Network Machine Learning for Drug-Drug Interaction prediction
EE-452

Jacob Bamberger, Martin Dahl

September 9, 2022

1 Introduction

1.1 OGBl-ddi dataset

Open Graph Benchmark (OGB) [4] contains several graph datasets which are useful for researchers to benchmark their methods, and to provide new challenges to researcher. The datasets are organized by problem descriptions, namely node property prediction, link property prediction, graph property prediction, and large-scale graphs.
OGBL-drug-drug-interactions (ogbl-ddi) is an OGB dataset in the link property prediction class. It consist of a homogeneous, unweighted, undirected graph, representing the drug-drug interaction network. Each node is a drug, and edges represent interactions between drugs. The graph does not come with any signal on the nodes or edges. The graph consists of $4\ 267$ nodes and $1\ 067\ 911$ undirected edges. Statistics and data exploration about the graph can be found in Section 2.

1.2 Problem description

The prediction task is that of predicting whether or not two drugs interact in the graph. This is an example of link prediction, where the goal is to predict if there is an edge between two given nodes, one of the main graph machine learning problems. Common approaches are briefly described in Section 3
The evaluation metric is the Hits@K metric. For each positive drug interactive (edge existing in the graph), we compare it to $100\ 000$ randomly sampled negative drug interactions (non-existing edges). We then count the ratio of positive edges whose score is in the range of the K highest negative scores. Intuitively, we want positive edges to have higher scores than most $(100\ 000 - K)$ negative edges. The OGB leaderboard [1] for ogbl-ddi is based on Hits@20.
Data Splitting is done by the OGB team, who provide use with an $80\%,\ 10\%,\ 10\%$ split of the ogbl-ddi edges for training, validation and test set. Additional negative edges (i.e. edges not in the ogb-ddi graph) are also provided for validation and testing, which are both in the order of $100\ 000$. Note that no negative training edges are specified, so we are free to sample any edge that is not in the positive training set. One feature of the splitting is that the edges in the test set correspond to drugs that interact differently in the body compared to the validation and training edges, which is an additional challenge making the task more realistic.

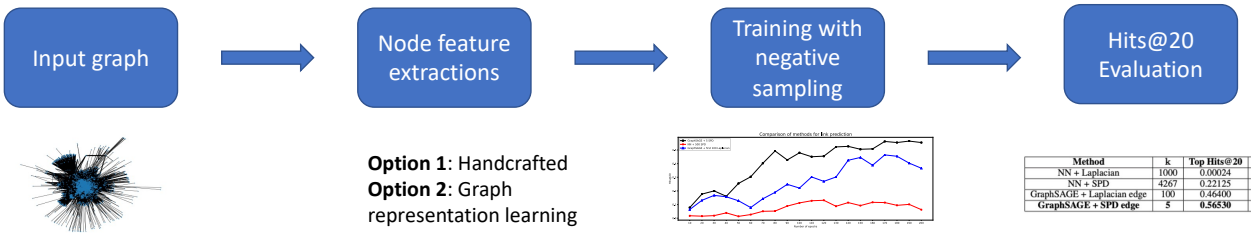


Figure 1: Overview of pipeline described in Section 3

1.3 Structure of the report

In Section 2, we first do some data exploration to understand the structure of the graph at hand. In Section 3 we describe possible link prediction pipelines. In Section 4, we design some baseline methods for the link prediction task, and describe the method reported by [6] which achieved second place on the OGB leaderboard at the time of writing. Their code is available on github [5]. In Section 5 our results are presented and discussed.

2 Data Exploration

2.1 Visualisation

A visualisation of the ogbl-ddi graph can be seen in Figure 2. The spring layout shown in the Figure provides useful insights: the graph has a tightly connected core, and many very low degree nodes. Furthermore, it can be seen that the graph is connected.

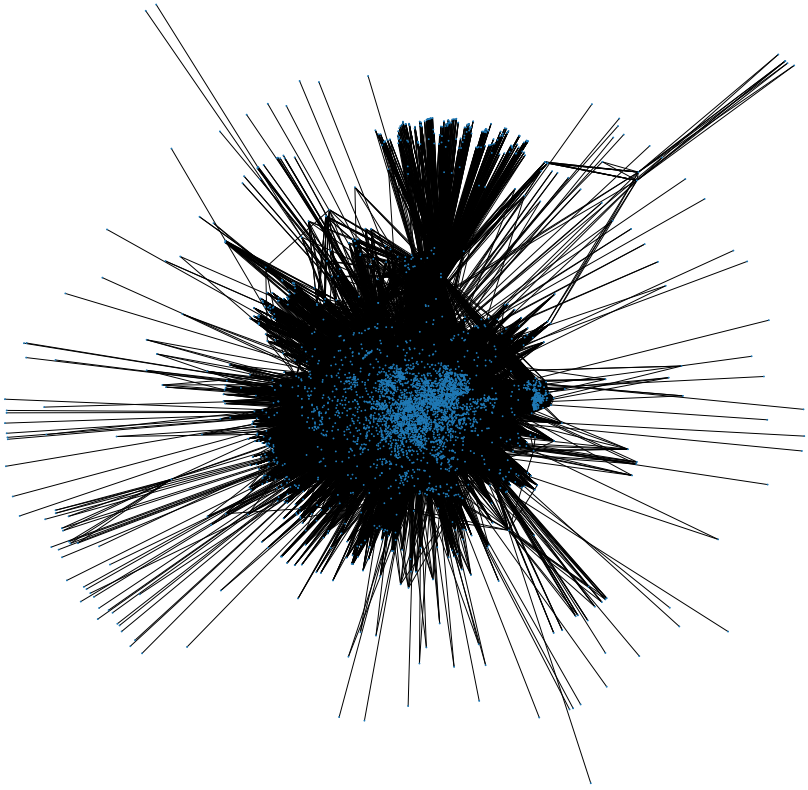


Figure 2: Spring layout of the ogbl-ddi training graph

2.2 Basic statistics

To gain a better understanding of the ogbl-ddi graph, we computed some useful statistics:

- It is undirected and connected.
- The full ogbl-ddi (training) graph has 4267 nodes with $1\ 334\ 889$ out of $9\ 101\ 511$ possible undirected edges.
- The average degree is 500. See degree distribution in Figure 3.
- The diameter of the graph is 5
- The eccentricity (furthest distance to other nodes) of nodes is either 3, 4 or 5, with respective proportion of 26%, 68% and 6%.
- Using methods from the OGB python package, the data is split into train, test and validation with $1\ 067\ 911$, $133\ 489$, $133\ 489$ undirected edges, respectively (a 80, 10, 10 split).

