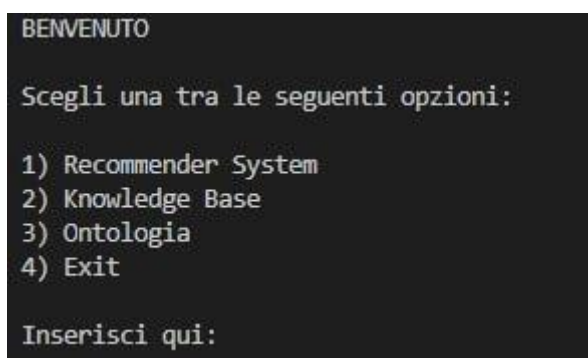


Figura 1-1: Menu principale del programma

1.1 DATASET UTILIZZATI E LIBRERIE

Il dataset utilizzato è: <https://www.kaggle.com/datasets/nikdavis/steam-store-games>, a cui sono già state applicate delle operazioni di data cleaning che consistono nella rimozione di elementi compromessi, ordinamento e un controllo di qualità.

Tutto il lavoro è stato scritto in linguaggio Python, utilizzando il code editor VsCode. Per l'intero progetto è stato fatto uso delle librerie **Pandas** e **NumPy**.

2 RECOMMENDER SYSTEM

Un sistema di raccomandazione o motore di raccomandazione è un software di filtraggio dei contenuti che crea delle raccomandazioni personalizzate specifiche per l'utente così da aiutarlo nelle sue scelte. Esistono tre tipi di approccio utilizzato per creare raccomandazioni: l'approccio collaborativo, l'approccio basato sul contenuto e l'approccio ibrido.

Abbiamo deciso di utilizzare l'approccio basato sul contenuto, anche detto **Content-based Filtering**, poiché grazie al dataset scelto abbiamo potuto accedere a una serie di informazioni dettagliate sui videogiochi, suddivise in categorie. Questa tipologia di approccio incrocia il contenuto di un elemento e il profilo di un utente. Il contenuto di un elemento è costituito dalla sua descrizione, attributi, parole chiave ed etichette. Questi dati vengono messi a confronto con il profilo utente che ne racchiude le preferenze.

Nel nostro caso, non avendo a disposizione un profilo utente precostruito, abbiamo optato per una interazione diretta con l'utente, il quale inserisce i dati del videogioco da lui scelto o preferito, per poi ricevere una raccomandazione basata su di esso.

```
GET RECOMMENDED

Benvenuto, digita le caratteristiche del gioco su cui vuoi che si avvii la raccomandazione

Inserisci il nome:
Counter-Strike

Inserisci lo sviluppatore:
Valve

Inserisci la casa editrice:
Valve

Inserisci le piattaforme:      (ricorda tra una parola e l'altra di mettere il simbolo ';' )
windows;mac;linux

Inserisci il genere:          (ricorda tra una parola e l'altra di mettere il simbolo ';' )
Action;Classic;Singleplayer[]

Questo è il videogioco che hai inserito:

      name developer publisher      platforms      genres
0 Counter-Strike   Valve      Valve windows;mac;linux Action;Classic;Singleplayer

E' corretto?:  si[]
```

2.1 SCELTA DELLE METRICHE PER RECOMMENDER

Prima di poter avanzare nella realizzazione del recommender system, ci siamo occupati dello studio sulle metriche che calcolassero la similarità.

Abbiamo messo a confronto 3 metriche preselezionate: similarità del coseno, correlazione di Pearson e distanza Euclidea.

Tempo di esecuzione

Come primo passo, ci siamo occupati di misurare i tempi di elaborazione per ciascuna metrica nel contesto della raccomandazione dei giochi: grazie all'utilizzo della libreria Python **Time**, si è potuta calcolare la quantità di tempo utilizzata per elaborare una raccomandazione in base alla metrica scelta.

```
Tempo di esecuzione con similarità del coseno: 213.8078534603119
Tempo di esecuzione con distanza euclidea: 310.39506936073303
Tempo di esecuzione con correlazione di Pearson: 15.91775631904602
```

Come possiamo vedere dall'immagine, la correlazione di Pearson si mostra, rispetto alle altre due metriche, estremamente più veloce.

Il fattore tempo si mostra importante ai fini dello sviluppo del progetto: per poter implementare e testare il codice, è necessario che l'elaborazione non sia troppo lunga e complessa.

Il secondo punto di questo studio è stato un confronto diretto delle caratteristiche delle 3 metriche.

Similarità del coseno

Sappiamo che la similarità del coseno calcola il coseno dell'angolo tra due vettori. Sappiamo anche che è molto utilizzata per quanto riguarda l'analisi di documenti e testi. Viene usata per determinare quanto simili sono i documenti tra loro ignorandone la dimensione. Infatti, ritroviamo le tecniche di TF-IDF di analisi del testo che aiutano a convertire i documenti in vettori dove ogni valore all'interno di questi corrisponde allo score di una parola nel documento generato dal TF-IDF. Ogni parola così ha il suo asse, la similarità del coseno infine determina la similarità dei documenti.

Risulta così popolare nel contesto dell'**NLP** (Natural Language Processing) poiché una caratteristica ricercata in questo campo è quella di avere delle strategie computazionali che siano più veloci nel caso in cui si abbia a che fare con dati che presentano grande sparsità.

Nel nostro caso, però, dopo aver effettuato un calcolo per poter misurare la sparsità del dataset utilizzato, abbiamo notato che la percentuale fosse molto bassa. Questo potrebbe spiegare come mai i tempi di esecuzione, utilizzando come metrica la similarità del coseno, siano così alti.

```
Sparsità del dataset: 10.23120960295476 %
```

Correlazione di Pearson

L'indice di correlazione di Pearson è una tecnica per investigare un'eventuale relazione di linearità tra due quantitativi, che siano continui. È una misura legata alla forza e alla direzione di una relazione lineare.

Può avere un range di valori da -1 a +1. Ottenere un valore equivalente ad uno dei due estremi corrisponde all'avere una perfetta correlazione lineare positiva o negativa.

Distanza Euclidea

La distanza euclidea è una distanza tra due punti, in particolare è una misura della lunghezza del segmento avente per estremi i due punti. Usando questa distanza, lo spazio euclideo diventa uno spazio metrico. I metodi basati sulla distanza danno priorità agli elementi con i valori più bassi per poter riconoscere una possibile similarità tra di essi.

Differenza tra coseno e Pearson

La similarità del coseno e la correlazione di Pearson possono essere visti come due varianti del prodotto interno (dot product).

Avendo due vettori \mathbf{x} e \mathbf{y} e vogliamo misurare la similarità tra loro, quella più semplice disponibile è il prodotto interno.

$$Inner(\mathbf{x}, \mathbf{y}) = \sum_i x_i y_i = \langle \mathbf{x}, \mathbf{y} \rangle$$

Se il vettore \mathbf{x} tende ad essere alto dove anche \mathbf{y} lo è e basso dove anche \mathbf{y} lo è, il prodotto interno sarà alto a suo modo, e i vettori saranno quindi più simili tra loro.

