Rotten Potatoes: Final Project Write-Up

By: Mohsin Ismail & Nivethan Sethupathi - DSC478 (Winter 2019)

Application URL: ***http://rottenpotatoes2-env.5qt2cegbay.us-east-2.elasticbeanstalk.com/***
Demo Video URL: ***https://youtu.be/qxxVM28ho8k***
GitHub URL: ***https://github.com/nsethupathi/RottenPotatoes***

Overview

Our project, Rotten Potatoes, provides movie recommendations based on one of two parameters: keywords similarity or ratings similarity. The ratings similarity measure mimics the K-nearest-neighbor algorithm from previous assignments, while the keywords similarity measure identifies movies based on frequency of similar keywords.

The application utilizes two HTML files with basic CSS implementation. Additionally, the application uses a MovieData class to preprocess and store information on the movies, the keywords, and the user ratings. The Engine class contains a variety of functions used in the similarity algorithms. Specifically, the Engine class utilizes three different distance functions, as well as two pairs of functions used for the two possible similarity measures. The Engine's intersection() function further processes keywords data for optimal results.

In this paper we discuss the datasets used in our analysis, steps taken in developing the front-end and back-end of our application, as well as the obtained results and challenges faced throughout the process.

Datasets

The dataset we used for our analysis was sourced from Kaggle and can be found using the link given in the Citation section[1]. For this project, three .csv data files were used:

- 'movies_metadata.csv' : This file contains information on a number of movies, including each movie's ID, Title, Popularity, Genres, and Vote Count.
- 'keywords.csv' : This file contains a list of keywords describing each movie's content and description.
- 'ratings_small.csv' : This file contains a set of user ratings for various movies. The original file, 'ratings.csv' (also found on Kaggle) contains about 26 million reviews and presented too much of a computational challenge to preprocess for the purposes of this project. The 'ratings_small.csv' contains closer to 100,000 ratings and was more appropriate for our purposes.

Much of the preprocessing done on these data files took place in a Jupyter Notebook, which will be included along with the .csv files as a packaged .zip file in the final D2L submission, as well as on the project's GitHub page.

Front-End

The application uses two HTML files: one for the homepage, and one for the results page. The ‘home.html’[a] serves as a landing page for user input. The ‘home.html’ makes use of two radio buttons to select between a “keywords” search or a “ratings” search. Finally, the users input the name of a movie for which they want recommendations.

At this point, the ‘application.py’[c] takes the user input within its function home_form_post() to render a submission template using ‘submitted_form.html’[b]. First and foremost, ‘application.py’ creates an instance of the Engine class, which contains an instance of the MovieData class as a member variable. Thus, the creation of the Engine instance consequently creates an instance of the MovieData class, which in turn performs all the data loading and preprocessing. Depending on whether the user selected “keywords” or “ratings”, the ‘application.py’ makes the call to the Engine class’s get_recommendations() function using the appropriate parameters, which includes a string of the movie title inputted by the user. Next, the application.py handles three possible outcomes.

In the first outcome, the user inputs a movie name not listed within our dataset. When this happens, the Engine’s get_recommendations() searches for the index of the movie, which it subsequently sends to a helper function for further analysis. When the user inputs an unlisted movie name however, the returned movie index is -1, and get_recommendations() returns an array with a single string ‘film not found’ without accessing any further functions. The rendered HTML page notifies the user that their inputted movie does not exist within our dataset. In the second outcome, the user inputs a movie with too few ratings. In this case, the application alerts the user to insufficient ratings data, also by checking a string in the returned array.

The third outcome represents a successful run of the program, in which get_recommendations() returns a string array of five movie titles recommended to the user based on their input parameters. By taking the user’s inputted string ‘title’, get_recommendations() makes further function calls respective to the similarity parameter (‘keywords’ or ‘ratings’) defined by the user, returning movie titles closest in similarity to the inputted movie. The rendered HTML page prints the inputted movie name, the similarity search used, and finally the movie titles. This post-submission page also has a link to the home.html in case the user wants a new search.

Back-End

The recommendation engine relies on two main python source code files to run: ‘Engine.py’, which contains the code for the Engine class, and ‘Data.py’, which contains the MovieData class. MovieData stores the movie, keywords, and ratings data imported from the dataset into DataFrame objects, while the Engine class executes the methods to calculate and provide the movie recommendations to the user.

Upon initialization of an instance of the MovieData class, the class’s __init__() method reads in various .csv files from the ‘data’ folder in the current working directory. Two .csv files, ‘movies_metadata.csv’ and ‘keywords.csv’ are the original data files sourced from the movie dataset we used. The .csv file ‘ratings.csv’ reorganizes the data in the ‘ratings_small.csv’ data file originally found in the dataset. Certain algorithmic calculations during the preprocessing step for this file took an abnormally long time to finish executing, which would not be acceptable for a user actually searching for movie recommendations. In order for our recommendation engine to run efficiently, we preprocessed the data gathered from ‘ratings_small.csv’ beforehand and imported the resulting ‘ratings.csv’ file into the deployed project directly. Additionally, we generated a fourth .csv file, ‘ratings_map.csv’ during the preprocessing stage. The data stored in this file contains a simple index map from the ratings data to the associated movies, and calculates recommended movies from similar ratings information. The code used to preprocess the ratings data and generate the ‘ratings_map.csv’ file can be found in ‘Data.py.’ Because we skip this step in the final project, we commented out the code in the file.

The Engine class holds an instance of the MovieData class, which gets initialized and preprocessed as a part of the Engine’s initialization. The primary functionality used for finding recommendations rests in the get_recommendations() method, which takes as input parameters a string ‘title’ representing the movie title of interest, an integer ‘k’ representing the number of movies to look for and return, a string ‘option’ which includes one of two values depending on the radio button selected by the user, and a function name ‘metric’ defaulted as the name ‘cosSim’, which defined the similarity metric algorithm used in finding the most similar movies. In addition to ‘cosSim’, there exist two other similarity algorithms within the Engine class: ‘ecludSim’ and ‘pearsSim’, which represent the Euclidean distance similarity metric and Pearson correlation similarity metric, respectively.

The Engine class utilizes two sets of methods to determine recommendations based on either keyword or rating similarity. The functions rec_keyword() and neighbors_keyword() rely on finding the movies with the highest number of similar keywords to the movie title entered. rec_keyword() takes as input parameters the DataFrame ‘mov’ containing information on the movies, a movie ID, and a parameter ‘k’ to specify how many movies to find. This function calls neighbors_keyword(), iterates through the movies DataFrame, compares each entry to the inputted movie ID by finding the number of keywords in common, and stores the ID and number of keywords in common within a list of tuples. Next, the function sorts this list in descending order based on the number of keywords, returning the top movie IDs. These movie IDs are then used to find the associated movie titles in the movie DataFrame. The functions rec_rating() and neighbors_rating() execute in a similar fashion to their counterparts previously described, except that they employ a k-Nearest-Neighbor algorithm based on one of the three listed similarity metrics to find movies with rating information most similar to the inputted movie.

