ΜΥΕ053–Επεξεργασία Φυσικής Γλώσσας

Kaggle Challenge

Team Name: Trump Tariffed My Datasets

The above photo is AI-Generated using ChatGPT's 4o image generation, and used only for logo

Νικόλαος Γιαννόπουλος
ΑΜ: 5199
E-mail: cs05199@uoi.gr

Contents

Introduction

The goal of this challenge is to build a machine learning model capable of predicting whether one research paper cites another, given a large citation network. This is framed as a Link-Prediction problem.

We are provided with the following datasets:

- abstracts.txt: raw textual abstracts of the papers,
- authors.txt: the list of authors associated with each paper,
- edgelist.txt: an existing citation graph representing known citations,
- test.txt: a list of paper pairs for which prediction must be made and then upload to Kaggle.

To address this challenge, a structured workflow was followed:

1. Data Preprocessing: This initial step ensured all datasets were cleaned, aligned, and formatted for later stages. Proper preprocessing was critical for compatibility and performance across various methods.
2. Embedding Techniques: Multiple approaches for generating embeddings were explored. This phase was both experimental and educational, helping to identify which representations were most informative, even if some methods proved ineffective in the final model.
3. Feature Engineering: Considerable effort was devoted to designing features that capture meaningful patterns in the data. This included both handcrafted logic-driven features and data-driven insights. Feature engineering was the most time-consuming and delicate part of the pipeline, especially due to the risk of overfitting.
4. Model Classification: In the final stage, various classification models were tested and fine-tuned. The focus was on achieving high accuracy while maintaining generalization to unseen data. Avoiding overfitting was a key consideration during model selection and tuning.

Due to the complexity and size of the datasets, the process was computationally intensive and time consuming. The methods used relied heavily on hardware.

The challenge was accomplished on a computer with the following characteristics:

CPU	Ryzen 5 5600X
RAM	32Gb DDR4 3600MT/s
GPU	RTX 5070 Ti
OS	Windows 11
LANGUAGE	Python 3.12 with Jupyter Notebook
IDE	Pycharm 2024.1

Finally, throughout development, multiple Python libraries were used, with careful version control to avoid dependency conflicts.

Preprocessing

Authors Preprocessing

The authors.txt file contained information about the authors of each paper in the dataset, where each line followed the format:

<node_id>|--|<author_names>

For the cleaning of the authors the following steps were followed:

1) Each line was read and then split using the custom delimiter |--| to separate the node from the corresponding authors.

2) Authors were then split from one another using commas, as many of them were included in the same string. Then, each author:
   a) Was converted to lowercase to ensure consistency
   b) Was stripped of whitespace
   c) Cleansed of punctuation using a regular expression to retain only alphanumeric characters and whitespace.

   All this ensured that variations in formatting (e.g. punctuation or capitalization) did not result in distinct representations of the same author.

3) Finally, all the results were saved in a form of dictionary, mapping each node to it's corresponding list of cleaned authors. The file(authors_preprocessing.pkl) was created for future use.

The cleaned authors have the following form:

node	author	cleaned_author
0	James H. Niblock,Jian-Xun Peng,Karen R. McMenemy,George W. Irwin	[james h niblock, jianxun peng, karen r mcmenemy, george w irwin]
1	Jian-Xun Peng,Kang Li,De-Shuang Huang	[jianxun peng, kang li, deshuang huang]
2	J. Heikkila	[j heikkila]
3	L. Teslic,B. Hartmann,O. Nelles,I. Skrjanc	[l teslic, b hartmann, o nelles, i skrjanc]
4	Long Zhang,Kang Li,Er-Wei Bai,George W. Irwin	[long zhang, kang li, erwei bai, george w irwin]

Abstracts Preprocessing

The abstracts.txt file contained the textual abstracts of each paper, with each line in the format:

<node_id>|--|<abstracts_text>

For the tokenization of the abstracts the following steps were followed:

1) Each line was read and then split using the custom delimiter |--| to separate the node from the abstract text.

2) Then, a custom function was applied to clean and tokenize each abstract:
   a) All text was converted to lowercase for uniformity.
   b) All non-alphanumeric characters (except whitespace) were stripped out using regular expressions.

c) The cleaned text was then split into individual word tokens using word_tokenize().

d) Common English stopwords (like "the", "is", "and") were removed using NLTK's stopwords list.

e) Each word was lemmatized using WordNetLemmatizer to reduce it to its base or dictionary form (e.g. "studies" to "study").

All these preprocessing steps helped standardize the textual content of the abstracts and reduce noise. By lowercasing, removing punctuation, eliminating stopwords, and lemmatizing words, it was ensured that semantically similar words (e.g., "methods" and "method") were treated as the same token. This normalization is crucial for the next step which is the Textual Embeddings.

3) The result was a dictionary mapping each node to a list of cleaned tokens from its abstract. This dictionary was then saved as a file (abstracts_tokenized.pkl) for future use.

