



UNIVERSITÀ DEGLI STUDI DI FIRENZE
SCUOLA DI INGEGNERIA - DIPARTIMENTO DI INGEGNERIA
DELL'INFORMAZIONE

Tesi di Laurea Triennale in Ingegneria Informatica

**STUDIO DI SISTEMI DI RECOMMENDATION PER
SMART TOURISM**

Candidato
Matteo Tinacci

Relatore
Prof. Marco Bertini

Anno Accademico 2021/2022

Indice

Introduzione	i
1 Recommendation systems	1
1.1 Collaborative filtering	1
1.2 Hybrid recommendation	3
1.3 LightFM	4
1.3.1 Dataset Class	5
1.3.2 LightFM class	6
1.4 Doc2Vec	9
1.4.1 Esempi Doc2Vec	10
2 RecommendAPI	12
2.1 Flask	13
2.2 API	14
2.2.1 Database	14
2.2.2 Monument recommendation	16
2.2.3 Flask app	18
3 Conclusioni	23
Bibliografia	25

Introduzione

L' app creata da Lorenzo Massa chiamata Smart Tourism è un'applicazione Android di riconoscimento immagini finalizzato al far scoprire ai turisti interessati a un certo monumento alcune curiosità su di essi. L' utente che desidera avere informazioni relative ad un determinato monumento lo scaniona attraverso l'applicazione e, grazie ad una rete neurale e le immagini presenti nel database, questo viene riconosciuto fornendo quindi una guida testuale che il cliente può consultare e una guida audio da ascoltare.

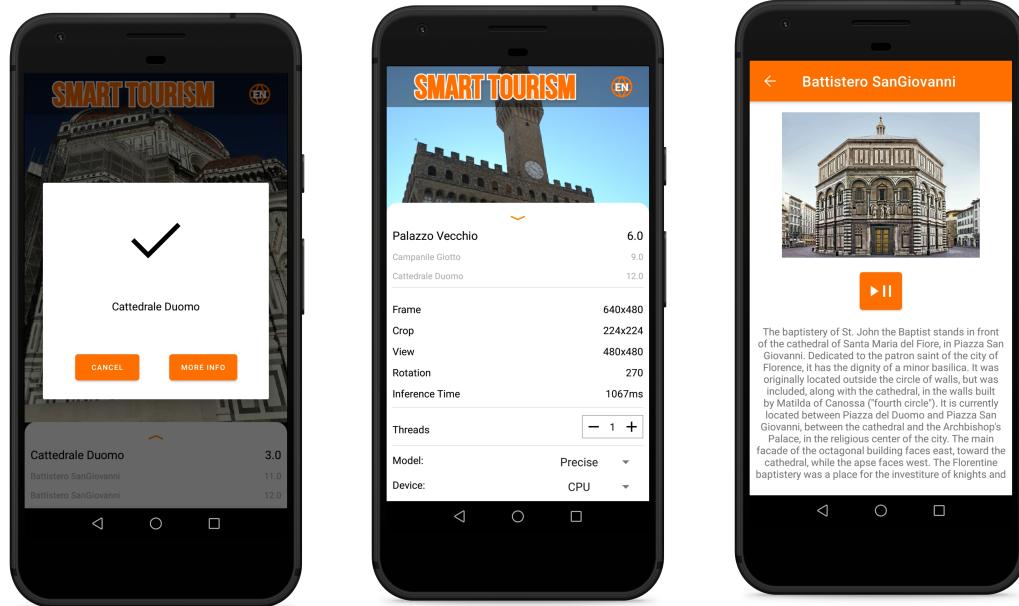


Figura 1: Screen dell' applicazione Smart Tourism.

Gli utenti dunque dovranno scansionare il monumento d'interesse una volta incontrato. L'idea alla base del presente lavoro di tesi è quella di inserire una nuova funzionalità, ovvero un sistema di raccomandazione che consiglia all'utente cosa visitare in modo tale che possa scoprire nuovi posti, magari a egli sconosciuti, che potrebbe apprezzare.

Capitolo 1

Recommendation systems

In generale un recommendation system è un tipo di algoritmo che utilizza tecniche di intelligenza artificiale e machine learning per suggerire prodotti, servizi o contenuti a utenti in base alle loro preferenze e comportamenti passati. Questi sistemi raccolgono quindi informazioni sulle attività e scelte dei clienti, come per esempio acquisti effettuati, recensioni, ricerche o interazioni con i vari contenuti e le analizzano per trovare modelli e correlazioni significative. In questo modo i recommendation system sono in grado di fornire raccomandazioni personali e pertinenti a ciascun utente, migliorando quindi l'esperienza di navigazione e aiutando a scoprire nuovi prodotto o servizi che potrebbero essere di interesse. Questi sono dunque ampiamente utilizzati in diversi settori come l'intrattenimento, la pubblicità, gli store online e molto altro.

