Citation Link Prediction Kaggle Challenge

Ioannis Bouzas cs05025@uoi.gr, Stylianos Papaspyros cs05162@uoi.gr

Supervisor: Konstantinos Skianis kskianis@cs.uoi.gr

Department of Computer Science & Engineering, University of Ioannina
Natural Language Processing Course, Summer Semester 2024-2025

KEYWORDS

Citation
Link Prediction
Feature Extraction
Embeddings
Bert
Tree Models
SHAP analysis

INTRODUCTION

The task of link prediction—forecasting the existence of connections between entities in a network—has emerged as a central problem in modern graph analysis. In this report, we address the specific challenge of predicting citation links between academic papers in a large-scale citation network. Each paper is represented not only by its connections to other papers, but also by rich metadata, including its abstract and authorship. By integrating textual, structural, and authorship information, our goal is to develop a supervised learning pipeline capable of identifying potential future citations. This report begins by introducing the dataset and its structure, followed by a detailed explanation of our preprocessing steps. We then explore our initial approach to the problem, the design and extraction of various features, and highlight the importance of advanced techniques such as transformer-based models (e.g., BERT) and node embeddings in improving performance. Subsequent sections discuss the models we implemented, the evaluation metrics used—particularly log loss as defined by the competition—and present a thorough analysis of our experimental results. We also include a dedicated discussion on the challenges we encountered and the strategies used to address them. The report concludes with reflections on future directions, an overview of our best-performing solution, and the corresponding score. References to relevant literature and tools are included at the end.

1. Get to know our Data

The citation dataset comprises a total of 138,499 academic papers, each potentially represented in the citation network and associated with metadata including abstracts and authorship. Among these, 131,250 abstracts contain meaningful content, while 7,249 abstracts are empty, which introduces a potential challenge for text-based models. This discrepancy highlights the importance of handling missing textual data—either by inputting, filtering, or augmenting it using structural information from the citation graph or authorship data. Early inspection of sample abstracts confirms the technical and formal nature of the content, suggesting that embeddings from domain-adapted language models like SciBERT / DistilBERT and SentenceTransformers(more on them later) could yield better performance than general-purpose models. Moreover, the distribution of textual content—with frequent terms like *data*, *model*, *method*, *proposed*, and *algorithm*—indicates a strong methodological focus, which may enhance the effectiveness of content-based similarity metrics and embedding techniques for link prediction.

The citation network is a directed graph with over one million edges (1,091,955), averaging 7.88 in- and out-degrees per node, which implies a moderately sparse yet information-rich structure. The existence of nodes with very high degrees (up to 3,030 citations made and 273 received) introduces strong centrality and hierarchy into the network—these hubs may disproportionately influence predictions and require normalization or special treatment during feature extraction (e.g., personalized PageRank, normalized centrality features). The authorship data is also substantial, with 456,810 author-paper mentions across 149,682 unique authors, resulting in an average of 3.3 authors per paper. This collaborative nature of academic work suggests that author similarity and co-authorship graphs could be highly

informative features. High-frequency authors such as Dacheng Tao (426 papers) and Jiawei Han (340 papers) may act as hubs in the co-authorship graph, again reinforcing the value of incorporating author embeddings or similarity measures. These statistics collectively inform both our feature engineering strategy and our model design—justifying a multimodal approach that blends graph-based features, textual representations, and author metadata to capture the rich structure underlying citation behavior.

2. Preprocessing Phase

Before diving into our initial approach of the problem, we began with a comprehensive **data preprocessing pipeline**, particularly focusing on cleaning and normalizing the **abstracts**, which are central to the citation prediction task. Each abstract initially contained raw text with varying degrees of formatting, punctuation, and casing inconsistencies. For example, early samples included phrases which contained special characters, abbreviations, and capitalized words. Our first step was to apply a series of **basic Natural Language Processing (NLP) techniques** to standardize the text: we converted all characters to lowercase and removed special characters. We then removed stopwords to eliminate common but semantically weak words and tokenized the remaining content into individual words.

For text normalization, we opted for **stemming**, which simplifies each word to its root form by stripping suffixes. For example, "development" becomes "develop" and "assessment" becomes "assess." Although stemming can lead to non-dictionary words, it offers a **fast and computationally efficient** method suitable for large datasets like ours. While **lemmatization** (which maps words to their base dictionary forms) is often preferred for semantic clarity, it is computationally heavier, and in this early phase, stemming served our needs better. As a result, transformed abstracts now reflect concise, content-rich tokens ideal for embedding models and similarity measures.

