ΕΠΕΞΕΡΓΑΣΙΑ ΦΥΣΙΚΗΣ ΓΛΩΣΣΑΣ

CITATION PREDICTION

NAME: Georgios Strouggis
AM: 5357
TEAM NAME: AI Slop Squad

Περιεχόμενα

OPENING

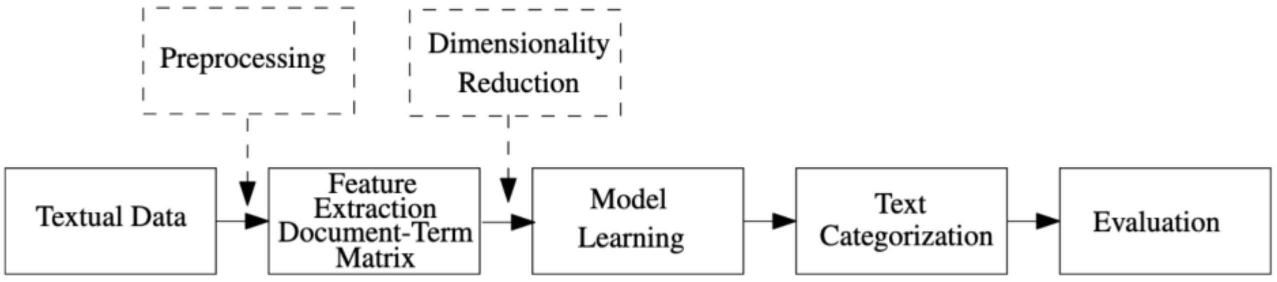
The following report documents the code developed for the production of a link prediction process as part of the course "ΕΠΕΞΕΡΓΑΣΙΑ ΦΥΣΙΚΗΣ ΓΛΩΣΣΑΣ" (ΜΥΕ 053) for the academic year 2024-2025.

We were tasked with developing a program that examines textual data from two research papers to assess whether or not one of them cites the other using a variety of different methods to extract, transform and analyze that data. Provided to us were four txt files:

- abstracts.txt : A numbered list of short texts that make up the information of the 'research papers'.
- authors.txt : A numbered list of authors that correspond to the respective abstracts, being the authors of those texts.
- edgelist.txt : A list of nodes separated by commas used for easier processing of the data, intended to be used to create a graph that maps out papers that are confirmed to be connected by an edge.
- test.txt : A list of edges separated by commas that are to be used once the model has already been trained to predict whether or not they are connected and check its accuracy.

Additionally, there is a fifth file known as 'submission_random.csv' that shows us what our output is supposed to look like, having 106692 and a header. It contains a list of IDs corresponding to the different test examples and their Labels corresponding to the model's confidence of the existence of a link between them.

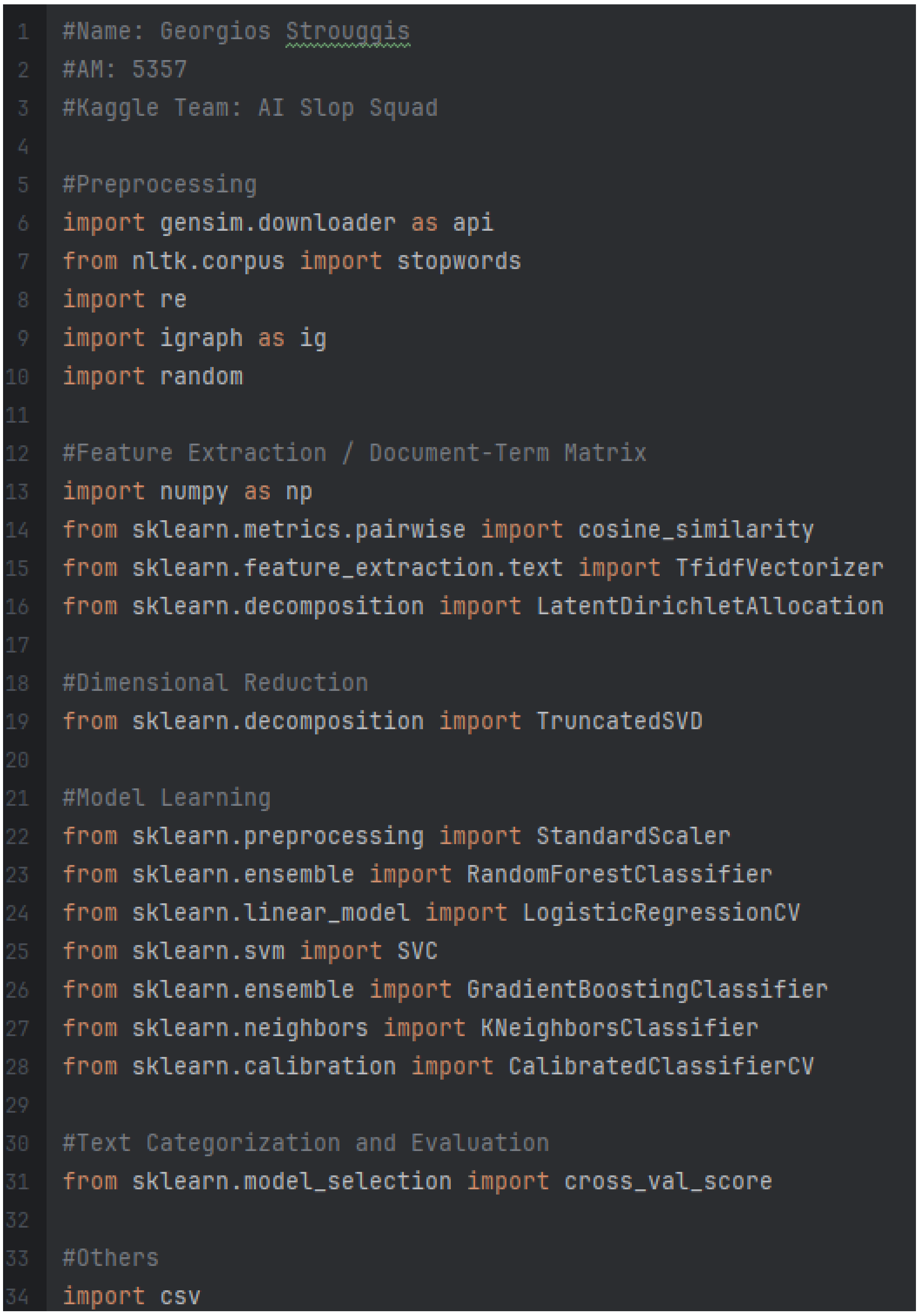
We were given various suggestions for libraries to use and processing methods to develop as well as the following step-by-step plan on how to go about the program, starting with Preprocessing that we will explain first:



PREPROCESSING

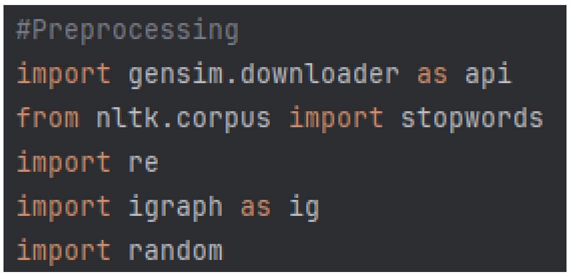
Libraries

JuPyter Notebook was utilized to develop this program and a number of libraries were imported to help us in the stages of the process that were neatly divided according to which section they were used for:



```
#Name: Georgios Strouggis
#AM: 5357
#Kaggle Team: AI Slop Squad

#Preprocessing
import gensim.downloader as api
from nltk.corpus import stopwords
import re
import igraph as ig
import random

#Feature Extraction / Document-Term Matrix
import numpy as np
from sklearn.metrics.pairwise import cosine_similarity
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.decomposition import LatentDirichletAllocation

#Dimensional Reduction
from sklearn.decomposition import TruncatedSVD

#Model Learning
from sklearn.preprocessing import StandardScaler
from sklearn.ensemble import RandomForestClassifier
from sklearn.linear_model import LogisticRegressionCV
from sklearn.svm import SVC
from sklearn.ensemble import GradientBoostingClassifier
from sklearn.neighbors import KNeighborsClassifier
from sklearn.calibration import CalibratedClassifierCV

#Text Categorization and Evaluation
from sklearn.model_selection import cross_val_score

#Others
import csv
```

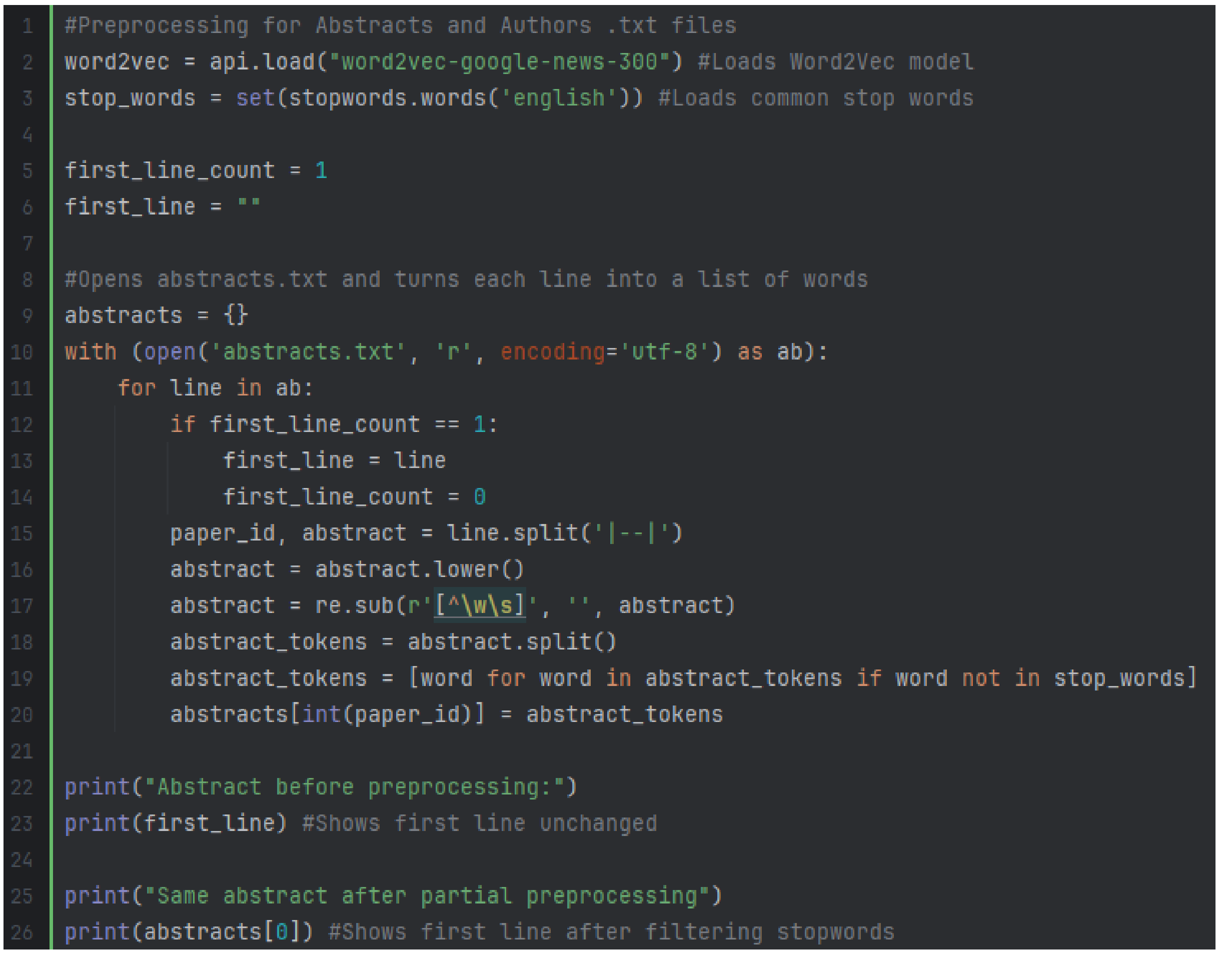
At the moment the ones that concern us are those used for the Preprocessing:



```python
#Preprocessing
import gensim.downloader as api
from nltk.corpus import stopwords
import re
import igraph as ig
import random
```

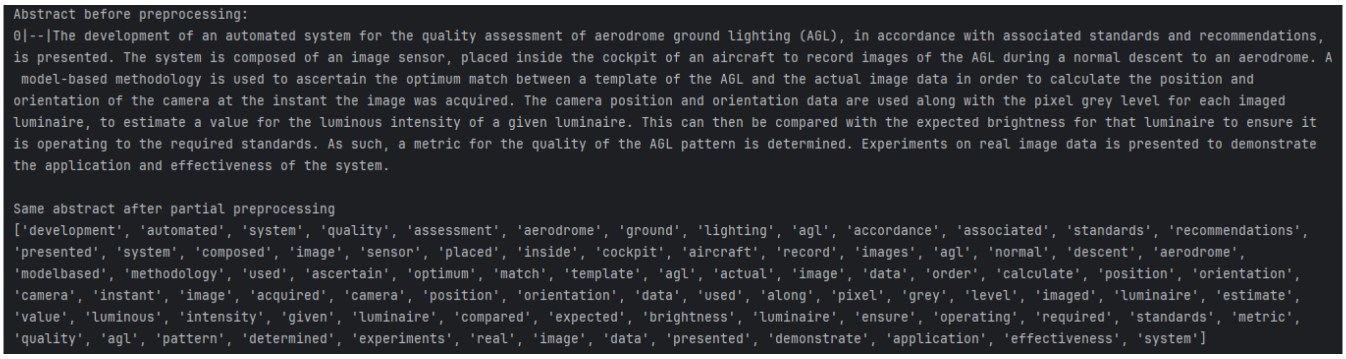
Abstracts

Gensim.downloader provides us with the ‘word2vec-google-news-300’ model[i], a pretrained word embedding model that represents words as 300-dimensional vectors that capture semantic relationships between words and will be utilized during the Feature Engineering section, but for now was imported at the start to be readily available.
Stopwords provides us with a log of common filler words such as ‘the’, ‘and’ etc. that serve only to be noise and need to be filtered out to prevent false positives.
Re serves a similar purpose, though it’s used to clean out punctuation.