The Appendix section of this report includes screenshots of sample output from the final application, as well as code snippets. Additionally, this report includes a link to our project’s GitHub repository which contains all of the relevant files used in developing this project. The repository will have a README file detailing how to download and try the recommender system in your own IDE.

Results

Of the two methods that we employed for finding recommendations, we found that the ones based on keyword similarity were more relevant and appropriate given the inputted movie title than those based on ratings similarity, although both methods had their own benefits and compromises.

For example, the recommended movies for ‘Despicable Me’ based on keyword similarity[d] represents an approximate average case scenario for our application using keywords as a metric, where the list of recommended movies can be seen as mostly accurate with some titles included that appear to be unrelated in content to the movie entered. In the case of ‘Despicable Me’, the keywords[e] for two of the recommended movies, ‘Despicable Me 2’ and ‘300: Rise of an Empire’ compared to the list of keywords for ‘Despicable Me’ are presented in the Appendix. We can see that the keywords in common between ‘Despicable Me’ and ‘Despicable Me 2’ are ‘father daughter relationship’, ‘minions’, ‘duringcreditsstinger’, and ‘3d’. Where the first two keywords are relevant to the plot of both movies, the last two are more generic in nature. Comparing these to the keywords in common for ‘Despicable Me’ and ‘300: Rise of an Empire’ which are ‘minions’, ‘duringcreditsstinger’, and ‘3d’ we can infer that ‘minions’ is included even though it carries a different meaning in relation to each movie, while the same two generic keyword terms are included as well. This illustrates how, more often than not, keyword similarity allows our engine to recommend movies similar in content, but occasionally there can be a match based on other factors as well.

The output showing recommended movies for ‘Iron Man’ based on keyword similarity[f] serves as an example of a much better sample case, where the recommended movies are very much in line with what a user might like to see after seeing ‘Iron Man’. Based on the list of keywords[g] comparing ‘Iron Man’ and ‘Iron man 2’, we can see several keywords in common such as ‘marvel comic’, ‘superhero’, ‘based on comic’, ‘aftercreditsstinger’, and ‘marvel cinematic universe’ which all of the recommended movies likely share in common. In cases like this, the results are much more accurate. On the opposite end of the spectrum, the recommended movies for ‘Toy Story’ based on keyword similarity[h] has movie titles which have very little in common with ‘Toy Story’ and serves as an example where recommending movies based on keyword similarity can fall short.

In comparison, while recommended movies based on ratings similarity[j][k] may not be similar in content or plot, they represent movies that users rated most similarly to the inputted movie title. One way to understand this distinction is to see the keywords metric as returning “Movies similar to this one”, whereas the ratings metric returns “Other movies that viewers liked”.

Discussion & Challenges

Some challenges faced when developing the front end included accounting for the three possible outcomes after the user submission. Originally, a failed search due to either a non-existent film or a failed search for a film with too few ratings both culminated in the Engine’s get_recommendations() function returning an empty array. However, we wanted the program to specifically state why a particular attempt failed. Originally, we wanted the program to check whether an array was empty, and then based on the similarity parameter specified by the user, alert the user to why the application yielded no results. For instance, an empty array with a parameter of ‘keywords’ alerts that the film was not found, while an empty array with the parameter ‘ratings’ alerts that there were too few ratings.

However, we realized that this sometimes provides inaccurate explanations. For instance, if the user selected the parameter ‘ratings’, and the movie did not exist in our dataset, then the application incorrectly alerts the user to insufficient ratings instead of the film not being found. Therefore, we decided to implement a system in which the returned array carried a string stating why the application failed to yield recommendations (either ‘film not found’ or ‘insufficient ratings’). ‘application.py’ checks the string, and notifies the user accordingly.

Deployment of the application also proved problematic, especially when attempting to use Google App Engine. Google App Engine prevents the uploading of files greater than 32MB, which prevented our movies_metadata.csv from uploading. We worked around this issue by uploading a .zip file of the data, which the application unzips upon execution. However, further complications led us to abandon Google App Engine. Google services apparently utilize Python 2.7, which caused discrepancies during our attempts to upload the libraries numpy and pandas. Instead, we deployed the project using Amazon Web Services.

The development of the recommendation itself also presented several of its own challenges. From the start of the project, we were restricted by our limited computing power in our ability to preprocess large amounts of data from the source datasets. For example, where the full data file of user ratings contained over 26 million individual user ratings, we had to include in our engine the abbreviated dataset which contained just over 100,000 user ratings. Even still, the preprocessing stage took quite a long time to complete with this shortened dataset. Because of this, many movies included in our preprocessed movie dataset were not present in our preprocessed ratings dataset because of a lack of user ratings.

With respect to our recommendation algorithms, each method presented its own set of flaws that could be improved upon. Our keyword similarity metric found those movies with the most keywords in common with our inputted movie title; however, the problem with this approach that was observed in our results was that certain movies were associated with a much larger number of keywords than others. Because of the nature of our algorithm, these movies had a much higher probability of being included as recommended movies for unrelated movie titles that a user might input.

Additionally, our ratings similarity metric found recommended movies based on the closeness in user ratings across movies. However, because we could only include user ratings for movies that were present in our preprocessed movies DataFrame, several user ratings were discarded during the preprocessing stage. Because we had ratings from about 700 users, and no user had ratings for every single movie in our dataset, we had to fill in the missing rating values from a user for a certain movie by taking the average of the rest of the ratings for a specific movie. This may have skewed the results since the ratings were smoothed out more than initially intended.

Conclusion

Of the two parameters for keywords similarity and rating similarity, the keywords appears to provide consistently better results. This probably occurs because the keywords between movies refer to similarities in plotline, characters, etc. Ratings similarity, on the other hand, reflect user preferences. Individuals who rated one movie highly may also have rated movies highly in a completely different genre, and as such follows a more subjective set of standards.

Looking forward, we would want to experiment with other methods of determining recommendations that are based on several different attributes of our movies. In addition, we would benefit greatly from having our datasets stored in a database that could be dynamically updated, whenever any new movies are added, or additional user ratings are entered. This would allow us to look at the changing trend of how movies are received by users, rather than dealing with a static dataset.

This project provided a practical application of concepts covered throughout the course, particularly with respect to preprocessing data and implementation of the k-nearest-neighbor algorithm. Furthermore, this class facilitated the acquisition of new skills, such as web-application development using front-end frameworks such as Flask.

