On the Use of Collaborative Filtering and Optimization Techniques for Recommender Systems

Noah F. Teshima
Electrical Engineering Laboratory
Department of Mathematics
University of California
San Diego, California 92093
Email: nteshima@ucsd.edu

Luis D. Jibaja
Electrical Engineering Laboratory
Department of Computer Science and Engineering
University of California
San Diego, California 92093
Email: ljibajap@ucsd.edu

For platforms such as Spotify, eBay, and Netflix, the ability to make inferences on content that users may like is a problem commonly approached with machine learning. We introduce the concept of a recommender system. Additionally, content-based and collaborative filtering, two of the most favoured approaches to building recommender systems, are discussed alongside their technical trade-offs. Furthermore, this work demonstrates the construction of a collaborative filtering recommender system on the MovieLens dataset. Finally, unconstrained matrix factorization with gradient descent, the machine learning model chosen for our recommender system, is also discussed.

Nomenclature

Uniform Distribution We generate synthetic data for features with a uniform distribution. If $\Omega \subset R^n$ is a bounded region, then a random variable $X \sim Unif(\Omega)$ if its probability density function f_X is defined as

$$f_X(x) = \begin{cases} \frac{1}{\text{vol}(\Omega)} & x \in \Omega \\ 0 & x \notin \Omega \end{cases} \quad (1)$$

where $\text{vol}(\Omega)$ denotes the volume of the bounded region.

Vector We denote vectors with boldface font. For instance, $\boldsymbol{x} := (x_1, \ldots, x_n)^T$ would be a vector in R^n.

Rank Our paper uses rank to refer to column rank. Specifically, for $A \in \mathbf{M}^{n\times m}$, the rank of A denotes the dimension of the column space of A.

1 Introduction

We begin by first introducing the purpose of a recommender system. Many content platforms allow users to buy or browse for products. For instance, Netflix, a popular streaming service, allows users to browse and watch documentaries, movies, and television shows. However, users have a vague notion of what type of products they like and dislike, and cannot reasonably expand on their preferences. Informally speaking, recommender systems solve this by utilizing information about users and products in order to infer customer interests. [1]. With regards to our example with Netflix, if a user likes the television shows ”Breaking Bad” and ”Ozark”, it seems reasonable to recommend other crime dramas such as ”Better Call Saul”. Two approaches to building recommender systems include content-based filtering and collaborative filtering.

1.1 Content-Based Filtering

In content-based recommender systems, item-specific features are used as a basis for making recommendations. In other words, based on the feature values of an item, we can recommend different items given a user’s preferences. For instance, suppose a user gives positive feedback for the television show ”Breaking Bad” on Netflix. By using content-based filtering, genre specific keywords for other shows can

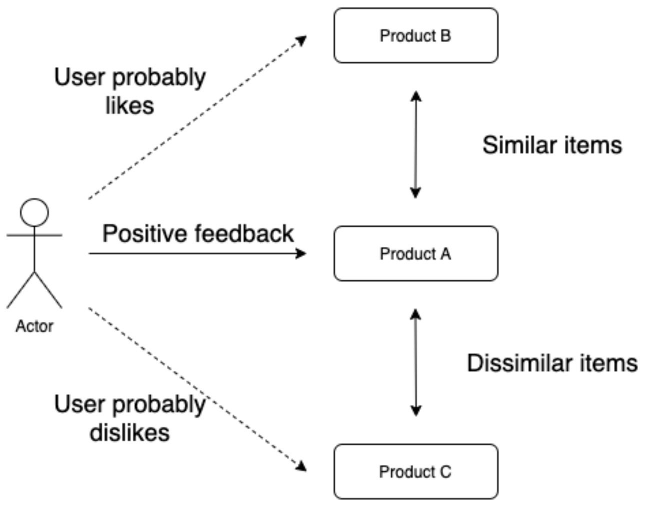


Fig. 1. A recommender system with content-based filtering looks for items with similar features for inference.

serve as an indicator for other good recommendations. Likewise, the value of item-specific features serves as an indicator for bad recommendations. Figure 1 illustrates the intuition behind content-based filtering for predicting relevant content.

1.2 Collaborative Filtering

A somewhat similar approach used in recommender systems is defined as collaborative filtering. While content-based filtering uses item-specific features for making recommendations, collaborative filtering uses item-specific features in addition to user feedback for predictions. We again illustrate this distinction with our Netflix example: suppose user A and user B both gave positive feedback for the television show "Breaking Bad". It can be inferred that user A and user B have similar preferences. Therefore, if user B gives positive feedback for the television show "Better Call Saul", collaborative filtering would suggest this as a recommendation to user A. Figure 2 helps to illustrate this distinction.

1.3 Technical Trade-Offs of Content-Based and Collaborative Filtering

Both content-based filtering and collaborative filtering provide two approaches to alleviate the issues presented by information overload: the complexity in making reasonable decisions in the presence of large amounts of data. For collaborative filtering, exploiting user similarity in order to provide recommendations presents the key advantage of serendipity. While a user may not explicitly provide feedback for a specific type of product, the use of user similarity is able to provide recommendations beyond the scope of their interests solely from feedback received. Despite this, the extent to which serendipitous recommendations are made is limited by sparse feedback. A large subset of items in information systems are not rated for each user. As the number of items in a system increases, the chance of finding similar items between users decreases. [2]. A contrasting issue with collaborative filtering is the instance where we are

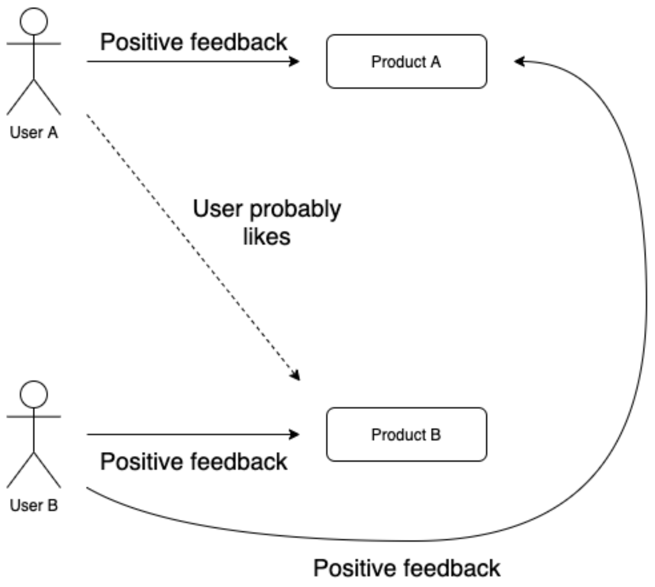


Fig. 2. A recommender system with collaborative filtering looks collectively at items and users with similar feedback for inference.

