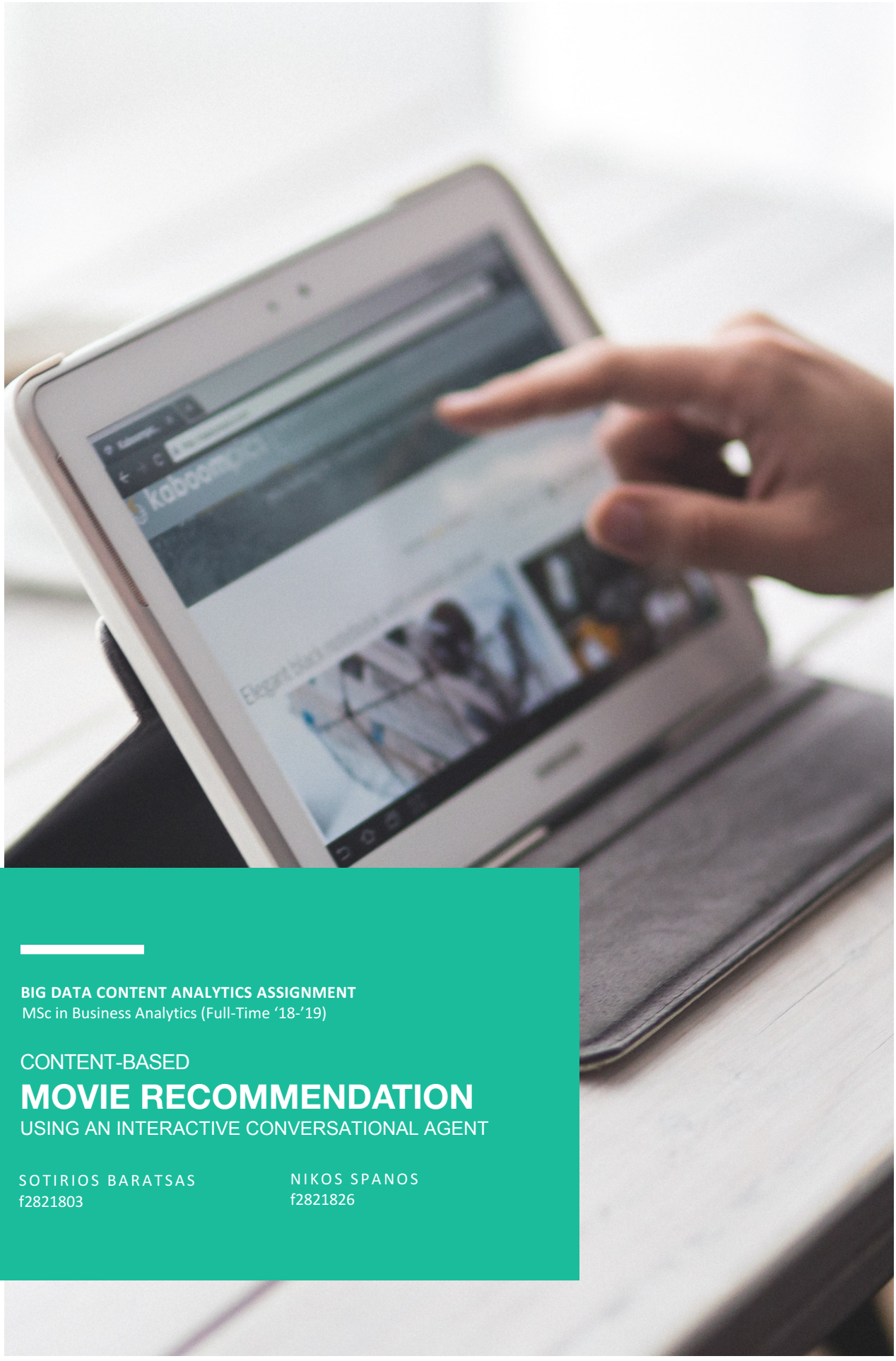


BIG DATA CONTENT ANALYTICS ASSIGNMENT
MSc in Business Analytics (Full-Time '18-'19)

CONTENT-BASED

MOVIE RECOMMENDATION

USING AN INTERACTIVE CONVERSATIONAL AGENT

SOTIRIOS BARATSAS
f2821803

NIKOS SPANOS
f2821826

Outline

Executive summary

The goal off this project is to create an interactive movie recommendation system, based on neural network embeddings and deliver it to the end user via a Messenger chatbot.