Il prodotto interno si presenta senza limiti; un modo per poterlo limitare tra -1 e 1 è quello di dividerlo per le norme L2 dei vettori, ottenendo così la similarità del coseno.

$$CosSim(\mathbf{x}, \mathbf{y}) = \frac{\sum_i x_i y_i}{\sqrt{\sum_i x_i^2} \sqrt{\sum_i y_i^2}} = \frac{\langle \mathbf{x}, \mathbf{y} \rangle}{\|\mathbf{x}\| \|\mathbf{y}\|}$$

La similarità del coseno non è invariante ai cambiamenti. Se \mathbf{x} venisse cambiato in $\mathbf{x} + \mathbf{1}$, di conseguenza anche la similarità cambierebbe. Invece ciò che si presenta invariante è la correlazione di Pearson.

$$\begin{aligned} Corr(\mathbf{x}, \mathbf{y}) &= \frac{\sum_i (x_i - \bar{x})(y_i - \bar{y})}{\sqrt{\sum_i (x_i - \bar{x})^2} \sqrt{\sum_i (y_i - \bar{y})^2}} \\ &= \frac{\langle \mathbf{x} - \bar{x}, \mathbf{y} - \bar{y} \rangle}{\|\mathbf{x} - \bar{x}\| \|\mathbf{y} - \bar{y}\|} \\ &= CosSim(\mathbf{x} - \bar{x}, \mathbf{y} - \bar{y}) \end{aligned}$$

La correlazione non è altro che la similarità del coseno tra delle versioni centrate dei vettori \mathbf{x} e \mathbf{y} , limitate tra -1 e 1. A differenza del coseno, la correlazione è invariante sia ai cambiamenti di scala che di posizione di \mathbf{x} e \mathbf{y} .

Riassumendo, la similarità del coseno è un prodotto interno normalizzato e la correlazione è un coseno di similarità centrato.

Differenza tra coseno e distanza Euclidea

Confrontata col coseno, la distanza euclidea non è usata molto spesso nel contesto delle applicazioni dell'NLP. È appropriata perlopiù per variabili numeriche continue. Non è invariante alla scala, inoltre di solito viene raccomandato di ridimensionare i dati prima di calcolarne la distanza. Inoltre, moltiplica l'effetto delle informazioni ridondanti nel set di dati.

Decisione finale

Dopo aver analizzato più punti per ciascuna metrica, abbiamo composto una tabella, che potesse aiutarci a selezionare quella più utile ai nostri scopi e per i fini del progetto:

Metriche	Contesto NLP	Tipi di variabili	Invariante alle modifiche	Range di valori	Tempo di computazione codice
Coseno	si	Vettori non vuoti con N variabili	no	[-1,1] o [0,1]	213 s
Pearson	//	continue	si	[-1,1]	15.9 s
Euclidea	no	numeriche continue	no	[0,R]	310 s

La metrica scelta è stata la correlazione di Pearson.

Motivazioni

Nonostante il coseno della similarità sia utilizzato spesso nel contesto dell’NLP e si trovi in sintonia con l’uso di tecniche di TF-IDF, abbiamo optato per la correlazione di Pearson poiché si presenta invariante alle modifiche e presenta un tempo di computazione molto basso. L’invarianza alle modifiche risulta adatta in un contesto di raccomandazione dove i dati possono essere modificati nel tempo. Inoltre, il fattore tempo risulta molto importante e le altre due metriche hanno dimostrato di eseguire il codice in tempi molto più lunghi.

2.2 REALIZZAZIONE

Per la realizzazione del recommender system con Content-based Filtering, abbiamo utilizzato la libreria open source di Python **SKLearn**, la quale ci ha permesso di utilizzare una serie di algoritmi dedicati al Natural Language Processing.

```
import sklearn
from sklearn.feature_extraction.text import TfidfVectorizer
```

Figura 2-1 Libreria sklearn

Per poter calcolare la raccomandazione, abbiamo sfruttato 5 categorie prese dal dataset: “name”, “developer”, “publisher”, “platforms” e “genre”. Poiché però i dati prelevati da ciascuna categoria si presentano in un formato non adatto al calcolo della similarità tra videogiochi diversi, si è trovata la necessità di trasformare la concatenazione di queste 5 categorie in dei vettori di parole.

Questi vettori di parole sono una rappresentazione vettorizzata delle parole in un “documento”, delle quali il vettore porta con sé un significato semantico per ciascuna di esse.

$$TF(t, d) = \frac{\text{number of times } t \text{ appears in } d}{\text{total number of terms in } d}$$

$$IDF(t) = \log \frac{N}{1 + df}$$

$$TF - IDF(t, d) = TF(t, d) * IDF(t)$$

Figura 2-2: Formula TF-IDF

Tramite l’utilizzo della TF-IDF (term frequency-inverse document frequency) i vettori vengono computati elaborando così una matrice dove ogni colonna rappresenta una parola presa dalla concatenazione delle categorie scelte e allo stesso tempo rappresenterà il videogioco in sé.

Essenzialmente lo score del TF-IDF rappresenta la frequenza di una parola in un documento, pesata sul numero di documenti in cui compare. Questo vien fatto per ridurre l'importanza delle parole che possono apparire frequentemente e di conseguenza anche l'impatto che avrebbero nel calcolo finale.

La libreria **SKLearn** contiene una classe built-in chiamata *TfidfVectorizer* che si occupa di produrre la matrice TF-IDF.

```
vectorizer = TfidfVectorizer(analyzer='word')
tfidf_matrix = vectorizer.fit_transform(steam_data['all_content'])
```

Figura 2-3: Utilizzo della classe *TfidfVectorizer*

Una volta prodotta, si può procedere nel calcolo della similarità, con l'utilizzo della correlazione di Pearson.

$$\begin{aligned} \text{Corr}(x, y) &= \frac{\sum_i (x_i - \bar{x})(y_i - \bar{y})}{\sqrt{\sum (x_i - \bar{x})^2} \sqrt{\sum (y_i - \bar{y})^2}} \\ &= \frac{\langle x - \bar{x}, y - \bar{y} \rangle}{\|x - \bar{x}\| \|y - \bar{y}\|} \\ &= \text{CosSim}(x - \bar{x}, y - \bar{y}) \end{aligned}$$

Figura 2-4: Formula della correlazione di pearson

Si otterrà così una matrice che conterrà gli score di similarità di ogni videogiochi con ogni altro videogiochi. Ovvero, nella matrice ogni videogiochi è rappresentato da una colonna-vettore dove ogni sua posizione rappresenterà lo score di similarità che ha con ogni altro videogiochi.

```
tfidf_matrix = vectorize_data(data)
tfidf_matrix_array = tfidf_matrix.toarray()

indices = pd.Series(data['name'].index)

id = indices[index]
correlation = []
for i in range(len(tfidf_matrix_array)):
    correlation.append(pearsonr(tfidf_matrix_array[id], tfidf_matrix_array[i])[0])
```

A questo punto per la raccomandazione vera e propria utilizzeremo un reverse mapping dei nomi dei videogiochi e degli indici presi dal dataframe, ovvero un meccanismo che possa identificare l'indice del videogiochi dato il nome.

Verranno mostrati infine i primi 5 videogiochi più simili a quello dato in input dall'utente.

