Task 1: Recommender System Challenge

Recommender systems are algorithms that are widely used in the information era. The most popular use of these systems is by Netflix, Amazon etc. These corporations have access to a lot of user data thanks to their streaming services and e-commerce websites. The prediction system aims to guess the rating or preference a user would give to an item. The item can be anything like a movie or a food item or even a friend on a social media site.

The data these systems work on is user generated data. It comes in two forms. **Explicit and Implicit** rating.

Explicit feedback refers to the ratings given by the users for any item. This can be a movie on IMDB or a product on Amazon. This type of data is very hard to come by as people would have to make the effort to give the rating. On the other hand **Implicit** feedback is not specifically directed towards a rating. Implicit feedback refers to the user's actions. A user while online clicks on a number of things and we automatically move towards things we like. For example a TV show that a user likes to watch multiple times etc. This data is easy to come by as it doesn't require any action from the user.

The data we are given shows the interactions of users with different items. So what we have is implicit data. The most popular algorithm for dealing with recommender systems is **Matrix Factorization.**

Matrix Factorization

The main data involved here is the **user item interactions**. It is a matrix showing the different ratings given by users to different items. This matrix is decomposed in to two smaller matrices. These smaller matrices introduce a new factor or latent variable. We can call this latent variable a genre.

Say there are m users and n items. The user item interaction matrix will have the dimensions (m x n). If we have millions of users that will lead to a huge matrix which is very difficult to work with. Matrix factorization decomposes that matrix into two matrices of dimensions (m x k) and (n x k).

Here k is the number of hidden dimensions that we specify. The first matrix is a user feature matrix. Each row of the matrix represents how interested the user is in each genre. The second matrix is the item features. This matrix shows how each item is related to each genre.

How the code works

The process is very simple. We have a user item interaction matrix that we generate from thee given data. This is a sparse matrix with a lot of 0s. We add 1 to the entire matrix to make the non-interacted items 1. And then change the items with value greater than 1 to 0.

For a given user we get the user feature vector and dot product it with all the item feature vectors. This will give us the affinity of the user with all the items. This item list can be ordered according to decreasing affinity between the user and the item. This will generate the recommendations list.

Logistic Matrix Factorization

The most commonly used optimization method is minimizing the RMSE. LMF uses probability. The probability of a user interacting with a new item is predicted using a logistic function parametrized by the dot product of the user and item vectors.

Factors	Regularization	Iterations	Accuracy
20	0.1	50	10.8
20	0.1	100	13.2
20	0.005	100	11.4
8	0.5	300	16.8

Alternating Least Squares

Another optimization method is minimising the mean square error. But the mean square error here has two unknown variables. The user and item features. Fixing one and optimizing the other makes this a simple linear regression problem. ALS is a two step iterative process that minimizes two loss functions alternatively. It keeps the user matrix fixed and runs the gradient descent algorithm on the item matrix and then vice versa.

Factors	Regularization	Iterations	Accuracy
20	0.01	100	18.9
10	0.01	100	20.3
8	0.1	300	21.3

Analysis

Of the two algorithms ALS performed better with the highest accuracy at 21.3%. In both the models reducing the number of factors helped. When the number of factors was reduced from 20 to 8 the accuracy for LMF jumped 6% and for ALS there was a 3% improvement

From the above experiments we see that ALS had more than 20% accuracy even with 100 iterations. This is because ALS converges quicker than other algorithms. Looking at LMF we see that it is a gradual climb with increasing iterations.

Task 2: Node Classification in Graphs

Node Classification is the process of classifying a large graph of nodes into subgraphs with labels. These labels represent characteristics of the nodes. The nodes could be anything. They could represent individual users in a social medial platform or documents in a corpus that are connected by edges to similar articles etc

For this task we are given a set of document titles. Each document has an ID and a title. Some of these articles are similar to others and these relationships between articles is given in the adjacent list. Each entry in the list has a node which is the centre node and the remaining nodes in the entry are the centre node's adjacent nodes. Using this adjacent list we can build a graph and perform node classification. We can do this in two ways. We have the node IDs and the title associated with each node. So we can try both **Graph Embedding** and **Text Classification.**

Graph Embedding

Node2Vec is very similar to Word2Vec. The skip gram model of word2vec takes the text and tries to predict the surrounding context words using the centre word. This is how node2vec is also supposed to work. But we don't have any sentences in the graph. Just a bunch of nodes connected to each other. What node2vec does is use a process called random walks. From every node a random walk is generated for a pre-determined number of steps. This number of steps in each walk and the total number of walks are given to the model as parameters. These walks are then used as equivalent sentences and the model learns the data.