The components of the system are:

- A *Python script*, which includes access to a dataset of movies, the recommendation algorithm and the trained embeddings.
- The *Messenger user interface*, which is the front-end where the user interacts with the system via text messages.
- A *Dialogflow* agent, which acts as a mediator between the Python script and the user interface.

Our team implemented 2 different approaches for the movie recommendation algorithm. Our first approach involved parsing movie articles from Wikipedia, extracting the links to which each article points, and training our own embeddings based on {movie, link} pairs. This way, movies with similar links were mapped closer together and similar links were mapped closer to each other, too. The nature of the links themselves was vital for the representation ability of the embeddings, since the links contained information regarding actors, directors, production companies, year of release, awards, genres and more. Overall, even by using a smaller sample of the dataset, we managed to product thoughtful and accurate recommendations. For our second approach, we started with an IMDb dataset of 5000 movies, which we enriched with additional features. Using a combination of features, such as the cast, genres, plot keywords, director, and others, we created word embeddings with the FastText method. We also created a scoring and matching algorithm, which used user inputs to compare movies based on their Cosine Distance, penalize or reward their score based on other movie features (e.g. ratings) and conclude to a final recommendation.

The Challenge

Introduction

Our challenge is to build an effective movie recommendation system, that interacts with the end user through a Messenger chatbot. We decided to split this core challenge into 3 distinct challenges, in order to manage and prioritize our work more effectively.

Challenge 1: Capture what makes a good movie recommendation

The challenge that will make or break our recommendation system, is whether our algorithm will be able to use the features of a film to compare it to other movies and calculate their similarity. To solve this challenge, first, we need to find or create a dataset with explanatory features and, secondly, implement an effective training algorithm to map the embeddings in a way that accurately captures the similarity between movies.

Challenge 2: Quantity and variety of dataset

To make good recommendations, our algorithm needs to have access to a large and varied dataset of movies, with which it will perform comparisons and draw recommendations. For the purposes of our project, we have used datasets of medium size (e.g. around 4-6k movies) in order to keep a balance between good recommendations and flexibility/performance. However, our code can easily scale to work with a larger dataset, without any significant modifications.

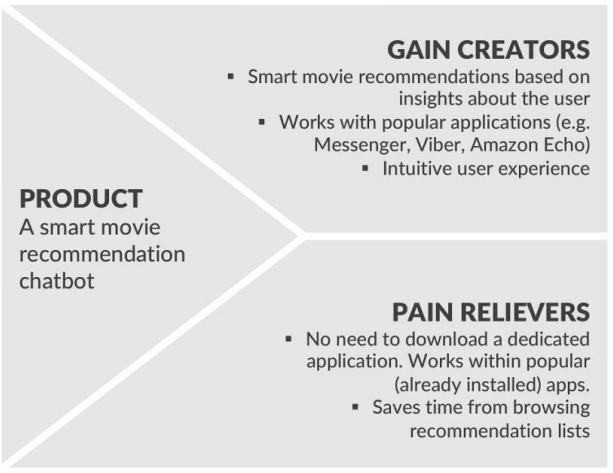
Challenge 3: Optimisation of user interaction

Finally, the third challenge was to optimize the way the end user interacts with the system and the system's responses. There are a lot of configuration options we could add on Dialogflow and a lot of cutting-edge NLP algorithms we could add to make our chatbot more intuitive, However, we realized that this part fell outside the main objectives of the project and we chose to implement a Minimum Viable Product (MVP) approach. It goes without saying, that solving this challenge is a high priority in the project's next steps.