The final representation of the abstracts has the following form:

```
0: ['development', 'automated', 'system', 'quality', 'assessment', 'aerodrome', 'ground', 'lighting', 'agl',
'accordance', 'associated', 'standard', 'recommendation', 'presented', 'system', 'composed', 'image', 'sensor',
'placed', 'inside', 'cockpit', 'aircraft', 'record', 'image', 'agl', 'normal', 'descent', 'aerodrome', 'modelbased',
'methodology', 'used', 'ascertain', 'optimum', 'match', 'template', 'agl', 'actual', 'image', 'data', 'order',
'calculate', 'position', 'orientation', 'camera', 'instant', 'image', 'acquired', 'camera', 'position', 'orientation',
'data', 'used', 'along', 'pixel', 'grey', 'level', 'imaged', 'luminaire', 'estimate', 'value', 'luminous', 'intensity',
'given', 'luminaire', 'compared', 'expected', 'brightness', 'luminaire', 'ensure', 'operating', 'required', 'standard',
'metric', 'quality', 'agl', 'pattern', 'determined', 'experiment', 'real', 'image', 'data', 'presented', 'demonstrate',
'application', 'effectiveness', 'system'],
1: ['paper', 'proposes', 'novel', 'hybrid', 'forward', 'algorithm', 'hfa', 'construction', 'radial', 'basis', 'function',
'rbf', 'neural', 'network', 'tunable', 'node', 'main', 'objective', 'efficiently', 'effectively', 'produce', 'parsimonious',
'rbf', 'neural', 'network', 'generalizes', 'well', 'study', 'achieved', 'simultaneous', 'network', 'structure',
'determination', 'parameter', 'optimization', 'continuous', 'parameter', 'space', 'mixed', 'integer', 'hard',
'problem', 'proposed', 'hfa', 'tackle', 'problem', 'using', 'integrated', 'analytic', 'framework', 'leading',
'significantly', 'improved', 'network', 'performance', 'reduced', 'memory', 'usage', 'network', 'construction',
'computational', 'complexity', 'analysis', 'confirms', 'efficiency', 'proposed', 'algorithm', 'simulation', 'result',
'demonstrate', 'effectiveness']
```

Graph Split

The edgelist.txt file defined the citation relationships between papers, with each line formatted as:

```
<source_node>, <target_node>
```

The preprocessing that follows takes account the fact that the data will be used to train models(e.g. Bert, SPECTER) to extract the textual embeddings. Thus:

1) The edgelist was loaded into a NetworkX undirected graph and converted to a StellarGraph object to support edge sampling with EdgeSplitter(could also be done with PyG – PyTorch Geometric Graphs but StellarGraph was easier to use).
2) A global edge split was performed using the StellarGraph's EdgeSplitter. It sampled approximately 272,988 positive and 272,988 negative edges to form the training set(finetuning and validation phase later). The graph remained connected after each split (keep_connected=True), preserving structural consistency for embedding models.

The sampling didn't happen based on arbitrary proportions (like 75% train, 25% test on the whole edgelist), a fixed number of positive and negative samples was selected for two key reasons:

i. Link prediction with models like BERT or SPECTER is cast as a binary classification task (link vs. no link). Balanced datasets (equal positive and negative samples) help the model avoid class imbalance, which could otherwise bias it toward predicting non-links (the majority class in real graphs).
ii. Citation graphs can be large, and using all possible edge pairs would be computationally expensive. Sampling a large but manageable subset of edges allows for efficient training while still providing enough variety and diversity for the model to generalize well. Basically, the 25% was chosen as a pragmatic tradeoff - large enough to train and evaluate with certainty, but small enough to keep training feasible in terms of time and memory. Choosing larger samples instead(such as 50%) would result in huge training times which would be disproportional to the performance gains induced by such a strategy.

For general usage of the edgelist.txt, we just utilize NetworkX to split the pairs and then save the nodes and edges.

Abstract Embeddings

Data Loading and Usage

First of all, we load the train pairs and labels we just created, and using the sklearn's train_test_split() we split the data into 80% train and 20% validation. The validation set (val_pairs, val_labels) is then used to evaluate the quality of the learned textual embeddings. Specifically, it is split into positive and negative samples, and for each pair, the cosine similarity between the corresponding paper embeddings is computed. The average similarity scores of positive and negative pairs are then compared. This serves as a sanity check: a well-trained model should yield significantly higher cosine similarity scores for linked (positive) pairs compared to unlinked (negative) ones. A large gap between these averages indicates that the model has effectively learned to distinguish meaningful semantic connections, and that gap, that distinction, is what we aim for.

Model Training

To generate textual embeddings for each paper, we batch the abstracts and pass them through the selected model — which can be BERT, SciBERT, SPECTER, or a fine-tuned variant of SPECTER(spoiler-alert for later). The model name is specified upfront (model_name), and the corresponding tokenizer and encoder are loaded accordingly. Each abstract is tokenized and embedded in batches to efficiently utilize GPU memory and power. The resulting embeddings are stored in a dictionary (txt2feat) and then compiled into a final list (article_textual_embeddings) to be later saved as a file (.pt). This modular approach allows easy switching between different models.

At first, this phase started with BERT only, but the gap as will be shown bellow wasn't meaningful. After research, SciBERT came into play. SciBERT is a pretrained version of BERT trained on scientific data. This resulted in slightly better results, but still no real gap.

It was apparent that transformer-based models would be necessary, thus SPECTER (as suggested in class) was brought into the equation. SPECTER is a pretrained language model designed specifically for generating document-level embeddings of scientific papers. Unlike general-purpose models like BERT or even domain-specific ones like SciBERT, SPECTER is trained using citation relationships between documents. Basically, it leverages the structure of the citation graph itself as a supervision signal. This means it captures not just textual semantics but also relational, citation-based context. Crucially, SPECTER is designed to be effective out-of-the-box, without requiring task-specific fine-tuning, making it ideal for scenarios like this one where document similarity and link prediction are central.

