Recommender System Documentation

Release 1.0.0

Eric Daoud

May 16, 2019

Contents:

CHAPTER 1

About

This is Notflix, a free movie database and recommendation website.

This website is simply a side project, that aims at displaying a fixed dataset of movies and provide recommendations about other movies to watch. I am building it mostly for fun and also to have a nice playground to implement various recommendation algorithms using Machine Learning.

NotFlix is based on data from the following sources:

- OMDB: The Open Movie DataBase
- **Grouplens' MovieLens:** Datasets behind MovieLens project.

CHAPTER 2

Installation

Pre-requisite:

1. Install the following software:
   - Docker
   - docker-compose
   - Python 3
   - virtualenv
2. Download the movielens data:
   - Download the `ml-1m dataset` by clicking here
   - Unzip it and place it under `datasets/movielens/ml-1m`
3. **To add the movie metadata to the downloaded dataset** (such as the actors, ...) I chose to use the OMDb API (the Open Movie Database). Support him on Patreon and get a key that you will place in a text file called `omdb.key` at the root of this repository

Once you have all the pre-requisite set up, follow these steps:

1. Copy the db-credentials.env template and add the credentials you want:

```
cp -n db-credentials.env.dist db-credentials.env;
```

2. Create a virtual environment and install the required packages:

```
virtualenv venv;
source venv/bin/activate;
pip install -r requirements.txt;
```

3. Build the Docker images:

```
docker-compose build;
```

4. Launch the PostgreSQL database:

```
docker-compose up -d postgres
```

5. Use the following flask-cli commands to insert the data into the DB:

```
export FLASK_APP="src/web";
export POSTGRES_HOST="127.0.0.1";

flask init-db;
flask insert-engines;
flask insert-pages;
flask download-movies;
flask insert-movies;
flask train-engines;
flask upload-engines;
```

6. **Launch the application with `make start` and then visit** `localhost:5000`.

CHAPTER 3

Usage

So far, Notflix exposes the following pages:

- **A home page, displaying the popular movies, the user browsing history** and some personalized recommendations.
- **A movie page, displaying basic informations about the selected movie** and recommendations on similar movies to watch.
- A **genres** page, that lets you browse movies by genres.
- A **search** page, that lets you search the movies.

The configuration for engines and pages is handled with the display.json file. You can use it to change the engines displayed, their names and order on the page.

CHAPTER 4

Repository organization

The repository is organized the following way:

- .circleci: Configuration file for CircleCI
- **datasets** [Folder containing the datasets] (so far only movielens)
- **docs** [Folder containing the documentation, auto-generated] by Sphinx.
- logs: Logs file are saved here
- models: Machine Learning models are saved here.
  - Under bin we save the models weights.
  - Under csv we save CSV files containing the predictions made by a given engine.
- notebooks: The exploratory Jupyter notebooks
- src: Source code
  - api: Flask API, responsible of computing the recommendations displayed on the web app.
  - data_interface: Code for interacting with the cache or the database.
  - recommender: Everything related to computing recommendations.
  - tracker: Code for tracking the user events.
  - utils: Various utility functions
  - web: Code for the Flask web application
- tests: Unit test code

CHAPTER 5

Notes

- **I am deliberately showing multiple engines on a web page to outline** the different recommendations results from an algorithm to another.
- **I am not removing the movie duplicates from an engine to another** for the same reason than above.
- **The Machine Learning algorithms are not very well trained yet, I spent** some time working on the application to make it easy to add new engines later.