Ecco a te i 5 giochi più simili a quello proposto:

	name	genres	developer	price
10	Counter-Strike: Source	Action;FPS;Multiplayer	Valve	7.19
7	Counter-Strike: Condition Zero	Action;FPS;Multiplayer	Valve	7.19
25	Counter-Strike: Global Offensive	FPS;Multiplayer;Shooter	Valve;Hidden Path Entertainment	0.00
5	Ricochet	Action;FPS;Multiplayer	Valve	3.99
3	Deathmatch Classic	Action;FPS;Multiplayer	Valve	3.99

Figura 2-6: Risultato della raccomandazione

2.3 CLASSIFICAZIONE

In aggiunta al recommender system, abbiamo incluso una classificazione degli elementi presenti nel dataset in base ad una nuova categoria nominata “*star*” creata da noi stessi utilizzando categorie già esistenti riguardanti i ratings dei videogiochi.

```
#creiamo le categoria star
steam_data['star'] = (steam_data['negative_ratings'] / steam_data['positive_ratings']) * 100

#assegniamo i dati alla sezione corretta
steam_data.loc[(steam_data['star'] >= 0) & (steam_data['star'] <= 12.5), ['star']] = 5
steam_data.loc[(steam_data['star'] > 12.5) & (steam_data['star'] <= 25), ['star']] = 4
steam_data.loc[(steam_data['star'] > 25) & (steam_data['star'] <= 37.5), ['star']] = 3
steam_data.loc[(steam_data['star'] > 37.5) & (steam_data['star'] <= 50), ['star']] = 2
steam_data.loc[(steam_data['star'] > 50), ['star']] = 1
```

Sono stati creati 5 range all’interno dei quali far rientrare i dati già presenti nel dataset.

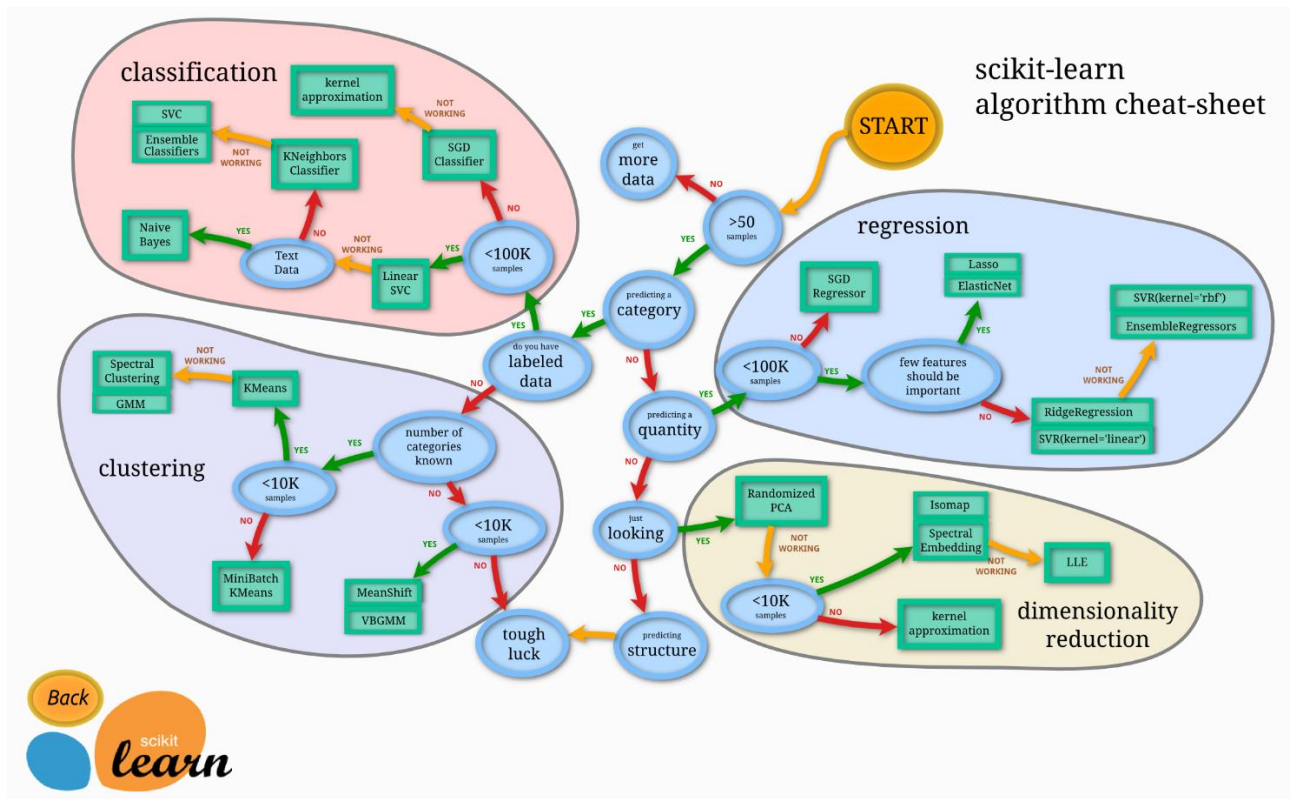
Inoltre, sono state effettuate una serie di valutazioni del modello scelto con una successiva fase di correzione e miglioramento di esso.

Per la selezione del modello di classificazione da utilizzare, abbiamo optato per il KNeighborsClassifier fornito sempre dalla libreria **SKLearn**.

Modello

Il **k-nearest neighbors**, abbreviato in K-NN, è un algoritmo utilizzato nel riconoscimento di pattern per la classificazione di oggetti basandosi sulle caratteristiche degli oggetti vicini a quello considerato. In entrambi i casi, l'input è costituito dai *k* esempi di addestramento più vicini nello spazio delle funzionalità. Sebbene possa essere utilizzato per problemi di regressione o classificazione, viene generalmente utilizzato come algoritmo di classificazione, basandosi sul presupposto che punti simili possono essere trovati l'uno vicino all'altro. L'output è un'appartenenza a una classe. Un oggetto è classificato da un voto di pluralità dei suoi vicini, con l'oggetto assegnato alla classe più comune tra i suoi *k* vicini più vicini (*k* è un numero intero positivo, tipicamente piccolo). Se *k* = 1, l'oggetto viene semplicemente assegnato alla classe di quel singolo vicino più prossimo.

Per la scelta del modello ci siamo affidati ad una mappa presa dal sito della libreria **Sklearn**, la quale mostra un percorso nella ricerca del miglior estimatore di machine learning in base ai propri scopi:



Si può vedere come, seguendo le linee per arrivare al knn, venga descritta la situazione del nostro progetto: predizione di una categoria (*star*), presenza di dati categorizzati, utilizzo di dati non testuali.

Dataset

Per avere un dataset appropriato all'utilizzo del modello scelto e alla classificazione, questo è stato suddiviso in due parti: training e test. La parte di training (addestramento), verrà utilizzata per allenare il modello, il quale poi verrà effettivamente testato sulla parte rimanente (test).

Nel nostro caso abbiamo deciso di conservare un 80% di dati per la parte di training e un 20% per la parte di test.

Un'altra importante caratteristica riguarda l'assegnazione casuale degli elementi del dataset alle due frazioni. In questo modo ci si assicura che i due sotto-dataset corrispondenti siano un buon esempio rappresentativo del dataset originale e che siano ben distribuiti. Ciò può essere fatto impostando il parametro "*random-state*" con un numero intero. Ogni valore andrà bene non essendo un parametro ricercabile (hyperparameters tuning).

