FIT5212 ASSIGNMENT 2 DISCUSSION REPORT

PART 1: RECOMMENDER SYSTEMS

INTRODUCTION

My research and experimentation with different recommender system algorithms and libraries led me to 4 different approaches:

- Alternating Least Squares (ALS)
- Logistic Matrix Factorisation (LMF)
- Hybrid Matrix Factorisation (HMF) with WARP loss
- Neural Collaborative Filtering (NCF)

I determined that the dataset provided could be of implicit feedback type – columns consisted of user ID, item ID, and a rating of 1 for each interaction with no variance in the rating column.

Initial exploration and algorithm tuning was done through the implicit library package, where I used the ALS and LMF algorithms.

Due to the lack of variety in the data, many libraries like Surprise were not able to be considered as they only work on data that has variety (explicit feedback) such as ratings of 1-5. I then considered libraries that worked with implicit data sets and came across LightFM[1].

Discussions during the unit lecture and consultation sessions suggested that ranking recommendations could be thought of as a regression problem, so I decided to explore a Neural Collaborative Filter as well.

Out of the 4 algorithms, **Hybrid Matrix Factorisation** performed the best with a **NDCG score of 0.23947** on the Kaggle dataset.

ALTERNATING LEAST SQUARES

Given that ALS ranks recommendations without any extra information on the users or items and produced the second highest NDCG score of 0.21097 on Kaggle is as testament to the strength of the algorithm. I was really impressed, although I initially dismissed it for poor results due to my lack of understanding of the hyperparameters.

ALS was my first approach for the task, and I initially only managed to get a score of 0.12759 on Kaggle because I expected the regularisation parameter λ to be around 0.001, and α value to be 10. I only came back to trying to improve my ALS score in the last few days of the competition after I decided to look a bit deeper into how the algorithm works and what effect α, number of factors f, and regularisation parameter λ have on the algorithm's learning because I was sure that my values for them were not ideal for the dataset.

According to "Collaborative Filtering for Implicit Feedback Datasets" [2] paper, the preference for a user and item can be described with the following function:

$$p_{ui} = \begin{cases} 1, & r_{ui} > 0 \\ 0, & r_{ui} = 0 \end{cases}$$

where p_{ui} is the preference of user u for item i and r_{ui} is the predicted rating. Implicit datasets are difficult to make recommendations on, since we only have 0 and 1 values, and 0 does not indicate that the user does not like the item – only that the user has not interacted with the item. Similarly, a 1 does not necessarily indicate that the user likes the item – it could have been a mistake. Because of

[1] https://making.lyst.com/lightfm/docs/home.html
[2] Y. Hu, Y. Koren and C. Volinsky, "Collaborative Filtering for Implicit Feedback Datasets", 2008

this uncertainty, the paper introduces a confidence variable c_{ui} that measures the confidence of observing p_{ui} that is defined as:

$$c_{ui} = 1 + \alpha r_{ui}$$

This gives every observed user-item pair a minimum confidence, but as more evidence is observed for $p_{ui} = 1$, the c_{ui} value increases accordingly. The constant α controls the rate of increase, so the higher the α and the more evidence that is observed, the more confident the model is in the predicted preference so setting a high value for α would have a big impact on the model's prediction.

This is directly tied to the regularisation parameter λ which is used to ensure that the model will not overfit the training data. Because our data is a sparse matrix, and we have added a confidence variable to our rating prediction, it makes sense that our λ value needs to be quite high (**much** higher than 0.001) to ensure the model doesn't learn noise and ensures the prediction is within a reasonable space from the middle of the factor graph (as per slide 26 of lecture 8 notes).

Based on this, I changed my approach to ALS and first tried out the values as stated in the paper of $\alpha = 40,\ \lambda = 500,\ f = 200$ and achieved a score of 0.17615 on Kaggle – a huge improvement! I then optimised the function parameters by fixing the number of factors to 50, to get a higher number of "top hits" (where the top predicted item is the top item in the validation set), and produced the following results on the validation set:

Table 1 Effect of different α and λ values on ALS

λ \ α	10	25	50	75	100	125	150
50	150	133	122	130	124	117	118
100	132	163	136	126	119	113	120
150	87	164	156	151	138	125	128
200	79	136	161	159	137	134	143
250	79	133	171	168	155	148	136
300	79	134	153	176	162	154	155
350	72	131	127	167	168	174	152

There is a clear connection between the α and λ values – increasing both α and λ generally leads to higher accuracy in the model. There is a clear sweet spot for each pair of α and λ, before which the model appears to underfit, and after which model seems to overfit the data as shown by the lower accuracy.

