**Quick Guide to Build a Recommendation Engine in Python**

[**AARSHAY JAIN**](https://www.analyticsvidhya.com/blog/author/aarshay/)**, JUNE 2, 2016**

**Introduction**

*This could help you in building your first project!*

Be it a fresher or an experienced professional in data science, doing voluntary projects always adds to one’s candidature. My sole reason behind writing this article is to get your started with recommendation systems so that you can build one. If you struggle to get open data, write to me in comments.

Recommendation engines are nothing but an automated form of a “shop counter guy”. You ask him for a product. Not only he shows that product, but also the related ones which you could buy. They are well trained in cross selling and up selling. So, does our recommendation engines.

The ability of these engines to recommend personalized content, based on past behavior is incredible. It brings customer delight and gives them a reason to keep returning to the website.

In this post, I will cover the fundamentals of creating a recommendation system using [GraphLab](https://www.analyticsvidhya.com/blog/2015/12/started-graphlab-python/" \t "_blank) in Python. We will get some intuition into how recommendation work and create basic popularity model and a collaborative filtering model.

**Topics Covered**

1. Type of Recommendation Engines
2. The MovieLens DataSet
3. A simple popularity model
4. A Collaborative Filtering Model
5. Evaluating Recommendation Engines

Before moving forward, I would like to extend my sincere gratitude to the **Coursera’s**[**Machine Learning Specialization**](https://www.coursera.org/specializations/machine-learning)**by University of Washington**. This course has been instrumental in my understanding of the concepts and this post is an illustration of my learnings from the same.

**1. Type of Recommendation Engines**

Before taking a look at the different types of recommendation engines, lets take a step back and see if we can make some intuitive recommendations. Consider the following cases:

**Case 1: Recommend the most popular items**

A simple approach could be to recommend the items which are liked by most number of users. This is a blazing fast and dirty approach and thus has a major drawback. The things is, there is **no personalization**involved with this approach.

Basically the most popular items would be same for each user since popularity is defined on the entire user pool. So everybody will see the same results. It sounds like, ‘a website recommends you to buy microwave just because it’s been liked by other users and doesn’t care if you are even interested in buying or not’.

Surprisingly, such approach still works in places like news portals. Whenever you login to say bbcnews, you’ll see a column of “Popular News” which is subdivided into sections and the most read articles of each sections are displayed. This approach can work in this case because:

* There is division by section so user can look at the section of his interest.
* At a time there are only a few hot topics and there is a high chance that a user wants to read the news which is being read by most others

**Case 2: Using a classifier to make recommendation**

We already know lots of **classification algorithms**. Let’s see how we can use the same technique to make recommendations. Classifiers are parametric solutions so we just need to define some parameters (features) of the user and the item. The outcome can be 1 if the user likes it or 0 otherwise. This might work out in some cases because of following advantages:

* Incorporates personalization
* It can work even if the user’s past history is short or not available

But has some major drawbacks as well because of which it is not used much in practice:

* The features might actually not be available or even if they are, they may not be sufficient to make a good classifier
* As the number of users and items grow, making a good classifier will become exponentially difficult

**Case 3: Recommendation Algorithms**

Now lets come to the special class of algorithms which are tailor-made for solving the recommendation problem. There are typically two types of algorithms – Content Based and Collaborative Filtering. You should refer to our [previous article](https://www.analyticsvidhya.com/blog/2016/03/exploring-building-banks-recommendation-system/) to get a complete sense of how they work. I’ll give a short recap here.

1. **Content based algorithms:**
   * **Idea:** If you like an item then you will also like a “similar” item
   * Based on similarity of the items being recommended
   * It generally works well when its easy to determine the context/properties of each item. For instance when we are recommending the same kind of item like a movie recommendation or song recommendation.
2. **Collaborative filtering algorithms:**
   * **Idea:** If a person A likes item 1, 2, 3 and B like 2,3,4 then they have similar interests and A should like item 4 and B should like item 1.
   * This algorithm is entirely based on the past behavior and not on the context. This makes it one of the most commonly used algorithm as it is not dependent on any additional information.
   * For instance: product recommendations by e-commerce player like Amazon and merchant recommendations by banks like American Express.
   * Further, there are several types of collaborative filtering algorithms :
     1. **User-User Collaborative filtering:** Here we find look alike customers (based on similarity) and offer products which first customer’s look alike has chosen in past. This algorithm is very effective but takes a lot of time and resources. It requires to compute every customer pair information which takes time. Therefore, for big base platforms, this algorithm is hard to implement without a very strong parallelizable system.
     2. **Item-Item Collaborative filtering:** It is quite similar to previous algorithm, but instead of finding customer look alike, we try finding item look alike. Once we have item look alike matrix, we can easily recommend alike items to customer who have purchased any item from the store. This algorithm is far less resource consuming than user-user collaborative filtering. Hence, for a new customer the algorithm takes far lesser time than user-user collaborate as we don’t need all similarity scores between customers. And with fixed number of products, product-product look alike matrix is fixed over time.
     3. **Other simpler algorithms:** There are other approaches like [market basket analysis](https://www.analyticsvidhya.com/blog/2014/08/visualizing-market-basket-analysis/), which generally do not have high predictive power than the algorithms described above.

**2. The MovieLens DataSet**

We will be using the MovieLens dataset for this purpose. It has been collected by the GroupLens Research Project at the University of Minnesota. MovieLens 100K dataset can be downloaded from [here](http://grouplens.org/datasets/movielens/100k/). It consists of:

* **100,000 ratings** (1-5) from 943 users on 1682 movies.
* Each user has rated **at least 20 movies.**
* Simple demographic info for the users (age, gender, occupation, zip)
* Genre information of movies

Lets load this data into Python. There are many files in the **ml-100k.zip** file which we can use. Lets load the three most importance files to get a sense of the data. I also recommend you to read the *readme* document which gives a lot of information about the difference files.

import pandas as pd

# pass in column names for each CSV and read them using pandas.

# Column names available in the readme file

#Reading users file:

u\_cols = ['user\_id', 'age', 'sex', 'occupation', 'zip\_code']

users = pd.read\_csv('ml-100k/u.user', sep='|', names=u\_cols,

encoding='latin-1')

#Reading ratings file:

r\_cols = ['user\_id', 'movie\_id', 'rating', 'unix\_timestamp']

ratings = pd.read\_csv('ml-100k/u.data', sep='\t', names=r\_cols,

encoding='latin-1')

#Reading items file:

i\_cols = ['movie id', 'movie title' ,'release date','video release date', 'IMDb URL', 'unknown', 'Action', 'Adventure',

'Animation', 'Children\'s', 'Comedy', 'Crime', 'Documentary', 'Drama', 'Fantasy',

'Film-Noir', 'Horror', 'Musical', 'Mystery', 'Romance', 'Sci-Fi', 'Thriller', 'War', 'Western']

items = pd.read\_csv('ml-100k/u.item', sep='|', names=i\_cols,

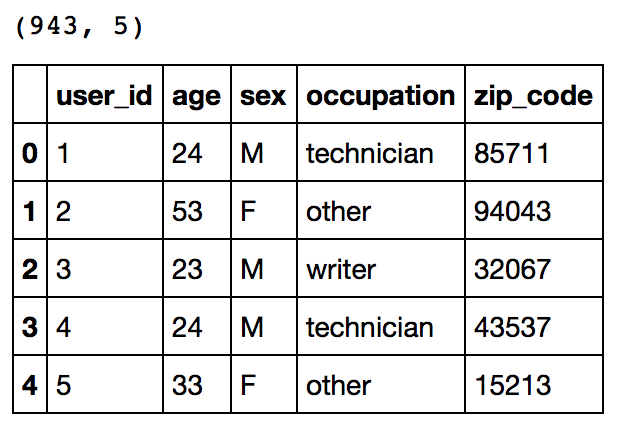
encoding='latin-1')

Now lets take a peak into the content of each file to understand them better.

* **Users**

print users.shape

users.head()

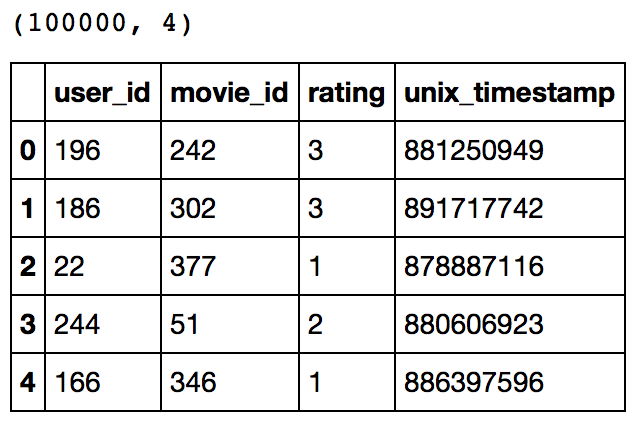
[](https://www.analyticsvidhya.com/wp-content/uploads/2016/06/1.-users.png)

This reconfirms that there are 943 users and we have 5 features for each namely their unique ID, age, gender, occupation and the zip code they are living in.

* **Ratings**

print ratings.shape

ratings.head()

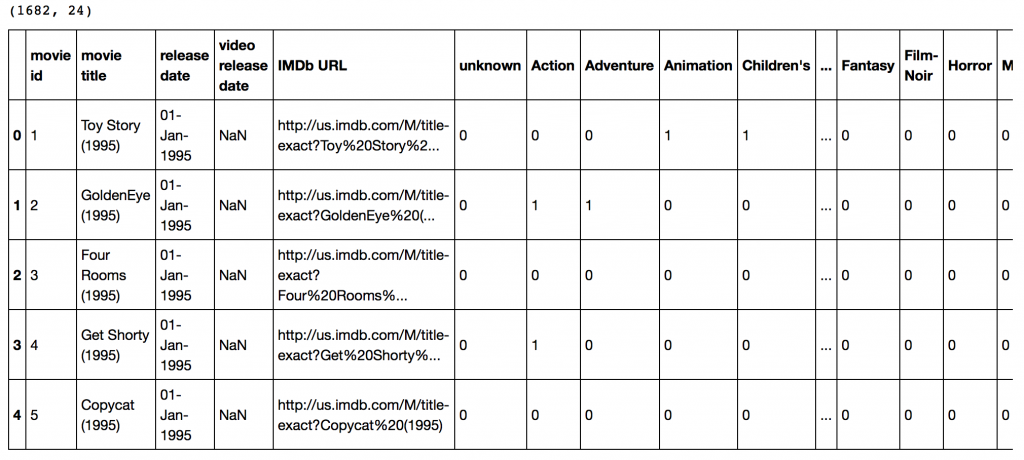
[](https://www.analyticsvidhya.com/wp-content/uploads/2016/06/2.-ratings.png)

This confirms that there are 100K ratings for different user and movie combinations. Also notice that each rating has a timestamp associated with it.

* **Items**

print items.shape

items.head()

[](https://www.analyticsvidhya.com/wp-content/uploads/2016/06/3.-items.png)

This dataset contains attributes of the 1682 movies. There are 24 columns out of which 19 specify the genre of a particular movie. The last 19 columns are for each genre and a value of 1 denotes movie belongs to that genre and 0 otherwise.

Now we have to divide the ratings data set into test and train data for making models. Luckily GroupLens provides pre-divided data wherein the test data has 10 ratings for each user, i.e. 9430 rows in total. Lets load that:

r\_cols = ['user\_id', 'movie\_id', 'rating', 'unix\_timestamp']

ratings\_base = pd.read\_csv('ml-100k/ua.base', sep='\t', names=r\_cols, encoding='latin-1')

ratings\_test = pd.read\_csv('ml-100k/ua.test', sep='\t', names=r\_cols, encoding='latin-1')

ratings\_base.shape, ratings\_test.shape

Output: ((90570, 4), (9430, 4))

Since we’ll be using GraphLab, lets convert these in SFrames.

import graphlab

train\_data = graphlab.SFrame(ratings\_base)

test\_data = graphlab.SFrame(ratings\_test)

We can use this data for training and testing. Now that we have gathered all the data available. Note that here we have user behaviour as well as attributes of the users and movies. So we can make content based as well as collaborative filtering algorithms.

**3. A Simple Popularity Model**

Lets start with making a popularity based model, i.e. the one where **all the users have same recommendation** based on the most popular choices. We’ll use the  graphlab recommender functions popularity\_recommender for this.

