Dynamic Graphs: Predicting Future Bigrams in Patents with NLP and LP

Atul Kumar

Supervised by Dr. Zamilur Rahman

Algoma University

Sault Ste. Marie, ON

Contents

1 Introduction

Today, technology advances at a very rapid pace, while researchers are pushing the boundaries of knowledge and exploring new areas where they can research. It is important to understand the emerging trends and patterns to predict the potential technological advancements and research areas. This is why, an analysis of the previous innovations is required for predictions. For their analysis, Link Prediction (LP) and Natural Language Processing (NLP) have emerged as important tools to learn the hidden trends and patterns.

Link Prediction (LP) is the art of predicting the links between the nodes. For example, in social networks, at time t, if user A is connected to user B and user B is connected to user C, how likely will there be an edge between user A and user C at time t+1? It calculates how similar the two nodes are, and based on the score, it predicts whether there will be an edge or not. It does so by identifying trends/patterns in the existing network. In this context, we will decide from the network topography, whether the pair of words would be a future bigram or not.

Natural Language Processing (NLP) allows the processing (decoding) of data which contain natural languages like English, French, etc. The machine doesn't know what the meaning of each word is when it comes to processing the textual data written in natural languages. However NLP helps a lot there, it computes the semantic relationship between the words and performs the appropriate task based on that. Several tasks are used to perform under NLP to transform that data to the appropriate form and make sense to the machine as well, which we will be discussing in the upcoming sections.

In this paper, we aim to predict future bigrams using Dynamic Graphs, LP, and NLP concepts. We'll begin by discussing the background information about the problem statement. Then we will provide a detailed description of the datasets, followed by tasks performed under NLP. Once performed, we will focus on the graph construction, which will be used to perform the link prediction methods like calculating the similarity coefficients and how they will be fed to the ML models. Finally, we will analyze how effective each implemented model was to predict future bigrams by calculating several performance measures.

2 Background

As we know from our COSC4086 project, social networks can be mapped as graphs. Considering them as graphs helps us visualize the evolving patterns in each period and apply the graph theory concepts to predict how similar the two nodes are. Let's take a look at the basics of graphs and how they work.

A graph G consists of two sets: a set of vertices, or simply nodes, and a set E of edges that connect the vertices. Here's a simple graph:

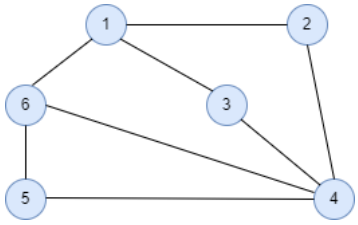


Figure 1: A graph with 6 vertices and 8 edges

As we said, a graph consists of two sets: a set of vertices and a set of edges, which is represented as:

$$V = \{1, 2, 3, 4, 5, 6\}$$
$$E = \{(1,2), (1,3), (1,6), (2,4), (3,4), (4,5), (4,6), (5,6)\}$$

Since this is an undirected graph, the edge between any two vertices goes in both directions. This means there is an edge between (1,2) and (2,1), and it can be represented as a single edge either by (1,2) or (2,1). Graphs may or may not contain the edges that have the directions or weights.

1. **Direction:** Edges can be bi-directional or uni-directional. When they are bi-directional, we don't have to show their directions with the arrows, since we can go in either direction. However, in the case of uni-directional, we can't go in both directions, but just the direction specified.

2. **Weight:** Edges can be weighted or unweighted. Weight often represents distance, cost, etc. between two nodes. If the graph has unweighted edges, we assume they all have unit

weights. For example, if we map the country map as a graph, the weight between two vertices (provinces) can be the distance between those two vertices (provinces).

2.1 Types of Graphs

There are several types of graphs, and each of them has its unique characteristics:

1. **Undirected Unweighted Graph:** The graph that we have just seen above is an undirected unweighted graph. Since that graph doesn't have any edges that have directions and weights associated with it.

2. **Undirected Weighted Graph:** This type of graph has no directions associated with it but has the edges weighted (might be representing distance, cost, etc.).

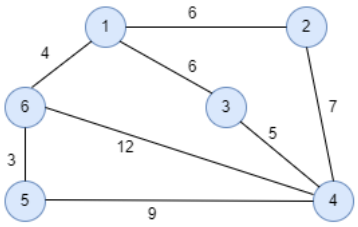


Figure 2: Undirected Weighted Graph

3. **Directed Unweighted Graph:** This graph has directions associated with the edges but not the weights. So, the weights for all the edges are considered to have a unit weight. Since there is an edge starts Vertex 2 and points to 1, this means, there is an edge between (2,1), but not between (1,2).