1.1 Collaborative filtering

La tecnica del collaborative filtering è una delle tecniche di più diffuse e comuni utilizzate nei sistemi di raccomandazione. Questa si basa sull'idea

che se due utenti hanno valutato in modo simile alcuni prodotti è probabile che condividano gusti simili in altri prodotti.

Il collaborative filtering dunque utilizza i dati storici delle interazioni con i prodotti per calcolare una matrice di similarità tra gli utenti o tra i prodotti, la quale verrà poi usata per generare raccomandazioni personalizzate. Le interazioni possiamo raggrupparle in due gruppi:

1. Esplicite: caratterizzate da reazioni dalle quali possiamo quantificare il grado di interesse o di soddisfazione che un utente ha rivolto verso il prodotto in questione. Per esempio un like o dislike, una recensione o una valutazione.
2. Implicite: caratterizzate da interazioni con i prodotti per quali non possiamo riscontrare una vera e propria valutazione, ma grazie ai quali possiamo presupporre che ci possa essere un qualche tipo di interesse da parte dell'utente. Per esempio la ricerca di un determinato prodotto, il tempo speso nel leggere un articolo, la visualizzazione di un determinato contenuto e molto altro.

I dati raccolti quindi possiamo rappresentarli come una matrice sparsa data l'assenza di interazione tra alcuni utenti e alcuni prodotti.

Ci sono due approcci principali al collaborative filtering:

1. Collaborative filtering basato su utenti: analizza le interazioni degli utenti e cerca di trovare altri utenti con gusti simili. Le raccomandazioni vengono quindi generate in base alle valutazioni dei prodotti fatte da questi utenti.
2. Collaborative filtering basato su prodotti: analizza le caratteristiche dei prodotti e cerca di trovare altri prodotti simili in base alle valutazioni

degli utenti. Le raccomandazioni dunque vengono generate in base alle valutazioni degli utenti su questi prodotti.

Il limite di questa tecnica sta nel fatto che in situazioni di cold start, ovvero quando viene aggiunto un nuovo utente o un nuovo prodotto, non si hanno informazioni sufficienti per poter fare una raccomandazione. Per risolvere questo problema si può combinare più tecniche di raccomandazione, questa è chiamata raccomandazione ibrida.

1.2 Hybrid recommendation

La tecnica di raccomandazione ibrida combina più tecniche di raccomandazione per fornire raccomandazioni personalizzate a gli utenti. L'obiettivo di tale procedura è sfruttare i punti di forza di diverse tecniche di raccomandazione, superando le loro eventuali limitazioni e migliorando la precisione e la pertinenza delle raccomandazioni. Per esempio, un sistema di raccomandazione ibrido potrebbe usare il collaborative filtering per suggerire prodotti simili ad altri prodotti valutati positivamente dall'utente in combinazione con un content-based filtering per suggerire prodotti in base alle caratteristiche specifiche dell'utente, ad esempio se l'utente ha mostrato interesse in prodotti di una determinata categoria o marca.

La raccomandazione ibrida può essere implementata in vari modi, a seconda delle esigenze specifiche dell'applicazione. Ad esempio si può usare una combinazione di tecniche di raccomandazione in modo sequenziale, in cui la prima tecnica viene utilizzata per fornire una lista di candidati raccomandati, e la seconda per filtrare e classificare i candidati in modo più preciso. In generale, la tecnica di raccomandazione ibrida è la più utilizzata nei si-

smetti di raccomandazione moderni dato che offre una maggiore flessibilità e precisione rispetto alle altre tecniche di raccomandazione singola.

Quella appena descritta è la tecnica che ho scelto di utilizzare per realizzare la recommendation per l'applicazione Smart Tourism, in particolare ho usato quella offerta dalla libreria LightFM che spiegherò nella prossima sezione.

1.3 LightFM

LightFM [1] è una libreria di machine learning open source scritta in python che fornisce implementazioni efficienti di algoritmi di raccomandazione per la costruzione di sistemi di raccomandazione personalizzati. Questa è progettata per essere flessibile e scalabile e supporta una vasta gamma di dati in input, come matrici di interazione item-user e dati ausiliari come le caratteristiche del prodotto e informazioni dell'utente. Ciò significa che la libreria può essere utilizzata per sviluppare una vasta gamma di sistemi di raccomandazione presonalizzati. I suoi punti di forza sono la velocità di addestramento dei modelli anche in presenza di dataset molto grandi che può variare da pochi minuti o secondi a secoda della complessità del modello e delle specifiche dell'hardware.

