|  |  |
| --- | --- |
| Name | Ujwal Kavalipati |
| Kaggle User Name | Ujwal K |
| Student ID | 30309832 |

# TASK- 1 : Recommender System Challenge

In this task, we are asked to recommend top 10 items for the user based on the interaction data given to us. There are 2 types of data, Explicit and Implicit feedback. Explicit data is where user explicitly gives rating to the content. Implicit data is where information about user interests is collected by the system, it can be number of minutes online, number of views, likes etc. The data given to us is implicit data. The data given to us is split into test, train and validation data. Recommending content is an important task in many information systems. For example online shopping websites like Flipkart give each customer personalized recommendations of products that the user might be interested in. Other examples are video streaming services like Netflix, Amazon Prime, YouTube that recommend movies to customers based on their interests. In this task, we are given data from an online social media platform.

There is a package called ‘implicit’ which is designed to work with implicit data. There are different models in it and 3 models are implemented along with a ensemble model. Let us go through one by one.

**AlternatingLeastSquares(ALS)**

ALS The alternating least squares (ALS) [[1]](#endnote-1)algorithm factorizes a given matrix RR into two factors UU and VV such that R≈UTVR≈UTV. The unknown row dimension is given as a parameter to the algorithm and is called latent factors. Since matrix factorization can be used in the context of recommendation, the matrices UU and VV can be called user and item matrix, respectively. The main basic idea is to take a large matrix and factor it into much smaller ones. ALS follows an iterative optimisation approach which becomes closer and closer to a factored representation of our original data. The data we have already has preferences set.

The rate of which our confidence increases is set through a linear scaling factor **α**.

**α** value between 15 and 30 gave good results to me. Especially **α**=30 gave decent results.

For building the ALS model, we have first combined train data and validation data into a new data frame. We have done that as train data has only interaction ‘1’ and validation has ‘0’ as well. Combining the data helps us to build a good model.

Then, we have to create two matrices, one for fitting the model (content-person) and one for recommendations (person-content)

ALS fits the model on the sparse-content-person matrix (sparse-item-user) multiplied by **α** which is 2174 2239 matrix of type numpy.int64(datatype can be float as well).

We implement the als model with factors=8, regularization=0.1 and iterations=2500 as that gave the best results compared to other als models with different hyper parameter settings.

model1 = implicit.als.AlternatingLeastSquares(factors=8, regularization=0.1, iterations=2500)

where

factors=The number of latent factors to use for the underlying model. It is equivalent to the dimension of the calculated user and item vectors. Setting 8 gave the best results.

Regularization factor : Tune this value in order to avoid overfitting or poor performance due to strong generalization. If this parameter is high, models with high complexity can be ruled out, if it is low, models with high training error can be ruled out. We selected a value of 0.1 .

We use a metric called NDCG to compute our model efficiency. A recommender returns some items and we’d like to compute how good the list is. Each item has a relevance score, usually a non-negative number. That’s gain. For items we don’t have user feedback for we usually set the gain to zero.

Now we add up those scores; that’s cumulative gain. We’d prefer to see the most relevant items at the top of the list, therefore before summing the scores we divide each by a growing number (usually a logarithm of the item position) - that’s discounting - and get a DCG.

DCGs are not directly comparable between users, so we normalize them. The worst possible DCG when using non-negative relevance scores is zero. To get the best, we arrange all the items in the test set in the ideal order, take first K items and compute DCG for them. Then we divide the raw DCG by this ideal DCG to get NDCG@K, a number between 0 and 1.

iterations: Maximum number of iterations it should take.

Below table shows few experiments which I made.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| alpha | Factors | Regularization | Iterations | NDCG |
| 30 | 8 | 0.1 | 2500 | 0.21418 |
| 30 | 16 | 0.15 | 2500 | 0.18147 |
| 30 | 16 | 0.1 | 2000 | 0.17987 |
| 25 | 8 | 0.15 | 2500 | 0.15473 |
| 30 | 5 | 0.15 | 30 | 0.19762 |
| 30 | 6 | 0.1 | 3000 | 0.20637 |
| 30 | 8 | 0.1 | 3000 | 0.21159 |

There is a function called recommend() in implicit.als.AlternatingLeastSquares package which takes user-id, csr matrix and which return the top n recommendations for every user. I have used that with my best hyper parameters and got a maximum NDCG score of 0.21418.