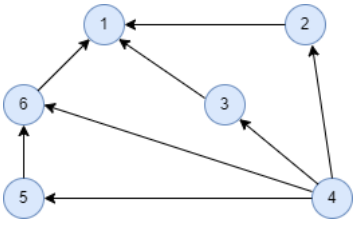


Figure 3: Directed Unweighted Graph

4. **Directed Weighted Graph:** This type of graph has both the directions and weights associated with it.

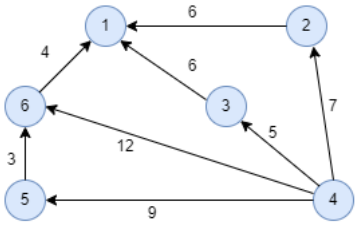


Figure 4: Directed Weighted Graph

2.2 Static and Dynamic Graphs:

1. **Static Graphs:** As the name says, the nodes and edges, in the static graph, don't change over time i.e. remains static. The network topology remains the same and algorithms that learn the trends or patterns using static graphs may not be useful when there's a new trend emerges. Also, it's not very useful, especially for networks that have nodes or edges constantly evolving with time. For example, in the case of social networks, every day new users (nodes) join the network and new connections(edges) are also made by new/existing users.

2. **Dynamic Graphs:** Opposite from static, they evolve. This change can consist of information about the addition or removal of the nodes, edges, or both. Basically, it contains a series of snapshots taken at different points in time, like we have for years 2017, 2018, and 2019. Each snapshot represents a static graph, and the next snapshot shows how it evolves over time. The algorithm learns all the present and previous trends and makes predictions about future connections. Just a note, we are considering a dynamic graph as the title says, however, nodes remain the same for all three time periods, but edges would change.

3 Datasets

Our dataset comprises a collection of patents, which were carefully picked one by one to ensure the content was related to what we were searching for. Here are some motivations behind choosing patents as the data source:

- Patents contain rich and diverse content about technological changes and advancements. They contain a lot of information almost across all domains, making them an ideal source for our approach.
- Patents contain domain-specific vocabulary, which helps us consider only the data which is useful for our approach, reducing the chances of having irrelevant data in our datasets.
- Patents are well structured and divided into sections like abstract, background, claims, etc. This well-structured organization of patents helps us to extract only the useful part of the patent.
- Due to having the domain-specific vocabulary in the data, predicted bigrams would be more related to that particular field and might be useful for predicting the potential technological advancements and research areas.

We have chosen three different years of patents as the source data i.e. 2017, 2018, and 2019. For each year, we have extracted 600 patents, making a total of 1800, which is kind of a fair amount to deal with. Since the graph is dynamic, the graph constructed from each year's patents would be treated as one snapshot and a total of three snapshots for three different years.

All the patents, that were used, have been extracted from the official website of the United States Patent and Trademark Office (USPTO) [1], which is home to a large number of patents across many domains. Patents are publicly accessible, which means we can access any of them available on the website or download the .pdf copy of that. However, we need to extract the ones related to our chosen domain to make our model perform effectively. That's why, we have used their Advanced Search tool to extract the domain-specific using the following query:

(MEDICAL ADJ APPARATUS OR Pharmaceuticals OR MEDICAL WITH DEVICES OR Healthcare AND Informatics OR Medical ADJ Imaging) AND @ay>="2017" AND @py<="2019"

However, it's worth noting that, we just extracted only three sections of each patent for all three years. Those are:

1. Title
2. Abstract
3. Claims

Each patent contains too much information about the innovations/research/developments, which may cause the information to overflow or make the model unnecessarily complex. That's why, we tried to extract only the relevant data from where we can get the most out of it. Due to this reason, we selected only the above given three sections.

There were also several steps taken to recognize the whole text from the patent, which wasn't possible when we were trying to import the .pdf directly, downloaded from the USPTO. Following are the steps performed to get the text into the Google Colab environment:

1. Uploaded all the downloaded patents to the google drive.
2. Since Google Docs has build-in OCR which recognizes the text, even if the document contains some scanned text. So, we opened all the 1800 patents through Google Docs so that they can be downloaded as Microsoft Word documents.
3. Now, the next challenge was to read the Word documents, which were again converted to Google Docs when uploaded to Google Drive, reading them would make the overall structure even more complex. So, we uploaded all the files to the runtime storage of the Drive, which kept the same format of the documents and we read them using "docx" and "files" python libraries.
4. Once they were in the Colab environment, we extracted only those sections which we mentioned above and output the text to text files(.txt), which eventually we read for our main process.

4 Natural Language Processing (NLP)

As we discussed in the introduction about NLP, it allows the processing of datasets that contain information written in natural languages like English. It involves several tasks to get the data to the appropriate form to process. Here is the list of actions we have performed, on each single patent, to get the text ready to work further:

1. **Numeric and Punctuation:** This process involved the removal of words that contain numeric characters and also those words which have numeric characters to ensure that the model focuses on just the textual content, without being influenced by the numeric information. Additionally, all types of punctuation marks such as commas, colons, semicolons, etc. were also eliminated from the text, to have more clean data for further processes.