User Profile & Value Proposition

Our next step, was to gain a better understanding of the ways someone might use our system to get a movie recommendation. As part of this step, we took a deep dive in the user's journey and the potential benefits of our solution. Moreover, we took a look at existing solutions to identify in which ways they fulfilled or came short of fulfilling the user's needs.

USER PROFILE



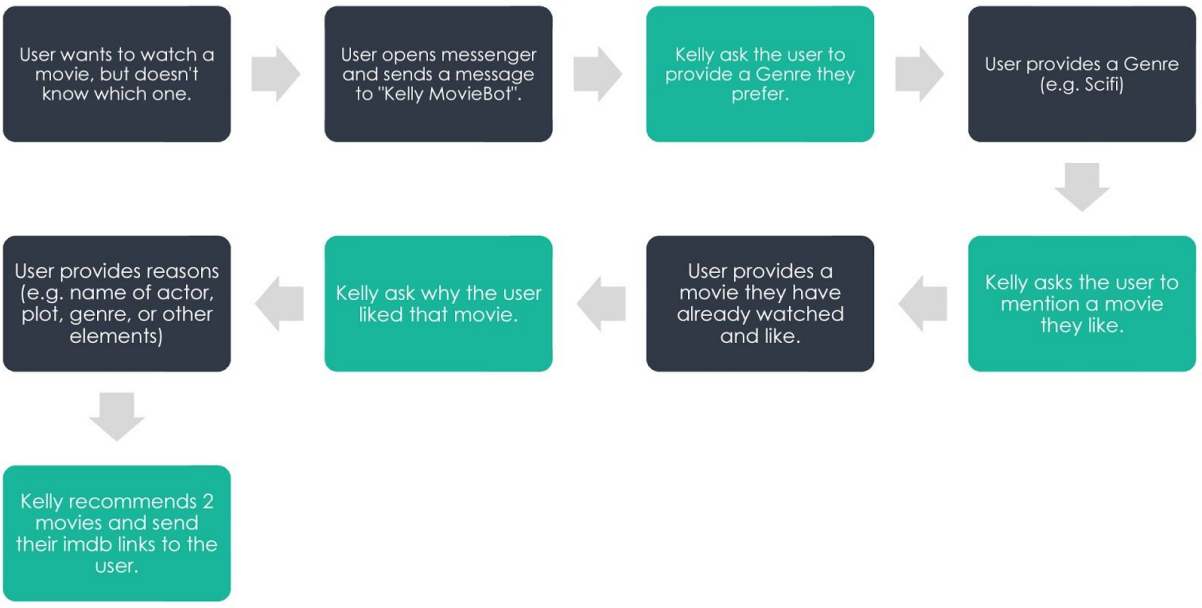
VALUE PROPOSITION

The outcome of this process was a user profile and a value proposition, that captures which characteristics we want our system to have, such as:

- Making recommendations based on a user input (insight)
- Being integrated in other popular apps (e.g. Messenger) instead of requiring to download a new application.

Optimal user journey

After deciding that our application will interact with the end user through a conversational agent (chatbot) in Messenger, which we named “Kelly the MovieBot”, we designed the optimal user journey and took key decision regarding the inputs we will request from a user.



More specifically, we decided that a user would provide:

- A **genre** they are in the mood for
- A **movie** they have previously watched and liked
- A few **reasons** why they likes that movie

Our decision to include these inputs is based on the insight, that even though a user might have a hard time selecting a movie to watch, they can usually very easily tell you what kind of genre they are in the mood for, and a movie they already like. Our recommendation algorithm will be able to use these inputs to provide a recommendation that the user has a high probability of enjoying.

Our Solution

Components

To make our system work, we need 3 key components: a python script, a user interface and a dialogue manager.

