UNIVERSITÀ
DEGLI STUDI DI BARI
ALDO MORO

|Steam Game Recommender|

Componenti: Antonio Graziani,

Matricola: 738723,

email: a.graziani2@studenti.uniba.it

|Repository GitHub|

AA 2024-25

Contenuti

1. Introduzione

Il progetto *Steam Game Recommender* è stato sviluppato con l'obiettivo di realizzare un sistema di raccomandazione personalizzato per giochi Steam. L'approccio adottato integra tecniche simboliche, basate su basi di conoscenza RDF e linguaggio SPARQL, con modelli di apprendimento supervisionato, al fine di analizzare e predire le preferenze di un utente sulla base della sua libreria di giochi.

Il sistema è in grado di operare sia in condizioni normali (recuperando i dati dell'utente dalla Steam Web API), sia utilizzando un meccanismo di fallback basato su cache locale, date eventuali limitazioni di accesso all'API utilizzata.

2. Obiettivi del progetto

L'obiettivo principale è la realizzazione di un sistema ibrido di raccomandazione che, dato un identificativo utente Steam, sia in grado di:

- Analizzare i giochi effettivamente giocati e ricavarne un profilo utente;
- Effettuare raccomandazioni simboliche mediante interrogazioni SPARQL su una KB RDF;
- Addestrare modelli supervisionati per classificare i giochi in "piaciuti" e "non piaciuti";
- Fornire una lista finale di giochi non ancora giocati ma raccomandati secondo entrambi gli approcci.

3. Tecnologie utilizzate

Il progetto è interamente sviluppato in Python e si basa sui seguenti strumenti:

- **Steam Web API** – Per il recupero della libreria giochi dell'utente;
- **pandas, numpy** – Per l'elaborazione dei dati e il preprocessing;
- **rdflib** – Per la costruzione della base di conoscenza RDF e l'esecuzione di query SPARQL;

- **scikit-learn** – Per l'addestramento e la valutazione di modelli di classificazione supervisionati;
- **joblib** – Per il salvataggio e riutilizzo di modelli ed encoder;
- **LibreOffice** – Per la redazione del presente documento tecnico.

4. Struttura del progetto

L'architettura del progetto è modulare, in modo da favorire la leggibilità, la manutenzione e l'estensione futura. Le funzionalità sono suddivise in moduli specifici. Di seguito una descrizione delle principali componenti:

- **`main.py`**
  È lo script principale, posto come entry-point del progetto. Esegue l'estrazione dei dati utente, la costruzione del profilo, l'esecuzione delle raccomandazioni (sia simboliche che supervisionate), l'addestramento dei modelli e la valutazione dei risultati.
- **`data/catalog.csv`**
  Contiene un catalogo di giochi Steam, con informazioni associate quali tag, generi, publisher e data di rilascio. Questo file rappresenta il dominio su cui lavorano sia la base di conoscenza che i modelli supervisionati.
- **`data/steam_user_games.csv`**
  File di cache utilizzato in alternativa all'accesso alla Steam API. Contiene l'elenco dei giochi giocati da un utente, con relativo tempo di gioco espresso in ore.
- **`kb/steam_kb.ttl`**
  È la base di conoscenza RDF in formato Turtle. Ogni gioco è descritto come istanza della classe *`Game`* e associato a proprietà come *`hasTag`*, *`hasGenre`*, *`hasPublisher`* e *`releaseDate`*. Questa struttura permette interrogazioni semantiche per individuare giochi simili a quelli preferiti dall'utente.
- **`output/models/`**
  Directory contenente i modelli supervisionati addestrati (`.pkl`) e i relativi encoder multilabel (`MultiLabelBinarizer`). Questi oggetti vengono salvati per evitare di ripetere l'addestramento a ogni esecuzione e per garantire coerenza nelle feature.

- **`src/extract_user_profile.py`**
  Modulo dedicato alla costruzione del profilo utente. Analizza i giochi giocati e determina i tag, generi, publisher e anni più rappresentativi, che verranno poi utilizzati per generare le raccomandazioni simboliche.
- **`src/recommend_games.py`**
  Contiene la logica per l'esecuzione delle query SPARQL a partire dal profilo utente. Le query sono generate dinamicamente in base agli attributi più rilevanti dell'utente e applicate alla KB RDF.
- **`src/models_trainer.py`**
  Gestisce l'addestramento dei modelli di classificazione supervisionata. I giochi vengono etichettati come "piaciuti" se giocati per almeno 8 ore, e come "non piaciuti" altrimenti. I dati vengono binarizzati e utilizzati per addestrare modelli come Random Forest, Decision Tree, Naive Bayes e Logistic Regression.
- **`src/evaluation.py`**
  Modulo per la valutazione automatica dei modelli supervisionati (mediante f1-score, precision, recall) e per l'applicazione delle predizioni su giochi non ancora giocati.