**LogisticMatrixFactorization**

A collaborative filtering recommender model that learns probabilistic distribution whether user like it or not. Our model takes a approach by factorizing the observation matrix R by 2 lower dimensional matrices Xn x f and Ym x f  where f is the number of latent factors. The rows of X are latent factor vectors that represent a user’s taste while the columns of YT are latent factor vectors that represent an item’s style, genre, or other implicit characteristics. We take a probabilistic approach to get our results.

LMR model implementation is similar to ALS.

We fit the model on the sparse-content-person matrix (sparse-item-user) multiplied by **α** which is 2174 2239 matrix of type numpy.int64(datatype can be float as well).

**α** is the rate we increase our confidence with the interaction.

We fit the model with

**α=15**

model2 = implicit.lmf.LogisticMatrixFactorization(factors=5, regularization=0.1, iterations=1500)

where factors, regularization, iterations are described above. Many experiments were made with lmf model and the maximum NDCG score achieved was 0.16228

We have used the model.recommend() function which takes user-id, csr matrix and returns top N recommendations.

I have noticed that increase in number of factors beyond 15 affected the NDCG score and it decreased.

**BayesianPersonalizedRanking**

BPR for personalized ranking uses the maximum posterior estimator derived from a Bayesian analysis of the problem. It uses a generic learning algorithm for optimizing models with respect to BPR-Opt[[2]](#endnote-2). The learning method is based on stochastic gradient descent with bootstrap sampling.It is a recommender model that learns a matrix factorization embedding based off minimizing the pairwise ranking loss.

BPR model implementation is similar to ALS.

We fit the model on the sparse-content-person matrix (sparse-item-user) multiplied by **α** which is 2174 2239 matrix of type numpy.int64(datatype can be float as well).

**α** is the rate we increase our confidence with the interaction.

We fit the model with **α=15**

model3 = implicit.bpr.BayesianPersonalizedRanking(factors=16, regularization=0.1, iterations=1500)

model3.fit(data)

where factors, regularization, iterations are described above in ALS. BPR was giving very low NDCG scores compared to other models and maximum achieved was 0.05214

We have used the model.recommend() function in implicit.bpr.BayesianPersonalizedRanking class which takes user-id, csr matrix and returns top N recommendations.

Some experiments with BPR are as follows:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| alpha | factors | Regularization | iteration | NDCG |
| 30 | 8 | 0.1 | 1500 | 0.04924 |
| 15 | 16 | 0.1 | 1500 | 0.05214 |
| 15 | 5 | 0.15 | 1000 | 0.05518 |

**LMF+ALS(Ensembling LMF and ALS):**

The next technique I used was to try was mixing two models and taking their results commonly called as ensembling. So LMF model was fitted first using this

model\_ensmb1 = implicit.lmf.LogisticMatrixFactorization(factors=8, regularization=0.1, iterations=1500)

model\_ensmb1.fit(data)

Then ALS model was fitted next with

model\_ensmb2 = implicit.als.AlternatingLeastSquares(factors=8, regularization=0.1, iterations=2500)

model\_ensmb2.fit(data)

with **α=30.**

The other hyper parameters which were chosen were factors, regularization and iterations. All 2174 items were taken for every user for each model and their mean of scores is derived. Then the items were sorted in the descending order based on highest mean and top 10 items were taken.

With this approach, a NDCG score of 0.15296 is achieved.

**Model Comparision**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Model Name | alpha | factors | Regularisation | iteration | NDCG |
| ALS | 30 | 8 | 0.1 | 2500 | 0.21418 |
| LMF | 15 | 5 | 0.1 | 1500 | 0.16228 |
| BPR | 15 | 16 | 0.1 | 1500 | 0.05214 |
| LMF+ALS | 30 | 8 | 0.1 | 1500(lmf),2500(als) | 0.15196 |

Thus, we can see that ALS gave the best score of 0.21418 and it is selected as best model which is submitted to Kaggle and named as 30309832.csv.

I thought ensembling will give a better result but it is less than that of ALS. LMF also performed averagely compared to ALS with a score of only 0.16228. BPR performance is not at all decent with our data and it gave only 0.05214 score which is the least.Thus, we can say that ALS performs better with many hyper parameter settings compared to other models for our dataset.

# Task 2: Node Classification in Graphs

This task basically involves doing node classification in a graph dataset. We are given a citation network. In this network, each node is paper, an edge indicates the relationship between two papers. As the network has extremely sparse network structure, we also provide text information for each paper, i.e., the title of each paper.

