

# CS410 Project Final Report – Team DJTV – Intelligent Movie Search

## Table of Contents

Team Members .....	1
Functional Description.....	1
Implementation.....	2
Architecture .....	2
Code Structure .....	3
Implementation Details .....	3
Data Collection and Preparation .....	4
Search Engine .....	5
Sentiment Analysis .....	8
Similar movies recommender .....	10
Web application .....	11
Usage Documentation.....	11
Installation for usage or peer grading .....	11
Installation for future extensions/additions .....	11
Use Cases and Screenshots .....	13
Tasks and Team member contributions .....	16
Team member contributions .....	16
Final Task list and effort.....	16

## Team Members

Danh Nguyen NetID: [danh2@illinois.edu](mailto:danh2@illinois.edu)

Justin Quach NetID: [jmquach2@illinois.edu](mailto:jmquach2@illinois.edu)

Tik On (Johnson) Lui NetID: [tlui2@illinois.edu](mailto:tlui2@illinois.edu)

Vamshidhar Kommineni NetID: [kommineni@illinois.edu](mailto:kommineni@illinois.edu)

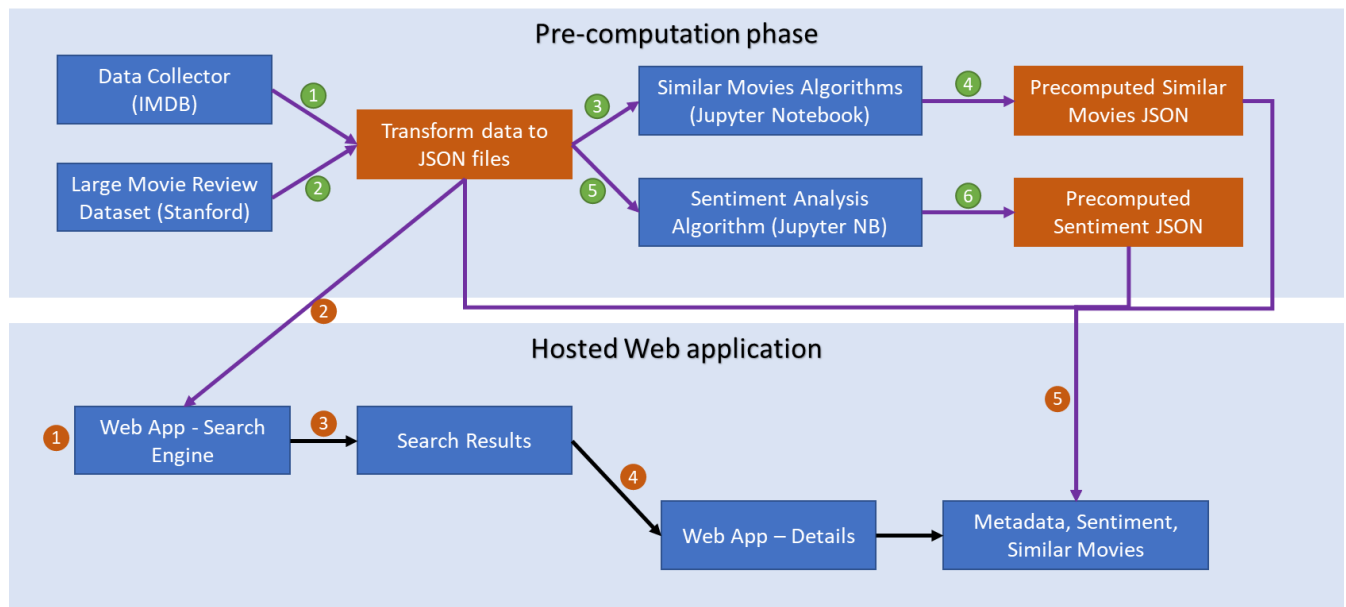
## Functional Description

Our project implements an Intelligent Movie Search web application. This movie search application is “Intelligent” because it implements some key capabilities that we learned through the CS 410 class. The main functions this application provides are:

1. **Search Engine:** The ability to search and find relevant movies by entering a query string. As an advanced feature, you can choose the ranker that is used to rank relevance and return results. There are two kinds of search that are possible:
  - a. **Search by Movie Name:** Find the most relevant movies based on a query string against a text corpus of only movie names. This is a common use case when you already have a movie in mind and want to find out more details about the movie
  - b. **Search by additional metadata:** Find the most relevant movies based on a query string against a text corpus of extended movie metadata.
2. **Movie Details Page:** The ability to view more details about a movie. There are 3 kinds of detail presented about a movie:
  - a. **Structured metadata:** Information like the movie poster, year released, runtime, IMDB rating, genre(s), movie summary, director and cast.
  - b. **Sentiment rating:** The reviews for the movie are analyzed and summarized as a sentiment rating (Positive, Negative, Neutral) with counts of reviews that are classified as each rating
  - c. **Similar movies:** A list of movies similar to the currently listed movie that are mined/discovered using the metadata corpus. You can choose to use one of three sophisticated techniques: BERT, Word2Vec and TF-IDF to display the list of similar movies

## Implementation

### Architecture



Our application consists of two phases:

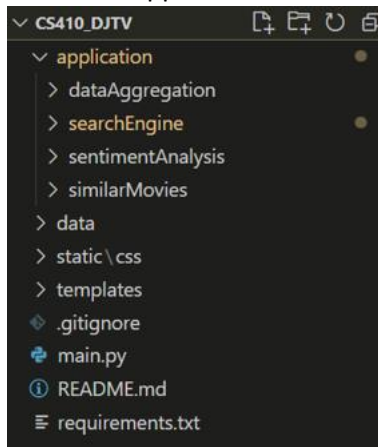
1. **Pre-computation phase:** In this phase, we first collected structured metadata from IMDB using the list of movies in the Stanford “Large Movie Review” dataset, read the corresponding reviews from the dataset and and schematized this data to two JSON files (step 1 and 2 above). We then used Jupyter notebooks to experiment with various algorithms and fine-tuning parameters of

the algorithms. When we obtained promising results for the full dataset, we committed the data on similar movies and sentiment analysis to two static JSON files that can be read later in the hosting process.

2. **Hosted web application:** In this phase, a user can use the search engine feature of the hosted web application. The metadata JSON files are searched (via a metapy ranker of choice) and the top 10 most relevant results are displayed. When the user clicks on a search result, they are taken to the details page which include the sentiment analysis of reviews from the users and a list of similar movies. This data is obtained from the pre-computation phase where they were explored and written

## Code Structure

This is the application structure:



A short description of each of the folders shown above

- **application:** contains different modules and functions
  - **dataAggregation:** The code to fetch and store the movie metadata and review data in a standard JSON format
  - **searchEngine:** The metapy based search engine
  - **sentimentAnalysis:** The Jupyter notebook where the sentiment analysis work resides
  - **similarMovies:** The Jupyter notebook where the similar movies analysis work resides.
- **data:** Contains the review data and metadata of the movies, and prediction results of sentiment analysis and similar movies
- **static:** contains CSS files for the website
- **templates:** contains html files used by the flask application to render the web application pages

## Implementation Details

The implementation details of the five major modules of our application are described below:

1. Data collection and preparation
2. Search Engine
3. Sentiment Analysis

4. Similar movies
5. Web application

## Data Collection and Preparation

### **data\_aggregator.py**

- We leveraged the Stanford “Large Movie Review” dataset (<http://ai.stanford.edu/~amaas/data/sentiment/>) and the metadata of movies from IMDB through a package called CinemaGoer (<https://cinemagoer.github.io/>)
- By getting the movie ID from the review\_url.txt in the reviews dataset, we provided the movie ID into CinemaGoer function to get metadata of the specified movie
- After retrieving the results, required metadata are extracted and combined with the corresponding reviews from the reviews text file

### **data\_seperation.py**

- After several rounds of internal discussion, we found that the file is too large for some of our teammates modules to load into memory. We targeted two separated files to store metadata and reviews of the movies
- However, it is quite time consuming to run the package to get the metadata (since it takes time to get the metadata every time when we call the function) and a bit wasteful (since the data was already downloaded). Therefore, we wrote another python script to separate the data and save time
- json\_zip
  - As mentioned before, the review data was still too large to load into memory. Therefore, json\_zip function was implemented. It compressed the JSON file into a pickle file using the base64 encoding method

Here is the list of metadata extracted from IMDB using the CinemaGoer API:

- localized\_title: Movie Title
- cast: Casting of the movie
- genres: Genres of the movie
- runtimes: Total runtime of the movie
- rating: Rating of the movie
- year: Release year of the movie
- producer: Producer of the movie
- director: Director of the movie
- cover\_url: URL of the movie's cover
- summary: Short Summary of the movie

## Search Engine

The search engine that we created was based on the MetaPy package that was used in this class. Before we could start working on the MetaPy implementation, we needed to implement a data extraction pipeline to create several databases that would be ingested by MetaPy.