We can train a recommendation as:

popularity\_model = graphlab.popularity\_recommender.create(train\_data, user\_id='user\_id', item\_id='movie\_id', target='rating')

Arguments:

* **train\_data**: the SFrame which contains the required data
* **user\_id**: the column name which represents each user ID
* **item\_id**: the column name which represents each item to be recommended
* **target:** the column name representing scores/ratings given by the user

Lets use this model to make top 5 recommendations for first 5 users and see what comes out:

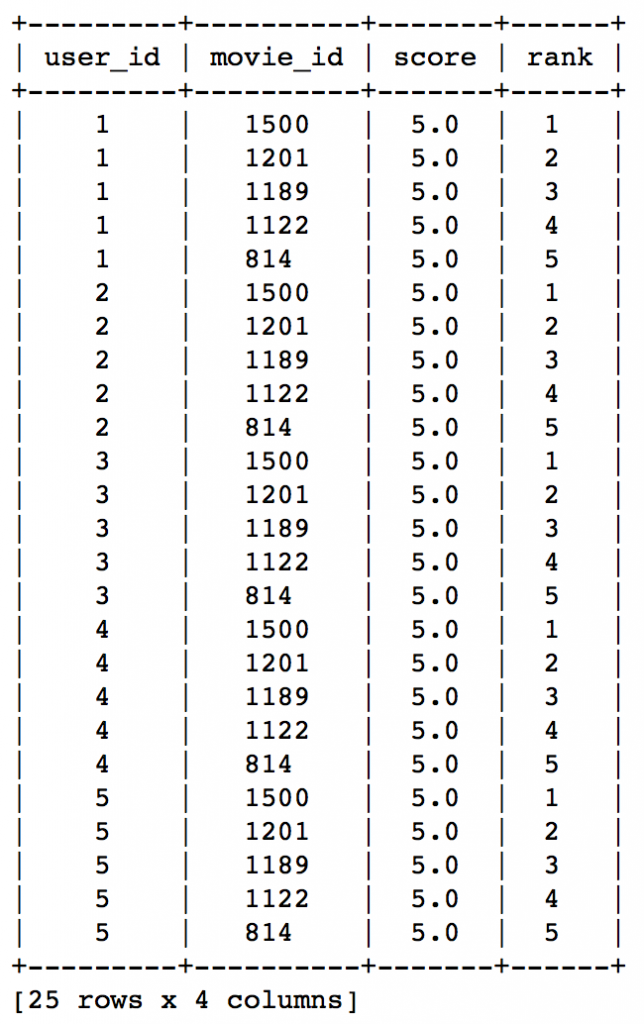
#Get recommendations for first 5 users and print them

#users = range(1,6) specifies user ID of first 5 users

#k=5 specifies top 5 recommendations to be given

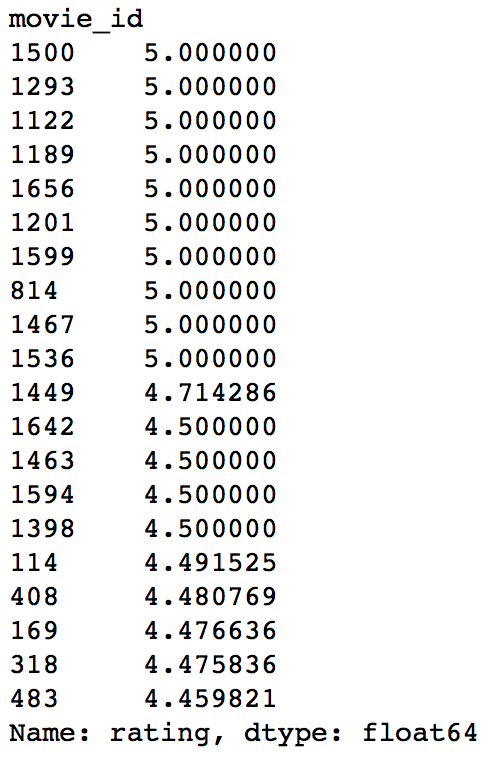
popularity\_recomm = popularity\_model.recommend(users=range(1,6),k=5)

popularity\_recomm.print\_rows(num\_rows=25)

[](https://www.analyticsvidhya.com/wp-content/uploads/2016/06/4.-popularity-recomm.png)

Did you notice something? The recommendations for all users are same – 1500,1201,1189,1122,814 in the same order. This can be verified by checking the movies with highest mean recommendations in our ratings\_base data set:

ratings\_base.groupby(by='movie\_id')['rating'].mean().sort\_values(ascending=False).head(20)

[](https://www.analyticsvidhya.com/wp-content/uploads/2016/06/5.-mean-ratings.png)

This confirms that all the recommended movies have an average rating of 5, i.e. all the users who watched the movie gave a top rating. Thus we can see that our popularity system works as expected. But it is good enough? We’ll analyze it in detail later.

**4. A Collaborative Filtering Model**

Lets start by understanding the basics of a collaborative filtering algorithm. The core idea works in 2 steps:

1. Find similar items by using a similarity metric
2. For a user, recommend the items most similar to the items (s)he already likes

To give you a high level overview, this is done by making an **item-item matrix** in which we keep a record of the pair of items which were rated together.

In this case, an item is a movie. Once we have the matrix, we use it to determine the best recommendations for a user based on the movies he has already rated. Note that there a few more things to take care in actual implementation which would require deeper mathematical introspection, which I’ll skip for now.

I would just like to mention that there are 3 types of item similarity metrics supported by graphlab. These are:

1. **Jaccard Similarity:**
   * Similarity is based on the number of users which have rated item A and B divided by the number of users who have rated either A or B
   * It is typically used where we don’t have a numeric rating but just a boolean value like a product being bought or an add being clicked
2. **Cosine Similarity:**
   * Similarity is the cosine of the angle between the 2 vectors of the item vectors of A and B
   * Closer the vectors, smaller will be the angle and larger the cosine
3. **Pearson Similarity**
   * Similarity is the pearson coefficient between the two vectors.

Lets create a model based on item similarity as follow:

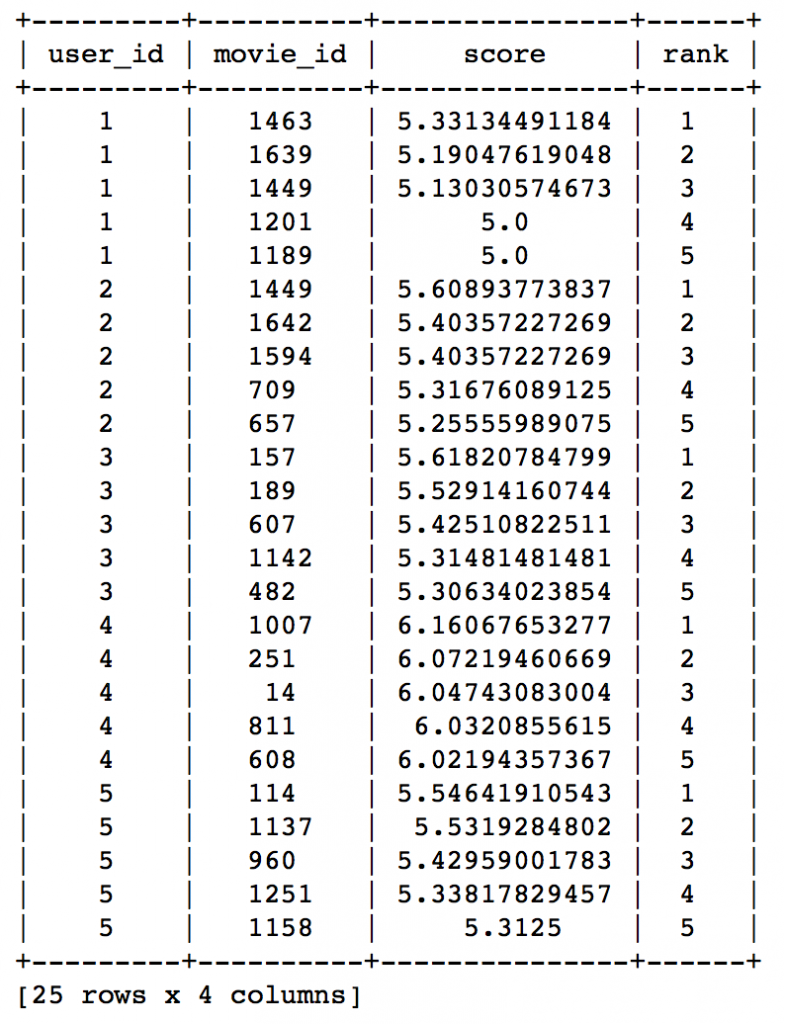
#Train Model

item\_sim\_model = graphlab.item\_similarity\_recommender.create(train\_data, user\_id='user\_id', item\_id='movie\_id', target='rating', similarity\_type='pearson')

#Make Recommendations:

item\_sim\_recomm = item\_sim\_model.recommend(users=range(1,6),k=5)

item\_sim\_recomm.print\_rows(num\_rows=25)

[](https://www.analyticsvidhya.com/wp-content/uploads/2016/06/6.-similarity-model-1.png)

Here we can see that the recommendations are different for each user. So, personalization exists. But how good is this model? We need some means of evaluating a recommendation engine. Lets focus on that in the next section.

**5. Evaluating Recommendation Engines**

For evaluating recommendation engines, we can use the concept of precision-recall. You must be familiar with this in terms of classification and the idea is very similar. Let me define them in terms of recommendations.

* **Recall:**
  + What ratio of items that a user likes were actually recommended.
  + If a user likes say 5 items and the recommendation decided to show 3 of them, then the recall is 0.6
* **Precision**
  + Out of all the recommended items, how many the user actually liked?
  + If 5 items were recommended to the user out of which he liked say 4 of them, then precision is 0.8

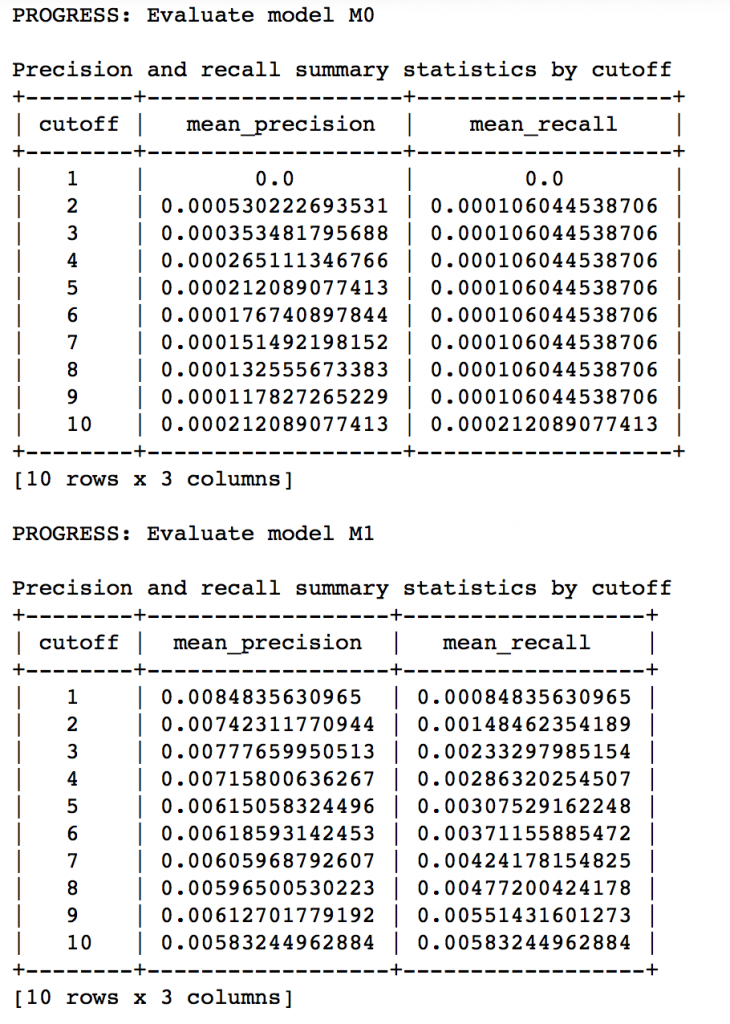
Now if we think about recall, how can we maximize it? If we simply recommend all the items, they will definitely cover the items which the user likes. So we have 100% recall! But think about precision for a second. If we recommend say 1000 items and user like only say 10 of them then precision is 0.1%. This is really low. Our aim is to maximize both precision and recall.

An idea recommender system is the one which only recommends the items which user likes. So in this case precision=recall=1. This is an optimal recommender and we should try and get as close as possible.

Lets compare both the models we have built till now based on precision-recall characteristics:

model\_performance = graphlab.compare(test\_data, [popularity\_model, item\_sim\_model])

graphlab.show\_comparison(model\_performance,[popularity\_model, item\_sim\_model])

[](https://www.analyticsvidhya.com/wp-content/uploads/2016/06/7.-evaluate.png)

Here we can make 2 very quick observations:

1. The item similarity model is definitely better than the popularity model (by atleast 10x)
2. On an absolute level, even the item similarity model appears to have a poor performance. It is far from being a useful recommendation system.

There is a big scope of improvement here. But I leave it up to you to figure out how to improve this further. I would like to give a couple of tips:

1. Try leveraging the additional context information which we have
2. Explore more sophisticated algorithms like matrix factorization

In the end, I would like to mention that along with GraphLab, you can also use some other open source python packages like the following:

* [**Crab**](http://muricoca.github.io/crab/).
* [Surprise](https://github.com/NicolasHug/Surprise)
* [Python Recsys](https://github.com/ocelma/python-recsys)
* [MRec](https://github.com/Mendeley/mrec)

**End Notes**

In this article, we traversed through the process of making a basic recommendation engine in Python using GrpahLab. We started by understanding the fundamentals of recommendations. Then we went on to load the MovieLens 100K data set for the purpose of experimentation.

Subsequently we made a first model as a simple popularity model in which the most popular movies were recommended for each user. Since this lacked personalization, we made another model based on collaborative filtering and observed the impact of personalization.

Finally, we discussed precision-recall as evaluation metrics for recommendation systems and on comparison found the collaborative filtering model to be more than 10x better than the popularity model.

**Comprehensive Guide to build a Recommendation Engine from scratch (in Python)**

[**PULKIT SHARMA**](https://www.analyticsvidhya.com/blog/author/pulkits/)**, JUNE 21, 2018**

**Introduction**

In today’s world, every customer is faced with multiple choices. For example, If I’m looking for a book to read without any specific idea of what I want, there’s a wide range of possibilities how my search might pan out. I might waste a lot of time browsing around on the internet and trawling through various sites hoping to strike gold. I might look for recommendations from other people.

But if there was a site or app which could recommend me books based on what I have read previously, that would be a massive help. Instead of wasting time on various sites, I could just log in and voila! 10 recommended books tailored to my taste.



This is what recommendation engines do and their power is being harnessed by most businesses these days. From Amazon to Netflix, Google to Goodreads, recommendation engines are one of the most widely used applications of machine learning techniques.

In this article, we will cover various types of recommendation engine algorithms and fundamentals of creating them in Python. We will also see the mathematics behind the workings of these algorithms. Finally, we will create our own recommendation engine using matrix factorization.

**Table of Contents**

1. What are recommendation engines?
2. How does a recommendation engine work?
   1. Data collection
   2. Data storage
   3. Filtering the data
      1. Content based filtering
      2. Collaborative filtering
3. Case study in Python using the MovieLens dataset
4. Building collaborative filtering model from scratch
5. Building Simple popularity and collaborative filtering model using Turicreate
6. Introduction to matrix factorization
7. Building a recommendation engine using matrix factorization
8. Evaluation metrics for recommendation engines
   1. Recall
   2. Precision
   3. RMSE (Root Mean Squared Error)
   4. Mean Reciprocal Rank
   5. MAP at k (Mean Average Precision at cutoff k)
   6. NDCG (Normalized Discounted Cumulative Gain)
9. What else can be tried?

**1. What are recommendation engines?**

Till recently, people generally tended to buy products recommended to them by their friends or the people they trust. This used to be the primary method of purchase when there was any doubt about the product. But with the advent of the digital age, that circle has expanded to include online sites that utilize some sort of recommendation engine.