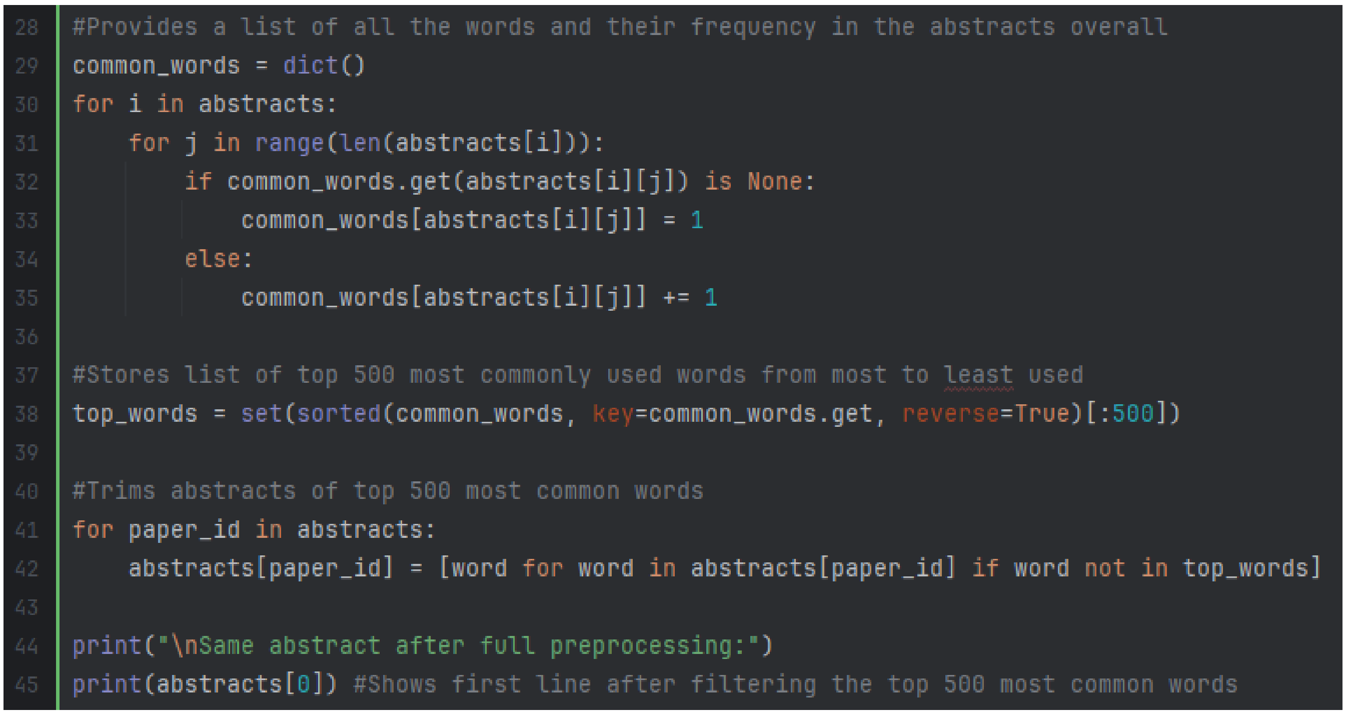
```python
#Preprocessing for Abstracts and Authors .txt files
word2vec = api.load("word2vec-google-news-300") #Loads Word2Vec model
stop_words = set(stopwords.words('english')) #Loads common stop words

first_line_count = 1
first_line = ""

#Opens abstracts.txt and turns each line into a list of words
abstracts = {}
with (open('abstracts.txt', 'r', encoding='utf-8') as ab):
    for line in ab:
        if first_line_count == 1:
            first_line = line
            first_line_count = 0
        paper_id, abstract = line.split('|--|')
        abstract = abstract.lower()
        abstract = re.sub(r'[^\w\s]', '', abstract)
        abstract_tokens = abstract.split()
        abstract_tokens = [word for word in abstract_tokens if word not in stop_words]
        abstracts[int(paper_id)] = abstract_tokens

print("Abstract before preprocessing:")
print(first_line) #Shows first line unchanged

print("Same abstract after partial preprocessing")
print(abstracts[0]) #Shows first line after filtering stopwords
```

To begin extracting the textual data, we open the abstracts.txt file and read through it line by line, saving the first one to print it as an example. As they are being read, each line is split into the number id of the research paper as well as the text, said text is then turned all lowercase to make identifying and matching the different words easier during the feature extraction. It is then stripped of punctuation, further split into tokens with one token for each word before the common stopwords are filtered out. Lastly, the remaining tokens are kept in a list which is then stored in a set with the matching paper id corresponding to it for easy access.
We have incorporated print statements to show the unedited first line straight from the file and what remains of it after the initial 'cleanup':



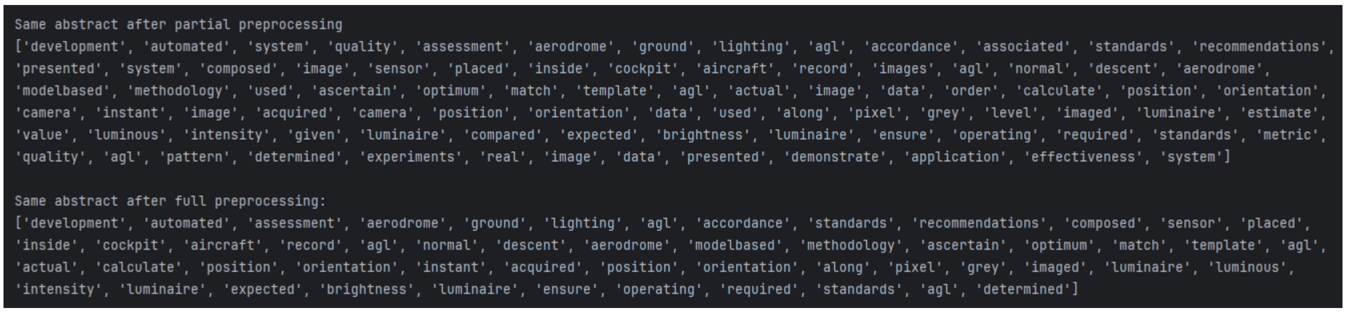
```
Abstract before preprocessing:
0|--|The development of an automated system for the quality assessment of aerodrome ground lighting (AGL), in accordance with associated standards and recommendations,
is presented. The system is composed of an image sensor, placed inside the cockpit of an aircraft to record images of the AGL during a normal descent to an aerodrome. A
 model-based methodology is used to ascertain the optimum match between a template of the AGL and the actual image data in order to calculate the position and
orientation of the camera at the instant the image was acquired. The camera position and orientation data are used along with the pixel grey level for each imaged
luminaire, to estimate a value for the luminous intensity of a given luminaire. This can then be compared with the expected brightness for that luminaire to ensure it
is operating to the required standards. As such, a metric for the quality of the AGL pattern is determined. Experiments on real image data is presented to demonstrate
the application and effectiveness of the system.

Same abstract after partial preprocessing
['development', 'automated', 'system', 'quality', 'assessment', 'aerodrome', 'ground', 'lighting', 'agl', 'accordance', 'associated', 'standards', 'recommendations',
'presented', 'system', 'composed', 'image', 'sensor', 'placed', 'inside', 'cockpit', 'aircraft', 'record', 'images', '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', 'standards', 'metric',
'quality', 'agl', 'pattern', 'determined', 'experiments', 'real', 'image', 'data', 'presented', 'demonstrate', 'application', 'effectiveness', 'system']
```

This is only part of the preprocessing done to the abstracts as there still remains a vast quantity of words that are common amongst the research papers themselves, even unrelated ones that might lead to false positives. So, a nested for loop compiles all the words present in these abstracts as well as their frequency, how many times they have appeared in total across all the abstracts:



```python
#Provides a list of all the words and their frequency in the abstracts overall
common_words = dict()
for i in abstracts:
    for j in range(len(abstracts[i])):
        if common_words.get(abstracts[i][j]) is None:
            common_words[abstracts[i][j]] = 1
        else:
            common_words[abstracts[i][j]] += 1

#Stores list of top 500 most commonly used words from most to least used
top_words = set(sorted(common_words, key=common_words.get, reverse=True)[:500])

#Trims abstracts of top 500 most common words
for paper_id in abstracts:
    abstracts[paper_id] = [word for word in abstracts[paper_id] if word not in top_words]

print("\nSame abstract after full preprocessing:")
print(abstracts[0]) #Shows first line after filtering the top 500 most common words
```

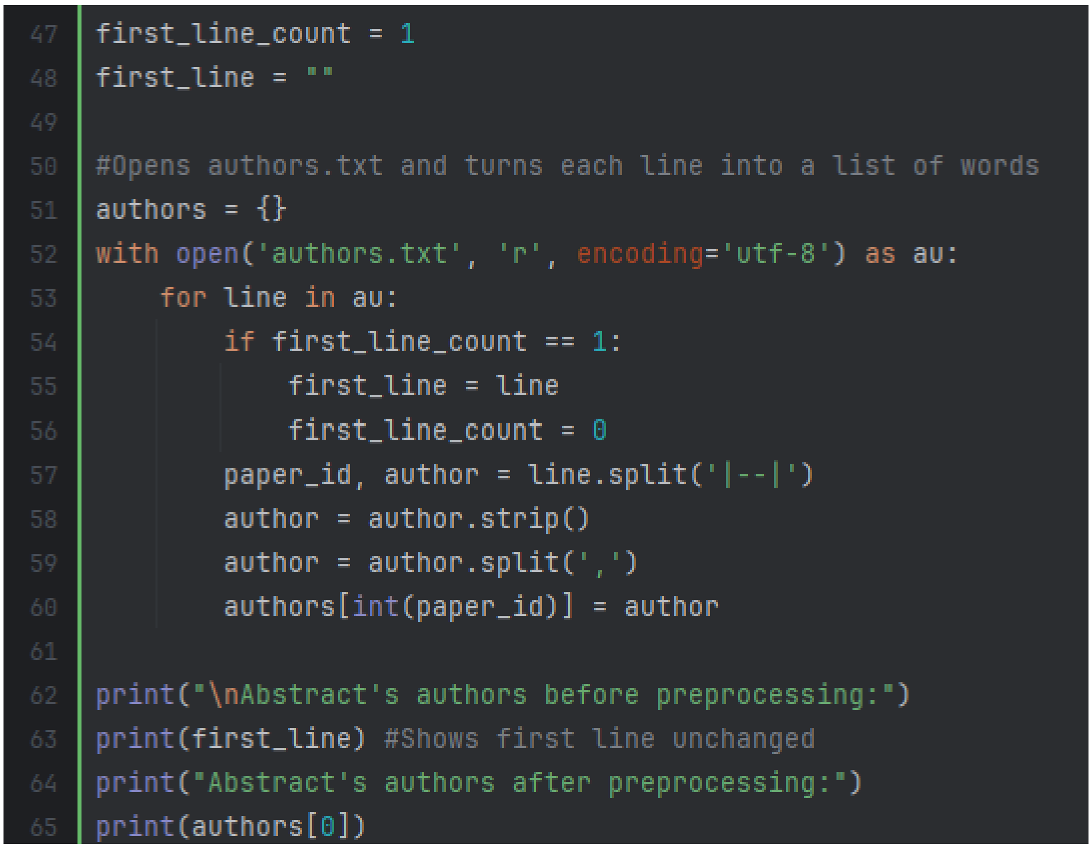
If it finds a word that's not in the dictionary yet it saves it with the value of 1, but if it's already present then it increments that value by 1. After that these words are sorted from most to least used (reverse=True is what placed them in descending order) and the top 500 of them are kept as the most commonly used words period. Initially the top 1000 were kept, but that number was shortened to the top 500 when it produced better results during optimization.
Now that we had the top 500 most common words, each abstract was taken and stripped of them, completing our preprocessing of the abstracts:



```
Same abstract after partial preprocessing
['development', 'automated', 'system', 'quality', 'assessment', 'aerodrome', 'ground', 'lighting', 'agl', 'accordance', 'associated', 'standards', 'recommendations',
'presented', 'system', 'composed', 'image', 'sensor', 'placed', 'inside', 'cockpit', 'aircraft', 'record', 'images', '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', 'standards', 'metric',
'quality', 'agl', 'pattern', 'determined', 'experiments', 'real', 'image', 'data', 'presented', 'demonstrate', 'application', 'effectiveness', 'system']

Same abstract after full preprocessing:
['development', 'automated', 'assessment', 'aerodrome', 'ground', 'lighting', 'agl', 'accordance', 'standards', 'recommendations', 'composed', 'sensor', 'placed',
'inside', 'cockpit', 'aircraft', 'record', 'agl', 'normal', 'descent', 'aerodrome', 'modelbased', 'methodology', 'ascertain', 'optimum', 'match', 'template', 'agl',
'actual', 'calculate', 'position', 'orientation', 'instant', 'acquired', 'position', 'orientation', 'along', 'pixel', 'grey', 'imaged', 'luminaire', 'luminous',
'intensity', 'luminaire', 'expected', 'brightness', 'luminaire', 'ensure', 'operating', 'required', 'standards', 'agl', 'determined']
```

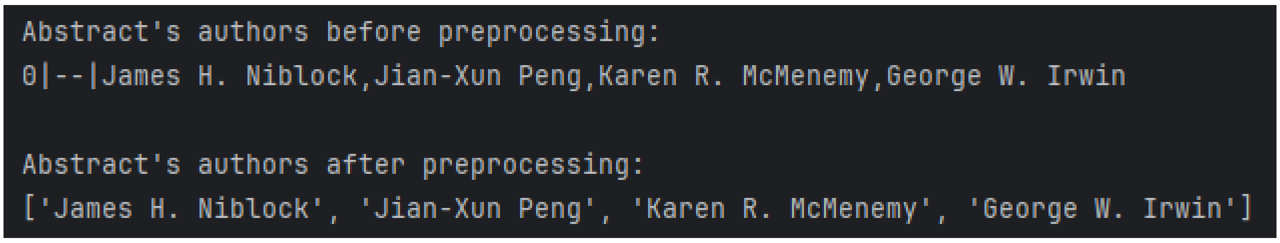
By removing these words we filter out the common terms in the scientific field that don't necessarily mean there is a relation between research papers.