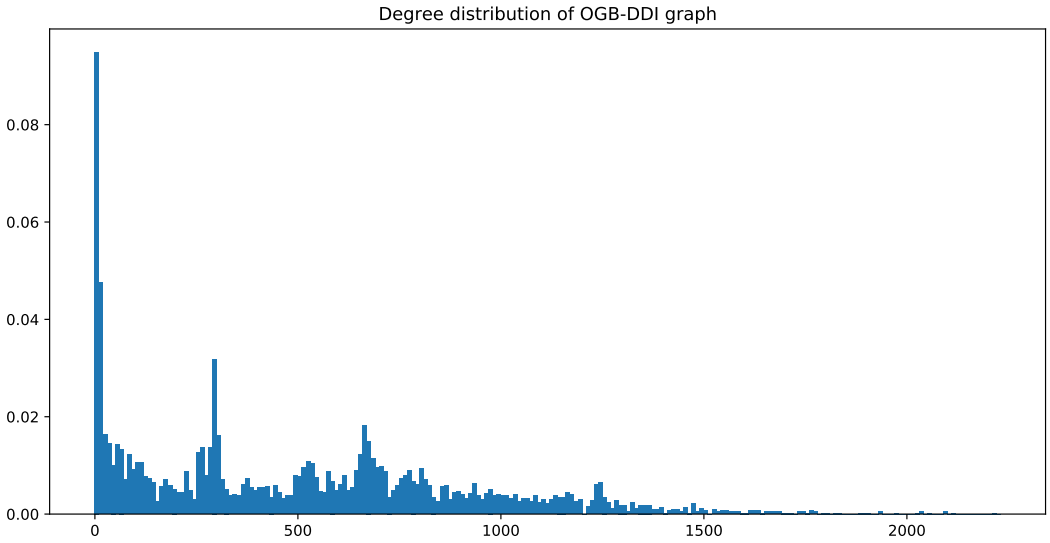


Figure 3: Full degree distribution of OGBL-DDI graph. Bin size is 10

2.3 Description of selected features

We now describe two statistics the Shortest Path Distance, and the Laplacian eigenfeatures, which we will study to test their predictive power for link prediction.

2.3.1 Shortest path distance (SPD)

The shortest path distance is a way to define a metric $d : \mathcal{V} \times \mathcal{V} \to \mathbb{R}_+$ on the set of nodes $\mathcal{V}$ of a graph $G = (\mathcal{V}, \mathcal{E})$. This metric is defined as

$$d(u, v) := \min_{path\ p:u->v} length(p),$$

where the length of a path is the number of edges in the path.

An efficient implementation of to find this metric is using either Djikstra's algorithm, or the Floyd-Warshall algorithm. However, since these algorithms are $\mathcal{O}(|\mathcal{V}|^3)$, this is impractical for large graphs as for the ogbl-ddi.

An approach to reduce complexity is adopted in [6]. It consists of subsampling a small ($k << |\mathcal{V}|$) set of *anchor nodes*, and to compute the shortest path distance from any node, to all of the k anchor nodes. Note that this approach is also more statistically robust. This can then be interpreted as node features, since for each node we get a k-dimensional vector where each entry is a distance to the anchor nodes.

See Figure 4 for a vizualisation of the distribution of average distance to other nodes. The mean average distance to other nodes is roughly 2.

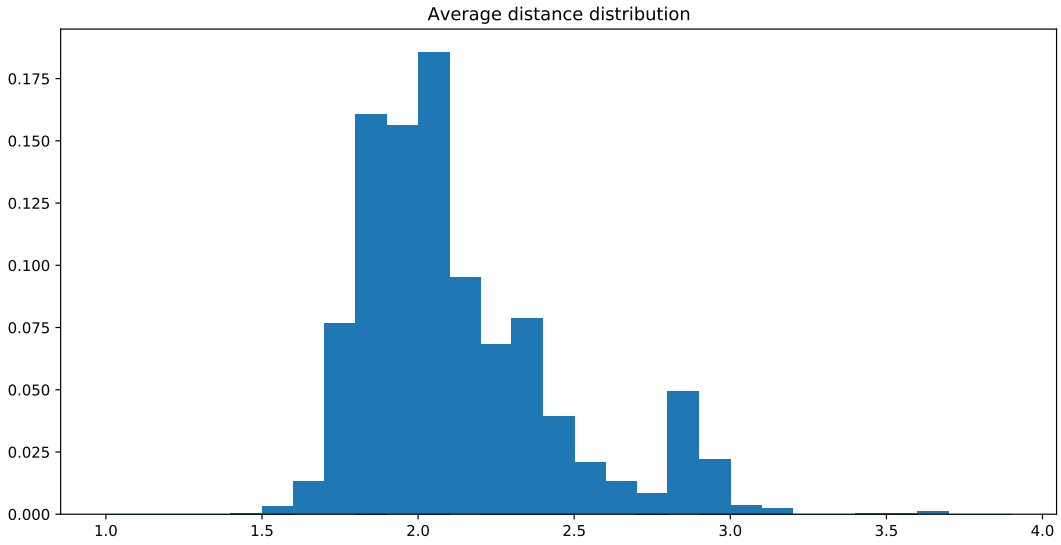
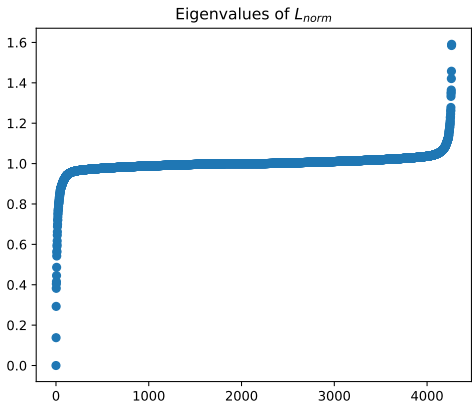


