Word2Vec in Recommendation Systems

Definition

Project Overview

This project aims to demonstrate how Word2vec can be used to build recommendation systems.

A recommendation system seeks to predict which items a given user would more likely interact with (click, buy, listen, watch and etc) based on previous interactions made by him or by similar users. This is a big area of interest for research and industry due to its comproved efficacy to increase user engagement and per user revenue. As an example, Netflix estimates that more than 80% of shows people watch on their platform are discovered through their recommendation systems (source).

Word2Vec is a set of techniques to generate word embeddings for Natural Language Processing tasks. Its main advantages over other NLP techniques it that it learns the context of words in a sentence based on the surroundings of other words, an illustration of this ability is that the algorithm could understand when the word "Buffalo" is used to mean the animal "buffalo" or the city "Buffalo" in the state of New York in a sentence. It does this by representing the words in a multi-dimensional space (embeddings) and these embeddings have the property that the distance between the embeddings is a signal of its similarity.

The dataset used for this project is the MovieLens, it is probably the most famous and used dataset for recommendation systems. Most libraries for recommendation systems and research articles use this dataset, so this simplifies the benchmarking and evaluation.

The final project I implemented differed from my original proposal on the following points:

- Dataset

  I initially planned to develop this project against the "Expedia Hotel Recommendations Kaggle competition", however this dataset showed difficult to work due to its size, complexity of features and lack of articles. After spending some time on EDA, I concluded that using a more common dataset for recommendations systems would be simpler to develop and to benchmark. For this reason, I decided to use the MovieLens dataset.

Also, the main benefit in my opinion to use a Kaggle competition for this project was the possibility to easily benchmark my results against the leaderboard. But since my proposal is to demonstrate how a technique of a different domain (NLP), can be used for recommendation systems, I do not expect that my final solution will perform better than purpose built algorithms.

- Algorithm

  My first idea was to build a custom model in PyTorch to generate the embeddings. However, I ended up using Gensim a common library for Word2Vec. After prototyping and failing to produce good results, I realized that creating my own model for this task would be too time consuming. I decided to use a specialized library so I could invest my time in the other parts of the project equally important and challenging.

Problem Statement

This project is defined as: Applying Word2Vec to learn similar movies using the MovieLens dataset.

This dataset is derived from the MovieLens platform, in this application users can rate and review movies they have watched, get rich information about movies and receive recommendations of movies to watch next.

In this project I used the smaller version of the dataset called MovieLens100K which contains 100.000 of ratings of 9.000 movies, made by 600 users (link for dataset).

The goal here is to demonstrate how a technique of NLP can be abused to generate recommendations, I explore ways to serve the recommendations based on embeddings in a scalable way and I also create a sample application to visualize the generated recommendations.

My intention is not to produce state of the art results, but simply demonstrate a good enough and simple technique to generate recommendations using a really interesting approach.

Metrics

The main metric I used to evaluate my model was "Average Precision at k" (source). This is a common metric used by recommendation systems.

Since the recommendation system I built suggests similar items (item-to-item) and the MovieLens dataset doesn't have a ground truth dataset for this purpose, I calculated the metric using the following approach:

1. Isolated 10% of my users to use for evaluation
2. For each user U in the evaluation set, I selected all his positive movie ratings MR
3. For each MR, I calculated k suggestions for the movie rated
4. I calculated the cardinality of the intersection between the calculated similar movies and movies the user rated, and divided by k. This value was the precision P for a given user
5. I summed all P's and divided by the number of users to have my final average precision at k.

The source code for the calculation of this metric can be found here.

Analysis

Data Exploration

The data exploration I performed for this project can be found on the notebook 10_eda.

