Capitolo 5: Raccomandazioni e Mood

Introduzione

Il sistema di raccomandazione di CineMatch rappresenta il cuore dell'applicazione, fornendo suggerimenti personalizzati basati sui gusti cinematografici dell'utente. L'architettura implementata si basa su tecniche avanzate di Natural Language Processing (NLP) e similarity search tramite FAISS (Facebook AI Similarity Search), garantendo raccomandazioni accurate ed efficienti.

Il sistema si compone di cinque componenti principali:

1. **Generazione degli embedding** tramite modelli transformer pre-addestrati
2. **Costruzione dell'indice FAISS** per ricerche di similarità veloci
3. **Servizio di raccomandazione** che combina similarity search e re-ranking intelligente
4. **Aggiornamento incrementale** per mantenere il sistema sincronizzato con nuovi contenuti
5. **Sistema mood-based** per suggerimenti basati sullo stato d'animo dell'utente

5.1 Generazione degli Embedding

5.1.1 Architettura

Il file [bge_embedding_generator.py](file:///C:/Users/Utente/Desktop/Esame-Fenza/backend/embedding_and_faiss/bge_embedding_generator.py) implementa la pipeline di generazione degli embedding vettoriali per i film presenti nel catalogo MongoDB.

Il sistema utilizza il modello **multilingual-e5-base** di Sentence Transformers, che offre un ottimo compromesso tra qualità e dimensioni (768 dimensioni, ~1.1GB). Questo modello è particolarmente adatto per testi multilingua e produce rappresentazioni vettoriali dense e semanticamente significative.

5.1.2 Pipeline di Processamento

Il processo di generazione si articola in sei fasi:

Fase 1: Caricamento del Modello

```
model = SentenceTransformer(BERT_MODEL)
# BERT_MODEL = "intfloat/multilingual-e5-base"
```

Il modello viene scaricato automaticamente al primo utilizzo e successivamente caricato dalla cache locale.

Fase 2: Estrazione Dati da MongoDB

I film vengono estratti dalla collezione `movies_catalog` filtrando solo i documenti con `imdb_id` valido:

```
cursor = client[DB_NAME][COLL_INPUT].find(
    {"imdb_id": {"$exists": True, "$ne": None, "$ne": ""}},
```

```
    {"imdb_id": 1, "title": 1, "description": 1, "genres": 1,
     "actors": 1, "director": 1, "_id": 0}
)
```

Fase 3: Preprocessing e Vocabolari

Il sistema costruisce vocabolari limitati per attori e registi per evitare sparsità eccessiva negli embedding:

- **Generi**: tutti i generi presenti vengono mantenuti
- **Attori**: vengono mantenuti i top 3000 attori più frequenti
- **Registi**: vengono mantenuti i top 1000 registi più frequenti

Questo approccio riduce la dimensionalità preservando le informazioni più rilevanti.

Fase 4: Multi-Hot Encoding

Per generi, attori e registi viene applicato un encoding multi-hot tramite `MultiLabelBinarizer` di scikit-learn:

```
mlb_genre = MultiLabelBinarizer(classes=sorted(all_genres))
genre_enc = mlb_genre.fit_transform([m["genres_list"] for m in movies_data])
```

Questo produce vettori binari sparsi dove ogni posizione rappresenta la presenza (1) o assenza (0) di un determinato elemento.

Fase 5: Generazione Embedding Semantici

Le descrizioni dei film vengono codificate utilizzando il modello E5:

```
desc_emb = model.encode(
    descriptions,
    batch_size=BATCH_SIZE,
    show_progress_bar=True,
    convert_to_numpy=True,
    normalize_embeddings=True
)
```

Le caratteristiche principali:

- Elaborazione in batch per efficienza
- Normalizzazione L2 automatica
- Conversione diretta in array NumPy

Fase 6: Persistenza su MongoDB

Ogni film viene salvato nella collezione `bert_movies_vectors_bge` con quattro tipologie di embedding:

```
{
    "imdb_id": "tt0111161",
    "title": "The Shawshank Redemption",
    "embedding_description": [0.123, -0.456, ...],  # 768 dim
    "embedding_genre": [1, 0, 1, ...],              # N generi
    "embedding_actors": [0, 1, 0, ...],             # 3000 dim
    "embedding_director": [0, 0, 1, ...],           # 1000 dim
    "genres_list": ["drama"],
    "actors_list": ["tim robbins", "morgan freeman"],
    "director_list": ["frank darabont"]
}
```

5.1.3 Caratteristiche Tecniche

Parametro	Valore
Modello	intfloat/multilingual-e5-base
Dimensione embedding descrizione	768
Batch size	32
Max attori	3000
Max registi	1000
Collezione output	bert_movies_vectors_bge

5.2 Costruzione dell'Indice FAISS

5.2.1 Obiettivo

Il file [faiss_index_builder_bge.py](file:///C:/Users/Utente/Desktop/Esame-Fenza/backend/embedding_and_faiss/faiss_index_builder_bge.py) costruisce un indice FAISS ottimizzato per eseguire ricerche di similarità in modo efficiente su decine di migliaia di vettori.