It is important to note that while this pre-processing improved the quality of our abstract representations, we later encountered **issues related to data leakage** when working with the citation graph. As a result, we revised parts of the pipeline to ensure our preprocessing respected the **temporal and structural integrity** of the citation relationships. A more detailed discussion of these challenges and the adjustments we made is provided in a dedicated section later in this report.

3. Initial Approach: From Raw Data to First Predictions

At the early stages of our project, we mapped out a clear and structured process to follow: collect and preprocess the data, extract meaningful features, train and validate models, and finally test the model to evaluate its performance on Kaggle. This supervised learning pipeline served as our backbone throughout the project. After preprocessing the abstracts (as previously described), we moved to constructing an initial feature set, aiming to capture different types of similarity and interaction between papers.

Feature Engineering: Capturing Paper Relationships

We started with five core features. First, we used TF-IDF (Term Frequency–Inverse Document Frequency) to represent the abstracts in a reduced vector space of 500 dimensions. This allowed us to compute cosine similarity between pairs of paper abstracts, quantifying how similar their content is. This is especially valuable since semantically similar papers are more likely to cite each other.

Next, we explored authorship information by computing author overlap, defined as the Jaccard similarity of the sets of authors between two papers. This reflects the intuitive idea that researchers are likely to cite their own previous work or papers authored by colleagues in their circle.

We then extracted network-based features from the citation graph. The common neighbors metric counts how many papers are cited by both papers in question—higher counts may indicate shared research context. Similarly, Jaccard Coefficient measures the ratio of common neighbors to total unique neighbors, offering a normalized version of neighborhood overlap. Lastly, preferential attachment calculates the product of the out-degrees of the two papers, based on the principle that nodes with higher degrees are more likely to attract new links.

Data Splitting Strategy

For model evaluation, we mostly adopted an 80/20 train-validation split, which we found to provide stable and generalizable results.

Model Choice and First Results

To keep things simple and interpretable in our early experiments, we chose the Random Forest classifier—a strong, non-linear ensemble model that is relatively robust to feature scaling and overfitting. Before training, we standardized the feature values using z-score normalization to ensure each feature contributed equally to the model's decisions. After training the Random Forest with 100 estimators, our model achieved a promising log loss of **0.96256**, sounds good for starting point!!

4. Next Steps

After building our initial feature foundation and validating them with a Random Forest classifier, we turned our attention to a more powerful and widely-used model for structured data—**XGBoost**. We began experimenting cautiously, testing a shallow version of the model with just **2 estimators**, a **maximum depth of 2**, and a **learning rate of 1**, which yielded a **log loss of 0.34223**. Recognizing the impact of model complexity, we increased the depth to **6**, which immediately brought an improvement, reducing the log loss to **0.29163**. Continuing our incremental approach, we raised the number of estimators to **3** (while keeping depth at 6), achieving **0.28206**, showing that deeper and slightly more robust trees capture more intricate patterns in the data.

Parallel to model tuning, we revisited our TF-IDF vectorizer settings, particularly the **max_features** parameter. Starting from **500** and testing **750**, we eventually increased it to include the **maximum number of unique terms** in our corpus, thus preserving more textual nuance in the representation. This enriched representation, when paired with **XGBoost (2 estimators, depth 3)**, improved the performance to **0.27052**.

Our most significant leap, however, came from adjusting the **XGBoost objective function** to **reg:logistic**, aligning it better with the binary nature of our link prediction task. This single change dropped the log loss to **0.23315**, marking our best score yet. This underscores how **both model hyperparameters and feature engineering** (such as TF-IDF dimensionality) are critical to optimizing performance. Shallow trees can quickly capture decision boundaries but combining them with deeper representations and the correct learning objective unlocks their full potential.

So until now we have **5 Features** and we have experimented with **2 models**. More specifically we have:

- **Text Similarity(using TFIDF)**
- **Author Overlap**
- **Common Neighbors**
- **Jaccard Coefficient**
- **Preferential Attachment**

And two models:

- **Random Forest classifier**
- **XGBoost classifier**

Models	Log-Loss For Our 5 Initials Features
RF(100)	0.96256
XGB(2,4)	0.34223
XGB(2,6)	0.29163
XGB(3,6)	0.28206
XGB(2,3,max_tfidf)	0.27052
XGB(2,3,max_tfidf,reg)	**0.23315**

5. Adding More Sophisticated Features

After the 5 initial features we thought that it is time to look for more and even better features. One way is the graph embedding algorithm that we used which is Node2Vec

We computed cosine similarity between two node embeddings to obtain the edge feature.

But how does the Node2Vec algorithm actually works.