Authors



```python
first_line_count = 1
first_line = ""

#Opens authors.txt and turns each line into a list of words
authors = {}
with open('authors.txt', 'r', encoding='utf-8') as au:
    for line in au:
        if first_line_count == 1:
            first_line = line
            first_line_count = 0
        paper_id, author = line.split('|--|')
        author = author.strip()
        author = author.split(',')
        authors[int(paper_id)] = author

print("\nAbstract's authors before preprocessing:")
print(first_line) #Shows first line unchanged
print("Abstract's authors after preprocessing:")
print(authors[0])
```

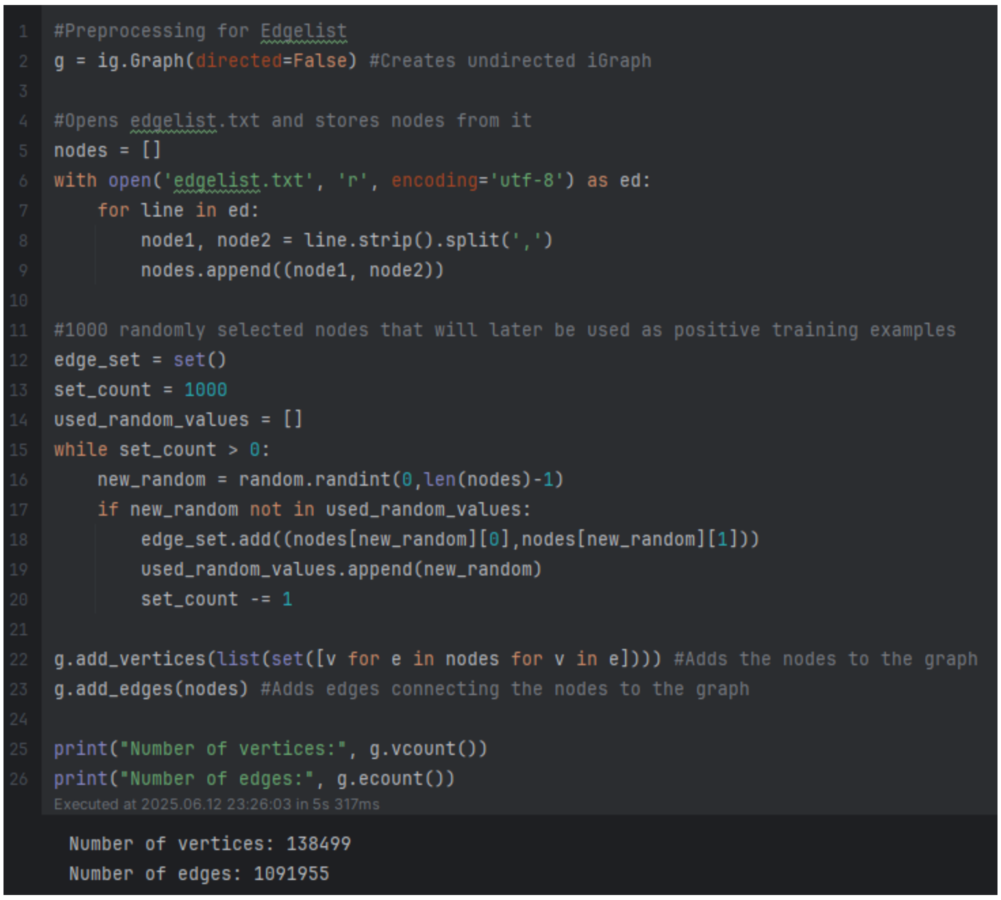
The preprocessing of the authors is simpler in comparison as all we needed to do was split the lines between their number and the name list, then trim it of things like the '\n' character, split it into names and save each list by the respective paper id. There is not much further preprocessing needed to be done as they are simple names. A person's full name is kept as a single token.



```
Abstract's authors before preprocessing:
0|--|James H. Niblock,Jian-Xun Peng,Karen R. McMenemy,George W. Irwin

Abstract's authors after preprocessing:
['James H. Niblock', 'Jian-Xun Peng', 'Karen R. McMenemy', 'George W. Irwin']
```

Graph

Next we construct an undirected graph to handle the edgelist.txt. During lessons for this course iGraph was specifically mentioned and recommended by the professor so it was chosen to be used for the graph creation. The reason it is undirected is because we are not concerned with which of the two papers cited the other and making it directed would limit our options.



```python
#Preprocessing for Edgelist
g = ig.Graph(directed=False) #Creates undirected iGraph

#Opens edgelist.txt and stores nodes from it
nodes = []
with open('edgelist.txt', 'r', encoding='utf-8') as ed:
    for line in ed:
        node1, node2 = line.strip().split(',')
        nodes.append((node1, node2))

#1000 randomly selected nodes that will later be used as positive training examples
edge_set = set()
set_count = 1000
used_random_values = []
while set_count > 0:
    new_random = random.randint(0,len(nodes)-1)
    if new_random not in used_random_values:
        edge_set.add((nodes[new_random][0],nodes[new_random][1]))
        used_random_values.append(new_random)
        set_count -= 1

g.add_vertices(list(set([v for e in nodes for v in e]))) #Adds the nodes to the graph
g.add_edges(nodes) #Adds edges connecting the nodes to the graph

print("Number of vertices:", g.vcount())
print("Number of edges:", g.ecount())
```

Executed at 2025.06.12 23:26:03 in 5s 317ms

```
Number of vertices: 138499
Number of edges: 1091955
```

Opening edgelist.txt we see that it looks something like this:

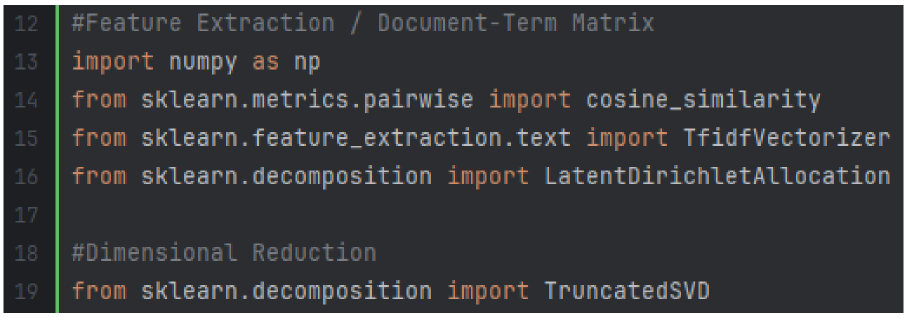
0,1
0,2
1,3

So for each line we split it and append the two nodes (vertices) connected by an edge to a list. Additionally, after completing the list of nodes we randomly select 1000 edges from it while keeping track of the ones we have already selected so we don't duplicate them. This edge_set will serve as the confirmed positive connections that will be used for training, since they come from the list of confirmed connections in edgelist. Originally we had chosen 5000 positive examples, but that proved to overtrain the program so we reduced it to 1000.

After that we add all of the nodes to the graph as well as the edges connecting them before we print the total number of those nodes as well as the edges.
Now that we have all of the confirmed citations in a graph it can later be used to extract features, so this is the part where we complete the Preprocessing section.

FEATURE ENGINEERING

These are the libraries we used for this section of the code:

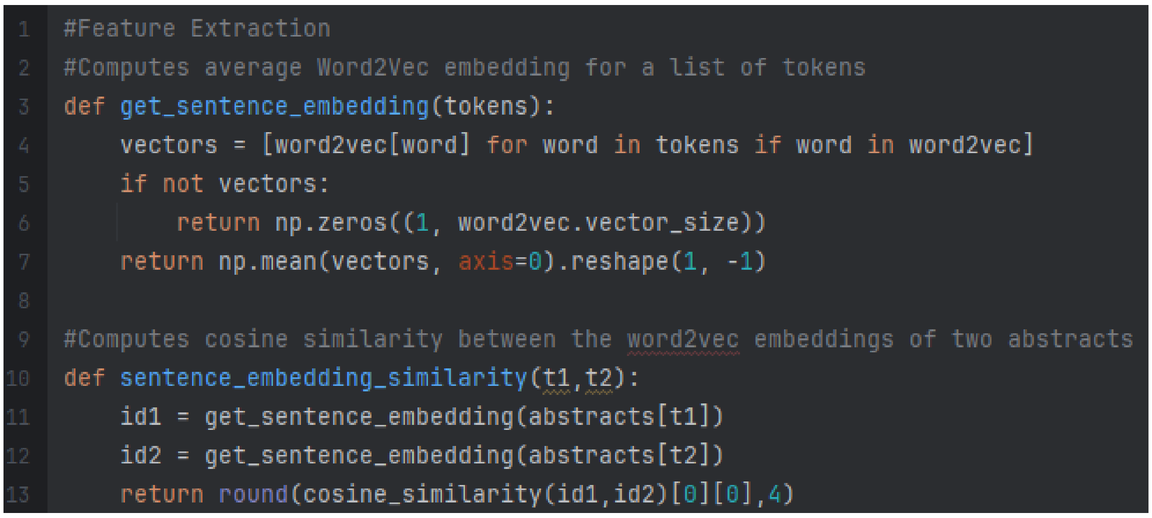


```python
#Feature Extraction / Document-Term Matrix
import numpy as np
from sklearn.metrics.pairwise import cosine_similarity
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.decomposition import LatentDirichletAllocation

#Dimensional Reduction
from sklearn.decomposition import TruncatedSVD
```

Numpy helped perform mathematical tasks, cosine similarity, TF-IDF, Latent Dirichlet Allocation were used to create certain features and TruncatedSVD for dimensional reduction.

A variety of different features were employed. First we will go over the ones that made it to the final version of the code and then discuss different features that were attempted at various points during production but were cut as well as why and our inspiration for those features, both the ones scrapped and the ones kept.

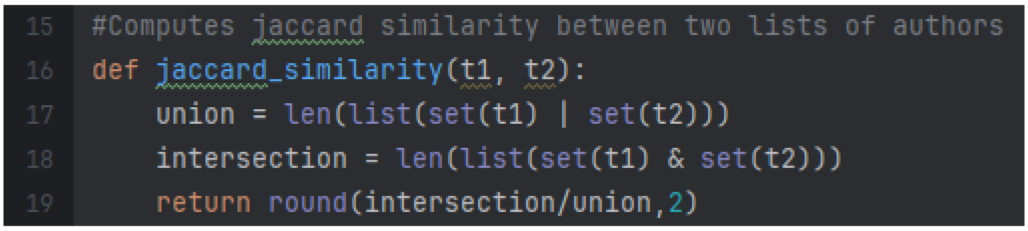
Final Features



```python
#Feature Extraction
#Computes average Word2Vec embedding for a list of tokens
def get_sentence_embedding(tokens):
    vectors = [word2vec[word] for word in tokens if word in word2vec]
    if not vectors:
        return np.zeros((1, word2vec.vector_size))
    return np.mean(vectors, axis=0).reshape(1, -1)

#Computes cosine similarity between the word2vec embeddings of two abstracts
def sentence_embedding_similarity(t1,t2):
    id1 = get_sentence_embedding(abstracts[t1])
    id2 = get_sentence_embedding(abstracts[t2])
    return round(cosine_similarity(id1,id2)[0][0],4)
```

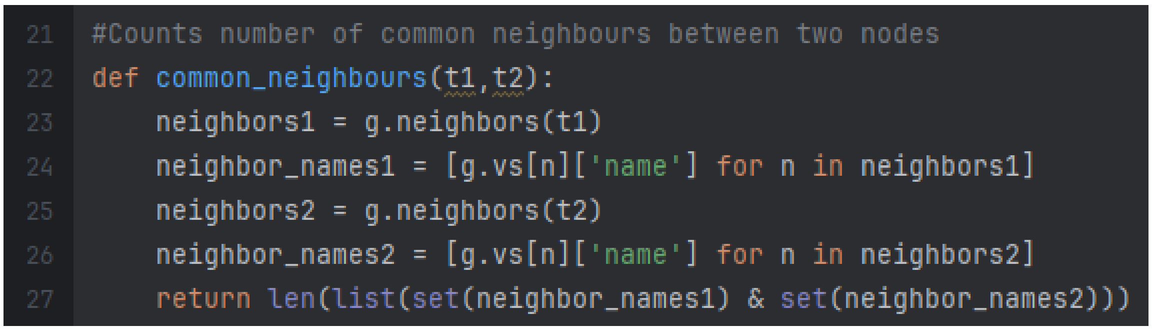
As we said above, word2vec was imported into the program, allowing us to look up its embedding in the pretrained model. Word2Vec maps words to high-dimensional space where semantically similar words are closer, so similar words will have similar values. For the get_sentence_embedding function we insert a list of tokens (the preprocessed abstract) and find the pretrained embedding for that word (300-dimension vector) which we then store in a list. If word2vec does not recognise any of the words in its database then it returns a zero vector, otherwise it returns the mean average of all the vectors for that abstract. This provides us with a value unique to a particular abstract.