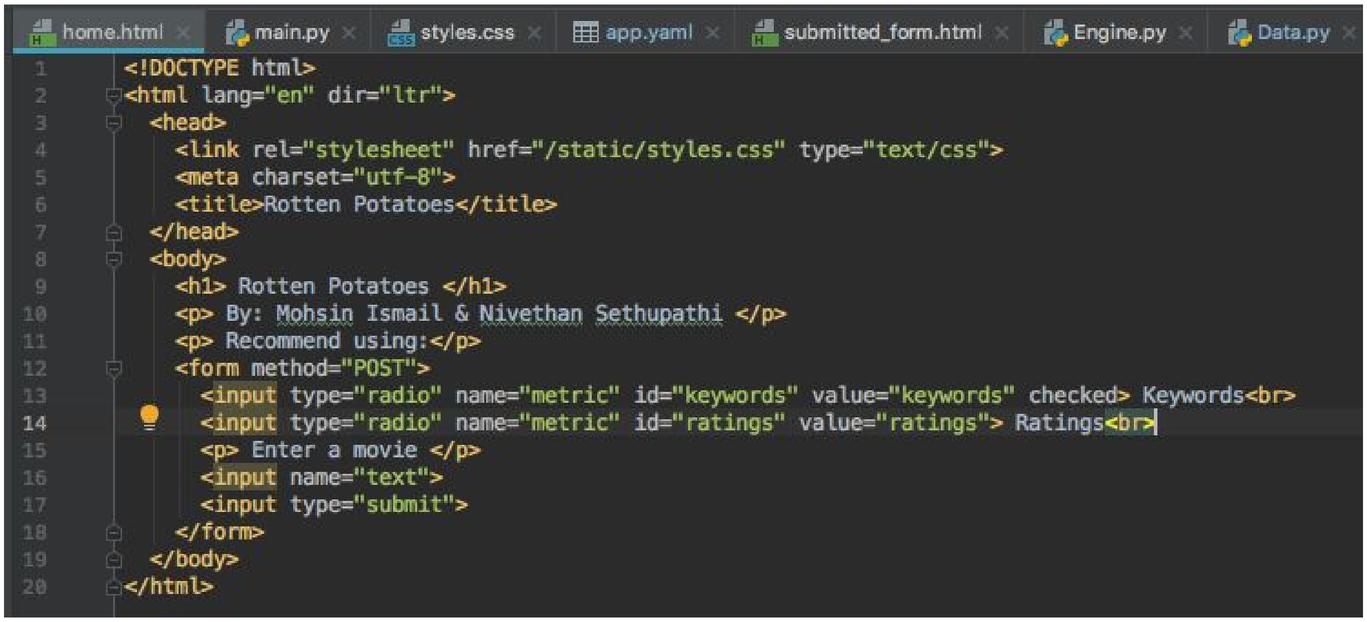
Citations

[1] Source Dataset: https://www.kaggle.com/rounakbanik/the-movies-dataset

Appendix

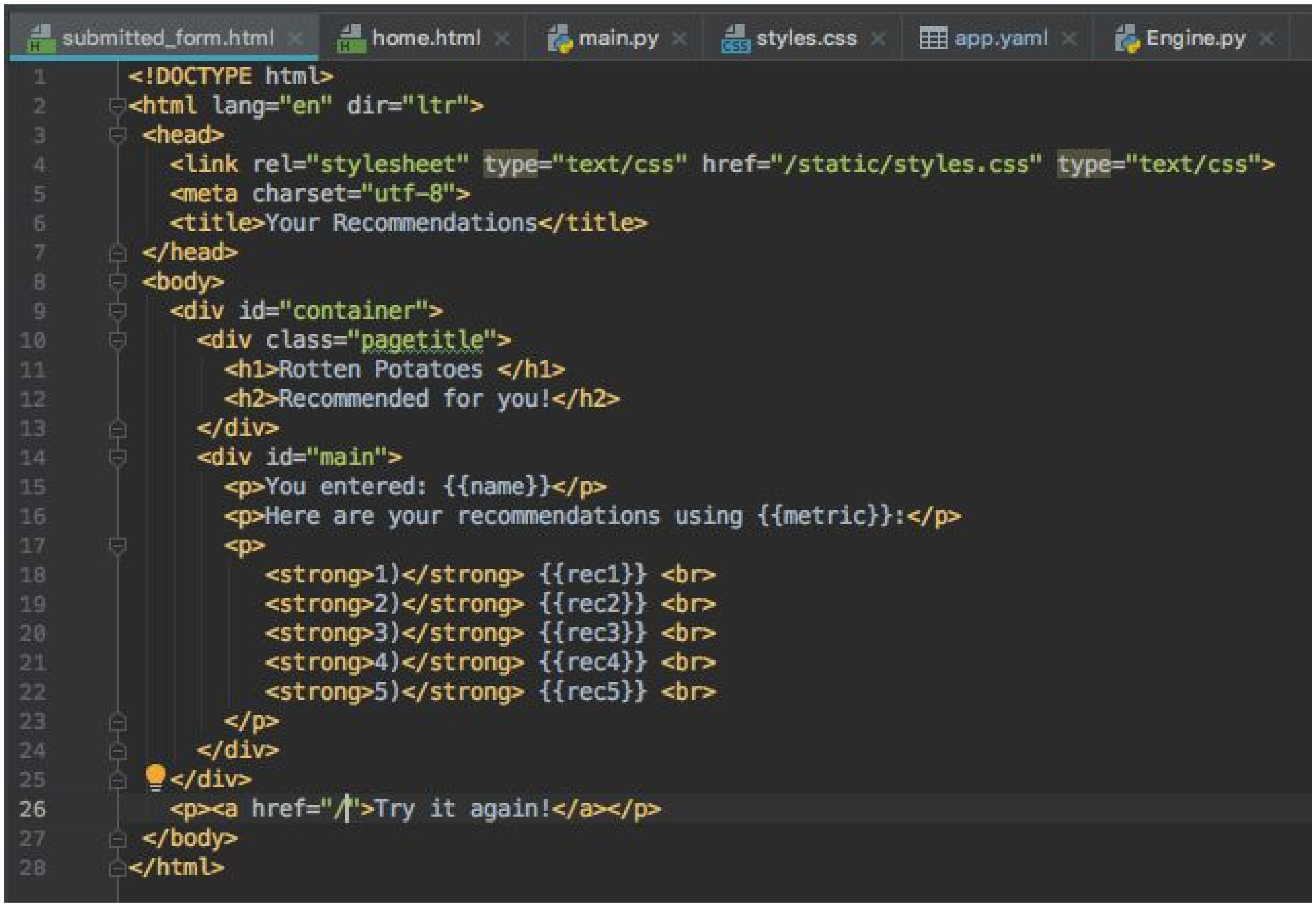
Documented source code, datasets, standalone application files, and our intermediate work done using a Jupyter notebook can all be found on GitHub at https://github.com/nsethupathi/RottenPotatoes. Included below are screenshots of the code we used, sample outputs from the final application build, and supplemental code snippets from our Jupyter notebook used in explaining our results above. We also included on the GitHub page a README file which contains instructions on how to test this application.