2. **Special Characters:** Following the last step, we also removed the special characters like asterisks, ampersands, etc. Again, this step is performed to make sure that our Machine Learning models don't predict any of them to be a future bigram.

3. **Tokenization:** This step involves breaking down the sentences into individual words. Once they are broken down into words, it will help us perform the next three steps.

4. **Stopwords:** To reduce the further complexity and enhance the relevant information of the text to our chosen domain, we also eliminated redundant words like "is", "were", "and", etc. using the NLTK library. This allows the model to focus only the meaningful content, improving the overall results.

5. **Lemmatization:** This step is performed to reduce the redundancy in the datasets like the last step. This involves removing the words that have different forms but the same root meaning. For example, "Chances" and "Chance" represent the same thing if we look at the root meaning of them. So, we eliminate all the words in higher forms while keeping the root word.

6. **Repetitive Words:** Finally, we finished our preprocessing steps with the removal of repetitive words. While making the datasets, we saw several words repeated a lot of

times and kept a note of them such as "claims", "according to", etc. Once all steps were performed, we removed those words as well to make it more cleaner.

Once we are done with these NLP tasks, we combine the tokenized content as well as it was before but rather cleaned.

5 Graph Construction

Before looking into the steps of graph construction, it is quite important to discuss a few terms:

1. **Unigrams:** This just represents individual words or tokens, which is a basic building block of language analysis.

2. **Bigrams:** As the name says, this means the pair of words. It involves pairing a word with another word and performing operations like how similar they both are in a particular context (which we'll see shortly).

3. **Term Frequency (TF):** It calculates the frequency (count) of a particular word(a) in a document(d). If a word appears often, the TF score will be higher too.

4. **Inverse Document Frequency (IDF) [2]:** This parameter measures the uniqueness of the term across the collection of documents(D). If a word has a high IDF, it means the term is less common and more distinctive. It can calculated as follows:

$$IDF(a, D) = \log\left(\frac{\text{number of patents in document (D)}}{\text{number of patents containing this term (a)}}\right)$$

5. **Term Frequency-Inverse Document Frequency (TF-IDF) [2]:** We combine the TF and IDF parameters to calculate how important the particular word is within the context of a given document. TF-IDF allows to extraction of those words which are not only frequently used but also distinctive. The higher the TF-IDF, the more important the word will be. It can be calculated as follows:

$$TF\text{-}IDF(a, d, D) = \text{TF(a,d)} \cdot \text{IDF(a,D)}$$

Now let's take a look at how will the graph be generated using this textual data and generate/predict the links between them. In section 2, we discussed the graphs, where we said that they contain the nodes (or vertices) representing entities and edges representing the relationship between them. If there is an edge between two nodes, it means they are connected.

So, for making the graph, we need both the nodes and edges. For extracting the nodes out of the textual data, we have extracted the top 600 tokenized words(unigrams) according to their tf-idf values from the first year's patents. Those extracted words will have the highest tf-idf values, which means they have the highest importance across that year's patents. Those words would be the nodes of all three year's graphs. However, it is to be noted that there is no certain reason for choosing only 600 words as the nodes of the graphs, this could be either increased or decreased. This amount of nodes was looking reasonable and keeping the overall complexity of the approach fair enough.

Now, let's take a look at how the edge between the extracted nodes will be generated. the word will be considered connected if they aren't separated by more than two unigrams. This approach will make sure that it captures only meaningful co-occurrences instead of pairing any unrelated words while increasing the complexity. For example, consider the following list of items as nodes of the graph:

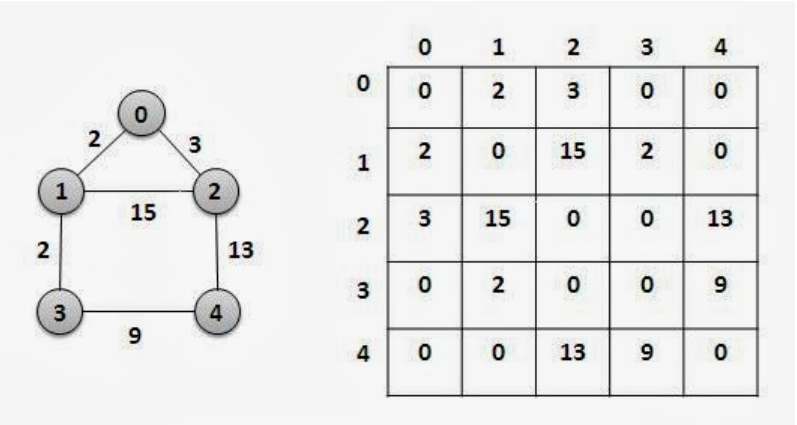
$$(a, b, c, d, e)$$

There can be an edge between the elements if they are not separated by more than two unigrams. This means, there can be an edge between the "a" and "b", "a" and "c", or "a" and "d", but not between "a" and "e" since then they will be having 3 unigrams in between them. Similarly, for the rest of the elements, where "b", "c", and "d" will have a connection with the rest of the nodes, since they are separated by only 2 unigrams. However, in the case of "e", it again won't be connected to "a", because of the distance between them. Lastly, there will be no loops in the graph, which means the same node can't be connected to itself through a edge, as words not be repeated in the patents.