Figure 4: Distribution of the average distance to other nodes

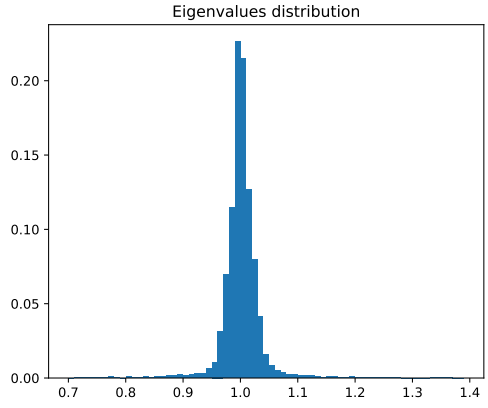
2.3.2 Laplacian Spectra

Since we only consider the graph structure in our task, it can be insightful to study the graph's spectra. We consider the normalized Laplacian $\tilde{L}$, and its eigen-decomposition $\tilde{L} = U\Lambda U^T$.

The OGBL-DDI graph's eigenvalues are plotted in increasing order in Figure 5a, and the distribution of eigenvalues can be found in Figure 5b. Note that there are large gaps between the lowest eigenvalues and between the largest ones, whereas most of them center around 1. This suggests that there might be more structural information in the eigenvectors corresponding to both the lower and the higher eigenvalues. In Section 4.3 we attempt to leverage the information from the eigenvectors corresponding to these eigenvalues, by treating them as structure encoding node features. The eigenvectors are also visualized in Figures 9 & 10 found in appendix A.



(a) Eigenvalues of normalized Laplacian $\tilde{L}$



(b) Eigenvalue distribution, the outliers (< 0.7, and > 1.4) are ignored in theis representaiton.

Figure 5: Visualisations of the ogb-ddi spectra

3 Link prediction with or without GNNs

We now describe a framework to do link prediction, similar to the exposition given in [3]. Usually, a link prediction pipeline works in two steps: node embeddings are first either learned or handcrafted, then two nodes are fed into a link predictor to compute the probability of having a link between these two nodes. These two steps can also be combined and learned simultaneously. To train the networks, one usually treats it as a supervised learning problem where the label is 1 the pair of nodes have a link in the training graph, and 0 otherwise. We first describe link predictors, then describe GraphSAGE which is a method to learn node embeddings, finally we describe a common method to train these models.

3.1 Link Predictor neural network

A common method for undirected link prediction is to feed pairs of nodes into an MLP (or other differentiable parametrized model), after taking their Hadamard product (pointwise multiplication) which makes the network invariant to permutation of the two input nodes. A summary of the pipeline described in this section can be seen in Figure 1. In addition, the output is passed through a sigmoid activation to obtain a probability score between in the range $[0, 1]$. See Equation 1 for a symbolic summary where y is the probability score of having an edge between nodes i and j, and x_i & x_j are the node embeddings for node i & j, respectively.

$$y = \sigma(\mathcal{F}_{network}(\mathbf{x}_i \circ \mathbf{x}_j)) \tag{1}$$

3.2 GraphSAGE

GraphSAGE (Sample and Aggregate) [7] is an inductive GNN generating node features by layer-wise aggregation of the features of a nodes neighbourhood. Different to other GNNs, GraphSAGE only samples a sub-set of a node's neighbourhood, allowing for scalability with bigger graphs when computational time is a constraint. Intuitively, this makes it also more robust to graphs where edges are missing, as is the case in the link prediction task.

Let $G = (\mathcal{V}, \mathcal{E})$ be a graph with vertices $\mathcal{V}$ and edges $\mathcal{E}$. Furthermore, let every node in the graph have an initial node feature $x_v \in \mathbb{R}^d$ for selected $d \in \mathbb{N} \geq 1$. The goal is to generate deep features h from the initial features x. For each layer of a graph neural network, the node features are aggregated as in Equation 2 with Equation 3. Let $h_v^0 = x_v$ and $\mathcal{N}(v)$ be a subset of the neighbour nodes of v. $A_k(x)$ is the aggregation function for layer k while $CONCAT(x)$ is the concatenation function, concatenating vectors. $\mathbf{W}_k$ is a matrix with the model weights for layer k, the dimension of the weight matrix depends on the wanted input and output of the layer. One common aggregation function $A(x)$, which will also be used in following experiments is the averaging function, taking the average of the members in a (multi) set.

$$h_{\mathcal{N}(v)}^k = A(\{\{h_v^{k-1}, \forall u \in \mathcal{N}(v)\}\}) \tag{2}$$

$$h_v^k = \sigma(\mathbf{W}_k \cdot CONCAT(h_v^{k-1}, h_{\mathcal{N}(v)}^k)) \tag{3}$$

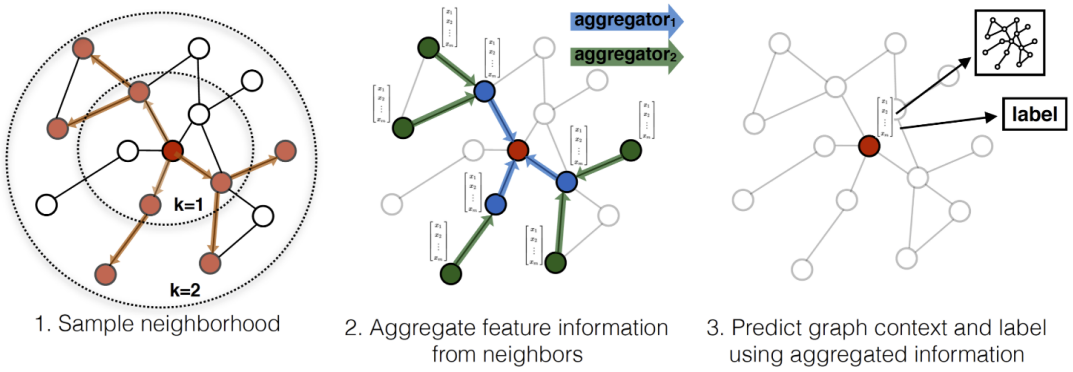


Figure 6: Figure giving an overview of the GraphSAGE layers. Taken directly from [7].