**A recommendation engine filters the data using different algorithms and recommends the most relevant items to users. It first captures the past behavior of a customer and based on that, recommends products which the users might be likely to buy.**

If a completely new user visits an e-commerce site, that site will not have any past history of that user. So how does the site go about recommending products to the user in such a scenario? One possible solution could be to recommend the best selling products, i.e. the products which are high in demand. Another possible solution could be to recommend the products which would bring the maximum profit to the business.

If we can recommend a few items to a customer based on their needs and interests, it will create a positive impact on the user experience and lead to frequent visits. Hence, businesses nowadays are building smart and intelligent recommendation engines by studying the past behavior of their users.

Now that we have an intuition of recommendation engines, let’s now look at how they work.

**2. How does a recommendation engine work?**

Before we deep dive into this topic, first we’ll think of how we can recommend items to users:

* We can recommend items to a user which are most popular among all the users
* We can divide the users into multiple segments based on their preferences (user features) and recommend items to them based on the segment they belong to

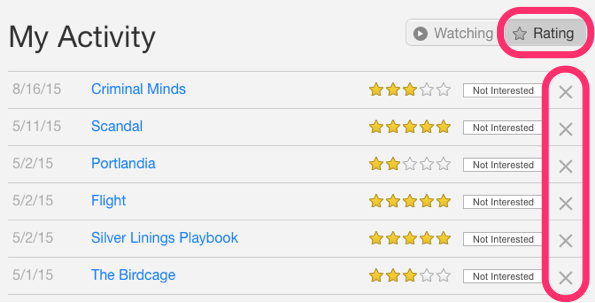
Both of the above methods have their drawbacks. In the first case, the most popular items would be the same for each user so everybody will see the same recommendations. While in the second case, as the number of users increases, the number of features will also increase. So classifying the users into various segments will be a very difficult task.

The main problem here is that we are unable to tailor recommendations based on the specific interest of the users. It’s like Amazon is recommending you buy a laptop just because it’s been bought by the majority of the shoppers. But thankfully, Amazon (or any other big firm) does not recommend products using the above mentioned approach. They use some personalized methods which help them in recommending products more accurately.

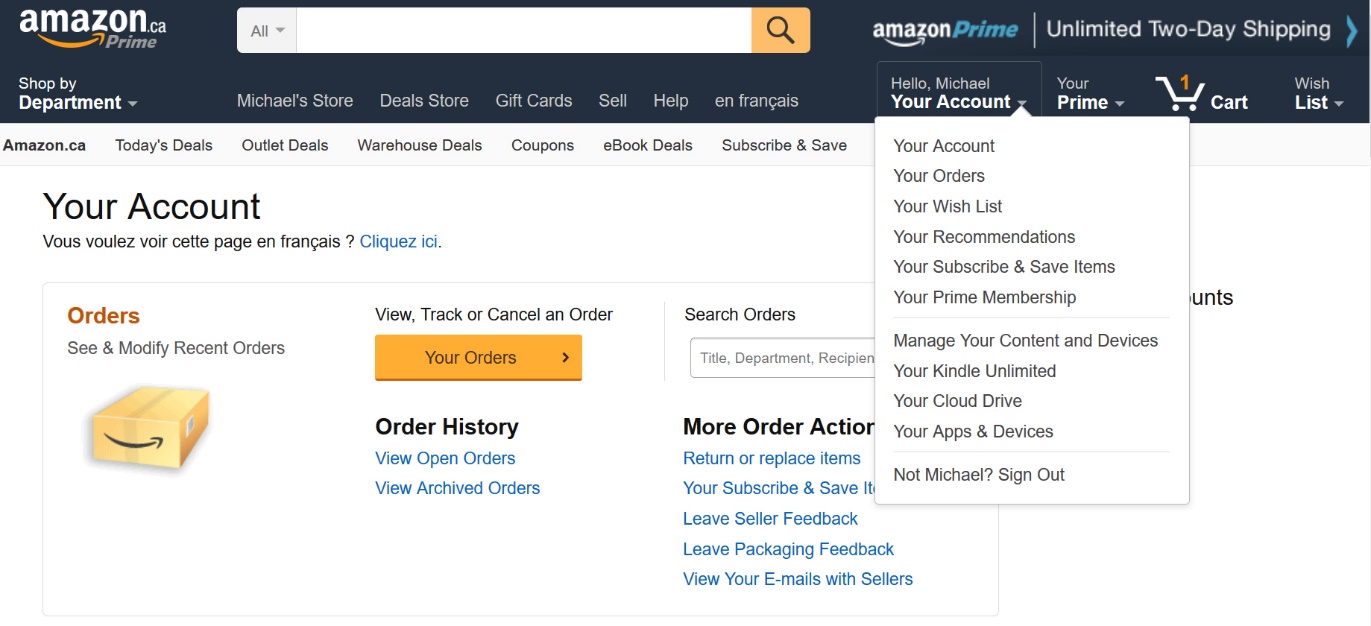
Let’s now focus on how a recommendation engine works by going through the following steps.

**2.1 Data collection**

This is the first and most crucial step for building a recommendation engine. The data can be collected by two means: explicitly and implicitly. Explicit data is information that is provided intentionally, i.e. input from the users such as movie ratings. Implicit data is information that is not provided intentionally but gathered from available data streams like search history, clicks, order history, etc.



In the above image, Netflix is collecting the data explicitly in the form of ratings given by user to different movies.



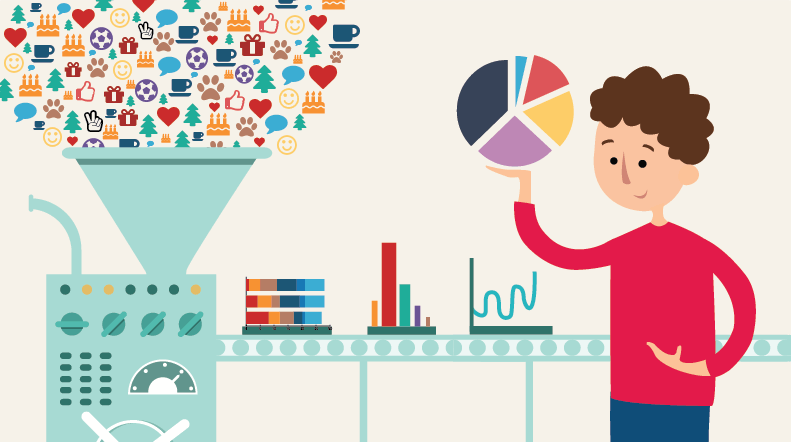
Here the order history of a user is recorded by Amazon which is an example of implicit mode of data collection.

**2.2 Data storage**

The amount of data dictates how good the recommendations of the model can get. For example, in a movie recommendation system, the more ratings users give to movies, the better the recommendations get for other users. The type of data plays an important role in deciding the type of storage that has to be used. This type of storage could include a standard SQL database, a NoSQL database or some kind of object storage.

**2.3 Filtering the data**

After collecting and storing the data, we have to filter it so as to extract the relevant information required to make the final recommendations.

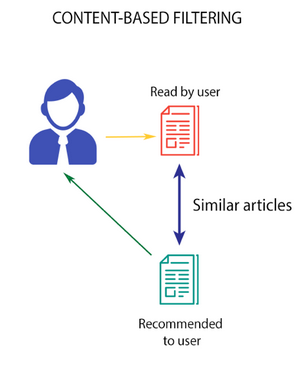


*Source: intheshortestrun*

There are various algorithms that help us make the filtering process easier. In the next section, we will go through each algorithm in detail.

**2.3.1 Content based filtering**

This algorithm recommends products which are similar to the ones that a user has liked in the past.

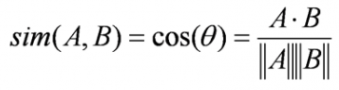


*Source: Medium*

For example, if a person has liked the movie “Inception”, then this algorithm will recommend movies that fall under the same genre. But how does the algorithm understand which genre to pick and recommend movies from?

**Consider the example of Netflix.** They save all the information related to each user in a vector form. This vector contains the past behavior of the user, i.e. the movies liked/disliked by the user and the ratings given by them. This vector is known as the *profile vector*. All the information related to movies is stored in another vector called the *item vector*. Item vector contains the details of each movie, like genre, cast, director, etc.

The content-based filtering algorithm finds the cosine of the angle between the profile vector and item vector, i.e. **cosine similarity**. Suppose A is the profile vector and B is the item vector, then the similarity between them can be calculated as:



Based on the cosine value, which ranges between -1 to 1, the movies are arranged in descending order and one of the two below approaches is used for recommendations:

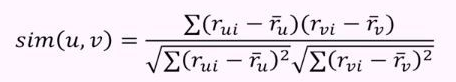
* **Top-n approach**: where the top n movies are recommended (Here n can be decided by the business)
* **Rating scale approach**: Where a threshold is set and all the movies above that threshold are recommended

Other methods that can be used to calculate the similarity are:

* **Euclidean Distance**: Similar items will lie in close proximity to each other if plotted in n-dimensional space. So, we can calculate the distance between items and based on that distance, recommend items to the user. The formula for the euclidean distance is given by:

https://s3-ap-south-1.amazonaws.com/av-blog-media/wp-content/uploads/2018/05/2zjgw1x1.png

* **Pearson’s Correlation**: It tells us how much two items are correlated. Higher the correlation, more will be the similarity. Pearson’s correlation can be calculated using the following formula:



A major drawback of this algorithm is that it is limited to recommending items that are of the same type. It will never recommend products which the user has not bought or liked in the past. So if a user has watched or liked only action movies in the past, the system will recommend only action movies. It’s a very narrow way of building an engine.

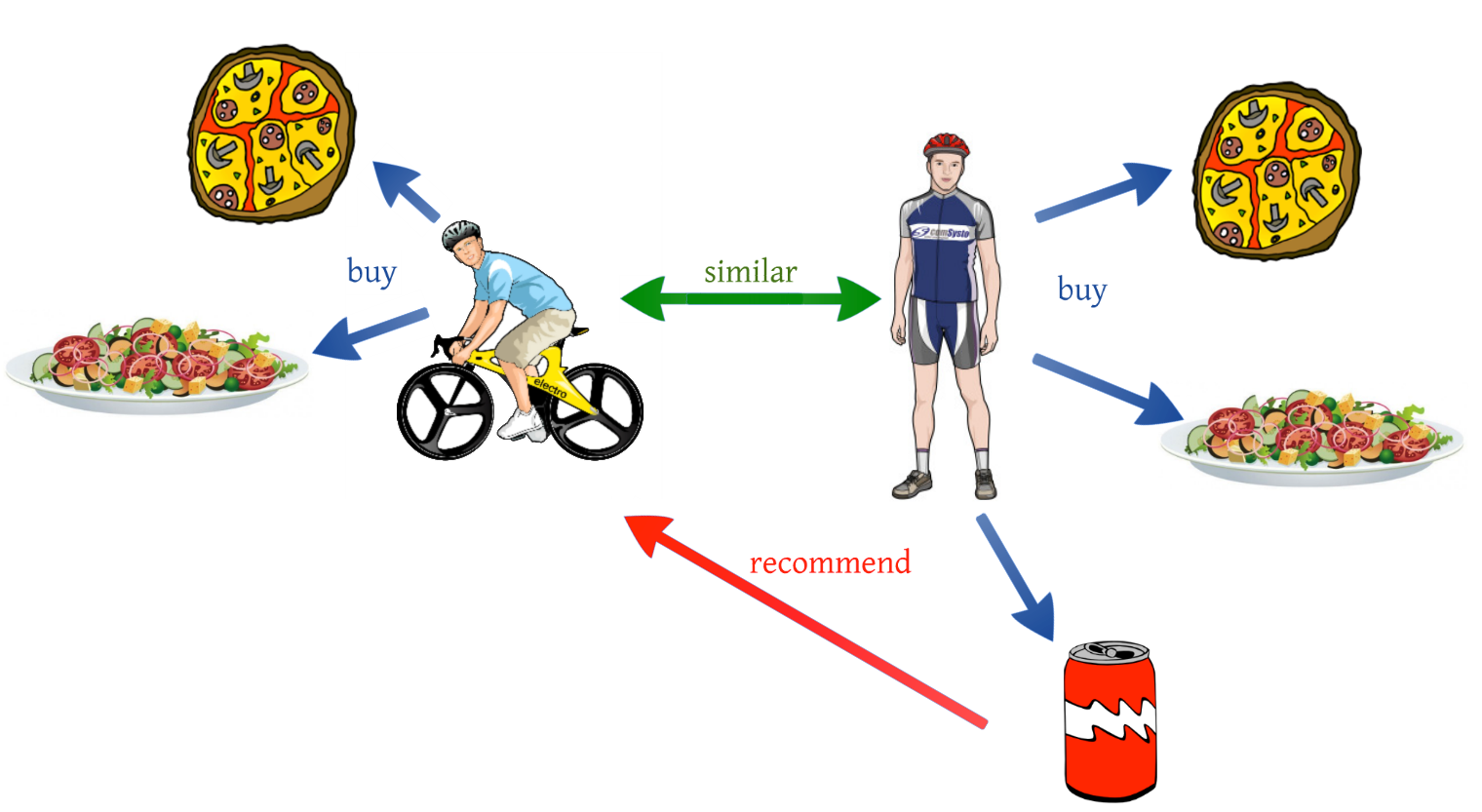
To improve on this type of system, we need an algorithm that can recommend items not just based on the content, but the behavior of users as well.

**2.3.2 Collaborative filtering**

Let us understand this with an example. If person A likes 3 movies, say Interstellar, Inception and Predestination, and person B likes Inception, Predestination and The Prestige, then they have almost similar interests. We can say with some certainty that A should like The Prestige and B should like Interstellar. The collaborative filtering algorithm uses “User Behavior” for recommending items. This is one of the most commonly used algorithms in the industry as it is not dependent on any additional information. There are different types of collaborating filtering techniques and we shall look at them in detail below.

**User-User collaborative filtering**

This algorithm first finds the similarity score between users. Based on this similarity score, it then picks out the most similar users and recommends products which these similar users have liked or bought previously.



*Source: Medium*

In terms of our movies example from earlier, this algorithm finds the similarity between each user based on the ratings they have previously given to different movies. The prediction of an item for a user *u* is calculated by computing the weighted sum of the user ratings given by other users to an item *i*.