After coming this far, we must have gotten the idea of how the graph is going to be constructed and have noticed that this is going to be an undirected graph (edges will be in both directions). As discussed in the section 2, dynamic graphs evolve over time. This change may alter the

nodes, edges or even both. However, in our approach, we kept the same nodes of the graph for all three years, but the edge status will be checked separately using the corresponding year's patents. Following this, the graphs will have the weights assigned to them as well, which will be determined through the co-occurrence of the words that are connected through this edge. So, the number of times the same pair of words appear will be frequency and hence the edge weight.

Once, we are done counting them, we have the adjacency matrix ready. Using that, we will construct the graph with the help of the NetworkX Python library, where the top row or the very left column represents the nodes of the graph and if the number is greater than zero there will be an edge between those two nodes. The number 0 means there is not going to be any edge between those two nodes as shown in Figure 5



	0	1	2	3	4
0	0	2	3	0	0
1	2	0	15	2	0
2	3	15	0	0	13
3	0	2	0	0	9
4	0	0	13	9	0

Figure 5: Adjacency Matrix Representation of Weighted Graph [3]

6 Link Prediction Methods

According to N.N. Daud et. Al [4], we can classify the method of Link Prediction into four broad categories:

- Similarity: This is one of the most common approaches in link prediction where we calculate how similar the two nodes of the graph are whether they are connected or not. That similarity depends on several factors like as what type of nodes they are connected to, the number of connections, etc. The higher the similarity scores, the higher the chances of

being connected in the future.

- Probabilistic: It calculates the probability value of each pair of unconnected nodes. The higher the probability, the higher the chances of getting connected in the future.
- Algorithmic: This is also a popular and common technique to predict the links between the nodes. In this approach, algorithms are used to analyze the structure and identify hidden trends and patterns to predict future links. Here is where the Machine Learning models come into play.
- Hybrid: We can even create a more promising method to predict the links by combining any of the above-mentioned methods.

  For this project, we have combined the Similarity and Algorithmic approach, making our approach a hybrid approach to make our model more robust. Let's look at what similarity coefficients and machine learning models we have used to predict future bigrams.

6.1 Similarity

- **Common Neighbours:** It calculates the number of common neighbours between two nodes.

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

- **Adamic Adar [5] :** A High score will be assigned to the node whose common neighbours will have fewer degrees, as it will be shared by fewer nodes in the network and will be more specific in that context.

$$AA(u, v) = \Sigma_{z \in \Gamma(u) \cap \Gamma(v)} \frac{1}{\log |\Gamma(z)|}$$

- **Jaccard:** It calculates the number of common neighbours between two nodes with respect to the neighbours they both have in total.

$$JC(u, v) = \frac{\mid \Gamma(u) \cap \Gamma(v) \mid}{\mid \Gamma(u) \cup \Gamma(v) \mid}$$

- **Preferential Attachment [6] :**The node will have a high chance to connect to that node which will have a high degree compared to nodes having a lower degree.

$$PA(u,v) = \mid \Gamma(u) \cdot \Gamma(v) \mid$$

- **SimRank [7]:** This metric considers that two nodes are similar depending on the nodes they are connected to. If they are connected to similar nodes in their neighbourhood, then these two are also considered to be similar. If there are no neighbours connected to those two nodes, this, obviously, means there will be zero SimRank similarity between them.

$$SR(u,v) = \frac{C}{|\Gamma(u)|\,|\Gamma(v)|} \sum_{i=1}^{|\Gamma(u)|} \sum_{j=1}^{|\Gamma(v)|} SR(\Gamma_i(u), \Gamma_j(v))$$

  where,

  C is a constant between 0 and 1

- **RootedPageRank [8]:** Before discussing this, let's have a brief look at what PageRank is, as RootedPageRank is almost similar with a small modification. PageRank involves ranking the nodes according to their importance in the network. It starts at an arbitrary node and keeps following the same edge until "n" iterations are over. Once started following the link, there can be several edges open to follow (as the node may connected to several edges), so it considers the most important one i.e. nodes with higher degrees tend to have high importance, and rank the nodes accordingly depending on how often the node is get visited. RootedPageRank is almost the same with just the exception that PageRank starts at any arbitrary node and iterates for n iterations, as said. However, RootedPageRank always starts from some starting point or we can say root, and does the same process as PageRank, after then.