Then using sentence_embedding_similarity we perform this process for two abstracts and then compute the cosine similarity which measures the similarity between two vectors by computing the cosine of the angle between them. A small cosine similarity close to zero means they are unrelated while a high one close to one means they are related, thus it's likely that one paper cites the other.
This was our first feature meant to find the textual similarity between two abstracts, we came up with it by doing research on tokenizers[ii] and settling on word2vec[iii] after it gave us the best results. Word2Vec was one of the suggested models for the project alongside others that we will talk about in the scrapped feature section. Also, we chose to use a pretrained word2vec model seeing as relevant scientific models were encouraged by the professor.



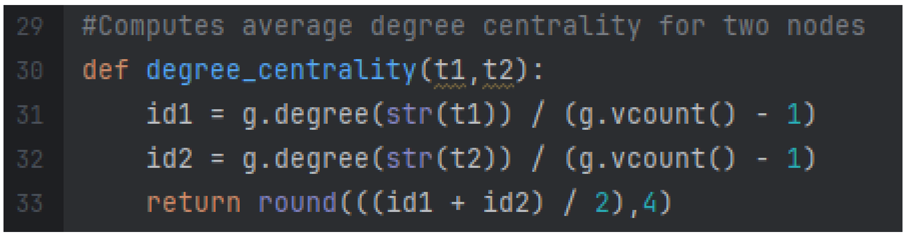
```python
#Computes jaccard similarity between two lists of authors
def jaccard_similarity(t1, t2):
    union = len(list(set(t1) | set(t2)))
    intersection = len(list(set(t1) & set(t2)))
    return round(intersection/union,2)
```

Next we implemented a function to calculate the jaccard similarity between two lists of authors. Jaccard similarity measures the overlap between two sets to see how many they have in common in proportion to the total number of authors for both. It provides a score between zero and one with higher values meaning they share more authors, so it's more likely they collaborated for both research papers or cited themselves or each other. There isn't a lot of available information on the authors besides their name so this was all we could do to derive features from them using a well known design that we discovered with only a little bit of research[iv].



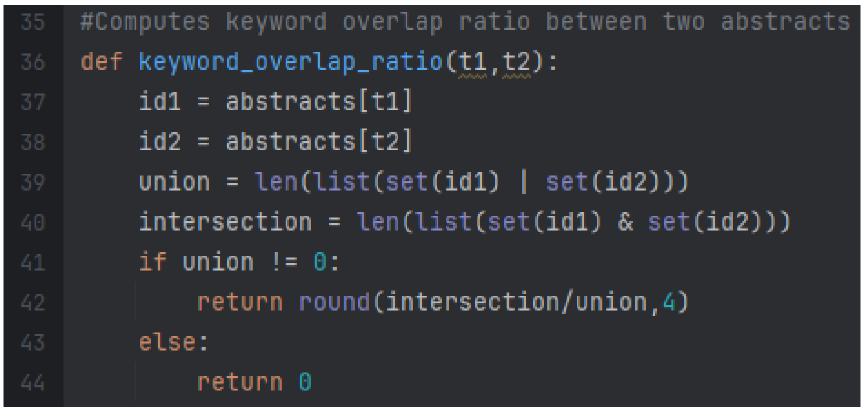
```python
#Counts number of common neighbours between two nodes
def common_neighbours(t1,t2):
    neighbors1 = g.neighbors(t1)
    neighbor_names1 = [g.vs[n]['name'] for n in neighbors1]
    neighbors2 = g.neighbors(t2)
    neighbor_names2 = [g.vs[n]['name'] for n in neighbors2]
    return len(list(set(neighbor_names1) & set(neighbor_names2)))
```

This next function utilizes the graph to find the list of neighbours between the nodes corresponding to two research papers on the graph. It then compiles list of the names of those neighbours and returns the number of the ones they have in common. A higher value of common neighbours means a higher likelihood of correlation between them. It was implemented since it was suggested in the description for the project as a possible feature to extract after we did research on the official iGraph page[v].



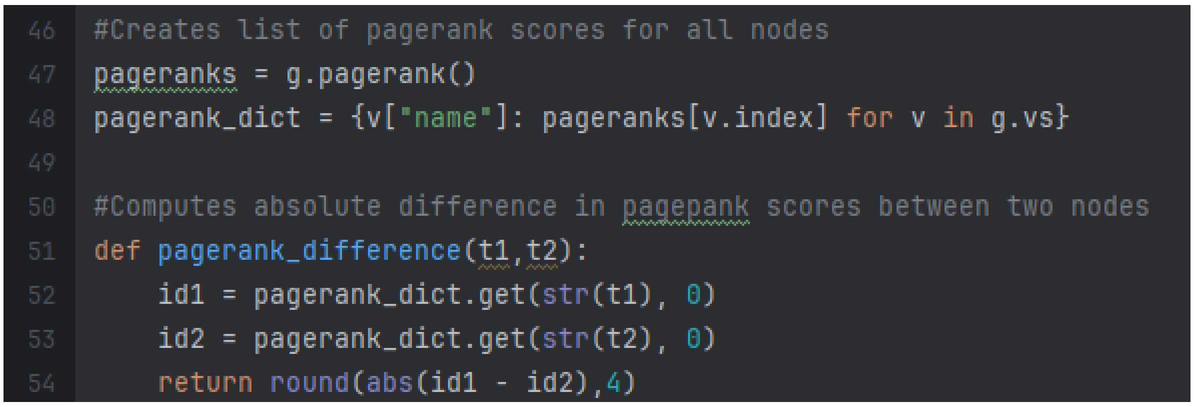
```python
#Computes average degree centrality for two nodes
def degree_centrality(t1,t2):
    id1 = g.degree(str(t1)) / (g.vcount() - 1)
    id2 = g.degree(str(t2)) / (g.vcount() - 1)
    return round(((id1 + id2) / 2),4)
```

Using g.degree we find the number of edges connected to a node in the graph and then divide it by the total number of nodes in the graph as a whole. What this achieves is that it tells us how ‘popular’ a node is, how often it is cited by others which helps us determine if it is a well known or obscure research paper which increases and decreases the likelihood of a link between it and another paper existing. We compute the average between them to determine their collective popularity in the network. We were inspired to include this feature by looking up different functions we can implement to extract features from a graph[vi][vii] but also because it was mentioned in the project summary.



```python
#Computes keyword overlap ratio between two abstracts
def keyword_overlap_ratio(t1,t2):
    id1 = abstracts[t1]
    id2 = abstracts[t2]
    union = len(list(set(id1) | set(id2)))
    intersection = len(list(set(id1) & set(id2)))
    if union != 0:
        return round(intersection/union,4)
    else:
        return 0
```

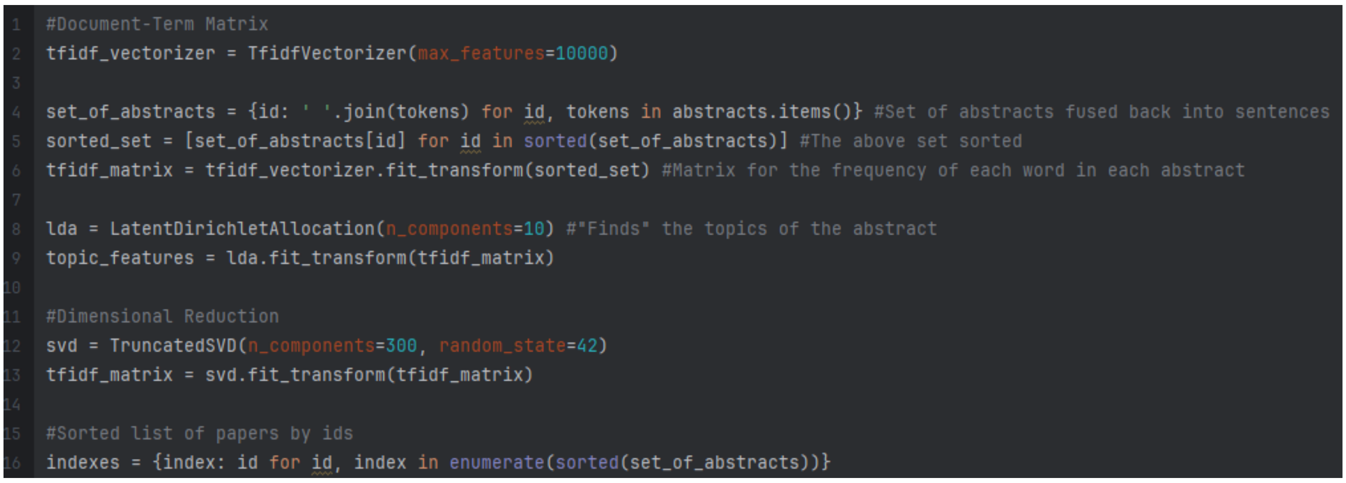
Keyword_overlap_ratio performs a similar process to jaccard similarity albeit for the abstracts, giving us the ratio of words shared between the two research papers in proportion to the total number of words between them. The higher the ratio the more words are common between them and the likelier it is that one cited the other. The if statement was included to amend a divided by zero error that we got during the process. It serves as an alternative method of feature extraction implemented after we finalized word2vec just to extract another feature from the abstracts to produce better results with a simple but effective method.



```python
#Creates list of pagerank scores for all nodes
pageranks = g.pagerank()
pagerank_dict = {v["name"]: pageranks[v.index] for v in g.vs}

#Computes absolute difference in pagepank scores between two nodes
def pagerank_difference(t1,t2):
    id1 = pagerank_dict.get(str(t1), 0)
    id2 = pagerank_dict.get(str(t2), 0)
    return round(abs(id1 - id2),4)
```

Pagerank[viii] is an algorithm that ranks nodes based on their importance in the graph. This might seem similar to degree centrality, but it doesn't just measure the number of connections a node has but also the quality of them in proportion to the network as a whole, for that reason we have to calculate the pagerank for the entire graph rather than a select number of nodes as that can give us the full scope of information which we then store in a dictionary for easy access. After the scores have been computed for all nodes it computes the absolute difference between the two nodes we give it, a low score close to zero means the two nodes have relative importance and higher likelihood of being connected. It should also be mentioned that this is the algorithm Google uses to rank web pages in their search engine results[ix].