Questa organizzazione riflette la volontà di mantenere il codice separato per responsabilità e favorisce il riutilizzo delle componenti.

5. Flusso di esecuzione

Il flusso di esecuzione del progetto segue una sequenza ben definita di fasi, ognuna delle quali contribuisce alla generazione delle raccomandazioni finali.

1. ***Recupero dei giochi giocati dall'utente***

   Il sistema accede alla Steam Web API per recuperare la lista dei giochi giocati da un determinato utente, identificato dal proprio Steam ID. Per ogni gioco viene considerato il tempo totale di gioco in ore.

```python
# Estraggo i giochi dell'utente
print("Scarico giochi utente da Steam API...")
try:
    # Provo a connettermi al profilo utente
    user_games = get_user_games(STEAM_ID)
    if user_games.empty:
        raise Exception()

    # Salvo i giochi dell'utente in un csv
    user_games.to_csv(USER_DATA_PATH, index=False)
except:
    # Nel caso la connessione non vada a buon fine, uso la cache locale
    print("Uso la cache locale per i giochi utente.")
    user_games = pd.read_csv(USER_DATA_PATH)
```

   Se la chiamata API fallisce (es. per limiti di richiesta HTTP 429), il sistema utilizza un file CSV di cache per continuare l'esecuzione.

2. ***Costruzione del profilo utente***

   Il profilo dell'utente viene costruito a partire dai giochi giocati per almeno 8 ore. Per questi giochi, il sistema identifica:

   - I tag più ricorrenti (es. "Singleplayer", "Action");
   - I generi prevalenti (es. "Adventure", "RPG");
   - I publisher più frequenti (es. "Valve", "Bandai Namco");
   - Gli anni di uscita più rappresentati.

```python
# Carico il catalogo giochi
catalog = pd.read_csv(CATALOG_PATH)

# Costruisco il profilo dell'utente e prendo i giochi da lui giocati
print("Costruisco profilo utente...")
profile = build_user_profile(user_games, catalog)
played_ids = user_games["game_id"].tolist()
```

   La costruzione viene effettuata tramite la funzione *build_user_profile()* definita nel file `extract_user_profile.py` e le informazioni vengono raccolte in un dizionare che funge da descrizione semantica del profilo

utente; inoltre, vengono anche memorizzati i giochi giocati dall'utente per escluderli durante la fase di raccomandazione.

3. *Raccomandazioni simboliche (SPARQL su RDF)*

l profilo viene usato per costruire dinamicamente una serie di query SPARQL. Ogni query cerca giochi nella KB RDF che condividano almeno un tag, un genere, un publisher o l'anno di rilascio.

```
# Raccomando giochi usando la KB RDF
print("Genero raccomandazioni dalla KB RDF...")
recomm = recommending(profile, played_ids, kb_path=KB_PATH)

# Costruisco un DataFrame con le prime 15 raccomandazioni
recommendation = pd.DataFrame(recomm[:15])
if recommendation.empty:
    print("Nessuna raccomandazione trovata.")
    exit()
```

```
print("\nTop giochi consigliati:")
for _, row in recommendation.iterrows():
    print(f"- {row['name']} (score: {row['score']})")
```

I giochi trovati vengono aggregati e ordinati per frequenza di apparizione (score), per poi fornire una prima lista di 15 raccomandazioni simboliche.

4. *Addestramento modelli supervisionati*

Parallelamente, il sistema prepara un dataset supervisionato etichettando i giochi in "piaciuti" (≥ 8 ore) e "non piaciuti". Tag e generi vengono convertiti in vettori binari mediante encoder `MultiLabelBinarizer`. Il dataset così preparato viene usato per addestrare diversi modelli (Random Forest, Naive Bayes, Decision Tree, Logistic Regression), salvati in formato `.pkl` per usi futuri.

```
train_models(USER_DATA_PATH, CATALOG_PATH, MODELS_DIR)
```

Tutto effettuato nella funzione *train_models()* di `models_trainer.py`.

5. *Valutazione modelli supervisionati*

I modelli vengono valutati tramite validazione incrociata, producendo metriche di precision, recall, accuracy e f1-score. Questo consente di confrontare le performance dei modelli e selezionare quelli più affidabili.

```
evaluate_models(USER_DATA_PATH, CATALOG_PATH, MODELS_DIR)
```

Tutto effettuato nella funzione *evaluate_models()* di `evaluation.py`.