In Figure 6 three steps are shown describing the main principles of GraphSAGE. In step 1, $\mathcal{N}(v)$ is created by sampling a fix number of neighbours of a node v. In step 2, the aggregation as described in Equations 2 and 3 is performed.

Finally in step 3 a representation $\mathbf{z}_u$ of selected dimension has been generated for each node $u \in \mathcal{V}$, being available for a task such as graph, link or node class prediction.

3.3 Training:

When no GNN is used to generate node features, we simply train a Link Predictor neural network. When a GNN is used, we can train both the GNN and the link predictor simultaneously by letting the gradient flow to the GNN's parameter.

The problem is treated as a supervised learning problem on pairs of nodes, where pairs of nodes with an edge between them are labeled 1 and called *positive edges*, and labelled 0 and called *negative edges* otherwise.

Most real world graphs being sparse, there are many more negative edges than positive ones. The supervised learning problem is therefore unbalanced. *Negative sampling* is usually used to fix this problem: one negative edge is sampled randomly per positive edge seen during training.

The choice of loss function is usually a sum of of a positive loss $\mathcal{L}_+$, and a negative loss $\mathcal{L}_-$. In order to perform well on the Hits@K metric, the positive scores should be ranked higher than the negative ones. Therefore $\mathcal{L}_+$ shall be high for low values of the input probability score, and $\mathcal{L}_-$ shall be high for high values of the input probability score. One popular such choice is given in Equation 4, which is the one we use it to optimize all the networks in the project.

$$\mathcal{L}_+(y) := -\log(y) \;\; \& \;\; \mathcal{L}_-(y) := -\log(1-y)\,. \tag{4}$$

4 Model descriptions

4.1 Baselines: Link Predictors on handcrafted features

We first consider models without using GNNs. These models all consist of a Link Predictor neural network as defined in Equation 1, and operate on different handcrafted features, the details of each network can be found below. The goal of the models are twofold: they will serve as baselines to compare to more complex models that use GNNs for node embeddings, and are also used to serve as a heuristic for comparing the importance of different hand crafted features. We thus consider two different models, differing only through the features they operate on:

- **Link Predictor neural network on SPD node features.** The input node features are k-dimensional vectors corresponding to the SPD features for a sampled set of anchor nodes, as described in Section 2.3.1. We repeated the experiment for k taking values 100, 500, 1000 and 4267. Results are reported in Figure 11.
- **Link Predictor neural network on Laplacian node features.** The input node features are k-dimensional vectors corresponding to the eigenvectors corresponding to the k smallest eigenvalues of the normalized Laplacian Spectra features defined in Section 2.3.2. This is identically to the spectral node embeddings of the graph using eigen-maps seen in class. The choice of smallest eigenvalues is to respect the graph's structure. We repeated the experiment for k taking values 100, 500, 1000 and 4267. Results are reported in Figure 12.

We parameterize the Link Predictors as MLPs with ReLU nonlinearity and dropout after each hidden layer, except the output layer where Sigmoid activation without dropout is used. The exact hyperparameters are summarized in Table 1, we fixed the same hyperparameters for both networks, the choice was based on [6]. The loss function is the sum of both positive and negative loss from Equation 4. The optimizer is Adam optimizer.

# in channels	# hidden channels	# hidden layers	# out channels	Dropout	learning rate	batch size
k	256	5	1	0.3	0.003	64*1024

Table 1: Quick summary of link predictor neural network.

4.2 Reproducing GraphSAGE + SPD Edge Attributes from [6]

We now describe the method presented in [6], which is the GNN method we chose to investigate. This method scores second place on the ogbl-ddi leader-board [1], and is much simpler than the other top scoring methods.

The method consists of a preprocessing step, followed by a node representation learning step, and finally a link prediction step. The input is the training graph $G = (\mathcal{V}, \mathcal{E})$, which consists only of positive edges. From this graph, the authors propose a pre-processing step which consists of computing random node features and SPD based edge features.

Pre-processing: The nodes features are set to be random 256 dimensional with Xavier initialization, which are also learned through back propagation.

The novelty of their approach seems to be the structural edge features, which are computed as follows:

- They first compute SPD using 500 anchor nodes, giving a 500 dimensional vector for each node of the graph.
- They then repeat the following procedure 5 times, and concatenate the output to end up with a 5-dimensional feature for each edge.
  - They sub-sample 200 anchor nodes without replacement, giving a 200 dimensional vector for each node.
  - For each edge uv in the graphs, they take the mean of u's and v's 200 dimensional vector.
  - For each edge uv, compute the mean of the previous vector, giving one scalar for each edge.
  - Finally min-max normalize across all edges.
- Each of these 5 dimensional edge features features are then multiplied by a matrix (of size 5×256), where the matrix is learned and initialized with Xavier initialization.

Node representation learning: They use a GNN to learn node representations. The specific architecture used is two layers of a modified SAGEConv [7] layers, which in addition to node features, also takes edge features into account which is described below. The non-linearities are ReLU, and the network uses dropout for each layer. The number of input channel, hidden channels, and output channels are all 256. **GraphSAGE modification:** The main difference to SAGEConv is that when computing messages from one node to another, it also uses the edge features. Indeed, the only difference to pytorch geometric's implementation of SAGEConv is the *message* function, which instead of taking as input a node feature and outputing the same node feature (thus is the identity map), it takes as input a node feature and a corresponding edge attribute, and output the ReLU of the sum of both. The rest is exactly like SAGEConv: after propagation the output is passed through a fully connected layer, and (since we use root_weight=True) it is summed with the previous layer's node feature passed through another fully connected layer, finally (since we use normalize=True) the output is normalized.

Link Predictor: The link predictor is a 2 layer fully connected MLP, one from 256 dimensions to 256, and one from 256 to 1. The forward method takes as input two 256 dimensional node features, then point-wise multiplies them (ensuring permutation invariance), then passes them through each layers, where after each fully connected layer (except the last one) a ReLU nonlinearity and dropout is applied with parameter 0.3. The output is then the sigmoid (which lies in $[0, 1]$) score. See Table 2 for the hyperparameter summary.

# in channels	# hidden channels	# hidden layers	# out channels	Dropout	# of parameter
256	256	2	1	0.3	131 841

Table 2: Quick summary of link predictor neural network.