So far, the movie page looks like this: 

5.1 notflix

5.1.1 config module

5.1.2 src package

Subpackages

src.api package

src.api.**create_app**(*test_config=None*)

Submodules

src.api.errors module

src.api.errors.**page_not_found**(*e*)

src.api.recommend module

src.api.recommend.**item**(*item_id*)

src.api.recommend.**session_recommendations**(*session_id*)

src.api.recommend.**user**(*user_id*)

src.api.wsgi module

src.data_interface package

Submodules

src.data_interface.cache module

class src.data_interface.cache.**Cache**

Bases: object

append(*key*, *value*)

Append to a redis list

Parameters

- **key** (*str*) – cache key
- **value** (*str*) – object to store in cache

get(*key*, *start=None*, *end=None*)

Get an object from cache by its key

Parameters

- **key** (*str*) – cache key
- **start** (*int*) – when querying a redis list, starting range of the list.
- **end** (*int*) – when querying a redis list, ending range of the list.

Returns cached object

Return type str

set(*key*, *value*)

Set an object in cache by its key

Parameters

- **key** (*str*) – cache key
- **value** (*str*) – object to store in cache

src.data_interface.downloader module

This module contains wrappers to download various movies datasets. So far we are only using Movielens but we can add more if we want.

Every dataset should have its wrapper class that inherits from `Downloader`.

class src.data_interface.downloader.**Downloader**
Bases: abc.ABC

download_to_file()

insert_in_db()

item_from_api(*id*)

read_api_key(*key_filepath*)

class src.data_interface.downloader.**MovielensDownloader**
Bases: *src.data_interface.downloader.Downloader*

download_to_file()

insert_in_db()

src.data_interface.model module

class src.data_interface.model.**BaseTable**(***kwargs*)
Bases: sqlalchemy.ext.declarative.api.Model

created_at = Column(None, DateTime(), table=None, nullable=False, default=ColumnDefaul

id = Column(None, Integer(), table=None, primary_key=True, nullable=False)

updated_at = Column(None, DateTime(), table=None, onupdate=ColumnDefault(<function dat

class src.data_interface.model.**Engine**(***kwargs*)
Bases: *src.data_interface.model.BaseTable*

created_at

display_name

id

priority

type

updated_at

class src.data_interface.model.**Genre**(***kwargs*)
Bases: *src.data_interface.model.BaseTable*

created_at

id

name

updated_at

class src.data_interface.model.**Movie**(***kwargs*)
Bases: *src.data_interface.model.BaseTable*

actors

as_dict()

awards

country

created_at

description

director

duration

genres

id

image

language

name

rating

updated_at

year

class src.data_interface.model.**Page**(***kwargs*)
Bases: *src.data_interface.model.BaseTable*

created_at

engines

id

name

updated_at

class src.data_interface.model.**Recommendation**(***kwargs*)
Bases: *src.data_interface.model.BaseTable*

created_at

engine_name

id

recommended_item_id

score

source_item_id

source_item_id_kind

updated_at

class src.data_interface.model.**User**(***kwargs*)
Bases: *src.data_interface.model.BaseTable*

created_at

email

favorite_genres

id

password

updated_at

username

src.data_interface.model.**init**()

src.data_interface.model.**insert**(*to_insert*)

src.data_interface.model.**truncate_table**(*table*)

src.recommender package

Subpackages

src.recommender.engines package

This is the `engines` package, where we define the recommendation engines.

This package contains the following modules:

- **`engine`: Module where the base classes are defined. All the created** engines should inherit from one of these base classes. They define the skeleton of the engine, like which methods they must overwrite.
- `collaborative_filtering`: All collaborative filtering engines
- `content_based`: All content based filtering engines
- **`generic`: All the generic engines, that are not really collaborative** or content based (e.g display the most popular items, or the items in the user browsing history, etc . . .)

Submodules

src.recommender.engines.collaborative_filtering module

class src.recommender.engines.collaborative_filtering.**Item2Vec**

Bases: *src.recommender.engines.engine.OfflineEngine*

train()

Method for training the engine. This method should load the dataset, compute the recommendations and then persist them to disk using `save_recommendations_to_csv`.

class src.recommender.engines.collaborative_filtering.**Item2VecOnline**

Bases: *src.recommender.engines.engine.OnlineEngine*

load_model()

Load the ML model from disk and return it

Returns The ML model to be saved as `self.model`

predict(*context*)

Predict using the loaded model and the context.