Python script

Our recommendation algorithm, written in Python, is the heart of our system. First of all, it has access to the dataset of movies we use to compare movies and make recommendations. It also includes the steps needed for the preparation, loading and cleaning of the dataset. Secondly, it includes the steps needed to train our embeddings using neural networks. Last but not least, it is where the comparison between movies happened and were our scoring algorithm is implemented, in order to select the best suggestions for our user.

User interface

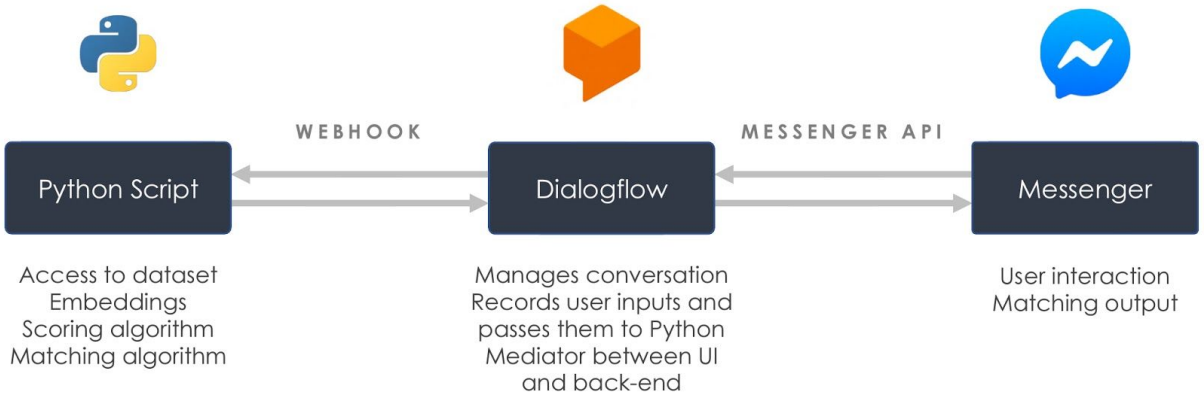
In contrast with the Python script, which is completely invisible to the end user, we decided to use Facebook's Messenger platform as the place to interact with the end users. The Messenger app started rising in 2014, when Facebook decided to split it from the main Facebook app and give incentives to users to download it and use it as a separate app. Since then, it as exploded in popularity, with more than 1.2 billion monthly active users. It has also transformed into a hub of useful or entertaining applications, taking the form of an extension (e.g. an appointment scheduling app) or an automated messaging service (i.e. a chatbot). To do that, it gave access to independent developers to develop their own applications, which is what will take advantage of to connect our application with Messenger's developer API. The format in which the user will interact with our system is Text Messages.

Dialogue manager

Finally, the connecting component is the dialogue manager, which will act as a mediator between the Python script and Messenger. We chose to use Google's Dialogflow service, which offers customizable and easy-to-use NLP services for text-based conversational interfaces. Dialogflow offers many configuration options, however, since this part was not part of the scope of this project, we chose to configure the core functionality. Dialogflow is connected with our Messenger interface through Facebook's API and receives the messages of the user. After identifying entities and intents, it send an input to our Python script (in JSON), and after the script performs its functions, Dialogflow returns the output to the user in Messenger. To make the Dialogflow work properly and fetch our custom messages to the user, we used the so called HTTP callback named webhook that is

automatically invoked whenever certain criteria is fulfilled. To connect webhook with our custom functions of movie recommendation, we run webhook using ngrok. Ngrok is a web tunnelling tool that provides a way to test APIs and webhooks from local server.

For the Dialogflow to work two components are needed:
Component 1: app_v6.py, which contains the steps of calling the webhook responses.
Component 2: movie_recommendation_v5.py, which contains the 6th version of the recommendation algorithm.



Approach A: Training embeddings using Wikipedia links

Our first approach involved parsing movie articles from Wikipedia, extracting the links to which each article points, and training our own embeddings based on {movie, link} pairs.

Part 1: Parsing Wikipedia articles

Wikipedia is one of the largest sources of knowledge and data in the world. It also manages to keep its vast amount of information quite well-structured and easily accessible to anyone. This characteristic made it ideal for the purposes of our project, since it meant that we could export a well-structured dataset to match our analytical needs.