The prediction *Pu,i* is given by:

https://s3-ap-south-1.amazonaws.com/av-blog-media/wp-content/uploads/2018/05/Screenshot-from-2018-05-29-20-15-31.png

Here,

* *Pu,i* is the prediction of an item
* *Rv,i* is the rating given by a user *v* to a movie *i*
* *Su,v* is the similarity between users

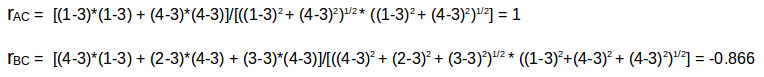
Now, we have the ratings for users in profile vector and based on that we have to predict the ratings for other users. Following steps are followed to do so:

1. For predictions we need the similarity between the user u and v. We can make use of Pearson correlation.
2. First we find the items rated by both the users and based on the ratings, correlation between the users is calculated.
3. The predictions can be calculated using the similarity values. This algorithm, first of all calculates the similarity between each user and then based on each similarity calculates the predictions. **Users having higher correlation will tend to be similar.**
4. Based on these prediction values, recommendations are made. Let us understand it with an example:

Consider the user-movie rating matrix:

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| User/Movie | x1 | x2 | x3 | x4 | x5 | Mean User Rating |
| A | 4 | 1 | – | 4 | – | 3 |
| B | – | 4 | – | 2 | 3 | 3 |
| C | – | 1 | – | 4 | 4 | 3 |

Here we have a user movie rating matrix. To understand this in a more practical manner, let’s find the similarity between users (A, C) and (B, C) in the above table. Common movies rated by A/[ and C are movies x2 and x4 and by B and C are movies x2, x4 and x5.



The correlation between user A and C is more than the correlation between B and C. Hence users A and C have more similarity and the movies liked by user A will be recommended to user C and vice versa.

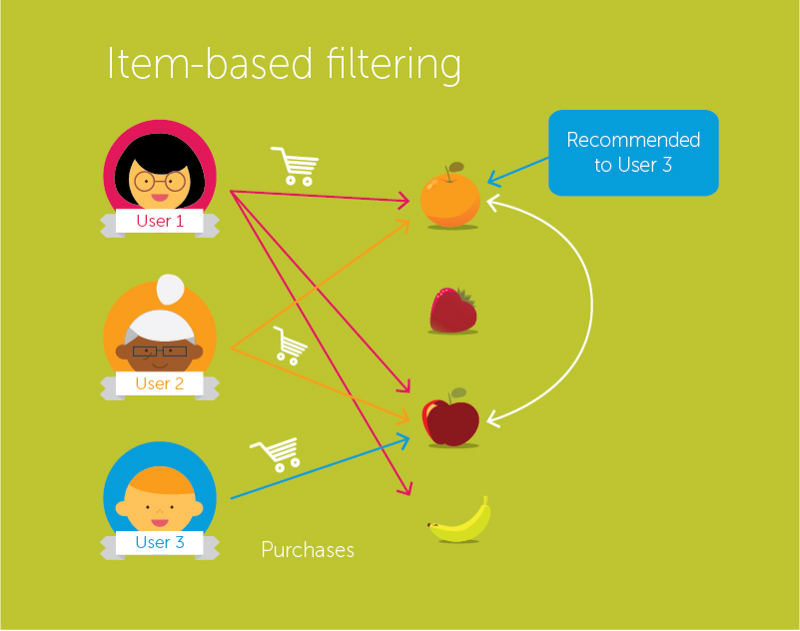
This algorithm is quite time consuming as it involves calculating the similarity for each user and then calculating prediction for each similarity score. One way of handling this problem is to select only a few users (neighbors) instead of all to make predictions, i.e. instead of making predictions for all similarity values, we choose only few similarity values. There are various ways to select the neighbors:

* Select a threshold similarity and choose all the users above that value
* Randomly select the users
* Arrange the neighbors in descending order of their similarity value and choose top-N users
* Use clustering for choosing neighbors

This algorithm is useful when the number of users is less. Its not effective when there are a large number of users as it will take a lot of time to compute the similarity between all user pairs. This leads us to item-item collaborative filtering, which is effective when the number of users is more than the items being recommended.

**Item-Item collaborative filtering**

In this algorithm, we compute the similarity between each pair of items.



*Source: Medium*

So in our case we will find the similarity between each movie pair and based on that, we will recommend similar movies which are liked by the users in the past. This algorithm works similar to user-user collaborative filtering with just a little change – instead of taking the weighted sum of ratings of “user-neighbors”, we take the weighted sum of ratings of “item-neighbors”. The prediction is given by:

https://s3-ap-south-1.amazonaws.com/av-blog-media/wp-content/uploads/2018/06/Screenshot-from-2018-06-27-16-02-19.png

Now we will find the similarity between items.

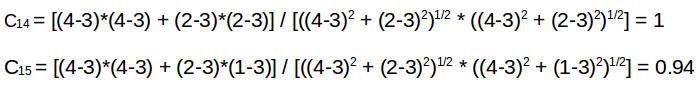
https://s3-ap-south-1.amazonaws.com/av-blog-media/wp-content/uploads/2018/05/Screenshot-from-2018-05-29-20-23-23.png

Now, as we have the similarity between each movie and the ratings, predictions are made and based on those predictions, similar movies are recommended. Let us understand it with an example.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| User/Movie | x1 | x2 | x3 | x4 | x5 |
| A | 4 | 1 | 2 | 4 | 4 |
| B | 2 | 4 | 4 | 2 | 1 |
| C | – | 1 | – | 3 | 4 |
| Mean Item Rating | 3 | 2 | 3 | 3 | 3 |

Here the mean item rating is the average of all the ratings given to a particular item (compare it with the table we saw in user-user filtering). Instead of finding the user-user similarity as we saw earlier, we find the item-item similarity.

To do this, first we need to find such users who have rated those items and based on the ratings, similarity between the items is calculated. Let us find the similarity between movies (x1, x4) and (x1, x5). Common users who have rated movies x1 and x4 are A and B while the users who have rated movies x1 and x5 are also A and B.



The similarity between movie x1 and x4 is more than the similarity between movie x1 and x5. So based on these similarity values, if any user searches for movie x1, they will be recommended movie x4 and vice versa. Before going further and implementing these concepts, there is a question which we must know the answer to – what will happen if a new user or a new item is added in the dataset? It is called a **Cold Start**. There can be two types of cold start:

1. Visitor Cold Start
2. Product Cold Start

Visitor Cold Start means that a new user is introduced in the dataset. Since there is no history of that user, the system does not know the preferences of that user. It becomes harder to recommend products to that user. So, how can we solve this problem? One basic approach could be to apply a popularity based strategy, i.e. recommend the most popular products. These can be determined by what has been popular recently overall or regionally. Once we know the preferences of the user, recommending products will be easier.

On the other hand, Product Cold Start means that a new product is launched in the market or added to the system. User action is most important to determine the value of any product. More the interaction a product receives, the easier it is for our model to recommend that product to the right user. We can make use of Content based filtering to solve this problem. The system first uses the content of the new product for recommendations and then eventually the user actions on that product.

Now let’s solidify our understanding of these concepts using a case study in Python. Get your machines ready because this is going to be fun!

**3. Case study in Python using the MovieLens Dataset**

We will work on the MovieLens dataset and build a model to recommend movies to the end users. This data has been collected by the GroupLens Research Project at the University of Minnesota. The dataset can be downloaded from [**here**](https://grouplens.org/datasets/movielens/100k/). This dataset consists of:

* 100,000 ratings (1-5) from 943 users on 1682 movies
* Demographic information of the users (age, gender, occupation, etc.)

First, we’ll import our standard libraries and read the dataset in Python.

import pandas as pd

%matplotlib inline

import matplotlib

import matplotlib.pyplot as plt

import numpy as np

# pass in column names for each CSV as the column name is not given in the file and read them using pandas.

# You can check the column names from the readme file

#Reading users file:

u\_cols = ['user\_id', 'age', 'sex', 'occupation', 'zip\_code']

users = pd.read\_csv('ml-100k/u.user', sep='|', names=u\_cols,encoding='latin-1')

#Reading ratings file:

r\_cols = ['user\_id', 'movie\_id', 'rating', 'unix\_timestamp']

ratings = pd.read\_csv('ml-100k/u.data', sep='\t', names=r\_cols,encoding='latin-1')

#Reading items file:

i\_cols = ['movie id', 'movie title' ,'release date','video release date', 'IMDb URL', 'unknown', 'Action', 'Adventure',

'Animation', 'Children\'s', 'Comedy', 'Crime', 'Documentary', 'Drama', 'Fantasy',

'Film-Noir', 'Horror', 'Musical', 'Mystery', 'Romance', 'Sci-Fi', 'Thriller', 'War', 'Western']

items = pd.read\_csv('ml-100k/u.item', sep='|', names=i\_cols,

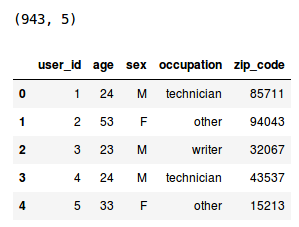
encoding='latin-1')

After loading the dataset, we should look at the content of each file (users, ratings, items).

* **Users**

print(users.shape)

users.head()

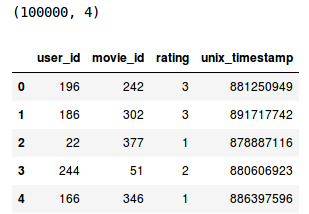


So, we have 943 users in the dataset and each user has 5 features, i.e. user\_ID, age, sex, occupation and zip\_code. Now let’s look at the ratings file.

* **Ratings**

print(ratings.shape)

ratings.head()

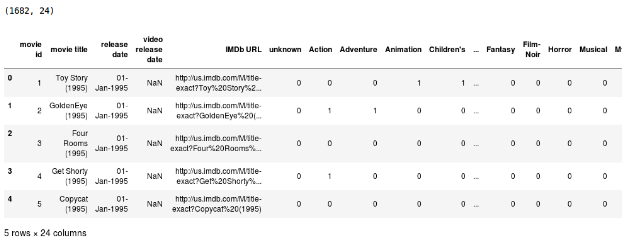


We have 100k ratings for different user and movie combinations. Now finally examine the items file.

* **Items**

print(items.shape)

items.head()



This dataset contains attributes of 1682 movies. There are 24 columns out of which last 19 columns specify the genre of a particular movie. These are binary columns, i.e., a value of 1 denotes that the movie belongs to that genre, and 0 otherwise.

The dataset has already been divided into train and test by GroupLens where the test data has 10 ratings for each user, i.e. 9,430 rows in total. We will read both these files into our Python environment.

r\_cols = ['user\_id', 'movie\_id', 'rating', 'unix\_timestamp']

ratings\_train = pd.read\_csv('ml-100k/ua.base', sep='\t', names=r\_cols, encoding='latin-1')

ratings\_test = pd.read\_csv('ml-100k/ua.test', sep='\t', names=r\_cols, encoding='latin-1')

ratings\_train.shape, ratings\_test.shape

https://s3-ap-south-1.amazonaws.com/av-blog-media/wp-content/uploads/2018/05/Screenshot-from-2018-05-29-20-31-48.png

It’s finally time to build our recommend engine!

**4. Building collaborative filtering model from scratch**

We will recommend movies based on user-user similarity and item-item similarity. For that, first we need to calculate the number of unique users and movies.

n\_users = ratings.user\_id.unique().shape[0]

n\_items = ratings.movie\_id.unique().shape[0]

Now, we will create a user-item matrix which can be used to calculate the similarity between users and items.

data\_matrix = np.zeros((n\_users, n\_items))

for line in ratings.itertuples():

data\_matrix[line[1]-1, line[2]-1] = line[3]

Now, we will calculate the similarity. We can use the *pairwise\_distance* function from *sklearn* to calculate the cosine similarity.

from sklearn.metrics.pairwise import pairwise\_distances

user\_similarity = pairwise\_distances(data\_matrix, metric='cosine')

item\_similarity = pairwise\_distances(data\_matrix.T, metric='cosine')

This gives us the item-item and user-user similarity in an array form. The next step is to make predictions based on these similarities. Let’s define a function to do just that.

def predict(ratings, similarity, type='user'):

if type == 'user':

mean\_user\_rating = ratings.mean(axis=1)

#We use np.newaxis so that mean\_user\_rating has same format as ratings

ratings\_diff = (ratings - mean\_user\_rating[:, np.newaxis])

pred = mean\_user\_rating[:, np.newaxis] + similarity.dot(ratings\_diff) / np.array([np.abs(similarity).sum(axis=1)]).T

elif type == 'item':

pred = ratings.dot(similarity) / np.array([np.abs(similarity).sum(axis=1)])

return pred

Finally, we will make predictions based on user similarity and item similarity.

user\_prediction = predict(data\_matrix, user\_similarity, type='user')

item\_prediction = predict(data\_matrix, item\_similarity, type='item')

As it turns out, we also have a library which generates all these recommendations automatically. Let us now learn how to create a recommendation engine using turicreate in Python. To get familiar with turicreate and to install it on your machine, refer[**here**](https://github.com/apple/turicreate/blob/master/README.md).

**5. Building a simple popularity and collaborative filtering model using Turicreate**

After installing turicreate, first let’s import it and read the train and test dataset in our environment. Since we will be using turicreate, we will need to convert the dataset in SFrames.

import turicreate

train\_data = turicreate.SFrame(ratings\_train)

test\_data = turicreate.Sframe(ratings\_test)

We have user behavior as well as attributes of the users and movies, so we can make content based as well as collaborative filtering algorithms. We will start with a simple popularity model and then build a collaborative filtering model.

