How to build a Recommender System for Airbnb in Python

Alexandra Gastone · Follow

9 min read · Apr 14, 2020

A simple guide to building a Collaborative Filtering system using Sentiment Intensity estimates for Airbnb data in Boston

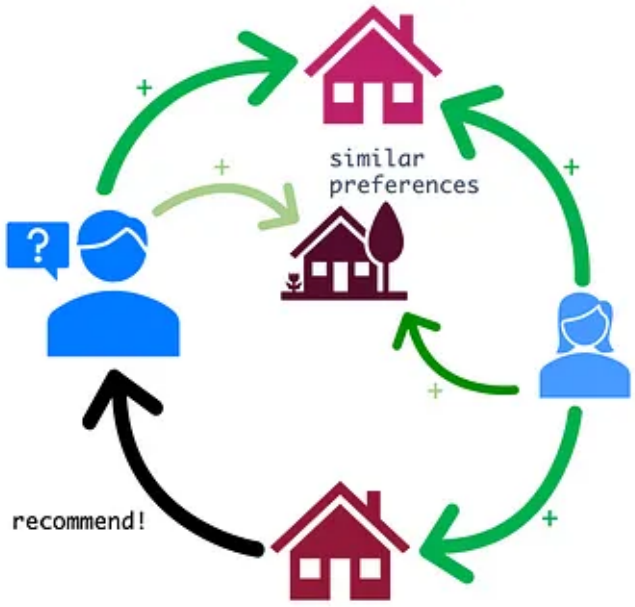
Photo by Brodie Vissers on Burst

Why should you care about recommender systems?

Recommender systems are one of the most important and successful applications of data science and machine learning technologies to business. They've become vital tools for companies that want to provide personalised user experiences, largely due to their powerful and scalable approaches to personalising content based on a user's predicted interests. Take Amazon or Netflix for instance: at its core, their success lies in their ability to drive engagement through **valuable** and **personalised recommendations** for each of their users.

While there are a lot of good resources for digging into the theory and novel algorithms, the field of recommender systems is extensive and continuously expanding, making it a bit of a daunting task to find **practical, real use cases** that put it all together. The aim of this post is therefore to **make the scope and implementation of such systems more straightforward**.

We'll be using the publicly available Airbnb dataset in Boston to build a recommender system that will give Airbnb users personalised listing recommendations based on what users with similar tastes have liked in the past. For example, if Joe and Billy both really liked their stays at Houses A and B, and Billy also liked House C, then we can recommend House C to Joe since Joe and Billy seem to have similar preferences in houses. This type of recommendation system is called **User-Based Collaborative Filtering**.



Collaborative Filtering for personalised user recommendations

Problem

> *So let's break the problem down: what are the main tasks we'll need to tackle in order to build our system?*

1. Determine which users have similar preferences
2. Make recommendations based on those preferences

Approach

> *How are we going to tackle these tasks?*

There are a few different ways that we could go about determining the similarity between user preferences (i.e. Pearson correlation between rating scores, cosine similarity between ratings vectors, etc.). Since our dataset includes all the comments that each user wrote for their individual stays, we're going to be extracting the **sentiment** of these comments to gauge how positively or negatively they rated them. Using a model-based approach that'll estimate a comment's latent factors (more on this when we get to the recommendation step), we'll then **predict** and **rank** a user's sentiment polarity score for listings they haven't yet visited.

We'll therefore split our approach into three parts:

- **Part 1:** Estimate the **sentiment** of comments
- **Part 2:** Build our recommendation engine to **predict** sentiment score for all reviewer-listing pairs
- **Park 3:** Make personalised recommendations for each user based on their **ranked** preferences

We'll build up our recommendation system by exploring and implementing each of these steps sequentially in the sections below, using readily available libraries in Python.

To illustrate what the output of each step should give us, we'll use a random user with the following comment on a particular stay :

```
'Great location for both airport and city - great amenities in the
house: Plus Islam was always very helpful even though he was away'
```

Part 1: Estimate the sentiment of comments

Context

Sentiment Analysis is a subfield of NLP (Natural Language Processing) that as of late has been gaining lots of traction at the industry level, with more and more companies looking to extract the opinions of their clients or users from textual data such as reviews, chats, or emails in order to gain **actionable insights**. Given that most of the data that's readily available usually comes in the form of text-heavy unstructured datasets, finding ways of extracting valuable information from these sources has become a key target for data-driven company growth.

Preparing our data

The dataset itself that we'll be working with includes all unique listing and user IDs (*listing_id*, *reviewer_id*), and each user's reviews on the listings they visited (*comments*).

