A Collaborative Filtering Spotlight Into Machine Learning Song Recommendation Systems using kNN

Zakaria Gorgis
University of California, San Diego
Email: zgorgis@ucsd.edu

This paper will go into detail about the implementation of a collaborative filtering approach on two music data sets. The algorithm that was used was k-Nearest Neighbors and its implementation proved to be both simple to implement, thanks to various libraries. However, the algorithm was hard to tweak to get any meaningful results. We used two data sets, one of which we were unsuccessful with, while the other proved to be successful. After various tests of the algorithm, the songs that were recommended were very hit or miss, thus the probability of recommending a similar song would be around 40 to 60 percent.

1 Introduction / Motivation

When a layperson thinks of a recommendation system, more often than not, they point to a framework like movies. Most notably, they refer to Netflix, or YouTube as sources of refined recommendation systems. Without a doubt, they would be right about their assumption. Companies like Netflix implement a Collaborative filtering design in which the algorithm takes ”user-to-user” interactions to recommend movies that are similar. On the other hand, Amazon is well known to have robust and extremely sophisticated implementation of Content-based filtering Machine Learning algorithms, in which the software makes recommendations that are based on a collection of user preferences with respect to product features. While a Content-based filtering algorithm would be impressive to implement, the focus of this project was to attempt to execute a rudimentary Collaborative filtering design in the form of a song recommendation system.
Why?, the answer is simple. To attempt to take what was learned in a few weeks in *ECE* 196 and implement a practical framework that could be used in the real world. Everyone listens to music on some platform, whether it be Apple Music, Spotify, Tidal, or Amazon Music; the list is extensive. All of these aforementioned platforms use some type of recommendation algorithm to create the most personalized experience for their users. While this project is in no place to be as revolutionary, state-of-the-art or even advanced, it highlights the implication that anyone with just a basic understanding of some python libraries and Machine Learning algorithms can implement a simple recommendation system.

2 Problem-Solving Approach

When looking at this project from an eagle's eye perspective, there are a few problems that come to fruition that quickly needed to be dealt with. The first is finding a formidable data set. There were two different data sets that were used in the implementation of this project. The first was found on Kaggle.com, **Spotify Data set 1921-2020**. The second data set was an extremely popular one that was created to promote the general public to implement their own recommendation algorithms, **Million Song Data set**.
After locating two promising data sets, the next problem was what kind of Machine Learning algorithm to implement. Since the main focus was collaborative filtering, the possibility of more refined and high-level algorithms like ANN, tensor factorization, and deep learning was out of the discussion. Instead, the choice became between kNN or matrix factorization. There are obvious pros and cons to each of these two implementations, which will be discussed later on. But, due to our novice background of Machine Learning, this project opted to use kNN. After taking care of the first two problems, only one remains, which is the actual implementation of the algorithm and data sets, which will be discussed extensively now in the next section.

3 Solution

To begin with, it's important to lay an extensive background as to why kNN was chosen, and what exactly kNN does in a recommendation algorithm. kNN, or k-Nearest Neighbors, is known as a supervised machine learning algorithm. What this means is that it isn't very modular in nature. For example, if a distant company would need to use this project, they would not only need the source code to the algorithm implementation, BUT also the data that the algorithm learned on. We did not find this downside a big issue, as modularity was not very important. More over, kNN is a lazy learning method that relies on not being parametric. According to an article by Jason Brownlee, being non-parametric in machine learning means that an algorithm does not make a strong assumption about the mapping functions. And when a machine learning algorithm is lazy, it means that the function does not generalize the training data, until after a query is created as an input. So what exactly does this mean for kNN as an algorithm for a recommendation system? It means that its flexible in nature, we can feed it many different feature vectors of length n without much problem. Which is exactly what we needed as the data sets contained hundreds of thousands of songs as well as fifteen plus parameters.