Le classi principali presenti nella libreria, le quali sono state fondamentali per la realizzazione del sistema di raccomandazione sono la classe Dataset, utile per l'organizzazione dei dati, e la classe principale LightFM con la quale si addestra il modello e si estrae la recommendation.

1.3.1 Dataset Class

La classe Dataset di LightFM è una classe utile per caricare, pre-processare e suddividere i dati utilizzati per addestrare il sistema di raccomandazione. Questa classe facilita il processo di creazione di un dataset offrendo una serie di funzionalità per la gestione dei dati. La classe Dataset è progettata per gestire dati rappresentati come matrici di interazione user-item, in cui ogni riga rappresenta un utente e ogni colonna rappresenta un prodotto. Questi dati possono essere rappresentati come matrici sparse, questo perchè solo una frazione delle interazioni possibili risulterà non nulla. Inoltre fornisce funzioni per pre-processare i dati, come la normalizzazione delle valutazioni, la rimozione di utenti o prodotti rari, la creazione di matrici di confidenza e la generazione di suddivisioni di training e test. Per quanto riguarda quest'ultima funzionalità viene utilizzata la tecnica di campionamento stratificato per garantire che i dati di training e di test contengano una distribuzione equilibrata di utenti e prodotti. Ciò è particolarmente utile quando si lavora con dataset di grandi dimensioni in cui una suddivisione casuale dei dati potrebbe produrre dati di test che non riflettono accuratamente il comportamento dell'utente.

Le funzioni fornite dalla classe Dataset sono dunque le seguenti:

1. `fit(interactions, user_features=None, item_features=None, sample_weight=None, **kwargs)`: questa funzione viene utilizzata per creare una rappresentazione interna dei dati di input forniti come matrici di interazione utente-prodotto, e, se presenti, come matrici di features per utenti e prodotti. Inoltre calcola anche le statistiche necessarie per la suddivisione dei dati in training e test come il numero di utenti e prodotti totali.

2. `build_interactions(data, normalize = False)`: questa funzione converte una lista di tuple(`user_id`, `item_id`, `rating`) in una matrice di interazione user-item sparsa. La matrice risultante ha dimensioni (`num_users`, `num_features`), dove `num_users` è il numero di utenti unici e `num_features` il numero di features uniche.
3. `build_user_features(data, normalize=False)/build_item_features(data, normalize=False)`: queste funzioni convertono una lista di tuple (`user_id/item_id`, `feature_id`, `value`) in una matrice sparsa di features per gli utenti/item. La matrice risultante ha dimensioni (`num_users/num_items`, `num_features`) dove `num_items` è il numero di prodotti unici e `num_features` è il numero di features uniche.
4. `build_sample_weight(interactions)`: questa funzione restituisce una matrice di pesi per la matrice di interazione user-item fornita, dove i valori dei pesi sono proporzionali al numero di interazioni nella riga corrispondente della matrice di interazione. Questa è utile per garantire che gli utenti con un maggior numero di interazioni abbiano un peso maggiore durante l'addestramento del modello di raccomandazione.

Le matrici così ottenute sono utilizzabili per l'addestramento dei modelli di raccomandazione.

1.3.2 LightFM class

LightFM è la classe principale per la creazione di modelli di raccomandazione in LightFM. Si tratta di un modello di raccomandazione che utilizza rappresentazioni latenti ibride. Il modello apprende le embeddings ,ovvero rappresentazioni latenti in uno spazio ad alta dimensionalità, per gli utenti

e gli item in modo da codificare le preferenze dell’utente sugli item. Moltiplicando insieme queste rappresentazioni si ottengono punteggi per ogni elemento per un dato utente. Gli elementi con un punteggio elevato è più probabile che siano di interesse per l’utente.

Le rappresentazioni dell’utente e dell’item sono espresse in termini di rappresentazioni delle loro caratteristiche : viene stimato un embedding per ogni caratteristica, e queste vengono quindi sommate insieme per arrivare a rappresentazioni per utenti ed item. Le embeddings sono apprese attraverso metodi di discesa stocastica del gradiente.

Sono disponibili 4 funzioni di loss:

1. **Loss logistica** : è utile quando sono presenti interazioni positive (1) e negative (-1). La funzione di loss massimizza la log-verosimiglianza dei dati.
2. **Loss BPR**(Bayesian Personalised Ranking): è utile quando sono presenti solo iterazioni positive e si vuole ottimizzare l’area sotto la curva ROC AUC. Questa funzione massimizza la differenza di previsione tra un esempio positivo e un esempio negativo scelto casualmente.
3. **Loss WARP**(Weighted Approximate-Rank Pairwise): è utile quando sono presenti solo interazioni positive e si vuole ottimizzare la precisione dei primi elementi della lista di raccomandazione. Questa funzione massimizza il rango degli esempi positivi di violare il rango degli esempi negativi.
4. **Loss k-OS WARP**(k-th order statistic WARP): è una modifica di WARP che utilizza l’ennesimo esempio positivo per un dato utente come base per gli aggiornamenti a coppie. Questa funzione è utile

quando si vuole ottimizzare la precisione dei primi elementi della lista di raccomandazione ma con una preferenza per gli esempi più rari.