4.2.1 Training:

We now describe the training procedure.

Negative sampling: For each positive sample, a negative edge is also sampled using the 'dense' method in PyTorch Geometric's negative sampling method.

Loss function: The loss function is again the sum of the positive and negative losses defined in Equation 4.

Optimizer: The chosen optimizer is the Adam optimizer with learning rate 0.003. The parameters that are optimized are the random node features, edge embedding matrix as well as the GraphSAGE parameters and link predictor parameters, for a total of $1\,488\,129$ parameters. Grad-clipping is done on all parameters using torch's *clip_grad_norm* function with norm parameter 1, meaning that the used gradient is the minimum between 1 and the actual gradient. The batch size is set to $64 \cdot 1024$.

4.3 GraphSAGE + Laplacian edge features

Similar to the two choices of Link Predictor baselines, we compare SPD features to Laplacian eigen-features by repeating the method previously described in Section 4.2 but replacing the input SPD edge features by Laplacian Edge features obtained from node features described in Section 2.3.2.

To create the Laplacian edge features from the node features, we let v_i^k be the Laplacian node feature with k elements as defined in Section 4.1. Then the undirected Laplacian edge feature of edge (i, j) is defined as $v_{i,j}^k = v_{j,i}^k = \frac{v_i^k + v_j^k}{2}$. Proceeding, these new Laplacian edge feature are used just like the SPD edge features, training on GraphSAGE with edge attributes as described in Section 4.2. Results of this experiment is presented in Figure 8.

We use the same architecture and hyperparameters as the reproduction above.

5 Results and Discussion

5.1 Results

Link Predictor baselines: We first compare the performance of the Link Predictor baselines for different choices of k in Table 3. The evolution of performance per epoch can also be found in Figures 11 & 12 of the Appendix. We then fixed $k = 500$ for the SPD Link Predictor, and $k = 1000$ for the Laplacian Link Predictor. For the former, we did not pick $k = 4267$ since the maximum was attained abruptly by overshooting rather than by learning progressively as it was for $k = 500$ as can be observed in Figure 11 of the Appendix.

Method	k	Top Hits@20	Num epochs for top Hits@20
NN + SPD	100	0.11508	20
NN + SPD	500	0.13114	120
NN + SPD	1000	0.01802	10
NN + SPD	**4267**	**0.22125**	**20**
NN + Laplacian	100	0.00014	20
NN + Laplacian	500	0.00016	10
NN + Laplacian	**1000**	**0.00024**	**20**
NN + Laplacian	4267	0.00010	50

Table 3: Comparison of Link Predictor performance on test set for different values of k. Top performance for each model is in bold.

Reproduction: In Figure 7, we plot the training curves of the validation and test Hits@20 metric for the GraphSAGE + SPD Edge Attributes, thus reproducing the results from [6].

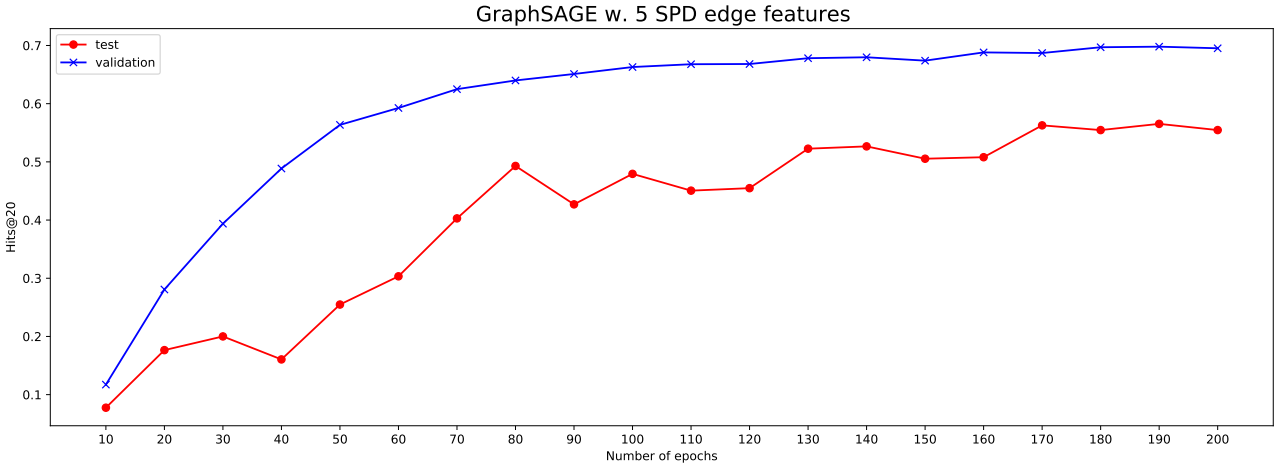


Figure 7: Hits@20 per epoch on test set and validation set of GraphSAGE trained with SPD edge features, exactly as done in [6].

Model comparison: Finally, we compare the evolution of performance per epochs of our models in Figure 7. We could not include the Laplacian Link Predictor because it's performance is orders of magnitude worse than the others. We summarize the top achieved test performance in Table 4. In this Table, note that SPD 5 for GraphSAGE+SPD edge is misleading, as the 5 refers to the number of subsampling described in Section 4.2 and not the number of anchor nodes.

Method	k	Top Hits@20	Num epochs for top Hits@20
NN + Laplacian	1000	0.00024	20
NN + SPD	4267	0.22125	20
GraphSAGE + Laplacian edge	100	0.46400	170
GraphSAGE + SPD edge	**5**	**0.56530**	**190**

Table 4: Comparison of all model's performance. Top model is in bold.