6. ***Predizione finale sui giochi non giocati***

   Infine, viene applicato il modello supervisionato più performante sui giochi presenti nel catalogo ma non ancora giocati dall'utente. Le predizioni permettono di identificare ulteriori giochi potenzialmente apprezzati, che completano la lista di raccomandazioni.

```
# Raccomando giochi tramite modello supervisionato (RandomForest)
model_path = os.path.join(MODELS_DIR, "RandomForest.pkl")
predict_liked_games(model_path, catalog, played_ids)
```

6. Raccomandazione simbolica (SPARQL + RDF)

Una parte centrale del progetto consiste nell'utilizzo di tecniche simboliche per fornire raccomandazioni.
In particolare, è stata sviluppata una **base di conoscenza** contenente giochi Steam descritti attraverso informazioni come i loro tag, i generi di appartenenza, il publisher e l'anno di pubblicazione.
Questa base è rappresentata nel formato RDF (Resource Description Framework) e salvata in un file `.ttl` (Turtle).

Ogni gioco nella knowledge base è identificato come una risorsa di tipo `Game` e viene descritto con proprietà che specificano:

- i **tag** associati (es. "Multiplayer", "Atmospheric")
- i **generi** (es. "Action", "Adventure")
- il **publisher** (es. "Valve", "Bandai Namco")
- la **data di rilascio** (in formato anno, es. "2000")

Esempio semplificato di una voce nella KB:

```
steam:10 a steam:Game ;
    steam:hasTag steam:Multiplayer ;
    steam:hasGenre steam:Action ;
    steam:publishedBy steam:Valve ;
    steam:releaseDate "2000"
```

Costruzione delle query SPARQL

Una volta costruito il profilo dell'utente, il sistema genera una serie di query SPARQL in modo automatico, basandosi sulle caratteristiche principali dei giochi che l'utente ha giocato a lungo (almeno 8 ore).
Ogni query è pensata per trovare giochi simili a quelli apprezzati.

```
# Creo la lista delle query SPARQL da eseguire
query_templates = []

# 1. Query sui tag preferiti
for tag in profile.get("top_tags", []):
    uri = to_uri(tag)
    query_templates.append(f"""
    SELECT ?game WHERE {{
        ?game steam:hasTag <http://example.org/steam#{uri}> .
    }}
    """)

# 2. Query sui generi preferiti
for genre in profile.get("top_genres", []):
    uri = to_uri(genre)
    query_templates.append(f"""
    SELECT ?game WHERE {{
        ?game steam:hasGenre <http://example.org/steam#{uri}> .
    }}
    """)

# 3. Query sui publisher preferiti
for pub in profile.get("top_publishers", []):
    uri = to_uri(pub)
    query_templates.append(f"""
    SELECT ?game WHERE {{
        ?game steam:publishedBy <http://example.org/steam#{uri}> .
    }}
    """)

# 4. Query per gli anni di uscita più frequenti
for year in profile.get("top_years", []):
    query_templates.append(f"""
    SELECT ?game WHERE {{
        ?game steam:releaseDate "{year}" .
    }}
    """)
```

Ad esempio, se il profilo dell'utente include tra i tag preferiti "Action" e "Multiplayer", verranno generate due query come queste:

```
SELECT ?game WHERE {
    ?game steam:hasTag steam:Action .
}

SELECT ?game WHERE {
    ?game steam:hasGenre steam:Adventure .
}
```

```python
# Funzione per trasformare un testo in una URI RDF valida
def to_uri(value):
    return value.replace(" ", "_").replace("-", "_").replace("'", "").replace(",", "").replace("&", "and")
```

E' stata anche definita una funzione per facilitare la conversione

Aggregazione dei risultati

Tutte le query vengono eseguite sulla base di conoscenza RDF usando la libreria `rdflib`.
I giochi che compaiono in più query (es. condividono più caratteristiche con il profilo utente) ricevono un punteggio maggiore.
Alla fine, i giochi vengono ordinati in base al numero di "match" e viene proposta una lista personalizzata, escludendo quelli che l'utente possiede già.

```python
# Inizializzo il dizionario dei risultati
results = defaultdict(int)

# Eseguo tutte le query SPARQL generate
for query in query_templates:
    # Eseguo la query sul grafo g, con il prefisso steam
    res = g.query(query, initNs={"steam": STEAM})

    # Per ogni riga dei risultati della query
    for row in res:
        # Converto l'uri in una stringa
        game_uri = str(row[0])

        # Estraggo l'ID del gioco e se non si trova nei già giocati
        # incremento il suo score di 1
        game_id = game_uri.split("#")[-1]
        if game_id not in played_ids:
            results[game_id] += 1

# Avviso in caso di nessun risultato trovato
if not results:
    print("Nessuna raccomandazione trovata.")
```

```python
# Ordino i risultati per punteggio decrescente
ranked = sorted(results.items(), key=lambda x: x[1], reverse=True)

# Ritorno la lista finale dei giochi raccomandati
return [{"game_id": game_id, "score": score} for game_id, score in ranked]
```