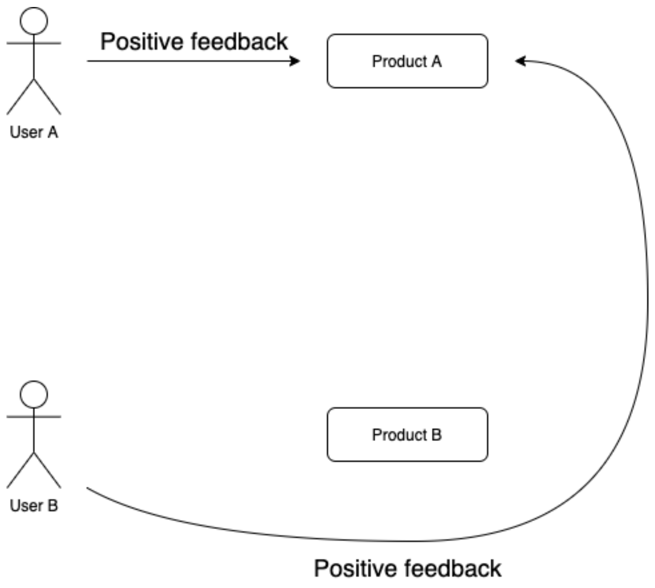


Fig. 3. Illustration of an extreme case of the item cold-start problem. Item B in this diagram has yet to receive feedback from either user and therefore will not be recommended.

presented with no user feedback or item feedback. This is known as the cold-start problem, and presents issues with drawing reasonable inferences when a sufficient amount of information has yet to be gathered. There are two instances of the cold-start problem that occur with collaborative filtering: the item cold-start problem and the user cold-start problem. The item cold-start problem occurs when insufficient information is gathered for any particular item. As illustrated by figure three, since recommendations with collaborative filtering are drawn from user similarity, items with insufficient feedback will have limited interactions. In contrast, the user cold-start problem happens when insufficient feedback is gathered from a user. In this instance, it is diffi-

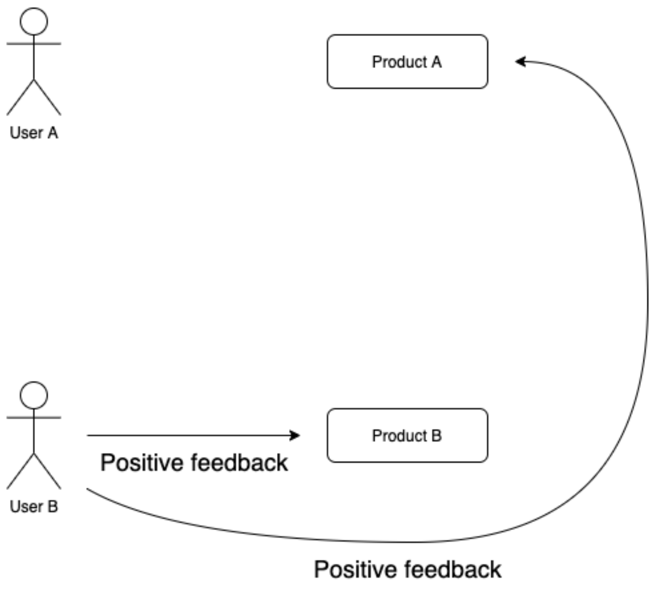


Fig. 4. Illustration of an extreme case of the user cold-start problem. User A in this diagram has yet to give feedback to any items. This creates a strong degree of uncertainty in our prediction.

cult to draw accurate predictions for user likes and dislikes.

In comparison, content-based filtering exploits content similarity in order to make predictions. This approach possesses the primary advantage of robustness against the item cold-start problem. Unlike collaborative filtering, content-based filtering does not rely on the amount of item feedback, since it does not leverage the feedback of other users in making decisions. Instead, content-based filtering relies on the content of each item, which is dependent on the feature representation for items. In spite of this, content-based filtering is also accompanied by the disadvantage of not being able to expand on user interests. Given a set of feedback from users, it is difficult to expand on user interests beyond their current feedback with content-based filtering [3]. In this sense, content-based filtering provides trivial recommendations. Finally, while content-based filtering is resistant to the item cold-start problem, it is not resistant against the user cold-start problem. This is due to the fact that training a model with content-based methods for a target user requires the use of their feedback.

2 Project Description

2.1 Dataset

The MovieLens dataset contains information on five-star ratings and tagging activty on the MovieLens movie recommendation platform. Collectively, there are 100,836 ratings across 9742 movies, with 3683 tag applications for 610 users. For the MovieLens platform, ratings are given on a scale of 0.5 stars to 5 stars, with 0.5 star increments. Users were randomly sampled from a pool of candidates that have at least 20 movie ratings. Furthermore, movies are included from the pool of candidates with at least one rating or tag.

2.2 Recommender System Approach

We are primarily interested in constructing a recommender system in order to give movie suggestions to any existing users existing inside of this dataset. Given the manner in which users were selected, we have that the user cold-start problem is not a severe issue, since each user has at least 20 movie ratings associated with their account. Similarly, since only movies with at least one rating are included in this dataset, the item cold-start problem is mitigated to some degree. This a priori knowledge about our dataset makes it reasonable to construct our recommender system with collaborative filtering. Given that both users and items are represented with feedback to a reasonable degree, we have sufficient user-item information in order to draw inferences for recommendations.

The construction of our recommender system with collaborative filtering was partitioned into the following epics: constructing a feedback matrix, building a latent factor model with matrix factorization, and item retrieval from our matrix factorization model. For the sake of transparency, we discuss these approaches in addition to their technical trade-offs alongside the construction of our recommender system.

2.3 Feedback matrix

The user feedback matrix is a matrix representation of user and item feedback for our dataset. Specifically, let $\mathcal{F}^n$ define our feature space: a vector-space representation of the features we choose to include in our model. In this context, n denotes the number of distinct items in our MovieLens dataset. To construct a matrix to further identify user-item similarities, we define $F \in \mathcal{F}^{m\times n}$ to be our feedback matrix, where m denotes the number of distinct users. For our feedback matrix, entry F_{ij} denotes the rating given by user i for item j. The definition of our feedback matrix must also take into account how items with no feedback should be managed. In order to account for this issue of nonresponse bias (with regards to items a user has not reviewed yet), our feedback matrix follows the given piecewise definition:

$$F_{ij} = \begin{cases} k, \text{User } i \text{ reviewed item } j \\ 0, \text{User } i \text{ has not reviewed item } j \end{cases} \tag{2}$$

where k in this definition denotes the number of stars for the rating of item i by user j. For instance, a rating of three and a half stars would correspond to a value of $k = 3.5$.