[a] home.html



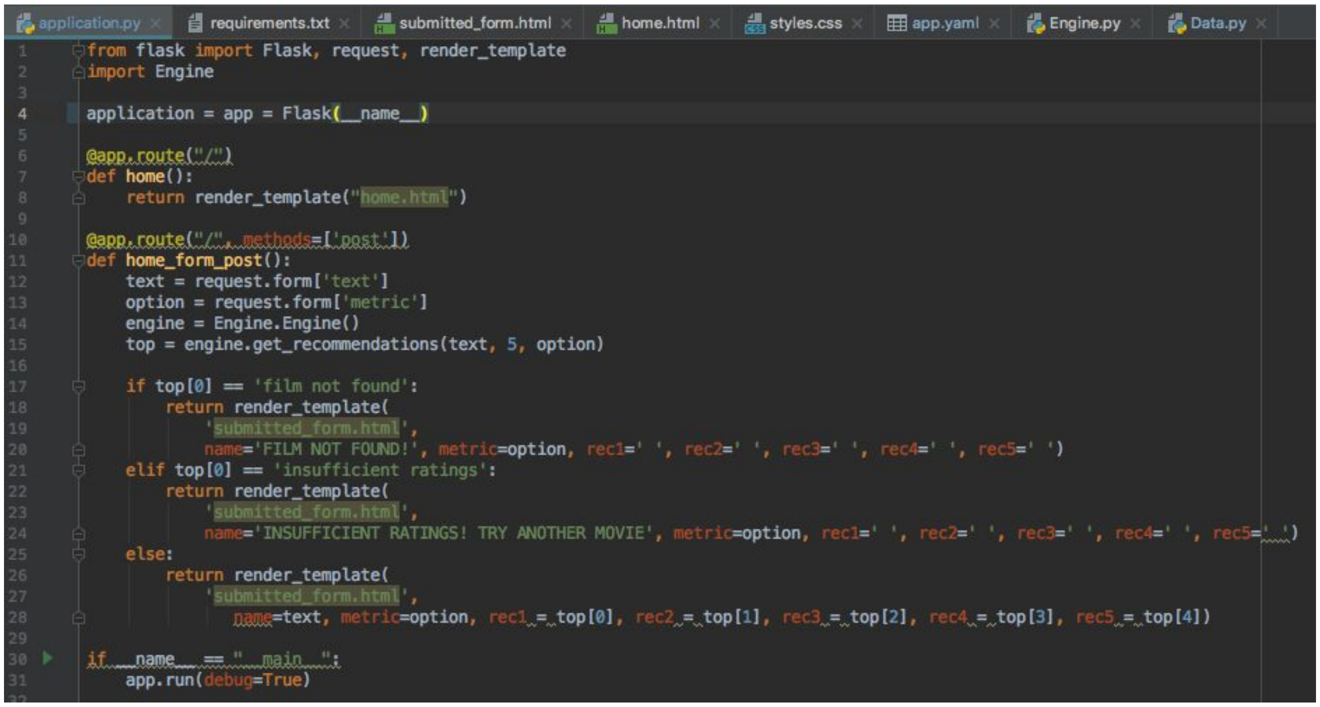
```
<!DOCTYPE html>
<html lang="en" dir="ltr">
  <head>
    <link rel="stylesheet" href="/static/styles.css" type="text/css">
    <meta charset="utf-8">
    <title>Rotten Potatoes</title>
  </head>
  <body>
    <h1> Rotten Potatoes </h1>
    <p> By: Mohsin Ismail & Nivethan Sethupathi </p>
    <p> Recommend using:</p>
    <form method="POST">
      <input type="radio" name="metric" id="keywords" value="keywords" checked> Keywords<br>
      <input type="radio" name="metric" id="ratings" value="ratings"> Ratings<br>
      <p> Enter a movie </p>
      <input name="text">
      <input type="submit">
    </form>
  </body>
</html>
```

[b] submitted_form.html



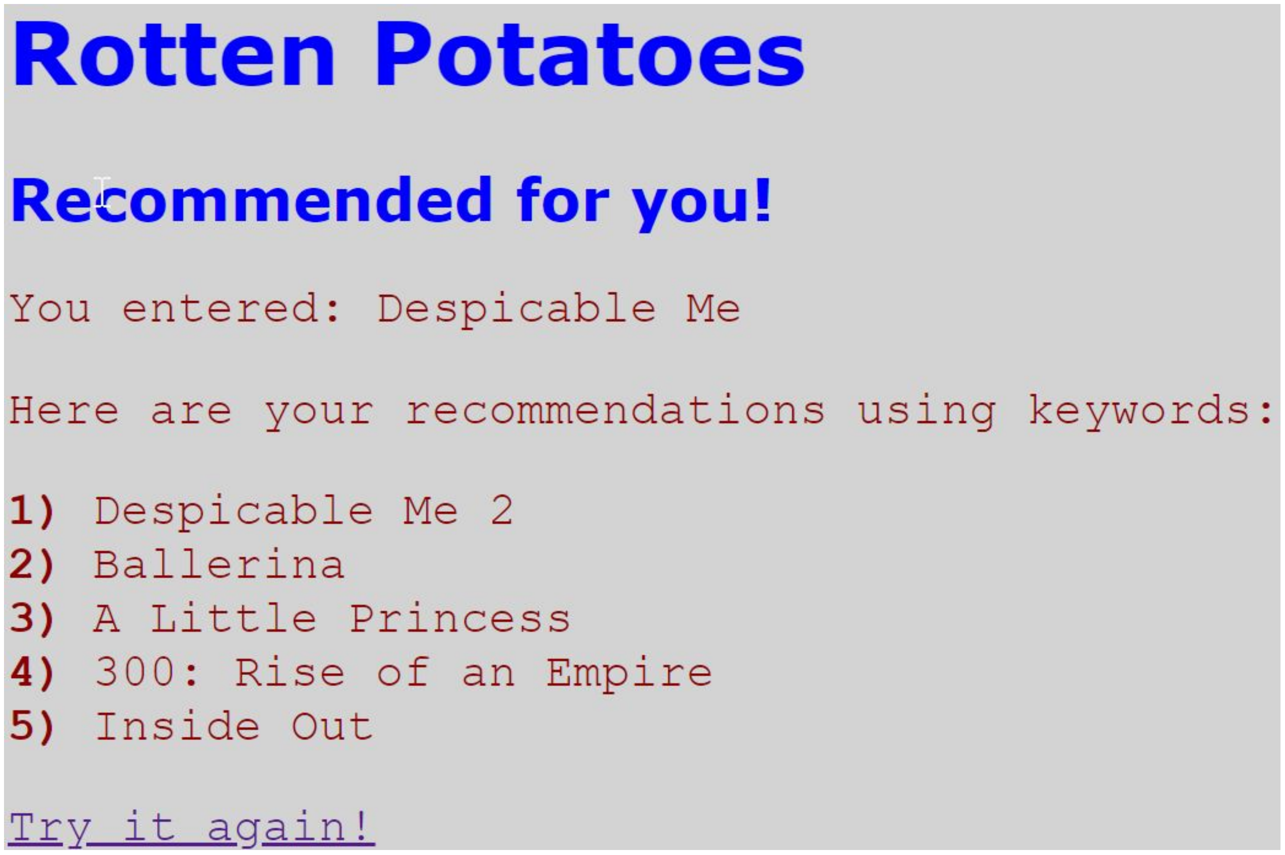
```html
<!DOCTYPE html>
<html lang="en" dir="ltr">
 <head>
   <link rel="stylesheet" type="text/css" href="/static/styles.css" type="text/css">
   <meta charset="utf-8">
   <title>Your Recommendations</title>
 </head>
 <body>
   <div id="container">
     <div class="pagetitle">
       <h1>Rotten Potatoes </h1>
       <h2>Recommended for you!</h2>
     </div>
     <div id="main">
       <p>You entered: {{name}}</p>
       <p>Here are your recommendations using {{metric}}:</p>
       <p>
          <strong>1)</strong> {{rec1}} <br>
          <strong>2)</strong> {{rec2}} <br>
          <strong>3)</strong> {{rec3}} <br>
          <strong>4)</strong> {{rec4}} <br>
          <strong>5)</strong> {{rec5}} <br>
       </p>
     </div>
   </div>
   <p><a href="/">Try it again!</a></p>
 </body>
</html>
```