A first look at our *comments* feature shows a couple of things we might want to take care of before doing any computations:

- some entries contain empty or **null** comments, remove those
- some reviews will have the prototypical “Host canceled this booking x days before…” comment when a stay is **canceled**, so we can also drop those entries
- not all reviews are in English, we have to detect the **language** of the comments and, depending on what percentage of them are non-English, either drop them or translate them (it turns out the non-English comments only constitute 0.06% of all comments at this point, so we can go ahead and drop them for now, but feel free to try your hand at translating — Google offers a free python library implementing Google Translate’s API called googletrans)
- other **text normalization** steps (i.e. remove numerical characters)

```
def preprocess_text(reviews):
  """
  :param reviews: dataframe of each users reviews, includes a 'comments' feature column

  returns preprocessed reviews dataset
  """
  # drop null
  reviews.dropna(subset =['comments'], how='any', axis=0, inplace=True)

  # drop cancelled entries
  index_canceled = reviews[ reviews['comments'].str.match('The host canceled this reservation')].i
  reviews.drop(index_canceled, inplace=True)

  # drop dash/numeric characters
  index_dash = reviews[reviews['comments'].str.match('-')].index
  reviews.drop(index_dash, inplace=True)

  alphanumeric = lambda x: re.sub('\w*\d\w*', ' ', x)
  reviews['comments'] = reviews['comments'].map(alphanumeric)

  # make sure to reset index before run langdetect, otherwise throws errors
  reviews.reset_index(inplace=True, drop=True)

  # create new language column
  reviews['language'] = reviews['comments'].apply(detect)
```

```
26
27      # isolate all non-en entries and drop
28      index_nonen = reviews[~reviews['language'].str.match('en')].index
29      reviews.drop(index_nonen, inplace=True)
30      reviews.reset_index(inplace=True, drop=True)
31
32      return reviews
```

preprocess_text.py hosted with ❤ by GitHub view raw

Preprocessing the 'comments' column for Sentiment Analysis

Note that we're purposefully not going to be removing punctuation or capitalization, which are typically part of the text normalization step of the NLP pipeline. The reason for this is that the Sentiment Analyzer we'll be working with actually incorporates this kind of information when estimating sentiment! **NLTK's Vader** (Valence Aware Dictionary and sEntiment Reasoner) lexicon and pre-trained sentiment analysis tool has had a lot of success in dealing with reviews and social sentiment. Part of the reason it works so well is that it adds valence to words that are capitalized, that have degree modifiers (eg. *very*), are accompanied by punctuation (eg. *!!!*) or emoticons (eg. *:)*), and it can even handle shifts in sentiment in a single sentence, also known as conjunctions.

Implementation

Vader actually returns a score for positive, negative and neutral sentiment, but we'll only be looking at the **compound** score of all three for our system.

```
1   from nltk.sentiment.vader import SentimentIntensityAnalyzer
2   import nltk
3   nltk.download('vader_lexicon')
4
5   def get_polarity(reviews):
6     """
7     :param reviews: reviews dataframe, includes 'comments' column
8
9     return the reviews dataframe with added 'polarity' column estimated using NLTK's Sentiment Analy
10    """
11    analyzer = SentimentIntensityAnalyzer()
12    polarity_compound = lambda s: (analyzer.polarity_scores(s))['compound']
13    reviews['polarity'] = reviews.comments.map(polarity_compound)
14
```

```
14
15      return reviews
```

get_polarity.py hosted with ❤ by GitHub view raw

Generating polarity scores

comments	language	polarity
My stay at islam's place was really cool! Good location, away from subway, then from downtown. The room was nice, all place was clean. Islam managed pretty well our arrival, even if it was last minute ;) i do recommand this place to any airbnb user :)	en	0.962600
Great location for both airport and city - great amenities in the house: Plus Islam was always very helpful even though he was away	en	0.906100
We didn't meet Izzy at all!!!! After we arrived nobody was there except some spanish speaking people. Our room was disgusting!!! It smelted like mold and was very dirty!!!! The Bathroom wasn't cleaned either and my friend and i wanted to leave right after we came... The Dryer or washer was not usable because the assistant from Izzy washed all the time. The Neighborhood is not safe at all and every other morning u could hear people fight. Don't stay there!!!!!! It's nothing like u would expect it! And it's defenetly to expensive for this nasty place!!!!	en	-0.779800
The house was in a very convenient location to the airport (we walked to the shuttle) and the bed/room was clean. Not a great neighborhood, but again it was booked to be close to our early morning flight.	en	-0.076500
Parking situation was bad. Very hard to find a spot. :-(So if you are driving I don't recommend. On the positive side of things, the place was very clean and close to the airport.	en	-0.298200

