Build a Personal Movie Recommender with Django

Estimated time needed: 120 minutes

Choosing an interesting movie to watch on a weekend evening is always hard. It would be great if there is a handy movie recommender to help you make a good decision.

In this guided project, you will be building such a personal movie recommender using the Django web framework. The movie recommendations will be generated based on your or your family's previously watched movies. The recommendation algorithm included in this project is extremely simple but effective so you do not need any hardcore Machine Learning knowledge to understand and complete this project.

Prerequisite

- Python
- Basic HTML/CSS
- Basic knowledge about the `Set` data structure

Objectives

After completing this Guide Project, you will be able to:

1. Describe the key components of Django web framework
2. Describe the basic principles of a recommendation system
3. Build a simple but effective personal movie recommender with Django

Django Crash

Django is a very popular Python web framework designed for rapid and full-stack web app development.
Many popular apps are built on Django such as Youtube, Spotify, Dropbox, edX, and more.

Django Model-View-Template

Like other web frameworks, Django splits its application logic into the following three Model-View-Controller like components:

- `Django Model` manages data modelling and database mapping as well as business logic to process data
- `Django View` describes which data is presented, but not how it is presented. Typically,
  Django View delegates and renders an HTML page, which describes how the data is presented
- `Django Template` generates dynamic HTML pages to present data

When a client sends a request, the Django server routes the request to the appropriate view based on the Django URL configuration and acts as a traditional `Controller`

Django Models

Django uses `Django Models` to represent database tables and map them to objects, such as process is called ORM.
Django Models tries to make the developer's life easier by abstracting databases and mapping objects and methods into tables and SQL queries automatically.

You just need to define classes as Django Models, and will be later mapped to database tables accordingly.
Then you can simply use Django Models API to perform CRUD on the database tables without writing a single line of SQL

Django Views

In Django, a View is essentially a Python function. Such a function takes a Web request and applies the necessary logic to generate a Web response, such as the HTML contents of a Web page, a redirect, a 404 error, an XML document, an image, or any other Web response.
Often, View interacts with Django Models to get required data in the form of QuerySet or objects to generate a Web response.

Django Template

Django uses a template to generate dynamic Web pages which are sent back and rendered in a user's browser.
The Django template contains static HTML elements, as well as special Python code describing how the dynamic content of HTML pages will be generated.

Django App Development Process

Let's look at a typical Django development process. The order of these steps may vary.

First, we create a Django project which is a container for Django apps and settings. Then we create and add one or more Django apps to the project.

In Core Development, we create Django models to model the data and create views to determine which data need to be presented to the UI. We also map the request URLs to our views so Django can forward requests to corresponding views via URLs. Then we can start designing and building the UI.

OK, by now you should have enough knowledge about Django to get you started on this Guided Project.
If you have more time, you could also learn more about Django framework from many other tutorials and courses online.

Download and Run a Django Template App

Before starting the lab, make sure your current directory is `/home/project`.

To simplify your development process, we have provided a skeleton Django project, which you can just download using the following `wget` command:

```
wget "https://cf-courses-data.s3.us.cloud-object-storage.appdomain.cloud/build-a-personal-movie-recommender-with-django/app_template/recommender.zip"
unzip recommender.zip
rm recommender.zip
```

Then, we need to install required Python packages

- CD to `/recommender` folder then
- Run `pip install` from a requirements file:

  ```
  python3 -m pip install -r requirements.txt
  ```

- If face any pillow installation error, fix the pillow installation error

Run the following commands:

```
python3 -m pip install --upgrade pip
python3 -m pip install --upgrade Pillow
```

To save your works in progress, please constantly save the project files in your local computer, cloud storage, or in a `GitHub`/`GitLab` repository.

Review Django Project Structure

Next, let's take a look at the created Django project.

Key Project Files

A Django project is essentially a Python package that contains all the settings for instances of Django apps.

You need to know the usage for some core files that Django creates.