The whole training process was done utilizing Pytorch's support of CUDA, reducing the time by huge margins and thus allowing continuous tweaking.

After the training we come up with the following results for each model used:

MODELS USED	Similarity on Positive	Similarity on Negative	Gap
BERT	0.87142039	0.82712536	**0.04429503**
SciBERT	0.860955	0.79847723	**0.06247777**
SPECTER	0.827263	0.6578483	**0.1694147**

Widening the gap - SPECTER Fine-tuning

During the feature engineering phase, there came a point where additional handcrafted features no longer led to meaningful performance improvements. Upon reviewing the current features and their impact, it became clear that while SPECTER performed well, its general-purpose embeddings could potentially be adapted more closely to our specific task. This motivated the decision to fine-tune SPECTER on our domain-specific citation graph data. Given SPECTER's citation-aware architecture, it was also the choice that made most sense at that point.

To this end, a fine-tuning procedure was set up using the positive and negative paper pairs from the training set(the 80% split mentioned before). Each pair was formatted as a SentenceTransformer InputExample, with their corresponding abstracts and binary labels (linked or not). The original SPECTER model was then fine-tuned over 3 epochs(could use 5 but it would have taken very long time and the improvements would be minimal). After training, the updated model was saved and later used to regenerate embeddings. This fine-tuning step noticeably increased the similarity gap between positive and negative pairs, confirming that the model had learned more meaningful document representations for the task.

The whole fine-tuning process was both time-consuming and computationally intensive, with the GPU running at full throttle for hours - enough to heat a small apartment. This made the earlier data split especially valuable, as it allowed training on a large enough sample to be effective, while keeping runtime, electricity bills, and potential fire hazards within reason. Without that controlled subset, the cost, in time, energy, and perhaps personal comfort, would have skyrocketed with little added benefit.

The final gains from the fine-tuned SPECTER:

MODELS USED	Similarity on Positive	Similarity on Negative	Gap
BERT	0.87142039	0.82712536	**0.04429503**
SciBERT	0.860955	0.79847723	**0.06247777**
SPECTER	0.827263	0.6578483	**0.1694147**
SPECTER fine-tuned	0.82576215	0.14071403	**0.68504812**

Author and Citation Graph Embeddings

Node2Vec Co-Authorship Graph Method

In an attempt to incorporate additional graph-based signals, I experimented with building a co-authorship network. The idea was to capture scholarly collaboration patterns that might correlate with citation behavior.

Basically, what I did was:

1. Using the preprocessed authors_preprocessing.pkl file, which contained author lists for each paper, I first extracted and cleaned valid entries. Only papers with at least two authors were considered since co-authorship by definition requires a pair.

   From these cleaned lists a unique pair of co-authors was treated as an edge. These edges were then added to a graph using NetworkX, where:

   - Nodes = individual authors
   - Edges = co-authorship relationships

   The resulting graph represents the entire collaboration network across the dataset — essentially a social graph of scientists.

2. To transform this graph structure into usable numerical features, Node2Vec was applied, a graph embedding technique that learns vector representations for nodes based on biased random walks

   The model learns a vector for each author based on their position in the graph and how they're connected to others. Authors who often co-author together or share common co-authors are placed closer in embedding space.

   This embedding helps capture collaboration similarity or proximity, which could (in theory) correlate with citation likelihood.

3. To make the embeddings portable and reusable, they were saved using:

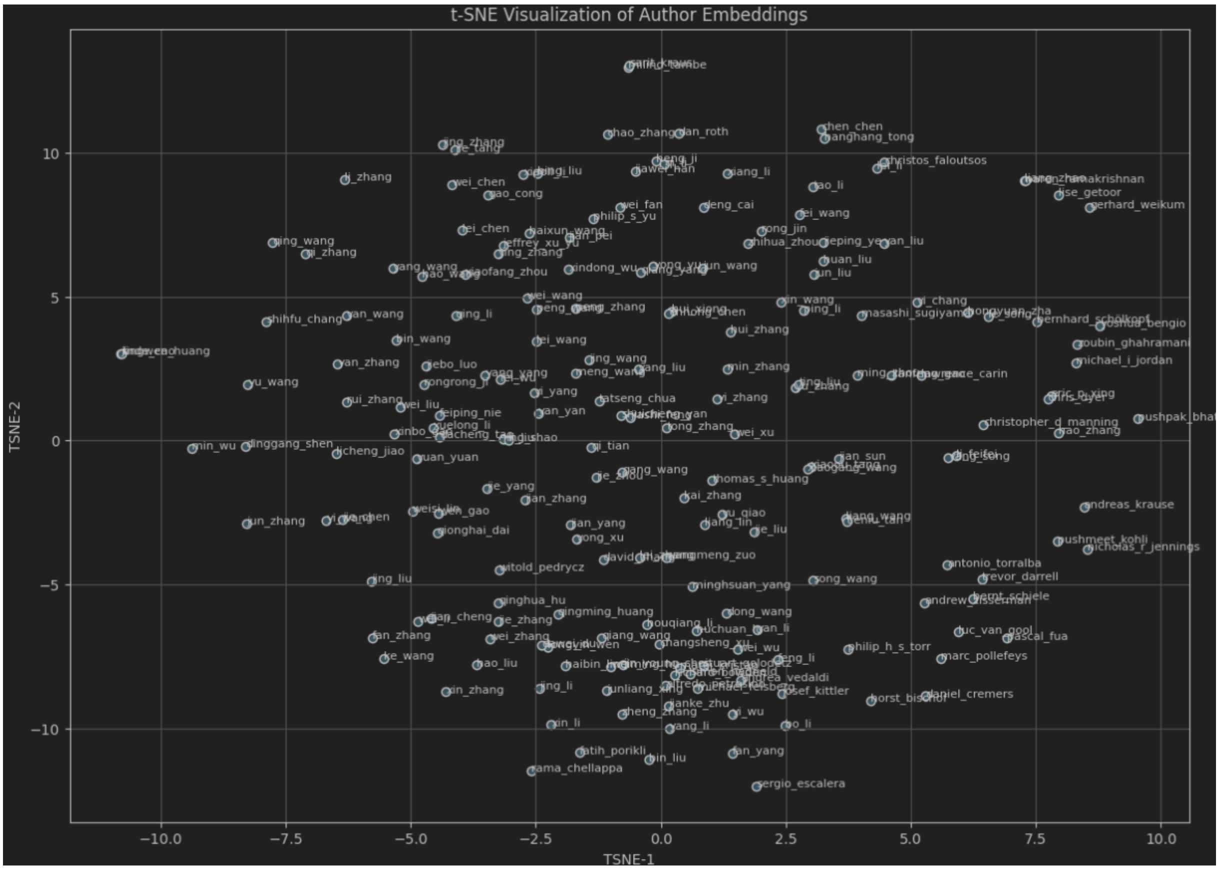
   ```
   cleaned_kv.save_word2vec_format("author_embeddings_cleaned.txt", binary=False)
   ```