The file that does this exact task is called “db\_conversion.py”

```
data = json.load(f)
if os.path.isfile(os.getcwd() + '\movies.dat'):
    os.remove(os.getcwd() + '\movies.dat')

if os.path.isfile(os.getcwd() + '\movies_whole.dat'):
    os.remove(os.getcwd() + '\movies_whole.dat')

if os.path.isfile(os.getcwd() + '\movies_ids.dat'):
    os.remove(os.getcwd() + '\movies_ids.dat')
```

First, we removed all of the previous generated files, if there are any.

```
with open('movies.dat', 'a') as write_file:
    for id in data:
        # print(id, data[id])
        write_file.write(data[id]['localized_title'].strip().lower() + '\n')

with open('movies_whole.dat', 'a', encoding='utf-8') as write_file:
    for id in data:
        info = data[id]['localized_title'].strip().lower()
        for c in data[id]['cast']:
            info += ' ' + c.strip().lower()
        for g in data[id]['genres']:
            info += ' ' + g.strip().lower()
        for p in data[id]['producer']:
            info += ' ' + p.strip().lower()
        for d in data[id]['director']:
            info += ' ' + d.strip().lower()
        info += ' ' + str(data[id]['year'])
        write_file.write(info + '\n')
```

Next, we need to generate 2 files, movies.dat and movies\_whole.dat. These two files are going to be the corpuses of our search engine. Movies.dat is used for the search engine queries that only search based on movie titles. Movies\_whole.dat contains the corpus that will be used in the search engine queries that can search for movies based on names of the actors, year of the movie, genres of the movie, producers and/or directors of the movie.

```
with open('movies_ids.dat', 'a') as write_file:
    for id in data:
        write_file.write(id + '\n')
```

We also created a movies\_ids.dat file as a cache to quickly convert the index within the movie corpuses to the movie ids.

```

source = os.getcwd() + '\movies.dat'
destination = os.getcwd() + '\movies\movies.dat'
try:
    shutil.move(source,destination)
except FileNotFoundError:
    print(source + " was not found")

source = os.getcwd() + '\movies_whole.dat'
destination = os.getcwd() + '\movies_whole\movies_whole.dat'
try:
    shutil.move(source,destination)
except FileNotFoundError:
    print(source + " was not found")

f.close()

```

The rest of this block is just moving files to its corresponding places that will be used later.

After the previous steps, we are ready to start implementing search.py, which is the main python file that contains all of the methods and necessary methods to search for movies.

Within this file, there are several independent functions that will let users...

- ...search for movies simply by checking if the query string exists within the movie title

```

def search(search_param):
    # lower and strip leading + trailing white spaces
    search_param = search_param.strip().lower()
    # result dictionary
    titles = {}
    for movie_id in movie_reviews:
        localized_title = movie_reviews[movie_id]['localized_title'].lower()
        if search_param in localized_title:
            titles[movie_reviews[movie_id]['localized_title']] = movie_id
    return titles

```

- ...search for movies that are within a certain year

```

def find_movies_with_year(year):
    movies = []
    for id in movie_reviews:
        if movie_reviews[id]['year'] == year:
            movies.append(movie_reviews[id]['localized_title'])
            print(movie_reviews[id]['localized_title'] + ' ' + str(movie_reviews[id]['year']))
    if len(movies) == 10:
        break
    print('')
    return movies

```

- ...search for movies with certain ratings that are higher than the specified rating

```
def find_movies_with_gte_rating(rating):
    movies = []
    for id in movie_reviews:
        if movie_reviews[id]['rating'] >= rating:
            movies.append(movie_reviews[id]['localized_title'])
            print(movie_reviews[id]['localized_title'] + ' ' + str(movie_reviews[id]['rating']))
        if len(movies) == 10:
            break
    print('')
    return movies
```

- ...search for movies with certain actor(s)

```
def find_movies_with_cast(actor):
    movies = []
    for id in movie_reviews:
        if actor in movie_reviews[id]['cast']:
            movies.append(movie_reviews[id]['localized_title'])
            print(movie_reviews[id]['localized_title'] + ' Actors: ' + ', '.join(movie_reviews[id]['cast']) + '\n')
        if len(movies) == 10:
            break
    return movies
```

We also have several rankers that users can choose from. One can decide to run the query several times to decide which rankers work best in different situations.