2.4 Matrix Factorization

We first motivate the idea of latent factor models with the curse of dimensionality before discussing Matrix Factorization. Consider the following figures. In each figure, we have randomly generated a dataset with color corresponding to the value of a categorical target feature. Each feature is uniformly sampled on the real interval $[0, 100]$. Note that as the number of features in our data increases, the number of unit-length cells in our feature space increases exponentially.

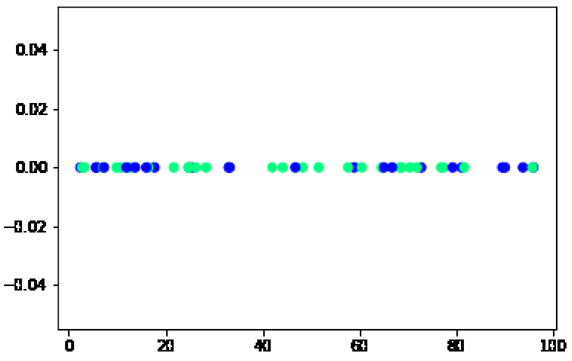


Fig. 5. Dataset of the form $\{x_i\}_{i=1}^{100}$ with targets $\{y_i\}_{i=1}^{100}$, where x_i is sampled from the uniform distribution $Unif[0,100]$ and targets $y_i \in \{0,1\}$ are categorical.

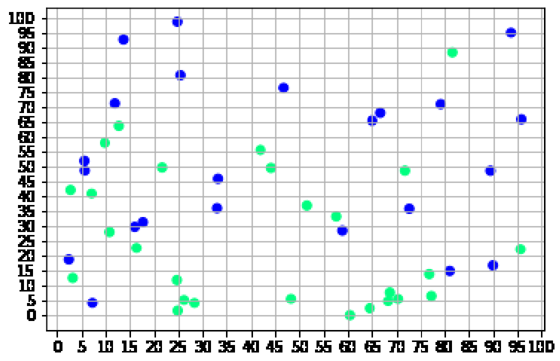


Fig. 6. Dataset of the form $\{(x_{i1}, x_{i2})\}_{i=1}^{100}$ with targets $\{y_i\}_{i=1}^{100}$, where x_i is sampled from the multivariate uniform distribution $Unif[0,100]^2$ and targets $y_i \in \{0,1\}$ are categorical.

For instance, in the first plot, our data is described by a single feature and therefore densely packed into 100 unit-length cells. In comparison to the second and third plot, the number of such cells grows exponentially (to 100^2 and 100^3 unit-cells, respectively) with the dimension of our feature space. Hence, in order to reasonably classify unseen data in this feature space, the size of a training set for this model would need to grow exponentially with each added feature.

The issues presented by the curse of dimensionality are further compounded by sparse datasets. For our feedback matrix, since the proportion of ratings given by each user relative to the total number of movies included is small, the chosen model must be able to address both issues. When working with a dataset with a large number of features, some of the information conveyed by these features may be redundant. This is to say that there may be features that are highly-correlated with each other, which arises from indirect measurement for some underlying, immeasurable source [4]. In other words, we hope that there is some underlying structure in our data that extracts a significant amount of information with a much smaller number of latent, or unobserved, features [1]. By constructing a latent factor model, we are able to model our users and items in our feedback matrix F with the corresponding latent vectors [5]. Matrix factorization is a type of latent factor model that exploits the fact that any highly correlated rows or columns in our feedback matrix can be well-estimated by a smaller number of latent features. Specifically, given that our feedback matrix F is an m by n matrix, we hope that F has the "structure" of a rank d matrix, where $d << \min\{m,n\}$. F can then be approximated as a decomposition of rank d matrices, specifically

$$F \approx UV^T \tag{3}$$

where $U \in \mathcal{F}^{m\times d}$ is denoted as our user matrix and $V \in \mathcal{F}^{n\times d}$ is denoted as our item matrix. The columns of U and V are the latent vectors, and describe how heavily users and items load on each of our latent factors [2]. It is important to note that matrix factorization only provides a rank d approximation

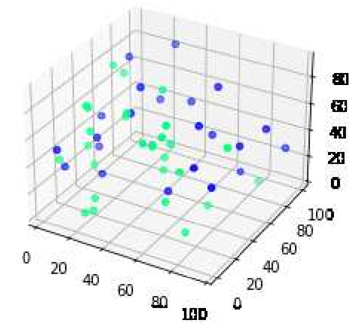


Fig. 7. Dataset of the form $\{(x_{i1}, x_{i2}, x_{i3})\}_{i=1}^{100}$ with targets $\{y_i\}_{i=1}^{100}$, where x_i is sampled from the multivariate uniform distribution $Unif[0,100]^3$ and targets $y_i \in \{0,1\}$ are categorical.

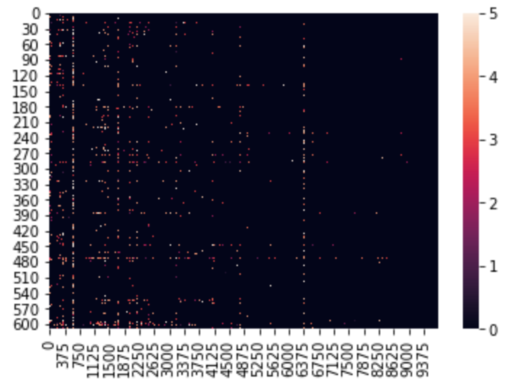


Fig. 8. Heatmap of the MovieLens dataset. The lack of color illustrates the sparsity.

[1]These latent features may or may not have a strong semantic interpretation; that is not their purpose. It is best to consider latent factors as features that best describe the structure of our data.

[2]The amount by which users and items load on each of our latent factors can be thought of as weights in a linear combination of our latent vectors.

of F. We will see in a later section how we choose d to minimize loss.