Infine, poiché alcuni problemi di classificazione presentano delle categorie con un numero non bilanciato di esempi, è desiderabile suddividere il dataset in parte di training e test, ma in modo che si possa preservare la stessa porzione di esempi per ogni categoria presente nel dataset originale. Questo viene chiamato uno split stratificato. Si può infatti modificare il parametro "*stratify*" con il componente "*y*" del dataset originale.

```
#splittiamo il dataset in due parti, training e test, con una ratio di 80% training e 20% test
X_train, X_test, y_train, y_test = train_test_split(x, y, test_size=0.2, random_state=1, stratify=y)
```

Pre-processing dei dati

Ci siamo occupati della scalatura dei dati per normalizzare il range di variazione delle caratteristiche del dataset. In questo modo abbiamo migliorato la qualità dei risultati finali: lo scaling riduce il tempo in cui l'algoritmo di apprendimento converge al risultato finale.

```
#trasformiamo i dati per renderli adeguati
scaler = StandardScaler()
X_train = scaler.fit_transform(X_train)
X_test = scaler.transform(X_test)
predict_data = scaler.transform(predict_data)
```

Successivamente il modello viene addestrato sulla parte di training con dei parametri di default, grazie alle funzioni fornite dalla stessa libreria **SKlearn**.

```
print('\n\nIniziale composizione del modello con hyperparameters basici...')
knn = KNeighborsClassifier(n_neighbors=5, weights='uniform', algorithm='auto', p=2, metric='minkowski', metric_params=None, n_jobs=None)
knn.fit(X_train,y_train)
```

Miglioramento del modello e Hyperparameters Tuning

Dopo aver testato il modello creato con parametri di default, abbiamo provato ad elaborare delle possibili predizioni sulla categoria “star” e di conseguenza abbiamo anche valutato il modello per poter testare l’accuratezza delle predizioni.

```
Iniziale composizione del modello con hyperparameters basici...

Predizioni dei primi 5 elementi: [5. 5. 1. 2. 1.] Valori effettivi: [1. 1. 2. 4. 4.]

Valutazione del modello...

Classification report:
              precision    recall  f1-score   support

     1.0         0.40      0.61      0.48       1818
     2.0         0.16      0.08      0.11        557
     3.0         0.15      0.10      0.12        650
     4.0         0.24      0.18      0.20        964
     5.0         0.34      0.29      0.31       1426

 accuracy          0.33       5415
 macro avg         0.26       5415
weighted avg         0.30       5415

ROC score: 0.5502741408667262

La nostra accuratezza è bassa, dobbiamo migliorare la qualità delle nostre predizioni
```

Abbiamo utilizzato un “*Classification report*” della **sklearn.metrics** per poter includere più metriche di valutazione e infine abbiamo calcolato un ROC AUC score, il quale ci ha dato un risultato di 0.55. **ROC AUC** (Compute Area Under the Receiver Operating Characteristic Curve), ovvero la misura dell'intera area bidimensionale sotto l'intera curva ROC, ci fornisce una misurazione aggregata del rendimento di un modello di classificazione in tutte le possibili soglie di classificazione.

In seguito, abbiamo sottoposto il nostro modello a un miglioramento introducendo una ricerca dei migliori parametri da potergli affidare per aumentare il ROC score, l’accuratezza e le altre metriche di valutazione.

Abbiamo optato per l'utilizzo di una *RandomizedSearchCv*, per la ricerca dei parametri, includendo l'utilizzo di una *RepeatedKfold* cross validation.

La Randomized Search è una tecnica attraverso la quale combinazioni casuali di iperparametri vengono usate per cercare la migliore soluzione per il modello di classificazione da implementare. A differenza della Grid Search risulta più veloce proprio perchè la combinazione di parametri viene scelta in maniera casuale e inoltre risulta più efficace poiché riesce a restituire dei parametri che a confronto sono nettamente migliori.

```
cvFold = RepeatedKfold(n_splits=10, n_repeats=3, random_state=1)
randomSearch = RandomizedSearchCV(estimator=knn, cv=cvFold, param_distributions=hyperparameters)
```

Una volta deciso quale approccio utilizzare, sono stati creati una serie di parametri di ricerca, ovvero il numero k di neighbors, la tipologia di "weights" e la tipologia di metrica da utilizzare.

```
n_neighbors = list(range(1,30))
weights = ['uniform', 'distance']
metric = ['euclidean', 'manhattan', 'hamming']
```

La ricerca è stata ripetuta ben 15 volte, dove ogni volta i parametri e il risultato del ROC score venivano salvati all'interno di un dizionario, creato appositamente e in seguito ordinato sulla base del ROC score, così da poter scegliere il miglior set di parametri tra tutti per ogni elaborazione.

Una volta trovati i parametri migliori, si è reimpostato il modello con essi e si è provato ad elaborare delle predizioni sulla categoria "star".

```
Best weights: uniform
Best metric: manhattan
Best n_neighbors: 29

ROC score: 0.5834366489378043

Abbiamo incrementato la accuratezza del nostro modello
```

Il risultato del ROC score risulta effettivamente incrementato, da 0.55 a 0.58.

Predizione su nuovi dati

Dopo aver migliorato i parametri del nostro modello, abbiamo provato a fare delle predizioni sui giochi raccomandati all'utente, sulla categoria star, utilizzando il nuovo modello.

Ora possiamo procedere alla fase di recommendation...

Ecco a te i 5 giochi più simili a quello proposto con una predizione sulla categoria star:

	name	genres	developer	price	star	star_prediction
10	Counter-Strike: Source	Action;FPS;Multiplayer	Valve	7.19	5.0	5.0
7	Counter-Strike: Condition Zero	Action;FPS;Multiplayer	Valve	7.19	5.0	4.0
25	Counter-Strike: Global Offensive	FPS;Multiplayer;Shooter	Valve;Hidden Path Entertainment	0.00	4.0	5.0
5	Ricochet	Action;FPS;Multiplayer	Valve	3.99	4.0	4.0
3	Deathmatch Classic	Action;FPS;Multiplayer	Valve	3.99	4.0	5.0

Possiamo notare che le predizioni, pur non essendo perfette al 100% rispetto ai risultati ottenuti precedentemente, abbiano restituito dei miglioramenti.

Conclusione

Nonostante le predizioni siano state effettuate sulla categoria "star", vi era originariamente un'altra

possibilità, ovvero quella di utilizzare la categoria già esistente *“english”*, che presentava valori di 0 o 1 per ogni videogioco.

Abbiamo comunque deciso di utilizzare la prima categoria, poiché nella fase di valutazione del modello, nel caso della categoria *“english”*, l’accuratezza si presentava con un valore molto alto sin dall’inizio (98%) e le predizioni erano quasi completamente perfette.

Quindi, il lavoro di miglioramento e ricerca di nuovi parametri non avrebbe avuto un reale riscontro nel modificare una situazione già ottimale. Così siamo rimasti decisi nella scelta di utilizzare la categoria *“star”*, per mostrare un effettivo miglioramento e dare una validità al lavoro svolto.

3 KNOWLEDGE BASE

Una knowledge base è una banca dati che riesce a fornire un supporto all'utente grazie alle informazioni presenti al suo interno, e rappresenta fatti sul mondo, un ragionamento su quei fatti e utilizza delle regole e altre forme di logica per dedurre nuovi fatti o evidenziare incongruenze. Quindi, la knowledge base (o KB) è definibile come un insieme di assiomi, cioè delle proposizioni che possono essere asserite essere vere che costituiscono dunque un ambiente volto a facilitare la raccolta, l'organizzazione e la distribuzione della conoscenza.

In merito al nostro caso di studio, la KB viene utilizzata per consentire all'utente di porre domande inerenti al dominio approfondito.

3.1 GESTIONE DELLE KNOWLEDGE BASE

Per la gestione della knowledge base, abbiamo utilizzato l'estensione **Pytholog** del linguaggio Python basata sul linguaggio di programmazione logica Prolog.

Sulla base del Prolog, quindi, abbiamo iniziato con il popolamento dei fatti della knowledge base, che avviene automaticamente prelevando i dati interessati alla costruzione del fatto direttamente dal dataset. Così facendo, nel caso in cui il dataset venisse aggiornato con nuovi dati, si aggiornerebbe direttamente anche la KB. Abbiamo poi aggiunto le regole, basandoci su come potessimo gestire i fatti che popolano la KB e su come si sarebbe potuta svolgere l'interazione da parte dell'utente.