5.2.2 Processo di Costruzione

Step 1: Caricamento Embedding

Gli embedding vengono caricati dalla collezione MongoDB `bert_movies_vectors_bge`:

```
cursor = coll.find({}, {
    "imdb_id": 1,
    "embedding_description": 1,
    "_id": 0
})
```

Vengono utilizzati solo gli embedding delle descrizioni (768 dimensioni) per la ricerca di similarità principale.

Step 2: Arricchimento con Titoli

Il sistema carica i titoli originali dalla collezione `movies_catalog` per creare un mapping completo:

```
catalog_cursor = client[DB_NAME]["movies_catalog"].find(
    {},
    {"imdb_id": 1, "original_title": 1, "title": 1, "_id": 0}
)
```

Questo permette di associare ogni vettore al film corrispondente.

Step 3: Preparazione Matrici

Gli embedding vengono convertiti in una matrice NumPy e normalizzati:

```
embeddings_matrix = np.array(embeddings, dtype=np.float32)
faiss.normalize_L2(embeddings_matrix)
```

La normalizzazione L2 è fondamentale per utilizzare il prodotto scalare come metrica di similarità coseno.

Step 4: Costruzione Indice

Viene utilizzato un indice `IndexFlatIP` (Inner Product) che garantisce ricerche esatte:

```
dimension = embeddings_matrix.shape[1]  # 768
index = faiss.IndexFlatIP(dimension)
index.add(embeddings_matrix)
```

Questa tipologia di indice è appropriata per dataset di dimensioni medio-piccole (< 1 milione di vettori) dove si richiede precisione massima.

Step 5: Salvataggio

L'indice e il mapping vengono salvati su disco:

```
faiss.write_index(index, INDEX_FILE)
# OUTPUT: /app/data/faiss_bge.index

with open(MAPPING_FILE, "wb") as f:
    pickle.dump(id_mapping, f)
# OUTPUT: /app/data/faiss_bge_mapping.pkl
```

Il file mapping contiene la corrispondenza tra indice posizionale e informazioni del film:

```
[
    {"imdb_id": "tt0111161", "title": "The Shawshank Redemption"},
    {"imdb_id": "tt0068646", "title": "The Godfather"},
    ...
]
```

5.2.3 Caratteristiche Tecniche

Parametro	Valore
Tipo indice	IndexFlatIP (exact search)
Metrica	Inner Product (cosine similarity)
Vettori indicizzati	>80,000
File output	/app/data/faiss_bge.index
Mapping	/app/data/faiss_bge_mapping.pkl

5.3 Sistema di Raccomandazione

5.3.1 Architettura

Il file [recommendation_service.py](file:///C:/Users/Utente/Desktop/Esame-Fenza/backend/recommendation_service.py) implementa il servizio di raccomandazione come classe singleton `RecommendationService`, garantendo efficienza e coerenza delle risorse.

5.3.2 Flusso di Raccomandazione

Il processo di generazione raccomandazioni si articola in diverse fasi:

Fase 1: Gestione Cache

Il sistema implementa un meccanismo di caching intelligente:

```
cached = user.get("recommendations")
if cached and cached.get("generated_at"):
    cache_time = cached.get("generated_at")
    data_updated = user.get("data_updated_at") or ""

    if cache_time >= data_updated:
        return cached  # Cache ancora valida
```

Le raccomandazioni vengono ricalcolate solo quando l'utente aggiunge o modifica rating, ottimizzando le performance.