- **Katz [6] :** This metric calculates the number of all the paths from node u to v and gives more weight to the path that has a shorter length relative to other paths from u to v.

$$Katz(u,v) = \sum_{l=1}^{\infty} \beta^l \cdot |path^l_{u,v}| = \beta A + \beta^2 A^2 + \beta^3 A^3 + \dots$$

where l is the path length

β is a damping factor

$|path^{l}_{u,v}|$ is the set of all the paths from u to v

- **Resource Allocation [9]:** Similarly like common neighbours, it looks for the nodes that are common between both the two unconnected nodes. Additionally, it also computes how close those two nodes are, depending on their common neighbour(s).

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

So far, what we have seen in terms of similarity coefficients depends on the graph and we computed all those based alone on the graph. However, it is also possible to have a link between those words which are very similar. For that, we have used the popular NLP library which is used to represent the texts as a vector, which we'll see next..

- **Word2vec Embedding [10] :** While working with text datasets, it's quite necessary to convert the data into form, so that the machine can understand, since it doesn't know the semantic meaning behind each word and how different is it from another word. Here's where word embedding techniques come into play. In this method, each token (word) is transformed into vectors, which enables to computation of the semantic relationships between words within the context of the given data.

  Word2vec is a popular word embedding technique which represents each token as a vector and computes the cosine similarity between them. When a pair of words(nodes) is given, this looks for how similar one vector is to another. More the similar, the closer the result is to the 1 or angle close to zero.

  Cosine similarity can be represented as:

$$CS(u,v) = \frac{u \cdot v}{\|u\| \cdot \|v\|}$$

Now, let's what how we have calculated them and prepared the data to feed to the ML model:

1. **For years 2017 and 2018 patents:**

   - Firstly, we paired up every node with the rest of the nodes.
   - Calculated all the above-mentioned similarity coefficients on each possible pair of nodes and calculated the similarity scores between them.
   - Then, we also added the cosine similarity score that we got from the Word2vec embedding, so the model must consider that as well and predict.
   - Also, calculated the TF and TF-IDF scores and included them with other similarity scores.
   - As we said that graph is dynamic, for that we have added one more column to get the data ready for the ML models. That column contains the edge status of whether the current edge status persists or not for the next year's graph. If there's an edge appears or disappears in or from the next year's graph, it will be considered that too for predicting last year's patents.
   - Lastly, the above steps will be performed between all pairs of nodes, doesn't matter there is an edge exists or not.

2. **For year 2019 patents:**

   - Again, we paired up every node with the rest of the nodes.
   - Calculated all the above-mentioned similarity coefficients on each possible pair of nodes and calculated the similarity scores between them.
   - Similarly, we also added the cosine similarity score that we got from the Word2vec embedding, so the model must consider that as well and predict.
   - Calculated the TF and TF-IDF scores and included them with other similarity scores.
   - Again, we have performed the above steps for all the pairs of nodes.
   - As we noticed, the above steps are common to what we have done for the first two years of patents. But, here is an exception, there will be no column added containing the information about whether the edge persists or not for the next year. This is what we will be predicting.

- We will train the ML models on these prepared datasets to predict the edge status of next year's patents.

6.2 Algorithmic

As for using a hybrid approach, the second method that we have used is Algorithmic. It involves using the algorithm and identify hidden patterns and trends. To identify them, we also need a reasonable amount of data to predict the future edges between the nodes. Here's a general representation of using algorithmic method:

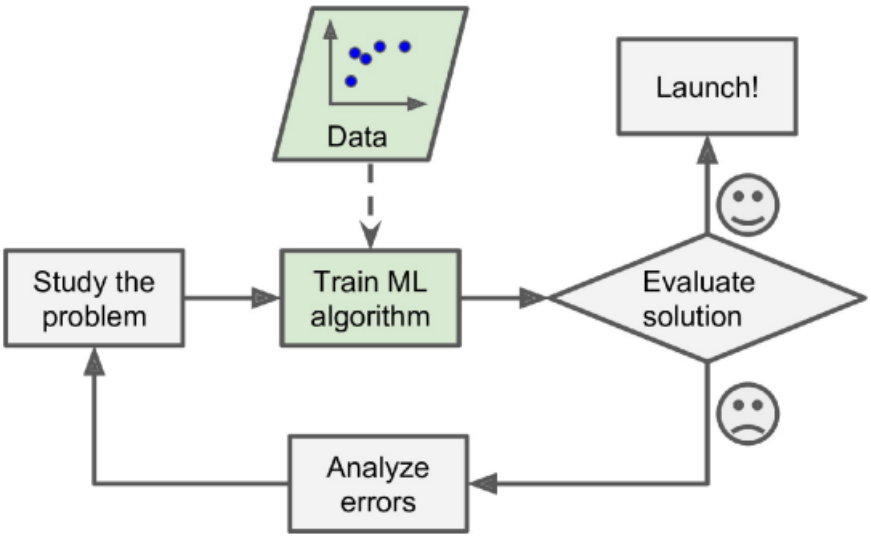


Figure 6: The Machine Learning Approach [11]

As we can see from the above figure, we must start by studying the problem where we look for the needs and requirements. Once we analyzed, we provided the pre-processed data, as we prepared so far, to the model. Then the ML models (or algorithms) look for the trends and patterns and predict the solution. If we are not satisfied with the solution, we can tweak the parameters, use the other models, etc. until we get the desired result.