3.1.1 Fatti

I fatti rappresentano gli assiomi sempre veri della knowledge base, e sono la base per il funzionamento delle regole. Abbiamo 6 tipologie di fatti, ognuno rappresentante una relazione tra i giochi e una delle possibili caratteristiche associate a questi.

Le tipologie di fatti presenti nella KB sono le seguenti, e per ognuno è riportato un esempio:

- 1) `"developer(name, developer)"`
Rappresenta la relazione tra un gioco e chi lo ha sviluppato.
es. `developer(counter-strike, valve)`
- 2) `"publisher(name, publisher)"`
Rappresenta la relazione tra un gioco e chi lo ha rilasciato.
es. `publisher(counter-strike, valve)`
- 3) `"prices(name, price)"`
Rappresenta la relazione tra un gioco e il suo prezzo.
es. `prices(counter-strike, 7.19)`

4) "stars(name, star)"

Rappresenta la relazione tra un gioco e l'apprezzamento da parte degli utenti. 'star' è il valore in percentuale del rapporto tra i rating negativi e i rating positivi, successivamente convertiti in dei numeri interi compresi tra 1 e 5 in base al risultato del rapporto.

```
steam_data.loc[(steam_data['star'] >= 0) & (steam_data['star'] <= 12.5), ['star']] = 5
steam_data.loc[(steam_data['star'] > 12.5) & (steam_data['star'] <= 25), ['star']] = 4
steam_data.loc[(steam_data['star'] > 25) & (steam_data['star'] <= 37.5), ['star']] = 3
steam_data.loc[(steam_data['star'] > 37.5) & (steam_data['star'] <= 50), ['star']] = 2
steam_data.loc[(steam_data['star'] > 50), ['star']] = 1
```

es. stars(counter-strike, 5)

5) "genre(name, steamspy_tags)"

Rappresenta la relazione tra un gioco e il suo genere.

es. genre(counter-strike, action;fps;multiplayer)

6) "english(name, english)"

Rappresenta la relazione tra un gioco e la presenza o meno della lingua inglese. I valori "0" e "1" sono stati rispettivamente convertiti nelle stringhe "no" e "yes".

```
steam_data.loc[(steam_data['english'] == 0), ['english']] = 'no'
steam_data.loc[(steam_data['english'] == 1), ['english']] = 'yes'
```

es. english(counter-strike, yes)

3.1.2 Regole

Le regole sono il fulcro dell'interazione con l'utente. Rappresentano delle domande che l'utente può rivolgere alla knowledge base senza il bisogno di conoscere la sintassi necessaria. Utilizzano i fatti che popolano la KB per poter mostrare all'utente i dati che questi ha richiesto.

Le regole presenti nella KB sono le seguenti, e per ognuna è riportato un esempio:

1) "has_price(X, Y) :- prices(Y, X)"

Permette di ottenere informazioni sul prezzo di un gioco o su giochi di un determinato prezzo.

es. "has_price(X, counter-strike)" -> X = 7.19

2) "quality(X, Y) :- stars(Y, X)"

Permette di ottenere informazioni sulla qualità di un gioco, basandosi sulle stelle.

es. "quality(X, counter-strike)" -> X = 5

3) "developed_by(X, Y) :- developer(Y, X)"

Permette di ottenere informazioni su chi ha sviluppato un gioco.

es. "developed_by(X, counter-strike)" -> X = valve

4) "released_by(X, Y) :- publisher(Y, X)"

Permette di ottenere informazioni su chi ha rilasciato un gioco.

es. "released_by(X, counter-strike)" -> X = valve

- 5) `"is_genre(X, Y) :- genre(Y, X)"`
 Permette di ottenere informazioni sul genere di un gioco.
 es. `"is_genre(X, counter-strike)" -> X = action;fps;multiplayer`
- 6) `"has_english(X, Y) :- english(Y, X)"`
 Permette di chiedere se il gioco presenta la lingua inglese o meno.
 es. `"has_english(X, counter-strike)" -> X = yes`
- 7) `"quality_check(X, Y, T, Z) :- stars(X, T), stars(Y, Z)"`
 Permette di ottenere informazioni sulla qualità di due giochi diversi, mettendoli a confronto basandosi sulle stelle.
 es. `"quality_check(team fortress classic, counter-strike, T, Z)"`
`-> T = 4, Z = 5`

3.1.3 Liking Probability

Ci siamo occupati anche di creare dei fatti e delle regole che interagissero tra di loro in modo di dare all'utente la possibilità di calcolare la probabilità che un nuovo gioco gli possa piacere in base ai fatti che popolano la KB.

Stress

- 1) `"has_work(utente, ore_di_lavoro)"`
 Fatto che permette di controllare le ore di lavoro dell'utente.
 - 2) `"stress(utente, Prob1) :- has_work(Persona, Prob2), Prob1 is Prob2"`
 Regola che permette di calcolare l'indice di stress dell'utente
- Questi due fattori servono per calcolare, in base allo stress e alle ore di lavoro dell'utente, se il nuovo gioco gli possa piacere.

Gioco piaciuto

- 1) `"liked_game(utente, gioco_piaciuto)"`
 Fatto che serve a controllare se un gioco piace all'utente.

Average Playtime

- 1) `"avg_playtime(name, average_playtime)"`
 Fatto che rappresenta la relazione tra un gioco e l'avg_playtime di quel gioco.
- 2) `"has_avg_playtime(X, Y) :- avg_playtime(X, Y)"`
 Regola che permette di risalire all'avg_playtime di un gioco.
- 3) `"avg_playtime_comp(Gioco_Piaciuto, Gioco_Nuovo, Y) :- liked_game(utente, Gioco_Piaciuto), has_avg_playtime(Gioco_Piaciuto, Avg1), has_avg_playtime(Gioco_Nuovo, Avg2), Y is Avg1 - Avg2"`
 Regola che fa la differenza tra l'avg_playtime del gioco che già piace all'utente e del gioco nuovo.

Il valore ottenuto da avg_playtime_comp verrà confrontato con dei range di valori che rappresentano la somiglianza di avg_playtime. Più il valore è vicino allo 0, più sono simili. Il valore corrispondente nel range viene salvato in una variabile Playtime_Similarity.

Genre

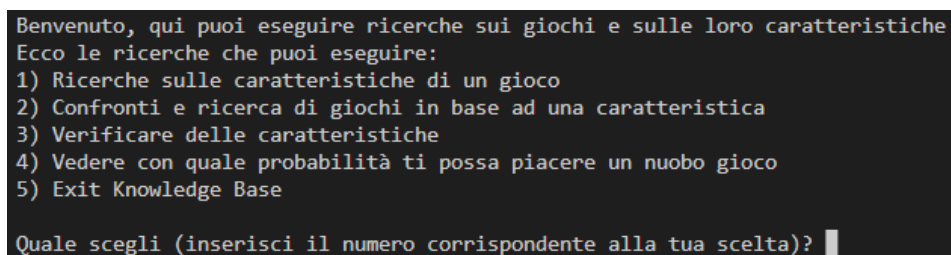
- 1) `"same_genre(genre, genre)"`
Fatto che permette di controllare se il genere di due giochi è lo stesso. Se il risultato della query è **Yes**, il fatto esiste e quindi è uguale.
- 2) `"has_same_genre(Gioco1, Gioco2,) :- is_genre(Gioco1, X), is_genre(Gioco2, Y), same_genre(X, Y)"`
Regola che controlla se due giochi hanno lo stesso genere. Al termine di una query su questa regola, verrà creata una variabile Genre_Similarity, che avrà valore 2.5 se il risultato è **Yes** e valore -2.5 se il risultato è **No**.