5.1 Node2Vec

Node2Vec is an algorithmic framework that can learn continuous feature representations (embeddings) of the nodes of any graph. It is heavily inspired by the word2vec skip-gram model in the sense that it uses random walks to generate a corpus of "sentences" (which are actually paths) from a given network. These paths of predefined length are computed a given number of times for each node of the graph. Then, these embeddings are fed into the word2vec model, which outputs continuous representations for each word in the vocabulary (nodes in the graph here). We decided to use the following parameters: a number of walks per node of 10, a walk length of 100, a dimension of 64 and a window size of 10. Higher dimension, longer walks and more walks allow for capturing more complex relationships but increase computational cost. **It is important that we used all our G graph in order to compute the node2vec embeddings. Later on we will discuss why we changed that.**

The average time that it took to compute the node2vec embeddings was 20 or 30 seconds but when we were fitting the model the process was running about 1 or 2 minutes

5.2 Abstracts Embeddings

Another way is the text embedding algorithms like **SciBERT, ALLENAI/SPECTER and DistilBERT**. We computed cosine similarity between two abstract embeddings to get feature values. More specifically

we experimented with **three different BERT models** to compare their performance and suitability for scientific text:

5.2.1 SciBERT: A BERT model trained specifically on scientific publications from Semantic Scholar. Its vocabulary and pre-training corpus are adapted to the structure and terminology of scientific language, making it particularly effective for tasks involving academic papers.

5.2.2 ALLENAI/SPECTER: A BERT-based model fine-tuned on citation-based tasks. Unlike general-purpose transformers, SPECTER is trained to position semantically related scientific documents closer in the embedding space, making it especially suitable for citation prediction and paper similarity applications.

5.2.3 DistilBERT: A distilled, lighter version of BERT that offers a trade-off between speed and performance. **DistilBERT** retains **97%** of **BERT's** language understanding capabilities. Though not specialized for scientific content, DistilBERT provides a faster alternative for generating embeddings when computational efficiency is a priority.

To capture deeper semantic meaning from the text beyond simple frequency-based representations like TF-IDF, we employed **transformer-based contextual embeddings** using **BERT** models. The process involved generating fixed-size vector representations for each paper's abstract using pre-trained BERT variants. We implemented a batching mechanism to efficiently handle large volumes of abstracts by processing them in groups of 6. For each paper in a batch, we first attempted to extract its abstract.

To **handle** the cases where the **abstract was missing** we explored **two ways**. The **first one** was the obvious one, to **skip the papers that had empty abstracts** and only process the papers that had abstracts. But although this solution was **pretty easy** and gave us **some good results**, we **worried** that maybe we will **have inconsistencies with our data**, we wanted to **explore another solution** to see if we could solve this problem and also get even better results. The other solution was to **construct a textual input based on the authors' names**. So even though it's **"just names," BERT** finds **meaningful semantic patterns that correlate with research areas**, **author combinations are actually semantic signals that BERT can learn from, not just random text.**

In rare cases where both the abstract and author list were unavailable, we used a placeholder text ("No information available") to maintain consistency. We then tokenized the input batch using the corresponding BERT tokenizer with truncation and padding applied up to a maximum length of 512 tokens. After feeding the tokenized input into the model, we computed the final embedding by taking the mean of the last hidden layer's output across all tokens. This mean-pooled vector effectively summarizes the document's semantic information in a dense format and is stored for downstream use in our link prediction task.

Below are the time that every model took to compute the embeddings where **B** means before we handle the empty abstracts(skip them) and **A** means after the authors-based solution

Model	Computation Time(minutes)
SciBERT	13.45(B),17.17(A)
ALLENAI/SPECTER	13,56(B),17.26(A)
DistilBERT	7,23(B),9.05(A)

5.3 Evaluation Of Embeddings

Below are the evaluation of our embeddings. We added to our initial 5 features the node embedding similarity and we also added the BERT embeddings(**important: at this point we only handle the empty abstracts by skipping them**) afterwards in order to how the impact the log loss of our prediction. Because we got good results before we use XGB model again but we try to tune the different results in order to see how the model will behave

Features	Model	Log-Loss
Initial 5(max_features=500) + Node2vec	XGB(300,5)	0.2760
Initial 5(max_features=750) + Node2vec	XGB(300,5)	0.26465
5(mx=750) + Node2vec+SciBert	**XGB(500,5)**	**0.21350**
5(mx=750) + Node2vec+SPECTER	XGB(500,5)	0.22459
5(mx=750) + Node2vec+Distil	XGB(500,5)	0.24513

The observed improvements in performance can be explained by the **complementary nature** of the embeddings we incorporated. Initially, our model relied on hand-engineered features and TF-IDF vectors, which capture shallow lexical similarity but lack semantic understanding or structural awareness of the citation network. By introducing **Node2Vec**, we gained a powerful representation of the **graph structure**, capturing relationships between papers

based on their citation patterns. This allowed the model to learn from how papers are interconnected, not just their content — which explains the immediate improvement in log-loss when Node2Vec was added.