Esempio di raccomandazioni simboliche

```
Top giochi consigliati:
- Yu-Gi-Oh! Master Duel (score: 5)
- METAL GEAR RISING: REVENGEANCE (score: 5)
- METAL GEAR SOLID V: THE PHANTOM PAIN (score: 5)
- METAL GEAR SURVIVE (score: 4)
- Sanctum 2 (score: 4)
- Satisfactory (score: 3)
- Deep Rock Galactic (score: 3)
- Valheim (score: 3)
- Sanctum (score: 3)
```

Questo metodo ha il vantaggio di essere interpretabile: possiamo sapere esattamente *perché* un gioco è stato consigliato (condivide tag, genere, ecc.).

7. Raccomandazione supervisionata (Machine Learning)

Accanto all'approccio simbolico, il sistema include anche una modalità di raccomandazione basata sull'apprendimento supervisionato.
L'obiettivo è quello di addestrare modelli di classificazione in grado di predire se un gioco potrà piacere all'utente, sulla base delle sue caratteristiche principali (tag e generi), anche se non è mai stato giocato.

Etichettatura dei dati

Per addestrare un modello supervisionato è necessario disporre di dati etichettati. L'etichetta è stata assegnata sulla base di una scelta progettuale ragionata:

- un gioco viene considerato *"piaciuto"* se è stato giocato per almeno 8 ore; altrimenti è "*non piaciuto*".

Questa soglia è stata scelta come compromesso tra la necessità di avere dati significativi e l'evitare falsi positivi legati a prove brevi o casuali. Per aver giocato un gioco almeno 8 ore, vuol dire che ormai si è andati oltre la prova iniziale e si reputa il gioco di proprio gradimento.

Da questo, si costruisce un dataset di esempio in cui ogni riga rappresenta un gioco, accompagnato dalla sua etichetta binaria (0 o 1).

Preprocessing: da testo a numeri

I dati relativi a **tag** e **generi** sono originariamente contenuti come stringhe testuali separate da virgole (es. `"Action, Adventure"`).
Per poterli usare in un modello di machine learning, devono essere trasformati in una rappresentazione numerica.
A questo scopo viene utilizzato `MultiLabelBinarizer`, una tecnica che:

- converte ogni tag e ogni genere in una colonna binaria;
- per ogni gioco, indica con 1 la presenza di un certo tag/genere, e con 0 l'assenza.

```
# Creo i vettori numerici binari per tag e generi
binary_tags = MultiLabelBinarizer()
binary_genres = MultiLabelBinarizer()

# Trasformo le liste di tag e generi in vettori binari
X_tags = binary_tags.fit_transform(data["tag_list"])
X_genres = binary_genres.fit_transform(data["genre_list"])
```

Ad esempio:

```
Input: ["Action", "RPG"]
Output: [1, 0, 0, 1, 0, ...]  ← dove ogni colonna è
un possibile tag o genere
```

Il risultato finale è una matrice numerica che può essere data in pasto ai modelli supervisionati.

Addestramento dei modelli

Una volta preparato il dataset, vengono addestrati quattro diversi modelli di classificazione:

- **Random Forest**
- **Decision Tree**
- **Logistic Regression**
- **Naive Bayes**

```
# Definisco i modelli supervisionati
models = [
    ("DecisionTree", DecisionTreeClassifier(max_depth=5, class_weight='balanced', random_state=42)),
    ("RandomForest", RandomForestClassifier(n_estimators=100, class_weight='balanced', random_state=42)),
    ("LogisticRegression", LogisticRegression(max_iter=1000, class_weight='balanced')),
    ("NaiveBayes", BernoulliNB())
]

# Addestramento, valutazione e salvataggio dei modelli
for name, classifier in models:

    # Addestra il modello sul dataset
    classifier.fit(X, y)

    # Salvo il modello in formato .pkl
    joblib.dump(classifier, os.path.join(models_dir, f"{name}.pkl"))
```

L'addestramento viene eseguito in modo automatico, e i modelli vengono salvati su disco per poter essere riutilizzati in futuro senza dover ripetere il processo.

```
joblib.dump(binary_tags, os.path.join(models_dir, f"binary_tags.pkl"))
joblib.dump(binary_genres, os.path.join(models_dir, f"binary_genres.pkl"))
print("\n Tutti i modelli sono stati salvati in output/models/")
```

Durante il training, vengono anche salvati i binarizzatori utilizzati, così da poter codificare i dati di test nello stesso modo.