We started by downloading one of the recent Wikipedia dumps, which included all the different articles of Wikipedia in a compressed (bz2) XML format. Working with such a huge dataset would be difficult and cumbersome, so we used parsers to isolate the information that we needed, in a format with which we could work. At the end, we sourced a dataset of 3130 movies, with data such as their titles, directors, budgets and most importantly, all other internal (i.e. Wikipedia articles) and external pages each article links to.

Part 2: Creating movie-link pairs

Our approach is based on a simple hypothesis: “Movies which are similar to each other will share links to common or similar pages”. For example, 2 similar movies might share links to Wikipedia articles about actors who appear in both films, common directors, screenwriters or produces, awards that both movies won, the categories these movies belong to and much more. It is the nature and variety of these links that manages to encompass a large amount of information that make a movie similar to another, but is often difficult to standardize and represent in a fixed format. To move on to our analysis, we created movie-link pairs, which we will use to train our embeddings.

Part 3: Creating embeddings

The next step involved creating and training our embeddings, using neural networks. The idea is simple: we will feed the model with pairs of movies and links {movie, link} and train the model to define embeddings of size=50. Our hope is that the model will place movies that link to the same pages closer together and that placement will reflect the similarity of the movies. Moreover, the model will be trained to understand if certain links are similar to each other (e.g. "Category: Action movies" will be similar to "Category: Adventure movies" since a lot of movies will have both links).

Part 4: Calculating similarity

Finally, after training the embeddings for movies and links, we can calculate the similarity of a given movies, with all other movies on our dataset, by using the normalized weights of the embedding layer as an approximation of the cosine similarity.

Overall, we could say that, even by using a smaller sample of the dataset, we managed to product thoughtful and accurate recommendations using this approach.

Approach B: FastText embeddings using movie characteristics

Our solution consists of four different parts. In the first part of our solution we explain the steps of how we got the dataset and what were the necessary updates and transformations we did in order to clean the dataset from the noise it already had. In the second part of our solution we create extra columns that will help us to construct a more accurate recommendation model. In the third part, we create our own word embeddings based on text phrases from three different columns of our dataset. Finally, in the last part we create a naive approach of the movie recommendation algorithm using cosine distances, trained embeddings and a matching score. All the above parts will be extensively described in the pages to come.

It is important to mention, that our initial approach of using cosine similarity was inspired by the article *"Building a Movie Recommendation Engine in Python using Scikit-Learn"*.

Part 1: Read, Update & Clean the data

The dataset can be found in the following link: https://bit.ly/2J9XjLt

CLEANING

The dataset contained many irrelevant columns compared to our needs. So, our first action was to delete any unnecessary column. For example, the facebook likes of each actor, or the color, were totally off topic. We continued by removing unnecessary special characters and symbols, which hurdled the string manipulation in Python. Next we observed some duplicate records in the column of the imdb links, so we removed those rows explicitly from the dataset. Last but not least, we observed that for some rows the column of duration had values less that 70 minutes. So, our first thought was to search for some of those rows and we found that they were referring to Series episodes. So, we filtered the dataset by keeping the rows with a duration greater than 70 minutes. To avoid recommend to the end user a title that it was derived from a tv series episode.

UPDATING

The second important part of the cleaning the dataset was also to update its columns. The IMDB rate of some movies was outdated, which totally makes sense, since the IMDB rating can change from time to time. And because we later use the IMDB rating to award or penalise each different movie, we didn't want to bias the result from an outdated IMDB ratings. Thus, using BeautifulSoup, a third party Python library for web scraping, we updated the IMDB rating and we also added the plot summary of each movie (a feature not existed in the initial dataset).