For this project, we have used several ML models involving deep neural networks. All the models used the supervised machine learning techniques, in which the data will be labelled(column containing the edge status) on which it is trained and then have to predict the data which would have no labels.

6.2.1 Test and Train Data

As said earlier, we must need a fair amount of data to train our ML models. We have the years 2017 and 2018 data as the training data(or labelled) and the year 2019 as test data(or unlabelled), using last year's data we have to predict the potential bigrams for the year 2020. However, we can't provide the whole 2017 and 2018 data as training data, because we won't be able to tell how accurate each model is performing in terms of accuracy scores. To know first how accurate each model performs, we have to test the approach on the labelled data, once the model predicts the labels of current data (for which we have labels), we can compare the predicted labels with the actual labels and tweak the model as required if we are not satisfied with the accuracy score. Once satisfied, we can predict the bigrams for the year 2020.

Keeping the above point in mind, we have divided the labelled data in the ratio of 80:20, which means 2017 and 2018 data will be divided in the ratio where 80% of the data will be used to train our ML models and 20% will be used to test them. The columns containing the next's years edge status will be the labels for the models. To improve the result further, we have set the cross-validation to three. This means, the model will be trained and tested three times on the training data, where it will divide the training data into three parts, treating two parts as training data and the third one (unseen data by the model) as test data three times. Each time both training and test data will be different. This is performed for all the conventional models and runs two epochs (times) for the deep neural network models.

Once trained on that 80% data, we will test all the models on the remaining 20% data. The best-performing model, in terms of test data accuracy, will be considered to test the year 2019 network and to predict the future bigrams.

6.2.2 ML Models

Here are the list of models that we have used to test the accuracy of the approach taken:

1. **Logistic Regression [11]:** Logistic Regression computes a probability score where it calculates how probably an instance belongs to a particular class. If the score is less than 50%, it classifies an instance belonging to a negative class, if greater than 50% it belongs to a positive class.

2. **Decision Tree:** This algorithm is used for large datasets. It involves splitting the data into subsets based on input features (or columns). Once the subsets are generated, it again picks up the best input feature and splits from there, assigning this as either a left/right child of the root node and repeating the same process from that child node until all the input features are exhausted. So, as expected, it uses the tree data structure to make predictions. Once this algorithm builds a tree, it traverses the branches to make a prediction and decides which child node to choose next.

3. **XGBoost:** Similar to a Decision Tree, but on each iteration, this algorithm builds a new tree with the expectation of correcting the error made by the previous decision tree. It keeps repeating this process until it runs for all n-iterations, ending up providing better results than the Decision Trees. /item **Random Forest:** This involves generating multiple independent decision trees, where each tree follows the same process as the Decision Tree algorithm and makes predictions independently. Each tree's root node is selected randomly. Once all the predictions are made, this algorithm aggregates the result and makes a final prediction.

4. **Extra Trees:** Similar to random forest, it generated multiple decision trees. However, instead of just selecting the root node randomly, it selects even all the child nodes, from where it splits, randomly.

5. **Stochastic Gradient Descent [11]:** At each step, this algorithm picks a random instance in the training set and computes the gradient using only that single instance. This keeps computing until the cost function reaches to minimum.

6. **Voting:** This model just combines the above-mentioned models and then they all act like a one, making that one model more robust.

7. **Multilayer Perceptron (MLP) [11]:** It is a type of Artificial Neural Network, which computes a weighted sum of inputs, and each input is associated with the weight. Once computed the sum, it applies the step function to the computed sum and predicts the results.

$$z(weightedsum) = w_1x_1 + w_2x_2 + \ldots + w_nx_n = x^Tw \tag{1}$$

$$h_w(X) = step(z) \tag{2}$$

8. **Long Short Term Memory (LSTM):** This is a type of Recurrent Neural Network which is designed to store information for a longer period. It starts from a very start feature and computes the prediction score while using the information extracted previously. Once the result is made, it is also capable of back-propagating if it wants to update the weight of inputs to get better results.

9. **Bidirectional Long Short Term Memory (Bi-LSTM):** This algorithm is kind of an extension to LSTM, with the exception that it is capable of propagating in both backward and forward directions to adjust the assigned weights, which overall makes it result better than LSTM.