As a preface, kNN becomes hard to visualize after three parameters. In order to explain our logic, we can visualize a kNN with two parameters, tempo and energy. So in other words, we have a feature vector v, where x_1 = tempo and x_2 = energy.

$$v = \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} \tag{1}$$

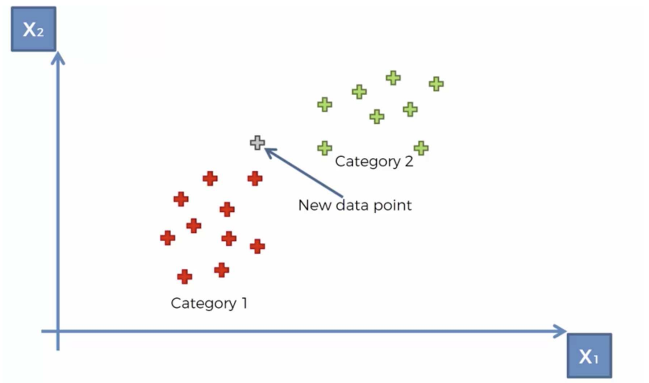


Fig. 1. Note that its easy to find the best classification for the new data point when only in two dimensions. This becomes much harder when $n > 3$

In figure 1, it's easy to see that the classification of the new song falls in either category 1 or 2 dependent on whatever value k was chosen to be. However, in our case, a song recommendation isn't as clearly black or white. In the sense that it's not valuable to the user to just label a song as having more "tempo" or being more "energetic". Originally, we thought that this problem of not being able to classify our songs as being a inherent fault of the distance metric used by kNN. Why exactly did we think this? Well to best explain it, we need to look at the **Spotify Data set 1921-2020**. The data set itself is stored on a file called **data.csv**. In that particular file, each song has 19 attributes. This already became a red flag due to the phenomena that is created when dealing with high amounts of dimensions in medium to large size data sets like the aforementioned Spotify Data set. What exactly is the curse of dimensionality? According to **DeepAI.org**, its when the volume of space that is represented grows exponentially that the data that does exist becomes too sparse and computationally taxing on present day computers. Because of this sparsity, data points become much harder to classify as there is much more space between them. Think of the example that was presented before, where we had a feature vector v, where x_1 = tempo and x_2 = energy. In a $2D$ space, its easy to visualize, and easy to see the commonality between the attributes. Now if we kept x_1 and x_2, but instead introduced 8 more parameters, this commonality becomes much more difficult to accurately detect. Obviously the curse of dimensionality probably plays a much bigger role in data sets where their feature vectors are in the hundreds or thousands,however we still thought that it might play some problem in correctly creating a list of similar songs. With this in mind, we had to chose a viable distance measure to be used in finding the nearest neighbor. We were able to gather information, mainly from the **sklearn.neighbors** documentation, as well as from other examples where distance measurements were used and came up with two different viable measurements: the euclidean distance, or the cosine similarity.

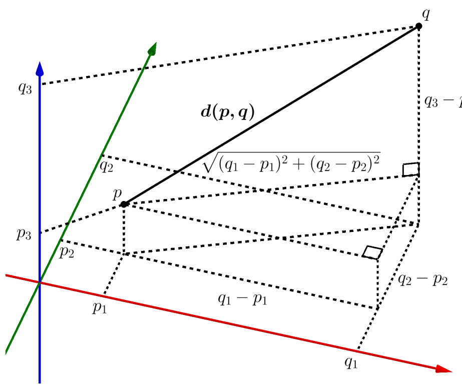


Fig. 2. A visualization of the euclidean distance when given three parameters, i.e in 3D

As visualized above, it tends to get pretty convoluted the more dimensions that are added. This is especially apparent when taken into account the amount of computations that are executed on an n-dimensional data set. If we look at the equation itself, we begin to understand why there are so many computations;

$$d(p,q) = \sqrt{(p_1 - q_1)^2 + (p_2 - q_2)^2 + \cdots + (p_n - q_n)^2}. \quad (2)$$