Fase 2: Costruzione del Profilo Utente

Il sistema calcola il "gusto" dell'utente creando vettori medi pesati basati sui film valutati:

```
def _build_user_taste_vectors(self, user_movies: List[dict]):
    for m in user_movies:
        rating = m.get("rating", 3)
        weight = rating / 5.0  # Normalizzazione 0-1

        # Accumula vettori pesati
        desc_vectors.append(vec["embedding_description"])
        weights.append(weight)

    # Media pesata
    user_desc = np.average(desc_vectors, axis=0, weights=weights)
```

Questo approccio produce quattro vettori del profilo utente:

- `user_desc`: preferenze semantiche dalle descrizioni
- `user_genre`: preferenze di genere
- `user_actors`: attori preferiti
- `user_director`: registi preferiti

Fase 3: Ricerca di Similarità con FAISS

Il vettore descrizione dell'utente viene utilizzato per una ricerca FAISS:

```
user_desc_query = user_vectors["description"].reshape(1, -1)
faiss.normalize_L2(user_desc_query)

distances, indices = self.index.search(
    user_desc_query,
    TOP_K_CANDIDATES + len(seen_ids)
)
```

Vengono recuperati i top 500 candidati più simili.

Fase 4: Query Esplicite

Il sistema identifica registi e attori preferiti ed esegue query MongoDB dedicate:

```
def _extract_favorite_people(self, matched_films):
    # Calcola score per ogni regista/attore
    for film in matched_films:
        weight = rating / 5.0
        for d in directors:
            director_scores[d] += weight
```

```
    # Top 5 registi, top 10 attori
    return top_directors, top_actors
```

Questo garantisce che film di registi/attori amati dall'utente vengano considerati anche se semanticamente diversi.

Fase 5: Re-ranking Multi-dimensionale

I candidati vengono ri-classificati combinando quattro metriche di similarità:

```
similarity_score = (
    sim_desc * W_DESCRIPTION +      # 30%
    sim_genre * W_GENRE +           # 10%
    sim_actors * W_ACTORS +         # 50%
    sim_director * W_DIRECTOR       # 10%
)
```

Viene inoltre integrato un quality score basato su voto medio e numero di voti (Bayesian weighted rating):

```
def _calculate_quality_score(self, avg_vote: float, votes: int):
    MIN_VOTES = 1000
    GLOBAL_MEAN = 6.0

    weighted_rating = (
        (votes / (votes + MIN_VOTES)) * avg_vote +
        (MIN_VOTES / (votes + MIN_VOTES)) * GLOBAL_MEAN
    )
    return weighted_rating / 10.0
```

Lo score finale combina similarità (70%) e qualità (30%):

```
final_score = ALPHA_SIMILARITY * similarity_score + BETA_QUALITY * quality_score
```

Fase 6: Film Non Raccomandati

Il sistema genera anche una lista di film "anti-raccomandazioni":

```
not_rec_candidates = list(self.db[COLL_CATALOG].aggregate([
    {"$match": {
        "year": {"$gte": 2010},
        "votes": {"$gte": 10000},
        "imdb_id": {"$nin": list(seen_ids)}
    }},
```

```
    {"$sample": {"size": 500}}
])
```

Questi film hanno alta qualità ma bassa similarità con i gusti dell'utente, fornendo contrasto e serendipity.

5.3.3 Parametri di Configurazione

Parametro	Valore	Descrizione
TOP_K_CANDIDATES	500	Candidati iniziali da FAISS
N_RECOMMENDATIONS	66	Raccomandazioni totali (3 pagine × 22)
W_DESCRIPTION	0.30	Peso similarità descrizione
W_GENRE	0.10	Peso similarità genere
W_ACTORS	0.50	Peso similarità attori
W_DIRECTOR	0.10	Peso similarità regista
ALPHA_SIMILARITY	0.70	Peso componente similarità
BETA_QUALITY	0.30	Peso componente qualità

5.3.4 Output

Il servizio restituisce un oggetto JSON strutturato:

```
{
    "recommended": [
        {
            "imdb_id": "tt0111161",
            "title": "The Shawshank Redemption",
            "year": 1994,
            "poster": "https://...",
            "rating": 9.3,
            "genres": ["Drama"],
            "director": "Frank Darabont",
            "matchScore": 87
        },
        ...
    ],
    "not_recommended": [...],
    "matched_films": 45,
    "total_films": 52
}
```

5.4 Aggiornamento Incrementale

5.4.1 Motivazione

Il file [incremental_embedding_update.py](file:///C:/Users/Utente/Desktop/Esame-Fenza/backend/embedding_and_faiss/incremental_embedding_update.py) risolve il problema della sincronizzazione tra il catalogo film e gli embedding.