   This stores the learned vectors in a standard Word2Vec plain-text format, which is compatible with gensim's KeyedVectors, a convenient way to reload and query them later (e.g., for similarity, clustering, or downstream tasks).

4. To inspect the quality of the embeddings, they were reloaded using KeyedVectors and then a sample of 200 authors was picked.

   Then, by reducing the high-dimensional vectors to 2D using t-SNE, a popular dimensionality reduction technique, it was possible to finally, visualize them with matplotlib, labeling each point with the author's name.

   The resulting plot gives a visual sense of the author space as clustered authors are likely connected, either directly or through shared co-authors:



Walklets Citation Graph Method

As part of broader efforts to engineer better features from graph structures, another avenue I explored was learning unsupervised structural embeddings using the karateclub's Walklets algorithm. Unlike Node2Vec, which I had previously applied to the co-authorship network, this time the focus shifted to the citation graph, where papers are linked directionally through citations (e.g. paper A cites paper B).

Walklets is a scalable node embedding method that builds upon skip-gram models (similar to Word2Vec) but with higher-order random walks, allowing it to capture both local and global graph structure more efficiently. The hope was that papers sharing similar citation neighborhoods, either by citing similar works or being cited by similar ones, would yield meaningful low-dimensional representations that could later be plugged into the main model.

Basically, the citation graph was built from the edgelist.txt. Then, Walklets was configured in a way that wouldn't take too long to run.

The Walklets Configuration consists of:

- walk_number: how many random walks to launch per node
- walk_length: how long each walk should be
- diensions: the size of the resulting embedding vectors
- window_size: context window used during training
- workers: the amount of threads created

In the end, the result is saved.

Conclusions

Both the co-authorship network and the citation graph seemed like natural candidates for extracting meaningful structural information. Using Node2Vec on the co-author graph and Walklets on the citation graph, dense vector representations for authors and papers were generated, hoping these embeddings would capture latent relationships that could boost performance. However, despite the conceptual appeal, neither approach ultimately led to a notable improvement in log-loss or other metrics. Quite the contrary, they increased the log-loss.

There are several likely reasons as to why it may have happened:

- Mismatch with the task: The embeddings captured structural or social proximity, but the prediction task (whether two papers are linked) relied more heavily on semantic similarity than graph topology.
- Graph sparsity and noise: The co-authorship graph had lots of overlapping author names and possible inconsistencies. This overlap created a huge noise, which was really apparent on the t-SNE for higher numbers of sampled authors. Similarly, the citation graph lacked context around the nature of citations (e.g. background vs. core reference).
- Embedding integration challenges: Combining these embeddings with textual features proved non-trivial. Combining these features with others, didn't yield any significant change and mostly caused noise, which, in return, decreased performance.

In the end, these were interesting experiments that added to the understanding of the data, but didn't make the final feature set, so they act more like honorable mentions than main contributors.

Important note is that during initial testing for the early bonus, a TF-IDF was created, although it didn't end up to the final program. More over, Doc2Vec as well as Word2Vec were also tried out to see potential gains in performance. Since not much attention were given to them, and they are neither included in the code, the is no reason to elaborate more on them.

Feature Engineering

Main Idea

The main idea for this part of the project was to find, at first, as many features as possible and then try to eliminate them one by one until the final feature set is built. The process of designing features was driven by a combination of domain intuition, exploratory experimentation, and good old-fashioned trial and error. The goal was to construct features that could help the model distinguish between paper pairs that likely have a citation link and those that don't based on how papers "look," "sound," and "connect."

For the Test set, the test.txt was utilized by creating a feature set for it as well and then, later on, used only to predict.

Also, in this part, research was conducted into various scientific documents regarding the features used.

The total time required to create each feature matrix was:

- The training takes about **15 minutes** on the processor(CUDA was not available), using float32(although no difference with float16).
- Adding Node2Vec and Walklets, increases the training to **45 minutes**.
- For the Test matrix creation, the time required is **1-2 minutes**.