- `manage.py` is a command-line interface used to interact with the Django project to start the server, migrate models, and so on.
- `settings.py` contains the settings and configurations for your Django project.
- `urls.py` contains the URL and routing definitions of your Django app, such as URLs for a `movierecommender` app

Key Application Files

In addition to the Django project container `/recommender` folder, we also created a Django app in `movierecommender` folder, which contains the following important app files.

- `admin.py` includes everything you need to create and customize an admin site to manage users and content.
- `models.py` contains the data models and ORM.
- `views.py` contains view functions
- `urls.py` contains the URL declarations and routing for the app.
- `app.py` contains the application configuration class.
- `/migrations` folder contains scripts for model migration.
- `/management/commands` contains your customized Django commands which can be evoked by `manage.py` file. Some useful commands may include load data into database or run some batch jobs

Start Django Server

Make sure you are in `/home/project/recommender/` folder.

- Before starting the app, you will need to perform migrations to create necessary database tables:

```
python3 manage.py makemigrations
```

- and run migration

```
python3 manage.py migrate
```

and you should see many tables are created in the terminal console.

Meanwhile, a new SQLite database `recommender/db.sqlite3` has been created by Django for you.

It is the default SQLite database configured in `recommender/settings.py`.
In case you are not familiar with SQLite, it is an embedded and file-based database. As its name suggests, it is small and fast, which makes it ideal for our guide project.

Review a Predefined Movie Model

In this template app, we have created a `Movie` model for you. To review this model, simply go to `recommender/movierecommender/models.py` and find the `Movie` class.

This `Movie` class extends from Django `models.Model` super class.
The Django `models` provides a set of Object-Relationship Mapping APIs such as mapping your sub-class into a database table, insert/update/delete database records, mapping database records into objects, etc.

The defined `Movie` class contains the following fields:

- `imdb_id`: IMDB id
- `genres`: Movie genres like animation, family, action science fiction, etc.
- `original_language`: Movie original language
- `original_title`: Movie title
- `release_date`: Movie release date, year only
- `overview`: The overview or short description of the movie
- `vote_average`: Average voting for the movie
- `vote_count`: Total vote counts for the movie, may reflect the popularity
- `poster_path`: The relative URL of movie poster image
- `watched`: If you have watched the movie
- `recommended`: If the movie has been recommended to you

After the Django migration you executed earlier, Django Model will also create a `movie` table automatically in our SQLite database with corresponding columns.

Load Movie Data into an SQLite Database

Since the generated `movie` database table is empty, we can use a prepared load command to load movie data from `/recommender/movie.csv` to the database table.

The prepare load command can be found in `recommender/movierecommender/management/commands/load_movies.py`

If you open this Python script file, you can find a `handle` method to
handle this command evoke with arguments such as data file path.

The logic of the `handle` method is very simple, it iterates the `movie.csv` file, get each movie row, and populate objects of `Movie` and call the `save()` method to save them into the `movie` database table.

Now you can run the `load_movies` command

- First, make sure you are in `home/project/recommender` folder,
- Then run the following cmd with `path` argument points to `movies.csv`
  in the same folder

```
python3 manage.py load_movies --path movies.csv
```

You should see movies are saved into database one by one, and the entire process may take several minutes to complete.

There will be more than 40000 movies loaded into our database so it
should be a relatively comprehensive movie database for our movie recommender.

To save your works in progress, please constantly save the project files in your local computer, cloud storage, or in a `GitHub`/`GitLab` repository.

Review Movie Database with Django Admin Site

Now you may wonder where can I see these movies saved to my database. One typical way is to use some database management tool to open the `db.sqite3` database and run SQL to query data from the `movie` table. However, this may be too troublesome and there is a much easier way to manage the content of a Django app, which is using Django Admin.

Django admin is a powerful tool provided by Django framework for users to manage both the metadata and the content of their apps.