The dataset used is the MovieLens 100k. It contains the following files:

- Movies.csv: Information about each movie like title and genres.
- Links.csv: A lookup table that maps for each movie the internal MovieLens id to the platforms: The Movie Database and IMDb
- Ratings.csv: This is the main dataset. It contains the rating from a user to a movie and when it made.
- Tags.csv: A user can also create custom tags for movies. I did not use this dataset in my model

In this project I focused on the ratings.csv file since it had the the user interactions, which is what I needed to train my model.

```
import pandas as pd

ratings_df = pd.read_csv("../data/ml-latest-small/ratings.csv")
ratings_df.head()
```

	userId	movieId	rating	timestamp

0	1	1	4.0	964982703
1	1	3	4.0	964981247
2	1	6	4.0	964982224
3	1	47	5.0	964983815
4	1	50	5.0	964982931

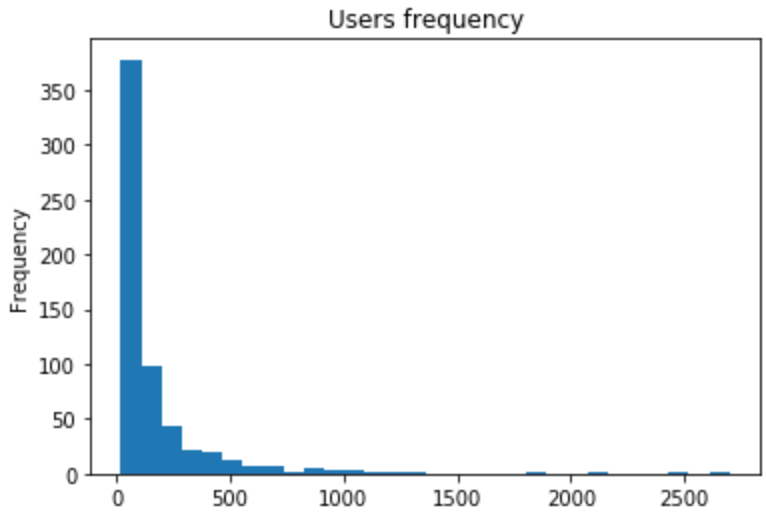
```
ratings_df.shape
```

(100836, 4)

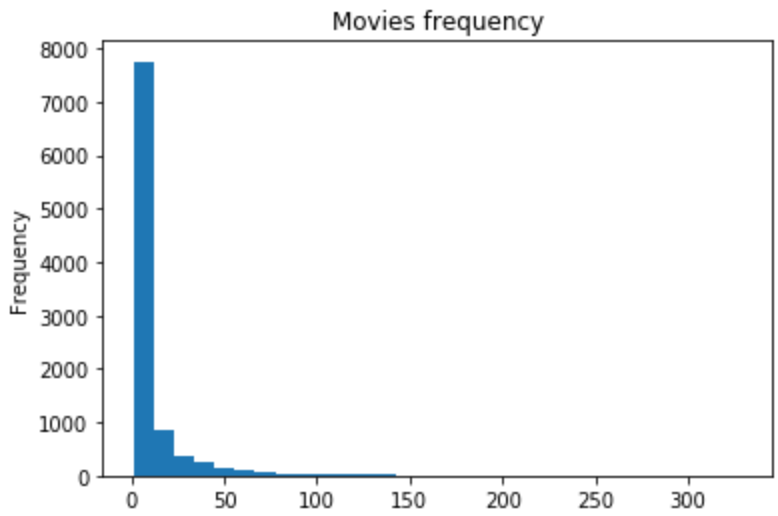
The ratings dataset is quite simple. It consists of the userId, movieId, rating and timestamp.

There are over 100.000 ratings and they are not evenly distributed across the dimensions: users, movies and ratings, as we can observe:

```
ratings_df["userId"].value_counts().plot(kind='hist', bins=30, title="Users frequency")
```




```
ratings_df["movieId"].value_counts().plot(kind='hist', bins=30, title="Movies frequency")
```



We can observe a few properties that are common in recommendation systems:

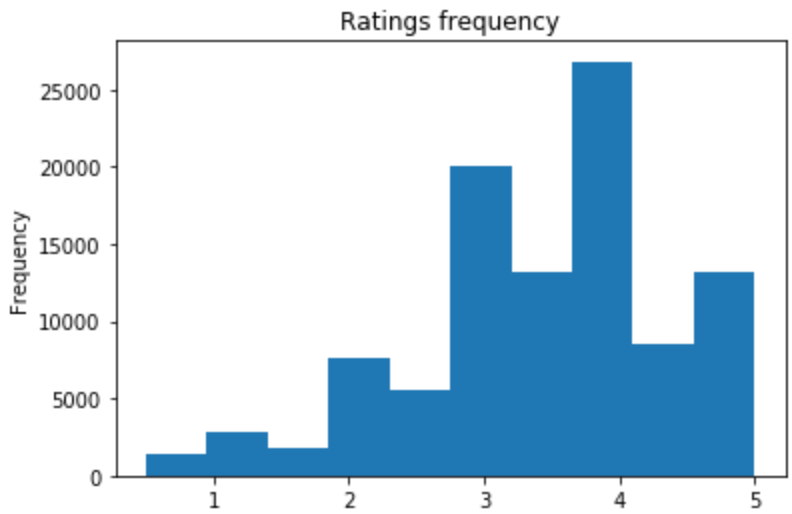
1. few users generates most of the interactions
2. few items concentrates most of the interactions

The second property is referenced as long-tail, and it is common for recommender systems to learn to recommend just the most popular items (this is known as the richer-get-richer problem) and perform well during evaluation and in production, but depending on the business, it may be interesting to diversify more the recommendations. The property of a recommender system to provide diverse recommendations is called serendipity and there is a huge collection of research articles about this (source).

To illustrate this property in this domain (movies), consider the movie "The Lord of the Rings - A fellowship of the Ring", users that saw this movie intuitively are more likely to also watch other movies of the "Lord of the Rings" franchise and "The Hobbit", so these recommendations are obvious.

I'm not planning to tackle the serendipity problem in this problem, however I'll analyse the coverage of my recommendations to have an idea if my recommender is subject to the richer-get-richer problem.

```
ratings_df["rating"].plot(kind='hist', title="Ratings frequency")
```




```
ratings_df_with_positive = ratings_df
ratings_df_with_positive["is_positive_review"] = ratings_df_with_positive["rating"] >= 3.0
ratings_df_with_positive["is_positive_review"].value_counts()
```

```
True     81763
False    19073
Name: is_positive_review, dtype: int64
```

I used the threshold value of 3 to consider a review as positive. Since I only use positive reviews in my model, this removed almost 20000 ratings from the dataset.

Algorithms and Techniques

This project is unusual since I do not try to find the best machine learning model and tune it against a metric. Here I try to apply to recommendation systems a technique from Natural Language Processing.

There are two techniques I'm using here:

1. Generate item embeddings using Word2Vec
2. Find similar items (generate recommendations) with embeddings

Generate item embeddings

There are multiple ways to generate embeddings, word2vec is just one approach to it.

Word2vec works by learning the context of words in a sentence. A usual sentence for word2vec is something in the form: 'A rabbit is in a hole', so a sentence here is a real language sentence.

The first approach I used was to form sentences for each review using the movieId and the genres of the movie. An example of this approach is the sentence "110 action drama war", this did not work well: instead of learning the similarities between movies (movieIds) it actually learned the similarities across categories. Which was not what I intended and not very useful.

After a few more interactions I generated sentences using the following algorithm:

1. sort all interactions by userId and timestamp. So all interactions of a user are together and sorted by timestamp
2. create a sentence consisting of all movies a user has rated

With this approach, a sample sentence has this format: "318391 2312 123923 763 6123".

The assumption I made was that users watch and review similar movies close to each other, so the word2vec model should learn how frequent these movies are watched together and create embeddings that represent these similarities.

Find similar items using embeddings

The output of the model training are embeddings. To generate recommendations for a movie, I have to get its representation as embedding, search for the K closest embeddings and for each of the embeddings found, discover the movieId for this movie.

The problem of finding the K closest embeddings is just the classic problem to find the K Nearest Neighbors.

For this project, I implemented a recommender (source) completely decoupled from the Gensim library (the library used to generate the embeddings). This approach allows the separation between training and prediction. This can simplify the deployment since you do not need to include in your runtime libraries used for training (and you don't even need to use the same programming language). Since the recommender is completely decoupled from Gensim, using just the embeddings (that are numpy arrays) it can also be used to serve recommendations created by other models that also generate embeddings.