The best performance on the validation set was achieved with $\alpha = 75,\ \lambda = 300,\ f = 50$ and this combination scored 0.21097 on Kaggle.

LOGISTIC MATRIX FACTORISATION

Logistic Matrix Factorisation (LMF) uses a similar approach to ALS in that it decomposes the user-item interaction matrix into two learned latent factor matrices. The difference between LMF and ALS is that "While [ALS] minimizes the weighted RMSE between a binary based preference matrix and the product of U and V, [LMF takes] a probabilistic approach"[3] – meaning LMF learns the probabilistic distribution of the user liking the item or not.

[3] https://web.stanford.edu/~rezab/nips2014workshop/submits/logmat.pdf

I observed the effect of α and λ on the validation set by fixing the number of factors to be 150 and saw that they had a strong influence on the "top hits" (where the top item predicted == top item in validation set). Compared to ALS, LMF seems to prefer a smaller value of α and λ:

Table 2 Effect of different α, λ and f values on LMF

λ \ α	10	25	50	75	100	125	150
10	147	125	111	76	75	49	42
25	162	150	129	137	114	112	81
50	118	147	165	161	151	136	143
100	81	120	130	123	134	147	146
150	83	83	116	119	129	121	141
200	83	83	83	115	128	125	133
250	82	83	84	82	105	129	132

LMF performed better with a lower α, more factors and a lower regularisation factor compared to ALS, possibly due to the difference in determining the rating. According to the LMF paper, "Increasing α places more weight on the non-zero entries while decreasing α places more weight on the zero entries … choosing α to balance the positive and negative observations generally yields the best results" [3]. This contrasts with the effect of α in ALS where it controls the rate of increase in confidence of a prediction, however its performance was similar to ALS.

After tuning the parameters, I used the parameters $(\alpha = 50,\ \lambda = 50,\ f = 150)$ for the test set and produced a score of 0.21011 on the Kaggle set.

HYBRID MATRIX FACTORISATION

LightFM was one of the libraries that I came across in my research and upon the first use of the library, it doubled the initial score of my ALS method. LightFM is a hybrid matrix factorisation model that combines content-based filtering with collaborative filtering by accepting the feature matrices for items and users alongside the interaction matrix.

In traditional matrix factorisation, item and user features are latent vectors that are learned by the model. LightFM works slightly differently, and as per the in-function documentation -

> *When a feature matrix is provided, it should be of shape (num_<users/items> x num_features). An embedding will then be estimated for every feature: that is, there will be num_features embeddings.*
> *To obtain the representation for user i, the model will look up the i-th row of the feature matrix to find the features with non-zero weights in that row; the embeddings for these features will then be added together to arrive at the user representation.*

The algorithm can directly learn about the latent features from the feature matrices and learns which items to recommend based on these provided features.

Another feature of LightFM that I found very interesting was the WARP loss function. WARP (Weighted Approximate-Rank Pairwise) loss "*maximises the rank of positive examples by repeatedly sampling negative examples until rank violating one is found. Useful when only positive interactions are present and optimising the top of the recommendation list (precision@k) is desired.*"[4]

[4] https://making.lyst.com/lightfm/docs/lightfm.html#id2

More thoroughly, the steps are broken down as follows[5]:

1. *For a given (user, positive item pair), sample a negative item at random from all the remaining items. Compute predictions for both items; if the negative item's prediction exceeds that of the positive item plus a margin, perform a gradient update to rank the positive item higher and the negative item lower. If there is no rank violation, continue sampling negative items until a violation is found.*

2. *If you found a violating negative example at the first try, make a large gradient update: this indicates that a lot of negative items are ranked higher than positives items given the current state of the model, and the model must be updated by a large amount. If it took a lot of sampling to find a violating example, perform a small update: the model is likely close to the optimum and should be updated at a low rate.*

Meaning the sooner the model identifies a negative example, the higher the penalty. The issue with this is that the model has a chance of negative sampling an item that has a positive rating in the validation or test set. Regardless, WARP is what drew me to the library in the first place – since our dataset comprises of positive interactions only, WARP or BPR (Bayesian Personalised Ranking) were the two best loss functions available.

WARP managed to produce an NDCG score of 0.23947 on the test set, whereas as BPR only produced 0.21597. For the assignment, it made the most sense to use WARP since the aim was to optimise the recommendation list and BPR is recommended for optimising ROC AUC. I would like to explore how WARP performs in a NCF or NMF model in the future when I have more time, as although it is currently not widely used, but it shows very promising results.