Document-Term Matrix

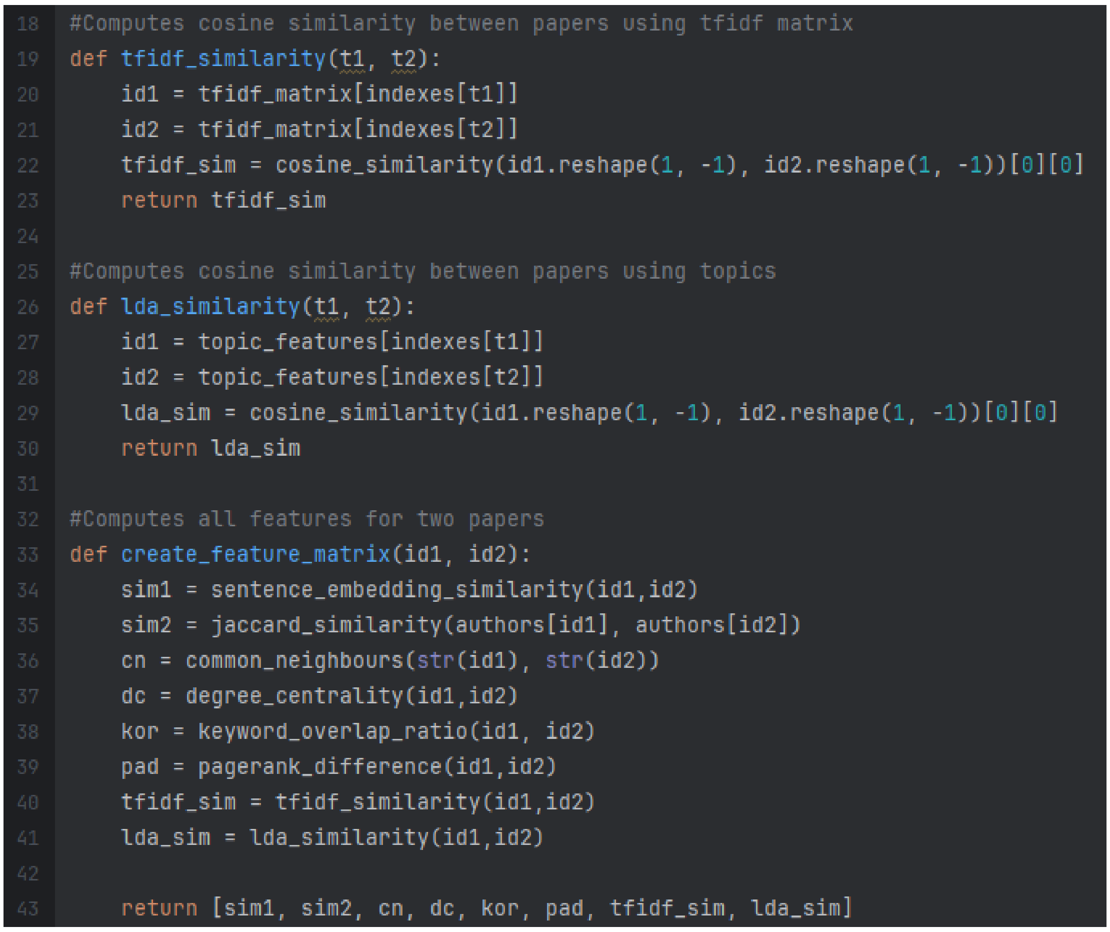


```python
#Document-Term Matrix
tfidf_vectorizer = TfidfVectorizer(max_features=10000)

set_of_abstracts = {id: ' '.join(tokens) for id, tokens in abstracts.items()} #Set of abstracts fused back into sentences
sorted_set = [set_of_abstracts[id] for id in sorted(set_of_abstracts)] #The above set sorted
tfidf_matrix = tfidf_vectorizer.fit_transform(sorted_set) #Matrix for the frequency of each word in each abstract

lda = LatentDirichletAllocation(n_components=10) #"Finds" the topics of the abstract
topic_features = lda.fit_transform(tfidf_matrix)

#Dimensional Reduction
svd = TruncatedSVD(n_components=300, random_state=42)
tfidf_matrix = svd.fit_transform(tfidf_matrix)

#Sorted list of papers by ids
indexes = {index: id for id, index in enumerate(sorted(set_of_abstracts))}
```

TF-IDF is a well known model for documenting the importance of a word within a document or a collection of documents. The max_features[x] builds a vocabulary that will only use the top x most used terms with x being 10000 for us, this prevents overtraining by feeding it too much information. We initially chose 1000, however further testing revealed 10000 gave better results.

First we fuse the tokens of a sentence back into one with spaces dividing them and then we sort them for consistency, then they are given to the vectorizer which will construct

a matrix with 10000 features. What the TF part of TF-IDF does is that it measures the frequency of a term within a document compared to the total number of terms in it and then the IDF part measures its rarity among the whole collection of documents, providing us with unique values for those 10000 most common features[xi].
These TF-IDF vectors are high dimensional so we used TruncatedSVD to reduce dimensionality to 300 dimensions which filters out noise while maintaining the important parts[xii].
TF-IDF was discussed during classes for the course so it was noted to be implemented into the program and the dimensional reduction was done as part of the whole process detailed in the opening segment for the sake of data management.
Additionally, we implemented Latent Dirichlet Allocation[xiii][xiv] which is essentially a method of finding topics present in the matrix, in this case n_components=10 topics. What this does is that it can determine the topic of two papers to see if they are about the same subject. This feature was discovered when searching for different feature ideas.
Lastly we keep a set of indexes that numbers all of these abstracts by their ids for easy access.



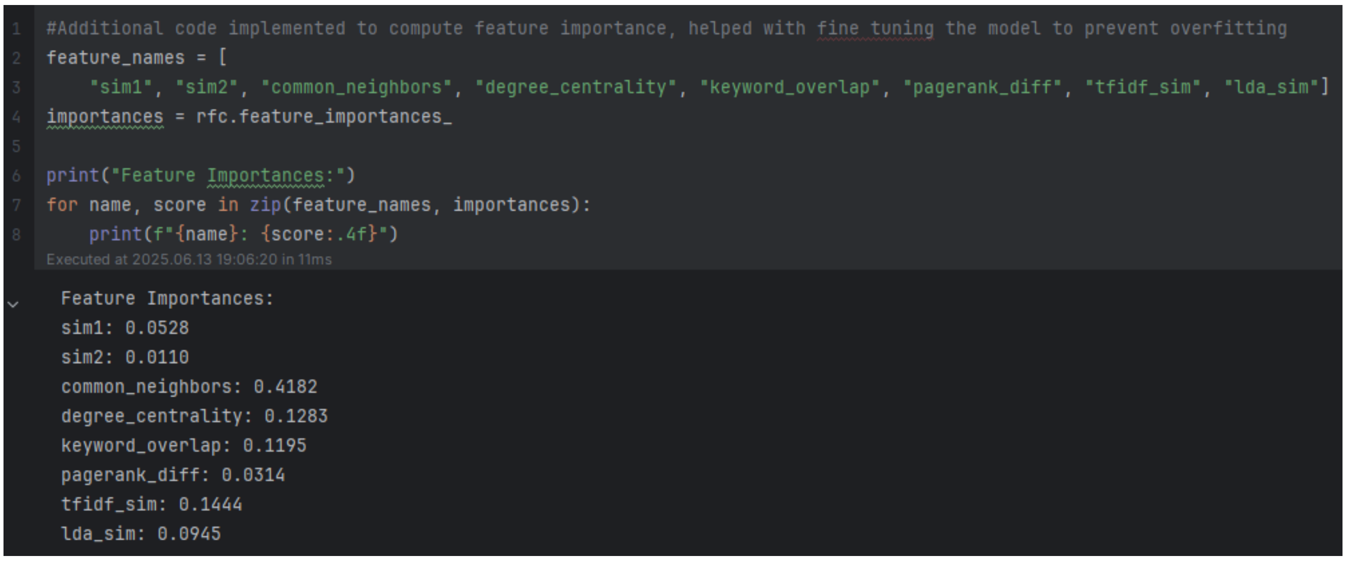
```python
#Computes cosine similarity between papers using tfidf matrix
def tfidf_similarity(t1, t2):
    id1 = tfidf_matrix[indexes[t1]]
    id2 = tfidf_matrix[indexes[t2]]
    tfidf_sim = cosine_similarity(id1.reshape(1, -1), id2.reshape(1, -1))[0][0]
    return tfidf_sim

#Computes cosine similarity between papers using topics
def lda_similarity(t1, t2):
    id1 = topic_features[indexes[t1]]
    id2 = topic_features[indexes[t2]]
    lda_sim = cosine_similarity(id1.reshape(1, -1), id2.reshape(1, -1))[0][0]
    return lda_sim

#Computes all features for two papers
def create_feature_matrix(id1, id2):
    sim1 = sentence_embedding_similarity(id1,id2)
    sim2 = jaccard_similarity(authors[id1], authors[id2])
    cn = common_neighbours(str(id1), str(id2))
    dc = degree_centrality(id1,id2)
    kor = keyword_overlap_ratio(id1, id2)
    pad = pagerank_difference(id1,id2)
    tfidf_sim = tfidf_similarity(id1,id2)
    lda_sim = lda_similarity(id1,id2)

    return [sim1, sim2, cn, dc, kor, pad, tfidf_sim, lda_sim]
```

Now that we have these models set up, we created two more functions to compute the cosine similarity between papers based on their TF-IDF and LDA representations, allowing us to quantify how similar two abstracts are based on word usage and topic distribution.

At last we have the create_feature_matrix function that takes two research papers and computes all of the above features for them, numbering eight features in total that will be used for training.

Feature Importance

Before we proceed to the scrapped features, an important part of the optimization process was determining the feature importances.



```
#Additional code implemented to compute feature importance, helped with fine tuning the model to prevent overfitting
feature_names = [
    "sim1", "sim2", "common_neighbors", "degree_centrality", "keyword_overlap", "pagerank_diff", "tfidf_sim", "lda_sim"]
importances = rfc.feature_importances_

print("Feature Importances:")
for name, score in zip(feature_names, importances):
    print(f"{name}: {score:.4f}")
Executed at 2025.06.13 19:06:20 in 11ms
```

```
Feature Importances:
sim1: 0.0528
sim2: 0.0110
common_neighbors: 0.4182
degree_centrality: 0.1283
keyword_overlap: 0.1195
pagerank_diff: 0.0314
tfidf_sim: 0.1444
lda_sim: 0.0945
```

Essentially, after training our model we wanted to analyze to what extent each feature influenced it, a high score meant it was important for our results, but too high meant it ran the risk of overtraining the model with specific data or making it too easy to find connections that led to it being overconfident and not perform as well on foreign data, while a lower score meant it bears little impact so it could range from being harmless to potentially useless and just slowing down the code. By analyzing it repeatedly as we included different features to see their impact we decided to keep or remove them.

Keep in mind that a high importance isn't necessarily bad, common_neighbors' importance is the highest one but removing it only worsens our scores. Essentially what a high importance means is that including or removing this particular feature will show noticeable changes, whether good or bad.

Scrapped Features

The list of scrapped features includes:

- Shortest path distance in graph: one of the features suggested in the project description, it was fully implemented but massively overtrained the program with the test inputs so it had poor results for foreign inputs. It had a feature importance of 0.6 which was much higher than anything else, so it was removed for better results.
- BERT embeddings[xv][xvi]: Pretrained tokenizer that represents words as vectors that reflect their meaning within the context of the surrounding words. We were inspired to use it after hearing about it during classes for the course. Fully implemented and functional originally as part of get_sentence_embedding before it was replaced by word2vec which had better results, BERT was too confident and gave way too high scores for textually distinct papers.
- Sentence BERT embeddings: A more fine tuned version of BERT meant to produce semantically meaningful sentence embeddings, discovered through research on versions of BERT. Never implemented because it couldn't be imported into the program.
- Node2Vec embeddings[xvii]: Model that learns vector representations of nodes in a graph by simulating random walks, it captures both nodes that are directly connected and nodes that share similar neighbourhoods, allowing structurally or contextually similar nodes to have similar embeddings. Considered and partially implemented but had some issues, it needed to use network graph because it wouldn't accept iGraph so we had to transfer all the nodes to it, but even after that it wouldn't work, so we scrapped it.
- Transitivity difference[xviii]: The probability that the adjacent vertexes of a node are also connected. Fully implemented and would compare if two nodes had similar results, but it was ultimately scrapped because it didn't serve a practical purpose that helped identify if two papers were related.
- Eigenvector Centrality difference[xix]: Quantifies the importance of a node based on the importance of its connections. Fully implemented but found to be redundant since we already have degree centrality and pagerank, it only made the score worse.

- Authority Score difference[xx]: Computes how much ‘authority’ a node has based on the number and quality of links to it. Fully implemented but again, it was found to be redundant and only made the score worse.
- Hub Score difference[xxi]: A hub is a node that links to many high-authority nodes, so it and the authority score work together. Fully implemented but once again unhelpful.
- Coreness difference[xxii]: Coreness is how deep a node is in the graph’s ‘core’, meaning how dense the connections to it and around it are. Fully implemented but redundant, didn’t improve score so it was removed.
- Constraint difference[xxiii]: Measures redundancy of a node’s connections, a high value means that a lot of its neighbours are connected. Fully implemented but ultimately unhelpful because it doesn’t actually provide any information that can be used to determine if the papers themselves are linked, just tells us something about their respective neighbours. Also did not improve scores.

All of the above scrapped features besides the shortest distance, BERT and SBERT and Node2Vec were discovered by reading through the official iGraph manual[xxiv] and finding different functions that on a first glance seemed like they would be useful, so they were considered.

Timeline

The timeline of events for the different features we had implemented, added or removed is as follows:

- Phase 1 - Sentence Embedding Similarity, Jaccard Similarity, Common Neighbours, Shortest Path
  - Very poor results, massively overtrained by the Shortest Path as we discovered from the feature importances.
- Phase 2 – Sentence Embedding Similarity, Jaccard Similarity, Common Neighbours
  - Much better results but still not good enough, needed more fine tuning and new features.
- Phase 3 – Same as above, but fine tuned in a lot of different parts like the preprocessing, the positive/negative selection, the classifiers etc.
  - Better results, still needed more features.

- Phase 4 - Sentence Embedding Similarity, Jaccard Similarity, Common Neighbours, Degree Centrality, Transitivity Difference, Keyword Overlap Ratio, TF-IDF Similarity
    - Better results, could be better.
- Phase 5 – Same as above, just fine tuned the classifier and some other things
    - Marginally better.
- Phase 6 - Sentence Embedding Similarity, Jaccard Similarity, Common Neighbours, Degree Centrality, Transitivity Difference, Keyword Overlap Ratio, TF-IDF Similarity, Eigenvector Centrality Difference, Authority Score Difference, Hub Score Difference, Coreness Difference, Constraint Difference
    - Worse, it was clear all of these extra features weren't giving it any useful information.
- Phase 7 - Sentence Embedding Similarity, Jaccard Similarity, Common Neighbours, Degree Centrality, Keyword Overlap Ratio, Pagerank Difference, TF-IDF Similarity, LDA Similarity
    - Best result to date, satisfying conclusion.