KNN scales linearly with the cardinality of embeddings, so for problems with millions or billions of embeddings, a naive approach of KNN would not work - it would take too long to produce any recommendations. This is an interesting topic and industry are tackling it with creative ways: Facebook created a library for performatic approximate KNN search called FAISS (https://github.com/facebookresearch/faiss) and other companies, are storing the embeddings directly in search engines like Solr to leverage its distributed nature to speed up queries.

Benchmark

The MovieLens dataset is widely used for recommendation systems, so benchmarks are widely available. Microsoft maintains this repository with several benchmarks of recommendation systems applied in the MovieLens dataset.

My initial idea is that I could simply compare the results I got against this benchmark, however, it turned out to be more difficult than expected due since the way I generate the recommendations and calculate the metrics are not compatible to this benchmark.

Due to time constraints, I did not have time to benchmark the performance of my recommendation system. However, since my goal was to demonstrate the technique, I was not considering the benchmark as a must have in this project.

Also, offline metrics even though are very useful for model tuning and comparison. It does not always correlate to online metrics. This is due to how subjective relevance is to every person.

To allow a qualitative evaluation, I created one web application (source) to visualise recommendations for a movie. This was also useful to demonstrate how to deploy this model in production.

Methodology

Data Preprocessing

The dataset with ratings was already clean (no NAs or missing data).

I added these extra pre-processing steps in my solution:

1. Remove negative ratings: I removed all ratings less than 3.0. This step was not needed for my model (since it learns similarities between movies), however, it made sense to me to exclude bad ratings to don't polute with bad movies.
2. Generate sentences: I described the needed for this and the approach under Algorithm and Techniques.
3. Isolate 10% of users for evaluation. I'm isolating users and all its interactions.

The code for data processing can be found here.

Implementation

Model and Training

All code related to the model and training are in the model notebook. You can find examples of usage and generated recommendations.

I used the Gensim library - which is a very popular Word2Vec library for Python - to generate the movie embeddings. I already had prior experience with this library, so when I realised that creating my own model would be too much effort for this project, the decision to use Gensim was no-brainer.

The main challenge was to create sentences in a way that the Word2Vec model would learn something meaningful, it took me several attempts until I had something that worked well.

To simplify things, I created a class "Word2VecMovieModel" that encapsulates all the steps regarding model training, evaluation and serialization of model.

Initially I trained a model with default hyper-parameter values, I analysed if the recommendations made sense to have a feel if my approach was working, and it was. For this model, I got a P@5 of 0.17.

Knowing that my approach was working, I was interested in improving the performance of my model. I created the function hyper_parameter_tunning to try different combinations of hyper-parameters and to select the best one. The only parameters I permuted were: number of epochs, window size (the length of words in a sentence the word2vec considers in a context) and negative (for negative sampling).

By automating the tuning of the model, I didn't have to understand every single aspect of the Word2Vec model and the implementation details of Gensim. A simple brute-force approach allowed me to have a good enough model.

The best model I got with this technique had P@5 of 0.26, over 50% of improvement over the naive model.

With the best model in hand, I saved it to disk, so I could reuse it for the next steps and I also calculated the coverage (how many distinct items my recommender can recommend) and I got a coverage of 29%.

The coverage of 29% means that my recommender is not able to suggest every movie from my inventory. So it is not a good model if one goal is to have good serendipity.

Predicting (aka generating recommendations)

All code related to generating recommendations are in the recommender notebook.

To serve recommendations in a production environment, usually two approaches can be taken:

1. Generate all recommendations in batch store it and serve from storage
2. Or Generate the recommendations on-line, deploying the model

The first approach works really well when the model is complex to deploy and the cardinality of items is low. This is actually the case in this project since the cardinality is low: only 10.000 items.

However, I decided to implement an online recommender to show how to serve recommendations from embeddings. For this I created the class KnnRecommender that had as dependency just the embeddings and the word indexes (of list of all the movieIds ordered by index in the embedding). This means that this Recommender is decoupled from the model, so even if I changed how to generate embeddings I could still reuse it to produce recommendations. Another benefit is that my serving application will not need to have Gensim installed.