Calcolo della probabilità

- 1) `"compatibility(Num1, Num2, S1) :- S1 is Num1 + Num2"`
Regola che somma i valori di Similarity di avg_playtime e genre.
- 2) `"to_like(utente, Num1, Num2, Probabilità) :- stress(utente, Prob1), compatibility(Num1, Num2, Score), Probabilità is Prob1 + Score"`
Regola che calcola la probabilità che un gioco nuovo possa piacere all'utente in base agli score di similarità e allo stress.

3.2 INTERAZIONE CON L'UTENTE

Durante l'esecuzione del programma, la seconda opzione permetterà all'utente di interagire con la knowledge base e porre domande riferite ai dati dei giochi.



```
Benvenuto, qui puoi eseguire ricerche sui giochi e sulle loro caratteristiche
Ecco le ricerche che puoi eseguire:
1) Ricerche sulle caratteristiche di un gioco
2) Confronti e ricerca di giochi in base ad una caratteristica
3) Verificare delle caratteristiche
4) Vedere con quale probabilità ti possa piacere un nuovo gioco
5) Exit Knowledge Base

Quale scegli (inserisci il numero corrispondente alla tua scelta)?
```

Figura 3-1: Schermata di benvenuto della KB

Qui l'utente potrà scegliere tra 3 tipi di domande da porre alla KB:

- 1) Ricerche sulle caratteristiche di un gioco passato in input dall'utente, che, tramite le regole, permettono di ottenere la caratteristica che l'utente richiede:

```
Quale scegli (inserisci il numero corrispondente alla tua scelta)? 1
Dammi il nome di un gioco: counter-strike
Queste sono le caratteristiche che puoi cercare:
1) Chi lo ha sviluppato
2) Chi lo ha distribuito
3) Quanto costa
4) Quante stelle ha (su una scala da 1 a 5)
5) Di che genere è
6) Se è disponibile in lingua inglese
Selezionane una: █
```

Figura 3-2: Menu mostrato alla scelta di ricerca 1

Da qui, l'utente sceglie un tipo di informazione che vuole ricevere dalla KB selezionando un numero da 1 a 6 e al termine della richiesta ha la possibilità di fare una nuova ricerca nello stesso campo o tornare indietro al menu principale di interazione con la KB (figura 4-1).

```
Selezionane una: 3

counter-strike costa: 7.19

Vuoi eseguire un'altra ricerca o vuoi tornare indietro? Indietro (sì), Continua (no) █
```

Figura 3-3: Risultato dell'operazione richiesta

- 2) Confronti di due giochi scelti dall'utente in base alla qualità di questi e ricerca di giochi in base ad una caratteristica:

```
Quale scegli (inserisci il numero corrispondente alla tua scelta)? 2
Queste sono ricerche che puoi eseguire sulle caratteristiche:
1) Lista di giochi di un prezzo
2) Confronto di qualità tra 2 giochi
3) Indietro

Selezionane una (inserisci il numero corrispondente alla tua scelta): █
```

Figura 3-4: Menu mostrato alla scelta di ricerca 2

Qui l'utente può scegliere quale tipo di ricerca approfondita esplorare selezionando un numero tra 1 e 2 per le ricerche e 3 per tornare indietro.

```
Selezionane una (inserisci il numero corrispondente alla tua scelta): 2
Dimmi il nome del primo gioco: counter-strike
Dimmi il nome del secondo gioco: rune lord

I due giochi hanno la stessa qualità, perché la qualità di counter-strike e rune lord è uguale
```

Figura 3-5: risultato dell'operazione di confronto richiesta

- 3) Verifiche sulle informazioni dei giochi, interrogando direttamente i fatti:
All'utente viene chiesto di inserire manualmente il tipo di informazione da verificare, che corrisponde alle tipologie di fatti presenti nella KB.

```
Quale scegli (inserisci il numero corrispondente alla tua scelta)? 3
Questi sono le caratteristiche che puoi verificare:
1) developer
2) publisher
3) prices
4) stars
5) genre
6) english
```

Figura 3-6: inserimento dei dati da parte dell'utente e risultato dell'operazione richiesta

L'utente deve quindi inserire il nome del gioco e un'informazione corrispondente al tipo di verifica che vuole eseguire. Ha la possibilità poi di ripetere nuovamente l'interazione o di tornare indietro al menu principale di interazione con la KB (figura 4-1).

```
Selezionane una (scrivi il nome dell'operazione da eseguire, tutto in minuscolo): developer
Quale gioco vuoi controllare? counter-strike
Inserisci un dato corrispondente alla caratteristica scelta: valve
['Yes']
Vuoi eseguire un'altra verifica o vuoi tornare indietro?          Indietro (sì), Continua (no)
```

4) Calcolo della probabilità che un gioco possa piacere:

All'utente viene chiesto di inserire un gioco che gli è piaciuto e un nuovo gioco di cui vuole sapere la probabilità che gli piaccia, delle informazioni riguardanti questi giochi e quante ore al giorno lavora o/e studia:

```
Quale scegli (inserisci il numero corrispondente alla tua scelta)? 4
Inserisci il nome del gioco che ti è piaciuto: Portal
Dimmi il genere del gioco: Action
e il tempo di gioco medio: 288

Inserisci il nome di un gioco di cui vuoi calcolare la probabilità che ti piaccia: Counter-Strike
Dimmi il genere del gioco: Action
e il tempo di gioco medio: 17162

Quante ore lavori/studi al giorno? 8

La probabilità che counter-strike ti possa piacere è: 10.5 %
```

4 ONTOLOGIA

L'ontologia, in informatica, è un modello di rappresentazione formale della realtà e della conoscenza. È una struttura di dati che consente di descrivere le entità (oggetti, concetti, ecc.) e le loro relazioni in un determinato dominio di conoscenza.

Abbiamo deciso di utilizzare e creare una ontologia che potesse rappresentare il dominio di un negozio di videogiochi, nel caso in cui l'utente, incuriosito da un apparecchio elettronico, utilizzi questo per farsi consigliare sui videogiochi presenti nel negozio o per esplorare le risorse e le informazioni su di essi, in maniera completamente automatizzata, senza la richiesta di una presenza "umana".

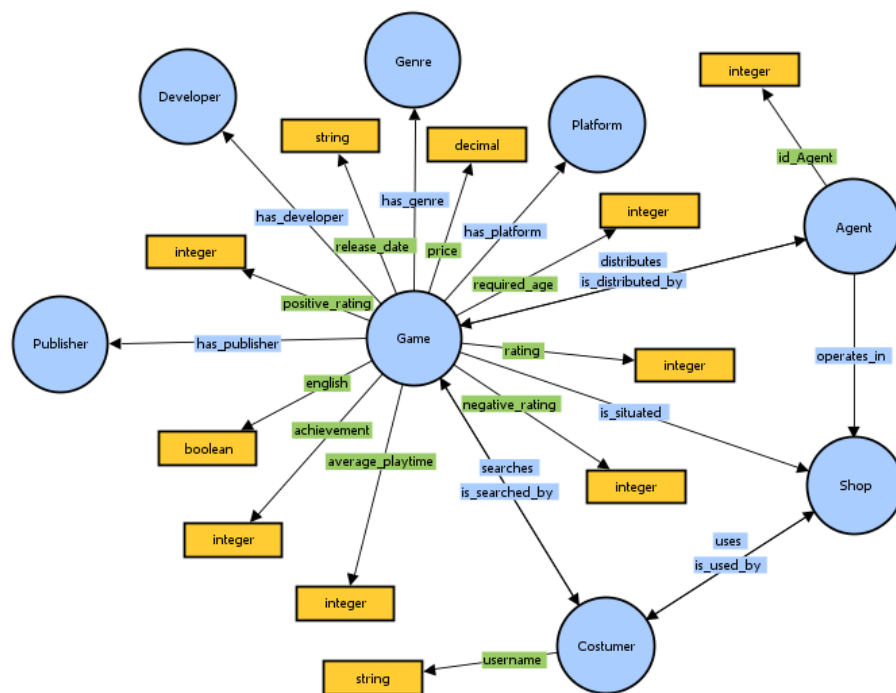


Figura 4-1: Struttura della Steam-Ontology

4.1 PROTÉGÉ

Per la realizzazione dell'Ontologia abbiamo deciso di utilizzare l'editor di ontologie open source **Protégé**. L'ontologia è organizzata in diverse classi, la cui maggior parte rappresenta una categoria legata al videogioco, che lo descrive nel dettaglio (developer, publisher, platform, genre); per il resto invece avremo le classi rappresentanti gli "attori" presenti nel negozio (agent, costumer) e lo stesso negozio in sé (shop).