Example of sentiment polarity compound scores for comments in English

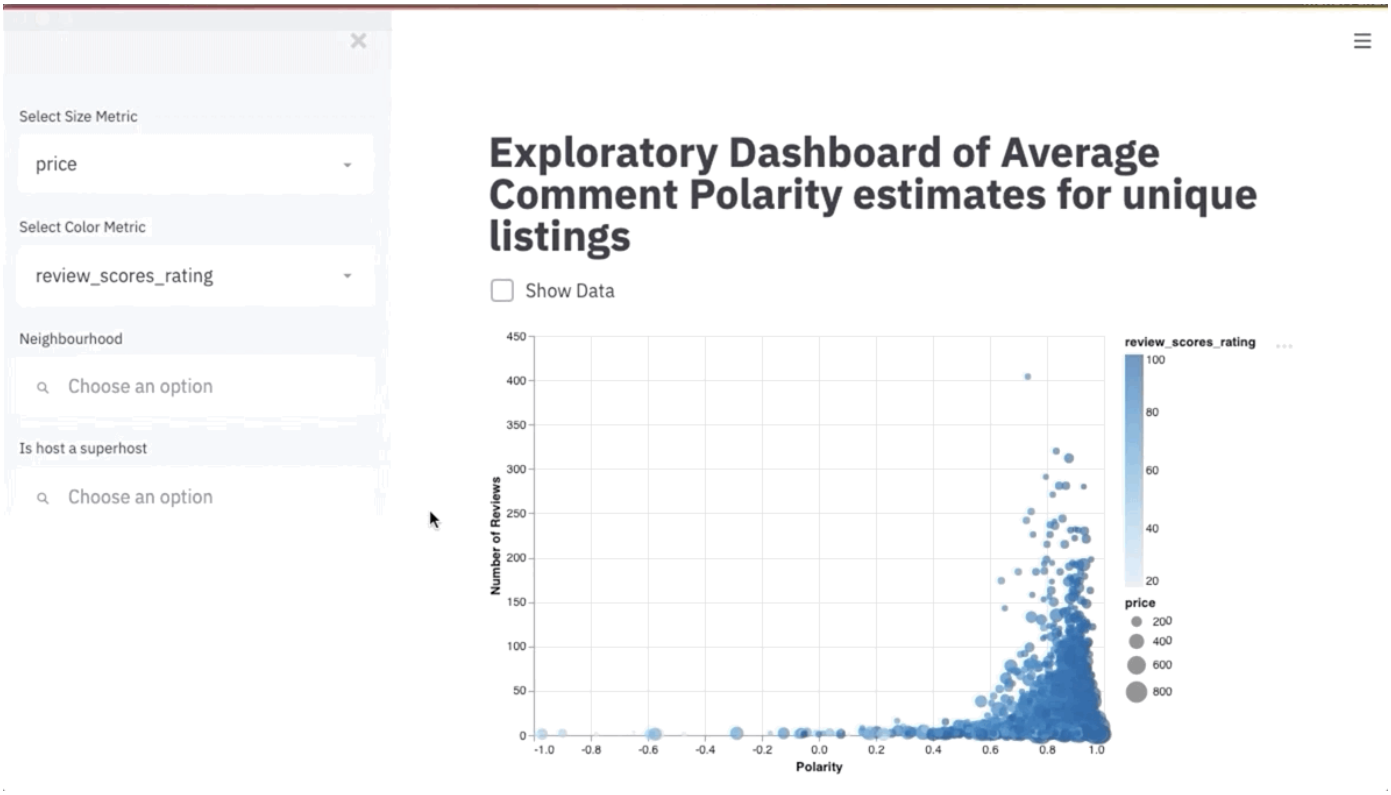
For our example comment (it's the second comment in the image above):

```
polarity = 0.91
```

which seems to agree pretty well with the content of the message. Other negative and positive comments also look like they're being picked up pretty well, so looks like our Sentiment Analyzer is doing a good job!