Part 2: Building the movie recommendation model

In the second and the next part we transform the dataset, in order to make it eligible for the recommendation algorithm built in the last part. Having said this, it is obvious that we made many back and forth moves to those two parts because the recommendation algorithm was changing very fast. Initially, we started by creating the variable "Combined features". This variable is a single text that contains the value of each column in the dataset separated by single spaces. With this column, we have all the valuable information connected to a movie in a single text. Moving on, we replaced missing values with either a space or 0 and we changed the type of each feature to a string, except the IMDB Rating of the movie which has been transformed to a float. So at the end of the second part, we ended up with one more column, the "combined features".

Part 3: Word Embeddings

Moving on, we create three more columns. Those related to the word embeddings of the movie cast, movie plot and the movie's features assembled in part 2.
To produce the word embeddings, we deployed an unsupervised model from the FastText library. The model is unsupervised, since we only have X independent features and no dependent ones. Out of the two options FastText algorithm provides, we chose to train a skipgram model instead of a CBOW one. We tested both approaches although we chose to proceed with the one yielded the best results and that is the skipgram approach.

Cast Embeddings

Step 1: Take the three actor names per movie and write them in a txt file
Step 2: Insert the text file to following formula:

File = text file. **Model** = Skipgram. **Learning rate** = 0.05. The higher the learning rate, the sooner the model will converge to a solution although the chance of overfitting is inevitable. Thus, a best practice according to the documentation is a learning rate in the range 0.01 to 1. **Dim** = 100. The dimension (dim) controls the size of the vectors, the larger they are the more information they can capture but requires more data to be learned. By default, we use 100 dimensions, but any value in the 100-300 range is as popular. **Ws (windows size)** = 3: Window size is determined by the number of words each text has. Since we have three actors, the window size will also be 3. **Epoch** = 500. The epoch parameter controls how many times the model will loop over the data. The result of this model is a list of vector of size 1*100. Where 1 indicates each different triplet of actors and 100 the number of values per triplet.

Plot Embeddings

Following the same steps with the cast embeddings, we created a txt file to make the unsupervised model get successfully trained.

File = text file. **Model** = Skipgram **Learning rate** = 0.05. The higher the learning rate, the sooner the model will converge to a solution although the chance of overfitting is inevitable. Thus, a best practice according to the documentation is a learning rate in the range 0.01 to 1. **Dim** = 300. We have increased the embeddings, because the text per movie is larger. **Ws (windows size)** = 6: Window size has been also increased, because more information is now gained for each word. **Epoch** = 500. The epoch parameter controls how many times the model will loop over the data. The result of this model is a list of vector of size 1*300. Where 1 indicates each different triplet of actors and 300 the number of values per triplet.

Feature Embeddings

(For training the feature embeddings we used the exact same parameters with the plot embeddings).

File = text file related to the combined features column. **Model** = Skipgram. **Learning rate** = 0.05. The higher the learning rate, the sooner the model will converge to a solution although the chance of overfitting is inevitable. Thus, a best practice according to the documentation is a learning rate in the range 0.01 to 1. **Dim** = 300. We have increased the embeddings, because the text per movie is larger. **Ws (windows size)** = 6: Window size has been also increased, because more information is now gained for each word. **Epoch** = 500. The epoch parameter controls how many times the model will loop over the data. The result of this model is a list of vector of size 1*300. Where 1 indicates each different triplet of actors and 300 the number of values per triplet.

At the end of part 3 we added to the dataset three columns containing for each row one array of size 1*dim.

Part 4: Assembling the movie recommendation algorithm

Having passed through all the necessary steps of data cleaning, data transformation and word embeddings, it is now time to create the algorithm that will actually use the data as input to find what the user will ask for; a movie recommendation. The creation of the final three recommendation movies, consists of three different phases. The *first phase* is collecting the user's inputs and match the data for each movie with those inputs.

The quality of the output depends on the quality of the input. In our case we have two kinds of inputs. The dataset with movie features and custom embeddings, but also the user's input. The dataset is completely transformed to our needs, although the user's input could contain noise that would spoil the result of the algorithm. To avoid that we created three functions that will process the input given by the user and will transform it to a form that will match the values of our dataset.