I set the initial components (number of latent features) to be 150 and applied a very tiny alpha value of 1e-10 to the item features, and this proved to be very effective. I am not sure why such a small alpha value proved to be more effective than 0, but less effective than any larger value. The model also performed worse when I applied any alpha value to the user features, possibly indicating a

LightFM's documentation also suggested that supplying feature matrices may lead to a less expressive model *"because no per-user features are estimated, the model may underfit. To combat this, include per-user (per-item) features (that is, an identity matrix) as part of the feature matrix you supply."*[5]

I implemented this method on the Kaggle set and it produced a dismal result of 0.03922. Perhaps the model required more hypertuning, but I was very surprised since I expected it to perform much better, given that it was advised to be better performing. I did some research and found this explanation from the developer –

> *"The model simply averages the embeddings of all the features it is given. Because of the averaging, the model is incapable of figuring out which features are uninformative and ignoring them.*
>
> *Consequently, if you add lots of uninformative features they will degrade your model by diluting the information provided by your good features. To prevent this, you may have to adopt more sophisticated models whose implementations are not offered by LightFM"* [6]

Clearly adding the identity matrix caused the item and user features to become diluted and such an approach on our given dataset is not compatible with LightFM.

I tried to tune the parameters of LightFM using the same approach as I did with ALS and LMF (counting the number of times the model predicted the top item to be the true top item as in the validation set), but interestingly it produced 0 every time, no matter what parameters I tuned. This shows that although LightFM didn't correctly predict the top item in the top position, it predicted the top item within the top 10 for each user more than the other algorithms did. This could be due to the extra user and item features that I was able to pass to LightFM which allowed it to predict the top item

[5] https://making.lyst.com/lightfm/docs/examples/warp_loss.html
[6] https://github.com/lyst/lightfm/issues/551#issuecomment-752715907

to be within the top 10, even if it never managed to put the top item in the top position. Despite this, it still outperformed all the other algorithms and I think this is largely due to the WARP loss function and its effectiveness on implicit data.

NEURAL COLLABORATIVE FILTERING

I was quite surprised and disappointed by the performance of my NCF model. It performed the worst out of all the models and produced a range of results, the highest being 0.12148 on the Kaggle set.

Initially I started off with 4 hidden layers thinking that it would make my model stronger, but as literature says, more layers do not lead to better performance. I found that increasing the nodes in a single layer increased the NDCG performance on the validation set, however it still produced a poor score on Kaggle of 0.11372.

I tried a range of parameters for NCF including the number of embedding factors, number of hidden layers, and different batch sizes but it was still unable to perform anywhere near the other algorithms and produced poor hit rate results on the validation dataset.

I also evaluated the model with and without the user and item features, but the model performed much worse without these extra matrices (0.04551 on Kaggle), so it leads to me to think that the sparse dataset does not have enough data for a machine learning model to learn from.
On average each user and item has approximately 13 ratings each, where some users have rated up to 75 items, and some have only rated 6. It is a similar case for individual items – some items have many ratings, and some have very few. Considering that there are over 2000 users and items, the performance of NCF indicates that this is not enough data for the model to learn the behaviour and preference for each user and item.

Regardless, I still spent a lot of time trying to optimise my NCF, but I was unable to improve its performance throughout the duration of the assignment. I would like to apply the NCF method on a dataset where the user and item information is denser.

RESULTS

In order of best to worst performance on Kaggle test set:

Method	Kaggle NDCG Score
Hybrid Matrix Factorisation	0.23947
Alternating Least Squares	0.21097
Logistic Matrix Factorisation	0.21011
Neural Collaborative Filtering	0.12148

CONCLUSION

The hybrid matrix factorisation method implemented through LightFM produced the best results for the Kaggle dataset, followed by ALS, LMF and NCF. Based on the performance of each of the algorithms, I think that all the models except for NCF could be tuned to produced similar scores on Kaggle, but since we had limited daily attempts, I could not spend much time optimising them.

Both ALS and LMF had an interesting relationship between their α and λ values, with the two parameters serving different roles in both algorithms. It was important to determine the best value for both the models

With the NCF model, I would need significantly more dense data to produce similar results to the other algorithms. In future I would like to attempt a Neural Matrix Factorisation model to see the effect of adding a General Matrix Factorisation step to the Neural Network. It would be interesting to see if adding a GMF step would still produce poor results, or if it would improve the score due to having some extra information in the NN model.