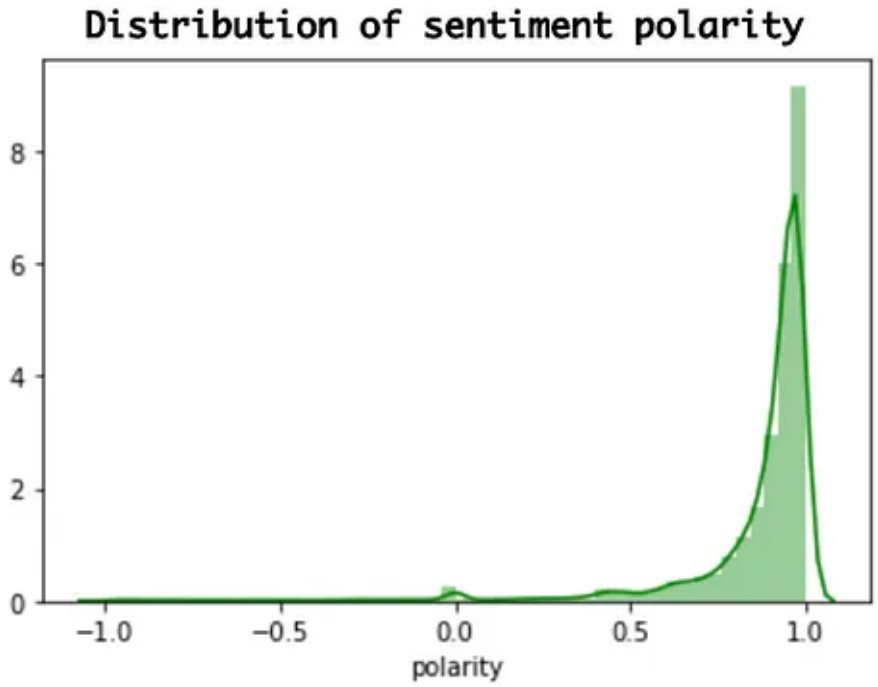
Data exploration and data-driven actionable insights

Since the goal of this project is not to analyze the *polarity* feature per se, we won't be going into too much detail here, but there are a couple of preliminary observations that I'll point out that might pique your interest. I also created this **dashboard** using Streamlit and deployed on Heroku to visualize some relevant data, so feel free to play around with it!



Screencast of polarity dashboard

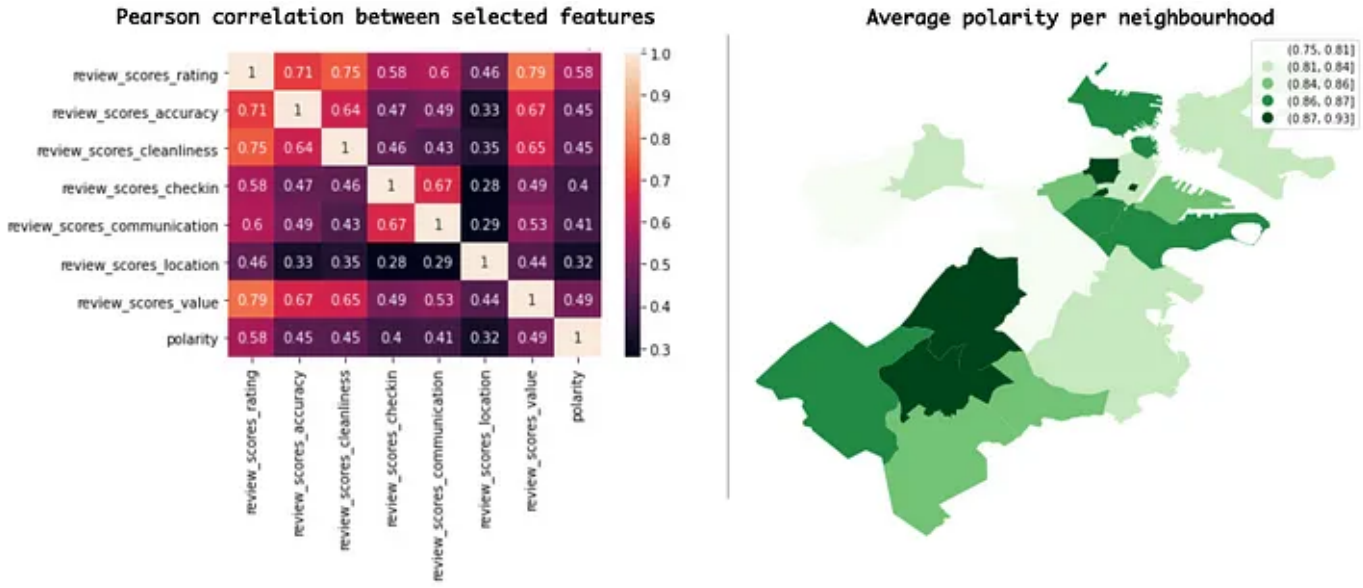
Distribution



Looking at the distribution of *polarity*, we can see that the values are overwhelmingly positive. Some food for thought: do we tend to write comments more often when they

are positive? Do we avoid writing negative comments? Do we have a high threshold of what constitutes a negative experience?