2.4.1 Using the Frobenius Norm as an Objective Function

In order to measure how well UV^T describes F, we consider the residual matrix $F-UV^T$. In this case, $(F-UV^T)_{ij}$ measures the difference in the actual and predicted feedback for movie j by user i. We are primarily interested in the magnitude of this difference, in addition to penalizing completely incorrect guesses more-so than somewhat-incorrect guesses. Therefore, in order to quantify the total loss of information in our matrix factorization model, we use the Frobenius norm of the residual matrix $E := F-UV^T$, defined as

$$\begin{aligned} J(U,V) &= \frac{1}{2}\|E\|_F^2 \\ &= \frac{1}{2}\left\|F-UV^T\right\|_F^2 \\ &= \frac{1}{2}\sum_{i=1}^{m}\sum_{j=1}^{n}(F_{ij}-(UV^T)_{ij})^2 \end{aligned}$$

Note that we square the norm and multiply by a factor of $\frac{1}{2}$. We could have equivalently used $\|E\|_F$, but the former makes our work easier in the next section. From here, we can immediately deduce that the best U and V to approximate F are the minimizers of J. Since we wish to minimize J, J is known as our objective function. The corresponding optimization problem we are interested in solving is defined as [6]:

$$\underset{U^*\in\mathcal{F}^{m\times d},V^*\in\mathcal{F}^{n\times d}}{\arg\min} \quad J(U^*,V^*) \tag{4}$$

We have reduced our selection of U and V to finding a minimizer of the objective function J. In order to find our optimal user and item matrices, we will use gradient descent. Formally, gradient descent is an iterative optimization algorithm that allows us to minimize our objective function J with respect to the underlying parameters of J [7]. A healthy analogy for gradient descent is to imagine we are located somewhere on a mountain with the goal of traveling to a lower elevation. The fastest way for us to get to lower elevation is to take steps in the steepest direction downhill. In this analogy, the mountain represents our optimization function J, gradient descent represents our "strategy" of going downhill to find the lowest elevation possible, and the lowest elevation represents our minimizer for the objective function. We will soon see that in this analogy, the size of our steps downhill represents what is known as the learning rate. For our objective function, the minimizer is simply the the entries of U and V. We iteratively update U and V with the update equations defined as

$$U_{ij} = U_{ij} - \alpha\nabla_{U_{ij}}J(U,V) \tag{5}$$

$$V_{ij} = V_{ij} - \alpha\nabla_{V_{ij}}J(U,V) \tag{6}$$

In these update equations, U_{ij} denotes the (i,j) entry of the current user matrix U, V_{ij} denotes the (i,j) entry of the item matrix V, α denotes the learning rate, and $\nabla_{U_{ij}},\nabla_{V_{ij}}$ denote the value of the gradient of J for U_{ij} and V_{ij}. We perform these update equations on each entry of U and V, since we are updating U and V collectively during each iteration of gradient descent. We now calculate the partial derivatives of J to further develop our update equations. For ease of notation, define our user and item matrices as

$$U = \begin{bmatrix} \boldsymbol{u}_1 \\ \boldsymbol{u}_2 \\ \vdots \\ \boldsymbol{u}_m \end{bmatrix}, \boldsymbol{u}_i = (u_{i1},u_{i2},\ldots,u_{id}) \tag{7}$$

$$V = \begin{bmatrix} \boldsymbol{v}_1 \\ \boldsymbol{v}_2 \\ \vdots \\ \boldsymbol{v}_n \end{bmatrix}, \boldsymbol{v}_i = (v_{i1},v_{i2},\ldots,v_{id}) \tag{8}$$

$$\begin{aligned} \frac{\partial}{\partial u_{i_0j_0}}J(U,V) &= \frac{\partial}{\partial u_{i_0j_0}}\frac{1}{2}\sum_{i=1}^{m}\sum_{j=1}^{n}(F_{ij}-(UV^T)_{ij})^2 && (9) \\ &= (F_{i_01}-\boldsymbol{u}_{i_0}\boldsymbol{v}_1^T)(-v_{1j_0})+\ldots+(F_{i_0n}-\boldsymbol{u}_{i_0}\boldsymbol{v}_n^T)(-v_{nj_0}) && (10) \\ &= -\sum_{j=1}^{n}(F_{i_0j}-\boldsymbol{u}_{i_0}\boldsymbol{v}_j^T)(v_{jj_0}) && (11) \\ &= -\sum_{j=1}^{n}E_{i_0j}(v_{jj_0}) && (12) \end{aligned}$$

$$\begin{aligned} \frac{\partial}{\partial v_{i_0j_0}}J(U,V) &= \frac{\partial}{\partial v_{i_0j_0}}\frac{1}{2}\sum_{i=1}^{m}\sum_{j=1}^{n}(F_{ij}-(UV^T)_{ij})^2 && (13) \\ &= (F_{1j_0}-\boldsymbol{u}_1\boldsymbol{v}_{j_0}^T)(-u_{1i_0})+\ldots+(F_{mj_0}-\boldsymbol{u}_m\boldsymbol{v}_{j_0}^T)(-u_{mi_0}) && (14) \\ &= -\sum_{i=1}^{m}(F_{ij_0}-\boldsymbol{u}_i\boldsymbol{v}_{j_0}^T)(u_{ii_0}) && (15) \\ &= -\sum_{i=1}^{m}E_{ij_0}(u_{ii_0}) && (16) \end{aligned}$$

Hence, the update equations 5 and 6 can be rewritten as

$$U_{i_0j_0} = U_{i_0j_0} + \alpha\sum_{j=1}^{n}E_{i_0j}(v_{jj_0}) \tag{17}$$

$$V_{i_0j_0} = V_{i_0j_0} + \alpha \sum_{i=1}^{m} E_{ij_0}(u_{ii_0}) \tag{18}$$

Finally, we can vectorize our update equations to get the update equations in terms of U and V instead of their individual entries. Doing so yields the following update equations:

$$U = U + \alpha EV \tag{19}$$

$$V = V + \alpha E^T U \tag{20}$$

```
def GD(F, alpha=0.0005, K=50, d=2):
    m = F.shape[0]
    n = F.shape[1]
    U = np.random.rand(m, d)
    V = np.random.rand(n, d)
    cost = []
    for i in np.arange(1, K):
        E = F - (U @ V.T)
        iter_cost = calc_loss(E)
        cost.append(iter_cost)
        U_new = U + (alpha * (E @ V))
        V_new = V + (alpha * (E.T @ U))
        U = U_new
        V = V_new

    return (U, V, cost)
```