PART 2: NODE CLASSIFICATION

INTRODUCTION

The following 2 node embedding approaches were explored:

- Node2Vec
- DeepWalk

Along with the following 3 classification algorithms:

- Random Forest Classifier (RFC)
- Logistic Regression (LogReg)
- K Nearest Neighbours (KNN)

RFC with DeepWalk node embeddings performed the best with **80.56%** accuracy.

Table 3: Algorithm Accuracy with Node2Vec vs. DeepWalk

	Node2Vec	DeepWalk
RFC	0.6439	0.8056
LogReg	0.5796	0.7819
KNN	0.6172	0.6797

NODE2VEC VS. DEEPWALK

I wanted to explore the difference in the two node embedding methods and so I decided to implement both with the different classification algorithms.

I first tried to create Node2Vec embeddings with walk lengths of 100, with 200 walks for each node. This hit my already increased RAM limit of 25GB on Colab and caused my notebook to crash after an hour. I then tried to reduce the number of walks to 80 for each node, however this was still too many walks per node. Eventually I settled for 10 walks per node, and even then, the fitting of the model took almost an hour.

Both Node2Vec and DeepWalk have the same default parameters, and these are the values that I went with:

Parameters	**Node2Vec**	**DeepWalk**
No. of random walks	10	10
Walk length	80	80
Dimensions	128	128
Window Size	5	5

Node2Vec has a default value of 1.0 for both p and q, meaning it is equally likely to go to a new node, and to return to the previous node.

I also trained a Node2Vec model with slightly longer walks (100) and larger window sizes (10), as I thought it might lead to an improvement in accuracy, however the difference was not significant in any of the algorithms, so I decided to use the same parameters as DeepWalk.

	WalkLen 100, Window 10 (Accuracy)	**WalkLen 80, Window 5 (Accuracy)**
RFC	0.6457	0.6439
LogReg	0.5919	0.5796
KNN	0.6275	0.6172

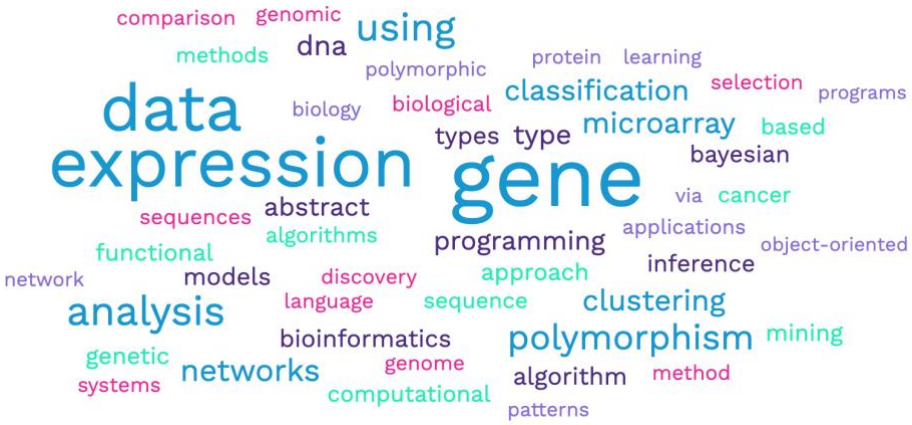
I wanted to explore Spectral Clustering as an approach for node embedding, however Colab crashed when I converted the graph to an adjacency matrix and tried to fit sklearn's Spectral Clustering algorithm over it. Evidently our given dataset is too large for such explorations, so perhaps in future I will get a meaningful subset of the data and try to apply Spectral Clustering for my own knowledge.