The first function relates to the correction of the movie genre. For example, after giving the application to users for testing, we observed that many users wrote "romantic" instead of "romance". This was a problem in the first place, because the algorithm couldn't work without giving the exact match of a genre. The second function we created, correctly match the movie title given by the user to the one that is closer to a movie title of the dataset. For example given as input "Spiderman 3" is totally the same by giving as input "Spider-man 3". So, to avoid mismatches caused by an extra "the" or by a comma, the function we created properly fixed that. The third function assembled, properly removes unnecessary words, the ones called stop words, while keeps the subject from the ones that determine the final recommendation. The process of keeping the subject of a word is calling stemming. For example if the user, in the reasons why (s)he likes a movie, writes "I am loving spies", our function will throw away the words "I", "am" and will keep the subject of the words "love" and "spy".

Example of phase 1
User's genre: Romantic -> Romance (Romance is one of the genre in the dataset)
User's Movie Title: The Titanic -> Titanic (Titanic is the correct spelling of the movie)

User's Input: "I like titanic because it is a romantic story" -> ['like', 'titanic', 'romance', 'story']

Moving on to *phase 2*, it is time to match the user's inputs with values from the dataset and calculate the cosine distance of the corresponding word embeddings. Initially, based on the movie genre given us input we slice the rows of the dataset that do not belong to the given genre. So, out of the 5000 movies we keep for example only the 1200 "romantic" ones. Out of those 1200 action movies we find the one that match the movie title given by the user. If for example, the user gave us input the movie "Spectre", which is for sure an action movie we match the whole row of the dataset that belongs to that movie. From the row matched, we extract the movie's plot summary and the movie's embeddings. We add the movie plot summary to the third input given by the user, which first we cleaned from stop words and stemmed. So, if we go back to example of phase 1, the user's input "I like titanic because it is a romantic story" will be transformed to ['like', 'titanic', 'romance', 'story'] + plot summary. This list that contains both the user's input and the unique words of the plot summary is the one out of the two important outputs of phase 2. The second also important output is the most 5 similar movies to the one given by the user. To calculate those 5 most similar movies, we calculated, as already stated above, the cosine distance between the feature embeddings of the movie and the embeddings of 1200 romantic movies.

Embeddings of the movie: 1x300

Embeddings of the 1200 romantic movies: 1200x300 } Result: Array of 1x1200 movies

Sorting the movies and locating the 6 movies on the top of the list except the first one, we end up with 5 most similar movies to the movie given as input. Concluding the end of the second phase, we end up with a list of unique words that combine the user's input and the plot of the movie, and also the a dataframe with the indexes of the 5 most similar movies to the one given as input.

The *last phase* of Part 4, is were the scoring variable is created to award the movies with the highest score and penalize the others. The movie score consists of three elements:

First element: Primary genre
Award those movies were their first or second genre values are matched to the genre given by the user.

Second element: IMDB Rating
The movie promoted first should have a higher imdb rating than the other two movies. Although, since we don't want the result to get biased from a high IMDB rate, we don't give a high weight to that.

Third element: Number of words.

This is the most highly weighted scoring element of the two aforementioned elements. To calculate the number of words, we use the list, which is the first output of phase 2. From this list we count count how many words are found in the column “combined features”, created in part 2. For each word found we increase the scoring element by one. Thus, the higher the score of this element, the more relevant the movie to the one given as input by the user.

The final function to calculate the movie score of the 5 most similar movies according to the cosine distance is the following:
(Primary Genre x 0.2) + (IMDB Rating x 0.3) + (Number of words x 0.5).
Sorting the movies based on the relative score we end up with the three movies that will be proposed to the user.