La classe lightFM fornisce anche due programmi di tasso di apprendimento:

1. **Adagrad**: algoritmo di ottimizzazione della discesa stocastica del gradiente (SGD) utilizzato per aggiornare i pesi di un modello durante l'addestramento. Utilizza un tasso di apprendimento adattivo per ogni parametro del modello. Questo approccio aiuta il modello ad adattarsi meglio alle varie scale dei dati in input.
2. **Adadelta**: variante di Adagrad che cerca di risolvere alcuni dei suoi problemi, come la diminuzione del tasso di apprendimento durante l'addestramento e l'accumulo di troppa storia dei gradienti passati. Utilizza una media mobile esponenziale dei gradienti passati per adattare il tasso di apprendimento in modo dinamico e quindi non richiede di specificare un tasso di apprendimento fisso come in Adagrad. In questo modo, Adadelta è in grado di fornire un aggiornamento dei pesi più stabile e migliore rispetto ad Adagrad.

Le principali funzioni che la classe LightFM offre sono le seguenti:

1. `fit(train_data, epochs=1, num_threads=1, verbose=False)`: questa funzione addestra il modello sui dati forniti come input. Il parametro `train_data` deve essere un oggetto `Dataset` contenente le matrici di interazione e features, il parametro `epochs` indica il numero di epoche di addestramento da eseguire. Il parametro `num_threads` indica il numero di thread da utilizzare per l'addestramento mentre `verbose` indica se stampare o meno le informazioni sull'addestramento.
2. `predict(user_ids, item_ids, userfeatures = None, item_features=None, num_threads=1)`: questa funzione restituisce le previsioni del modello

per gli utenti e i prodotti forniti come input. I parametri `user_ids` e `item_ids` devono essere array numpy di id di utenti e item, mentre `user_features` e `item_features` sono opzionali e possono essere usati per fornire le features degli utenti e degli item rispettivamente. Infine il parametro `num_threads` indica il numero di thread da utilizzare per la predizione.

3. `get_user_representations(user_ids)` e `get_item_representations(item_ids)`: queste funzioni restituiscono le rappresentazioni latenti degli utenti e degli item forniti come input rispettivamente. I parametri `user_ids` e `item_ids` devono essere array numpy di id di user e item rispettivamente.
4. `save(filepath, compress=True)`: questa funzione salva il modello su file. Il parametro `file_path` indica il percorso del file di output. Se il parametro `compress` viene posto come `True`, il file viene compresso usando gzip.

1.4 Doc2Vec

Osservando le necessità che avevo date funzioni presenti nelle classi `Dataset` e `LightFM` ho realizzato che avevo necessità di trovare un sistema che mi rappresentasse grandi testi in modo tale da poter utilizzare le descrizioni o le guide dei documenti come features per gli item da utilizzare come input nell'addestramento del modello.

Sono giunto alla conclusione che il metodo più efficace era l'utilizzo di Doc2Vec.

Doc2Vec [2], sviluppato come estensione dell'algoritmo Word2Vec e presente nella libreria gensim, è un modello di apprendimento di vettori di parole

basato su reti neurali che consente di rappresentare le parole in un vettore di numeri reali di dimensione ridotta, di generando quindi una rappresentazione vettoriale densa di un intero documento. Doc2vec funziona creando una rappresentazione vettoriale di ogni parola all'interno del documento e combinando queste rappresentazioni in un'unica rappresentazione vettoriale per l'intero documento. Ci sono due approcci principali per creare queste rappresentazioni:

1. **Modello DM**(Distributed Memory): il modello Doc2Vec aggiunge un vettore di contesto al modello Word2Vec. Questo è un vettore di numeri reali di dimensione fissa che rappresenta il contesto globale del documento e consente al modello di comprendere il testo globale del documento durante la creazione della rappresentazione vettoriale del documento.
2. **Modello DBOW**(Distributed Bag of Words): il modello Doc2Vec ignora il contesto delle parole all'interno del documento e si concentra solo sulla rappresentazione vettoriale dell'intero documento. Questo modello è simile al CBOW(Continous Bag of Words) in Word2Vec, nel quale il modello cerca di prevedere una parola basandosi solo sulle parole circostanti.

In generale Doc2vec è quindi una tecnica utile per la rappresentazione di documenti in un formato vettoriale compatto e potente, perfetto quindi per essere utilizzato per la costruzione delle features di mio interesse.