Ranker	Input	Parameters
<b>Okapi BM25</b>	"bm25"	K1=1.2, b=0.75, k3=500
<b>Jelinek Mercer</b>	"jm"	Lambda=0.7
<b>Dirichlet Prior</b>	"dp"	Mu=2000

After several rounds of testing, we decided that these parameters yielded the best performance. The results are highly dependent on the preferences of the human subjects.

Next is the most important piece of the search engine, the search function itself.

We start by creating an inverted index using the various settings that are configured using the config file. There are 2 separate config files that are used within our project:

- Config.toml: this config file is used when you only need to search movie titles and nothing else.
- Config\_whole.toml: this config file is used when you need to search the whole movie corpus that contains other metadata, like actors, years, genres, etc.

```
idx = metapy.index.make_inverted_index(cfg)
```

Next, we load the specified ranker:

```
ranker = load_ranker(ranker_input)
if ranker == -1:
    print('ranker is not set correctly')
    return
```

Next, we run the query using the provided query string. Then, we put all the relevant movie indexes inside the ranked\_all\_relevants object.

```

query = metapy.index.Document()
ranked_all_relevants = None
query.content(query_string.strip().lower())
results = ranker.score(idx, query, top_k)
ranked_all_relevants = results

```

Using the aforementioned movies\_ids.dat, we create a hash map to map all indices to their corresponding ids.

```

all_ids = []
with open('movies_ids.dat') as movies_ids_file:
    for line in movies_ids_file:
        all_ids.append(line.strip())

```

From there, we can easily get all the relevant movie ids, ranked.

```

all_rev_movie_ids = []
for ranked_all_relevant in ranked_all_relevants:
    all_rev_movie_ids.append(all_ids[ranked_all_relevant[0]])

```

Last but not least, we return the final array:

```

return all_rev_movie_ids

```

After several rounds of testing using various queries, the results are fairly accurate. There are several extensions we can make this search engine a bit more powerful. We stopped here in the interest of time and due to the need for the author of this module to focus on other classes.

### Sentiment Analysis

- To produce the sentiment score (positive, neutral, and negative), we are first required to preprocess reviews into a list of words – known as tokenizing. We perform the following to do preprocessing: trim contractions and punctuation, perform lemmatization, and remove any whitespaces as necessary
- Once we have performed preprocessing, we can use VADER (Valence Aware Dictionary for Sentiment Reasoning) to perform sentiment analysis. Once the scores have been computed (in numerical form), we can then run this function:

```

def get_sentiment_label(sentiment):
    if sentiment['compound'] >= 0.50:
        return 'pos'
    elif sentiment['compound'] <= -0.50:
        return 'neg'
    else:
        return 'neu'

```

- This allows us to categorize scores into positive, neutral, and negative, and return the sentiment for that review. Once we have collected all the sentiment scores for that movie, we give a running count on how many reviews have shown positive, neutral, or



negative scores. We then return the sentiment that has the highest amount of positive, neutral, or negative scores as such:

```
highest_key = max(results, key=results.get)
overall_sentiment = highest_key.split('_')[1]

results['sentiment'] = overall_sentiment
results['num_reviews'] = len(reviews)

return results
```

- We split on the overall sentiment since the keys are labeled as 'num\_pos', 'num\_neu', and 'num\_neg'. We want to return the value as 'pos', 'neu', and 'neg'
- We then perform evaluation using the labeled training data for a given movie. Do note that some movies only contain a small number of reviews, in which case the F1 scores will return 0 (for inconclusive; we simply cannot perform training and test data on a movie that contains only 1 review for example).
- We use training and test split on a test size of 0.33 and a random state of 42
- We fit our parameters to model\_svm and use prediction to get our 'y\_pred'. With 'y\_test' and 'y\_pred', we can then use this function as such:

```
f1_scores = f1_score(y_test, y_pred,
                    average=None,
                    labels = ['neg', 'neu', 'pos'])
return {
    'neg': f1_scores[0],
    'neu': f1_scores[1],
    'pos': f1_scores[2],
}
```

- Once we have collected all the necessary information, we get an output as so:

```
{'0064354': {'num_neg': 0,
             'num_neu': 1,
             'num_pos': 2,
             'sentiment': 'pos',
             'num_reviews': 3},
 '0100680': {'num_neg': 0,
             'num_neu': 3,
             'num_pos': 17,
             'sentiment': 'pos',
             'num_reviews': 20},
 '0047200': {'num_neg': 2,
             'num_neu': 3,
             'num_pos': 3,
             'sentiment': 'neu',
             'num_reviews': 8},
```