> *Relation to other features*



Relation between polarity and other selected features

We can also look towards answering the question of **what contributes to a positive comment** by examining the correlation between the *polarity* scores and other features in our data such as *rating scores* for *cleanliness*, *location*, *value*, etc… *Polarity* seems to be most highly correlated with the global *review_scores_rating* feature. Interestingly, location seems to be the least important factor taken into account for comment sentiment: *polarity* is least correlated with *review_scores_location.* If we further look at the *polarity* scores grouped by *neighbourhood*, we can see that the neighbourhoods with highest *polarity* vs *location* scores are different. For instance, the top three neighbourhoods ranked by *polarity* are the Leather District, Roslindale, and Jamaica Plain, whereas the top three neighbourhoods ranked by *review_scores_location* are Longwood Medical Area, Back Bay, and the North End. If you were in charge of **providing advice to hosts** regarding ways to increase their rate of positive comments, you're in luck! Arguably the hardest feature for a host to change is the location of their listing, but by working on improving their other scores (eg. cleanliness, ease of check in, etc…) the data would suggest that this will actually reflect more strongly on their comments!

Part 2: Build our recommendation engine

Context

There are many different models for **Collaborative Filtering** that we can use to predict our reviewer's *polarity* scores on listings they haven't visited. **Matrix factorization** tends to yield the best results, it gives us how much a user is aligned with a set of *k* latent features, or underlying tastes. Singular Value Decomposition (**SVD**), which is the analysis model we'll be using, takes a similar approach.

You may be familiar with SVD as a dimensionality reduction technique because, amongst other operations, it gives us the eigenvectors of an input matrix. It has also, however, proven itself to be quite useful for building recommendation systems. Essentially, SVD states that any given matrix *M* can be factorized as a product of three matrices: ***U***, **∑**, and ***V***. *U* is the *n x k* user-latent features matrix, *V* is the *m x k* listing-latent features matrix, and ∑ is a *k x k* diagonal matrix containing the singular values of our original matrix *M*, which just means that each value will represent how important a latent feature is to to predict user preferences. Using these three matrices, we can therefore recover the full *M* matrix of polarity values for all reviewer-listing pairs.

Preparing our data

There's a key processing step we need to take before we can use our dataset for SVD: transform it into a **utility matrix**! A utility matrix is essentially a pivot of our 3 column dataset (*reviewer_id, listing_id, polarity*) into a sparse *n x m* table, where the axes correspond to our *reviewer_id* and *listing_id* columns, and the table is populated with *polarity* values for known *reviewer_id/listing_id* pairs and *NaN* if unknown. We can generate this table super easily with panda's *pivot_table* function.

Then we just remove the per item average of all entries, and voilà! We're ready to run our SVD algorithm!

Implementation

We'll be using scipy's *svds* function to recover the *U*, ∑, and *V* matrices, then convert the ∑ values to a diagonal matrix and use matrix multiplication of all three matrices to recover the full matrix and get the predicted *polarity* values.

```python
def recommend_predictions(df_rec, k):
    """
    :param df_rec: dataframe with polarity values for known reviewer/listing pairs
    :param k: number of features to keep for SVD

    returns the dataframe with predicted polarity values for all (user,item) pairs
    """
    # get utility matrix
    util_mat = df_rec.pivot_table(index='reviewer_id', columns='listing_id', values='polarity')

    # keep track of reviewers and listings
    reviewer_rows = list(util_mat.index)
    reviewer_index = {reviewer_rows[i]: i for i in range(len(reviewer_rows))}
    listing_cols = list(util_mat.columns)
    listing_index = {listing_cols[i]: i for i in range(len(listing_cols))}

    # mask NaN and remove means
    mask = np.isnan(util_mat)
    masked_arr = np.ma.masked_array(util_mat, mask)
    item_means = np.mean(masked_arr, axis=0)
    util_mat = masked_arr.filled(item_means)
    means = np.tile(item_means, (util_mat.shape[0],1))
    util_mat_demeaned = util_mat - means

    # run SVD
    U, sigma, Vt = svds(util_mat_demeaned, k = k)
    sigma = np.diag(sigma)
    all_predicted_polarity = np.dot(np.dot(U, sigma), Vt) + means

    return all_predicted_polarity, reviewer_index, listing_index
```

recommend_predictions.py hosted with ❤ by GitHub view raw

Building our recommendation engine with SVD

With *k=100* latent features, our model performs pretty well on training data with a RMSE of 0.17. However take this with a grain of salt, we didn't split the dataset into train and test data because the Cold Start issue of Collaborative Filtering techniques (i.e. can't introduce unseen listings or users) would require manually creating those datasets, and we might inadvertently start introducing selection bias into our model. The prediction score here was therefore solely based on training data.