1.4.1 Esempi Doc2Vec

Per valutare se l'utilizzo di Doc2Vec per processare le descrizioni fosse effettivamente utile nel calcolo della somiglianza tra di esse da parte di Light-

FM ho utilizzato la funzione `most_similar()` offerta dal modello di Doc2Vec, in modo da valutare se monumenti della stessa categoria risultassero i più simili tra loro. Di seguito mostro alcuni esempi dei risultati ottenuti tramite questo test. Come si può vedere in figura quindi i risultati ottenuti sembrano

```
the most similar to Piazza della Signoria  
Piazza Santa Maria Novella 0.9222671985626221  
  
the most similar to Porta San Giorgio (Firenze)  
Porta San Gallo 0.9297890663146973  
  
the most similar to Sagrestia Vecchia  
Cappella Brancacci 0.8717494010925293
```

Figura 1.1: Esempi di descrizioni processate con Doc2Vec ritenute simili

essere soddisfacenti dato che accoppia monumenti di categoria simile. Questo mi ha portato quindi a decidere che Doc2Vec era un metodo efficace per processare le descrizioni da utilizzare come features durante l'addestramento del modello LightFM

Capitolo 2

RecommendAPI

In questo capitolo andrò ad analizzare da un punto di vista architettonico l'API che ho realizzato, spiegando i dettagli implementativi, i metodi utilizzati e il framework grazie al quale sono riuscito crearla.

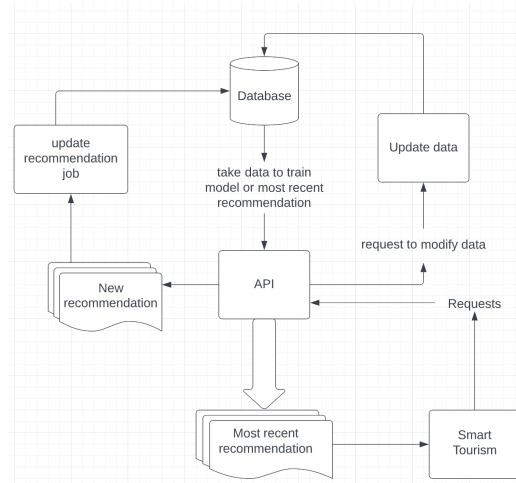


Figura 2.1: Schema dell'architettura

Le interazioni con l'applicazione principale avvengono grazie a chiamate anonime con l' API. Queste avverranno quando l'utente si registrerà in Smart Tourism inviando quindi le informazioni come nome e cognome all' api che

le salverà all'interno del database assegnandogli un identificativo numerico. Una seconda chiamata da parte dell'utente verrà effettuata ogni volta che questo farà una scansione di un monumento con successo, permettendo così di salvare sul database l'interazione che verrà poi utilizzata come materiale per la realizzazione della raccomandazione. Infine l'applicazione Smart Tourism farà richiesta all'API della raccomadazione personalizzata relativa all'utente che la sta utilizzando, quindi quella relativa all'id dell'utente in questione, la quale la preleverà dal database per inviarla così all'applicazione. Qui di seguito mostro il Sequence Diagram relativo al mio progetto.

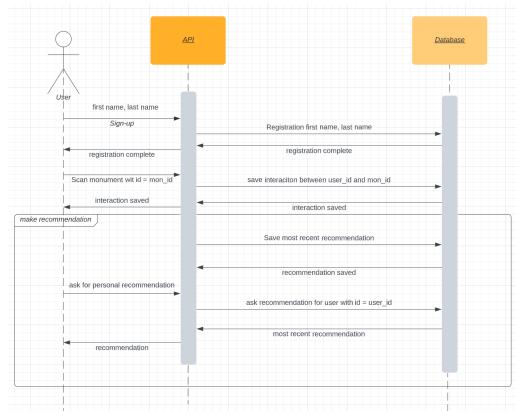


Figura 2.2: Sequence Diagram dell'interazione tra utente e API

2.1 Flask

Per realizzare l'api ho scelto di usare Flask. Flask è un framework web leggero e versatile scritto in Python che permette di creare applicazioni web di tipo REST in modo semplice ed efficiente. Fornisce una vasta gamma di funzionalità tra cui la gestione delle richieste http, la gestione delle sessioni, la gestione degli errori e la generazione di template HTML.

Uno dei vantaggi di Flask è la sua estendibilità: è possibile utilizzare una vasta gamma di librerie e pacchetti Python per aggiungere funzionalità all'applicazione. È inoltre altamente personalizzabile e consente a gli sviluppatori di creare applicazioni web adatte alle loro esigenze. Grazie alla sua leggerezza è quindi perfetto per essere usato per lo sviluppo di applicazioni web di piccole dimensioni. Questo grazie alla sua filosofia di "micro framework", ovvero una base essenziale e snella che consente agli sviluppatori di aggiungere solo le funzionalità di cui hanno bisogno, evitando di appesantire l'applicazione con funzionalità superflue. Inoltre non impone una specifica struttura del progetto o organizzazione di progetto o organizzazione del codice, lasciando libertà agli sviluppatori di organizzare il loro progetto come meglio credono.