Further enhancements came from integrating **contextual embeddings, SciBERT** performed the best because its embeddings are closely aligned with the language of our abstracts.

SPECTER learns to place documents closer in embedding space if they are semantically and citation-wise related, which made it slightly less aligned with our task than **SciBERT**.

DistilBERT lacks scientific domain tuning. This explains its lower performance compared to the other two.

We can say that the synergy between **graph-based structural information (Node2Vec)** and **semantic-rich textual embeddings (BERT variants)** enables the model to understand both **how papers are related** and **what they discuss**, leading to better citation link predictions.

So until now we have 7 **Features** and we have experimented even more with **XBG models**. More specifically we have:

- **Text Similarity(using TFIDF)**
- **Author Overlap**
- **Common Neighbors**
- **Jaccard Coefficient**
- **Preferential Attachment**
- **Node2Vec Embedding**
- **Bert Embedding**

And right now the **XBG model** has 500 estimators, max depth of 5 and the score now is **0.21350**

6. Even more Features

As we explore even more this problem and we try to decrease even more our log-loss we have two things in mind. We can try to add more features with the hope that we will pick the right ones and the ones that have meaningful impact on our "feature map" and the other way is to experiment even more with different models and parameters to see how other architectures of models will behave with our features. So we did both!!

We explored our citation graph even more and added attributes like:

- (cluster) **- Clustering Coefficient** which measures how connected a node's neighbors are to each other (i.e., the likelihood that a paper cites other papers that are also connected). Ranges from 0 to 1 and tell us that high clustering might indicate that a paper is embedded in a tightly-knit scientific community or topic area.
- (rank) **- PageRank** which was originally used by Google to measure the importance of nodes. In a citation graph, it estimates a paper's *influence* based on how many high-quality papers cite it. Influential papers (high PageRank) are more likely to be cited again, making this a valuable predictive feature.
- **HITS Algorithm (Hubs and Authorities)** has two parts: **1)Authorities** (a) are nodes (papers) that are heavily cited by hubs and **2)Hubs** (h) are nodes that cite many authorities. Separates papers that are influential (*authorities*) from those that serve as major *information aggregators* (hubs), both of which can play a role in citation likelihood.
- (bet) **- Betweenness Centrality** measures how often a node lies on the shortest paths between other nodes. We set the limits the computation to paths of maximum length 5, which is helpful for large graphs. Papers with high betweenness may serve as "bridges" between different research areas — they're often interdisciplinary or foundational.

Also we added even more edge features than before. We already had **Jaccard Coefficient** and **Preferential Attachment** and now we add **Academic Adar Index** and **Shortest Path**

Also we tried new architectures like **MLP** with **3 hidden layers of 250,100,50** neurons and activation function "**relu**" , because we wanted to capture even more the non-linearity nature of features and believed that the MLP would be ideal for that. Also we found another similar model that belongs to the wide range of **Gradient Boosting Machines** like **XGBoost** which is **LightGBM.** Furthermore we changed the max features in TF-IDF one more time because we thought that 500,750 and even 1000 and 2000 are way too little compared to the number of words that we have in all of our abstracts. So to capture the relative importance of words in the set of abstracts better we increased it to 20000(!) and use ngram of (1,2). The only problem with these settings is that because we compute norms and convert the sparse vectors into dense ones, these computations are very high RAM consuming.

Now with **12 Features(7 from the past section, 5 from the graph and Adamic Adar and DistilBERT)** and **3 models** from which the **2 are new**, the results are pretty interesting. Below is a table with these results:

Models	Log-Loss
XGB(500,5)	0.21639
MLP(250,100,50)	0.21224
LGB(default)	**0.19123**

But when we used the **SciBERT** with the same number of features like before we got:

Models	Log-Loss
XGB(500,5)	0.18987
LGB(default)	**0.18348**

Also after that we added **the sum and the absolute difference of the attributes of our citation graph** like this: **rank[source] + rank[target],**

abs(rank[source] - rank[target])

and we added **Katz's Centrality-**(katz) sum and absolute difference and from **12 Features** we ended up to **22 Features, crazy(!!)** and we got these results:

Models	Log-Loss
XGB(500,5)	0.18776
LGB(default)	**0.17571**

All of these time when we were testing the new features and we were getting the results we were doing feature importances and **SHAP analysis** for our models and more specific we saw that for our **XGB** model the features that are most important are **text similarity, preferential attachment, node2vec, common neighbors and Bert similarity**. From the graph attributes, the **rank sum and difference** and the **h-index sum** are high in the corresponding list

On the other hand for our **LGB** model the **embedding similarity from node2vec , h-index sum , page rank sum, preferential attachment, a-index difference and cluster sum** are high in the corresponding list.