2.4.2 Matrix Factorization with Regularization

So far, we have defined our gradient descent algorithm which we can use to train an unconstrained matrix factorization model. While this is a good initial approach, our current model is prone to overfitting. Specifically, during the training process, there is nothing stopping our model from ”forcefitting” itself to the patterns and random noise accompanied by the training data. Consequently, our model will poorly generalize to any unseen data. For instance, in our matrix factorization model's current state, large values of the coefficients in UV^T encourage complexity in our model. To illustrate, consider $(UV^T)_{ij}$, defined as

$$(UV^T)_{ij} = \sum_{k=1}^{d} u_{ik}v_{jk} \tag{21}$$

It's evident that our estimate of F_{ij} is a linear combination of the coefficients in column i of U and row j of V^T. When these coefficients are large, our objective function J is large as well. Consequently, when training our matrix factorization model, gradient descent will tend to step in the direction that minimizes the largest coefficients! As a resolution, one technique that can be employed in order to negate the chance of overfitting is to add regularization: a technique to reduce the tendency of overfitting at the expense of added bias in our model [1]. Informally speaking, regularization expresses the idea that our final model should exhibit some degree of smoothness. This smoothness is achieved by adding a penalizing (regularizing) term to our objective function that quantifies the complexity of our model. With regards to our current matrix factorization model, the regularizing term grows when the coefficients of U and V are large. Now that we understand the goal of regularization, we must consider which candidate term would work best with our objective function. Since we wish the regularizing term to be large when the coefficients of U or V are large, we define our regularizing term R as the following:

$$\begin{aligned} R(U,V) &= \frac{\lambda}{2}(\sum_{i=1}^{m}\sum_{j=1}^{d} u_{ij}^2) + \frac{\lambda}{2}(\sum_{i=1}^{n}\sum_{j=1}^{d} v_{ij}^2) && (22)\\ &= \frac{\lambda}{2}(\|U\|_F^2 + \|V\|_F^2) && (23) \end{aligned}$$

Again, note that we squared the Frobenius norm and multiplied by a factor of $\frac{1}{2}$ in order to make our lives easier when finding the update equations. With our regularizing term, we can define a new objective function as follows:

$$J_R(U,V) = J(U,V) + R(U,V) \tag{24}$$

At this point, our model now successfully has a regularizing term! We have transposed our goal into solving the following optimization problem.

$$\underset{U^* \in \mathcal{F}^{m\times d}, V^* \in \mathcal{F}^{n\times d}}{\arg\min} \quad J_R(U^*, V^*) \tag{25}$$

We again must compute the gradient in order to get the new update equations with respect to J_R.

$$\begin{aligned} \frac{\partial}{\partial u_{i_0j_0}} J_R(U,V) &= \frac{\partial}{\partial u_{i_0j_0}} J(U,V) + \frac{\partial}{\partial u_{i_0j_0}} R(U,V) && (26)\\ &= -\left[\sum_{j=1}^{n} E_{i_0j}(v_{jj_0})\right] + \lambda u_{i_0j_0} && (27)\\ & && (28) \end{aligned}$$

$$\begin{aligned} \frac{\partial}{\partial u_{i_0j_0}} J_R(U,V) &= \frac{\partial}{\partial v_{i_0j_0}} J(U,V) + \frac{\partial}{\partial v_{i_0j_0}} R(U,V) && (29)\\ &= -\left[\sum_{i=1}^{m} E_{ij_0}(u_{ii_0})\right] + \lambda v_{i_0j_0} && (30)\\ & && (31) \end{aligned}$$

The update equations for J_R can now be described as follows:

$$U_{i_0 j_0} = U_{i_0 j_0} + \alpha \nabla_{U_{i_0 j_0}} J_R(U,V) \quad (32)$$

$$= U_{i_0 j_0} + \alpha \left(- \left[\sum_{j=1}^{n} E_{i_0 j}(v_{j j_0}) \right] + \lambda u_{i_0 j_0} \right) \quad (33)$$

$$V_{i_0 j_0} = V_{i_0 j_0} + \alpha \nabla_{V_{i_0 j_0}} J_R(U,V) \quad (34)$$

$$= V_{i_0 j_0} + \alpha \left(- \left[\sum_{i=1}^{m} E_{i j_0}(u_{i i_0}) \right] + \lambda v_{i_0 j_0} \right) \quad (35)$$

Finally, we can again vectorize our update equations in order to get new update equations in terms of U and V [1].

$$U = U(1 - \alpha\lambda) + \alpha E V \quad (36)$$

$$V = V(1 - \alpha\lambda) + \alpha E^T U \quad (37)$$

From these update equations, it is clear that our original formulation of gradient descent is just a special case of gradient descent with regularization. Specifically, our original form of gradient descent is the case where the regularization term has no weight ($\lambda = 0$).

```
def GD_Reg(F, alpha=0.0005, K=50,
           d=2, lam=0):
    m = F.shape[0]
    n = F.shape[1]
    U = np.random.rand(m, d)
    V = np.random.rand(n, d)
    cost = []
    for i in np.arange(1, K):
        E = F - (U @ V.T)
        iter_cost = calc_loss(E)
        cost.append(iter_cost)
        U_new = (1 - (alpha * lam)) * U
                + (alpha * (E @ V))
        V_new = (1 - (alpha * lam)) * V
                + (alpha * (E.T @ U))
        U = U_new
        V = V_new

    return (U, V, cost)
```

2.5 Training our Matrix Factorization Model

2.5.1 Train, Test, and Validation Sets

We now have our gradient descent algorithm which we can use to train a matrix factorization model. In order to train our model, we must partition our dataset into three groups: the training set, the validation set, and the test set. The training set contains the data we will use to actually train our model. However, using the training set to determine our model's performance yields overly-optimistic results. This is due to the fact that this set has already been exposed to our model during the training process; re-using training data to evaluate the performance of our model yields overly-confident results! Therefore, we must keep a subset of our data to evaluate the performance of our model. From this, it is clear why we need a training set and a test set. The validation set holds a similar (but distinct) role as the test set. Since our model contains hyper-parameters (such as the number of latent factors in our matrix factorization model), we need a reasonable manner to choose the optimal value of these hyper-parameters for our model. Unfortunately, this cannot be done directly on the test set; by choosing the optimal hyper-parameters on the test-set, we are overfitting our model to unique characteristics in the test set instead of fitting our model to the patterns representative of the population our data was sampled from [4]. We therefore use a validation set to determine what type of model performs best, in addition to the choice of parameters that perform best on this model.