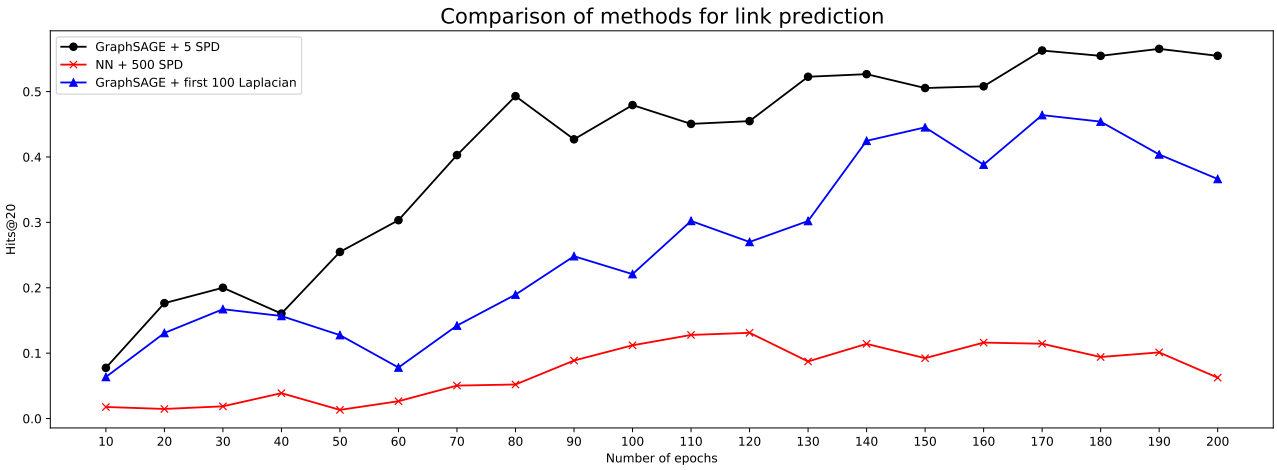


Figure 8: Comparison of Hits@20 per epoch on test set for GraphSAGE trained on SPD edge features, Laplacian edge features and Link Predictor trained on SPD node features.

5.2 Discussion

Link Predictor baselines: The Link Predictors are not comparable in terms of performance. The SPD Link Predictor consistently outperforms the Laplacian Link Predictor by 3 orders of magnitude, as seen in Table 3. Indeed, using SPD we get to 0.22 Hits@20, whereas we are capped at 0.00024 for Laplacian eigenfeatures which is comparable to a random predictor which would have performance of $\frac{20}{133489} = 0.00015$. Furthermore, the Laplacian eigen-feature's performance decays to 0 with the number of epochs, showing a strong tendency of overfitting, see Figure 12 of the Appendix. Usually, Laplacian eigenfeatures are used to represent graphs in the presence of a graph signal, in this case there is none except for the artificial binary graph signal from the adjacency matrix.
For the choice of hyperparameter k, we observed that it had a very little impact on the Laplaciand Link Predictor, and a moderate impact on the SPD Link Predictor. As mentioned earlier, we considered the $k = 4267$ peak of the SPD Link Predictor at epoch 20 to be 'luck', and a proper statistical analysis should be performed to confirm this hypothesis. Unfortunately, we could not investigate this further due to computational limits.

Reproduction: The reproduction of result from [6] was succesful. As seen in Figure 7, the model reaches high performance both on validation set and test set, reaching 0.7 and 0.56 Hits@20 respectively. These scores would position between eigth and eleventh position on the OGB Leaderboard [1]. These performance did however not allow us to reach second position, and this is likely due to the fact that we could only afford to run their model once for 200 epochs compared to several times until 400 epochs as they did.

Model comparison: The comparison of our models in Table 4 is also interesting. We first notice that both GNN models clearly outperforms both Baseline Link Predictors. Secondly, we notice that the GNNs seem to learn better as the maximum happens after around 180 epochs, whereas the Link Predictors reach top performance very early during training. This could be due to the fact that we based our choice of hyperparameters for the GNNs based on [6] which were likely already tuned to the architecture. Whereas we could not afford to do an extensive hyperparameter search for the Link Predictors. To confirm these results we would have to do a statistically valid hyperparameter search.
Finally, what is perhaps the most surprising is that for Link Predictors there is an extreme difference of 3 orders of magnitude in predictive power of SPD and Laplacian eigenfeatures, whereas the gap between both features becomes much smaller when using a GNNs. One possible interpretation the following. SPD features manage to capture 'neighborhood overlap', whereas Laplacian eigenfeatures does not cpature neighborhood overlap, but rather 'global node similarity'. We believe that the GNN's 'locality' implicit bias coming from the neural architecture compensates the lack of neighbourhood overlap measure from the Laplacian eigenfeatures. However, this is more speculation than result, and further investigation is needed.

6 Conclusion

We explored the Link Prediction problem on the specific task of predicting drug-drug interactions on the ogbl-ddi graph. This project helped us discover the Open Graph Benchmark challenges which has many different datasets with a wide range of applications. We also implemented graph data exploration and visualisation techniques seen in class to gain insight into the structure of the graph. We learned about and described a framework to do link prediction, made our own implementation of a link predictor network, and reproduced an existing top performing GNN architecture.

On the way, we aimed to answer two questions:

- What are good node embeddings for the task of Link Prediction?
- Are GNNs much more powerful than classical methods for Link Prediction?

Our answer to the first question is that node embeddings reflecting neighborhood overlap are good candidates for Link Prediction on the ogbl-ddi task. This can be achieved through handcrafted features such as SPD, or simply the use of a GNN. Our answer to the second question is that yes, GNNs do seem to work better to learn good node embeddings compared to handcrafted features. This is particularly true when seeing that Laplacian features perform poorly on for classical architectures, but work well when input to a GNN.

Unfortunately, the above answers are to be taken with a grain of salt since we were highly limited due to computational resources. Particularly to perform proper hyper parameter search, and proper evaluation by reporting the mean performance over several runs.

References

[1] Ogb ddi leaderboard, howpublished = https://ogb.stanford.edu/docs/leader_linkprop/#ogbl-ddi, note = Accessed: 2022-05-01.

[2] Ogb ddi python package, https://ogb.stanford.edu/docs/home/. Accessed: 2022-05-01.

[3] W. L. Hamilton. Graph representation learning. *Synthesis Lectures on Artificial Intelligence and Machine Learning*, 14(3):1–159.

[4] W. Hu, M. Fey, M. Zitnik, Y. Dong, H. Ren, B. Liu, M. Catasta, and J. Leskovec. Open graph benchmark: Datasets for machine learning on graphs, 2020.

[5] J. Y. Shitao Lu. Github link prediction with structural information.

[6] J. Y. Shitao Lu. Link prediction with structural information.

[7] J. L. William L. Hamilton, Rex Ying. Inductive Representation Learning on Large Graphs. 2017.