Valutazione delle performance

I modelli vengono valutati in base alla loro capacità di distinguere i giochi piaciuti da quelli non piaciuti.
Vengono calcolate le metriche più comuni:

- **accuracy** (percentuale di predizioni corrette),
- **precision** (percentuale di predizioni positive corrette),
- **recall** (percentuale di giochi realmente piaciuti identificati),

- **f1-score** (media armonica tra precision e recall, utile per valutare il bilanciamento tra i due).

Esempio di output per un modello:

```
RandomForest:
              precision    recall  f1-score   support

           0       1.00      0.96      0.98        27
           1       0.93      1.00      0.97        14

    accuracy                           0.98        41
   macro avg       0.97      0.98      0.97        41
weighted avg       0.98      0.98      0.98        41
```

Predizione su giochi non ancora giocati

Dopo la valutazione, il sistema applica il modello supervisionato più performante (di solito Random Forest) per analizzare i giochi non ancora giocati dall'utente.

```
# Effettuo la predizione del modello
predictions = model.predict(X_unseen)
not_played["predicted"] = predictions

# Prendo solo i giochi piaciuti "1" dalla predizione e stampo i primi 10
liked = not_played[not_played["predicted"] == 1]
print("\nGiochi raccomandati dal modello supervisionato:")
print(liked[["game_id", "name"]].head(10).to_string(index=False))
```

I dati di questi giochi vengono trasformati esattamente come in fase di addestramento, e il modello viene usato per predire se potrebbero piacere o meno.

```
# Carico
binary_tags = joblib.load(os.path.join(os.path.dirname(model_path), "binary_tags.pkl"))
binary_genres = joblib.load(os.path.join(os.path.dirname(model_path), "binary_genres.pkl"))

# Prendo i giochi nel catalogo che non sono tra i "giocati"
not_played = catalog[~catalog["game_id"].isin(played_ids)].copy()

# Converto le stringhe tipo "Action, RPG, Indie" in liste ["Action", "RPG", "Indie"]
not_played["tag_list"] = not_played["tags"].fillna("").apply(lambda s: s.split(", "))
not_played["genre_list"] = not_played["genres"].fillna("").apply(lambda s: s.split(", "))

# Converto le liste di tag e generi in vettori binari
# es. ["Action", "Multiplayer"] → [1, 1, 0, 0, 0, ...]
X_tags = binary_tags.transform(not_played["tag_list"])
X_genres = binary_genres.transform(not_played["genre_list"])

# Unisco i due vettori verticalmente, rendendoli colonne in una matrice
# dove le righe sono i giochi
X_unseen = np.hstack([X_tags, X_genres])
```

Vengono infine mostrati i titoli che risultano più promettenti secondo il modello.

Esempio di output:

```
 game_id                          name
  200210   Realm of the Mad God Exalt
  238960                Path of Exile
  211820                    Starbound
  268420                 Aura Kingdom
  322330        Don't Starve Together
  372360           Tales of Symphonia
  351970            Tales of Zestiria
```

8. Confronto tra approcci

Il progetto *Steam Game Recommender* integra due strategie differenti per la raccomandazione di giochi: *l'approccio simbolico*, che sfrutta una base di conoscenza RDF interrogabile via SPARQL, e *l'approccio supervisionato*, che utilizza modelli di apprendimento automatico per prevedere i giochi che potrebbero piacere a un utente.

Ognuno dei due approcci ha caratteristiche, vantaggi e limiti specifici, che li rendono adatti a situazioni diverse. Il loro confronto permette di apprezzarne la complementarità.

Approccio simbolico: conoscenza esplicita

Questo metodo si basa su una rappresentazione strutturata e semantica dei giochi (RDF), in cui ogni titolo è descritto tramite proprietà come tag, genere, publisher e anno di uscita.

A partire dal profilo dell'utente, vengono generate automaticamente query SPARQL che cercano giochi con attributi simili a quelli che l'utente ha già apprezzato.

Vantaggi principali:

- È interpretabile: è facile capire perché un gioco è stato raccomandato.
- È immediato: non richiede addestramento né dati etichettati.
- È adatto a utenti nuovi: funziona anche con pochi dati disponibili.

Limiti principali:

- Attribuisce peso uniforme a tutte le caratteristiche: non tiene conto dell'importanza relativa di un tag o di un genere.
- Non rileva pattern impliciti o combinazioni non ovvie di caratteristiche.
- Dipende dalla qualità e dalla completezza della base di conoscenza.