Every single attribute on the new data point must be subtracted by every existing points feature vector then squared, summed and rooted just to get the euclidean distance from one another. Thus the computational complexity of kNN is an outstanding $O(kdn)$, where k is the total nearest neighbors, d is the amount of attributes (dimensions), and n is the cardinality of the data set (most likely just training). This massive computational tax was very apparent when attempting to run euclidean distance on the data set over Google Colab, a Jupyter based collaborative notebook. Often times when k was large, or we fed a data set that was above a certain training threshold, the program would crash requiring much more RAM than what was provided. So we tried the other distance formula, cosine similarity. The equation of this distance measurement is given as;

$$\cos(\theta) = \frac{\mathbf{A} \cdot \mathbf{B}}{\|\mathbf{A}\| \|\mathbf{B}\|} = \frac{\sum_{i=1}^{n} A_i B_i}{\sqrt{\sum_{i=1}^{n} A_i^2} \sqrt{\sum_{i=1}^{n} B_i^2}} \quad (3)$$

Where **A** and **B** are non-zero vectors, as the dot product of these two is equal to their magnitude multiplied by $\cos(\theta)$. It's implementation looked daunting at first, however the **sklearn.neighbors** library (using sklearn.neighbors metric) made implementing cosine similarity and euclidean distance very easy. After some consideration, we opted for the cosine similarity distance measurement for kNN, as it proved to be the fastest in our case where the goal was to recommend songs similar to the user input. After we decided on what measurement to use, the next step was to parse the data from the Spotify set into usable chunks of information. To avoid the curse of dimensionality, and to promote a more efficient run time, we opted to split the data into four different groupings, and run kNN on three of the four. The data.csv file contained songs that each had 19 attributes such that attributes = ['acousticness', 'artists', 'danceability', 'durationMS', 'energy', 'explicit', 'id', 'instrumentalness', 'key', 'liveness', 'loudness', 'mode', 'name', 'popularity', 'name', 'releaseDate', 'speechiness', 'tempo', 'valence', 'year']. These attributes were subsequently broken into four groupings, one containing the song name, artists, year, and ID, while the others were broken into the similarity of their attributes for better cohesion. At this point, we were ready to test our implementation. We had chosen kNN as the machine learning algorithm, chosen the best fit distance measurement and split the data up into usable chunks to help with efficiency and recommendation. However, at this point in the project we hit a pretty major wall, the output to our code was most similar to that of kNN on the famous iris data set, it would just return a float value instead of the actual song itself. our implementation of the algorithm did not take into account how kNN classification worked, and it often times was too good on the training data but horrible on the testing data. We over fitted our hyper-parameters and in turn failed to create any meaningful recommendations (often times would just get a compiler error as the library was unable to parse the data correctly).

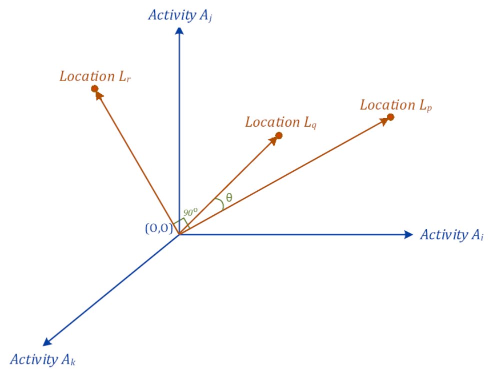
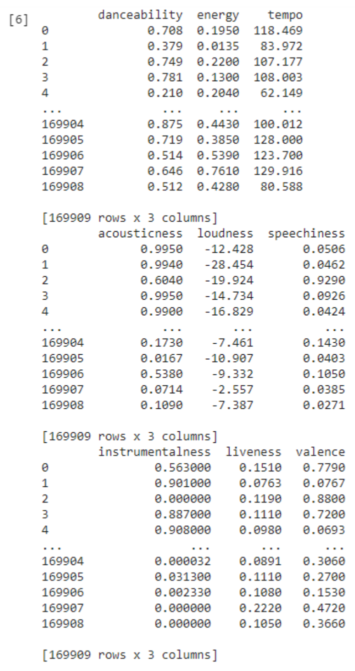