7. Sentence Transformers

While BERT undeniably shines in word embeddings, Sentence Transformers specialize in comprehending entire sentences or paragraphs. They are designed to create rich sentence embeddings, opening doors to numerous applications that demand sentence-level comprehension. Sentence Transformers can analyze and recognize the mixed sentiment within this sentence, highlighting their depth of comprehension. So we wanted to explore even this aspect of the problem. More specifically we used two models: **All-MiniLM-L6-v2** and **All-mpnet-base-v2**

- **All-mpnet-base-v2** is a sentence and short paragraph encoder. Given an input text, it outputs a vector(768 dimension) which captures the semantic information. The sentence vector may be used for information retrieval, clustering or sentence similarity tasks. all-mpnet-base-v2 is a fine-tuned model that uses the pretrained microsoft/mpnet-base model under the hood. This model has the best quality of the **SBERT** all family of models.
- **All-MiniLM-L6-V2** model is a powerful tool for sentence and short paragraph encoding, capable of mapping sentences and paragraphs to a 384-dimensional dense vector space. But how does it work? Essentially, it takes input text and outputs a vector that captures the semantic information of the input, making it useful for tasks like information retrieval, clustering, and sentence similarity. With its contrastive learning objective and AdamW optimizer, this model achieves state-of-the-art results in these areas. But what really sets it apart is its efficiency - it can handle input text up to 256-word pieces and produces dense vector representations that facilitate efficient and effective text analysis

The **all-mpnet-base-v2** model provides the best quality, while **all-MiniLM-L6-v2** is 5 times faster and still offers good quality

Below are the time that every model took to compute the embeddings:

Model	Computation Time(minutes)
all-mpnet-base-v2	36.40
all-MiniLM-L6-v2	19.05

But how do we computed these embeddings? The process involved splitting each abstract into individual sentences and generating dense vector embeddings for them using the models. For missing abstracts, we had again two options, skip them or somehow construct embeddings for the empty abstracts. For the second solution we wanted to do the same as the previous bert embeddings, to use the author informations but the computation time increased very much so we used a default embedding derived from an empty string to maintain consistency. This default embedding will be close to zero in most dimensions but not exactly zero - it's what the transformer model learned represents "no content."

By calculating pairwise cosine similarities between all sentence embeddings from the two papers, we extracted three key statistical measures: **the**

maximum similarity score (indicating the most similar sentence pair), **the average similarity across all sentence pairs** (providing an overall semantic relatedness measure), and the **count of sentence pairs exceeding a specified similarity threshold** (capturing the volume of highly similar content). They provide both granular (maximum similarity) and aggregate (average similarity, high-similarity count) perspectives on how closely two documents relate at the sentence level, while the threshold-based counting mechanism helps identify papers with substantial overlapping content that may warrant further investigation.

After we added these 3 new features and we removed the Edge-Betweenness Centrality we had **23 Features** and using the **all-mpnet-base-v2** which is more powerful (because the other model gave us not so good results), we got these results:

Models	Log-Loss
XGB(500,5)	0.16511
LGB(default)	**0.15488**

As we see the 3 new embeddings features that we added from our powerful model **gave us the best solution so far**(and the best one in all of our project).We had to wait a lot because these new features added a lot of computation time when we generated our training data but also the testing phase was pretty time consuming

So until now we have 23 **Features** and we have experimented even more with **XBG and LGB models**. More specifically we have:

- **Text Similarity(using TFIDF)**
- **Author Overlap**
- **Common Neighbors**
- **Jaccard Coefficient**
- **Preferential Attachment**
- **Node2Vec Embedding**
- **Bert Embedding**
- **Adamic Adar**
- **Maximum Similarity**
- **Average Similarity**
- **Count Above**
- **Clustering Coefficien(sum and diff)**
- **PageRank(sum and diff)**
- **Betweenness Centrality(sum and diff)**
- **Hits(sum and diff)**
- **Katz's Centrality(sum and diff)**

And right now the **XBG model** has 500 estimators, a max depth of 5 and the score now is **0.0.16511** and the LGB model still has the default settings and the score now is **0.15488(The best one)**

8. Playing With Our Models

Until now we have not explore very much the parameters of our models so after we found our best solution we tried to improve the results even better. We wanted to hyperopt our models, meaning that we had to explore even more the parameters of the models, we thought even if we had to reduce the number of our features and even combine the predictions of our models.