[8] L. H. Y. Z. H. S. S. C. Zhitao Wang, Yong Zhou. Pairwise Learning for Neural Link Prediction. 2022.

A Appendix

A.1 Eigenvectors

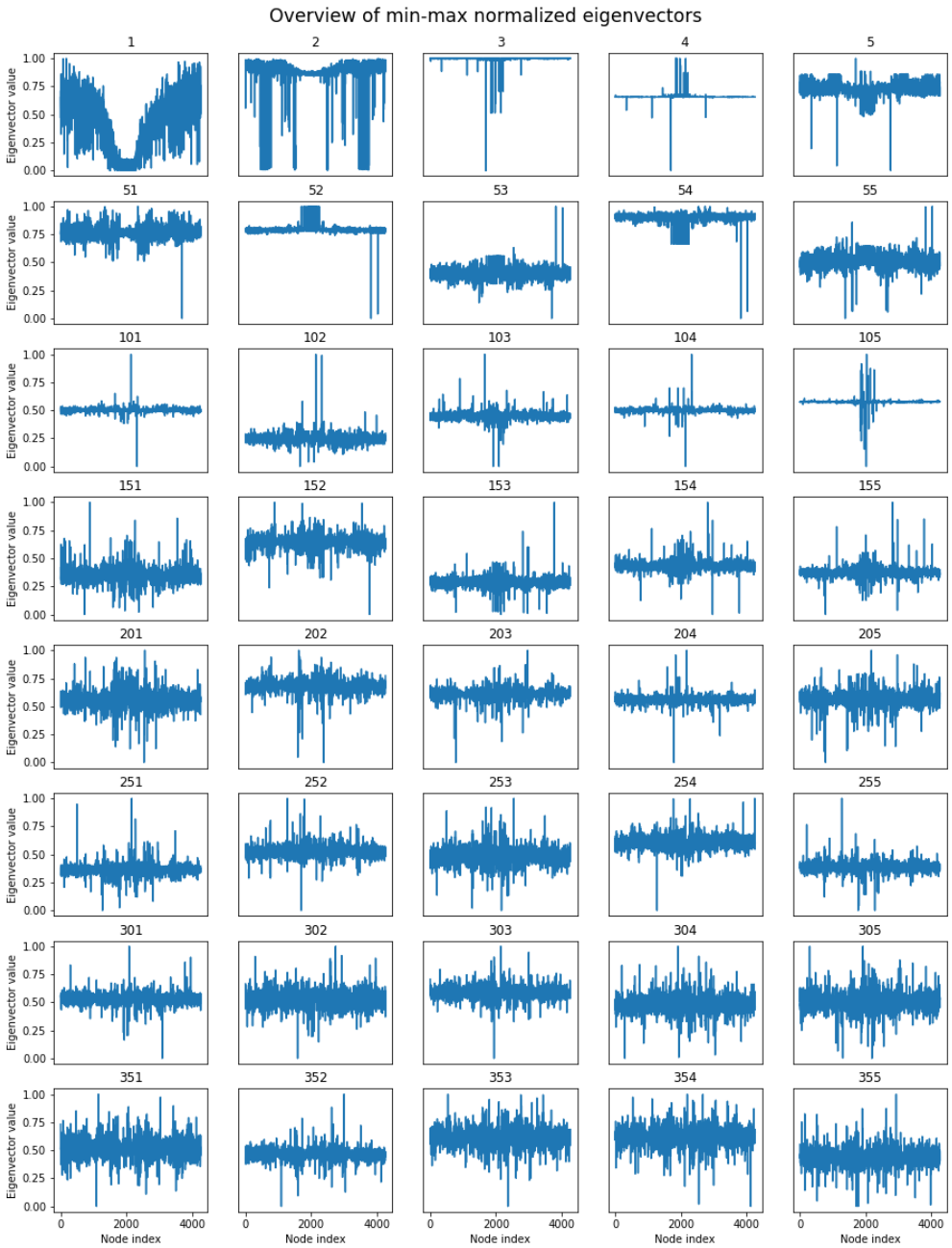


Figure 9: Figure displays sample of eigenvectors of N, where the subplot titles denote the index of the eigenvector corresponding to the eigenvalue with the same index. Worth noting is that the eigenvectors lose shape and gain noise with larger index. For high indices little changes from eigenvector to eigenvector, while for low indices one observes drastic changes.