Fig. 3. A visualization of the cosine similarity when given three parameters, i.e in 3D



```
[6]          danceability  energy    tempo
     0              0.708  0.1950  118.469
     1              0.379  0.0135   83.972
     2              0.749  0.2200  107.177
     3              0.781  0.1300  108.003
     4              0.210  0.2040   62.149
     ...              ...     ...      ...
     169904         0.875  0.4430  100.012
     169905         0.719  0.3850  128.000
     169906         0.514  0.5390  123.700
     169907         0.646  0.7610  129.916
     169908         0.512  0.4280   80.588

     [169909 rows x 3 columns]
             acousticness  loudness  speechiness
     0             0.9950   -12.428       0.0506
     1             0.9940   -28.454       0.0462
     2             0.6040   -19.924       0.9290
     3             0.9950   -14.734       0.0926
     4             0.9900   -16.829       0.0424
     ...              ...       ...          ...
     169904        0.1730    -7.461       0.1430
     169905        0.0167   -10.907       0.0403
     169906        0.5380    -9.332       0.1050
     169907        0.0714    -2.557       0.0385
     169908        0.1090    -7.387       0.0271

     [169909 rows x 3 columns]
             instrumentalness  liveness  valence
     0               0.563000    0.1510   0.7790
     1               0.901000    0.0763   0.0767
     2               0.000000    0.1190   0.8800
     3               0.887000    0.1110   0.7200
     4               0.908000    0.0980   0.0693
     ...                  ...       ...      ...
     169904          0.000032    0.0891   0.3060
     169905          0.031300    0.1110   0.2700
     169906          0.002330    0.1080   0.1530
     169907          0.000000    0.2220   0.4720
     169908          0.000000    0.1050   0.3660

     [169909 rows x 3 columns]
```

Fig. 4. Code snippet from the data sets split into three groups based on the similarity of their attributes

4 Solution 2

With our algorithm on the original Spotify data set not working, we opted to shift our focus on the second data set, **Million Song Data set**. While discouraged, the new knowledge obtained from the original data set made implementing it on this new one very modular. This new data set contained much less parameters than before. Whats nice about the million song data set is that the data sets are grouped by user ID, song ID, number of plays, and information about the song itself like its publication date and length. There no longer was data specific about the type of song as the focus shifted from implementing kNN based on characteristics of the song to implementing kNN based on similar songs played by users. The set up was nearly the same. We still used kNN, and still used cosine similarity distance measurement. The only difference this time was that we had to implement a sparse matrix to account for the songs, and how much they were played each. This new knowledge of sparse matrices increased the possibility of implementing high dimensional data. Because not everyone listens to every song in the data set, sparse matrices make it possible to look at the clusters of similar data points to then in turn recommend songs based on those similarities. With this new data set up for kNN, the focus turned to the sklearn library. Thanks to this library, the implementation of the kNN algorithm is actually very simple. In fact it can be done in a couple of lines where we set the training data, testing data, *k*, the metric and the algorithm. The transition from looking at song

```
kNN = NearestNeighbors(n_neighbors= 10,
                       radius = 1,
                       algorithm = 'brute',
                       metric = 'cosine')
kNN.fit(songs)
```