DATASET EXPLORATION

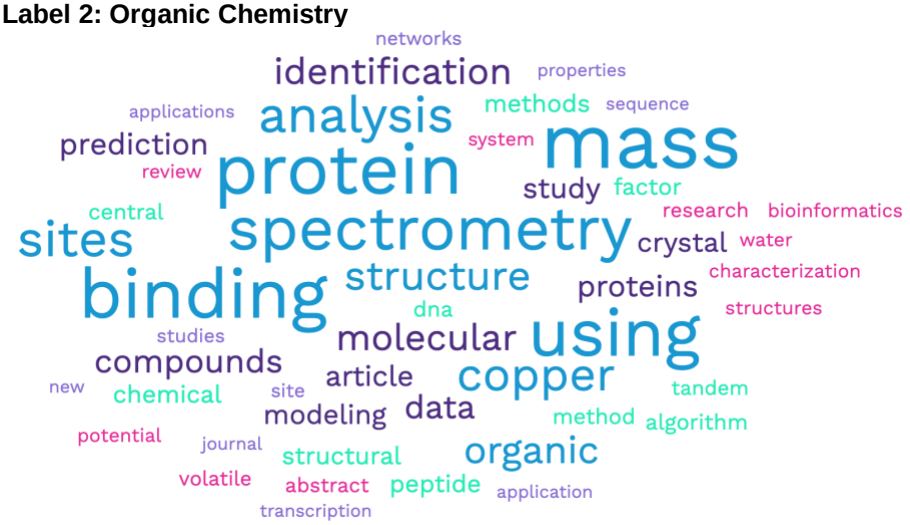
Looking at the dataset itself, I created word clouds for each label (https://www.freewordcloudgenerator.com/generatewordcloud) from some of the document titles to get a better understanding of the data itself, and determined possible topics:

Label 0: Global Politics

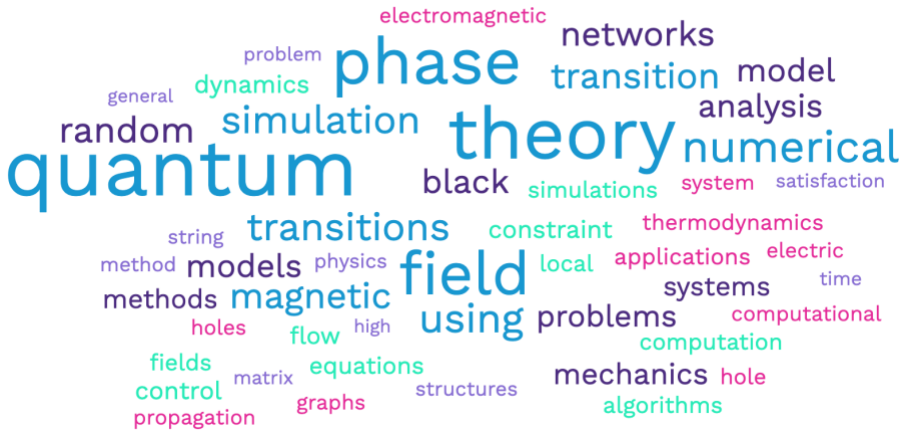
Label 1: Genomics



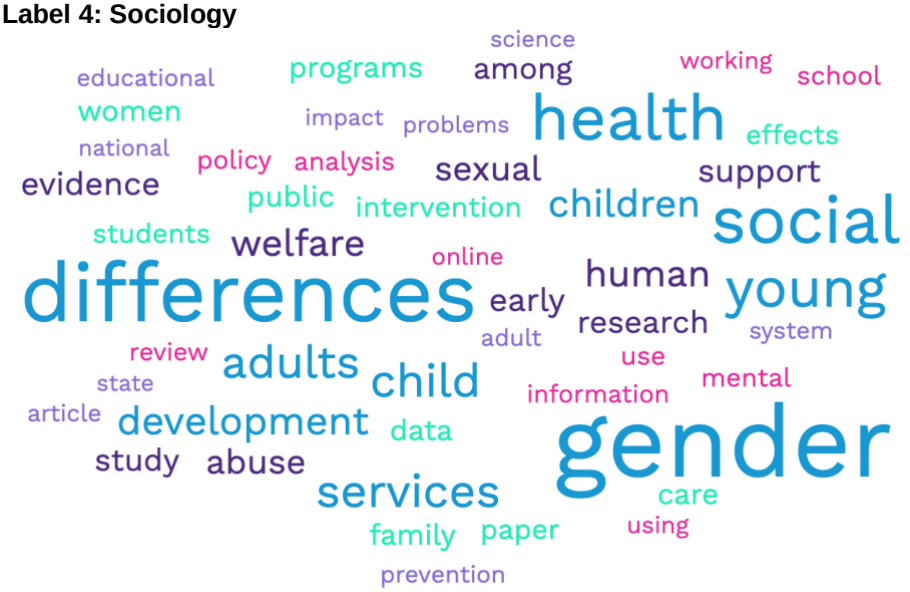
Label 2: Organic Chemistry



Label 3: Quantum Physics



Label 4: Sociology



There are clear similarities between labels 1, 2 and 3, and 0 and 4. I would expect there to be some level of misclassification between these topics if I was to do Word2Vec embeddings since they share a lot of similar terms and themes, but I am interested to see how Node2Vec performs.