- We get a movie ID, and that ID contains the number of positive, neutral, and negative scores, along with its highest sentiment key returned (the top movie has a sentiment of positive), with the number of total reviews

### Similar Movies

- Several experiments have been conducted to evaluate which methods can achieve the best result in terms of our evaluation definition
- Performance Evaluator has been implemented to have a standardized evaluation to all experiments
  - As we don't have label data to determine whether the ranking is correct or not, we leverage the genres into to compare between the target movie and similar suggested movies. If they have common genres, we will define they are similar
  - Mean DCG@5 is used as our metrics
- The definition of the movies is constructed by movie title and summary
- For the word preprocessing to have a better generalization, Vocabulary Building Function has been implemented and it includes
  - Changing all the words into lower case
  - Special Characters and stop word removal to remove non-meaningful input
  - Word Tokenization to minimizes text ambiguity
- Different methodologies of calculating movie similarity have been tried and they are:
  - Randomwalk
    - It has been created as baseline comparison; we sample the 5 movies list from the pool without replacement as the similar movies result.
  - Jaccard
  - TFIDF
  - Gensim - Pretrained Embedding
    - By using Gensim package, different pre-trained word embeddings are imported, and cosine similarity is used for calculating the movies' similarity.
  - BERT
    - By using the pre-trained BERT model, we want to get the model output as the embedding to calculate a new movie embedding for similarity checking. Cosine similarity is used for calculating the movies' similarity.
- Experiment Result:

Methodology	Mean DCG@5
Randomwalk	1.64
Jaccard	2.34
TFIDF (with stopword removal)	2.33
TFIDF (without stopword removal)	2.35
GenSim Similarity (word2vec-google-news-300)	2.45
GenSim Similarity (fasttext-wiki-news-subwords-300)	2.20
GenSim Similarity (glove-twitter-200)	2.26
BERT	2.61

- Since there are static dataset (reviews and set of movies in our scope), experiments are done in jupyter notebook. After that, the final prediction outputs are generated in JSON file based on the study we had in the experiments.
  - The result of BERT (which is the best result of the experiments), GenSim (word2vec-google-news-300) and TFIDF (without stopword removal) are stored in JSON format

## Web application

The web application is a simple application with two dynamic web pages (index.html and detail.html) that showcases the search engine, the movie details metadata, the sentiment analysis results and the similar movie results. It is written in Python as a Flask application. The search engine which is also written in python and uses metapy can be called directly and simply from the flask application. The results or additional metadata for movie details are fetched and saved in JSON format. Jinja is used to combine HTML templates with the JSON data and render the final HTML back to the browser.

We made the design decision to use pre-computed results for similar movies and sentiment analysis. This made the web application much faster since it doesn't read to run any code in-line with the web page request for movie details. Rather it just reads a small portion of various JSON files and returns quickly.

Further, because the raw data source (the metadata and reviews) are static (after the pre-computation phase), we cache the inverted index the first time it is generated. This also makes search results return quickly except when the host is warming up a webapp endpoint that hasn't been used in a while.

To make the installation and peer grading process simple, we hosted the application in both Azure as Web App (<http://cs410djtv.azurewebsites.net>) and on PythonAnywhere (<http://komminen.pythonanywhere.com>). This allowed us to have redundancy for the application with the side benefit of learning about two different Python/Flask hosting environments.

## Usage Documentation

### Installation for usage or peer grading

There is no installation needed to just try out the application! Simply visit one of these two web urls:

<http://cs410djtv.azurewebsites.net>

<http://komminen.pythonanywhere.com>

Both web applications are identical. The only reason for dual hosting is to have one serve as the backup of the other if something happens. Note that the first few uses/clicks after a while with no accesses can be slow because of container reloads, warming up data, etc. on the hosting platform.