Finally, it is important to mention that our recommendation algorithm has been developed with 6 different versions, each one to be the update of the previous one. Our last and final version has the following updates (also stated in the Jupyter Notebook):

1) Adding the IF/ELSE option so the user do not get a message error if the movie (s)he give as input does not exist in the dataset.
2) Filtering the word embeddings based on the movie genre given by the user.
3) Award the movies, which their genre belongs to columns: "genre_0" and "genre_1". So for example a movie that has the following genres: Action, Comedy, Drama and a following one that is: Mystery, Action, Romantic will be awarded higher that those movies that their genres is not in the first or the second place.
4) Ensemble the function "find_correct_movie", to match the movie given as input to the closest of the ones already existed in the dataset.

The sixth version of the algorithm is the one that will be used by the chatbot messenger to produce the list of the three recommendation movies proposed to the end user.

Next Steps

Improving our solution

Up to this point we have thoroughly explained out two approaches on movie recommendation. However, every approach has improvements and every approach can be optimized. Both the chatbot and the algorithm can be potentially improved. Some of the improvements we propose are the followings:

- Better configuration of the dialogue engine. Our messenger chatbot can be enriched with a wider pool of responses on the trigger of different events.
- Larger Dataset. To enrich the dataset with more imdb links will help us gain more information about the differences of the movies and thus to better train our custom word embeddings.
- Optimize scoring & matching algorithm. Up to this point we have tested cosine distance. Although, a wider pool of metrics exist to compare text documents with

each other. Some other approaches may include the “Word Mover Distance” or the “Siamese Manhattan LSTM”.

Potential business applications

Part of building a unique solution such the one presented, is also the commercial use of it. One of our first thoughts is to deploy the messenger chatbot as a movie recommendation solution to Village Cinemas or Greek local cinema theaters. Based on the movie that currently each theater hosts and the customer’s needs, we can successfully match those two opinions to reach a common decision, that will not only bring revenues to the cinema theater but also satisfaction to the end user. Furthermore, we scale things up and present an audience-facing entertainment app, sponsored by streaming companies to offer entertaining nights with friends.

Conclusions

Kelly the MovieBot is our movie recommendation chatbot that has been built after the findings of two different approaches presented in the section “Our Solution”. Having tested two different approaches, we integrated the second approach with the dialogflow engine. The second approach is more naive than the first one. While both approaches use trained models of word embeddings for every text document. Concluding this report, the reader will now have a clear view on why we chose to implement a movie recommendation engine, what is our value proposition and the niche market we try to hit, the ways on how to assemble your own recommendation algorithm and how this algorithm can successfully pass from the stage of development to the stage of a commercial and user-friendly application.

Bibliography - Online References

Getting the dataset:

1) https://www.kaggle.com/orgesleka/imdbmovies#imdb.csv
2) *Sotiris link*

Inspiration of the two approaches to build the algorithm:

1) https://medium.com/code-heroku/building-a-movie-recommendation-engine-in-python-using-scikit-learn-c7489d7cb145
2) *Sotiris link*

Building the chatbot through Dialogflow and Webhook:

1) https://www.pragnakalp.com/dialogflow-fulfillment-webhook-tutorial/
2) https://www.pragnakalp.com/dialogflow-tutorial-create-facebook-messenger-bot-using-dialogflow-integration/

Creating custom word embeddings:

1) https://fasttext.cc/
2) https://fasttext.cc/docs/en/unsupervised-tutorial.html
3) https://pypi.org/project/fasttext-win/
4) https://towardsdatascience.com/fasttext-under-the-hood-11efc57b2b3

Calculating Cosine Distance:

1) https://scikit-learn.org/stable/modules/generated/sklearn.metrics.pairwise.cosine_distances.html

TimePlan

July - August - September							
Weeks	**July 3rd week**	**July 4th week**	**August 1st week**	**August 2nd week**	**August 3rd week**	**August 4th week**	**September 1st week**
Planning and finalizing the scope of the project.							
Searching for data and ways to create a recommendation algorithm.							
Implementing two different approaches(“Our Solution”)							
Selecting one of the two approaches to integrate with Dialogflow							
Deploying the messenger chatbot and the 6 different versions of algorithms							
Testing the chatbot with actual users							
Writing the report and the presentation of the assignment							