Quando nuovi film vengono aggiunti al database `movies_catalog`, è necessario:

1. Generare gli embedding per i nuovi film
2. Aggiornare l'indice FAISS con i nuovi vettori
3. Mantenere la coerenza del sistema

5.4.2 Processo di Aggiornamento

Step 1: Identificazione Film Mancanti

Viene utilizzata un'aggregation MongoDB efficiente:

```
def find_missing_films(client, db_name: str):
    pipeline = [
        {
            "$lookup": {
                "from": "bert_movies_vectors_bge",
                "localField": "imdb_id",
                "foreignField": "imdb_id",
                "as": "embedding"
            }
        },
        {
            "$match": {
                "embedding": {"$eq": []}  # Nessun embedding trovato
            }
        }
    ]
    return list(catalog.aggregate(pipeline))
```

Questo approccio è molto più efficiente rispetto al caricamento completo di entrambe le collezioni.

Step 2: Ricostruzione Vocabolari

Per garantire coerenza con gli encoding esistenti, i vocabolari vengono ricostruiti da tutti i film:

```
def rebuild_vocabularies(client, db_name: str):
    # Conta tutti gli attori e registi
    for film in all_films:
        actors = parse_comma_separated(film.get("actors", ""))
        directors = parse_comma_separated(film.get("director", ""))
        actor_counter.update(actors)
        director_counter.update(directors)

    # Mantieni stessi limiti dello script iniziale
```

```
    top_actors = [a for a, _ in actor_counter.most_common(MAX_ACTORS)]
    top_directors = [d for d, _ in director_counter.most_common(MAX_DIRECTORS)]
```

Step 3: Generazione Embedding

Gli embedding vengono generati solo per i film mancanti utilizzando la stessa logica del generatore principale:

```
def generate_incremental_embeddings(missing_films, all_genres,
                                    top_actors, top_directors):
    model = SentenceTransformer(BERT_MODEL)

    # Multi-hot encoding
    genre_enc = mlb_genre.transform([m["genres_list"] for m in movies_data])
    actors_enc = mlb_actors.transform(actors_filtered)
    director_enc = mlb_directors.transform(directors_filtered)

    # Embedding descrizioni
    desc_emb = model.encode(descriptions, batch_size=BATCH_SIZE, ...)

    return embeddings, document_ids
```

Step 4: Aggiornamento MongoDB

I nuovi embedding vengono inseriti nella collezione vettori:

```
for i, doc_data in enumerate(document_ids):
    coll.insert_one({
        "imdb_id": doc_data["imdb_id"],
        "title": doc_data["title"],
        "embedding_description": desc_emb[i].tolist(),
        "embedding_genre": genre_enc[i].tolist(),
        ...
    })
```

Step 5: Aggiornamento Indice FAISS

L'indice FAISS esistente viene caricato e aggiornato:

```
def update_faiss_index(new_embeddings: np.ndarray, new_ids: List[dict]):
    # Carica indice esistente
    index = faiss.read_index(INDEX_FILE)
    with open(MAPPING_FILE, "rb") as f:
        id_mapping = pickle.load(f)

    # Aggiungi nuovi vettori
```

```
    index.add(new_embeddings)

    # Aggiorna mapping
    id_mapping.extend(new_ids)

    # Salva
    faiss.write_index(index, INDEX_FILE)
    pickle.dump(id_mapping, MAPPING_FILE)
```

5.4.3 Scheduling

Lo script viene eseguito automaticamente ogni giorno alle 00:30 tramite cron job configurato in `main.py`:

```
scheduler.add_job(
    func=run_incremental_update,
    trigger='cron',
    hour=0,
    minute=30,
    id='incremental_embedding_update'
)
```

Questo orario è stato scelto per essere successivo all'aggiornamento del catalogo film (mezzanotte) ma con margine sufficiente per completare l'elaborazione.

5.4.4 Vantaggi

- **Efficienza**: processa solo film nuovi, non l'intero catalogo
- **Velocità**: aggiornamenti rapidi (secondi/minuti vs ore)
- **Coerenza**: mantiene integrità dei vocabolari e encoding
- **Automazione**: esecuzione pianificata senza intervento manuale

5.5 Sistema Mood-Based

5.5.1 Architettura

Il file [mood.py](file:///C:/Users/Utente/Desktop/Esame-Fenza/backend/mood.py) implementa un sistema di raccomandazione alternativo basato sullo stato d'animo dell'utente, completamente indipendente dal sistema di raccomandazione personalizzato.