Included Features

All the included features could be split into 4 distinct categories:

Abstract-based Features

- Length-based features (total and difference): Simple proxies for abstract complexity or imbalance.
- Common words: Captures direct textual overlap. If two papers use the same terms, they might be linked.
- Cosine similarity via token counts: Using a custom cosine similarity over counter objects, this measures word frequency overlap a rough but effective proxy for semantic similarity without relying on embeddings.

Author-based Features

- Number of authors, difference in team size: Paper teams vary in size, thus imilarity may imply shared research practices.
- Shared authors: A strong positive signal as authors tend to cite their own work or their collaborators.
- Cosine similarity of author lists: Again using counter cosine similarity, this quantifies overlap in authorship by weighting repeated names and partial overlaps which is especially useful for larger teams where exact matches are rare.

Graph-based Features

For the graph-based features is where most research was conducted thus the following features were included:

- Jaccard coefficient[1]: Measures common neighbors, a classical local similarity metric.

$$J(u,v) = \frac{|\Gamma(u) \cap \Gamma(v)|}{|\Gamma(u) \cup \Gamma(v)|}$$

- Triangles and k-core values: Capture local clustering and connectivity as papers in dense regions tend to be more interlinked.

$$T(u,v) = |\{w : w \in \Gamma(u) \cap \Gamma(v) \; and \; (u,w),(v,w) \in E\}|$$

$$k(u), k(v) (core \; numbers \; of \; nodes \; u \; and \; v)$$

- Degrees sum: Reflects how globally connected the two papers are. A higher value suggests both papers are active in the citation network — possibly making a link between them more plausible.

$$Dsum(u,v) = \deg(u) + \deg(v)$$

- Degrees difference: Captures whether the papers are similarly popular or central. A smaller value implies a more balanced relationship, while a large difference may indicate a weak or one-sided connection.

$$Ddiff(u,v) = |\deg(u) - \deg(v)|$$

- Adamic-Adar[2]: Weights common neighbors of two nodes by their rarity. Shared neighbors that are not very connected are considered more meaningful.

$$AA(u,v) = \sum_{w \in \Gamma(u) \cap \Gamma(v)} \frac{1}{logdeg(w)}$$

- Salton[3] Index: It balances the overlap against how "social" each node is, emphasizing proportion rather than just volume of connection.

$$S(u,v) = \frac{|\Gamma(u) \cap \Gamma(v)|}{\sqrt{deg(u) \cdot deg(v)}}$$

- Hub Depressed Index: Penalizes one-sided high-degree nodes. If one paper is a hub and the other is obscure, their shared neighbors count less.

$$HD(u,v) = \frac{|\Gamma(u) \cap \Gamma(v)|}{max(deg(u), deg(v))}$$

- Resource Allocation[4] Index: Like Adamic-Adar, but without the log. It assumes each common neighbor "splits its attention" evenly, thus rarer neighbors give stronger signals.

$$RA(u,v) = \sum_{w \in \Gamma(u) \cap \Gamma(v)} \frac{1}{deg(w)}$$

- Preferential Attachment[5]: Based on the idea that popular nodes are more likely to connect.

$$PA(u,v) = deg(u) \times deg(v)$$

- Common[6]: This feature counts how many neighbors (connected nodes), two nodes of the graph, share in the graph.

$$CN(u,v) = |\Gamma(u) \cap \Gamma(v)|$$

Specter Features

- Cosine similarity: measures the cosine of the angle between two embedding vectors. It captures how directionally aligned the vectors are, regardless of their magnitude. As proven later, this metric is particularly effective in identifying how closely related two papers are in terms of content, even if their abstracts differ in length or scale.
- L2 Distance (Euclidean distance): measures the straight-line distance between the two embedding vectors in the high-dimensional space. Unlike cosine similarity, it is sensitive to the magnitude of the vectors. Smaller L2 distances indicate closer embeddings, suggesting higher similarity between the papers' abstracts.

Features that didn't work

As mentioned before, the co-authorship and citation graphs didn't help and instead increased the log-loss score. Additionally, here are some features who were removed from the final feature set due to weak or negative performance:

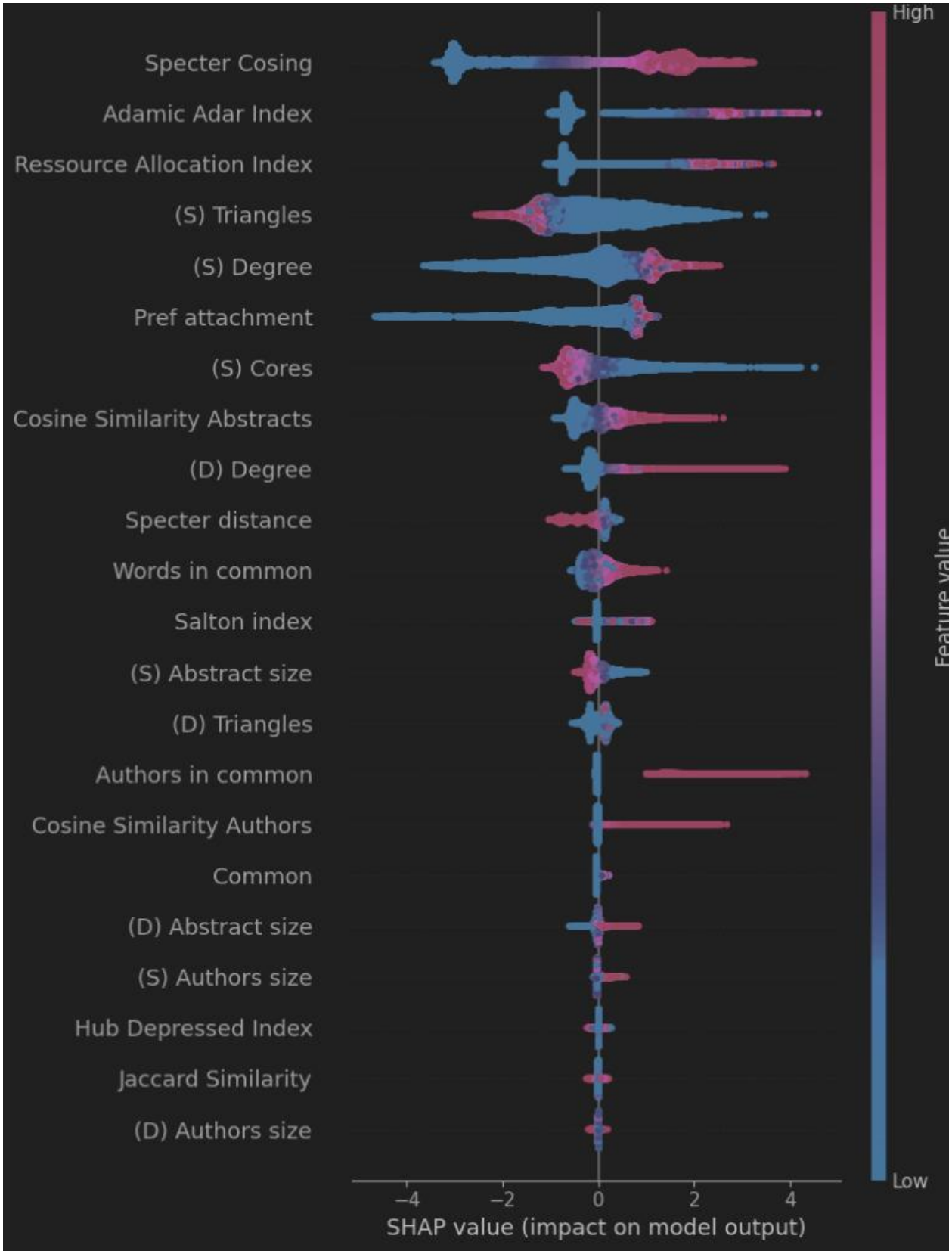
- Neighbors[7] (Average Neighbor Degree):
  This feature measures the average degree of a node's neighbors, capturing how well-connected a paper's immediate network is. It can reflect the local connectivity structure, but in this dataset, it didn't add meaningful signal beyond other degree-based features and also obstructed impact from other more significant.
- PageRank[8]:
  PageRank scores estimate the importance or influence of a node within the network, based on the number and quality of incoming links. Although theoretically useful to highlight key papers(and also used by Google), the metric did not correlate strongly with the link prediction task here, possibly due to the dense or noisy structure.
- Onion Decomposition[9]:
  This measures the core structure of the graph by peeling it like an onion, identifying layers of connectivity. While it can reveal nodes' structural roles, the decomposition did not improve prediction accuracy, mainly because it obstructed features such as Triangles or Cores.
- Clustering Coefficient[10]:
  The clustering coefficient quantifies how connected a node's neighbors are to each other, indicating local community density. Despite being a common network feature, it did not enhance the model.
- Greedy Coloring[11]:
  This feature assigns colors to nodes such that no two adjacent nodes share the same color, aiming to approximate a minimal coloring of the graph. It helps identify groups of papers that are structurally separated. However, in this context, it didn't improve performance, and even made it worse by a lot since it had a huge impact on the final result. In comparison, removing it, reduces the log-loss by 3 times!
- Shortest Path[12]:
  This feature, benefiting those nodes with shortest path resulted in overtraining and even impacted too highly, shadowing the other features and thus resulting in worse results.

Judging Methodology

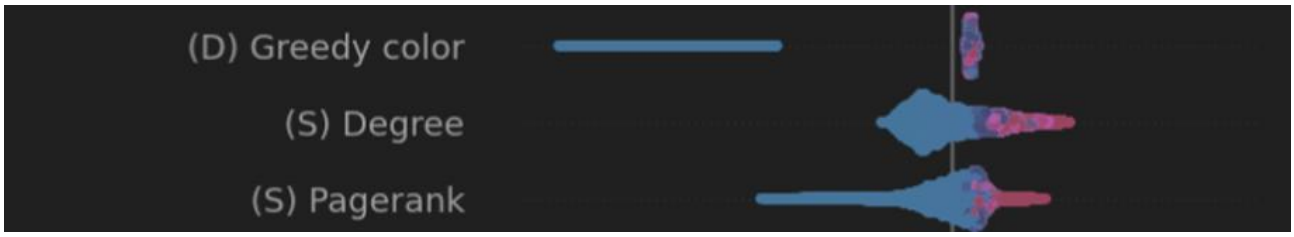
To judge, whether or not one feature was useful many methods were used.

First of all, a basic(not hyperparameter tuned) XGBoost was utilized to produce a model which was then validated using cross-validation and with metric on log-loss. This gave a picture whether a change was good or bad.