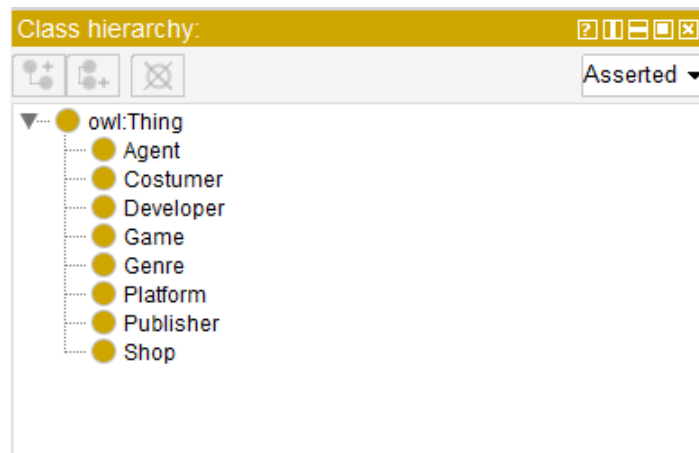


Figura 4-2: Classi dell'Ontologia

Oltre alla creazione delle classi, sono state create anche una serie di proprietà: object-properties e data-properties. Queste proprietà aiutano a relazionare due individui di classi uguali o diverse (object-properties) oppure permettono di relazionare un individuo al loro tipo di dato primitivo (data-properties).

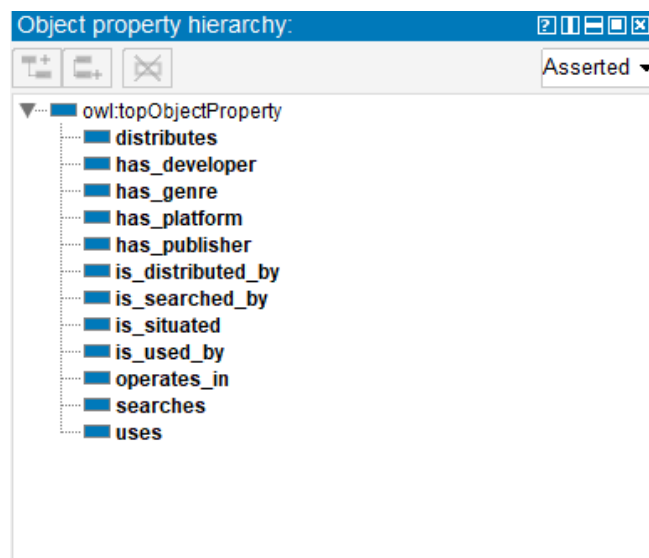


Figura 4-3: Object properties

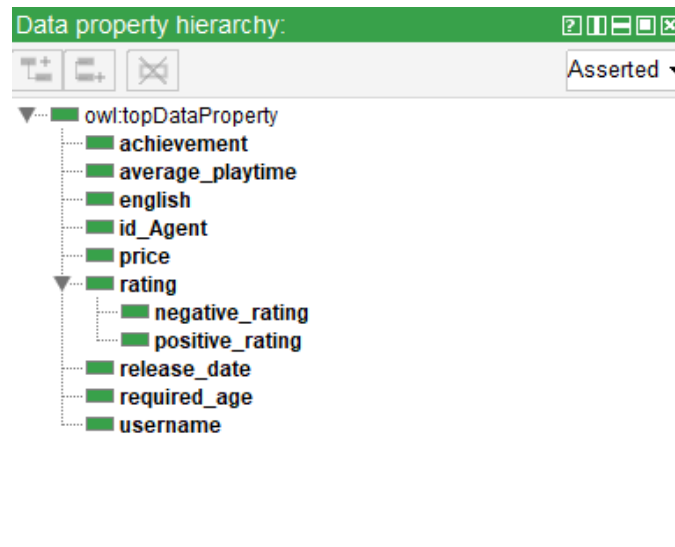


Figura 4-4: Data properties

Successivamente si è passati alla creazione degli individui stessi che rappresentano alcuni esempi di videogiochi presenti nel dataset.

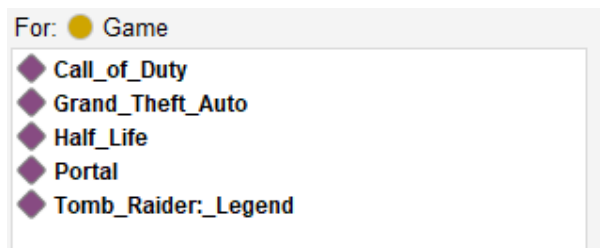


Figura 4-5: Individui della classe Game

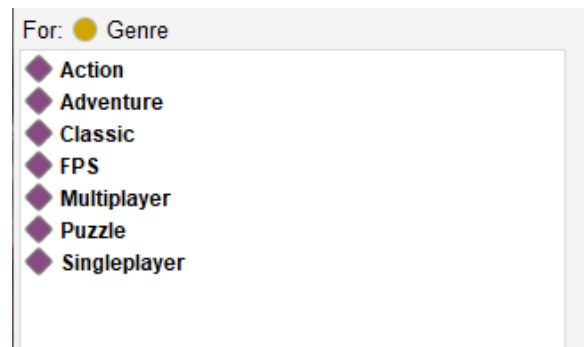


Figura 4-6: Individui della classe Genre

Description: Portal

Types
+
Game

Same Individual As
+

Different Individuals
+

Property assertions: Portal

Object property assertions
+

- has_developer Valve
- has_publisher Valve
- has_platform Mac
- has_platform Windows
- has_platform Linux
- has_genre Puzzle
- has_genre Singleplayer

Data property assertions
+

- achievement 15
- average_playtime 288
- required_age 0
- release_date "2007-10-10"^^xsd:string
- price 7.19
- english true
- negative_rating 1080
- positive_rating 51801

Figura 4-7: Esempio di individuo con properties annesse

È possibile anche interrogare l'ontologia tramite l'utilizzo di due tipologie di query, DL query e SPARQL.