Approccio supervisionato: apprendere dai dati

In questo caso, il sistema costruisce un dataset etichettato automaticamente (ad esempio, considerando "piaciuti" i giochi con almeno 8 ore di utilizzo) e addestra modelli di classificazione per riconoscere le caratteristiche che contraddistinguono i giochi apprezzati dall'utente.

Le informazioni testuali, come tag e generi, vengono trasformate in vettori numerici binari mediante l'uso di encoder. I modelli supervisionati apprendono dai dati e successivamente sono in grado di fornire predizioni anche per giochi non ancora giocati.

Vantaggi principali:

- È in grado di apprendere pattern complessi e non banali.
- Si adatta in modo preciso ai gusti individuali dell'utente.
- Fornisce metriche quantitative che permettono di confrontare oggettivamente i modelli (accuracy, f1-score, ecc.).

Limiti principali:

- Richiede una fase di addestramento e di preprocessing dei dati.
- È meno interpretabile: non sempre è evidente il motivo della raccomandazione.
- La qualità delle predizioni dipende fortemente dai dati a disposizione.

Integrazione nel progetto

Nel progetto entrambe le strategie sono state integrate. Il sistema utilizza inizialmente l'approccio simbolico per ottenere un primo insieme di raccomandazioni basato su similarità esplicite, come tag o generi in comune. In

seguito, applica modelli supervisionati per prevedere ulteriori giochi che potrebbero piacere all'utente, analizzando quelli ancora non giocati.

Questi due approcci, pur diversi per logica e funzionamento, si integrano efficacemente all'interno del sistema di raccomandazione, offrendo vantaggi complementari:

- L'approccio simbolico si rivela particolarmente utile quando il sistema ha a disposizione pochi dati (probabilmente dovuto ad un'utente che ha giocato poco) e l'obiettivo principale è offrire suggerimenti interpretabili e coerenti con le preferenze esplicite dell'utente;
- Il supervisionato, al contrario, dà il meglio di sé quando si dispone di una base più solida di giochi già giocati, potendo così apprendere pattern più complessi e offrire suggerimenti più raffinati e personalizzati.

La combinazione dei due approcci consente al sistema di adattarsi dinamicamente a diverse situazioni: da utenti con poca attività registrata (dove il simbolico garantisce raccomandazioni affidabili), a utenti con un profilo più ricco (dove il supervisionato può generare suggerimenti più intelligenti e meno ovvi).

9. Gestione degli errori

Nel corso dello sviluppo del sistema di raccomandazione, è stato necessario prevedere e gestire una serie di situazioni di errore o di malfunzionamento che potevano compromettere l'esperienza utente o la stabilità del progetto. La gestione degli errori è stata implementata privilegiando messaggi chiari e comportamenti alternativi che permettessero all'esecuzione di proseguire, ove possibile.

1. Errori nella chiamata all'API di Steam

La prima fase del sistema prevede il download della libreria di giochi dell'utente tramite l'API di Steam. In alcuni casi, la chiamata può fallire per cause esterne:

- **Errore 429 – Too Many Requests;**
- **Errore di rete o assenza di connessione;**
- **Steam ID non valido o profilo privato**.

In questi casi, il sistema intercetta l'errore e informa l'utente con un messaggio esplicito. Se i dati sono già stati scaricati in precedenza, viene automaticamente utilizzata una **copia locale salvata in cache**, in modo da non interrompere il flusso.

```
# Estraggo i giochi dell'utente
print("Scarico giochi utente da Steam API...")
try:
    # Provo a connettermi al profilo utente
    user_games = get_user_games(STEAM_ID)
    if user_games.empty:
        raise Exception()
```

1.1 Approfondimento del comportamento dell'API Steam

L'errore HTTP 429 – *Too Many Requests* è generato dai server di Steam quando un client (come questo sistema) effettua un numero eccessivo di richieste HTTP in un intervallo di tempo troppo breve. È una misura di protezione automatica contro l'abuso delle API, e non esistono specifiche pubbliche esatte sul numero massimo di richieste consentite.

Dal comportamento osservato durante lo sviluppo, si è notato che:

- Anche *una singola esecuzione del programma* può talvolta generare il 429, specialmente se effettuata poco dopo altre esecuzioni.
- L'errore *persiste nel tempo*, anche a distanza di ore, suggerendo che il blocco applicato da Steam potrebbe avere durata variabile, spesso superiore ai 10-15 minuti.
- L'errore *non dipende solo dall'API Key*, ma anche dall'indirizzo IP da cui proviene la richiesta: cambiare IP (es. tramite VPN) può in alcuni casi evitare temporaneamente il blocco.
- Steam *non fornisce header informativi* (es. `Retry-After`) nella risposta, rendendo impossibile sapere quando riprovare.