Parameters **context** (`src.recommender.wrappers.Context`) – Context wrapper

Returns list of recommended ids sorted by descending score scores (list(float)): list of scores for each recommended item

Return type ids (list(int))

train()
Train a ML model and save it to disk

class src.recommender.engines.collaborative_filtering.**ItemBasedCF**
Bases: *src.recommender.engines.engine.OfflineEngine*

train()
Method for training the engine. This method should load the dataset, compute the recommendations and then persist them to disk using `save_recommendations_to_csv`.

src.recommender.engines.content_based module

class src.recommender.engines.content_based.**OneHotMultiInput**
Bases: *src.recommender.engines.engine.OfflineEngine*

train()
Method for training the engine. This method should load the dataset, compute the recommendations and then persist them to disk using `save_recommendations_to_csv`.

class src.recommender.engines.content_based.**SameGenres**
Bases: *src.recommender.engines.engine.QueryBasedEngine*

compute_query(*context*)
Abstract method that computes the SQL query using SQLAlchemy

Parameters **context** (`recommender.wrappers.Context`) – context wrapper

Returns query result

class src.recommender.engines.content_based.**TfidfGenres**
Bases: *src.recommender.engines.engine.OfflineEngine*

train()
Method for training the engine. This method should load the dataset, compute the recommendations and then persist them to disk using `save_recommendations_to_csv`.

src.recommender.engines.engine module

class src.recommender.engines.engine.**Engine**
Bases: `abc.ABC`

Abstract class for all engines. You should not directly use this class, instead use the classes that inherit from this class.

init_recommendations(*context*)
Create an empty `src.recommender.wrappers.Recommendations`` object and fill in the engine type, display name and priority based on the informations stored in DB.

Parameters **context** (`src.recommender.wrappers.Context`) – Context wrapper, containing useful informations for the engine.

Returns Recommendations object filled with engine type, display name and priority

Return type (*src.recommender.wrappers.Recommendations*)

recommend(*context*)
Abstract method for all engines for recommending items.

The context wrapper stores all the informations the engine might need to compute the recommendations, like the current item_id, the current user_id, the user browsing history, etc ...

Every engine must override this method. They have to call `self.init_recommendations` first to create an empty `src.recommender.wrappers.Recommendations` object and then enrich it with the recommended items.

Parameters **context** (`src.recommender.wrappers.Context`) – the context

Returns the recommendation object

Return type *src.recommender.wrappers.Recommendations*

class src.recommender.engines.engine.**OfflineEngine**

Bases: *src.recommender.engines.engine.QueryBasedEngine*

These engines are a special kind of QueryBasedEngine because they require a training.

Most of the offline Machine Learning algorithms will inherit from this class.

The recommendations are computed offline with the `train` method, then saved on disk with `save_recommendations_to_csv` and finally uploaded to the DB using `upload`.

compute_query(*context*)

Get the recommended items from the DB.

Parameters **context** (`src.recommender.wrappers.Context`) – Context wrapper

Returns list of Recommendation

Return type list

save_recommendations_to_csv(*recommendations*)

Save recommendations to a CSV file.

Parameters **recommendations** (*list(tuple)*) – List of recommendation tuple corresponding to: (movie_id, recommended_movie_id, input_kind, score)

train()

Method for training the engine. This method should load the dataset, compute the recommendations and then persist them to disk using `save_recommendations_to_csv`.

upload()

Upload the recommendations from a CSV file to the DB.

class src.recommender.engines.engine.**OnlineEngine**

Bases: *src.recommender.engines.engine.Engine*

Online Machine Learning Engines that do not get their recommendations from a SQL query but from a loaded model.

The model is trained with the `train` method, and loaded at runtime with the `load_model` method.

load_model()

Load the ML model from disk and return it

Returns The ML model to be saved as `self.model`

predict(*context*)

Predict using the loaded model and the context.