First we’ll build a model which will recommend movies based on the most popular choices, i.e., a model where all the users receive the same recommendation(s). We will use the turicreate recommender function *popularity\_recommender* for this.

popularity\_model = turicreate.popularity\_recommender.create(train\_data, user\_id='user\_id', item\_id='movie\_id', target='rating')

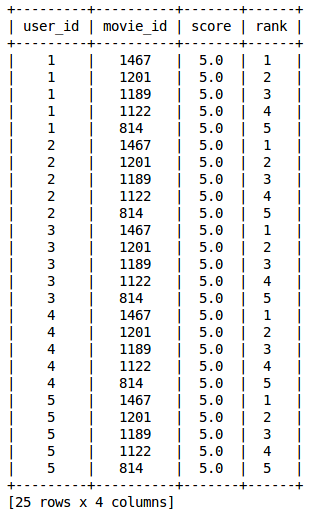
Various arguments which we have used are:

* **train\_data**: the SFrame which contains the required training data
* **user\_id**: the column name which represents each user ID
* **item\_id**: the column name which represents each item to be recommended (movie\_id)
* **target:** the column name representing scores/ratings given by the user

It’s prediction time! We will recommend the top 5 items for the first 5 users in our dataset.

popularity\_recomm = popularity\_model.recommend(users=[1,2,3,4,5],k=5)

popularity\_recomm.print\_rows(num\_rows=25)



Note that the recommendations for all users are the same – 1467, 1201, 1189, 1122, 814. And they’re all in the same order! This confirms that all the recommended movies have an average rating of 5, i.e. all the users who watched the movie gave it a top rating. Thus our popularity system works as expected.

After building a popularity model, we will now build a collaborative filtering model. Let’s train the item similarity model and make top 5 recommendations for the first 5 users.

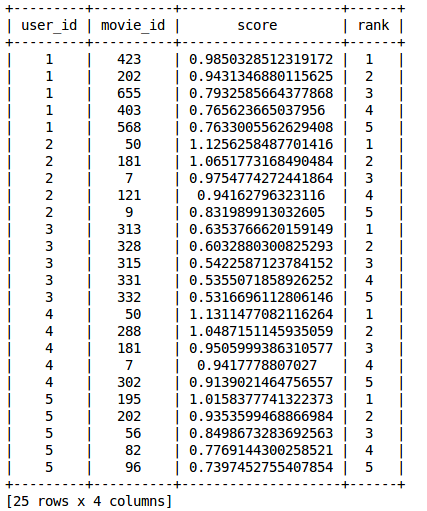
#Training the model

item\_sim\_model = turicreate.item\_similarity\_recommender.create(train\_data, user\_id='user\_id', item\_id='movie\_id', target='rating', similarity\_type='cosine')

#Making recommendations

item\_sim\_recomm = item\_sim\_model.recommend(users=[1,2,3,4,5],k=5)

item\_sim\_recomm.print\_rows(num\_rows=25)

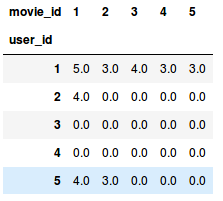


Here we can see that the recommendations (movie\_id) are different for each user. So personalization exists, i.e. for different users we have a different set of recommendations.

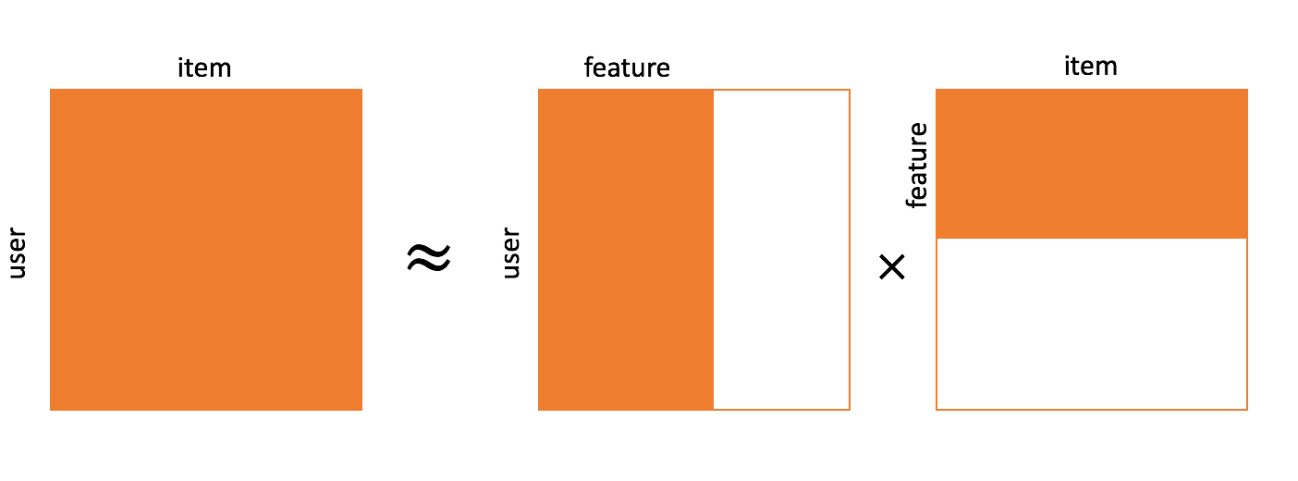
In this model, we do not have the ratings for each movie given by each user. We must find a way to predict all these missing ratings. For that, we have to find a set of features which can define how a user rates the movies. **These are called latent features**. We need to find a way to extract the most important latent features from the the existing features. Matrix factorization, covered in the next section, is one such technique which uses the lower dimension dense matrix and helps in extracting the important latent features.

**6. Introduction to matrix factorization**

Let’s understand matrix factorization with an example. Consider a user-movie ratings matrix (1-5) given by different users to different movies.



Here user\_id is the unique ID of different users and each movie is also assigned a unique ID. A rating of 0.0 represents that the user has not rated that particular movie (1 is the lowest rating a user can give). We want to predict these missing ratings. Using matrix factorization, we can find some latent features that can determine how a user rates a movie. We decompose the matrix into constituent parts in such a way that the product of these parts generates the original matrix.



Let us assume that we have to find *k* latent features. So we can divide our rating matrix R(MxN) into P(MxK) and Q(NxK) such that P x QT (here QT is the transpose of Q matrix) approximates the R matrix:

https://s3-ap-south-1.amazonaws.com/av-blog-media/wp-content/uploads/2018/05/Screenshot-from-2018-05-30-14-26-15.png, where:

* M is the total number of users
* N is the total number of movies
* K is the total latent features
* R is MxN user-movie rating matrix
* P is MxK user-feature affinity matrix which represents the association between users and features
* Q is NxK item-feature relevance matrix which represents the association between movies and features
* Σ is KxK diagonal feature weight matrix which represents the essential weights of features

Choosing the latent features through matrix factorization removes the noise from the data. How? Well, it removes the feature(s) which does not determine how a user rates a movie. Now to get the rating *rui* for a movie *qik* rated by a user *puk* across all the latent features *k*, we can calculate the dot product of the 2 vectors and add them to get the ratings based on all the latent features.

https://s3-ap-south-1.amazonaws.com/av-blog-media/wp-content/uploads/2018/05/Screenshot-from-2018-05-30-14-27-02.png

This is how matrix factorization gives us the ratings for the movies which have not been rated by the users. But how can we add new data to our user-movie rating matrix, i.e. if a new user joins and rates a movie, how will we add this data to our pre-existing matrix?

Let me make it easier for you through the matrix factorization method. If a new user joins the system, there will be no change in the diagonal feature weight matrix Σ, as well as the item-feature relevance matrix Q. The only change will occur in the user-feature affinity matrix P. We can apply some matrix multiplication methods to do that.

We have,

https://s3-ap-south-1.amazonaws.com/av-blog-media/wp-content/uploads/2018/05/Screenshot-from-2018-05-30-14-26-15.png

Let’s multiply with Q on both sides.

https://s3-ap-south-1.amazonaws.com/av-blog-media/wp-content/uploads/2018/05/Screenshot-from-2018-05-30-14-28-43.png

Now, we have

https://s3-ap-south-1.amazonaws.com/av-blog-media/wp-content/uploads/2018/05/Screenshot-from-2018-05-30-14-29-58.png

So,

https://s3-ap-south-1.amazonaws.com/av-blog-media/wp-content/uploads/2018/05/Screenshot-from-2018-05-30-14-30-36.png

Simplifying it further, we can get the P matrix:

https://s3-ap-south-1.amazonaws.com/av-blog-media/wp-content/uploads/2018/05/Screenshot-from-2018-05-30-14-31-01.png

This is the updated user-feature affinity matrix. Similarly, if a new movie is added to the system, we can follow similar steps to get the updated item-feature relevance matrix Q.

Remember, we decomposed R matrix into P and Q. But how do we decide which P and Q matrix will approximate the R matrix? We can use the gradient descent algorithm for doing this. The objective here is to minimize the squared error between the actual rating and the one estimated using P and Q. The squared error is given by:

https://s3-ap-south-1.amazonaws.com/av-blog-media/wp-content/uploads/2018/05/Screenshot-from-2018-05-30-14-49-11.png

Here,

* *eui* is the error
* *rui* is the actual rating given by user u to the movie i
* *řui* is the predicted rating by user u for the movie i

Our aim was to decide the p and q value in such a way that this error is minimized. We need to update the p and q values so as to get the optimized values of these matrices which will give the least error. Now we will define an update rule for *puk* and *qki*. The update rule in gradient descent is defined by the gradient of the error to be minimized.

https://s3-ap-south-1.amazonaws.com/av-blog-media/wp-content/uploads/2018/05/Screenshot-from-2018-05-30-14-40-36.png

https://s3-ap-south-1.amazonaws.com/av-blog-media/wp-content/uploads/2018/05/Screenshot-from-2018-05-30-14-41-15.png

As we now have the gradients, we can apply the update rule for *puk* and *qki*.

https://s3-ap-south-1.amazonaws.com/av-blog-media/wp-content/uploads/2018/05/Screenshot-from-2018-05-30-14-46-22.png

https://s3-ap-south-1.amazonaws.com/av-blog-media/wp-content/uploads/2018/05/Screenshot-from-2018-05-30-14-47-01.png

Here α is the learning rate which decides the size of each update. The above updates can be repeated until the error is minimized. Once that’s done, we get the optimal P and Q matrix which can be used to predict the ratings. Let us quickly recap how this algorithm works and then we will build the recommendation engine to predict the ratings for the unrated movies.

Below is how matrix factorization works for predicting ratings:

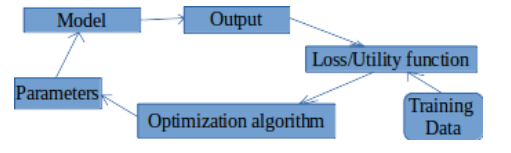
# for f = 1,2,....,k :

# for rui ε R :

# predict rui

# update puk and qki

So based on each latent feature, all the missing ratings in the R matrix will be filled using the predicted *rui*value. Then *puk* and *qki* are updated using gradient descent and their optimal value is obtained. It can be visualized as shown below:



Now that we have understood the inner workings of this algorithm, we’ll take an example and see how a matrix is factorized into its constituents.

Consider a 2 X 3 matrix, A*2X3* as shown below:

https://s3-ap-south-1.amazonaws.com/av-blog-media/wp-content/uploads/2018/05/Screenshot-from-2018-05-30-14-58-31.png

Here we have 2 users and their corresponding ratings for 3 movies. Now, we will decompose this matrix into sub parts, such that:

https://s3-ap-south-1.amazonaws.com/av-blog-media/wp-content/uploads/2018/05/Screenshot-from-2018-05-30-14-59-09.png

The eigenvalues of AAT will give us the P matrix and the eigenvalues of ATA will give us the Q matrix. Σ is the square root of the eigenvalues from AAT or ATA.

Calculate the eigenvalues for AAT.

https://s3-ap-south-1.amazonaws.com/av-blog-media/wp-content/uploads/2018/05/Screenshot-from-2018-05-30-15-04-48.png

https://s3-ap-south-1.amazonaws.com/av-blog-media/wp-content/uploads/2018/05/Screenshot-from-2018-05-30-15-05-44.png

So, the eigenvalues of AAT are 25, 9. Similarly, we can calculate the eigenvalues of ATA. These values will be 25, 9, 0. Now we have to calculate the corresponding eigenvectors for AAT and ATA.

For λ = 25, we have:

https://s3-ap-south-1.amazonaws.com/av-blog-media/wp-content/uploads/2018/06/Screenshot-from-2018-06-27-17-42-32.png

It can be [**row reduced**](http://www.sparknotes.com/math/algebra2/matrices/section4/) to:

https://s3-ap-south-1.amazonaws.com/av-blog-media/wp-content/uploads/2018/05/Screenshot-from-2018-05-30-15-10-32.png

A unit-length vector in the kernel of that matrix is:

https://s3-ap-south-1.amazonaws.com/av-blog-media/wp-content/uploads/2018/05/Screenshot-from-2018-05-30-15-14-12.png

Similarly, for λ = 9 we have:

https://s3-ap-south-1.amazonaws.com/av-blog-media/wp-content/uploads/2018/06/Screenshot-from-2018-06-27-18-08-42.png

It can be row reduced to:

https://s3-ap-south-1.amazonaws.com/av-blog-media/wp-content/uploads/2018/05/Screenshot-from-2018-05-30-15-19-00.png

A unit-length vector in the kernel of that matrix is:

https://s3-ap-south-1.amazonaws.com/av-blog-media/wp-content/uploads/2018/05/Screenshot-from-2018-05-30-15-21-49.png

For the last eigenvector, we could find a unit vector perpendicular to q1 and q2. So,

https://s3-ap-south-1.amazonaws.com/av-blog-media/wp-content/uploads/2018/05/Screenshot-from-2018-05-30-15-23-09.png

**Σ***2X3* matrix is the square root of eigenvalues of AAT or ATA, i.e. 25 and 9.

https://s3-ap-south-1.amazonaws.com/av-blog-media/wp-content/uploads/2018/05/Screenshot-from-2018-05-30-15-25-18.png

Finally, we can compute P*2X2* by the formula σpi = Aqi, or pi = 1/σ(Aqi). This gives:

https://s3-ap-south-1.amazonaws.com/av-blog-media/wp-content/uploads/2018/05/Screenshot-from-2018-05-30-15-27-49.png

So, the decomposed form of A matrix is given by:

https://s3-ap-south-1.amazonaws.com/av-blog-media/wp-content/uploads/2018/05/Screenshot-from-2018-05-30-15-29-21.png  https://s3-ap-south-1.amazonaws.com/av-blog-media/wp-content/uploads/2018/05/Screenshot-from-2018-05-30-15-30-30.png  https://s3-ap-south-1.amazonaws.com/av-blog-media/wp-content/uploads/2018/05/Screenshot-from-2018-05-30-15-35-38.png

Since we have the P and Q matrix, we can use the gradient descent approach to get their optimized versions. Let us build our recommendation engine using matrix factorization.