2.5.2 K-Fold Cross Validation vs. Holdout Validation

Recall the heatmap illustration of our feedback matrix presented earlier. This heatmap strongly indicates the sparsity presented by the MovieLens dataset. As a result, the size of each partition of data (training set, validation set, test set) are limited to the quantity of data presented. This presents the immediate issue of underfitting our model. In other words, underfitting causes our model to fail to pick up on the patterns from the training set that are representative of the population we sampled from [8]. In instances where the quantity of data is limited, sacrificing the quantity of data represented in our training set causes underfitting, whereas sacrificing the quantity of data represented in our validation and test set provide high-variance estimates of our model's performance [9]. In order to mitigate these issues, K-fold cross validation can be performed. With K-fold cross validation, our training set encapsulates the data for the validation set, in which the training set is partitioned into K equal-sized blocks (or folds). Each fold is chosen to represent the validation set once, during which the remaining $K-1$ folds are used to train our model. After performing K-fold cross validation, we are left with K estimates for the prediction accuracy of our model. Since each element in our train set is used in the validation set exactly once, K-fold validation provides a lower-variance estimate of our model's performance, while also remaining robust against data scarcity.

In contrast, holdout validation presents a more straightforward approach to hyper-parameter tuning against a validation set. With holdout validation, we randomly sample our training, validation, and test set. In order to tune our hyper-parameters, we train our model on the training set with distinct parameter combinations, after which we benchmark our results on the validation set. The best-performing parameters are accompanied by our final model, which is then evaluated against the test set to determine performance. Since holdout validation only requires training our model once in order to determine how well a choice of parameters performs, it is less computationally expensive [10]. Given the long training time accompanied with training a matrix factorization model with gradient descent, we made the decision to include holdout validation over K-fold cross validation. For our model, we will partition our data randomly into each set. 67.5% of our data will be contained in the training set, 22.5% will be contained in the validation set, and 10% will be contained in the test set. Since our dataset is relatively sparse, The majority of our data was kept in the training set in order to avoid underfitting, in which our model fails to capture all of the underlying trends in our dataset.

2.6 Ranking

At this point, we have a fully trained matrix factorization model that is ready to use for inference. We are interested in providing recommendations for movies based on a given user, in addition to providing recommendations for movies based on a given movie. In order to accomplish this, we must explore measurements of similarity. Specifically, consider our matrix factorization model (U,V), where $UV^T \approx F$. Let E denote our embedding space: the vector space containing our user and item representations in U and V respectively. A similarity measurement is a mapping $\cdot : E \times E \to R$ that takes two vectors in our embedding space and quantifies how similar they are [11]. For our recommender system, we explore two different approaches: cosine similarity and dot product similarity.

2.6.1 Cosine Similarity

Cosine similarity is a similarity measurement that quantifies similarity based on direction. Specifically, we wish for elements in our embedding space in the same direction to be considered similar, regardless of magnitude. Let $\boldsymbol{u}, \boldsymbol{v} \in E$. Cosine similarity $\cdot_{\text{cos}} : E \times E \to R$ is then defined as

$$\boldsymbol{u} \cdot_{\text{cos}} \boldsymbol{v} = \frac{\boldsymbol{u}^T \boldsymbol{v}}{\|\boldsymbol{u}\|_2 \, \|\boldsymbol{v}\|_2} \tag{38}$$

Note that the image of cosine similarity $Im(\cdot_{\text{cos}})$ is given by the real interval $[-1,1]$. This is because the two norm of $\boldsymbol{u}$ and $\boldsymbol{v}$ act as normalizing terms. As such, cosine similarity remains magnitude invariant. This remains the greatest advantage when using cosine similarity, since its mappings encapsulate to what extent the orientation of distinct elements in our embedding space agree.

2.6.2 Dot Product Similarity

Dot product similarity is a similarity measurement that quantifies similarity based on both magnitude and direction [12]. Unlike cosine similarity, dot product similarity is not magnitude invariant, and consequently the amount by which user and item embeddings in our embedding space are loaded by all d factors plays an important role. Let $\boldsymbol{u}, \boldsymbol{v} \in E$. Dot product similarity $\cdot_{\text{dot}} : E \times E \to R$ is defined as

$$\boldsymbol{u} \cdot_{\text{dot}} \boldsymbol{v} = \boldsymbol{u}^T \boldsymbol{v} \tag{39}$$

Note that $\cdot_{\text{dot}}$ can also be defined in terms of the cosine similarity as

$$\boldsymbol{u} \cdot_{\text{dot}} \boldsymbol{v} = \|\boldsymbol{u}\|_2 \, \|\boldsymbol{v}\|_2 \, (\boldsymbol{u} \cdot_{\text{cos}} \boldsymbol{v}) \tag{40}$$

From 40, it becomes clear why dot product similarity also depends on the magnitude of our user and item embeddings.

2.6.3 User and Item Ranking

User and item ranking forms the backbone for our recommender system. With user ranking, given any user our model was trained on, movie recommendations can be provided with respect to that user. Similarly, with item ranking, given a movie our model was trained on, we can provide recommendations with respect to that movie. Both user and item ranking can be approached by using a similarity measure in order to determine the order of our ranking. For instance, let (U,V) denote our matrix factorization model, defined as in (7) and (8). Furthermore, let $\cdot : E \times E \to R$ denote our chosen similarity measure (either cosine or dot product similarity). User-based rankings with respect to $\boldsymbol{u}_i$ are then defined as

$$R_{\boldsymbol{u}_i} := \{\boldsymbol{u}_i \cdot \boldsymbol{v}_j\}_{j=1}^{n} \tag{41}$$

Similarly, item-based rankings with respect to $\boldsymbol{v}_j$ are defined as

$$R_{\boldsymbol{v}_j} := \{\boldsymbol{v}_j \cdot \boldsymbol{u}_i\}_{i=1}^{m} \tag{42}$$

In order to obtain the most reasonable recommendations, the computed rankings can then be sorted in decreasing order.

2.6.4 Baseline Model

Our motivation was to implement a model using collaborative filtering. By using a matrix factorization model, we are able to generalize our model to serve as a recommender system for user and item ranking across multiple datasets. The accompanying section contains an analysis of our model's performance after tuning hyper-parameters with holdout validation.

Initially, we hypothesized tuning the hyper parameters of our model with grid search cross validation [3]. However, despite receiving the performance optimization provided from vectorizing our update equations for gradient descent, the training time for our model was not negligible. As

[3]The process of grid search cross validation is explained further in our conclusion.