Parameters **context** (`src.recommender.wrappers.Context`) – Context wrapper

Returns list of recommended ids sorted by descending score scores (list(float)): list of scores for each recommended item

Return type ids (list(int))

recommend(*context*)
Recommend movies based on context

Parameters **context** (src.recommender.wrappers.Context) – Context wrapper

Returns src.recommender.wrappers.Recommendations as dict

Return type recommendations (dict)

train()
Train a ML model and save it to disk

class src.recommender.engines.engine.**QueryBasedEngine**
Bases: *src.recommender.engines.engine.Engine*

Abstract class for an engine based on a SQL query performed at every call. These are engines require no training, for instance an engine that will recommend random items for DB.

compute_query(*context*)
Abstract method that computes the SQL query using SQLAlchemy

Parameters **context** (recommender.wrappers.Context) – context wrapper

Returns query result

recommend(*context*)
Method for recommending items, by calling *self.compute_query*.

Parameters **context** (recommender.wrappers.Context) – context wrapper

Returns recommendations as list of dict

Return type list(dict)

src.recommender.engines.generic module

class src.recommender.engines.generic.**MostRecent**
Bases: *src.recommender.engines.engine.QueryBasedEngine*

compute_query(*context*)
Abstract method that computes the SQL query using SQLAlchemy

Parameters **context** (recommender.wrappers.Context) – context wrapper

Returns query result

class src.recommender.engines.generic.**Random**
Bases: *src.recommender.engines.engine.QueryBasedEngine*

compute_query(*context*)
Abstract method that computes the SQL query using SQLAlchemy

Parameters **context** (recommender.wrappers.Context) – context wrapper

Returns query result

class src.recommender.engines.generic.**TopRated**
Bases: *src.recommender.engines.engine.QueryBasedEngine*

compute_query(*context*)
Abstract method that computes the SQL query using SQLAlchemy

Parameters **context** (recommender.wrappers.Context) – context wrapper

Returns query result

class src.recommender.engines.generic.**UserHistory**
Bases: *src.recommender.engines.engine.QueryBasedEngine*

compute_query(*context*)
Abstract method that computes the SQL query using SQLAlchemy

Parameters **context** (recommender.wrappers.Context) – context wrapper

Returns query result

Submodules

src.recommender.metrics module

The metrics functions are copied from this repository: https://gist.github.com/bwhite/3726239

src.recommender.metrics.**dcg_at_k**(*r*, *k*, *method=0*)
Score is discounted cumulative gain (dcg) Relevance is positive real values. Can use binary as the previous methods. Example from http://www.stanford.edu/class/cs276/handouts/EvaluationNew-handout-6-per.pdf >>> r = [3, 2, 3, 0, 0, 1, 2, 2, 3, 0] >>> dcg_at_k(r, 1) 3.0 >>> dcg_at_k(r, 1, method=1) 3.0 >>> dcg_at_k(r, 2) 5.0 >>> dcg_at_k(r, 2, method=1) 4.2618595071429155 >>> dcg_at_k(r, 10) 9.6051177391888114 >>> dcg_at_k(r, 11) 9.6051177391888114 :param r: Relevance scores (list or numpy) in rank order

(first element is the first item)

Parameters

- **k** – Number of results to consider
- **method** – If 0 then weights are [1.0, 1.0, 0.6309, 0.5, 0.4307, ...] If 1 then weights are [1.0, 0.6309, 0.5, 0.4307, ...]

Returns Discounted cumulative gain

src.recommender.metrics.**evaluate_recommendations**(*predictions*, *target*, *k*)
Evaluate the quality of recommendations with NDCG. We compare the predictions set with the target set that should reflect what items are relevant.