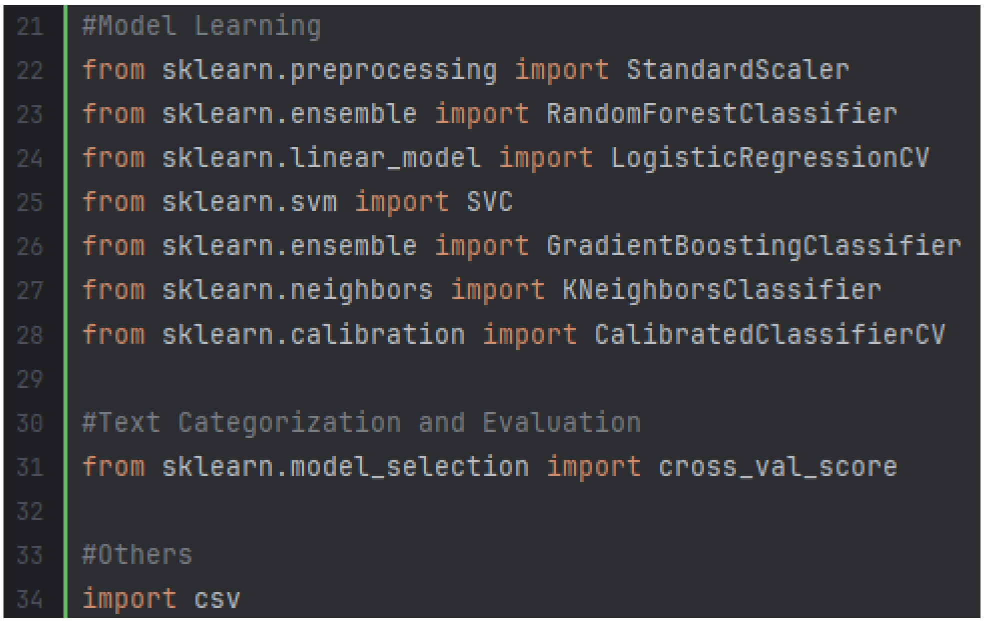
The quality of a feature was evaluated on its effect on the F1 score and logloss as an abnormally high score usually indicated some type of overfitting. The Kaggle public score was also observed to see if our features improved it, but we didn't want to focus on that too much to prevent overfitting it for the Kaggle score alone.

There were more minor changes implemented all throughout these phases, all scrapped features that are not mentioned in this timeline were either dropped before the evaluation phase and thus we never saw what kind of score they would have had or were just never finalized.

Largely when it came to features our mindset when developing them was to look for the obvious things first such as the similarity between abstracts of authors or abstracts in different forms, different ways nodes could be connected through the graph etc. before diving deeper with more obscure graph features or other well known methods of analyzing data such as TF-IDF and just trying out different things to see what works.

MODEL, TUNING AND COMPARISON

This next part of the report refers to the Model Learning, Text Categorization and Evaluation phases of the program.

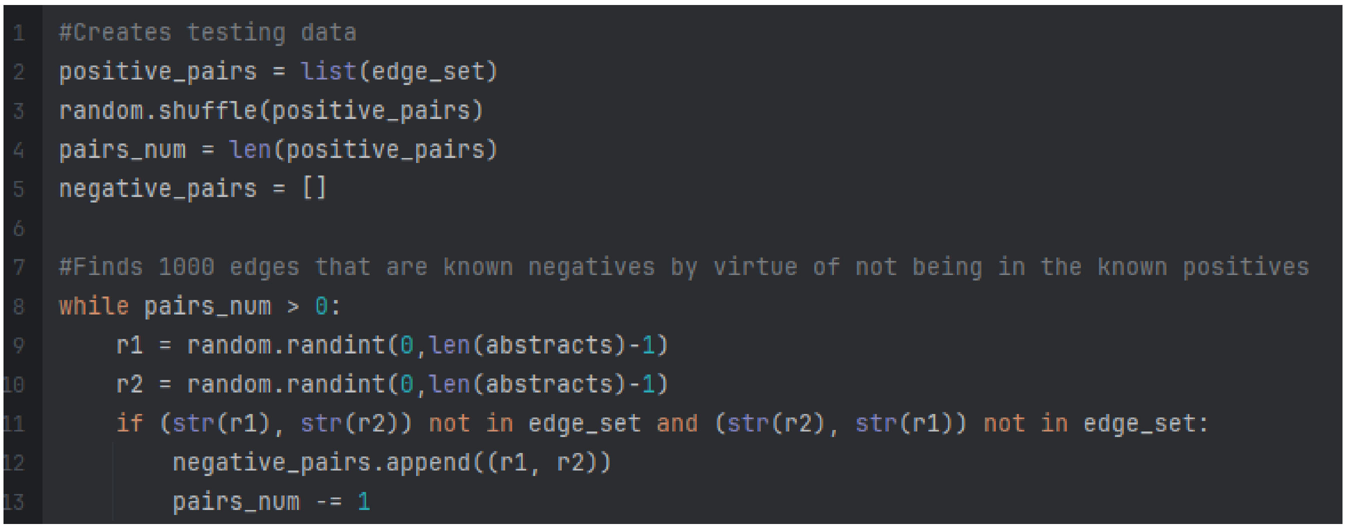


```python
#Model Learning
from sklearn.preprocessing import StandardScaler
from sklearn.ensemble import RandomForestClassifier
from sklearn.linear_model import LogisticRegressionCV
from sklearn.svm import SVC
from sklearn.ensemble import GradientBoostingClassifier
from sklearn.neighbors import KNeighborsClassifier
from sklearn.calibration import CalibratedClassifierCV

#Text Categorization and Evaluation
from sklearn.model_selection import cross_val_score

#Others
import csv
```

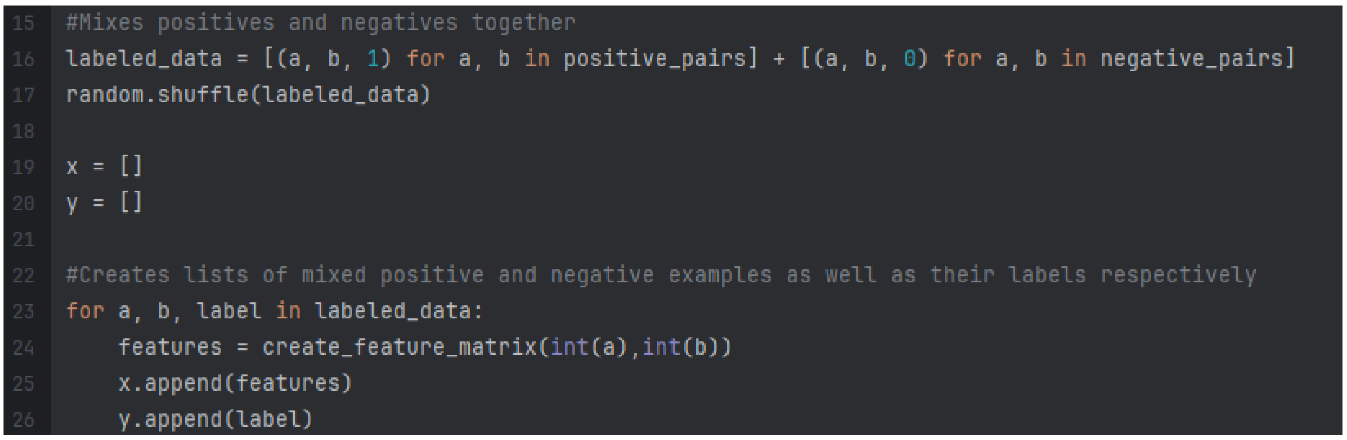
For this segment of the program we utilized 5 different well known classifiers, some of which will have their data undergo standardization from the Standard Scaler and all of them will receive calibration for better results.
Cross validation is used to assess the performance of the classifiers and the accuracy of their prediction while csv is imported so we can make the submissions file.

Training Data Creation

The first thing we needed to do for this part was create a list of known positive and negative examples that will be used for the training:



```python
#Creates testing data
positive_pairs = list(edge_set)
random.shuffle(positive_pairs)
pairs_num = len(positive_pairs)
negative_pairs = []

#Finds 1000 edges that are known negatives by virtue of not being in the known positives
while pairs_num > 0:
    r1 = random.randint(0,len(abstracts)-1)
    r2 = random.randint(0,len(abstracts)-1)
    if (str(r1), str(r2)) not in edge_set and (str(r2), str(r1)) not in edge_set:
        negative_pairs.append((r1, r2))
        pairs_num -= 1
```



```python
#Mixes positives and negatives together
labeled_data = [(a, b, 1) for a, b in positive_pairs] + [(a, b, 0) for a, b in negative_pairs]
random.shuffle(labeled_data)

x = []
y = []

#Creates lists of mixed positive and negative examples as well as their labels respectively
for a, b, label in labeled_data:
    features = create_feature_matrix(int(a),int(b))
    x.append(features)
    y.append(label)
```

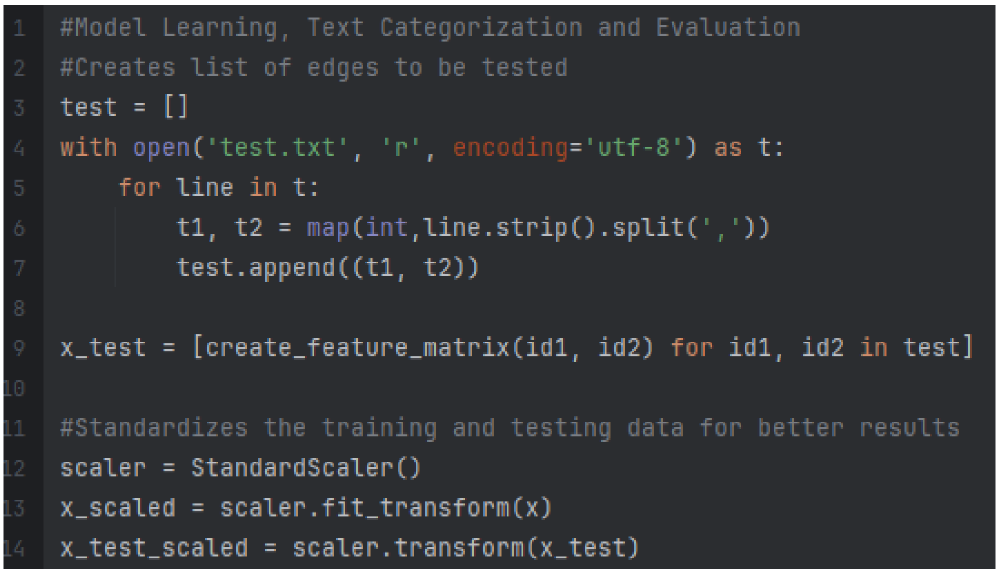
As said above, we have already created a list of 1000 positive and confirmed edges from the edgelist, originally they were 5000 but testing showed that also overtrained our model so it was reduced to 1000. We generate random numbers to find nodes that do not have a confirmed edge connecting them and store them in a list of also 1000 negative examples. Originally we wanted to make this process harder so we were calculating the cosine similarity between the sentence embeddings of those nodes on the spot and if they had a high value then they were considered ‘difficult’, meaning that while they were negative they looked positive and would prevent overtraining by not making the negatives too obvious, however that had the opposite effect and ***did*** overtrain our model. In fact any attempt to try and pick out ‘harder’ negatives resulted in overtraining, so we sufficed with the current approach of just using edges that were not in the list of confirmed positives.

After that we put all the positives and negatives into one list and shuffled it. In that list were included the two nodes alongside the label of their edge, the label was 1 if there was an edge connecting them (they were positive) and 0 if there was no connection (they were negative). We then created empty lists x and y and ran the labeled data through a for loop that computed all of their features, storing them in the list x while it kept their label (aka the answer on whether or not they were connected) in the corresponding position of the y list.

What these lists essentially are is a hypothesis and an answer, the x list has all the features of these nodes and their scores that in total make a hypothesis on whether or not they are connected while the y has the answer on if they really are. By providing our model with this information it can learn from it and proceed to make educated guesses on whether or not other nodes are connected.