More specifically, for our XGB models that use **logistic regression (binary:logistic)** with log-loss evaluation metric for probability-based predictions. Key hyperparameters include **n_estimators** (number of boosting rounds), **max_depth** (tree complexity control), and **learning_rate** (step size for each iteration). Regularization is handled through **subsample and colsample_bytree** (random sampling of data and features), **gamma** (minimum loss reduction for splits), and **min_child_weight** (minimum sum of instance weights in child nodes).

For our LGB models the **subsample** and **colsample_bytree** parameters provide regularization through random sampling of training instances and features respectively. The **num_leaves** and **min_child_samples** parameters are **LightGBM**-specific alternatives that could be used instead of **max_depth** and **min_child_weight** - **num_leaves** directly controls leaf nodes (more flexible than depth-based control), while **min_child_samples** sets minimum data points required in leaf nodes, offering different regularization strategies optimized for LightGBM's leaf-wise tree growth.

We used the hyperopt library from python to do the GridSearch for the best parameters automatically. The goal is to minimize a function that takes hyper-parameters as input from the search space and returns the loss. Here we use the log loss as loss function. The optimizer will decide which values to check and iterate again. The $fmin$ function is the optimization function that uses the history of evaluated hyper-parameters to create a probabilistic model to suggest the next set of hyper-parameters to evaluate.

Also we thought if we should decrease the number of features, so from the features importance we removed the features that did not have significant impact on our models.

Lastly we combined the predictions of our 2 main models ; we took the average of these two and we test it.

Here are the results:

Models	Log-Loss
Hyper_XGB	0.18771
Hyper_LGB	**0.16523**
XGB_reduction	0.17656
LGB_reduction	0.19289
Combine_Models(from hyper_models)	0.17907
Combine_Models(from reduction models)	0.16975

After that we continued to explore the number of features that our model should have and we also explored the stacked model architecture that stacks LGB and XGB so we got these results:

Models	Log-Loss
LGB_10	0.19034
XGB_13	0.17281
XGB_11	0.17222
Combine_Models_10	0.16920
XGB_10	0.16652
XGB_15	0.16656
Combine_Models_11	0.16531
LGB_11	0.16439
LGB_13	0.16392
Stacked_Model_15	0.16063
LGB_15	0.16063
Combine_Model_15	0.15724

So we can see that we did not improved our score but we are close to our best solution so we have to explore even more what we can improve.

9. Data Leakage

It is time to discuss how we generate the negative examples that we use in order to generate the training data which is a combination of negative(papers that do not link) and positive(papers that do have a link) examples. At first we simply were getting random examples of (source,target) from our G graph and if there is not in the existing positive examples and the negative list we added to the negative examples list. We computed the same amount of negative examples as the number of positive examples and after that we were splatted the examples into Train and Validation sets of 80/20. But there are problems in all that process. First, we did not check if the negative examples are inside the test pairs. We found out that the some negative examples where inside the test pairs. So we removed them from the negative examples. Second, we did not split correct the Train and Validation Sets so we created from the beginning from the already known edges train edges that we will use them only for training and edges of validation that we will use them to validate the models. With this way the process and the sets are very clear and distinct. And also we made a test set in order to test our predictions in real time without having to make a submission in Kaggle. So now the three very important sets are distinct and the model can not “see” them and “cheat” at the end. The third issue was that we were using the entire graph G (which included edges that should be "future" information) to compute features like node2vec and other features. This meant that our model had access to information it shouldn't have during training. This is why for a long time at the final stages of training we were seeing cross-validation scores of our models being very low but when we were making the predictions the log loss at the end had significant distance between them. So after that observation we made a G_train graph that contained only train edges that we had previously splatted. So here are the updated data:

Original graph - Nodes: 138499, Edges: 1091955

Train edges: 873563 (80.0%)

Val edges: 109196 (10.0%)

Test edges: 109196 (10.0%)

Training graph - Nodes: 138499, Edges: 873563

Data Leakage is a very common problem in machine learning and ai and we are glad that we found it. But even though we fixed the problem we were hoping that this would give us better results but now the results are more real and gave us other directions of what we could possibly change for better results. Overall, now we have a better approach of our data and more official splitting phase.

10. The 0.15488 Paradox

Interestingly, our best submission in terms of log-loss was achieved during an earlier stage of our pipeline—*before* we addressed some critical preprocessing steps that we initially believed would improve performance. Specifically, this submission was made prior to fixing a subtle **data leakage issue**, before we applied **whitening** on the BERT embeddings (a technique we considered to reduce the impact of their high dimensionality), and also before we **handled missing abstracts** by falling back to author information (which we had previously skipped entirely). We had strong expectations that these refinements would lead to a lower log-loss, as they addressed inconsistencies and aimed to improve the generalization of the model. However, despite these theoretically sound enhancements, we did **not observe a further reduction in log-loss**. On the upside, we did notice that the **gap between our cross-validation, validation, and test log-loss became smaller**, indicating more **stable and reliable model behavior**. This suggests that while the overall accuracy didn’t improve, the changes contributed to better generalization and robustness,

and that the remaining limitations might now lie elsewhere—potentially in the structure of the features, the model capacity, or even the underlying representation of the graph itself.