Learning Rate	Iterations	Latent Factors	Regulator
0.00001	40	5	0.001

Table 1. Baseline Hyper-Parameters

a compromise, we tuned the hyperparameters of our model using an "independent grid search", in which we perform grid search cross validation for each parameter. For instance, in order to determine the best learning rate, a discrete set of learning rates was used in the grid search cross validation process, after which the best performing learning rate was accepted for our final model. Despite the fact that our approach for parameter tuning fit the time constraint, this approach required the strong assumption of independence among hyper-parameters, and is discussed further in the conclusion. With regards to the best performing number of latent factors, a large number of latent factors was not needed for the best performing model. With only five, we could achieve a small validation loss. This is reasonably explained by our initial belief that, due to the sparsity of the MovieLens dataset, the feedback matrix F has low-rank structure.

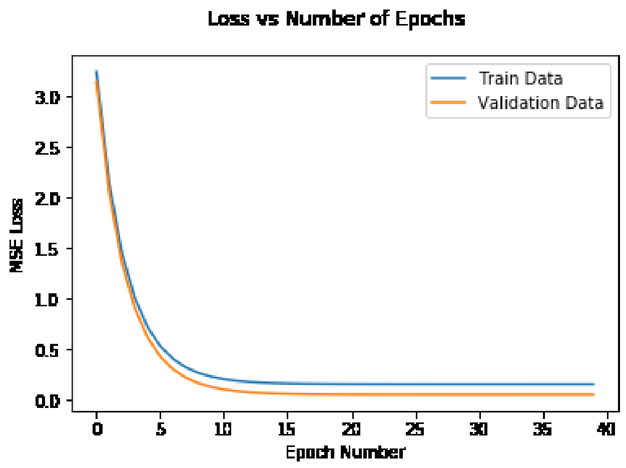


Fig. 9. Train and Validation Loss of the Baseline Model for 40 epochs

Figure 9 displays the optimal choice of hyper-parameters after tuning. Accompanied by these parameters is a plot displaying the training and validation loss of our matrix factorization model with the hyper-parameters shown in figure 9. We observed that the loss for the training and validation sets start around 3 and quickly converge after the first ten epochs. This is apparent by the fact that our loss forms an "elbow" before the first ten epochs of training. Consequently, this is where our model's convergence halted. The long-tailed behavior of the remaining epochs show that our model failed to minimize our objective function afterwards [4].

[4]In this context, epochs can be interpreted as iterations.

Table 2 presents the loss obtained from the test data set after performing matrix factorization. We observe that the loss is relatively smaller than the validation loss, which indicates our model generalizes well. However, it is also important to analyze the norm. This value came after the dot product between the user and item embeddings, and after getting the norm from this matrix. After tuning, we observed that this value was the smallest with relatively minimal loss on the test set. In later sections, we discuss that we do not want the the norm to be too small, since a small norm indicates our recommendation system might not give good recommendations, regardless of what similarity measure is used. Furthermore, figure 10 contains our reconstructed feedback matrix from our matrix factorization model with regularization after tuning the hyperparameters. Note that a large portion of this heatmap is close to zero, which indicates the regularizing term significantly penalized large values in UV^T during training. One reasonable source of error is that the independence assumption of our hyper-parameters was not reasonable to make. In order to remedy this, grid search cross validation can be performed in order to exhaustively find the best performing choice of hyper-parameters. We address this further in our conclusion.

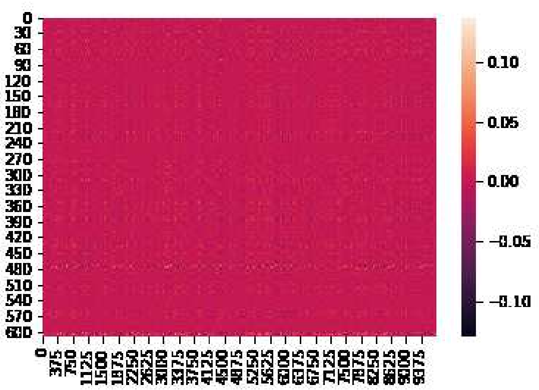


Fig. 10. Heatmap of feedback matrix approximation UV^T after tuning

MSE	$\|\|UV^T\|\|_F$
0.022443	20.758000

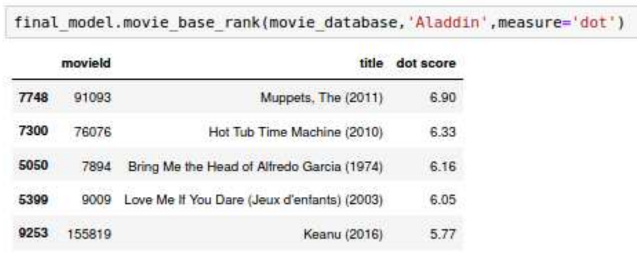
Table 2. Test Loss and Norm of the Baseline Model

Given our tuned model (U,V), table 2 presents the mean squared error (MSE) of our model against the test set. Furthermore, $\|\|UV^T\|\|_F$ denotes the Frobenius norm of our approximation of the feedback matrix F_{test} constructed from the test set. We observed that the MSE is relatively similar to the validation loss. However, since $\|\|UV^T\|\|_F$ is relatively large, we deduce that entries in our approximation of the feedback matrix are overly confident in their tendency of rating well-received movies from the test set. Consequently, we cannot gain a strong semantic interpretation from $\left\|UV^T\right\|_F$. In other words, the entires of $\left\|UV^T\right\|_F$ cannot be interpreted as approximations of ratings of F_{test}. Instead, $\left\|UV^T\right\|_F$ serves as an unconstrained matrix that attempts to minimize the sum of residuals $\left\|F_{test} - UV^T\right\|_F$. Greater semantic interpretation can be gained from a constrained matrix factorization model. We discuss this further in our conclusion.

2.7 Recommendation System

We designed and implemented our recommendation system such that we can provide recommendations by using similarity measures such as dot product similarity or cosine similarity. After we calculate the scores, we rank our recommendations base on the k top scores, where $k \in Z^+$. The following tables display the results of our recommender system during the ranking and retrieval process.

Figure 11 presents movie-based ranking: the recommender system takes the name of a movie, the number of desired recommendations and the dot product as the chosen similarity measure. For this example, we produced our movie-based ranking with the movie query ”Aladdin”, and then found the query embedding of this movie. In order to calculate the rankings of all movie embeddings in the MovieLens dataset, we used the dot product similarity measure. Afterwards, the system orders ranking and retrieve the information of the top 5 of these movies. ”The Muppets” have the highest score among the rest, and consequently have the highest rank. Note that the scores produced from the dot product fluctuate around 5. Since the dot product similarity measure is proportional to the two norm of our operands, it is evident that our model's ranking and retrieval was largely based on the magnitude of each movie embedding.