[c] application.py



```python
from flask import Flask, request, render_template
import Engine

application = app = Flask(__name__)

@app.route("/")
def home():
    return render_template("home.html")

@app.route("/", methods=['post'])
def home_form_post():
    text = request.form['text']
    option = request.form['metric']
    engine = Engine.Engine()
    top = engine.get_recommendations(text, 5, option)

    if top[0] == 'film not found':
        return render_template(
            'submitted_form.html',
            name='FILM NOT FOUND!', metric=option, rec1=' ', rec2=' ', rec3=' ', rec4=' ', rec5=' ')
    elif top[0] == 'insufficient ratings':
        return render_template(
            'submitted_form.html',
            name='INSUFFICIENT RATINGS! TRY ANOTHER MOVIE', metric=option, rec1=' ', rec2=' ', rec3=' ', rec4=' ', rec5=' ')
    else:
        return render_template(
            'submitted_form.html',
              name=text, metric=option, rec1 = top[0], rec2 = top[1], rec3 = top[2], rec4 = top[3], rec5 = top[4])

if __name__ == "__main__":
    app.run(debug=True)
```

[d] Recommendations for 'Despicable Me' using keywords:



Rotten Potatoes

Recommended for you!

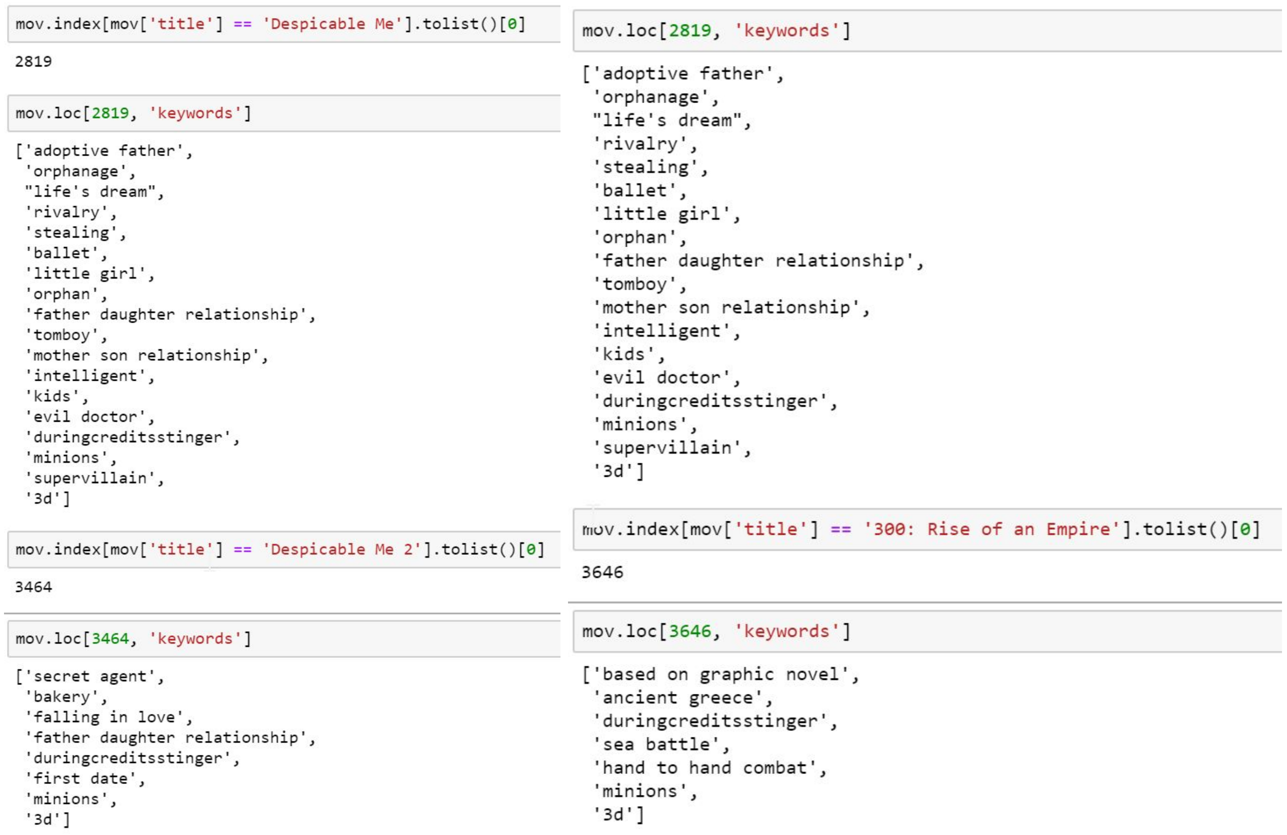
You entered: Despicable Me

Here are your recommendations using keywords:

1) Despicable Me 2
2) Ballerina
3) A Little Princess
4) 300: Rise of an Empire
5) Inside Out

Try it again!