11. Models

During this project we explored many different architectures trying to find the best one that will give us great results. More specifically we explored these models:

Logistic Regression

As a **baseline**, we employed **Logistic Regression**, which, while not used for actual submissions, provided a useful benchmark. It demonstrated surprisingly solid performance, with log-loss scores ranging from **0.19 to 0.3**, depending on the feature set, making it a reliable reference point for comparison.

Random Forest

The Random Forest algorithm is based on the creation of more than one decision tree and the interaction of these trees with each other. Each tree is trained by randomly selecting features and using a random subsampling (bootstrap) part of the dataset. In this way, each tree creates a prediction model on its own.

Random forest is formed by the combination of Bagging (Breiman, 1996) and Random Subspace (Ho, 1998) methods. Observations for trees are selected by bootstrap random sample selection method and variables are selected by Random Subspace method

Following this, we utilized a **Random Forest** classifier, which marked our first successful submission and served as an accessible yet powerful model capable of handling feature importance and non-linearity well. However, as our feature space grew more complex, we shifted toward more advanced models

XGBoost

XGBoost (eXtreme Gradient Boosting) is an expansion or enhancement of the Gradient Boosting Machine (GBM) algorithm. Unlike the GBM algorithm, XGBoost has a faster solution that can perform parallel computation on large data sets. In this way, it requires less computation time compared to other algorithms while making high-precision predictions. However, the XGBoost algorithm, unlike GBM, structures tree configuration and weighting differently to allow trees to grow better and more evenly

The **XGBoost (XGB)** model became a central part of our pipeline due to its strong performance and ability to handle intricate feature interactions, though it proved **computationally heavy** when the number of features increased significantly.

LightGBM

LightGBM is a derivative of the Gradient Boosting Machine (GBM) algorithm developed by Microsoft. LightGBM is designed to build high-performance machine learning models on large datasets. Unlike the GBM algorithm, LightGBM uses a histogram-based learning technique by dividing the data vertically and horizontally. This allows data to be processed faster and uses less memory. Also, LightGBM aims to get results faster than GBM by using low depth trees specifically to certain data features.

It is faster with the Leaf-wise growth strategy instead of the level-wise growth strategy.

LightGBM reduces computational cost. The duration of the decision tree trainings is obtained accurately by the calculation and the number of parts. Thanks to this method, both the training time is shortened and the resource usage is reduced.

Thanks to this feature, LightGBM is separated from other boosting processes. The model has fewer errors and learns faster with the leaf-oriented strategy.

However, the leaf-driven growth strategy allows the model, which is thought to have underestimation of data, to be prone to oversight

So **LightGBM (LGB)**, which turned out to be our **best-performing model**. It offered **faster training times**, handled large feature sets efficiently, and consistently produced **superior log-loss results**.

MLP(2 hidden layers and neurons from 5 to 10)

A Multilayer Perceptron (MLP) is a type of neural network that uses layers of connected nodes to learn patterns. It gets its name from having multiple layers — typically an input layer, one or more middle (hidden) layers, and an output layer.

Each node connects to all nodes in the next layer. When the network learns, it adjusts the strength of these connections based on training examples. For instance, if certain connections lead to correct predictions, they become stronger. If they lead to mistakes, they become weaker.

This way of learning through examples helps the network recognize patterns and relationships about nonlinear data are significant in machine learning and they can make pretty good predictions.

The MLP was particularly notable for its ability to capture **non-linear relationships** among features, suggesting that our engineered features interacted in more complex ways than tree-based models might fully exploit.

Combine Models

We also tried to take the average of our XGB and LGB models predictions and combine them.

Stack Models

The idea behind stacking is that the meta-model can learn to combine the strengths of the different base models. If one model performs well on certain types of instances and another on different types, the meta-model can potentially learn to weigh their predictions, accordingly, leading to better overall performance than any single base model. Here we combine XGBoost classifier, LightGBM classifier, Logistic Regression classifier

12. Future Vision

In such a parameterized problem we saw that there is a vast number of things that we could change in order to have better results. We have concentrated 5 things that we could change and hope that will help us to achieve even better results:

1. Negative examples generation

Our current approach is very basic. We find random pairs and we add them to our list. We could utilize the node's degrees or see the node's neighbors. With that we could make even more robust negative examples that will help us make the training better

2. Reduce Dimension(BERT/SBERT)

The 768 dimension that the vectors have from the BERT models is quite large which not only leads to increased **storage cost** but also the **computation** or **retrieval speed.** Also adds a lot of noise and is not good for the model. We could use PCA or whiten the data which is to remove the underlying correlation in the data and normalize the scale.