To start using Django Admin, you need to create a super user first:

- Run `createsuperuser` command, and fill in the credentials for your admin user

```
python3 manage.py createsuperuser
```

With `Username`, `Email`, `Password` entered, you should see a message that indicates the super user is created:

```
Superuser created successfully.
```

Let's start our app and login with the super user.

- Then start the development server

  ```
  python3 manage.py runserver
  ```

- Find and click `Launch Application` button on the top menu and
  enter the port for the development server `8000`

When the browser tab opens, add the `/admin` path and your full URL should look like the following

```
https://userid-8000.theiadocker-1.proxy.cognitiveclass.ai/admin
```

- Login with the user name and password for the super user, then you should see

admin site with `Groups`, `Users`, and `Movies` created for us. These are the database tables created during the Django migration process.

If you click the `Movie` entry link, you can see all your loaded movie data. You can search them by movie title, and you can also click a movie to update it. For example, you can mark a movie as `watched`, which will be used later to recommend you with new movies.

Review a Movie Recommendation Django HTML Template

Now you should have reviewed all the data as an admin but we also need to
build a nice HTML page to show these movies for regular users.

Being a web framework, Django provides a convenient way to generate HTML dynamically, that is Django template.
A template contains the static parts of the desired HTML output as well as some special
syntax describing how dynamic content will be inserted.

To simplify your development process, we have provided a skeleton HTML template, which is located in
`movierecommender/templates/movierecommender/movie_list.html`. You may notice we have two identical `movierecommender` in the path, which is a Django name space convention for the Django project to differentiate html files with the same name
but in different apps.

The `movie_list.html` starts with two basic elements: a navigation bar and an nearly empty HTLM body.

In the template, we use Bootstrap, which is a free and open-source CSS framework directed at responsive, mobile-first front-end web development, to help us build a stylized HTML page.

For now, let's just leave this `movie_list.html` untouched and work on the backend first. Once we figure out what kind of data will be presented in this template, we will revisit and update it accordingly.

Create a Movie Recommendation View

The movie list HTML page you created is just a template and static, which requires a Django view to provide data/context to render the movie list HTML page dynamically.

As we discussed earlier, a Django view is a Python function takes HTTP requests as input and generates HTTP response as output.
Here we want to create a view function to generate a movie list HTML page as a response.

Let's see how to do that.

- Open `movierecommender/views.py`, copy the following code to
  add a very simple view called `movie_recommendation_view`

  ```
  def movie_recommendation_view(request):
      if request.method == "GET":
        # The context/data to be presented in the HTML template
        context = {}
        # Render a HTML page with specified template and context
        return render(request, 'movierecommender/movie_list.html', context)
  ```

This view is very simple, it first checks if it receives a `GET` request, then create an empty dict called `context` to contain the context or data to be presented in the HTML template.

Then it calls `render()` method to render an HTML page with `movie_list.html` template
and `context`.

Configure the URL of `movie_recommendation_view`

Next, configure the URL for the `movie_recommendation_view` so that HTTP request with matching URL pattern will be forwarded to `movie_recommendation_view` by Django.

- Open `movierecommender/urls.py`, add the following code:

  ```
  path(route='', view=views.movie_recommendation_view, name='recommendations'),
  ```

to `urlpatterns` list to make it looks like the following:

```
from django.urls import path
from . import views
urlpatterns = [
    # route is a string contains a URL pattern
    path(route='', view=views.movie_recommendation_view, name='recommendations'),
]
```

As such, we want to make any url with pattern `app_url/movierecommender` be forwarded to `views.movie_recommendation_view`

That's it, now let's test the movie view and template together.

- Run Django sever if not started:

  ```
  python3 manage.py runserver
  ```

- Clicking `Launch Application` and enter `8000`.

After a new browser tab is opened, add a `/movierecommender` path to your full URL,
note that the `userid` is a place holder and should be replaced by your real id shown after clicking `Launch Application`

```
https://userid-8000.theiadocker-1.proxy.cognitiveclass.ai/movierecommender
```

Django will map any HTTP requests starting with `/movierecommender` to the app `movierecommender` and search any matches for paths defined in `movierecommender/urls.py`.