**7. Building a recommendation engine using matrix factorization**

Let us define a function to predict the ratings given by the user to all the movies which are not rated by him/her.

class MF():

# Initializing the user-movie rating matrix, no. of latent features, alpha and beta.

def \_\_init\_\_(self, R, K, alpha, beta, iterations):

self.R = R

self.num\_users, self.num\_items = R.shape

self.K = K

self.alpha = alpha

self.beta = beta

self.iterations = iterations

# Initializing user-feature and movie-feature matrix

def train(self):

self.P = np.random.normal(scale=1./self.K, size=(self.num\_users, self.K))

self.Q = np.random.normal(scale=1./self.K, size=(self.num\_items, self.K))

# Initializing the bias terms

self.b\_u = np.zeros(self.num\_users)

self.b\_i = np.zeros(self.num\_items)

self.b = np.mean(self.R[np.where(self.R != 0)])

# List of training samples

self.samples = [

(i, j, self.R[i, j])

for i in range(self.num\_users)

for j in range(self.num\_items)

if self.R[i, j] > 0

]

# Stochastic gradient descent for given number of iterations

training\_process = []

for i in range(self.iterations):

np.random.shuffle(self.samples)

self.sgd()

mse = self.mse()

training\_process.append((i, mse))

if (i+1) % 20 == 0:

print("Iteration: %d ; error = %.4f" % (i+1, mse))

return training\_process

# Computing total mean squared error

def mse(self):

xs, ys = self.R.nonzero()

predicted = self.full\_matrix()

error = 0

for x, y in zip(xs, ys):

error += pow(self.R[x, y] - predicted[x, y], 2)

return np.sqrt(error)

# Stochastic gradient descent to get optimized P and Q matrix

def sgd(self):

for i, j, r in self.samples:

prediction = self.get\_rating(i, j)

e = (r - prediction)

self.b\_u[i] += self.alpha \* (e - self.beta \* self.b\_u[i])

self.b\_i[j] += self.alpha \* (e - self.beta \* self.b\_i[j])

self.P[i, :] += self.alpha \* (e \* self.Q[j, :] - self.beta \* self.P[i,:])

self.Q[j, :] += self.alpha \* (e \* self.P[i, :] - self.beta \* self.Q[j,:])

# Ratings for user i and moive j

def get\_rating(self, i, j):

prediction = self.b + self.b\_u[i] + self.b\_i[j] + self.P[i, :].dot(self.Q[j, :].T)

return prediction

# Full user-movie rating matrix

def full\_matrix(self):

return mf.b + mf.b\_u[:,np.newaxis] + mf.b\_i[np.newaxis:,] + mf.P.dot(mf.Q.T)

Now we have a function that can predict the ratings. The input for this function are:

* R – The user-movie rating matrix
* K – Number of latent features
* alpha – Learning rate for stochastic gradient descent
* beta – Regularization parameter for bias
* iterations – Number of iterations to perform stochastic gradient descent

We have to convert the user item ratings to matrix form. It can be done using the pivot function in python.

R= np.array(ratings.pivot(index = 'user\_id', columns ='movie\_id', values = 'rating').fillna(0))

fillna(0) will fill all the missing ratings with 0. Now we have the R matrix. We can initialize the number of latent features, but the number of these features must be less than or equal to the number of original features.

Now let us predict all the missing ratings. Let’s take K=20, alpha=0.001, beta=0.01 and iterations=100.

mf = MF(R, K=20, alpha=0.001, beta=0.01, iterations=100)

training\_process = mf.train()

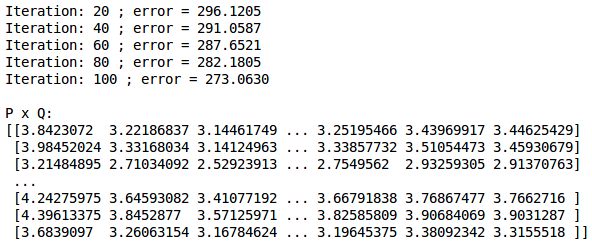
print()

print("P x Q:")

print(mf.full\_matrix())

print()

This will give us the error value corresponding to every 20th iteration and finally the complete user-movie rating matrix. The output looks like this:



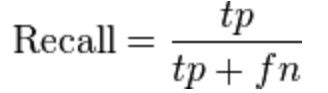
We have created our recommendation engine. Let’s focus on how to evaluate a recommendation engine in the next section.

**8. Evaluation metrics for recommendation engines**

For evaluating recommendation engines, we can use the following metrics

**8.1 Recall:**

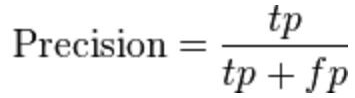
* What proportion of items that a user likes were actually recommended
* It is given by:



* + Here *tp* represents the number of items recommended to a user that he/she likes and *tp*+*fn*represents the total items that a user likes
  + If a user likes 5 items and the recommendation engine decided to show 3 of them, then the recall will be 0.6
  + Larger the recall, better are the recommendations

**8.2 Precision:**

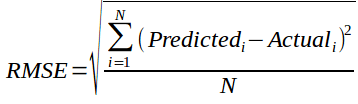
* + Out of all the recommended items, how many did the user actually like?
  + It is given by:



* + Here *tp* represents the number of items recommended to a user that he/she likes and *tp*+*fp*represents the total items recommended to a user
  + If 5 items were recommended to the user out of which he liked 4, then precision will be 0.8
  + Larger the precision, better the recommendations
  + But consider this case: If we simply recommend all the items, they will definitely cover the items which the user likes. So we have 100% recall! But think about precision for a second. If we recommend say 1000 items and user likes only 10 of them, then precision is 0.1%. This is really low. So, our aim should be to maximize both precision and recall.

**8.3 RMSE (Root Mean Squared Error):**

* It measures the error in the predicted ratings:



* + Here, Predicted is the rating predicted by the model and Actual is the original rating
  + If a user has given a rating of 5 to a movie and we predicted the rating as 4, then RMSE is 1
  + Lesser the RMSE value, better the recommendations

The above metrics tell us how accurate our recommendations are but they do not focus on the order of recommendations, i.e. they do not focus on which product to recommend first and what follows after that. We need some metric that also considers the order of the products recommended. So, let’s look at some of the **ranking metrics**:

**8.4 Mean Reciprocal Rank:**

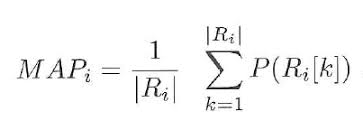
* Evaluates the list of recommendations

https://s3-ap-south-1.amazonaws.com/av-blog-media/wp-content/uploads/2018/05/Screenshot-from-2018-05-31-18-37-20.png

* + Suppose we have recommended 3 movies to a user, say A, B, C in the given order, but the user only liked movie C. As the rank of movie C is 3, the reciprocal rank will be 1/3
  + Larger the mean reciprocal rank, better the recommendations

**8.5 MAP at k (Mean Average Precision at cutoff k):**

* Precision and Recall don’t care about ordering in the recommendations
* Precision at cutoff k is the precision calculated by considering only the subset of your recommendations from rank 1 through k



* + Suppose we have made three recommendations [0, 1, 1]. Here 0 means the recommendation is not correct while 1 means that the recommendation is correct. Then the precision at k will be [0, 1/2, 2/3], and the average precision will be (1/3)\*(0+1/2+2/3) = 0.38
  + Larger the mean average precision, more correct will be the recommendations

**8.6 NDCG (Normalized Discounted Cumulative Gain):**

* The main difference between MAP and NDCG is that MAP assumes that an item is either of interest (or not), while NDCG gives the relevance score
* Let us understand it with an example: suppose out of 10 movies – A to J, we can recommend the first five movies, i.e. A, B, C, D and E while we must not recommend the other 5 movies, i.e., F, G, H, I and J. The recommendation was [A,B,C,D]. So the NDCG in this case will be 1 as the recommended products are relevant for the user
* Higher the NDCG value, better the recommendations

**9. What else can be tried?**

Up to this point we have learnt what is a recommendation engine, its different types and their workings. Both content-based filtering and collaborative filtering algorithms have their strengths and weaknesses.

In some domains, generating a useful description of the content can be very difficult. A content-based filtering model will not select items if the user’s previous behavior does not provide evidence for this. Additional techniques have to be used so that the system can make suggestions outside the scope of what the user has already shown an interest in.

A collaborative filtering model doesn’t have these shortcomings. Because there is no need for a description of the items being recommended, the system can deal with any kind of information. Furthermore, it can recommend products which the user has not shown an interest in previously. But, collaborative filtering cannot provide recommendations for new items if there are no user ratings upon which to base a prediction. Even if users start rating the item, it will take some time before the item has received enough ratings in order to make accurate recommendations.

A system that combines content-based filtering and collaborative filtering could potentially take advantage from both the representation of the content as well as the similarities among users. One approach to combine collaborative and content-based filtering is to make predictions based on a weighted average of the content-based recommendations and the collaborative recommendations. Various means of doing so are:

* **Combining item scores**
  + In this approach, we combine the ratings obtained from both the filtering methods. The simplest way is to take the average of the ratings
  + Suppose one method suggested a rating of 4 for a movie while the other suggested a rating of 5 for the same movie. So the final recommendation will be the average of both ratings, i.e. 4.5
  + We can assign different weights to different methods as well
* **Combining item ranks:**
  + Suppose collaborative filtering recommended 5 movies A, B, C, D and E in the following order: A, B, C, D, E while content based filtering recommended them in the following order: B, D, A, C, E
  + The rank for the movies will be:

Collaborative filtering

|  |  |
| --- | --- |
| Movie | Rank |
| A | 1 |
| B | 0.8 |
| C | 0.6 |
| D | 0.4 |
| E | 0.2 |

Content Based Filtering:

|  |  |
| --- | --- |
| Movie | Rank |
| B | 1 |
| D | 0.8 |
| A | 0.6 |
| C | 0.4 |
| E | 0.2 |

So, **a hybrid recommender engine will combine these ranks and make final recommendations based on the combined rankings**. The combined rank will be:

|  |  |
| --- | --- |
| Movie | New Rank |
| A | 1+0.6 = 1.6 |
| B | 0.8+1 = 1.8 |
| C | 0.6+0.4 = 1 |
| D | 0.4+0.8 = 1.2 |
| E | 0.2+0.2 = 0.4 |

The recommendations will be made based on these rankings. So, the final recommendations will look like this: B, A, D, C, E.

In this way, two or more techniques can be combined to build a hybrid recommendation engine and to improve their overall recommendation accuracy and power.

**End Notes**

This was a very comprehensive article on recommendation engines. This tutorial should be good enough to get you started with this topic. We not only covered basic recommendation techniques but also saw how to implement some of the more advanced techniques available in the industry today.

We also covered some key facts associated with each technique. As somebody who wants to learn how to make a recommendation engine, I’d advise you to learn the techniques discussed in this tutorial and later implement them in your models.

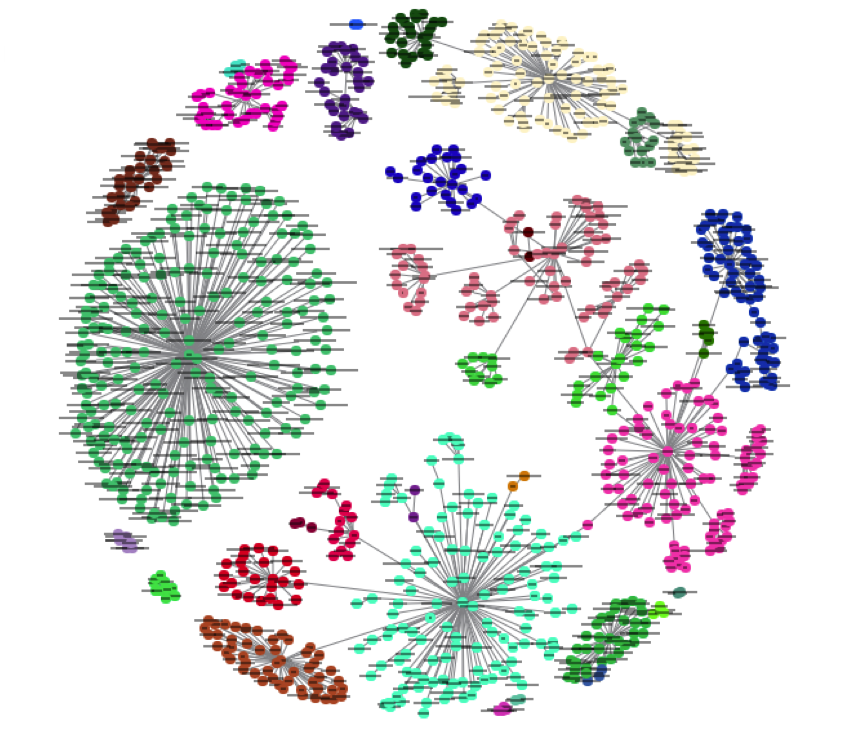
# A Practical Introduction to K-Nearest Neighbors Algorithm for Regression (with Python code)

[**AISHWARYA SINGH**](https://www.analyticsvidhya.com/blog/author/aishwaryasingh/)**, AUGUST 22, 2018**

## Introduction

Out of all the machine learning algorithms I have come across, KNN has easily been the simplest to pick up. Despite it’s simplicity, it has proven to be incredibly effective at certain tasks (as you will see in this article).

And even better? It can be used for both classification and regression problems! It’s far more popularly used for classification problems, however. I have seldom seen KNN being implemented on any regression task. **My aim here is to illustrate and emphasize how KNN can be equally effective when the target variable is continuous in nature.**



In this article, we will first understand the intuition behind KNN algorithms, look at the different ways to calculate distances between points, and then finally implement the algorithm in Python on the Big Mart Sales dataset. Let’s go!