3. FineTune The BERT/SBERT Models

We could finetune the BERT and SBERT models and train them even better on our data

4. Explore Even Better Features

We could find or even combine features with the hope that this will help us with better results.

5. Optimize Even More Model Parameters

We could explore even more the parameters of every model that we used and through the GRIDSearch that we did with the hyperopt library, hope to find better parameters that will lead us to better results

13. Easter Egg (Code Carbon)

Because energy matters and we live in difficult times duo to climate change, we want our programm to be as environmentally friendly as possible. So we found a python-library called **Code Carbon** which estimates the amount of carbon dioxide (CO2) produced by the cloud or personal computing resources used to execute the code. It then shows developers how they can lessen emissions by optimizing their code or by hosting their cloud infrastructure in geographical regions that use renewable energy sources.

So for the main programm which includes all the computations from start to finish we have:

Energy consumed for RAM : 0.002974 kWh. RAM Power : 19.86054039001465 W

Energy consumed for all CPUs : 0.006500 kWh. Total CPU Power : 42.5 W

Energy consumed for all GPUs : 0.002578 kWh. Total GPU Power : 16.94975023969902 W

0.012052 kWh of electricity used since the beginning.

14. Project Takeaways

This project provided us with a rich learning experience, combining practical machine learning implementation with a deeper understanding of real-world challenges in scientific link prediction. Through every stage—from preprocessing to feature engineering and model evaluation—we gained several important insights:

1. Strong Foundations Matter

Starting with a well-structured pipeline—**data collection, preprocessing, feature extraction, model training, validation, and evaluation**—helped us stay organized and iterate efficiently. The early use of simple models and interpretable features like TF-IDF similarity and author overlap allowed us to understand the data and validate our pipeline quickly.

2. Feature Quality Over Quantity

We learned that **well-thought-out features** (e.g., text similarity, citation graph metrics, Node2Vec, BERT-based embeddings) often mattered more than blindly increasing feature size. Interestingly, some

performance improvements came not from adding new models, but from combining features in meaningful ways, such as integrating Node2Vec and BERT embeddings with our initial five handcrafted features.

3. Data Leakage Is a Real Threat

One of the most important lessons was the **critical impact of data leakage**. Our best early submission actually occurred **before we fixed data leakage issues**, highlighting how misleading results can be if the evaluation isn't carefully designed. After resolving these issues, we saw more trustworthy but slightly lower scores, reinforcing the importance of robust validation strategies.

4. Embeddings Are Powerful—But Need Careful Handling

The use of **SciBERT, SPECTER, and DistilBERT** helped us capture semantic similarities across papers. However, we also realized the importance of **preprocessing BERT embeddings**, including **handling empty abstracts**, **whitening to reduce dimensionality**, and avoiding overfitting. Not all embeddings performed equally—**SciBERT aligned best with our task**, possibly due to its domain-specific training, while SPECTER emphasized citation prediction and thus performed slightly worse.

5. Model Choice Depends on Context

Different models offered different benefits. **Logistic Regression** gave us a fast and understandable baseline. **Random Forests** performed well early on, but6 couldn't scale well with complex features. **XGBoost** was powerful but computationally heavy. **LightGBM** emerged as the most effective model overall, offering both **speed and accuracy**. The **MLP** also showed promising results, especially with complex, nonlinear feature interactions.

6. Not All Improvements Lead to Better Results

Interestingly, we expected some changes to improve performance—like **whitening BERT embeddings**, **increasing TF-IDF max features**, or **replacing missing abstracts with author names**—did not reduce log-loss as expected. This reminded us that **every change must be tested**, and **intuition must be backed by validation metrics**.

7. Evaluation Gaps Are Key Signals

Initially, we had a noticeable gap between **cross-validation loss, validation loss, and test loss**. After addressing leakage and stabilizing the preprocessing, these gaps shrank, suggesting that **our models were generalizing better**. This shift helped us trust our results more and diagnose where future improvements could be targeted.

Ultimately, this project wasn't just about maximizing performance on a leaderboard—it was a valuable exercise in **end-to-end machine learning**, **critical thinking**, and **problem-solving under real-world constraints**. We leave this project not only with a competitive model but also with a much deeper understanding of the importance of **data integrity, model interpretability, and experimental rigor**.