Parameters

- **predictions** (*list*) – List of recommended items. Ordered by descending score.
- **target** (*list*) – List of relevant items.
- **k** (*int*) – Only consider the k first items in the set

Returns NDCG at k score

Return type float

src.recommender.metrics.**ndcg_at_k**(*r*, *k*, *method=0*)
Score is normalized discounted cumulative gain (ndcg) Relevance is positive real values. Can use binary as the previous methods. Example from http://www.stanford.edu/class/cs276/handouts/EvaluationNew-handout-6-per.pdf >>> r = [3, 2, 3, 0, 0, 1, 2, 2, 3, 0] >>> ndcg_at_k(r, 1) 1.0 >>> r = [2, 1, 2, 0] >>> ndcg_at_k(r, 4) 0.9203032077642922 >>> ndcg_at_k(r, 4, method=1) 0.96519546960144276 >>> ndcg_at_k([0], 1) 0.0 >>> ndcg_at_k([1], 2) 1.0 :param r: Relevance scores (list or numpy) in rank order

(first element is the first item)

Parameters

- **k** – Number of results to consider
- **method** – If 0 then weights are [1.0, 1.0, 0.6309, 0.5, 0.4307, ...] If 1 then weights are [1.0, 0.6309, 0.5, 0.4307, ...]

Returns Normalized discounted cumulative gain

src.recommender.recommender module

class src.recommender.recommender.**Recommender**
Bases: object

Recommender System base class.

recommend(*context*, *restrict_to_engines=[]*)
Call all the active engines based on a context and return their recommendations.

It is possible to restrict to a list of engines by using the restrict_to_engines parameter.

Parameters **context** (recommender.wrappers.Context) – Context wrapper, providing informations about the current item or user or session.

Returns List of recommendations as dictionaries

Return type list(dict)

src.recommender.wrappers module

class src.recommender.wrappers.**Context**(***kwargs*)
Bases: object

A wrapper for context that will help engines make recommendations.

class src.recommender.wrappers.**Recommendations**
Bases: object

A recommendation object that is returned by the engines

to_dict()
Convert recommendation object to dict

Returns the recommendations as a dictionary

Return type dict

to_string()
Convert recommendation object to string for debug purpose

Returns the recommendations as string, stating the type and number of items recommended.

Return type str

src.tracker package

Submodules

src.tracker.tracker module

class src.tracker.tracker.**Tracker**
Bases: object

get_views_history(*key*, *n=-1*)

store_item_viewed(*key*, *item*)
Store item viewed in cache

Parameters

- **key** (*str*) – cache key
- **item** (*str*) – value

Returns:

src.utils package

Submodules

src.utils.data module

src.utils.data.**matrix_from_df_with_vect**(*df*, *groupby_column*, *data_column*, *vectorizer*)

src.utils.data.**recommendations_from_similarity_matrix**(*movie_ids*, *sim_matrix*, *n_recommendations*, *input_kind*)

src.utils.data.**sparse_matrix_from_df**(*df*, *groupby*, *indicator*)
Make a scipy sparse matrix from a pandas Dataframe

Parameters

- **df** (*pd.DataFrame*) – Dataframe with the matrix desired rows as index
- **groupby** (*str*) – Name of the column to set as matrix column
- **indicator** (*str*) – Name of the column that will serve as data

Returns sparse matrix (scipy.sparse.csr_matrix) row values (list) column values (list)

src.utils.logging module

src.utils.logging.**setup_logging**(*config_path*)
Setup logging configuration

src.web package

src.web.**create_app**()

Submodules

src.web.errors module

src.web.errors.**page_not_found**(*e*)

src.web.genres module

src.web.genres.**genre**(*genre*)

src.web.genres.**index**()

src.web.home module

src.web.home.**index**()

src.web.item module

src.web.item.**index**(*item_id*)

src.web.login module

src.web.login.**signin**()

src.web.login.**signout**()

src.web.login.**signup**()

src.web.login.**valid_signin**(*username*, *password*)

src.web.login.**valid_signup**(*username*, *password*)

src.web.search module

src.web.search.**search**()

src.web.wsgi module

src.web.you module

src.web.you.**index**()

src.web.you.**taste**()

CHAPTER 6

Indices and tables

- genindex
- modindex
- search

Python Module Index

C

S

Index

L

M

N

O

P

Q

R

S

T

U

V

Y