Fig. 7. Example of how simple sklearn library makes kNN

characteristics to looking at the popularity of songs between users was the key to success. We were able to match the corresponding song ID to the user ID who played the song *n* times, and with the sparse matrix we were able to create a pivot table to see where the most hits/similarities were, creating an output list of the most similar songs to the input. While we were able to create a working song recommender, the output was interesting to say the least. The songs recommended seem to vary heavily on the distance measure used. As an example, we wanted to find similar songs to **Tears in the Rain by the Weeknd**. As a measurement of complexity, we imported the timeit library from python to check how long it would take for the kNN algorithm to find the ten most similar songs to Tears in the Rain. We also ran it using the cosine similarity distance measurement achieving these 10 songs in no particular order;**['Something About Us', 'Frases Mudas', 'Bottle Of Red Wine' ,'He Loves Me', 'Touchy! (Album Version)', 'Eyes Without A Face', 'Epiphany (LP Version)', 'Just Like A Woman', 'Soliloquy Of Chaos (Explicit)', 'Give A Little Bit']** and it took the algorithm **0:00:39.082440** to compute. We ran the same input instead opting to use the euclidean metric, and interestingly enough the results for vastly different, (again songs in no particular order); **['Ribbons', 'One Beat', 'Slan Le Maige', 'He Loves Me', 'clouding', 'La guitarra del joven soldado', 'Shook Ones pt. 2', 'Summer In Dixie (LP Version)', 'Open Your Eyes (Explicit LP Version)', 'Bottle Of Red Wine']** and it took the algorithm **0:00:48.225557** . Just to be sure that the variation in the song recommendations wasn't by chance, we ran the same algorithm over and over again, getting the same output with a 10 plus or minus margin for the time taken. At this point, we thought that it might be due to the fact that Tears in the Rain was a very recent song that just wasn't in the data set so it had to compromise. With this in mind, we chose a popular song in the 90s, **Say My Name by Destiny's Child**, and again the algorithm using the cosine similarity distance metric. Impressively, this time around, the algorithm found 10 songs that we would personally say are similar in one way or another; **['Ms. Jackson', 'Day Night', 'Back Down', ''You're So Last Summer (Album Version)'', 'Baby Doll (Edited)', ''Jumpin' Jumpin','Dip It Low', 'Salvation', 'The Irony Of It All**

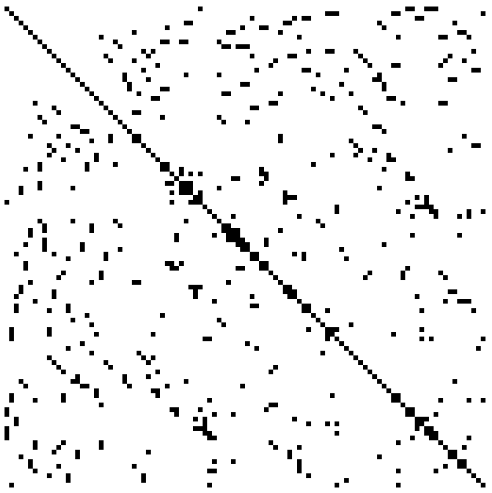
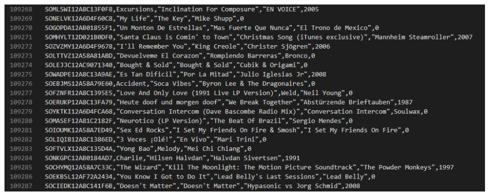