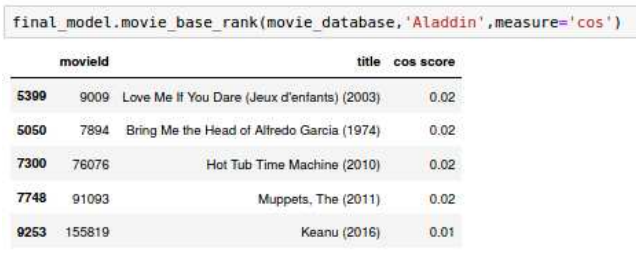


```
final_model.movie_base_rank(movie_database,'Aladdin',measure='dot')
```

	movieId	title	dot score
7748	91093	Muppets, The (2011)	6.90
7300	76076	Hot Tub Time Machine (2010)	6.33
5050	7894	Bring Me the Head of Alfredo Garcia (1974)	6.16
5399	9009	Love Me If You Dare (Jeux d'enfants) (2003)	6.05
9253	155819	Keanu (2016)	5.77

Fig. 11. Movie base rank, 5 top recommendations using dot product as similarity measure

To contrast the performance of dot product similarity, figure 12 presents movie-based ranking using cosine similarity. Our recommender system follows the same procedure as before; we only change the manner in which similarity is computed for movie embeddings. We observed that the scores produced are relatively small. This is due to the image of cosine similarity. Since this similarity measure has an image of $[0,1]$, it's normalized and solely based on the direction of our operands. Consequently, movie embeddings in the same direction as the query embedding ”Aladdin” are ranked higher. For instance, ”Love Me if You Dare” presents the strongest degree of colinearity with ”Aladdin”, since it has a cosine similarity of 0.02. The change in similarity measures demonstrates how magnitude influences the ranking decision for a recommender system.



```
final_model.movie_base_rank(movie_database,'Aladdin',measure='cos')
```

	movieId	title	cos score
5399	9009	Love Me If You Dare (Jeux d'enfants) (2003)	0.02
5050	7894	Bring Me the Head of Alfredo Garcia (1974)	0.02
7300	76076	Hot Tub Time Machine (2010)	0.02
7748	91093	Muppets, The (2011)	0.02
9253	155819	Keanu (2016)	0.01

Fig. 12. Movie base rank, 5 top recommendations using cosine as similarity measure

3 Conclusions

After completing our recommender system, we have that reasonable predictions are made for both user and item rankings. However, the scope of our project did not include a manner in which we could quantify how well these recommendations performed on actual users. Instead, our model's performance was solely measured based on the accuracy of estimated ratings. As such, future scope of work for this project includes measuring our model's performance with online methods such as A/B testing, in which the direct impact of our recommender system is evaluated [9]. Furthermore, while our recommender system makes reasonable predictions, we have that unconstrained matrix factorization leaves little room for semantic interpretability. In other words, since our objective function for matrix factorization was unconstrained to all real numbers, for our matrix factorization model (U,V), we cannot interpet entries of UV^T as approximated reviews in our feedback matrix F. In order to remedy this, the future scope of work also includes retraining our model using non-negative matrix factorization, which provides a strong degree of interpretability for user-item interactions.

Finally, our approach for hyperparameter tuning was ad hoc; in order to find the best choice of hyperparameters, we ended up running independent grid searches for each hyperparameter, in which all but one hyperparameter remains fixed. One solution to choose the most optimal hyperparameters is grid search cross validation. To motivate grid search cross validation, suppose we have hyperparameters $\{p_i\}_{i=1}^n$ which we must tune for our model. For each hyperparameter p_i, further suppose we have a discrete set of candidate values P_i containing the possible hyperparameter values for p_i. All possible permutations of parameters can be viewed as the set of n-tuples $P := P_1 \times \ldots \times P_n$. Since each P_i is discrete, P is discrete. Grid search cross validation aims to find the best performing tuple of hyperparameters in P. In order to do so, for each $p \in P$, K-fold cross validation is performed with model parameters specified by p. The parameter choice from grid search cross validation is the parameter tuple p^* with the highest training accuracy across all folds [13]. Therefore, the scope of future work would also include implementing grid search cross validation.

References

[1] Aggarwal, C., 2016. *Recommender Systems*. Springer, New York, NY.

[2] Melville, P., Mooney, R., and Nagarajan, R., 2002. "Content-boosted collaborative filtering for improved recommendations". *Association for the Advancement of Artificial Intelligence*, p. 1.

[3] Antaris, S., Demirtsoglou, G., Zisopoulos, X., and Karagiannidis, S. "Content-based recommendation systems".

[4] Friedman, J., Hastie, T., and Tibshirani, R., 2008. *The Elements of Statistical Learning*. Springer, New York, NY.

[5] Cheng, W., Shen, Y., Zhu, Y., and Huang, L., 2018. "Explaining latent factor models for recommendation with influence functions".

[6] Dong, X., Yu, L., Wu, Z., Sun, Y., Yuan, L., and Zhang, F., 2017. "A hybrid collaborative filtering model with deep structure for recommender systems". *Association for the Advancement of Artificial Intelligence*.

[7] Ruder, S., 2016. "An overview of gradient descent optimization algorithms".

[8] Jabbar, H., and Khan, R., 2015. "Methods to avoid overfitting and underfitting in supervised machine learning (comparative study)". *AET*.

[9] Bishop, C., 2006. *Pattern Recognition and Machine Learning*. Springer, New York, NY.

[10] Raschka, S., 2018. "Model evaluation, model selection, and algorithm selection in machine learning".

[11] The cosine similarity algorithm. On the WWW. URL `https://neo4j.com/docs/graph-algorithms/current/labs-algorithms/cosine/`.

[12] Bachrach, Y., Finkelstein, Y., Bachrach, R., Katzir, L., Koenigstein, N., Nice, N., and Paquet, U., 2020. "Speeding up the xbox recommender system using a euclidean transformation for inner-product spacesctor classifier and grid search cross validation hyperparameter tuning". *ACM*.

[13] Aqib, M., Abas, H., Ismail1, N., Azah, N., Ali, M., Tajuddin, S., and Tahir, N., 2020. "Agarwood oil quality classification using support vector classifier and grid search cross validation hyperparameter tuning". *International Journal of Emerging Trends in Engineering Research*.