Part 3: Make recommendations!

Now that we've taken care of our two main tasks, all that's left is generating our personalised recommendations for a particular user.

Using our initial example, we simply identify the user row in our full *all_predicted_polarity* matrix and rank our predictions according to *polarity*.

```python
listing_id_array = df_rec['listing_id'].unique()

def get_recommendations(predMat, user, N):
    """
    :param predMat: matrix of full predicted polarity sentiment values
    :param user: selected reviewer_id
    :param N: top N recommendations to show

    returns top N recommendations for specified user
    """
    u_index = users_index[user]
    item_i = [items_index[listing_id_array[i]] for i in range(len(listing_id_array))]

    pred_user = [predMat[u_index, i_index] for i_index in item_i]

    d = {'listing_id': listing_id_array, 'predicted_polarity': pred_user}
    user_rec = pd.DataFrame(data=d)
    user_rec.sort_values(by=['predicted_polarity'], ascending=False, inplace=True)
    user_rec.reset_index(inplace=True, drop=True)

    return user_rec[:N]
```

get_recommendations.py hosted with ❤ by GitHub view raw

Generating our top N recommendations for a particular user

	listing_id	predicted_polarity
0	14813006	0.995900
1	12603280	0.993300
2	14760739	0.993300
3	5584915	0.992850
4	8481291	0.992800
5	12699603	0.992800
6	13655073	0.992100
7	14220964	0.991700
8	7841193	0.991700
9	5719606	0.991700

Listing recommendations for our example user based on top 10 predicted polarities

Conclusions

Let's recap: using **Sentiment Analysis**, we were able to determine sentiment polarity for reviewer comments, and using **SVD** for **Collaborative Filtering** we were able predict polarity values for all reviewer-listing pairs based on latent user preferences, finally generating **recommendations** as a ranking of predicted polarities for any given user.

Once you've built your base recommender system, you could take it anywhere your data or business goals lead you. I've briefly mentioned a few insights that I thought may lead to **impactful recommendations**, such as exploring what drives positive polarities or how to maximize them for hosts, but feel free to use the tools we've learnt and implemented to explore to your heart's content.

> *Thanks for reading! To see the full code, check out my Github repository for this project.*

If you want to further delve into the topic, I'd suggest these posts as a good starting point:

- A good overview of Recommendation Systems using the MovieLes Dataset: https://towardsdatascience.com/how-to-build-a-simple-recommender-system-in-python-375093c3fb7d
- Very clear review of Nearest Neighbour and Matrix Factorization approaches to Collaborative Filtering: https://towardsdatascience.com/intro-to-recommender-system-collaborative-filtering-64a238194a26
- Extensive post covering Sentiment Analysis that includes explanations, use cases, tutorials, etc : https://monkeylearn.com/sentiment-analysis/
- The famous SVD algorithm for recommendations popularized by Simon Funk during the Netflix Prize contest, has some very innovative and creative ideas: https://sifter.org/simon/journal/20061211.html

> *And if you want to learn more about me, check out my Linkedin or personal portfolio page!*