The algorithm of choice is Linear SVC. The parameters tried are detailed below.

Result Analysis

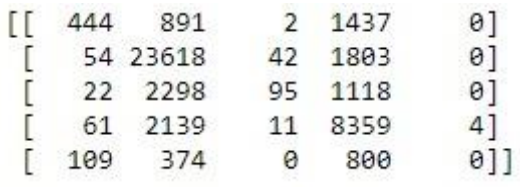
The table below shows the different parameters used for the SVC model

Model	Dimensions	Walk Length	Number of Walks	Neighbour Window	Accuracy
Linear SVC	20	15	50	10	73.7
Linear SVC	20	15	50	8	72.8
Linear SVC	20	15	50	20	74.4
Linear SVC	20	20	50	20	74
Linear SVC	30	15	50	30	75.4

The number of random walks and the walk length are given while initializing the model. Since the graph is sparsely connected the parameters aren't that high. When fitting the model to the data we give it the neighbour window. This is the word2vec equivalent of word

window for the sentences. Instead of surrounding words we're dealing with neighbouring nodes.

Increasing the walk lengths and the neighbour window improved the classification efficiency. The **C parameter** in the SVC specifies how much we want to avoid misclassifying an example while training. SVM tries to find a hyperplane differentiating the different points. A high C value chooses a small margin that will correctly classify all the point and a low C value increases the margin so that some points might be misclassified but will lead to a smoother model with less variance. For the current task C values from 0.001 to 1 were tried and 0.001 gave the best output



The **confusion matrix** shows the number of documents classified or misclassified. In the confusion matrix shown below the rows are the True values starting from 0 to 4 and the columns are the predicted values. Therefore, this means the diagonal elements are the correctly classified items. We can see from the confusion matrix below that the label 4 items have all been misclassified.

Text Classification

Text classification is the use of machine learning algorithms to classify different text like documents in a corpus into different classes. A popular way to do this is to extract word features from the text.

TF-IDF is a score that represents each word in the corpus. If a word appears a lot in a document the TF or term frequency increases. High TF means the word is important. If the word appears a lot in other documents then the IDF or inverse document frequency will decrease. High IDF means that the word appears a lot in all the documents and is hence a word of less significance.

The TFIDF feature extractor scores all the words in the corpus and we get a TF IDF feature matrix. This matrix is used to train classification models like logistic regression or linear SVC.

Result Analysis

The table below shows a few of the parameters that were tried.

Model	Min DF	N Gram	Max DF	Regularization	Accuracy
Logistic Regression	2	Bigrams	1	-	75.5
	5	Bigrams	1	-	74.3
Linear SVC	2	Bigrams	1	-	75.4
	5	Bigrams	1	-	72.9
Linear SVC	2	Bigrams	1	0.2	76.3
Linear SVC	1	Bigrams	1	0.2	77.03

Both Logistic Regression and Linear SVC had similar performances. There are a few interesting points to be noted here. **Max DF** is a parameter of TF IDF vectorizer. It ranges from 0 to 1. It specifies the maximum number of documents a word can be found in before it is removed. Since each title is only a few words long there may not be many words that occur in all the documents. Max DF values from 0.5 to 0.9 were tried and it made NO difference.

Min DF on the other hand performed best at 1. This also makes sense because we are dealing with titles only. Setting Min DF at 2 might eliminate a lot of rare terms that might add a lot of meaning to the corpus.

The **C parameter** in the SVC specifies how much we want to avoid misclassifying an example while training. SVM tries to find a hyperplane differentiating the different points. A high C value chooses a small margin that will correctly classify all the point and a low C value increases the margin so that some points might be misclassified but will lead to a smoother model with less variance. For the current task C values from 0.001 to 1 were tried and 0.2 was the optimal point.

```
[[4141  110  219   66   98]
 [ 214 2663  267   68    6]
 [ 570  278 2570  125    7]
 [ 225  154  236 1528    5]
 [ 629   44   94   24  635]]
```

This is the confusion matrix generated by the linear SVC model. We can see that here 635 nodes of label 4 are correctly classified. Which is much better than the node2vec model.

At 77% text classification did a better job than node embedding which only managed 74.4%.