[e] Example of keyword similarity for 'Despicable Me' results:



```
mov.index[mov['title'] == 'Despicable Me'].tolist()[0]
```

```
2819
```

```
mov.loc[2819, 'keywords']
```

```
['adoptive father',
 'orphanage',
 "life's dream",
 'rivalry',
 'stealing',
 'ballet',
 'little girl',
 'orphan',
 'father daughter relationship',
 'tomboy',
 'mother son relationship',
 'intelligent',
 'kids',
 'evil doctor',
 'duringcreditsstinger',
 'minions',
 'supervillain',
 '3d']
```

```
mov.index[mov['title'] == 'Despicable Me 2'].tolist()[0]
```

```
3464
```

```
mov.loc[3464, 'keywords']
```

```
['secret agent',
 'bakery',
 'falling in love',
 'father daughter relationship',
 'duringcreditsstinger',
 'first date',
 'minions',
 '3d']
```

```
mov.loc[2819, 'keywords']
```

```
['adoptive father',
 'orphanage',
 "life's dream",
 'rivalry',
 'stealing',
 'ballet',
 'little girl',
 'orphan',
 'father daughter relationship',
 'tomboy',
 'mother son relationship',
 'intelligent',
 'kids',
 'evil doctor',
 'duringcreditsstinger',
 'minions',
 'supervillain',
 '3d']
```

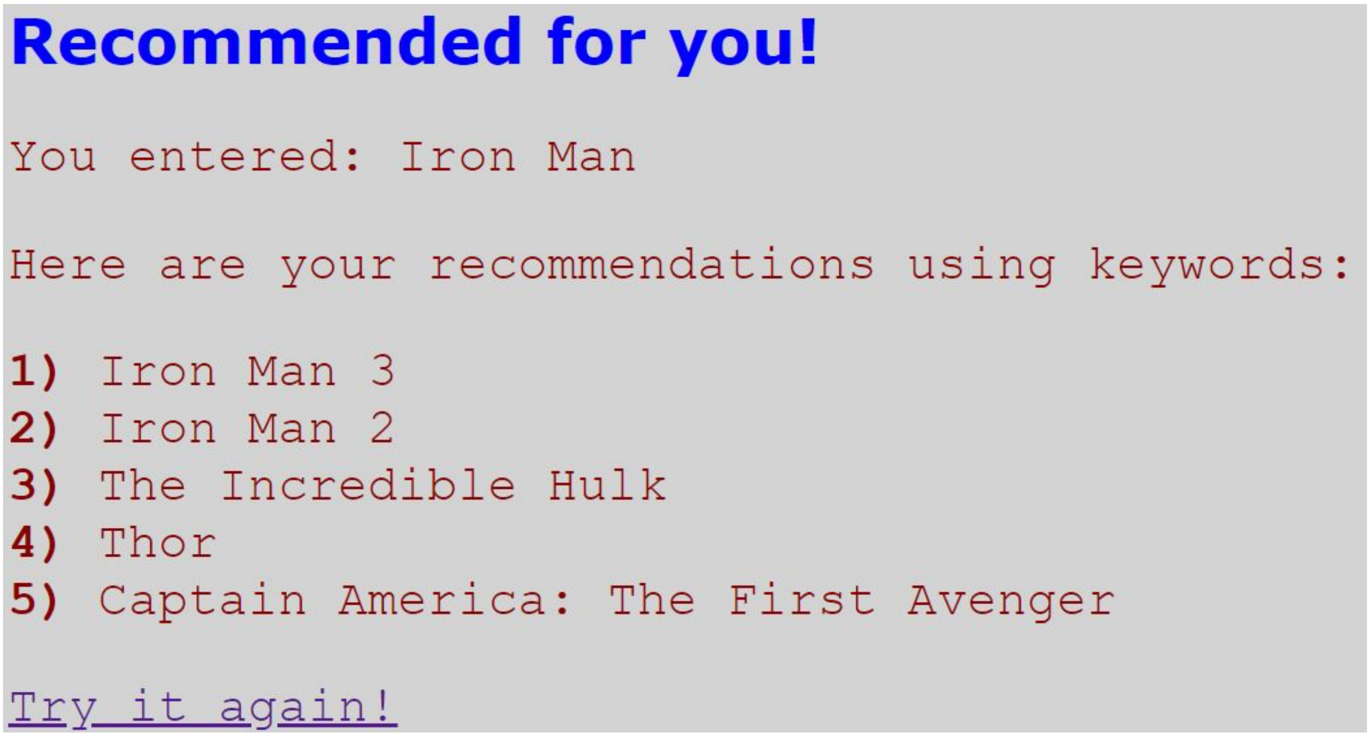
```
mov.index[mov['title'] == '300: Rise of an Empire'].tolist()[0]
```

```
3646
```

```
mov.loc[3646, 'keywords']
```

```
['based on graphic novel',
 'ancient greece',
 'duringcreditsstinger',
 'sea battle',
 'hand to hand combat',
 'minions',
 '3d']
```

[f] Recommendations for ‘Iron Man’ using keywords:



Recommended for you!

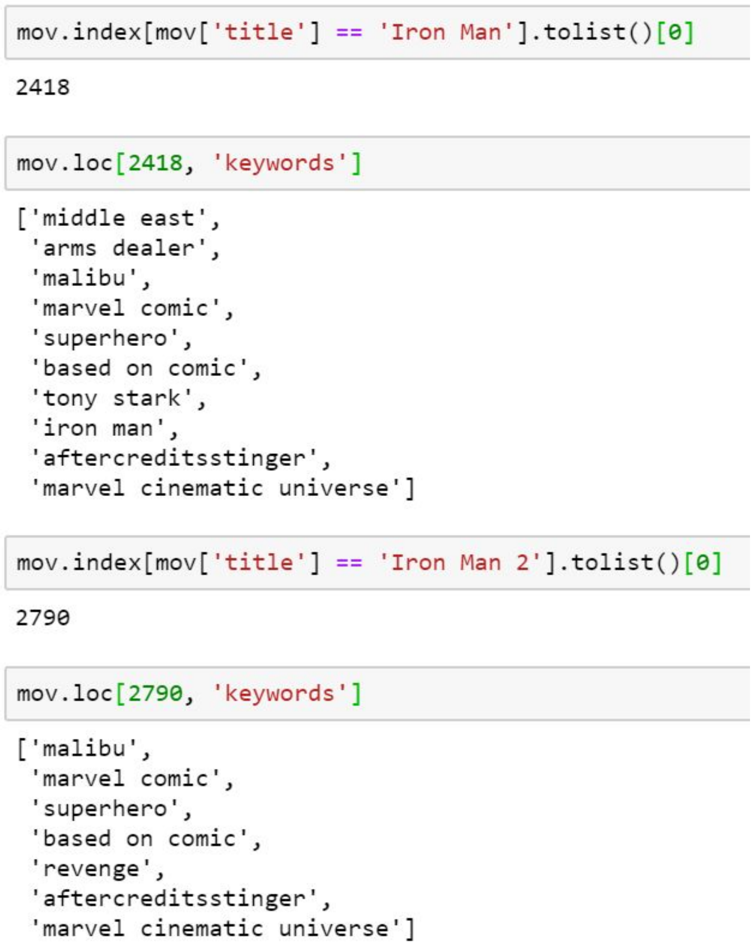
You entered: Iron Man

Here are your recommendations using keywords:

1) Iron Man 3
2) Iron Man 2
3) The Incredible Hulk
4) Thor
5) Captain America: The First Avenger

Try it again!