There is also an imbalance in the number of nodes per label:

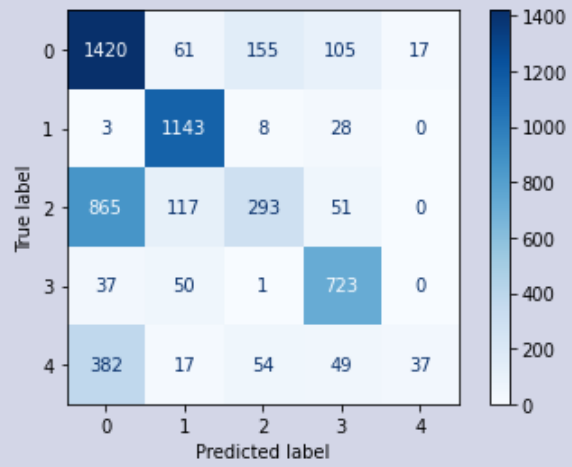
Label	No. of nodes
0	5860
1	4421
2	3941
3	2703
4	1795

Since labels 0 and 4 are potentially similar, I am expecting a few misclassifications. Looking at the word clouds, there are many similar terms between the labels, so I am also expecting some misclassifications on labels that may seem unrelated (such as organic chemistry and sociology).

It would be good to graph out the nodes and edges in a way where I could see the connections visually, however the graph took over an hour to produce, and the labels were completely unreadable, so I have not been able to view the relations between nodes. This would be a good activity to do to explore whether there are some nodes that are related to nodes in other labels.

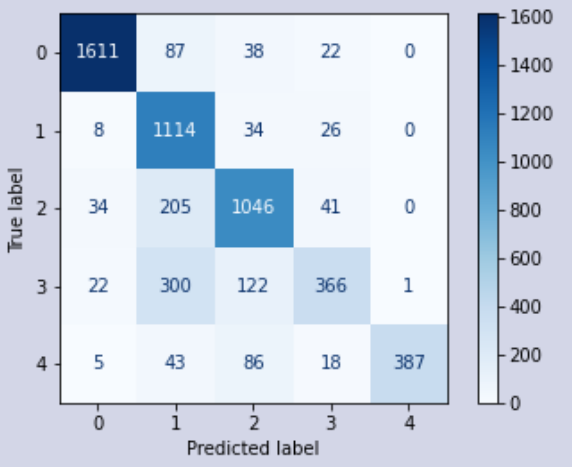
RANDOM FOREST CLASSIFIER

RFC || Node2Vec



Accuracy: 64.39%

RFC || DeepWalk



Accuracy: 80.56%

I expected RFC to perform well in a multilabel classification as I learnt from Assignment 1, but I didn't expect it to outperform KNN.

I also initially performed hyperparameter tuning on the training set, but the tuned model performed worse on the test set than the default RFC parameters in both Node2Vec and DeepWalk, so I decided to keep the default parameters.

RFC performed the best out of all 3 classifiers, and it performed exceptionally well with the DeepWalk embeddings. This may be due to the fundamental basics of how DeepWalk learns embeddings, and how RFC determine classifications is based on the same principle: traversal of trees.

When DeepWalk creates embeddings, it jumps to neighbour nodes to identify similar nodes based on its random walks. The nodes in the random walk will have similar embeddings.

When a decision tree traverses down the features and makes decisions at each step, it is logical that nodes that are similar would have a similar traversal path, so the decision tree would be able to identify the similarities in the embeddings of similar nodes and classify accordingly.