2.2 API

L' API che ho realizzato consiste in un applicazione Flask affiancata da un database Sqlite nel quale si tiene traccia degli utenti registrati in Smart Tourism, i monumenti, le interazioni e la raccomandazione generata per ogni utente. La raccomandazione è realizzata grazie all'utilizzo della classe Recommendation, che spiegherò nel dettaglio successivamente, e il compito dell'app Flask è quello di ricevere le chiamate dall'applicazione principale e gestirle in modo corretto utilizzando il db per estrarre o salvare informazioni.

2.2.1 Database

Il database che ho creato è costituito da quattro tabelle:

1. **user**: questa tabella contiene le informazioni degli utenti registrati nell'applicazione. In particolare ha come chiave primaria un id numerico

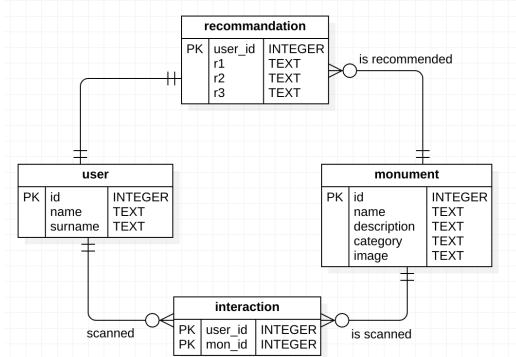


Figura 2.3: Schema ER del database dell' API.

unico per ogni utente e altri due campi che contengono il nome e il cognome.

2. **monument**: questa tabella contiene le informazioni relative ai monumenti che l'applicazione Smart Tourism può riconoscere e fornire le guide. Anche qui abbiamo come chiave primaria un id numerico unico per ogni monumento. Gli altri campi sono dedicati alle caratteristiche dei vari monumenti ovvero il nome, una descrizione testuale, una categoria in cui questo rientra, come per esempio 'Palazzo' oppure 'Chiesa' e infine un url relativo ad un immagine del monumento, utile nel caso si volesse mostrare una foto del monumento consigliato. La descrizione e la categoria sono le caratteristiche fondamentali che verranno utilizzate per il lato content-based della recommendation.
3. **interaction**: questa tabella è utile per salvare ciò con cui ogni utente ha interagito. E' composta da una chiave primaria formata da una coppia di id relativi rispettivamente a un user e un monumenti. Questo simboleggia la scansione e il riconoscimento da parte della applicazione Smart Tourism di un determinato monumento da parte di un utente.

Tale informazione è fondamentale per la parte di Collaborative filtering per fornire consigli in base alle preferenze degli utenti.

4. **recommendation:** questa tabella contiene i monumenti consigliati per ogni utente. La chiave primaria è l'id dell'utente e gli altri campi corrispondono ai primi tre monumenti consigliati restituiti dal sistema di raccomandazione. Questa è fondamentale per avere sempre a disposizione in modo rapido i primi tre elementi consigliati e non aver problemi durante l'aggiornamento del sistema di raccomandazione, il quale potrebbe durare diverso tempo a causa dell'addestramento del modello con i nuovi dati presi dal db.

2.2.2 Monument recommendation

Il sistema di raccomandazione è raccolto all' interno della classe Recommendation ed è composta da vari metodi per la costruzione degli elementi utili ad addestrare il modello e una funzione che restituisce la lista di raccomandazione contenente i 3 monumenti consigliati all'utente. I dati che vengono utilizzati per addestrare il modello vengono salvati negli attributi df_visit e df_item che contengono due dataframe pandas creati a partire da dataset di tipo csv che contengono rispettivamente i dati pesenti nella tabella interaction e monument del database. I metodi che ho utilizzato sono i seguenti:

1. `create_user_dict(self, name_col)` , `create_item_dict(self, id_col, name_col)` , `create_feature_dict(self, features)` : questi tre metodi servono a creare rispettivamente il dizionario degli utenti che hanno interagito almeno con un monumento, il dizionario dei monumenti contenente tutti i monumenti presenti all' interno del database e il di-

zionario delle features, che per ad ogni monumento associa la propria descrizione e categoria. I primi due vengono creati a partire dai dataframe pandas prima citati, mentre il terzo dizionario contiene le descrizioni vettorizzate grazie all'utilizzo dell' algoritmo Doc2Vec. Questi saranno fondamentali per creare le matrici di interazione e delle features che permetteranno al modello di essere addestrato.