Recommender Systems | Sentiment Analysis | Airbnb | Recommendations | How To

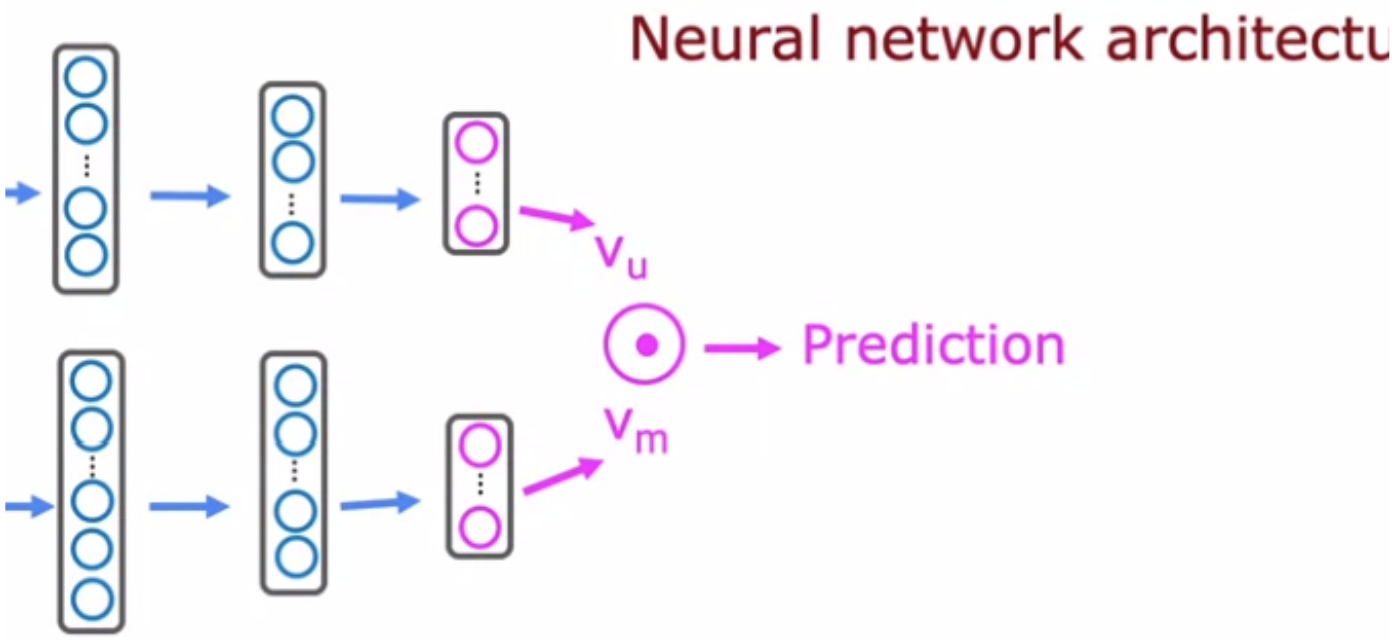
Follow

Written by Alexandra Gastone

4 Followers

Computational Neuroscientist

Recommended from Medium



George Pipis

Content-Based Recommender Systems in TensorFlow and BERT Embeddings

A practical example of Content-Based Recommender Systems using TensorFlow and BERT Embeddings on ads and click-through rate data.