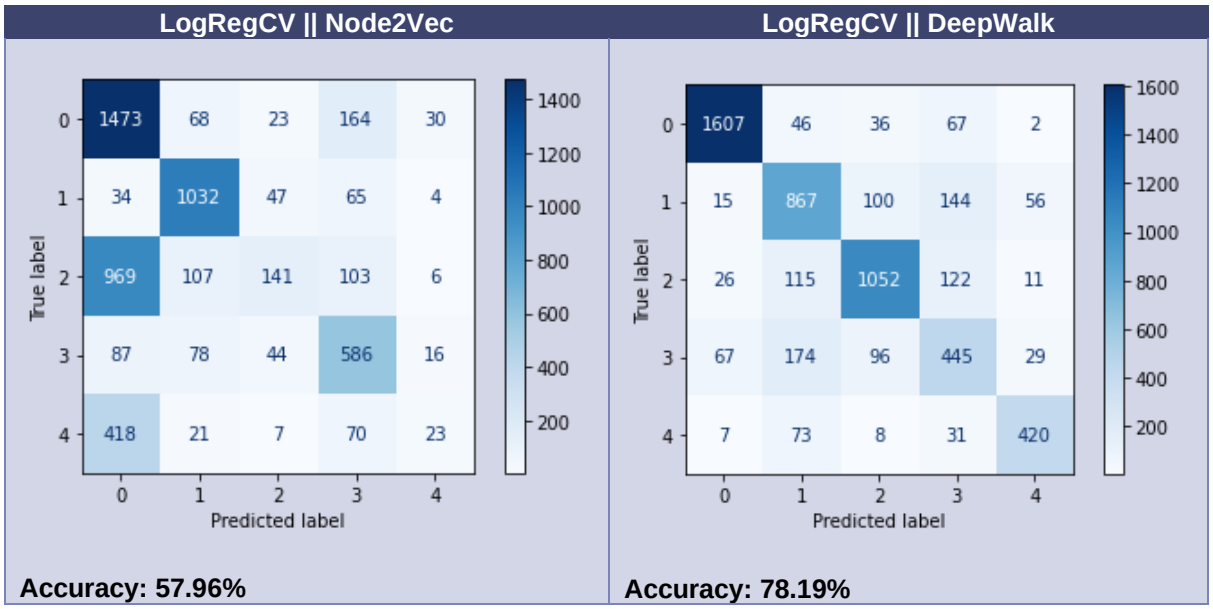
I think it is due to this similarity in principle in both building the features of the nodes and the classification of the nodes that makes RFC + DeepWalk such a strong performer.

That said, its performance on the Node2Vec embeddings was still the best performing out of all the classifiers and I believe this is because RFC is great for multilabel classification. In my research for assignment 1, I found the RFC was generally used for multilabel classification, and it performed quite poorly on the binary classification set. However, in a dataset like in assignment 2 where there is deep complexity in the input parameters, RFC is perfectly suited for such a problem.

Looking at the actual classifications themselves, RFC was the best classifier at identifying label 2 (organic chemistry) in DeepWalk, but heavily misclassified label 2 as label 0 (global politics) in Node2Vec. This is consistent with all the classifiers with Node2Vec, which implies that the random walks produced show that are similarities between the two classes when creating Node2Vec embeddings, however looking at the topics themselves this seems quite strange. The classifier failed to classify label 4 in Node2Vec, with a 0.0686 accuracy, compared to 0.7310 on DeepWalk. This could be due to the smaller available dataset for label 4. It is also possible that these documents are related even though they are in different classes due to their similar topics.

It is likely that to increase the performance of Node2Vec, we might need to tune parameter p to be a lower value to encourage a more Breadth-first search approach and get a better local view as it appears that the model is preferring a Depth-first search approach by learning the features for the outer nodes over those that are local and in the same class.

LOGISTIC REGRESSION CV



Logistic Regression performed quite well on the DeepWalk embeddings but was the worst performing algorithm on the Node2Vec embeddings. I initially thought LogReg would not perform as well as the other two classifiers due to usually being attributed to binary classification, but with DeepWalk it surprisingly outperformed KNN by almost 10% and performed almost as well as RFC.

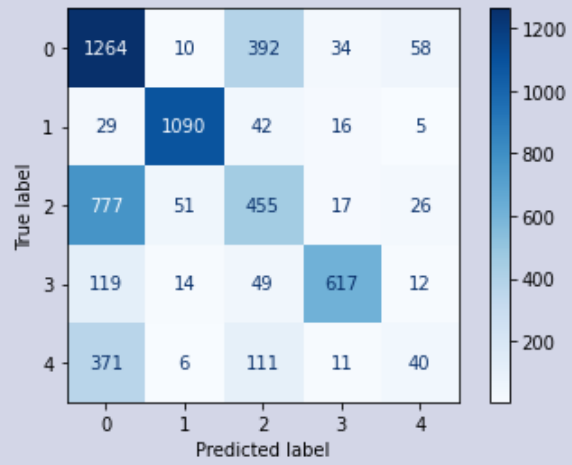
In DeepWalk, LogReg is more accurate in classifying labels 2 (organic chemistry), 3 (quantum physics), and 4 (sociology) and the fact that its performance is as accurate as RFC means that there is probably some linear relationship in the DeepWalk embeddings of these classes. This linearity is evidently not as strong in the Node2Vec embeddings which indicates that perhaps the Node2Vec function could benefit from a bit of finetuning of the p and q parameters to produce more local neighbourhood data which could lead to more linearly separable information.