Now you should see a nearly empty HTML page with `Movies to be shown here!`.

Query and Add Movie Records to the Context

Next, let's query the Movie records from the database, and put them into the `context` dict.
As such the `movie_recommendation_view` can render the template `movies_list.html` with actual movies.

- Copy the following query method into `movierecommender/views.py`

  ```
  def generate_movies_context():
      context = {}
      # Show only movies in recommendation list
      # Sorted by vote_average in desc
      # Get recommended movie counts
      recommended_count = Movie.objects.filter(
          recommended=True
      ).count()
      # If there are no recommended movies
      if recommended_count == 0:
          # Just return the top voted and unwatched movies as popular ones
          movies = Movie.objects.filter(
              watched=False
          ).order_by('-vote_count')[:30]
      else:
          # Get the top voted, unwatched, and recommended movies
          movies = Movie.objects.filter(
              watched=False
          ).filter(
              recommended=True
          ).order_by('-vote_count')[:30]
      context['movie_list'] = movies
      return context
  ```

Basically, this function has the following application logic:

- Create an empty context dict
- Use Django Models API `Movie.objects.filter(recommended=True)` to query any movies in `Movie` table with recommended field to be `True`

With Django Models APIs, you do not need to write any SQL to query database tables
This will return a QuerySet for further filter or aggregation. Then we have add a `count()` method to basically get the total number of recommended movies in the databases.

- Then we added an If-Else statement to handle two different user scenarios.
- If there are no recommended movies, we query the database again to get the top 30 popular unwatched movies,

order by `vote_count` desc.

- If there are some recommended movies, we query the database to get the top 30 recommended movies.
- Add the queried movies to `context` as context `['movie_list'] = movies`.
  Return the `context` dict for `movie_recommendation_view`

Note, we do not have any watched or recommended movies at the beginning so we just return the top-30 popular movies by default.
Later in this project, we will discuss how to mark a movie as watched and how to generate movie recommendations.

Next, we need to update `movie_recommendation_view` to render the `movie_list.html` with the queried movies context.

```
context = generate_movies_context()
```

Now your `movie_recommendation_view` method should look like the following:

```
def movie_recommendation_view(request):
    if request.method == "GET":
      # The context/data to be presented in the HTML template
```

```
            context = generate_movies_context()
            # Render a HTML page with specified template and context
            return render(request, 'movierecommender/movie_list.html', context)
```

Update the Movie Recommendation Django HTML with Context

Now, we have the backend service implemented to provide the movie data to the UI. Then we need to update the `movie_list.html` to present them.

- Open `movierecommender/templates/movierecommender/movie_list.html` add the following code snippet under the

comments `<!--Render Movie list as card columns here -->`

```
    <div class="container">
        <div class="card-columns">
            <!--For each movie in the movie list, add a movie card-->
            {% for movie in movie_list %}
            <div class="col-auto mb-3">
                <div class="card">
                    <!--Movie Poster image-->
                    <img class="card-img-left"
                         src="https://image.tmdb.org/t/p/original/{{movie.poster_path}}"
                         width="320" height="480">
                    <div class="card-body bg-light">
                        <!--Movie title-->
                        <h5 class="card-title">{{movie.original_title}}</h5>
                        <!--Movie overview-->
                        <p class="card-text">{{movie.overview}}</p>
                        <!--Movie genres-->
                        <p class="card-text">Genres: <b>{{movie.genres}}</b></p>
                        <!--Movie vote average-->
                        <p class="card-text">Rating: <b>{{movie.vote_average}}</b></p>
                        <!--Movie language and release date-->
                        <p class="card-text">{{movie.original_language}}, {{movie.release_date}}</p>
                    </div>
                </div>
            </div>
            {% endfor %}
        </div>
    </div>
```

Here we present each movie as a Bootstrap Card element, and organize movie cards wrapped into a `card-columns` as their main container.