### Installation for future extensions/additions

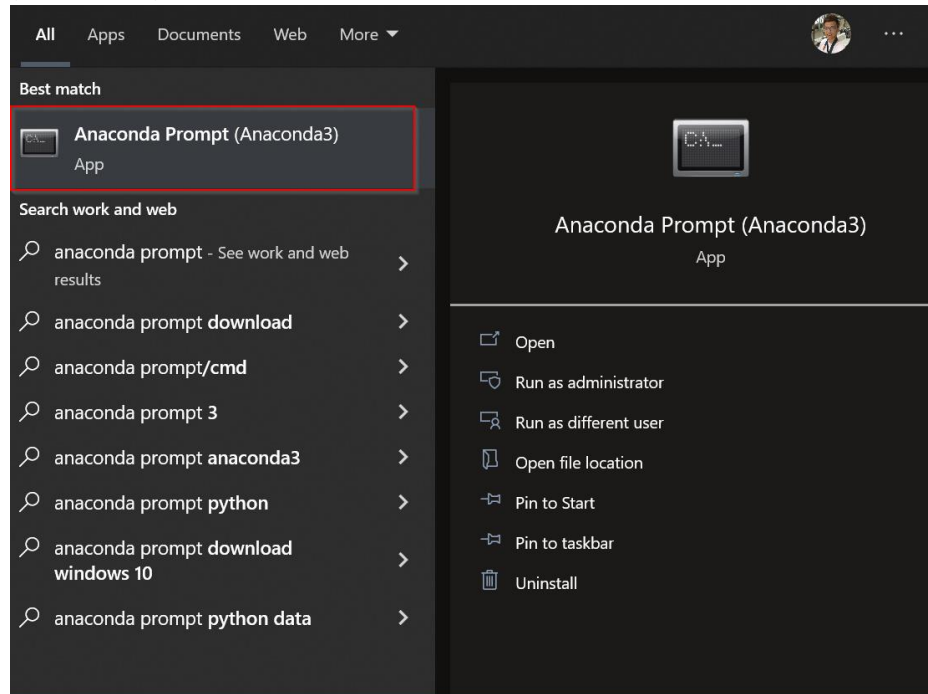
Here are the steps to get our code and start our application:

1. Clone our git repository  
*git clone [https://github.com/komminen/CS410\\_DJTV.git](https://github.com/komminen/CS410_DJTV.git)*

2. Download and install Anaconda (You may skip this part if you have already installed)  
<https://www.anaconda.com/products/distribution>
3. Open a new terminal and make sure you have (base) showing in your terminal

```
(base) C:\Users\j\Anaconda3>
```

- a. For windows user, search “anaconda prompt” and open “Anaconda Prompt (Anaconda3)”



anaconda prompt (Anaconda3)

4. Install Python 3.5 by type the command below  
`conda create -n py35 python=3.5`
5. Switch Python into 3.5  
`conda activate py35`

```
(base) C:\Users\j\Anaconda3>conda activate py35  
(py35) C:\Users\j\Anaconda3>
```

You will see (base) will be changed to (py35)

6. Change directory to the root of the cloned repo

```
(py35) C:\Users\Borislav\Master\CS410\Project\CS410_DJTV>dir
Volume in drive C is Local Disk
Volume Serial Number is 6C43-48F3

Directory of C:\Users\Borislav\Master\CS410\Project\CS410_DJTV

12/05/2022 06:23 PM <DIR>      .
12/05/2022 06:23 PM <DIR>      ..
11/23/2022 02:08 AM          13 .gitignore
11/23/2022 02:08 AM       25,477 2022-10-23_TeamDJTV_ProjectProposal.docx
11/23/2022 02:08 AM       175,670 2022-10-23_TeamDJTV_ProjectProposal.pdf
11/23/2022 02:08 AM       23,271 2022-11-14_TeamDJTV_ProjectUpdate.docx
11/23/2022 02:08 AM       935,420 2022-11-14_TeamDJTV_ProjectUpdate.pdf
12/05/2022 08:05 AM       25,398 2022-12-05_TeamDJTV_FinalReport.docx
11/23/2022 02:08 AM <DIR>      application
12/03/2022 06:04 PM <DIR>      data
12/05/2022 08:19 PM       1,742 main.py
11/23/2022 02:08 AM        248 README.md
12/05/2022 06:23 PM        172 requirements.txt
12/04/2022 12:46 AM <DIR>      static
12/04/2022 01:32 AM <DIR>      templates
                9 File(s)      1,187,411 bytes
                6 Dir(s)      47,957,065,728 bytes free

(py35) C:\Users\Borislav\Master\CS410\Project\CS410_DJTV>
```