Testing Data and Scaling

Before we tackle the classifiers we must first present the testing data:

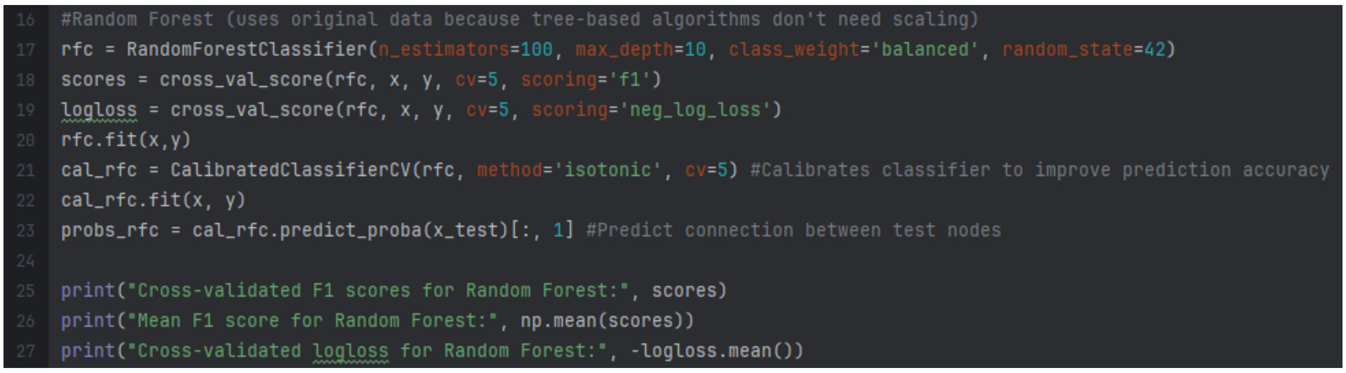


```python
#Model Learning, Text Categorization and Evaluation
#Creates list of edges to be tested
test = []
with open('test.txt', 'r', encoding='utf-8') as t:
    for line in t:
        t1, t2 = map(int,line.strip().split(','))
        test.append((t1, t2))

x_test = [create_feature_matrix(id1, id2) for id1, id2 in test]

#Standardizes the training and testing data for better results
scaler = StandardScaler()
x_scaled = scaler.fit_transform(x)
x_test_scaled = scaler.transform(x_test)
```

First we open the test.txt document and fill out a list of paper duos that we will examine about whether or not they are related, then we find the feature scores for those combinations of papers and store them into a list called x_test which is unlabeled. After our classifiers are trained they will read these feature scores and try to determine if the research papers they correspond to are linked or not.

For select classifiers we implemented the StandardScaler[xxv][xxvi] which standardizes features by changing mean to 0 and standard deviation to 1 which centers the data so it’s not too close to 0 or 1. This improved results by making the model less overconfident and absolute in its assessments. However scaling is not practical for all classifiers, we discovered it by researching ways to improve results but other internet users[xxvii] made it clear that tree-based methods such as Random Forest of XGBoost do not benefit from standardization, so we did not use the scaled data for them.

Random Forest[xxviii]:



```python
#Random Forest (uses original data because tree-based algorithms don't need scaling)
rfc = RandomForestClassifier(n_estimators=100, max_depth=10, class_weight='balanced', random_state=42)
scores = cross_val_score(rfc, x, y, cv=5, scoring='f1')
logloss = cross_val_score(rfc, x, y, cv=5, scoring='neg_log_loss')
rfc.fit(x,y)
cal_rfc = CalibratedClassifierCV(rfc, method='isotonic', cv=5) #Calibrates classifier to improve prediction accuracy
cal_rfc.fit(x, y)
probs_rfc = cal_rfc.predict_proba(x_test)[:, 1] #Predict connection between test nodes

print("Cross-validated F1 scores for Random Forest:", scores)
print("Mean F1 score for Random Forest:", np.mean(scores))
print("Cross-validated logloss for Random Forest:", -logloss.mean())
```

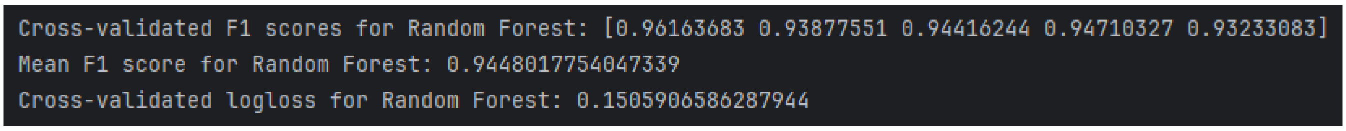
The first classifier we implemented and the one we did most of our tests with was RandomForestClassifier, a tree-based method that did not require scaling. It uses randomized decision trees and averages their result to make prediction and was picked for being a well known and credible classifier. N_estimators is the number of trees in the forest which we set to 100 which is the default value and largely gave us good results with higher or lower numbers not making much of a difference, max_depth is how deep a tree is allowed to be which we limited to 10 because higher numbers caused overtraining, class_weight was labeled as ‘balanced’ for simplicity’s sake and random_state=42 is simply an unwritten rule in data science[xxix].

Next the classifier was put through cross validation to train using the x and y lists we spoke of earlier. How cross validation works is, to avoid overfitting, the data is split into 5 equal parts: in each iteration, the model trains on 4 folds and validates on the remaining fold. This is repeated 5 times, and we compute the F1 score for each run. F1 measures how many examples were positive and how many it got right and scores the model’s performance for all of those batches, with a typically good score being around 0.7-0.9, but higher than that hints there is overfitting at play. We print the mean of the F1 scores and fit it with the training data again because after cross validation is complete sklearn discards the trained model.

After it’s trained, we put the model through a calibrated classifier that smooths out the predictions and makes them less polarized to prevent overconfidence. Originally we used method=’sigmoid’ as it is a well know method, however we later switched to ‘isotonic’ which is less strict and doesn’t require our outputs to be shaped like a logistic curve[xxx], though with too few data it would have its own issues. Calibration was discovered in the search for better results and was applied to all classifiers after proving it was helpful. Calibrated classifier performs cross validation and needs to be fitted again to work.

We then go from training to testing by giving our trained model the testing data and all its feature scores so it can determine the likelihood of one paper citing the other. At the same time as all this and specifically before the model is fitted, we measure the logloss using cross_val_score, meaning we employ the same method as before, however for logloss specifically rather than F1. We then print the F1 scores, their mean and the logloss mean. We avoid printing the logloss on its own for simplicity and we directly print its mean. A typically good logloss would be around 0.05-0.3, but as always if the score is too low then that might hint there is overfitting going on.

This was our output for RandomForestClassifier:

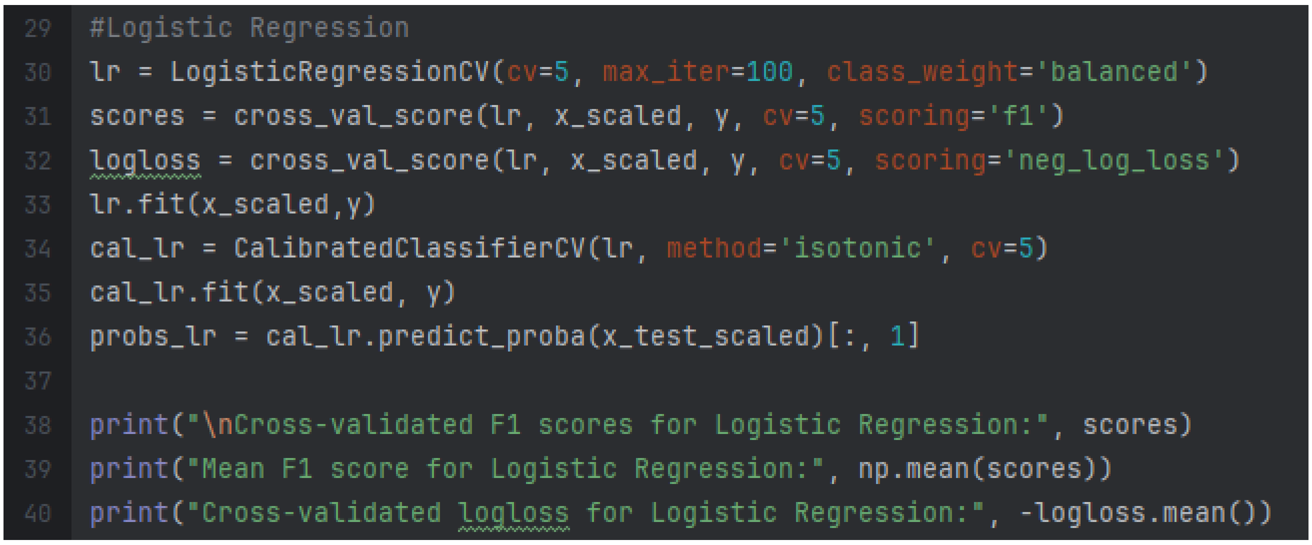


```
Cross-validated F1 scores for Random Forest: [0.96163683 0.93877551 0.94416244 0.94710327 0.93233083]
Mean F1 score for Random Forest: 0.9448017754047339
Cross-validated logloss for Random Forest: 0.1505906586287944
```

As you can see it falls within the preferred range for both the F1 and logloss scores.

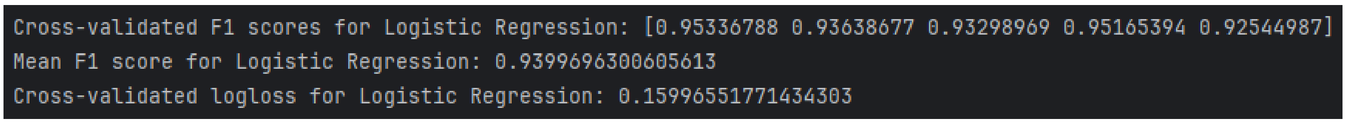
Once we got a satisfactory score we began to implement other classifiers to see how well they would perform. These are…

Logistic Regression[xxxi]:



```
#Logistic Regression
lr = LogisticRegressionCV(cv=5, max_iter=100, class_weight='balanced')
scores = cross_val_score(lr, x_scaled, y, cv=5, scoring='f1')
logloss = cross_val_score(lr, x_scaled, y, cv=5, scoring='neg_log_loss')
lr.fit(x_scaled,y)
cal_lr = CalibratedClassifierCV(lr, method='isotonic', cv=5)
cal_lr.fit(x_scaled, y)
probs_lr = cal_lr.predict_proba(x_test_scaled)[:, 1]

print("\nCross-validated F1 scores for Logistic Regression:", scores)
print("Mean F1 score for Logistic Regression:", np.mean(scores))
print("Cross-validated logloss for Logistic Regression:", -logloss.mean())
```

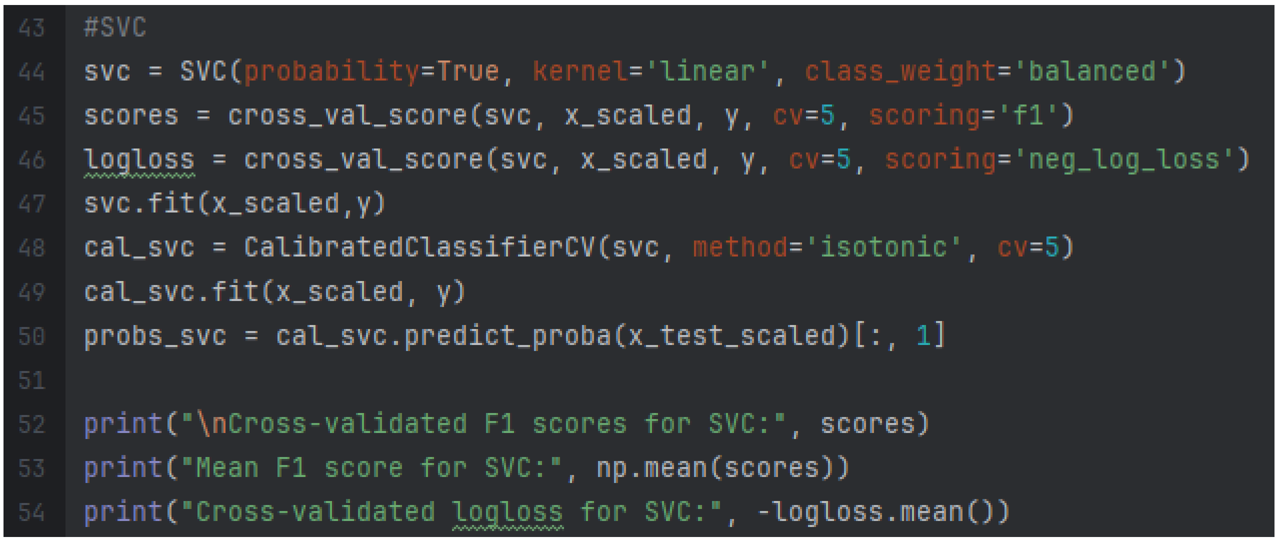
Cv was set to 5 because the classifier uses cross validation, so we configured it to be the same. Max_iter was set as 100 which is the default, tests with 1000 provided no visible difference in score and class_weight was labeled as balanced like Random Forest. We also used the scaled data since this is not a tree-based method. Besides that, the process remains the same and our results are:



```
Cross-validated F1 scores for Logistic Regression: [0.95336788 0.93638677 0.93298969 0.95165394 0.92544987]
Mean F1 score for Logistic Regression: 0.9399696300605613
Cross-validated logloss for Logistic Regression: 0.15996551771434303
```

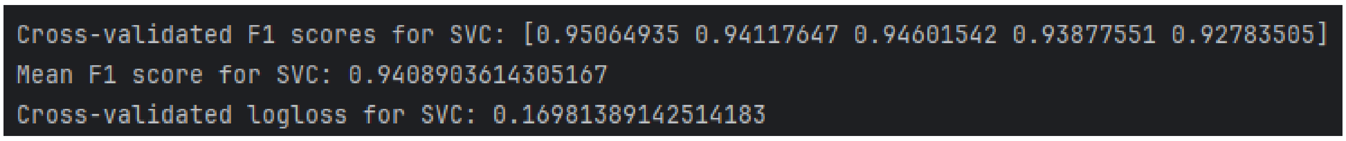
While it also falls within the preferred range both for F1 and logloss scores, its scores are slightly worse.