The basic steps which are done here:

1. Read the graph from the adjacency list into networkx
2. Train node2vec and get the vector for each node. Now we have a matrix X (where each row is a node vector).
3. Get the labels for each node. Now we have a vector Y
4. Train-test splits for the (X, Y) into (Xtrain, Ytrain) and (Xtest, Ytest)
5. Build a classifier on (Xtrain, Ytrain)
6. Test on (Xtest, Ytest) and get the accuracy.

The graph is read as a networkx graph with the help of networkx.parse\_adjlist() function.Then, we have to get the vectors for each node in the graph. We are using Node2Vec for this step.

Node2Vec:

**node2vec** is an algorithmic framework for representational learning on graphs. Given any graph, it can learn continuous feature representations for the nodes, which can then be used for various downstream machine learning tasks.[[3]](#endnote-3)

The node2vec framework learns low-dimensional representations for nodes in a graph through the use of random walks through a graph starting at a target node. It is useful for a variety of machine learning applications. Besides reducing the engineering effort, representations learned by the algorithm lead to greater predictive power. node2vec follows the intuition that random walks through a graph can be treated like sentences in a corpus. Each node in a graph is treated like an individual word, and a random walk is treated as a sentence[[4]](#endnote-4)

We have run node2vec with following parameters:

pre-compute the probabilities:

node2vec = Node2Vec(g, dimensions=64, walk\_length=30, num\_walks=100, workers=1)

embed the nodes:

model\_nd = node2vec.fit(window=10, min\_count=1, batch\_words=4)

We have selected the dimensions to be 64 as it gives good results.

The number of nodes in each walk(walk length) is 30 and number of walks is 100. Workers are set to 1 as it is run in Windows on a CPU, more workers means parallel execution which is faster.

`diemensions` and `workers` are automatically passed (from the Node2Vec constructor) to node2vec.fit().

Once we get the features for vectors, we read the labels, do the train-test split and build the classifiers.

The accuracies for different classifiers are as follows:

|  |  |
| --- | --- |
| Logistic Regresssion | 54.3 |
| Bernoulli | 37.6 |
| Linear SVC | 55.18 |
| Random Forest | 59.12 |

Let us try to improve the classification by using the titles given to each nodes. We shall take all the titles, create a corpus , convert the words into feature vectors, add them to our node vectors which we derived earlier and then fit the models and compute the accuracies.

We are doing some pre-processing on our titles which basically involves

* converting words into lower case : There would be uniformity when generating feature vectors
* tokenization: converting the text into tokens by removing punctuation
* removing stop words: Stop words are frequently occurred words in our corpus which don’t add any value to the features and they have to be removed
* removing numbers: Since we are dealing only with text, numbers are removed
* single character removal- tokens with single character are removed

After this pre-processing, TF-IDF vectorizer is defined and we call the fit\_transform() method on our corpus. We also use WordNetLemmatizer to lemmatize the words.

TF-IDF:

tf–idf or TFIDF, short for term frequency–inverse document frequency, is a numerical statistic that is intended to reflect how important a word is to a document in a collection or corpus. It is often used as a weighting factor in searches of information retrieval, text mining, and user modeling. The tf–idf value increases proportionally to the number of times a word appears in the document and is offset by the number of documents in the corpus that contain the word.[[5]](#endnote-5)

After generation of TF-IDF vectors from titles of each paper, we stack the features of Node2Vec and TF-IDF and do the train-test split with 20% training data.

Then, same algorithms are built which are made initially and their accuracies are as follows:

|  |  |
| --- | --- |
| Logistic Regresssion | 75.6 |
| Bernoulli | 54.6 |
| Linear SVC | 79.04 |
| Random Forest | 70.23 |

Thus, we can say that using Node embedding along with TF-IDF(text feature generation) gave us better results than only use Node2Vec embedding approach. Therefore, combining different features help us to know the latent features of our data and we can perform many machine learning tasks. Node2Vec is a very powerful algorithm.

1. <https://ci.apache.org/projects/flink/flink-docs-release-1.2/dev/libs/ml/als.html#:~:text=Description,and%20is%20called%20latent%20factors.> [↑](#endnote-ref-1)
2. <https://arxiv.org/ftp/arxiv/papers/1205/1205.2618.pdf> [↑](#endnote-ref-2)
3. <https://snap.stanford.edu/node2vec/> [↑](#endnote-ref-3)
4. <https://en.wikipedia.org/wiki/Node2vec> [↑](#endnote-ref-4)
5. <https://en.wikipedia.org/wiki/Tf%E2%80%93idf> [↑](#endnote-ref-5)