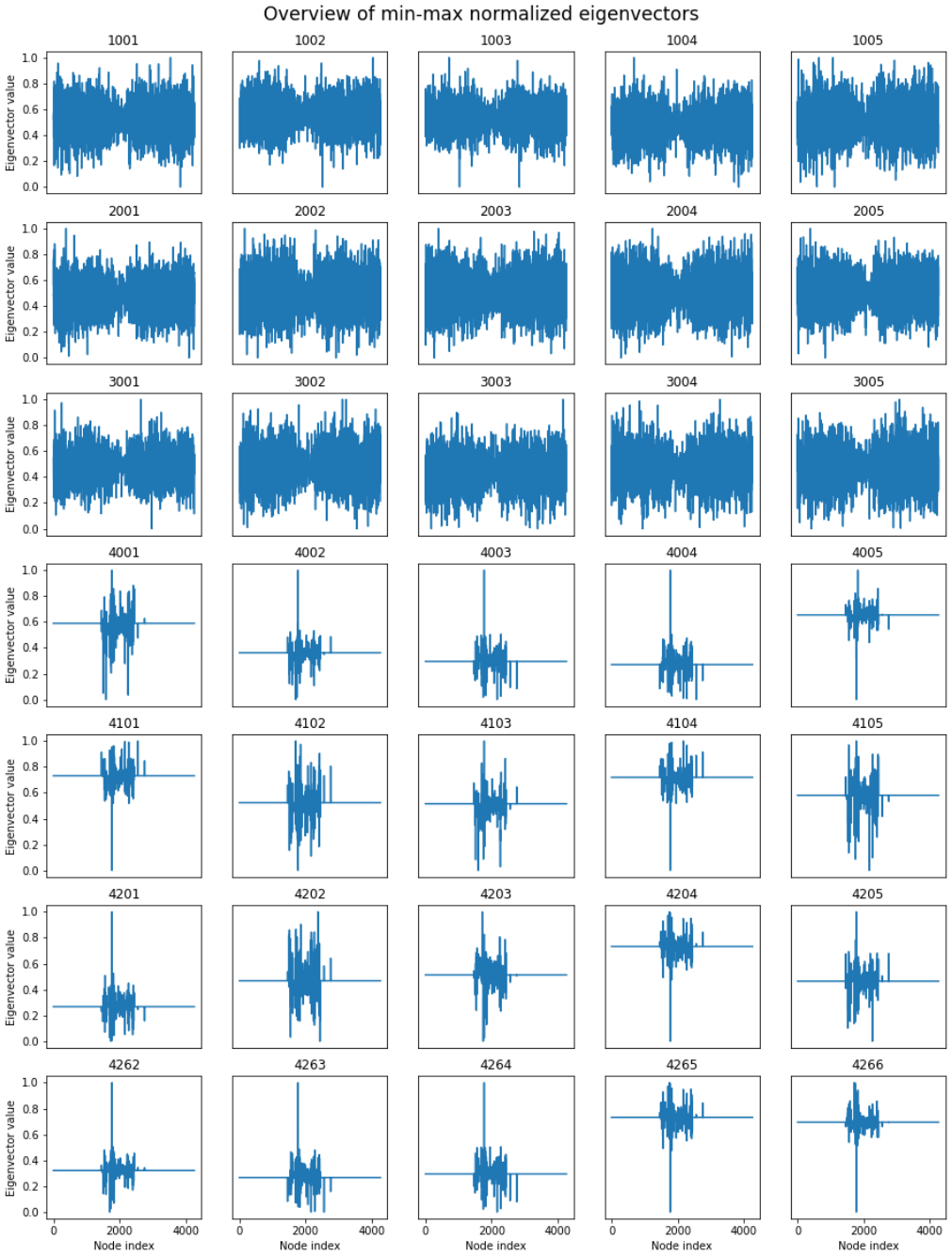


Figure 10: Like Figure 9, now with eigenvectors of higher indices showing some variation for the highest indices

A.2 Training curves for different Link Predictors

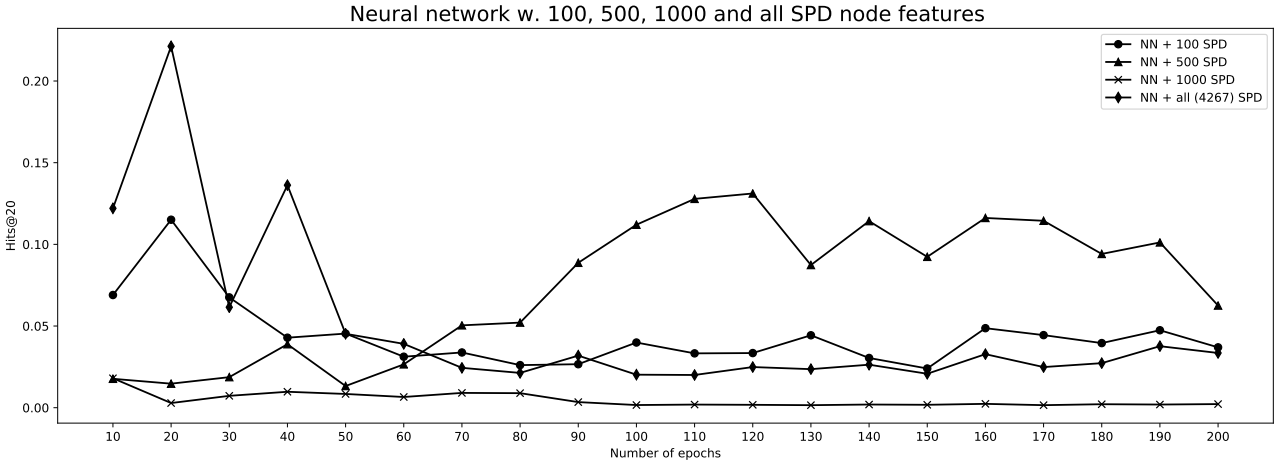


Figure 11: Hits@20 per epoch on test-set of Link Predictor trained on SPD node features for various number of features.

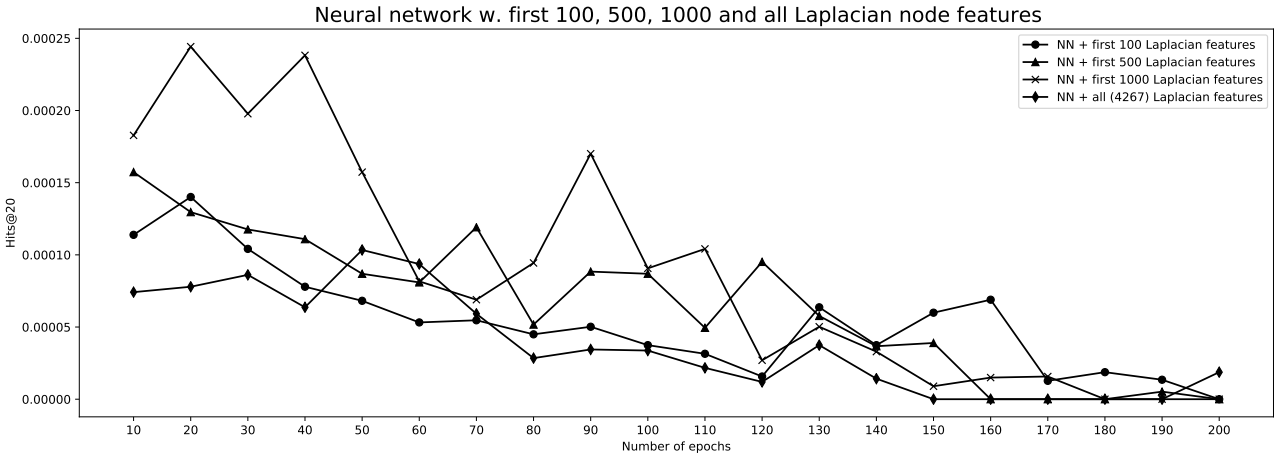


Figure 12: Hits@20 per epoch on test set of Link Predictor trained on Laplacian eigenfeatures for various number of features.