6.2.3 Extracting Future Bigrams

After passing the data through all the models, we noticed XGBoost was performing the best out of all (results for all the models are in the next section), in terms of 20% test data accuracy. So, we considered this model to predict the status of edges for the year 2020. To do this, we have performed the following steps to extract the future bigrams:

- Firstly, we predict the edge status using the last year's data (year 2019 data). That predicted result (either 0 or 1) will represent that two words can occur or not in the year 2020.

- Then we extracted the bigrams list, where we paired up every node(word) with the rest of the nodes(words) and merged the edge status column of all three years.

- Following this, we also merged the edge status that we predicted, for year 2020, on the test data (year 2019 data) using the XGBoost model.

- Once everything is merged, we run the query to extract the bigrams that never occurred before. It means it will extract only those rows of data which have no edge between the nodes for a given three years of data, but predicted one for the year 2020.

- Once the query is finished executing, we will end up with a list of all possible bigrams that never occurred before but may occur in the next year's patents.

7 Graphical Representation of Approach

Let's see the graphical representation of what we have done so far. This will summarize everything in a very concise way and help visually illustrate the idea.

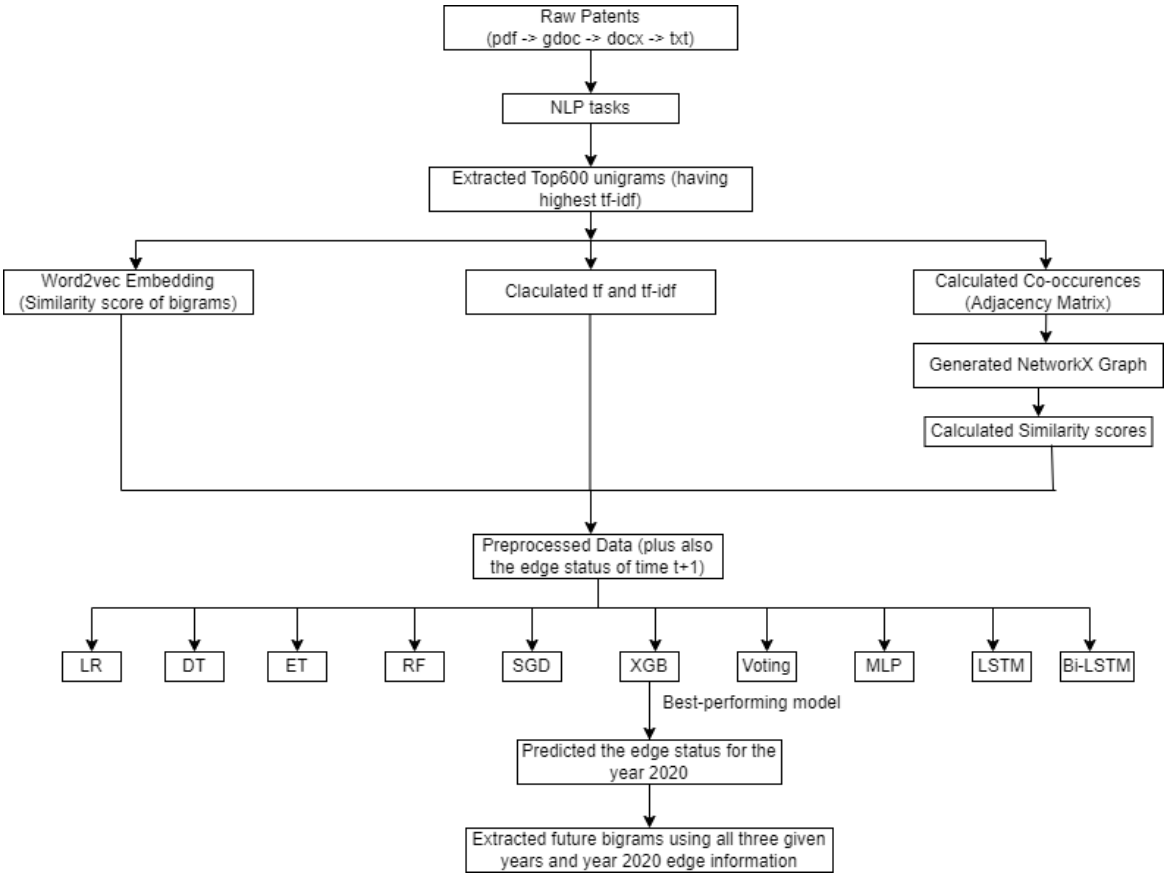


Figure 7: Graphical Representation of Approach

The workflow of the approach can be summarized as follows:

1. Starting the process by converting the files into the appropriate form to read.

2. Then performing NLP tasks on it to get the cleaned data with no redundant information.

3. Once all data is cleaned, we extract the top 600 unigrams according to their tf-idf values, keeping it constant for all three years.