· 4 min read · Feb 20

267 1

Abhay Parashar in Level Up Coding

Different Approaches For Building Recommender Systems Using Python

A Comprehensive Guide To Recommender Systems

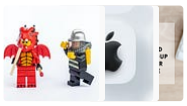
✨ · 8 min read · Mar 8

60 2

Lists

Practical Guides to Machine Learning

10 stories · 125 saves



Tech & Tools

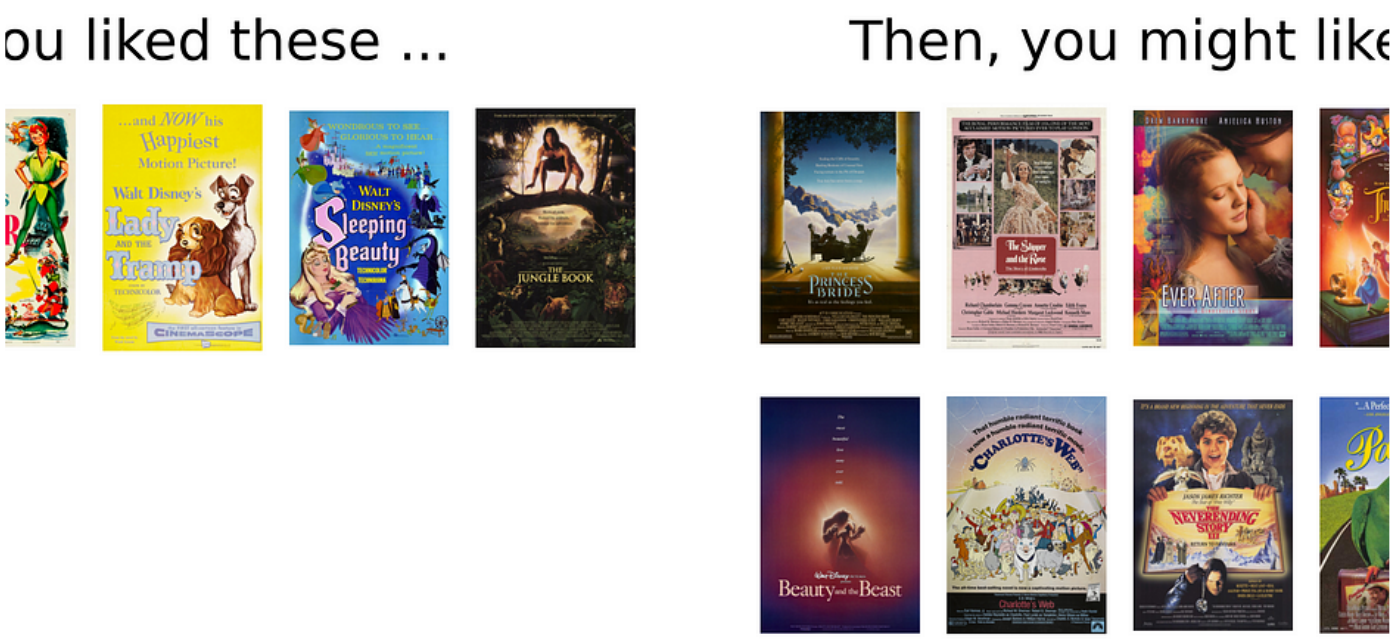
15 stories · 8 saves

Level Up Your Medium Game

14 stories · 65 saves

Business 101

25 stories · 155 saves



___ in Towards AI

How To Quickly Build A Content-Based Filtering Recommender System Using A Vector Database

No feature engineering needed 😵

5 min read · Feb 5

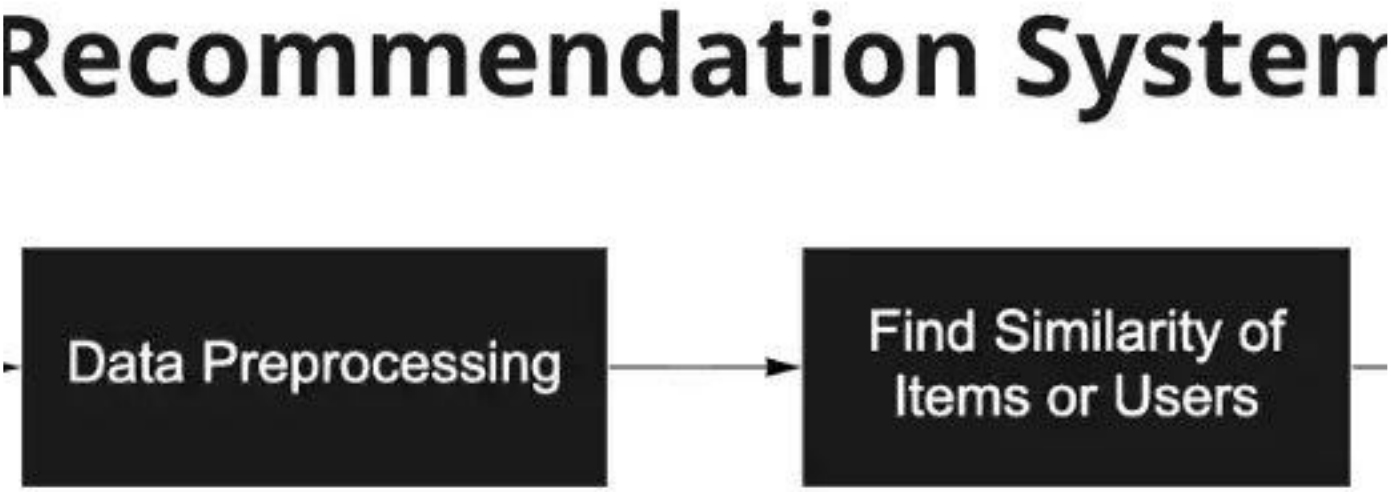
13

Dr. Robert Kübler in Towards Data Science

Introduction to Embedding-Based Recommender Systems

Learn to build a simple matrix factorization recommender in TensorFlow

· 13 min read · Jan 25



Ebuka (Gaus Octavio) in Level Up Coding

Music Recommendation System Built With Python And Machine Learning

With the dawn of tech, recommendation systems are now playing a crucial aspect in today's digital landscape. They are used to provide...

4 min read · Jan 15

23

Snehal Nair

Everything you need to know about Recommender Systems

Understand data munging in PySpark while building a recommender system that utilises matrix factorisation technique — Alternating Least...

· 11 min read · Mar 5

58