2. `get_descriptions_vec(self)`: questo è il metodo che permette di calcolare le descrizioni vettorizzate da passare alla funzione `create_feature_dict`. Questa estrae le descrizioni dal dataframe dei monumenti, le processa in modo da separare le varie parole e poi attraverso un modello Doc2Vec estraggo una matrice contenente le descrizioni vettorizzate e correttamente processate.
3. `interaction_feature_matrix(self)`: questo è il metodo che crea le matrici sparse di interazione e delle features utili ad addestrare il modello. All'interno di questo metodo vengono richiamate le funzioni sopracitate per la costruzione della matrice delle descrizioni vettorizzate e il dizionario delle features, attraverso la funzione `fit()` della classe `Dataset` della libreria LightFM si inseriscono i dati in un oggetto di tale classe e se ne ricavano le matrici di interazione e delle feature attraverso i metodi `build_interactions()` e `build_item_features` descritte nel precedente capitolo.
4. `model_fit(self)`: questo metodo serve per addestrare il modello. Qui viene richiamata la funzione `interaction_feature_matrix` per creare le matrici sparse delle interazioni e delle features e viene addestrato il modello utilizzando una funzione di loss di tipo WARP, questo perchè come spiegato nel capitolo scorso questa fuzione è utile nel caso si

voglia ottimizzare i primi valori della raccomandazione, e dato che l’ api salverà nel database unicamente i primi 3 valori della lista mi sembrava la più azzeccata per svolgere il compito.

5. `recommendation(self, user_id)`: questa è la funzione che restituisce una raccomandazione personalizzata per un utente. La funzione per prima cosa controlla se l’utente ha mai scannerizzato qualche monumento e in caso negativo risolve la situazione di cold-start suggerendo i 3 monumenti più popolari e più visitati dagli altri utenti. In caso positivo invece si utilizza la funzione `predict()` del modello LightFM, spiegata nel capitolo precedente, per ottenere la matrice di raccomandazione e da quella estrarre i primi 3 elementi ancora non visitati dall’utente. La funzione restituisce quindi una lista di tre elementi che verrà poi inserita nella tabella recommendation del database. Questo metodo porta a suggerire all’utente anche cose non inerenti al suo interesse ma considerate dal modello come le più viste. Questo è ottimo per far sì che l’utente possa incuriosirsi di monumenti che magari inizialmente non pensava di voler visitare, scoprendo quindi nuovi interessi.

2.2.3 Flask app

Il cuore dell’ API risiede all’inerno dell’applicazione Flask, la quale permette di far comunicare il database grazie all’utilizzo della libreria python sqlite3, la classe Recommendation e l’ applicazione Smart Tourism attraverso chiamate GET, POST, PUT e DELETE. L’ applicazione quindi può usufruire dei servizi dell’ API usando una delle seguenti chiamate:

1. `@app.route('/user', methods=['GET', 'POST'])`: grazie a questo con una chiamata GET è possibile prendere dal database i dati di ogni

utente e restituirli in formato JSON, mentre attraverso una chiamata POST è possibile inserire un nuovo utente nel database.

2. `@app.route('/monument', methods=['GET', 'POST'])`: grazie a questo con una chiamata GET è possibile prendere dal database i dati di ogni monumento e restituirli in formato JSON, mentre attraverso una chiamata POST è possibile inserire un nuovo monumento nel database. Queste chiamate saranno effettuate dall'incaricato all' inserimento di monumenti e guide all'interno dell'applicazione Smart Tourism.
3. `@app.route('/insert_mon_csv', methods=['POST'])`: grazie a questo è possibile inserire una insieme di monumenti dato un file csv attraverso una chiamata POST. Questo servirà quindi a velocizzare l'operazione di inserimento dei monumenti, in quanto non ha bisogno di inserirli uno per uno a mano.
4. `@app.route('/interaction', methods=['GET', 'POST'])`: grazie a questo con una chiamata GET è possibile prendere dal database i dati di ogni interazione utente-monumento e restituirli in formato JSON, mentre attraverso una chiamata POST è possibile inserire una nuova interazione nel database. Queste chiamate saranno effettuate dall'applicazione Smart Tourism non appena un utente effettuerà una scansione con successo.
5. `@app.route("/user/<int:id>", methods=['GET', 'PUT', 'DELETE'])`: grazie a questo si possono effettuare operazioni sul database relative ad un utente con uno specifico id passato tramite url. In particolare con una chiamata GET è possibile estrarre le informazioni di uno specifico utente dal database e restituirle in formato JSON, attraverso una

richiesta PUT invece è possibile aggiornare il database con i nuovi dati relativi all'utente ed infine con una richiesta DELETE è possibile eliminare l'utente dal database.