[g] Example of keyword similarity for ‘Iron Man’ results:



```
mov.index[mov['title'] == 'Iron Man'].tolist()[0]
```

```
2418
```

```
mov.loc[2418, 'keywords']
```

```
['middle east',
 'arms dealer',
 'malibu',
 'marvel comic',
 'superhero',
 'based on comic',
 'tony stark',
 'iron man',
 'aftercreditsstinger',
 'marvel cinematic universe']
```

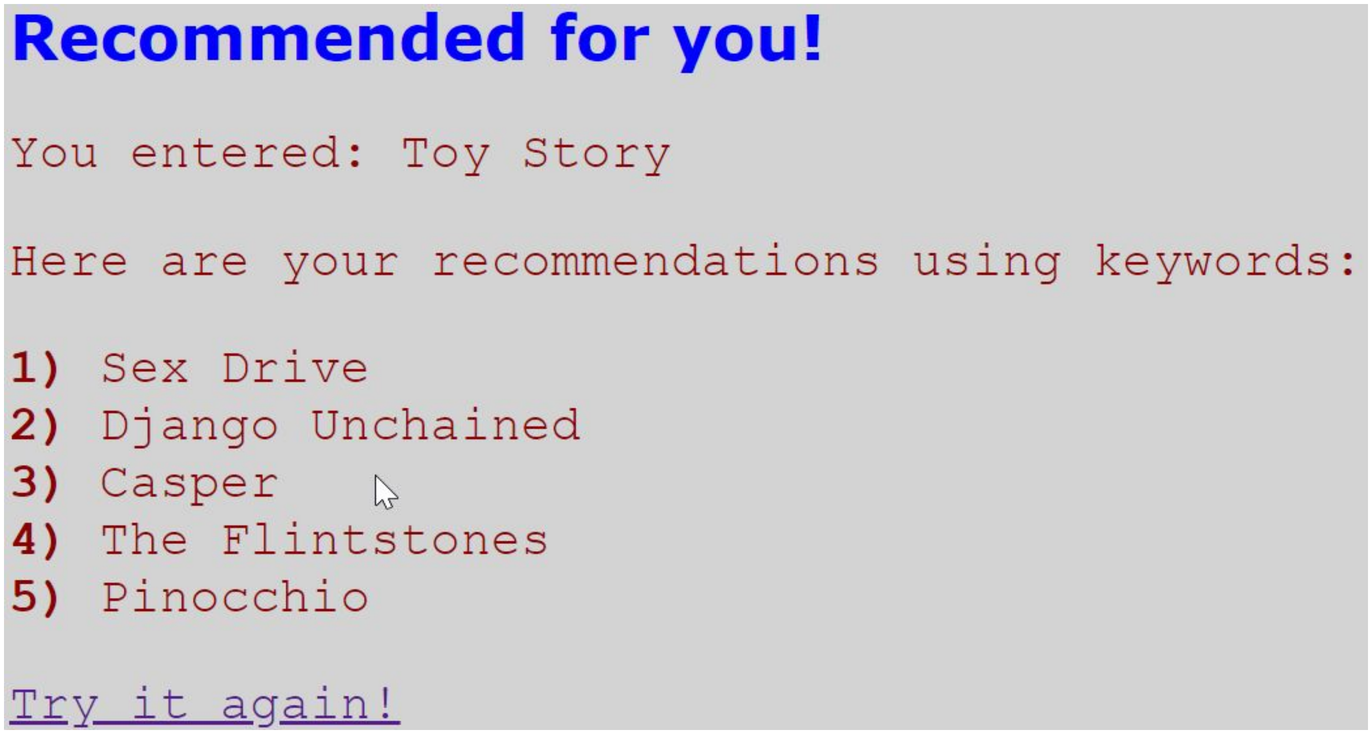
```
mov.index[mov['title'] == 'Iron Man 2'].tolist()[0]
```

```
2790
```

```
mov.loc[2790, 'keywords']
```

```
['malibu',
 'marvel comic',
 'superhero',
 'based on comic',
 'revenge',
 'aftercreditsstinger',
 'marvel cinematic universe']
```

[h] Less successful example of Recommendations for ‘Toy Story’ using keywords:



Recommended for you!

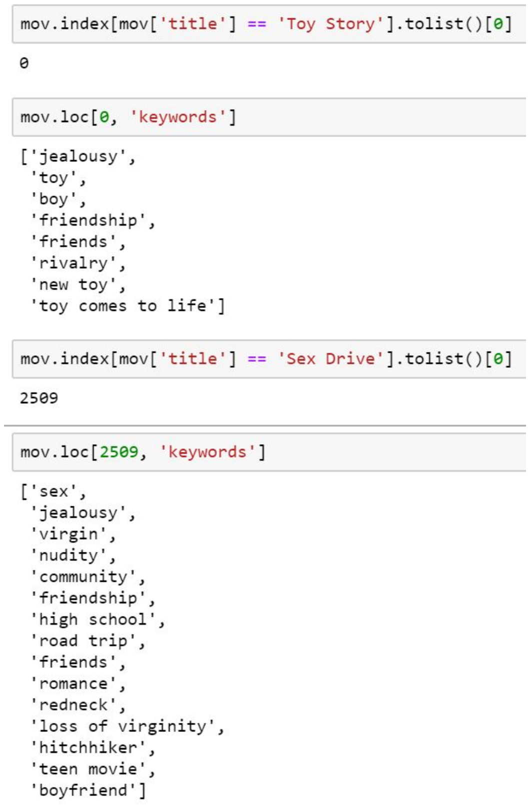
You entered: Toy Story

Here are your recommendations using keywords:

1) Sex Drive
2) Django Unchained
3) Casper
4) The Flintstones
5) Pinocchio

Try it again!

[i] Example of keyword similarity for ‘Toy Story’ results:



```
mov.index[mov['title'] == 'Toy Story'].tolist()[0]
```

```
0
```

```
mov.loc[0, 'keywords']
```

```
['jealousy',
 'toy',
 'boy',
 'friendship',
 'friends',
 'rivalry',
 'new toy',
 'toy comes to life']
```