LogReg heavily misclassified label 4 (sociology) as label 0 (global politics) in Node2Vec, but again this could be due to the smaller dataset available for label 4.

Like RFC, I tried to hypertune some of the LogReg parameters, but none of the tuned models produced high accuracies like the the default model so I chose to use the default values. Overall, I am pleasantly surprised with the performance of LogReg on the DeepWalk classification as it performed beyond my expectations, however its performance on the Node2Vec embeddings was quite poor, however I do not believe it is the fault of the classifier, but rather the lack of linear relationships between the embeddings produced that make it incompatible with Logistic Regression.

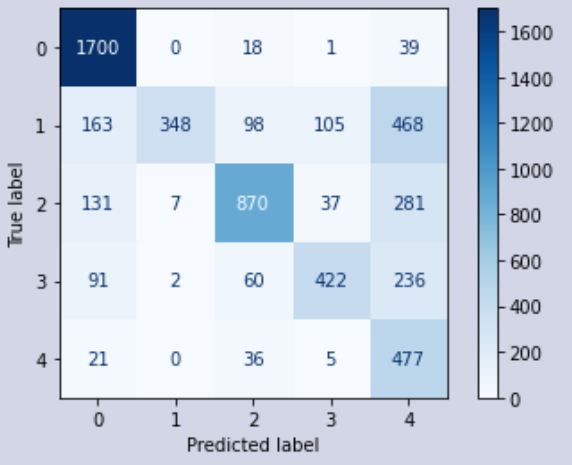
K NEAREST NEIGHBOURS

KNN || Node2Vec



Accuracy: 61.72%

KNN || DeepWalk



Accuracy: 67.97 %

I was very surprised by the performance of KNN and quite disappointed. I initially expected KNN to perform the best out of all 3 algorithms based on the logic of the algorithm – check the neighbours of the node and classify based on the majority vote.

It is interesting to note that KNN was the second best performed in Node2Vec – this may be due to KNN handling noisy data better than other algorithms and I think the Node2Vec embeddings have created a lot of noise in the data compared to DeepWalk. This is supported by the performance of all the classifiers on the Node2Vec data. In future this could be addressed by hypertuning the parameters to get the highest accuracy, but it is still interesting to see how KNN performed poorly even on the DeepWalk embeddings.

In Node2Vec, the classifier confused label 4 (sociology) to be mostly label 0 (global politics) and some label 2 (organic chemistry). Looking at the word clouds, there are some terms in organic chemistry that overlap with sociology such as “studies, research, review”, so I can see how Node2Vec has determined that these were related.

The results of KNN indicate that there are probably labels in different clusters that are near each other in the actual graph. This is supported by the fact that lower values of k performed better than higher values, and k=3 performed the best. Perhaps KNN is not the best algorithm for our node classification data since the true neighbours are not determined by the class labels, but by the connections in the graph. The nodes that are related *should* be similar, but there is no guarantee that they are part of the same class.

DISCUSSION

	Accuracy
RFC + DeepWalk	0.8056
LogReg + DeepWalk	0.7819
KNN + DeepWalk	0.6797
RFC + Node2Vec	0.6439
KNN + Node2Vec	0.6172
LogReg + Node2Vec	0.5796

All the algorithms performed better on the DeepWalk embeddings, which was surprising because in most of the research papers that I researched, Node2Vec usually performed better than DeepWalk. This makes me think that the Node2Vec requires some hyperparameter tuning to increase the

accuracy. The actual algorithms' performance was really telling of the type of relationships that the two embeddings learned – DeepWalk tends to produce more linear relationships, and this is probably due to the "fixed-length" nature of the walks that captures correlation between nodes well.

In future to improve the performance of Node2Vec, it would be worth investigating tuning the parameters for p and q. I think the default values for the algorithm are not suitable for our given dataset, but training the model takes a very long time and so I was not able to explore this avenue as part of the assignment, but considering that the default values of p and q are 1, it would be good to try out different values for each to see the effect of having a model based on the local neighbourhood vs. a more global view.

It would also be interesting to see the effect of Word2Vec on the document titles and compare the classification based on Word2Vec vs. Node2Vec vs. DeepWalk. Looking at the word clouds for each label, there are clearly very distinct topics, and I would expect Word2Vec to perform well with classification.