SPARQL query:		
<pre> PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> PREFIX owl: <http://www.w3.org/2002/07/owl#> PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#> PREFIX xsd: <http://www.w3.org/2001/XMLSchema#> PREFIX table: <http://www.semanticweb.org/loren/ontologies/2022/7/steam_ontology#> SELECT ?Game ?achievement ?price WHERE{ ?Game table:achievement ?achievement . ?Game table:price ?price} </pre>		
Game	achievement	price
Call_of_Duty	"0" ^{^^} <http://www.w3.org/2001/XMLSchema#integer>	"14.99" ^{^^} <http://www.w3.org/2001/XMLSchema#decimal>
Tomb_Raider_Legend	"0" ^{^^} <http://www.w3.org/2001/XMLSchema#integer>	"4.99" ^{^^} <http://www.w3.org/2001/XMLSchema#decimal>
Grand_Theft_Auto	"0" ^{^^} <http://www.w3.org/2001/XMLSchema#integer>	"0" ^{^^} <http://www.w3.org/2001/XMLSchema#decimal>
Half_Life	"0" ^{^^} <http://www.w3.org/2001/XMLSchema#integer>	"7.19" ^{^^} <http://www.w3.org/2001/XMLSchema#decimal>
Portal	"15" ^{^^} <http://www.w3.org/2001/XMLSchema#integer>	"7.19" ^{^^} <http://www.w3.org/2001/XMLSchema#decimal>
Execute		

Figura 4-8: SPARQL query con visualizzazione dei Game, Achievement e Price per gioco

SPARQL query:	
<pre> PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> PREFIX owl: <http://www.w3.org/2002/07/owl#> PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#> PREFIX xsd: <http://www.w3.org/2001/XMLSchema#> PREFIX table: <http://www.semanticweb.org/loren/ontologies/2022/7/steam_ontology#> SELECT ?Game ?has_platform WHERE{ ?Game table:has_platform ?has_platform . ?Game table:price 7.19} </pre>	
Game	has_platform
Half_Life	Mac
Half_Life	Linux
Half_Life	Windows
Portal	Mac
Portal	Windows
Portal	Linux
Execute	

Figura 4-9: SPARQL query con visualizzazione dei giochi che hanno come object properties "has_platform" e un prezzo di 7.19

DL query:

Query (class expression)
 Game **that** has_genre **value** Singleplayer **and** price **value** 7.19

Execute
 Add to ontology

Query results
 Instances (2 of 2)

◆ Half_Life	?
◆ Portal	?

Figura 4-10: DL query con ricerca di giochi che hanno come genere "Singleplayer" e come prezzo "7.19"

DL query:

Query (class expression)
 Game **that** has_genre **value** Singleplayer **and** has_genre **value** Action

Execute
 Add to ontology

Query results
 Instances (4 of 4)

◆ Call_of_Duty	?
◆ Grand_Theft_Auto	?
◆ Half_Life	?
◆ Tomb_Raider:_Legend	?

Figura 4-11: DL query con ricerca di giochi che hanno come genere "Singleplayer" e anche "Action"

4.2 OWLREADY2

Per la consultazione in Python dell'ontologia, è stata utilizzata la libreria **Owlready2**, attraverso la quale è stato possibile, in maniera semplice e comprensibile, interrogare l'ontologia precedentemente creata con il tool Protégé. Nell'esecuzione del programma si potrà scegliere tra la

visualizzazione delle classi, delle proprietà di oggetto, delle proprietà dei dati e della visualizzazione di alcune query d'esempio.

```
BENVENUTO NELLA STEAM-ONTOLOGY

Seleziona cosa vorresti esplorare:

1) Visualizzazione Classi
2) Visualizzazione proprietà d'oggetto
3) Visualizzazione proprietà dei dati
4) Visualizzazione query d'esempio
5) Exit Ontologia
```

Figura 4-8: Menù principale per l'esplorazione dell'Ontologia

```
Classi presenti nell'ontologia:

[Steam-Ontology.Agent, Steam-Ontology.Game, Steam-Ontology.Developer, Steam-Ontology.Genre, Steam-Ontology.Platform, Steam-Ontology.Publisher, Steam-Ontology.Costumer, Steam-Ontology.Shop]

Vorresti esplorare meglio qualche classe in particolare?

1) Agent
2) Game
3) Developer
4) Genre
5) Platform
6) Publisher
7) Costumer
8) Shop

Inserisci qui la tua scelta: 3

Lista degli Sviluppatori presenti:

[Steam-Ontology.Developer, Steam-Ontology.Infinity_Ward, Steam-Ontology.Crystal_Dynamics, Steam-Ontology.Rockstar_North, Steam-Ontology.Valve, Steam-Ontology.id_Software]
```

Figura 4-9: Visualizzazione delle Classi presenti nell'Ontologia

```
Proprietà d'oggetto presenti nell'ontologia:

[Steam-Ontology.distributes, Steam-Ontology.is_distributed_by, Steam-Ontology.has_developer, Steam-Ontology.has_genre, Steam-Ontology.has_platform, Steam-Ontology.has_publisher, Steam-Ontology.is_searched_by, Steam-Ontology.searches, Steam-Ontology.uses, Steam-Ontology.operates_in]
```

Figura 4-10: Visualizzazione delle proprietà d'oggetto presenti nell'Ontologia

```
Proprietà dei dati presenti nell'ontologia:

[Steam-Ontology.achievement, Steam-Ontology.average_playtime, Steam-Ontology.english, Steam-Ontology.id_Agent, Steam-Ontology.negative_rating, Steam-Ontology.rating, Steam-Ontology.positive_rating, Steam-Ontology.price, Steam-Ontology.release_date, Steam-Ontology.required_age, Steam-Ontology.username]
```

Figura 4-11: Visualizzazione delle proprietà dei dati presenti nell'Ontologia

Query d'esempio:

-Lista di Giochi che presentano la categoria 'Classic':

[Steam-Ontology.Grand_Theft_Auto, Steam-Ontology.Half_Life]

-Lista di Giochi che presentano lo sviluppatore 'Valve':

[Steam-Ontology.Half_Life, Steam-Ontology.Portal]

-Lista di Giochi che presentano la piattaforma 'Windows':

[Steam-Ontology.Call_of_Duty, Steam-Ontology.Grand_Theft_Auto, Steam-Ontology.Half_Life, Steam-Ontology.Portal, Steam-Ontology.Tomb_Raider:_Legend]

Figura 4-12: Visualizzazione di qualche query d'esempio

```
print("\nQuery d'esempio:")
print("\n-Lista di Giochi che presentano la categoria 'Classic':\n")
games = ontology.search(is_a = ontology.Game, has_genre = ontology.search(is_a = ontology.Classic))
print(games, "\n")
print("\n-Lista di Giochi che presentano lo sviluppatore 'Valve':\n")
games = ontology.search(is_a = ontology.Game, has_developer = ontology.search(is_a = ontology.Valve))
print(games, "\n")
print("\n-Lista di Giochi che presentano la piattaforma 'Windows':\n")
games = ontology.search(is_a = ontology.Game, has_platform = ontology.search(is_a = ontology.Windows))
print(games, "\n")
```

Figura 4-13: Esempio costruzione del codice per la formulazione delle query

5 CONCLUSIONI

Due possibili estensioni del progetto sono:

- L'ampliamento del dataset con l'aggiunta di un ulteriore dataset riferito ad un'altra grande piattaforma di distribuzioni digitale, Epic Games, e i dati possono essere combinati per ottenere risultati migliori e più precisi sia nella raccomandazione che nell'interrogazione della KB, implementando così l'interoperabilità semantica tra 2 domini simili.
- L'implementazione di una rete bayesiana per la previsione del successo di un gioco appena uscito sulla base di quello dei giochi attualmente presenti nel dataset.

La repository con tutti i file del progetto: https://github.com/Lunalorla/DI-CA_Icon-2021-2022

La lista delle dipendenze del progetto è presente nel file *requirements.txt* caricato sulla repository.

Il lavoro svolto è stato suddiviso in task assegnati ai due componenti del gruppo.