4. For each possible bigram of 600 unigrams, we are doing the following:

   - Calculating cosine between the similarity using Word2vec.
   - Calculating the tf and tf-idf scores.
   - Generating adjacency matrix (words co-occurrences) and constructing NetworkX graph for computing the similarity coefficients.

5. Once everything is done, we can combine them with the edge status for the first two years' data.

6. Then we can pass this data to our ML models to check which one performs best.

7. Once we identify the best-performing model, we can predict the edge status for the year 2020.

8. Having all four years of edge status, three given and one predicted, we can run the query to extract the future bigrams or those which never occurred before.

8 Experimental Results

The model approach was tested using the patents dataset discussed in section 3. Following are the results of all the ML models that were implemented.

Table 1: Experimental Result

Classifiers	Patents Dataset							
	TRAINING SET RESULTS				TEST SET RESULTS			
	Accuracy	Precision	Recall	F1_Score	Accuracy	Precision	Recall	F1_Score
LogisticRegression Classifier	91.93%	98.87%	92.75%	95.71%	91.94%	98.88%	92.75%	95.72%
DecisionTree Classifier	86.94%	100.00%	100.00%	100.00%	86.84%	92.33%	93.17%	92.75%
ExtraTree Classifier	91.83%	100.00%	100.00%	100.00%	91.88%	98.77%	92.78%	95.68%
RandomForest Classifier	91.89%	100.00%	100.00%	100.00%	91.90%	98.80%	92.78%	95.70%
StochasticGradientDescent Classifier	91.77%	99.62%	92.02%	95.67%	91.84%	99.63%	92.06%	95.70%
XGBoost Classifier	92.01%	99.03%	92.82%	95.82%	92.07%	98.97%	92.81%	95.79%
Voting Classifier	93.23%	100.00%	93.07%	96.41%	92.00%	99.22%	92.54%	95.76%
MLP Classifier	92.01%	98.67%	92.99%	95.75%	91.99%	98.67%	92.97%	95.74%
LSTM Classifier	91.79%	99.57%	92.07%	95.67%	91.84%	99.57%	92.11%	95.70%
Bi-LSTM	91.98%	99.17%	92.55%	95.75%	91.99%	99.17%	92.57%	95.76%

Let's discuss the highlights of the results:

- All the models are performing well enough achieving an accuracy of 90% except the decision tree with a bit low accuracy.
- None of the models are overfitting the training data, achieving almost the same accuracies on the test and training set.
- The Voting Classifier achieved the highest accuracy on the training set, but XGBoost is considered to be the best-performing model due to the highest accuracy on the test set.

9 Conclusions and Future Work

- All the implemented models were successfully able to predict future bigrams with a high accuracy score.
- This approach can be used to understand technological trends and future research areas.
- Future work can involve investigating running the same model on different domains or with even more large datasets.

References

[1] USPTO: Large patents collection. https://ppubs.uspto.gov/pubwebapp/static/pages/landing.html.

[2] tf-idf. https://en.wikipedia.org/wiki/Tf%E2%80%93idf, 2023.

[3] Neeraj Mishra. Representation of graphs: Adjacency matrix and adjacency list. https://www.thecrazyprogrammer.com/2014/03/representation-of-graphs-adjacency-matrix-and-adjacency-list.html.

[4] Nur Nasuha Daud, Siti Hafizah Ab Hamid, Muntadher Saadoon, Firdaus Sahran, and Nor Badrul Anuar. Applications of link prediction in social networks: A review. *Journal of Network and Computer Applications*, 166:102716, 2020.

[5] Lada A Adamic and Eytan Adar. Friends and neighbors on the web. *Social Networks*, 25(3):211–230, 2003.

[6] Wang Peng, Xu BaoWen, Wu YuRong, and Zhou XiaoYu. Link prediction in social networks: the state-of-the-art, 2014.

[7] Chonyy. Simrank: Similarity analysis explanation and python implementation from scratch. https://towardsdatascience.com/simrank-similarity-analysis-1d8d5a18766a, 2021.

[8] Mohammad Jaber, Panagiotis Papapetrou, Sven Helmer, and Peter T. Wood1. Using time-sensitive rooted pagerank to detect hierarchical social relationships. https://core.ac.uk/download/pdf/42132551.pdf.

[9] Resource allocation. https://neo4j.com/docs/graph-data-science/current/alpha-algorithms/resource-allocation/.

[10] Dhruvil Karani. Introduction to word embedding and word2vec. https://towardsdatascience.com/introduction-to-word-embedding-and-word2vec-652d0c2060fa, 2018.

[11] Aurélien Géron. *Hands-On Machine Learning witht Scikit-Learn, Keras, and TensorFlow.* O'Reilly, Canada, 2019.