On deciding what change, a SHAP graph was constructed to show the importance and impact of every feature in the final result. Here is the SHAP graph of the final set of features:

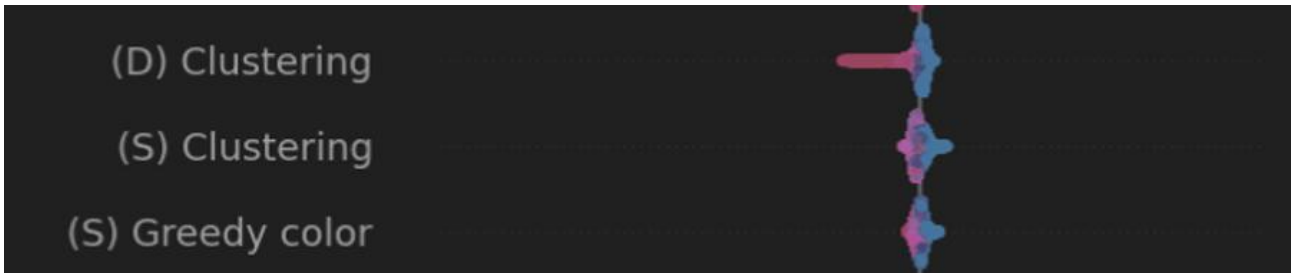


The methodology was a combination of intuition and observation; features that seemed to prefer negative/positive SHAP values, while at the same time not contributing, at all, to the opposite, usually held back the score. For example, the Greedy Color seemed very biased toward specific predictions:



Another thing that was experimented was removing high impact features to see which features increased their impact. For example, an increase of the SPECTER impact was something sought after. To validate, yet again, cross-validation run to approve of the change.

Some features in the other hand were unimpactful offering only noise to the result. Such features were removed. Not all low-impact features were removed, however. Basic features, such as abstract and author token overlaps, were often retained for their interpretability and foundational value. The main pruning focused on graph-based features that appeared to add noise or overlap with more useful ones.



Finally, the submission, many times, approved that the thinking was right and, indeed the public score decreased. Although, must be noted, that decrease wasn't taken for granted since the avoidance of overfitting the testing set was of absolute necessity and, for this, the leaderboard performance was used as a sanity check, not the main guide for important changes. Another thing to take into account is that, public score wouldn't be 100% accurate since every training run, had a basic divergence of about 0.01 to the log-loss.

In the end, the goal was to balance the final model's performance, generalization, and interpretability, and that's why, a very basic model(XGBoost without being fine-tuned) was used.

Model Classification

After finalizing the feature set, the next step was to build and fine-tune the models used for training and evaluation. To achieve optimal performance, the Optuna library was employed for automated hyperparameter optimization. Optuna creates a study environment in which it runs multiple trials, systematically exploring combinations of parameters within defined ranges to minimize a chosen objective function, which in this case is the log-loss via cross-validation.

In addition to automated tuning, manual adjustments based on domain knowledge and prior experience(Machine Learning course) were also considered. For example, unusually high or low learning rates suggested by Optuna were occasionally corrected to avoid unstable training behavior.

The models trained and evaluated in this phase were:

- XGBoost
- LightGBM
- CatBoost
- Logistic Regression
- Random Forest
- Stacking Classifier (CatBoost + XGBoost + LightGBM) with Logistic Regression

Each of these models underwent separate Optuna optimization, and performance was validated using cross-validation to ensure generalization. The trials were enough to determine the best values and in the same time, weren't as many as to overfit the model. In addition, the whole process takes a lot of time and it becomes imperative that a reasonable number of trials is considered.

Logistic Regression and Random Forest didn't use Optuna since running times on the given hardware were beyond rational, thus the testing became by hand.

As for the cross-validation itself, it was decided that splitting into 5 parts was more than enough as to avoid overfitting.

In the final stages, a stacking ensemble was constructed to combine the strengths of different models, with CatBoost and XGBoost serving as base learners. This would, in theory. have helped improve robustness and mitigate individual model weaknesses. In the actual usage environment however, it didn't prove as useful.

Hyper-parameter Tuning

Each model has different parameters so different things had to be considered for each one.

XGBOOST

The total time required for XGBoost was about an hour, while using CUDA. The following parameters were tackled:

- n_estimators: The number of boosting rounds (trees). More trees may improve learning, but also increase overfitting risk and training time.
- max_depth: Maximum depth of each tree. Deeper trees can model more complex patterns but might overfit.
- learning_rate: How much each tree contributes to the overall prediction. Lower values mean slower but more stable learning, although it could lead to overfitting.

- subsample: The fraction of samples used for training each tree. Helps with regularization by introducing randomness.
- colsample_bytree: Fraction of features used per tree. Like subsample, this reduces overfitting and increases diversity.
- min_child_weight: Minimum sum of instance weight (hessian) needed in a child. Controls overfitting by requiring a minimum number of samples in a node.
- eval_metric ('logloss'): Specifies the evaluation metric used during training. Log-loss is standard for binary classification.
- use_label_encoder=False: Prevents automatic label encoding, which is deprecated and not needed here.

LIGHTGBM

The total time required for LightGBM was about 2 hours and 20 minutes, as CUDA was not available despite what said about gpu offloading. The following parameters were tackled:

- n_estimators: Number of boosting iterations (trees).
- learning_rate: Step size shrinkage for each tree's contribution. Critical for convergence and performance.
- max_depth: Limits tree depth. Prevents overfitting and reduces training time.
- num_leaves: Number of leaf nodes per tree. Higher values allow more complexity, but can overfit if too high.
- min_child_samples: Minimum number of data points required in a leaf. Acts as a regularization mechanism.
- subsample: Fraction of data used per tree. Adds randomness and prevents overfitting.
- colsample_bytree: Fraction of features sampled per tree, promoting model diversity.