6. `@app.route("/monument/<int:id>", methods=['GET', 'PUT', 'DELETE'])`: similmente a quanto visto nel punto precedente grazie a questo si possono effettuare operazioni sul database relative ad un monumento con uno specifico id passato tramite url. In particolare con una chiamata GET è possibile estrarre tutte le informazioni relative al monumento richiesto, attraverso una richiesta PUT è possibile aggiornare i dati di tale monumento e infine con la chiamata DELETE è possibile eliminare dal database il monumento richiesto.
7. `@app.route('/getRecommendation/<int:id>', methods=['GET'])`: grazie a questo si può ricevere la raccomandazione personalizzata per l'utente desiderato. Viene quindi estratta dal database la raccomandazione più recente e gli url delle immagini relative a tali monumenti e infine vengono restituiti in formato json sotto forma di dizionario.

Oltre a queste funzioni, avevo la necessità di crearne alcune utili a gestire l'andamento nel tempo dell'applicazione. In generale queste sono funzioni utili nella gestione del database.

1. `table_creation()`: questa funzione viene richiamata subito prima della chiamata della funzione `run()` dell'applicazione flask, ovvero quella che permette all'applicazione di comportarsi come un server, e serve per creare le tabelle all'interno del database in caso che questo risulti ancora vuoto.
2. `update_recommendation_db()`: questa funzione serve per aggiornare la tabella recommendation inserendovi la raccomandazione più recen-

Monumenti consigliati



Palazzo Vecchio



Basilica di Santa Croce



Palazzo Strozzi

Monumenti consigliati



Palazzo Vecchio



Basilica di Santa Maria Novella



Cattedrale di Santa Maria del Fiore

Figura 2.4: Raccomandazione più recente per due utenti differenti, il primo con interesse maggiore verso i palazzi, mentre il secondo verso le chiese.

te. La funzione quindi crea un’istanza della classe Recommendation addestrando quindi il modello e con l’uso di una transazione aggiorna la tabella recommendation richiamando il metodo recommendation descritto nella sezione precedente. Questo avviene grazie all’utilizzo dello scheduler BackgroundScheduler() fornito dalla libreria apscheduler che richiama la funzione come job ogni due minuti, permettendo quindi di tenere la raccomandazione nel database più aggiornata possibile. In particolare il BackgroundScheduler è ottimo in questa circostanza dato che aspetta in caso di mancata terminazione del job, permettendo quindi al modello di addestrarsi correttamente e

3. `create_csv(cursor)`: questa funzione è utilizzata per creare i file csv, a partire dalle tabelle monument e interaction del database, da fornire alla classe Recommendation, la quale a partire da questi crea i dizionari e le varie matrici di cui il modello ha bisogno per poter essere

addestrato, come descritto precedentemente.

Capitolo 3

Conclusioni

Per semplicità e a causa della mancanza di dataset molto grandi e completi durante lo sviluppo dell'applicazione mi sono limitato ad utilizzare un piccolo numero di documenti estratti da wikipedia attraverso un piccolo script di scraping realizzato con la libreria Wikimedia API. Per questi motivo la precisione del modello o in alcuni casi potrebbe essere dubbia data la differenza di lunghezza o di accuratezza che le descrizioni stesse di wikipedia hanno, In casi però di monumenti con descrizioni accurate e lunghezza simile la raccomandazione risulta coerente. Quindi è compito del gestore dell'applicazione fornire descrizioni o guide all'applicazione pertinenti e di lunghezza adeguata in modo da non avere i problemi sopra citati. L' API che ho realizzato è disponibile sul mio profilo github al seguente indirizzo <https://github.com/Itina99/RecommendApi.git>, inoltre è stata utilizzata per il progetto ReInHerit, ovvero un progetto Horizon2020 che aspira a interrompere l'attuale status quo di comunicazione, collaborazione e scambio di innovazione tra musei e siti del patrimonio culturale, nel senso che collegherà collezioni e siti del patrimonio culturale e presenterà il patrimonio tangibile e immateriale dell'Europa ai cittadini e turisti nei loro più ampi contesti storici

e geografici.



Figura 3.1: Logo del progetto ReInHerit

Bibliografia

- [1] Maciej Kula. Metadata embeddings for user and item cold-start recommendations. In Toine Bogers and Marijn Koolen, editors, *Proceedings of the 2nd Workshop on New Trends on Content-Based Recommender Systems co-located with 9th ACM Conference on Recommender Systems (RecSys 2015), Vienna, Austria, September 16-20, 2015.*, volume 1448 of *CEUR Workshop Proceedings*, pages 14–21. CEUR-WS.org, 2015.
- [2] Quoc Le and Tomas Mikolov. Distributed representations of sentences and documents. In *International conference on machine learning*, pages 1188–1196. PMLR, 2014.