Fig. 5. Visual example of a sparse matrix in which clusters of data points show characteristics of similarities that can be used to recommend songs based on *k* and the features



```
SOMLSWI12A8C13F0F8,Excursions,"Inclination For Composure","EN VOICE",2005
SONELVK12A6D4F60C8,"My Life","The Key","Mike Shupp",0
SOGOPDA12AB01855F1,"Un Monton De Estrellas","Mas Fuerte Que Nunca","El Trono de Mexico",0
SOMHYLT12D021B0DF0,"Santa Claus is Comin' to Town","Christmas Song (iTunes exclusive)","Mannheim Steamroller",2007
SOZVZMY12A6D4F9678,"I'll Remember You","King Creole","Christer Sjögren",2006
SOLTTVZ12A58A81ABD,"Devuelveme El Corazon","Rompiendo Barreras",Bronco,0
SOLEJJC12AC9071340,"Bought & Sold","Bought & Sold","Cubik & Origami",0
SOWADPE12A8C13A9AE,"Es Tan Dificil","Por La Mitad","Julio Iglesias Jr",2008
SOEBJMS12A58A79E60,Accident,"Soca Vibes","Byron Lee & The Dragonaires",0
SOFZNFR12A8C1395E5,"Love And Only Love (1991 Live LP Version)",Weld,"Neil Young",0
SOERUKP12A8C13FA79,"Heute doof und morgen doof","We Break Together","Abstürzende Brieftauben",1987
SOYKTKI12A6D4FCA68,"Conversation Intercom (Dave Bascombe Radio Mix)","Conversation Intercom",Soulwax,0
SOMASEF12A81C2182F,"Neurotico (LP Version)","The Beat Of Brazil","Sergio Mendes",0
SOIOUMK12A58A7ED49,"Sex Ed Rocks","I Set My Friends On Fire & Smosh","I Set My Friends On Fire",0
SOLIQIB12A8C1386ED,"3 Veces ¡Olé!","En Vivo","Mari Trini",0
SOFTVLX12A8C135D4A,"Yong Bao",Melody,"Mei Chi Chiang",0
SONKGPC12AB0184AD7,Charlie,"Hilsen Halvdan","Halvdan Sivertsen",1991
SOCHYMQ12A58A7C33C,"The Wizard","Kill The Moonlight: The Motion Picture Soundtrack","The Powder Monkeys",1997
SOEKBSL12AF72A2434,"You Know I Got to Do It","Lead Belly's Last Sessions","Lead Belly",0
SOCIEDK12A8C141F6B,"Doesn't Matter","Doesn't Matter","Hypasonic vs Jorg Schmid",2008
```

Fig. 6. Example from the million song CSV file in which each song has its song ID and information which can later be matched with the user ID and their song played ID

(Album Version)', 'This Way'] with a surprisingly time average of **0:00:45.321584** after 10 successive runs. Again, we ran the same song input but this time using the euclidean distance metric, and got the 10 following songs, **['Check On It', 'One Beat', 'Runaway (2007 Remastered Remix Version)', 'Ribbons', 'Slan Le Maige','clouding', 'La guitarra del joven soldado', 'Romancipation', 'Summer In Dixie (LP Version)', 'Baby Doll (Edited)']** with an expected slow time average of **0:00:54.317009** after 10 successive runs. Thus, the testing leaves us with the ultimate question, were the recommended songs actually accurate? In this case there is really no way to specifically know unless we went one by one and compared the music through some database to find its characteristics to actually see if the songs were similar. Other than that, its success rate is based off of subjective metrics. Maybe a grouping of recommended songs sound similar to the song that the user played, or maybe its vice versa. We were in no technical space to be able to implement some data collection algorithm that would not just recommend the song but also create some sort of chart to pin point the similarities.

5 Summary / Conclusion / Future Work

The project in itself was a very interesting and enjoyable process, especially the debugging. It's funny, no one ever explicitly tells a software engineer that there going to spend 85 percent of their time debugging and only 15 percent writing actual code. As for how to improve the project, there are countless ways. One particular example would be to implement a matrix factorization recommender system instead of kNN. The one huge problem that was found throughout the testing process was the fact that kNN is insanely slow. The calculations are just gargantuan, and even if they are simple computations like addition and multiplication, the fact that the algorithm has to perform this hundreds of thousands to millions of times in a large data set is just too taxing. It would take nearly a minute every time we ran the algorithm, which is too much time spent doing nothing. Another bonus of using matrix factorization over kNN is that it is able to give a more personalised experience for a user, rather than just feeding it songs that others liked. Another area of improvement would be proper selection of data sets. We know this especially detrimental to the progression of a working algorithm. If we spent twice as long finding a suitable data set, we would have spent half as long actually implementing a working algorithm. Another way for us to fix our data set issue would be to combine our data set based on user interaction, with a data set that has song content data, and use the intersection of songs throughout both sets. This would not only give us the most data, but provide our system with much more dimensionality. The data set is like an overhead that cannot be avoided throughout the project process so it's essential to choose something that's efficient. That being said, the failure to make anything from the first data set taught us an immense amount about the kNN algorithm, distance measurements and how to handle data in the first place. Moving forward, we can take this failure and learn from it, as we did while implementing the second data set. Although we initially thought we had run into another flawed data set with our system's inability to recommend songs based off of **Tears in the Rain by the Weeknd,** we realised it was simply just out of date. Feeding the system a song it was able to recognize showed that we had successfully been able to create a recommendation system. One of our greatest constraints in this project, would obviously be the time limit. If given more time, we would have been able to use a data collection algorithm, such as **MAP@K,** to be able to visualise how useful our system is. If given a longer time frame, we could enhance our system by collecting new data, and expanding upon our model by creating a system for users to be able to tell us their reasoning behind whether or not they like a song. Being able to differentiate between whether a person dislikes an artist, a genre, or even if they simply just don't enjoy that particular song would prove very useful to our system. We can already see such implementation on YouTube's homepage, which just adds to this method's legitimacy. Having access to user rational would allow our system to be able to tweak its recommendations in the way the user needs. All in all, we have learned a lot about machine learning and deep learning through the creation of this project, and in the future, we plan to utilize our methods on more complex modalities, such as movies, or social media posts.