To quickly calculate the KNN for an embedding I used the NearestNeighbors from Scikit-learn. This implementation is quite efficient and can scale well even for high cardinalities, however, if we the cardinality was already above a few million embeddings, libraries like Faiss or Annoy would have been more appropriate. These libraries trades-off accuracy for speed since they are implementations of approximate KNN.

A few examples of recommendations produced:

1. Source movie: Star Wars: Episode VI - Return of the Jedi
    a. Star Wars: Episode V - The Empire Strikes Back
    b. Star Wars: Episode IV - A New Hope
    c. Raiders of the Lost Ark
    d. Matrix, The
2. Source movie: Toy Story
    a. Aladdin
    b. Toy Story 2
    c. Forrest Gump
    d. Big Hero 6
3. Source movie: Seven
    a. The Silence of the Lambs
    b. Braveheart
    c. Tesis

d. Red Rock West

Visualizing and Serving

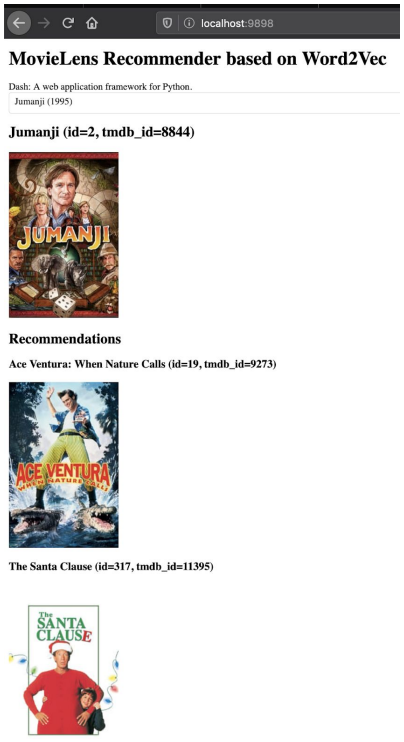
The code here refers to the explorer notebook.

To demonstrate how to use the recommender and to easily inspect the recommendations, I created one application using Dash.

To run it, one can simply run `docker-compose up` in the root of the repository and open the application in your browser link.

This application integrates with "The Movie Database" in order to fetch metadata for movies. The integration code is in the notebook tmdb, you need to get an API key to interact with the TMDB API.

This application uses the KNNRecommender and the embeddings, to suggest similar movies. It works as a good example of how a real application would do it.



Conclusion

The final model had precision@5 of 0.26 and coverage of 29%. The low coverage means that the model is subject to the richer-get-richer problem and is not a good fit if the goal is to have good serendipity.

The approach I presented definitely works: using Word2Vec for recommendations systems is able to produce relevant results, is simple to develop and is simple to deploy.

I demonstrated in this project all the steps commonly found in Machine Learning projects in the industry: from the conception of the idea to the deployment of the model in an application.

Even though I was not able to benchmark my solution against other models specific for recommendation systems, I'm satisfied with the results obtained since it was no trivial task and by qualitative analysis, results makes sense.

Interesting aspects

1. Applying Word2Vec an NLP technique for Recommendation systems is quite uncommon and was a fun project
2. Using embeddings for recommendations is a trend in the industry
3. Developed full life-cycle of a machine learning project
4. Using libraries like nbdev for development and Dash

Challenges and Difficulties

1. I lost too much time before I took the decision to switch dataset and not developing my own model
2. Not enough time to properly benchmark my model
3. Recommendation System are a tough problem. Relevance is too subjective.

Improvements

The following ideas are worth trying in the future:

1. Benchmark against classic recommendation systems. Recommendation systems that also generate embeddings (like Fast.AI Collaborative Filtering) should be interesting to compare.
2. Add more information to sentences. Maybe adding title, genres or actors in the sentences could help the model discover more relationships across movies and improve the serendipity.

3. Evaluate the model with real users. Create a Sagemaker Groundtruth job and let real users judge if recommendations are relevant for the given movie.