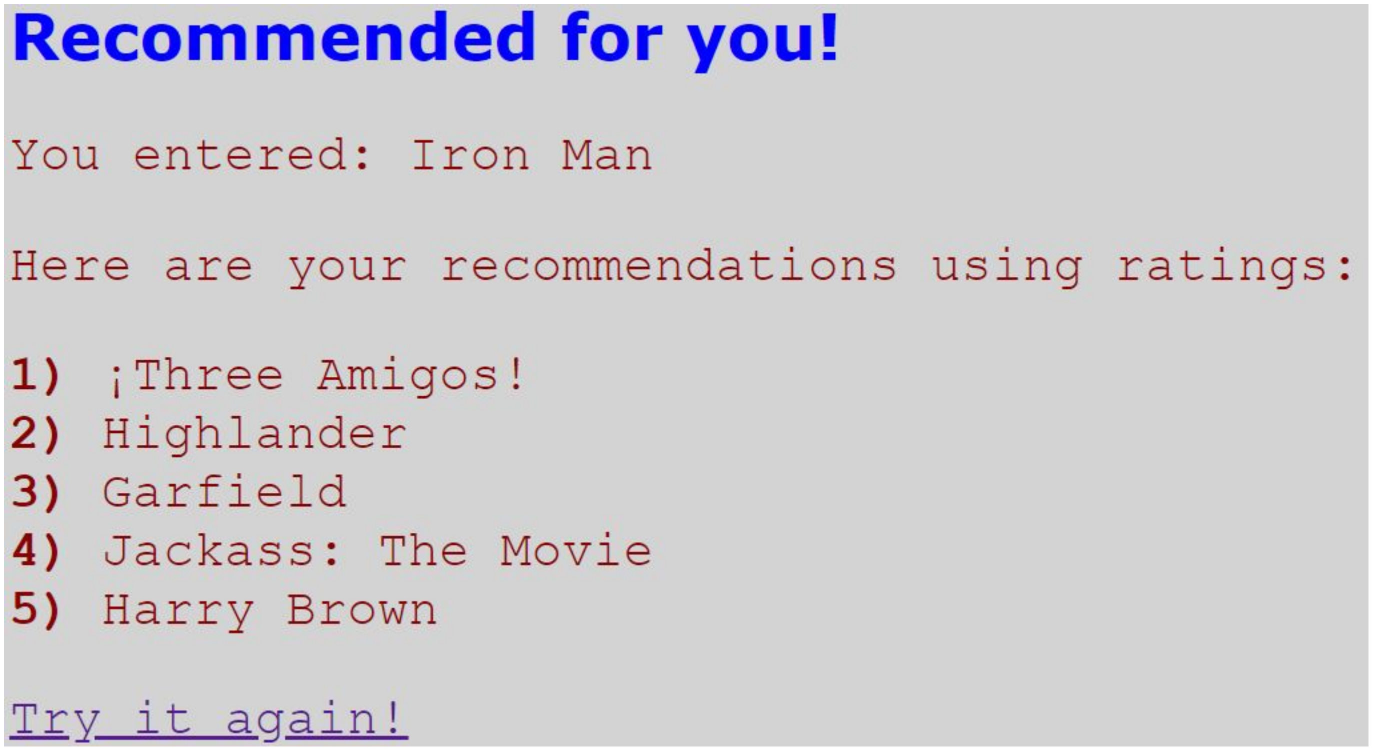
```
mov.index[mov['title'] == 'Sex Drive'].tolist()[0]
```

```
2509
```

```
mov.loc[2509, 'keywords']
```

```
['sex',
 'jealousy',
 'virgin',
 'nudity',
 'community',
 'friendship',
 'high school',
 'road trip',
 'friends',
 'romance',
 'redneck',
 'loss of virginity',
 'hitchhiker',
 'teen movie',
 'boyfriend']
```

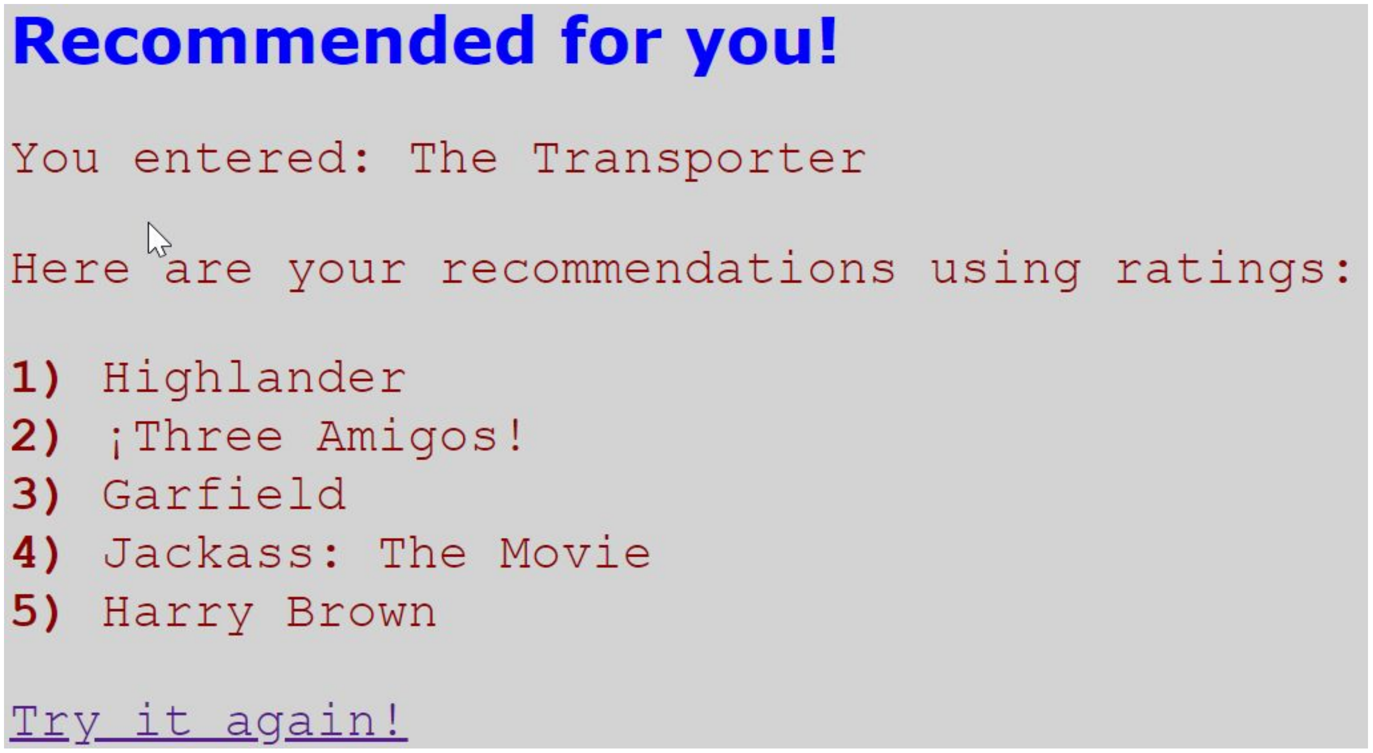
[j] Recommendations for ‘Iron Man’ using ratings:



Recommended for you!

```
You entered: Iron Man

Here are your recommendations using ratings:

1) ¡Three Amigos!
2) Highlander
3) Garfield
4) Jackass: The Movie
5) Harry Brown

Try it again!
```

[k] Recommendations for ‘The Transporter’ using ratings:



Recommended for you!

```
You entered: The Transporter

Here are your recommendations using ratings:

1) Highlander
2) ¡Three Amigos!
3) Garfield
4) Jackass: The Movie
5) Harry Brown

Try it again!
```