In the above HTML code snippet, you may notice some special tags wrapped within `{% %}` or `{{ }}`. These are the Django Template tags used for managing the dynamic content of the HTML page. The code within the tags will be interpreted as the Python code.

Next, we want to iterate the `movie_list` (included in the `context` dict generated by the movie view) provide by the view.
For each movie, we add a `<div class="card">` movie card accordingly:

```
{% for movie in movie_list %}
...
{% endfor %}
```

For each movie card, it contains elements like movie poster image, movie title, genres, etc. Their actual values will be obtained from the `{{movie}}` variable tag. Django template will populate the `{{movie}}` variable with the actual movie Python object in the `movie_list` Python list.

Now let's test our view and updated template.

- Run Django sever if not started:

```
python3 manage.py runserver
```

and Launch Application with port 8000 and add `/movierecommender` path, you should see many nice movie cards similar to the following screenshot:

To save your works in progress, please constantly save the project files in your local computer, cloud storage, or in a `GitHub`/`GitLab` repository.

Generate Movie Recommendations based on Watched Movies

OK, for now the movie recommender simply returns the most popular movies, and you might have watched all of them already!

To make the recommendation actually work, you need to first mark the movies you have watched using Django Admin site.
Then you will write a recommendation algorithm based on your watched movies.

Mark the Watched Movies in Django Admin

- Run Django sever if not started:

```
python3 manage.py runserver
```

and Launch Application and go to URL `app_url/admin`

If you click the `Movie` entry link, you can see all your loaded movie data in the previous step.

Now try to recall, as many as possible, what good movies you have watched before, and search them by title.
Then you click into the movie entry and mark it as `watched` and press Save.

Repeating the above step to create your own watched movie history, and the recommender algorithm will recommend new movies to you based on the history.

Run make_recommendations CMD to Generate Recommendations

Now it comes to the most interesting part, that is, how to recommend new movies based on your watched movies?

For any recommendation systems, the key idea is always to come up with a good algorithm/model to predict if a specific user will like or dislike his/her unseen item, as shown in the following screenshot:

There are probably hundreds of good recommendation algorithms and can be roughly divided into two categories:

- `Content filtering based`: The content filtering based recommendation algorithms assume you may like a new movie if you have watched very similar movies before. Or based on your user profile (like age, gender, interests), it will try to find new movies matching your profile.
- `Collaborative filtering based`: The collaborative filtering algorithms assume you may like a new movie if other users similar to you (similar profile or watched similar movies) have watched this movie.

In this project, we will use content filtering based algorithm, and we will try to recommend unwatched/new movies to you if they are similar to your watched movies.

Then how do we calculate such movie similarity? There are also many ways to do that, depends on your data and problem settings. Some popular ones include Jaccard similarity, Euclidean Distance, Cosine Similarity, or even using various Neural Networks.

Here we will use Jaccard similarity which is probably the simplest but very effective method to calculate similarity between two sets.

Jaccard Similarity is defined as the size of intersection of two sets divided by the size of union of that two sets.

```
sim(set1, set2) = len(insertection(set1, set2)) / len(union(set1, set2))
```

which means, the more two sets intersected or shared common elements with each other, the more similar they will be.

The Jaccard similarity is ranged from 0 to 1 or 0 to 100%.

For example, if we have four sets:

- set1 with `[apple, banana, orange]`
- set2 with `[grape]`
- set3 with `[apple, banana, orange]`
- set4 with `[apple, banana]`

Then we can calculate the similarity between set1 to set2, set3, set4 as follows:

- `sim(set1, set2)` = `[ ] (no intersection)` / `[apple, banana, orange, grape]` (union) = 0 (0 %)
- `sim(set1, set3)` = `[apple, banana, orange]` / `[apple, banana, orange]`= 1 (100 %)
- `sim(set1, set4)` = `[apple, banana]` / `[apple, banana, orange]`= 2/3 (~66.7%)

More specifically in our movie case, we will measure the similarity of two movies by comparing their genres. For example, if you watched Action-comedy movies, you are very likely to watch other popular Action-comedy movies.