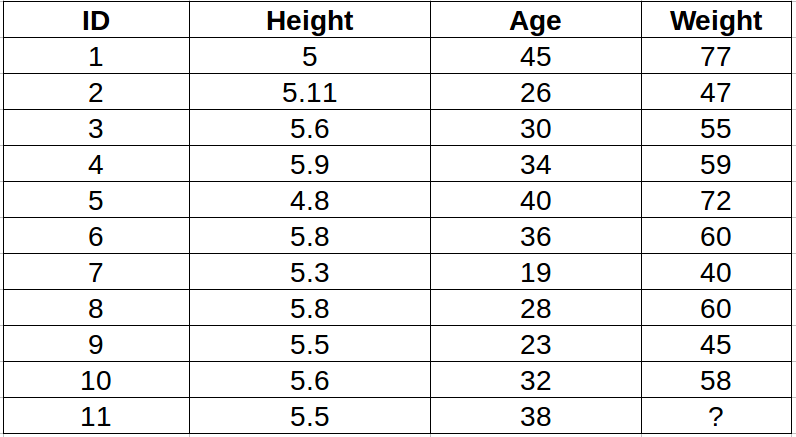
## Table of contents

1. A simple example to understand the intuition behind KNN
2. How does the KNN algorithm work?
3. Methods of calculating distance between points
4. How to choose the k factor?
5. Working on a dataset
6. Additional resources

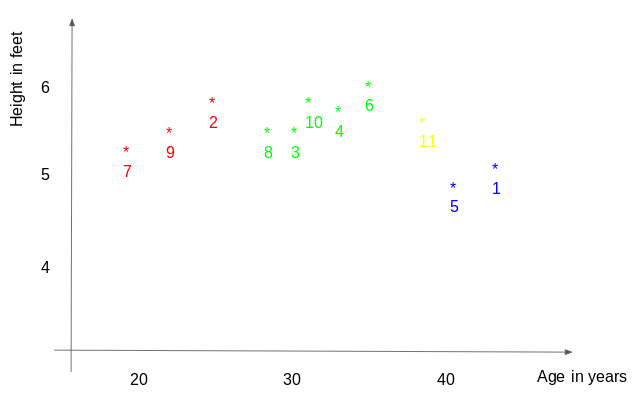
## 1. A simple example to understand the intuition behind KNN

Let us start with a simple example. Consider the following table – it consists of the height, age and weight (target) value for 10 people. As you can see, the weight value of ID11 is missing. We need to predict the weight of this person based on their height and age.

*Note: The data in this table does not represent actual values. It is merely used as an example to explain this concept.*



For a clearer understanding of this, below is the plot of height versus age from the above table:



In the above graph, the y-axis represents the height of a person (in feet) and the x-axis represents the age (in years). The points are numbered according to the ID values. The yellow point (ID 11) is our test point.

If I ask you to identify the weight of ID11 based on the plot, what would be your answer? You would likely say that since ID11 is **closer** to points 5 and 1, so it must  have a weight similar to these IDs, probably between 72-77 kgs (weights of ID1 and ID5 from the table). That actually makes sense, but how do you think the algorithm predicts the values? We will find that out in this article.

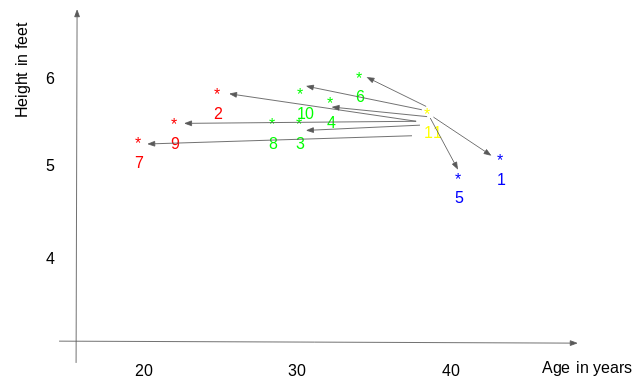
## 2. How does the KNN algorithm work?

As we saw above, KNN can be used for both classification and regression problems. The algorithm uses ‘**feature similarity**’ to predict values of any new data points. This means that the new point is assigned a value based on how closely it resembles the points in the training set. From our example, we know that ID11 has height and age similar to ID1 and ID5, so the weight would also approximately be the same.

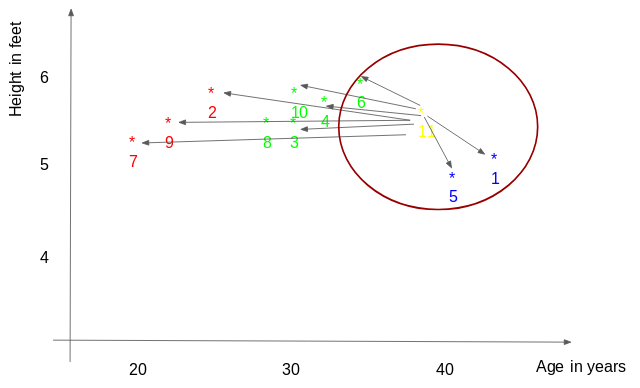
Had it been a classification problem, we would have taken the mode as the final prediction. In this case, we have two values of weight – 72 and 77. Any guesses how the final value will be calculated? The average of the values is taken to be the final prediction.

Below is a stepwise explanation of the algorithm:

1. First, the distance between the new point and each training point is calculated.



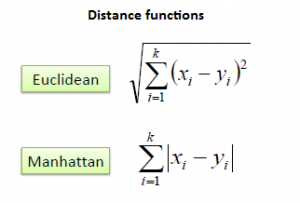
2. The closest k data points are selected (based on the distance). In this example, points 1, 5, 6 will be selected if value of k is 3. We will further explore the method to select the right value of k later in this article.

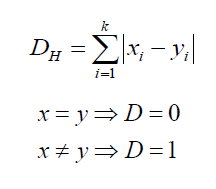
3. The average of these data points is the final prediction for the new point. Here, we have weight of ID11 = (77+72+60)/3 = 69.66 kg.

In the next few sections we will discuss each of these three steps in detail.

## 3. Methods of calculating distance between points

The **first step** is to calculate the distance between the new point and each training point. There are various methods for calculating this distance, of which the most commonly known methods are – Euclidian, Manhattan (for continuous) and Hamming distance (for categorical).

1. **Euclidean Distance:** Euclidean distance is calculated as the square root of the sum of the squared differences between a new point (x) and an existing point (y).
2. **Manhattan Distance** : This is the distance between real vectors using the sum of their absolute difference.
3. **Hamming Distance**: It is used for categorical variables. If the value (x) and the value (y) are same, the distance D will be equal to 0 . Otherwise D=1.

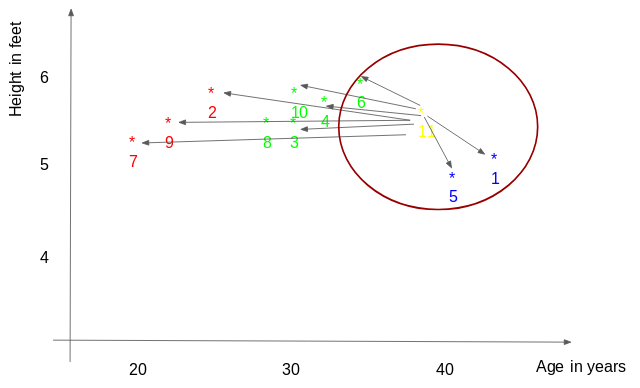


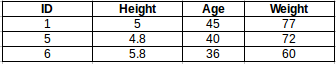
Once the distance of a new observation from the points in our training set has been measured, the next step is to pick the closest points. The number of points to be considered is defined by the value of k.

## 4. How to choose the k factor?

The **second step** is to select the k value. This determines the number of neighbors we look at when we assign a value to any new observation.

In our example, for a value k = 3, the closest points are ID1, ID5 and ID6.



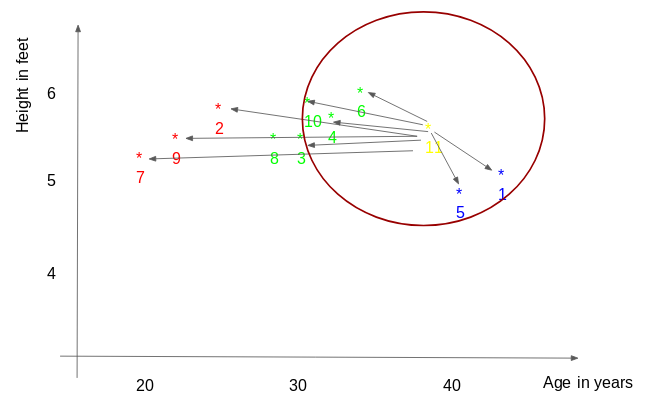


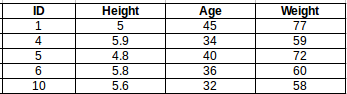
The prediction of weight for ID11 will be:

ID11 = (77+72+60)/3

ID11 = 69.66 kg

For the value of k=5, the closest point will be ID1, ID4, ID5, ID6, ID10.





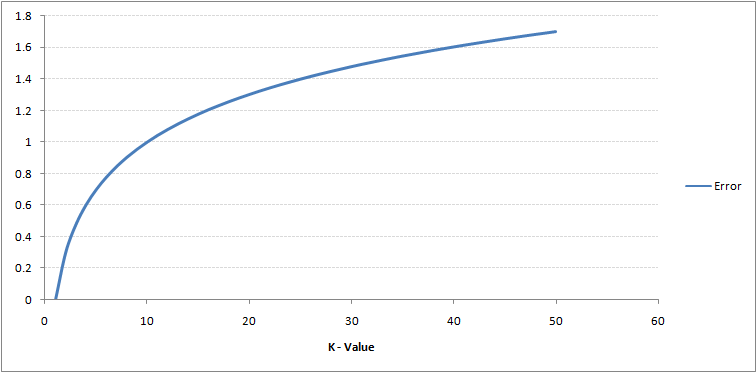
The prediction for ID11 will be :

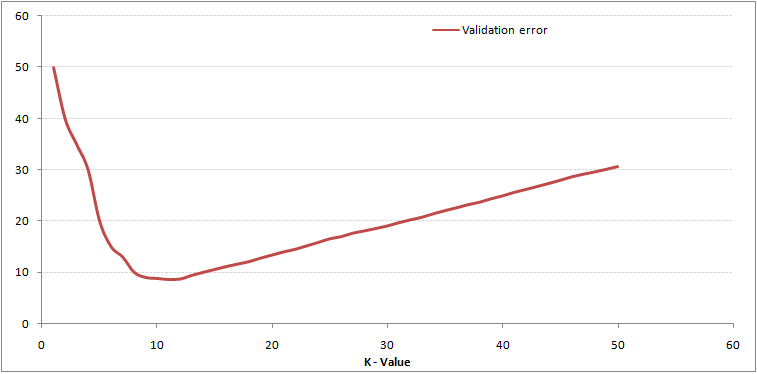
ID 11 =  (77+59+72+60+58)/5

ID 11 = 65.2 kg

We notice that based on the k value, the final result tends to change. Then how can we figure out the optimum value of k? Let us decide it based on the error calculation for our train and validation set (after all, minimizing the error is our final goal!).

Have a look at the below graphs for training error and validation error for different values of k.





For a very low value of k (suppose k=1), the model overfits on the training data, which leads to a high error rate on the validation set. On the other hand, for a high value of k, the model performs poorly on both train and validation set. If you observe closely, the validation error curve reaches a minima at a value of k = 9. This value of k is the optimum value of the model (it will vary for different datasets). This curve is known as an ‘**elbow curve**‘ (because it has a shape like an elbow) and is usually used to determine the k value.

You can also use the grid search technique to find the best k value. We will implement this in the next section.

## 5. Work on a dataset (Python codes)

By now you must have a clear understanding of the algorithm. If you have any questions regarding the same, please use the comments section below and I will be happy to answer them. We will now go ahead and implement the algorithm on a dataset. I have used the Big Mart sales dataset to show the implementation and you can download it from [this link](https://datahack.analyticsvidhya.com/contest/practice-problem-big-mart-sales-iii/).

**1. Read the file**

import pandas as pd

df = pd.read\_csv('train.csv')

df.head()

**2. Impute missing values**

df.isnull().sum()

#missing values in Item\_weight and Outlet\_size needs to be imputed

mean = df['Item\_Weight'].mean() #imputing item\_weight with mean

df['Item\_Weight'].fillna(mean, inplace =True)

mode = df['Outlet\_Size'].mode() #imputing outlet size with mode

df['Outlet\_Size'].fillna(mode[0], inplace =True)

**3. Deal with categorical variables and drop the id columns**

df.drop(['Item\_Identifier', 'Outlet\_Identifier'], axis=1, inplace=True)

df = pd.get\_dummies(df)

**4. Create train and test set**

from sklearn.model\_selection import train\_test\_split

train , test = train\_test\_split(df, test\_size = 0.3)

x\_train = train.drop('Item\_Outlet\_Sales', axis=1)

y\_train = train['Item\_Outlet\_Sales']

x\_test = test.drop('Item\_Outlet\_Sales', axis = 1)

y\_test = test['Item\_Outlet\_Sales']

**5. Preprocessing – Scaling the features**

from sklearn.preprocessing import MinMaxScaler

scaler = MinMaxScaler(feature\_range=(0, 1))

x\_train\_scaled = scaler.fit\_transform(x\_train)

x\_train = pd.DataFrame(x\_train\_scaled)

x\_test\_scaled = scaler.fit\_transform(x\_test)

x\_test = pd.DataFrame(x\_test\_scaled)

**6. Let us have a look at the error rate for different k values**

#import required packages

from sklearn import neighbors

from sklearn.metrics import mean\_squared\_error

from math import sqrt

import matplotlib.pyplot as plt

%matplotlib inline

rmse\_val = [] #to store rmse values for different k

for K in range(20):

K = K+1

model = neighbors.KNeighborsRegressor(n\_neighbors = K)

model.fit(x\_train, y\_train) #fit the model

pred=model.predict(x\_test) #make prediction on test set

error = sqrt(mean\_squared\_error(y\_test,pred)) #calculate rmse

rmse\_val.append(error) #store rmse values

print('RMSE value for k= ' , K , 'is:', error)

Output :

RMSE value for k = 1 is: 1579.8352322344945

RMSE value for k = 2 is: 1362.7748806138618

RMSE value for k = 3 is: 1278.868577489459

RMSE value for k = 4 is: 1249.338516122638

RMSE value for k = 5 is: 1235.4514224035129

RMSE value for k = 6 is: 1233.2711649472913

RMSE value for k = 7 is: 1219.0633086651026

RMSE value for k = 8 is: 1222.244674933665

RMSE value for k = 9 is: 1219.5895059285074

RMSE value for k = 10 is: 1225.106137547365

RMSE value for k = 11 is: 1229.540283771085

RMSE value for k = 12 is: 1239.1504407152086

RMSE value for k = 13 is: 1242.3726040709887

RMSE value for k = 14 is: 1251.505810196545

RMSE value for k = 15 is: 1253.190119191363

RMSE value for k = 16 is: 1258.802262564038

RMSE value for k = 17 is: 1260.884931441893

RMSE value for k = 18 is: 1265.5133661294733

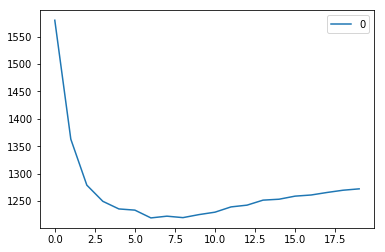
RMSE value for k = 19 is: 1269.619416217394

RMSE value for k = 20 is: 1272.10881411344

#plotting the rmse values against k values

curve = pd.DataFrame(rmse\_val) #elbow curve

curve.plot()



As we discussed, when we take k=1, we get a very high RMSE value. The RMSE value decreases as we increase the k value. At k= 7, the RMSE is approximately 1219.06, and shoots up on further increasing the k value. We can safely say that k=7 will give us the best result in this case.