7. Install required packages to start the application

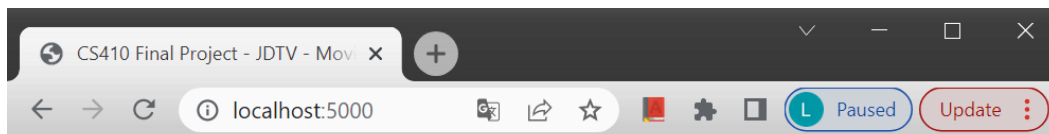
`pip install -r requirements.txt`

8. Start the application

`python main.py`

```
(py35) C:\Users\Borislav\Master\CS410\Project\CS410_DJTV>python main.py
* Serving Flask app "main" (lazy loading)
* Environment: production
  WARNING: This is a development server. Do not use it in a production deployment.
  Use a production WSGI server instead.
* Debug mode: off
* Running on http://0.0.0.0:5000/ (Press CTRL+C to quit)
```

9. Open a browser and enter <http://localhost:5000>. You should see a web application that looks like the screenshot below



## Movie Search

### Search Result

MOVIE POSTER	MOVIE TITLE	YEAR	SENTIMENT
--------------	-------------	------	-----------

### Use Cases and Screenshots

Start by visiting <http://cs410djt.azurewebsites.net> or <http://komminen.pythonanywhere.com>. Use cases/test cases:

1. **Search Engine:** Search for a movie without changing any options: Example search terms are “Batman”, “Superman”, etc. You should obtain a screen shot like the one below:




### Intelligent Movie Search

Choose a Ranker (Default: Okapi BM25)

Choose a Search corpus (Default: Movie Names only)

**Please Note:** The movies in the dataset are restricted to the ~50,000 movies in the [Stanford large movie review dataset](#) from early in the 2010s so some recent movies and some popular movies might be missing from the dataset. Some known good search terms are movies like "Mission Impossible", "Top Gun" and many more.

#### Search Results

MOVIE POSTER	MOVIE TITLE	YEAR	SENTIMENT
	Batman Returns	1992	Positive
	Batman: Vengeance	2001	Positive
	The Batman	2004	Negative

2. **Search Engine:** Choose a different ranker and/or search corpus and try a new search. A good example query with movie names only is “Meryl Streep” (doesn’t return any results) while it returns movies with Meryl Streep when the “All Movie Metadata” search corpus is chosen:


### Intelligent Movie Search

Choose a Ranker (Default: Okapi BM25)

Choose a Search corpus (Default: Movie Names only)

3. **Movie Details:** View the movie details for any movie returned in the search engine results. Direct link to an example movie for this use case and the ones below:  
<http://cs410djt.azurewebsites.net/detail/0062622/>
4. **Sentiment Analysis:** Look for the “Review Sentiment” row in the movie details table

# Movie Details




ATTRIBUTE	VALUE
Movie Poster	
Name (Click name for link to IMDB)	<a href="#">2001: A Space Odyssey</a>
Year Released	1968
Length (minutes)	149
Rating	8.3
Review Sentiment	<b>Positive</b> 30 Reviews Analyzed sentiment: 27 Positive   1 Negative   2 Neutral
Genre(s)	Adventure, Sci-Fi
Summary	After uncovering a mysterious artifact buried beneath the Lunar surface, a spacecraft is sent to Jupiter to find its origins - a spacecraft manned by two men and the supercomputer H.A.L. 9000.
Cast	Keir Dullea, Gary Lockwood, William Sylvester, Daniel Richter, Leonard Rossiter, Margaret Tyzack, Robert Beatty, Sean Sullivan, Douglas Rain, Frank Miller, Bill Weston, Ed Bishop, Glenn Beck, Alan Gifford, Ann Gillis, Edwina Carroll, Penny Brahms, Heather Downham, Mike Lovell, John Ashley, Jimmy Bell, David Charkham, Simon Davis, Jonathan Daw, Péter Delmár, Terry Duggan, David Fleetwood, Danny Grover, Brian Hawley, David Hines, Anthony Jackson, John Jordan, Scott MacKee, Laurence Marchant, Darryl Paes, Joe Refalo, Andy Wallace, Bob Wilyman, Richard Woods, Martin Amor, S. Newton Anderson, Ann Barrass, Jim Beasley, Sheraton Blount, Ann Bormann, John Clifford, Harold Coyne, Julie Croft, Harry Fielder, Penny Francis, Jane Hayward, Lew Hooper, Judy Keim, Kenneth Kendall, Maya Koumani, Vivian Kubrick, Roy Lansford, Maggie London, Marcella Markham, Irena Marr, Krystyna Marr, Chela Matthison, Colin McKenzie, Kim Neil, Jane Pearl, Penny Pearl, Ivor Powell, Doug Robinson, Jennifer Sanders, Kevin Scott, John Swindells, Burnell Tucker,