Suppose we have movie1=`[action comedy]` and movie2=`[action comedy drama]` then

- `sim(move1, movie2)` = `[action comedy]`/`[action comedy drama]` = 2/3 (~66.7%)

so if I watched movie1 before, then I probably will watch movie2 because it has similar genres (66.7%) with movie1.

OK, let's try to apply Jaccard similarity on our movies and determine if they will be recommended or not.

We will create a standalone command to generate recommendations (similar to our data loader command).

- Open `movierecommender/management/commands/make_recommendations.py`, add the `jaccard_similarity` method which takes two Python lists and converts them into two Python sets. Then it applies the Jaccard similarity formula.

```
# Method to calculate Jaccard Similarity
def jaccard_similarity(list1: list, list2: list) -> float:
    s1 = set(list1)
    s2 = set(list2)
    return float(len(s1.intersection(s2)) / len(s1.union(s2)))
```

Then let's add another Python method to actually calculate the similarity between two movies:

```
# Calculate the similarity between two movies
def similarity_between_movies(movie1: Movie, movie2: Movie) -> float:
    if check_valid_genres(movie1.genres) and check_valid_genres(movie2.genres):
        m1_generes = movie1.genres.split()
        m2_generes = movie2.genres.split()
        return jaccard_similarity(m1_generes, m2_generes)
    else:
        return 0
```

At last, let's update the `handle` method in `Command` class to determine the `recommended` boolean field for every unseen movie in the `Movie` database table

```
def handle(self, *args, **kwargs):
```

```
THRESHOLD = 0.8
# Get all watched and unwatched movies
watched_movies = Movie.objects.filter(
        watched=True)
unwatched_movies = Movie.objects.filter(
        watched=False)
# Start to generate recommendations in unwatched movies
for unwatched_movie in unwatched_movies:
    max_similarity = 0
    will_recommend = False
    # For each watched movie
    for watched_movie in watched_movies:
        # Calculate the similarity between watched_movie and all unwatched movies
        similarity = similarity_between_movies(unwatched_movie, watched_movie)
        if similarity >= max_similarity:
            max_similarity = similarity
        # early stop if the unwatched_movie is similar enough
        if max_similarity >= THRESHOLD:
            break
    # If unwatched_movie is similar enough to watched movies
    # Then recommend it
    if max_similarity > THRESHOLD:
        will_recommend = True
        print(f"Find a movie recommendation: {unwatched_movie.original_title}")
    unwatched_movie.recommended = will_recommend
    unwatched_movie.save()
```

Above method first query watched movies and unwatched movies, then for each unwatch movie, calculate the max similarity between all watched movies. If its max similarity is greater than a threshold, let's say 0.8 or 80%, then we update its `recommended` field to be True.

Now, let's run the command in terminal

```
python3 manage.py make_recommendations
```

and you should see many new movie recommendations are generated from the terminal console and our movie view should now show us the top recommended and popular movies.

OK, let's go back to our movie recommendation page and give it a try.

- Click Launch Application with port 8000 and add `/movierecommender` path, you should see the recommended

movies based on how similar to your watched movies as well as how popular they are.

Next Steps

Congratulations, you have built a handy movie recommender, please give it a try with your and your family to see if its recommendations make sense (hopefully it does).

- Improve the movie recommendations by considering more movie attributes/features
  In our previous movie recommendations algorithm, we only consider their genres, which is a simple but good start. In addition, you can also consider the movie title, movie overview, languages and other relevant attributes to improve the movie similarity measurements.
- Add user management to extend the recommender to multi-users
  Refer to Django website to improve your app by adding user management
- Deploy the movie recommender on the cloud

Author(s)

Yan Luo

Other Contributor(s)

Changelog

Date	Version	Changed by	Change Description
2021-08-10	1.0	Yan Luo	Initial version created
2023-01-23	1.2	Lavanya	Updated the markdown to new template