5.5.2 Mapping Mood-Generi

Il cuore del sistema è una mappatura statica tra stati d'animo e generi cinematografici:

```
MOOD_GENRE_MAPPING = {
    "felice": ["Comedy"],
    "malinconico": ["Drama"],
    "eccitato": ["Action", "Adventure", "Fantasy"],
    "rilassato": ["Animation", "History"],
```

```
    "romantico": ["Romance"],
    "thriller": ["Thriller", "Horror"]
}
```

Questa mappatura è stata definita sulla base di associazioni psicologiche comuni tra emozioni e contenuti cinematografici.

5.5.3 Algoritmo di Selezione

Per ogni mood, il sistema esegue una pipeline MongoDB con selezione casuale:

```
def generate_mood_recommendations():
    for mood, genres in MOOD_GENRE_MAPPING.items():
        pipeline = [
            {
                "$match": {
                    "avg_vote": {"$gte": MIN_AVG_VOTE},      # >= 6.0
                    "votes": {"$gte": MIN_VOTES},            # >= 20,000
                    "genres": {"$in": genres}                 # Almeno uno dei
generi
                }
            },
            {
                "$sample": {"size": FILMS_PER_MOOD}          # 10 film casuali
            },
            {
                "$project": {
                    "_id": 0,
                    "imdb_id": 1,
                    "title": {"$ifNull": ["$original_title", "$title"]}
                }
            }
        ]

        results = list(catalog_collection.aggregate(pipeline))
```

La fase `$sample` garantisce varietà nelle raccomandazioni ad ogni generazione.

5.5.4 Criteri di Qualità

Il sistema filtra solo film di qualità garantita:

Parametro	Valore	Motivazione
MIN_AVG_VOTE	6.0	Filtra film mal recensiti
MIN_VOTES	20,000	Garantisce popolarità e affidabilità rating
FILMS_PER_MOOD	10	Numero ottimale per scelta utente

5.5.5 Persistenza

I risultati vengono salvati in un unico documento nella collezione `mood_movies`:

```
mood_document = {
    "generated_at": "2026-01-25T02:00:00+01:00",
    "felice": [
        {"imdb_id": "tt0088763", "title": "Back to the Future"},
        ...
    ],
    "malinconico": [...],
    "eccitato": [...],
    "rilassato": [...],
    "romantico": [...],
    "thriller": [...]
}

mood_collection.replace_one(
    {"_id": "daily_mood_recommendations"},
    mood_document,
    upsert=True
)
```

Utilizzando un `_id` fisso e `replace_one`, il sistema mantiene sempre una sola versione aggiornata.

5.5.6 Scheduling

Il sistema viene eseguito automaticamente ogni notte alle 2:00:

```
scheduler.add_job(
    func=generate_mood_recommendations,
    trigger='cron',
    hour=2,
    minute=0,
    id='mood_recommendations_update'
)
```

Questo garantisce che ogni giorno ci siano nuovi suggerimenti casuali per ogni mood.

5.5.7 Vantaggi

- **Semplicità**: nessun calcolo di embedding o similarità richiesto
- **Performance**: query MongoDB semplici e veloci
- **Varietà**: selezione casuale giornaliera
- **Complementarità**: offre un'alternativa alle raccomandazioni personalizzate
- **Accessibilità**: funziona anche per utenti senza rating

5.6 Workflow

5.6.1 Workflow Iniziale

1. **Setup iniziale** (eseguito una volta):
    - Esecuzione di `bge_embedding_generator.py` per processare tutto il catalogo
    - Esecuzione di `faiss_index_builder_bge.py` per creare l'indice
2. **Popolamento**: collezioni `bert_movies_vectors_bge` e indice FAISS pronti

5.6.2 Workflow Giornaliero

00:00 - Aggiornamento catalogo film (script esterno)

00:30 - `incremental_embedding_update.py`

- Trova film senza embedding
- Genera embedding per nuovi film
- Aggiorna indice FAISS

02:00 - `mood.py`

- Genera 10 film casuali per ogni mood
- Salva in `mood_movies`

5.6.3 Workflow Runtime

Richiesta raccomandazioni personalizzate:

1. Utente richiede raccomandazioni
2. `recommendation_service.py` verifica cache
3. Se cache valida → restituisce risultati cached
4. Altrimenti:
    - Costruisce profilo utente
    - Esegue ricerca FAISS
    - Esegue query esplicite
    - Re-rank e combina risultati
    - Salva cache
    - Restituisce raccomandazioni

Richiesta raccomandazioni mood:

1. Utente seleziona mood
2. Backend legge da `mood_movies`
3. Arricchisce con dati catalogo
4. Restituisce 10 film