## 5. Similar Movies: Scroll down in the movie details section and look for a list of similar movies

Select a method for Similar Movies (Default: BERT)  
Select a method ▾

Update Similar Movies

### Similar Movies

MOVIE POSTER	MOVIE TITLE	YEAR	SENTIMENT
	<a href="#">Riders to the Stars</a>	1954	Neutral
	<a href="#">First Spaceship on Venus</a>	1960	Positive
	<a href="#">Farscape</a>	1999	Positive

6. **Similar Movies:** Choose a different method to generate similar movies. Clicking “Update Similar Movies” will usually return a different set of movies.

Select a method for Similar Movies (Default: BERT)

Select a method ▼

Select a method

BERT

Word2Vec

TF-IDF

ar Movies

## Tasks and Team member contributions

### Team member contributions

Across the team, we spent ~151 total hours of effort on this project. The high level summary of each team member’s contribution is below.

Team Member	Contribution
Danh	Built the search engine module
Justin	Built the sentiment analysis module
Tik On	Built the data collection and aggregation code. Built the similar movies module
Vamshi	Built the web application and the hosting process. General coordination and doc editing/finalization.

### Final Task list and effort

We built an initial task list in our proposal. Below is the same task list with new **columns describing the task status and notes:** *Completion Status, Actual time spent and Comments/Notes*

Task Type	Task	Description	Completion status (as of 11/12)	Actual time spent	Time estimate in hours as of 10/23 (initial, low confidence)	Comments/Notes
Research	Data Review	Understanding the data structure of the primary movie dataset	Completed	4	4	We each understand the research data set which contains 100,000 user reviews
Research	Package Review	Read and understand the documentation to understand how to use Cinemagoer API	Completed	4	4	We reviewed the API and confirmed that it can be used for our application to get metadata from IMDB
Design	Webpage Design	Designing a simple web application	In progress	4	4	We designed a simple web application with two pages and agreed on the JSON data schema formats.



Implementation	Fetch and store metadata	Fetch all structured metadata from IMDB matching the movie dataset	Completed	10	2	We have used the API to download structured metadata for all data. More time spent for data validation than actual creation of the code for download
Implementation	Structured data storage	Build a file-based system (e.g. simple JSON files) to store the retrieved metadata	Complete	8	8	All metadata related to the movies from IMDB and the research dataset is stored as JSON files. We may update this format over time as we implement the features.
Implementation	Web Application implementation	Implement the front end for different scenarios – Browse, search, etc.	Complete	18	16	We implemented the web application. Time taken was high since we weren't familiar with the technology/tools to build modern web apps.
Implementation	Movie Search Engine – <b>Uses CS410 learning</b>	Implement a movie search engine by using user's queries to get results from the database	Complete	10	28	Uses MetaPy with movie names or full movie metadata as corpus.
Implementation	Sentiment Analysis – <b>Uses CS410 learning</b>	Implementing a sentiment analysis classifier to detect positive or negative reviews	Completed	10	16	Worked with a Jupyter notebook to try different sentiment analysis methods and use the dataset labels for training data to evaluate f1 score. Settled on the VADER model for sentiment analysis
Implementation	Similar Movie Feature – <b>Uses CS410 learning</b>	Implementing a Similar Movie Finder to get similar movies in movie detail page	Completed	16	16	Comprehensive implementation of 3 methods with others tried and discarded.
E2E scenario testing	Integration and scenario testing	Writing wrap-up scripts to run and bundle all the required scripts and test the feature set	Completed	12	16	Complete. The earlier agreement on JSON formats and pre-computed data made integration testing easy. The hosting process and testing took some effort.
Documentation	Application Documentation	Writing a user-friendly readme.md to use and test our application per grading guidelines	Completed	1	2	Complete
Documentation	Presentation	Preparing a storyline and recording(s) a video to demonstrate our works	Completed	7	4	Includes time to create the presentation.

Documentation	Final Report	Final report with details on design, implementation, etc.	Completed	8	8	Around 1-2 hours per person plus a bit more for editing.
Team Meetings	Coordination, group work on some of the items above	Meetings over the course	Completed	39	8	~ 12 hrs of meetings * x attendees (usually all four of us)
<b>Total</b>				<b>151</b>	<b>116</b>	