6 References

[1] Behnoush Abdollahi. 2017. Accurate and justifiable: new algorithms for explainable recommendations. Ph.D. Dissertation. University of Louisville

[2] Damak, Khalil, and Olfa Nasraoui. "SeER: An Explainable Deep Learning MIDI-based Hybrid Song Recommender System." arXiv preprint arXiv:1907.01640 (2019).

[3] Istvan Pilaszy. "Neighbor methods vs matrix factorization: case studies of real life recommendations." Gravity Research and development. Sep 22, 2015. https://www.slideshare.net/domonkostikk/neighbor-methods-vs-matrix-factorization-case-studies-of-reallife-recommendations-gravity-lsrs2015-recsys-2015

[4] Tan, Ren Jie. "Breaking down Mean Average Precision (MAP)." Medium, Towards Data Science, 6 July 2020, towardsdatascience.com/breaking-down-mean-average-precision-map-ae462f623a52.

[5] Thierry Bertin-Mahieux, Daniel P.W. Ellis, Brian Whitman, and Paul Lamere. 2011. The Million Song Dataset. In Proceedings of the 12th International Conference on Music Information Retrieval(ISMIR 2011).

[6] DeepAI. "Curse of Dimensionality." DeepAI, 17 May 2019, deepai.org/machine-learning-glossary-and-terms/curse-of-dimensionality.

[7] "1.6. Nearest Neighbors." Scikit, scikit-learn.org/stable/modules/neighbors.html.

[8] Prabhakaran, Selva. "Cosine Similarity - Understanding the Math and How It Works? (with Python)." ML+, 11 Oct. 2020, www.machinelearningplus.com/nlp/cosine-similarity/.

[9] "Euclidean Distance." Euclidean Distance - an Overview — ScienceDirect Topics, www.sciencedirect.com/topics/mathematics/euclidean-distance.

[10] Brownlee, Jason. "A Gentle Introduction to Sparse Matrices for Machine Learning." Machine Learning Mastery, 9 Aug. 2019, machinelearningmastery.com/sparse-matrices-for-machine-learning/.

[11] Harrison, Onel. "Machine Learning Basics with the K-Nearest Neighbors Algorithm." Medium, Towards Data Science, 14 July 2019, towardsdatascience.com/machine-learning-basics-with-the-k-nearest-neighbors-algorithm-6a6e71d01761.

[12] Nelson, Daniel. "What Is a KNN (K-Nearest Neighbors)?" Unite.AI, 23 Aug. 2020, www.unite.ai/what-is-k-nearest-neighbors/.

[13] M'Haimdat, Omar. "Understand the Fundamentals of the K-Nearest Neighbors (KNN) Algorithm." Medium, Heartbeat, 12 May 2020, heartbeat.fritz.ai/understand-the-fundamentals-of-the-k-nearest-neighbors-knn-algorithm-533dc0c2f45a.