Per gestire il problema, il sistema è previsto l'uso di una *cache locale* dei dati utente: se l'errore 429 si verifica, e se è già stato eseguito almeno un download in precedenza, i dati vengono ricaricati dal file CSV salvato. Questo compromesso evita l'interruzione dell'esecuzione e garantisce una certa continuità operativa, pur in presenza di limitazioni esterne.

2. Profilo utente non disponibile o vuoto

Se la libreria dell'utente non contiene giochi oppure nessuno dei giochi supera la soglia minima per essere considerato "piaciuto" (es. 8 ore di gioco), il profilo risulta insufficiente per generare raccomandazioni.

In questo caso, il sistema:

- stampa un messaggio di errore ("Uso la cache locale per i giochi utente.")
- leggere la cache locale per prendere i giochi dell'utente.

```
except:
    # Nel caso la connessione non vada a buon fine, uso la cache locale
    print("Uso la cache locale per i giochi utente.")
    user_games = pd.read_csv(USER_DATA_PATH)
```

3. Assenza di raccomandazioni simboliche

Può accadere che, pur avendo un profilo valido, le query SPARQL generate non restituiscano alcun risultato dalla base di conoscenza RDF. Questo può essere dovuto a un numero troppo ristretto di giochi nella KB o a una bassa corrispondenza con i gusti dell'utente.

Il sistema intercetta il caso in cui la lista di raccomandazioni simboliche è vuota (DataFrame.empty) e lo comunica chiaramente all'utente, interrompendo l'esecuzione senza errori tecnici.

```
# Costruisco un DataFrame con le prime 15 raccomandazioni
recommendation = pd.DataFrame(recomm[:15])
if recommendation.empty:
    print("Nessuna raccomandazione trovata.")
    exit()
```

4. Incoerenza tra modello e dati (supervisionato)

Durante la valutazione dei modelli supervisionati o nella fase di predizione, possono verificarsi errori se:

- i dati di test contengono *feature non viste in fase di addestramento* (es. tag nuovi);
- il numero di feature nei dati attuali non coincide con quello atteso dal modello.

Per prevenire questo problema, è stato introdotto un meccanismo che salva e ricarica anche gli *encoder utilizzati per i tag e i generi*, garantendo coerenza tra l'addestramento e la predizione. In questo modo, i modelli supervisionati operano sempre su dati con la stessa struttura.

```
joblib.dump(binary_tags, os.path.join(models_dir, f"binary_tags.pkl"))
joblib.dump(binary_genres, os.path.join(models_dir, f"binary_genres.pkl"))
print("\n Tutti i modelli sono stati salvati in output/models/")
```

5. Mancanza di modelli addestrati

Se nella directory dei modelli (output/models/) non sono presenti i file salvati, il sistema procede automaticamente con l'addestramento dei quattro modelli supervisionati, salvandoli per usi futuri. Questo evita errori di caricamento e rende il sistema *autonomo nella preparazione*.

```
# Addestramento, valutazione e salvataggio dei modelli
for name, classifier in models:

    # Addestra il modello sul dataset
    classifier.fit(X, y)

    # Salvo il modello in formato .pkl
    joblib.dump(classifier, os.path.join(models_dir, f"{name}.pkl"))
```

6. Warning non bloccanti

In alcune fasi, in particolare durante l'uso degli encoder, possono comparire *warning informativi* (es. tag sconosciuti che vengono ignorati). Questi messaggi non interrompono l'esecuzione e sono stati resi invisibili per chiarezza dell'output; possono essere resi visbili facilmente commentando la seguente riga nel file `evaluation.py` alla riga 8.

```
warnings.filterwarnings("ignore", category=UserWarning, module="sklearn.preprocessing._label")
```

10. Conclusioni

Il progetto *Steam Game Recommender* ha rappresentato un esercizio completo di integrazione tra conoscenza strutturata e apprendimento automatico, applicato al problema concreto della raccomandazione personalizzata di giochi.

A partire da una semplice interazione con l'API di Steam, è stato possibile costruire un sistema capace di:

- analizzare automaticamente la libreria utente;
- generare un profilo personalizzato basato su tag, generi, publisher e anno di uscita;
- interrogare una base di conoscenza RDF tramite SPARQL per ottenere suggerimenti immediati;
- addestrare e valutare modelli supervisionati capaci di generalizzare preferenze implicite;
- produrre raccomandazioni su giochi mai provati con buona accuratezza.

L'approccio simbolico ha garantito un sistema semplice, interpretabile e utilizzabile anche con pochi dati, mentre quello supervisionato ha introdotto un livello di personalizzazione più sofisticato, in grado di riconoscere pattern ricorrenti nei giochi apprezzati, anche in assenza di regole esplicite.
L'integrazione dei due metodi ha dimostrato di essere efficace nel costruire un sistema in grado di adattarsi a diversi scenari: utenti con una cronologia di gioco ampia, ma anche utenti con profili più limitati. Inoltre, è stata posta particolare attenzione alla gestione degli errori e alla modularizzazione del codice.