CATBOOST

The total time required for CatBoost was about 1 hour while using CUDA. During the tuning, the following parameters were tackled:

- learning_rate: Controls how much each new tree influences the model. Smaller is safer but slower, still with the risk of overfitting however.
- depth: Maximum tree depth. Controls model complexity.
- l2_leaf_reg: L2 regularization term on leaf values. Helps reduce overfitting.
- border_count: Number of splits (bins) used to discretize continuous features. Higher can increase accuracy but also overfitting and memory use.

LOGISTIC REGRESSION

The 'liblinear' solver is a good default choice for smaller datasets and supports both L1 and L2 regularization. It works well for binary classification. Not many things were tried here since it was proved early on as the weakest of the models, at least when it was tried alone.

RANDOM FOREST

In random forest, the only thing that changed were the trees. To, evaluate this method, through cross-validation, takes about half an hour.

MODEL STACKING WITH LOGISTIC REGRESSION

Model stacking is an ensemble technique where multiple base models are trained independently, and their predictions are combined by a "meta-model" (also called a final estimator) that learns how to best blend their outputs. This approach leverages the strengths of different models to improve overall predictive performance and reduce overfitting.

To achieve this, XGBoost, CatBoost and LightGBM were used since they were the models with the best log-loss.

The final estimator in the stacking ensemble is Logistic Regression(and actually the place were Logistic Regression proved useful), which acts as a meta-classifier. It takes the predicted probabilities from the base models as input features and learns the optimal weighted combination to minimize prediction error (log-loss in this case).

Notably, Logistic Regression proved especially useful here because:

- It is simple and interpretable, reducing the risk of overfitting at the meta-level.
- It efficiently combines probabilistic outputs from heterogeneous models.
- It allows smooth calibration of predictions, improving overall log-loss.

The stacking model was validated using cross-validation with log-loss. Although interesting process, the results didn't live up to the expectations.

Final Results

The local log loss, computed through Cross-validation, is shown on the following table.

As it will be visible, basic models such as Logistic Regression and Random Forest didn't do much, while boosting models proved to be up to the task. Stacking, instead of helping, probably lead to overfitting, thus the worse results.

BASIC MODELS	LOGISTIC REGRESSION	RANDOM FOREST
LOG-LOSS	**0.2211274328744099**	**0.1284489195843651**

MODELS	XGBoost	LightGBM	CatBoost	Stack(CatBoost+XGBoost)	Stack(all)
LOG-LOSS	**0.10394**	**0.10585**	**0.10282**	**0.1202**	**0.14433**

To avoid possible overfitting that may have happened to any models, XGBoost and CatBoost were independently chosen as submissions despite the score.

Sources and Citations

PyTorch

CUDA Toolkit - Free Tools and Training | NVIDIA Developer

StellarGraph Machine Learning Library — StellarGraph 1.2.1 documentation

allenai/scibert_scivocab_uncased · Hugging Face

allenai/specter · Hugging Face

https://neo4j.com/docs/graph-data-science/current/algorithms/linkprediction/

https://networkx.org/documentation/stable/index.html

XGBoost Documentation — xgboost 3.0.2 documentation

Welcome to LightGBM's documentation! — LightGBM 4.6.0.99 documentation

CatBoost - open-source gradient boosting library

(1)M. J. van Rijsbergen, *"Information Retrieval,"* Butterworth-Heinemann, 1979.

(2)L. A. Adamic and E. Adar, *"Friends and neighbors on the Web," Social Networks*, vol. 25, no. 3, pp. 211–230, 2003.

(3)G. Salton and M. J. McGill, *"Introduction to Modern Information Retrieval,"* McGraw-Hill, 1983.

(4)T. Zhou, L. Lü, and Y.-C. Zhang, *"Predicting missing links via local information," The European Physical Journal B*, vol. 71, no. 4, pp. 623–630, 2009.

(5)A.-L. Barabási and R. Albert, *"Emergence of scaling in random networks," Science*, vol. 286, no. 5439, pp. 509–512, 1999.

(6)Liben-Nowell, D., & Kleinberg, J. (2007). *The link prediction problem for social networks*. Journal of the American Society for Information Science and Technology, 58(7), 1019–1031.

(7)M. E. J. Newman, *"Assortative Mixing in Networks," Physical Review Letters*, vol. 89, no. 20, 208701, 2002.

(8)L. Page, S. Brin, R. Motwani, and T. Winograd, *"The PageRank Citation Ranking: Bringing Order to the Web,"* Technical Report, Stanford InfoLab, 1999.

(9)H. Hébert-Dufresne, J. A. Grochow, and A. Allard, *"Multi-scale structure and topological anomaly detection via a new network statistic: The onion decomposition," Scientific Reports*, vol. 6, Article number: 31708, 2016.

(10)D. J. Watts and S. H. Strogatz, *"Collective dynamics of 'small-world' networks," Nature*, vol. 393, pp. 440–442, 1998.

(11)M. R. Garey and D. S. Johnson, *"Computers and Intractability: A Guide to the Theory of NP-Completeness,"* W. H. Freeman, 1979.

(12)E. W. Dijkstra, *"A Note on Two Problems in Connexion with Graphs," Numerische Mathematik*, vol. 1, pp. 269–271, 1959.