These are the predictions using our training dataset. Let us now predict the values for test dataset and make a submission.

**7. Predictions on the test dataset**

#reading test and submission files

test = pd.read\_csv('test.csv')

submission = pd.read\_csv('SampleSubmission.csv')

submission['Item\_Identifier'] = test['Item\_Identifier']

submission['Outlet\_Identifier'] = test['Outlet\_Identifier']

#preprocessing test dataset

test.drop(['Item\_Identifier', 'Outlet\_Identifier'], axis=1, inplace=True)

test['Item\_Weight'].fillna(mean, inplace =True)

test = pd.get\_dummies(test)

test\_scaled = scaler.fit\_transform(test)

test = pd.DataFrame(test\_scaled)

#predicting on the test set and creating submission file

predict = model.predict(test)

submission['Item\_Outlet\_Sales'] = predict

submission.to\_csv('submit\_file.csv',index=False)

**On submitting this file, I get an RMSE of 1279.5159651297.**

**8. Implementing GridsearchCV**

For deciding the value of k, plotting the elbow curve every time is be a cumbersome and tedious process. **You can simply use gridsearch to find the best value.**

from sklearn.model\_selection import GridSearchCV

params = {'n\_neighbors':[2,3,4,5,6,7,8,9]}

knn = neighbors.KNeighborsRegressor()

model = GridSearchCV(knn, params, cv=5)

model.fit(x\_train,y\_train)

model.best\_params\_

Output :

{'n\_neighbors': 7}

## 6. End Notes and additional resources

In this article, we covered the workings of the KNN algorithm and its implementation in Python. It’s one of the most basic, yet effective machine learning techniques. For KNN implementation in R, you can go through this article : [kNN Algorithm using R](https://www.analyticsvidhya.com/blog/2015/08/learning-concept-knn-algorithms-programming/" \t "_blank).

In this article, we used the KNN model directly from the sklearn library. You can also implement KNN from scratch (I recommend this!), which is covered in the this article: [KNN simplified](https://www.analyticsvidhya.com/blog/2018/03/introduction-k-neighbours-algorithm-clustering/).

If you think you know KNN well and have a solid grasp on the technique, test your skills in this MCQ quiz: [30 questions on kNN Algorithm](https://www.analyticsvidhya.com/blog/2017/09/30-questions-test-k-nearest-neighbors-algorithm/). Good luck!

# 6 Easy Steps to Learn Naive Bayes Algorithm (with codes in Python and R)

[**SUNIL RAY**](https://www.analyticsvidhya.com/blog/author/sunil-ray/)**, SEPTEMBER 11, 2017**

* **Note: This article was originally published on Sep 13th, 2015 and updated on Sept 11th, 2017**

## Introduction

Here’s a situation you’ve got into:

You are working on a classification problem and you have generated your set of hypothesis, created features and discussed the importance of variables. Within an hour, stakeholders want to see the first cut of the model.

What will you do? You have hunderds of thousands of data points and quite a few variables in your training data set. In such situation, if I were at your place, I would have used ‘**Naive Bayes**‘, which can be extremely fast relative to other classification algorithms. It works on Bayes theorem of probability to predict the class of unknown data set.

In this article, I’ll explain the basics of this algorithm, so that next time when you come across large data sets, you can bring this algorithm to action. In addition, if you are a [newbie in Python or R](https://www.analyticsvidhya.com/learning-paths-data-science-business-analytics-business-intelligence-big-data/learning-path-data-science-python/), you should be overwhelmed by the presence of available codes in this article.

## Table of Contents

1. What is Naive Bayes algorithm?
2. How Naive Bayes Algorithms works?
3. What are the Pros and Cons of using Naive Bayes?
4. 4 Applications of Naive Bayes Algorithm
5. Steps to build a basic Naive Bayes Model in Python
6. Tips to improve the power of Naive Bayes Model

## What is Naive Bayes algorithm?

It is a classification technique based on [Bayes’ Theorem](https://en.wikipedia.org/wiki/Bayes%27_theorem) with an assumption of independence among predictors. In simple terms, a Naive Bayes classifier assumes that the presence of a particular feature in a class is unrelated to the presence of any other feature. For example, a fruit may be considered to be an apple if it is red, round, and about 3 inches in diameter. Even if these features depend on each other or upon the existence of the other features, all of these properties independently contribute to the probability that this fruit is an apple and that is why it is known as ‘Naive’.

Naive Bayes model is easy to build and particularly useful for very large data sets. Along with simplicity, Naive Bayes is known to outperform even highly sophisticated classification methods.

Bayes theorem provides a way of calculating posterior probability P(c|x) from P(c), P(x) and P(x|c). Look at the equation below:

[](https://www.analyticsvidhya.com/wp-content/uploads/2015/09/Bayes_rule-300x172.png)Above,

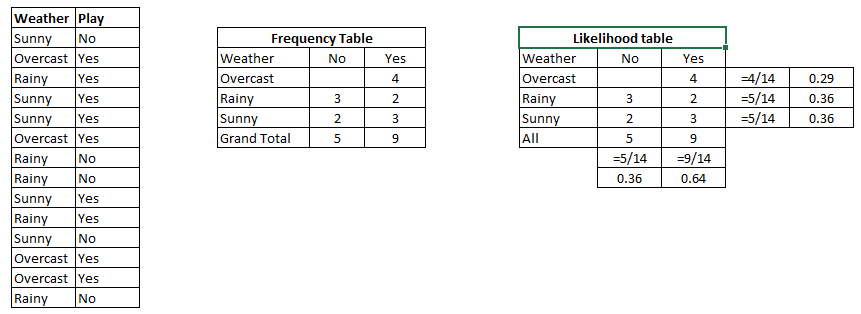
* *P*(*c|x*) is the posterior probability of *class* (c, *target*) given *predictor* (x, *attributes*).
* *P*(*c*) is the prior probability of *class*.
* *P*(*x|c*) is the likelihood which is the probability of *predictor* given *class*.
* *P*(*x*) is the prior probability of *predictor*.

## How Naive Bayes algorithm works?

Let’s understand it using an example. Below I have a training data set of weather and corresponding target variable ‘Play’ (suggesting possibilities of playing). Now, we need to classify whether players will play or not based on weather condition. Let’s follow the below steps to perform it.

Step 1: Convert the data set into a frequency table

Step 2: Create Likelihood table by finding the probabilities like Overcast probability = 0.29 and probability of playing is 0.64.

[](https://www.analyticsvidhya.com/wp-content/uploads/2015/08/Bayes_41.png)

Step 3: Now, use Naive Bayesian equation to calculate the posterior probability for each class. The class with the highest posterior probability is the outcome of prediction.

**Problem:**Players will play if weather is sunny. Is this statement is correct?

We can solve it using above discussed method of posterior probability.

P(Yes | Sunny) = P( Sunny | Yes) \* P(Yes) / P (Sunny)

Here we have P (Sunny |Yes) = 3/9 = 0.33, P(Sunny) = 5/14 = 0.36, P( Yes)= 9/14 = 0.64

Now, P (Yes | Sunny) = 0.33 \* 0.64 / 0.36 = 0.60, which has higher probability.

Naive Bayes uses a similar method to predict the probability of different class based on various attributes. This algorithm is mostly used in text classification and with problems having multiple classes.

## What are the Pros and Cons of Naive Bayes?

***Pros:***

* It is easy and fast to predict class of test data set. It also perform well in multi class prediction
* When assumption of independence holds, a Naive Bayes classifier performs better compare to other models like logistic regression and you need less training data.
* It perform well in case of categorical input variables compared to numerical variable(s). For numerical variable, normal distribution is assumed (bell curve, which is a strong assumption).

***Cons:***

* If categorical variable has a category (in test data set), which was not observed in training data set, then model will assign a 0 (zero) probability and will be unable to make a prediction. This is often known as “Zero Frequency”. To solve this, we can use the smoothing technique. One of the simplest smoothing techniques is called Laplace estimation.
* On the other side naive Bayes is also known as a bad estimator, so the probability outputs from predict\_proba are not to be taken too seriously.
* Another limitation of Naive Bayes is the assumption of independent predictors. In real life, it is almost impossible that we get a set of predictors which are completely independent.

## 4 Applications of Naive Bayes Algorithms

* **Real time Prediction:**Naive Bayes is an eager learning classifier and it is sure fast. Thus, it could be used for making predictions in real time.
* **Multi class Prediction:**This algorithm is also well known for multi class prediction feature. Here we can predict the probability of multiple classes of target variable.
* **Text classification/ Spam Filtering/ Sentiment Analysis:** Naive Bayes classifiers mostly used in text classification (due to better result in multi class problems and independence rule) have higher success rate as compared to other algorithms. As a result, it is widely used in Spam filtering (identify spam e-mail) and Sentiment Analysis (in social media analysis, to identify positive and negative customer sentiments)
* **Recommendation System:**Naive Bayes Classifier and [Collaborative Filtering](https://en.wikipedia.org/wiki/Collaborative_filtering) together builds a Recommendation System that uses machine learning and data mining techniques to filter unseen information and predict whether a user would like a given resource or not

## How to build a basic model using Naive Bayes in Python?

Again, scikit learn (python library) will help here to build a Naive Bayes model in Python. There are three types of Naive Bayes model under scikit learn library:

* [**Gaussian:**](http://scikit-learn.org/stable/modules/naive_bayes.html)It is used in classification and it assumes that features follow a normal distribution.
* [**Multinomial**](http://scikit-learn.org/stable/modules/naive_bayes.html)**:**It is used for discrete counts. For example, let’s say,  we have a text classification problem. Here we can consider bernoulli trials which is one step further and instead of “word occurring in the document”, we have “count how often word occurs in the document”, you can think of it as “number of times outcome number x\_i is observed over the n trials”.
* [**Bernoulli**](http://scikit-learn.org/stable/modules/naive_bayes.html)**:**The binomial model is useful if your feature vectors are binary (i.e. zeros and ones). One application would be text classification with ‘bag of words’ model where the 1s & 0s are “word occurs in the document” and “word does not occur in the document” respectively.

Based on your data set, you can choose any of above discussed model. Below is the example of Gaussian model.

### Python Code

#Import Library of Gaussian Naive Bayes model

from sklearn.naive\_bayes import GaussianNB

import numpy as np

#assigning predictor and target variables

x= np.array([[-3,7],[1,5], [1,2], [-2,0], [2,3], [-4,0], [-1,1], [1,1], [-2,2], [2,7], [-4,1], [-2,7]])

Y = np.array([3, 3, 3, 3, 4, 3, 3, 4, 3, 4, 4, 4])

#Create a Gaussian Classifier

model = GaussianNB()

# Train the model using the training sets

model.fit(x, y)

#Predict Output

predicted= model.predict([[1,2],[3,4]])

print predicted

**Output: ([3,4])**

### R Code:

require(e1071) #Holds the Naive Bayes Classifier

Train <- read.csv(file.choose())

Test <- read.csv(file.choose())

#Make sure the target variable is of a two-class classification problem only

levels(Train$Item\_Fat\_Content)

model <- naiveBayes(Item\_Fat\_Content~., data = Train)

class(model)

pred <- predict(model,Test)

table(pred)

Above, we looked at the basic Naive Bayes model, you can improve the power of this basic model by tuning parameters and handle assumption intelligently. Let’s look at the methods to improve the performance of Naive Bayes Model. I’d recommend you to go through [this document](http://www.inf.ed.ac.uk/teaching/courses/inf2b/learnnotes/inf2b-learn-note07-2up.pdf) for more details on Text classification using Naive Bayes.

## Tips to improve the power of Naive Bayes Model

Here are some tips for improving power of Naive Bayes Model:

* If continuous features do not have normal distribution, we should use transformation or different methods to convert it in normal distribution.
* If test data set has zero frequency issue, apply smoothing techniques “Laplace Correction” to predict the class of test data set.
* Remove correlated features, as the highly correlated features are voted twice in the model and it can lead to over inflating importance.
* Naive Bayes classifiers has limited options for parameter tuning like alpha=1 for smoothing, fit\_prior=[True|False] to learn class prior probabilities or not and some other options (look at detail [here](http://scikit-learn.org/stable/modules/generated/sklearn.naive_bayes.MultinomialNB.html#sklearn.naive_bayes.MultinomialNB)). I would recommend to focus on your  pre-processing of data and the feature selection.
* You might think to apply some classifier combination technique like ensembling, bagging and boosting but these methods would not help. Actually, “ensembling, boosting, bagging” won’t help since their purpose is to reduce variance. Naive Bayes has no variance to minimize.

## End Notes

In this article, we looked at one of the supervised machine learning algorithm “Naive Bayes” mainly used for classification. Congrats, if you’ve thoroughly & understood this article, you’ve already taken you first step to master this algorithm. From here, all you need is practice.

Further, I would suggest you to focus more on data pre-processing and feature selection prior to applying Naive Bayes algorithm.0 In future post, I will discuss about text and document classification using naive bayes in more detail.