Nel complesso, il progetto ha rappresentato un'applicazione concreta e ben strutturata dei concetti affrontati nel corso, mostrando come tecnologie simboliche e tecniche di machine learning possano convivere all'interno di uno stesso sistema, migliorandone l'efficacia e la flessibilità.

Risultati dei modelli supervisionati

Di seguito vengono riportati i risultati dei quattro modelli supervisionati addestrati sul dataset dell'utente. Per ciascun modello vengono mostrate le metriche principali: precision, recall e f1-score, suddivise per le due classi (0 = non piaciuto, 1 = piaciuto), insieme all'accuracy generale.

Random Forest

Classe	Precision	Recall	F1-score	N.Dati
0	1.00	0	0	27
1	0.93	0	0	14
Accuracy			0	41
Macro avg	0.97	0	0	41
Weighted	0.98	0	0	41

Logistic Regression

Classe	Precision	Recall	F1-score	N.Dati
0	0.93	0	0	27
1	0.92	0	0	14
Accuracy			0	41
Macro avg	0.93	0	0	41
Weighted	0.93	0	0	41

Decision Tree

Classe	Precision	Recall	F1-score	N.Dati
0	0.81	0	0	27
1	0.80	0	0	14
Accuracy			0	41
Macro avg	0.80	0	0	41
Weighted	0.80	0	0	41

Naive Bayes

Classe	Precision	Recall	F1-score	N.Dati
0	0.92	0	0	27
1	0.80	0	0	14
Accuracy			0	41
Macro avg	0.86	0	0	41
Weighted	0.88	0	0	41

- Il **Random Forest** è il modello che ha raggiunto le performance migliori su tutte le metriche, risultando particolarmente affidabile sia nella precisione che nella capacità di individuare correttamente i giochi piaciuti.
- La **Logistic Regression** si comporta in modo bilanciato e robusto, mantenendo ottimi valori di precision e recall.
- Il **Decision Tree** presenta il comportamento meno equilibrato, con una precisione accettabile ma una recall bassa sulla classe positiva, indicando una maggiore difficoltà nel riconoscere i giochi preferiti.
- Il **Naive Bayes** ha prestazioni buone, ma inferiori rispetto a Random Forest, soprattutto nella classe minoritaria (piaciuto).

Questi risultati confermano che il modello *Random Forest* è il più adatto tra quelli testati per effettuare raccomandazioni supervisionate in questo contesto.

11. Sviluppi futuri

Sebbene il progetto abbia raggiunto pienamente l'obiettivo di fornire raccomandazioni personalizzate a partire dalla libreria Steam di un utente, esistono diverse possibilità di estensione e miglioramento del sistema:

- *Arricchimento della base di conoscenza RDF*
  Attualmente la KB semantica contiene una selezione limitata di giochi e proprietà. Estenderla automaticamente tramite crawling o scraping da

Steam o Wikidata permetterebbe un ragionamento simbolico più ricco e preciso.

- *Miglioramento del profilo utente*
  Il profilo è basato su soglie semplici (es. 8 ore di gioco). Un raffinamento basato su tecniche di clustering o analisi latente delle preferenze potrebbe individuare gusti meno evidenti ma significativi.
- *Interfaccia grafica (GUI)*
  Integrare il sistema con un'interfaccia desktop o web-friendly renderebbe l'esperienza utente più accessibile anche per chi non ha competenze tecniche.

12. Riferimenti

- Steam Web API – https://developer.valvesoftware.com/wiki/Steam_Web_API
- RDFLib documentation – https://rdflib.readthedocs.io/
- SPARQL 1.1 Query Language – W3C Recommendation https://www.w3.org/TR/sparql11-query/
- scikit-learn documentation – https://scikit-learn.org/stable/
- pandas documentation – https://pandas.pydata.org/
- NumPy documentation – https://numpy.org/doc/
- joblib documentation – https://joblib.readthedocs.io/
- Turtle – Terse RDF Triple Language (W3C) https://www.w3.org/TR/turtle/
- KB RDF: D. Poole, A. Mackworth – Artificial Intelligence: Foundations of Computational Agents, Cambridge University Press, 3rd Ed., 2023 [Cap. 16]
- Apprendimento supervisionato: D. Poole, A. Mackworth – Artificial Intelligence: Foundations of Computational Agents, Cambridge University Press, 3rd Ed., 2023 [Cap. 7]