SVC[xxxii]:



```python
#SVC
svc = SVC(probability=True, kernel='linear', class_weight='balanced')
scores = cross_val_score(svc, x_scaled, y, cv=5, scoring='f1')
logloss = cross_val_score(svc, x_scaled, y, cv=5, scoring='neg_log_loss')
svc.fit(x_scaled,y)
cal_svc = CalibratedClassifierCV(svc, method='isotonic', cv=5)
cal_svc.fit(x_scaled, y)
probs_svc = cal_svc.predict_proba(x_test_scaled)[:, 1]

print("\nCross-validated F1 scores for SVC:", scores)
print("Mean F1 score for SVC:", np.mean(scores))
print("Cross-validated logloss for SVC:", -logloss.mean())
```

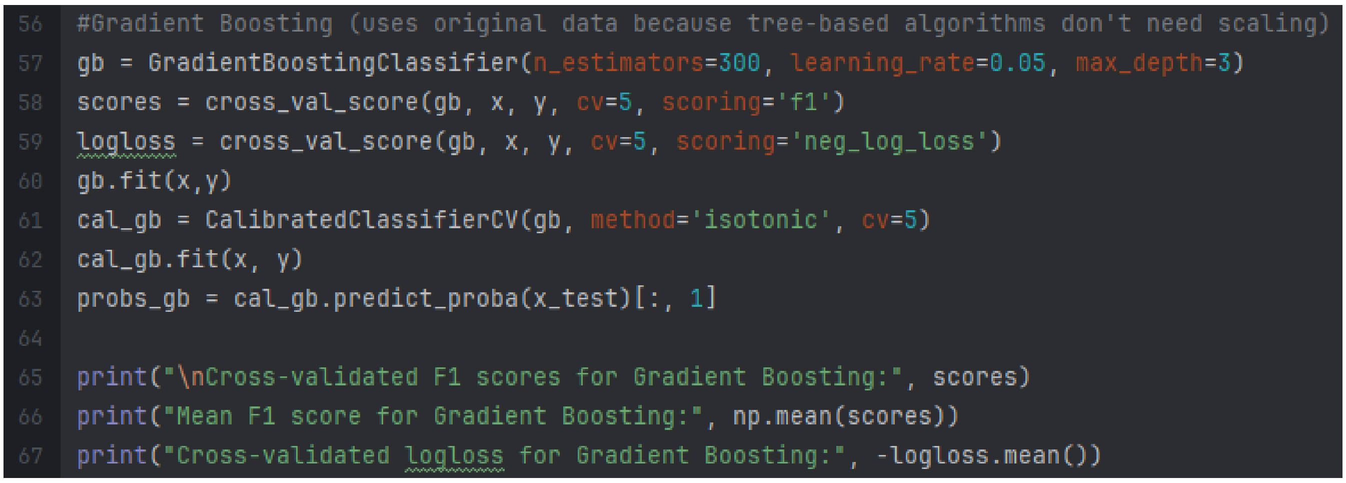
Probability was set to True because we want it to compute probability estimates, kernel was set to 'rbf' as default, but we tested all the alternatives which provided relative or worse scores except for 'linear' which was the best one so we kept that, class_weight was set to balanced like the others. Again, the process remains the same since this is a non-tree-based method. The results are:



```
Cross-validated F1 scores for SVC: [0.95064935 0.94117647 0.94601542 0.93877551 0.92783505]
Mean F1 score for SVC: 0.9408903614305167
Cross-validated logloss for SVC: 0.16981389142514183
```

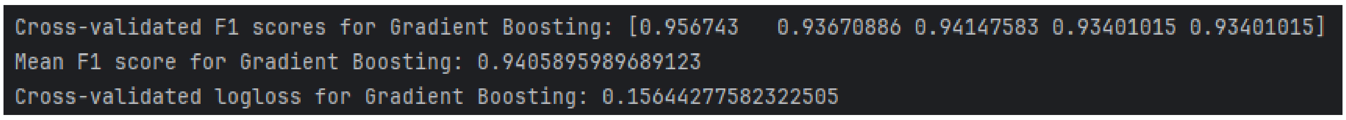
The F1 score is consistent but the logloss is evidently worse, but not bad.

Gradient Boosting[xxxiii]:



```python
#Gradient Boosting (uses original data because tree-based algorithms don't need scaling)
gb = GradientBoostingClassifier(n_estimators=300, learning_rate=0.05, max_depth=3)
scores = cross_val_score(gb, x, y, cv=5, scoring='f1')
logloss = cross_val_score(gb, x, y, cv=5, scoring='neg_log_loss')
gb.fit(x,y)
cal_gb = CalibratedClassifierCV(gb, method='isotonic', cv=5)
cal_gb.fit(x, y)
probs_gb = cal_gb.predict_proba(x_test)[:, 1]

print("\nCross-validated F1 scores for Gradient Boosting:", scores)
print("Mean F1 score for Gradient Boosting:", np.mean(scores))
print("Cross-validated logloss for Gradient Boosting:", -logloss.mean())
```

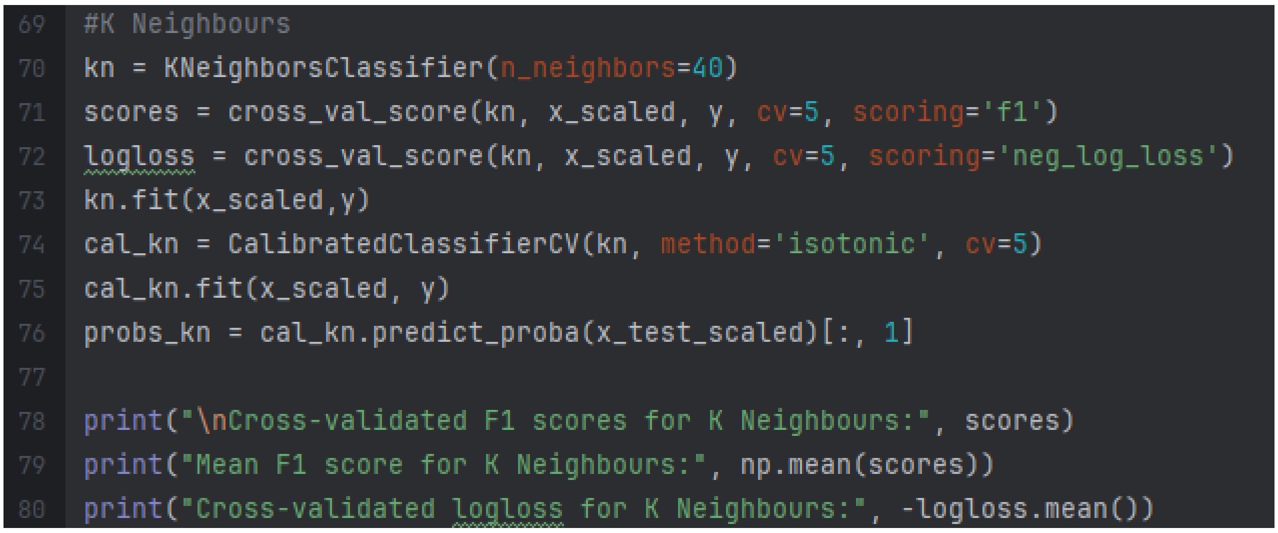
Another tree-based method that doesn't need scaling. N_estimators and learning_rate were set to 100 and 0.1 as the default and we tried different values for both of them. N_estimators=1000 on its own made the score worse, learning_rate=1 on its own made the score worse, both of them together also made the score worse, n_estimators=300 on its own didn't help, but decreasing the learning rate with it did help. Max_depth is 3 by default and higher values made the score worse, but keeping it low while changing the n_estimators and learning_rate ultimately gave us the best scores. The results are:



```
Cross-validated F1 scores for Gradient Boosting: [0.956743   0.93670886 0.94147583 0.93401015 0.93401015]
Mean F1 score for Gradient Boosting: 0.9405895989689123
Cross-validated logloss for Gradient Boosting: 0.15644277582322505
```

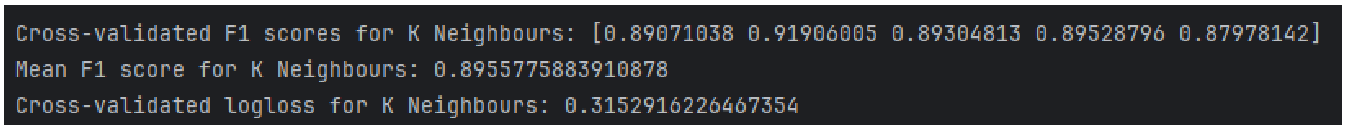
The F1 score and logloss are comparable to Random Forest since they are both tree-based methods.

K Neighbours[xxxiv]:



```python
#K Neighbours
kn = KNeighborsClassifier(n_neighbors=40)
scores = cross_val_score(kn, x_scaled, y, cv=5, scoring='f1')
logloss = cross_val_score(kn, x_scaled, y, cv=5, scoring='neg_log_loss')
kn.fit(x_scaled,y)
cal_kn = CalibratedClassifierCV(kn, method='isotonic', cv=5)
cal_kn.fit(x_scaled, y)
probs_kn = cal_kn.predict_proba(x_test_scaled)[:, 1]

print("\nCross-validated F1 scores for K Neighbours:", scores)
print("Mean F1 score for K Neighbours:", np.mean(scores))
print("Cross-validated logloss for K Neighbours:", -logloss.mean())
```

A non-tree based method so it needed scaling. The n_neighbors was set to 5 as the default originally but it gave out a very poor logloss around 1.2. Gradually increasing it gave us better and better results until it reached its peak around 40 where higher numbers of neighbours would no longer improve it. The results are:



```
Cross-validated F1 scores for K Neighbours: [0.89071038 0.91906005 0.89304813 0.89528796 0.87978142]
Mean F1 score for K Neighbours: 0.8955775883910878
Cross-validated logloss for K Neighbours: 0.3152916226467354
```

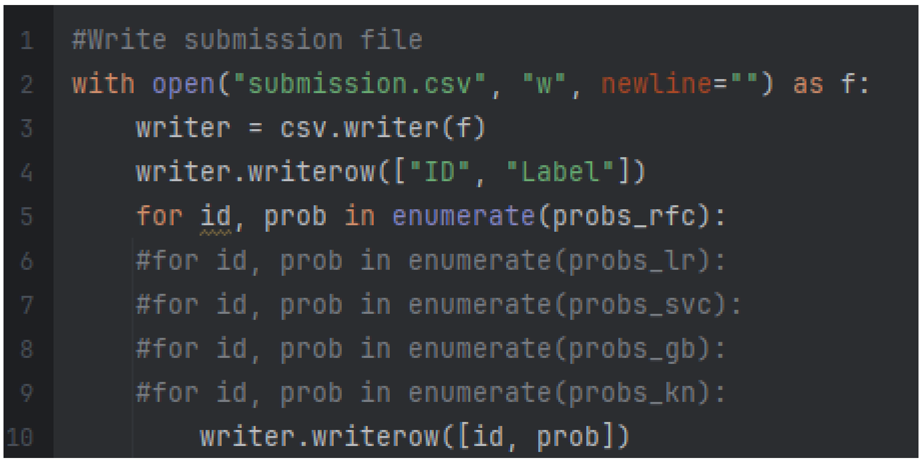
The scores are the worst ones out of them all, but not unacceptable. From our research efforts[xxxv] it seems that this is a common issue with K Neighbours when it comes to logloss and thus is entirely expected.

These classifiers were picked either because they were suggested by the project description or because they were generally recommended popular models found online.

In conclusion, what we can assess from all this is that they generally perform well for the task at hand with K Neighbours simply having some model-specific issues regarding the computation of logloss. They manage to produce respectable results with low waiting time as the most time-consuming part of the process is actually the StandardScaler. The difference between individual models isn't that major so for the most part it can be chalked up to intricacies in our own understanding/development of these methods, but for us the best results were given by Random Forest, hinting that perhaps tree-based methods might work best for this type of project.

Also, another thing we learned is that the default values advised for these classifiers on the scikit-learn website are usually the best option, but for some it is worth investigating alternate values like SVC where linear gave us the best results or Gradient Boosting where trying different combinations showed us a variety of different results.

Submission



```
#Write submission file
with open("submission.csv", "w", newline="") as f:
    writer = csv.writer(f)
    writer.writerow(["ID", "Label"])
    for id, prob in enumerate(probs_rfc):
    #for id, prob in enumerate(probs_lr):
    #for id, prob in enumerate(probs_svc):
    #for id, prob in enumerate(probs_gb):
    #for id, prob in enumerate(probs_kn):
        writer.writerow([id, prob])
```

Now that all of the models have been trained, we open a csv file and include a row with "ID,Label", what this means is that we have two vertical columns, one for the IDs which are specific combinations of research papers that were tasted and the Label which is the model's confidence on whether or not one of the papers cites the other. We write these down and the final csv file has in total 106692 lines. The five for loops correspond to the different classifiers and they can simply be uncommented to override the file with that specific classifier's result. The delivered code has the Random Forest uncommented for no particular reason, feel free to comment and uncomment and try them out.

This concludes our report on the project. Below is a list of websites we cited to understand these different methods we implemented as well as articles that explain different aspects and theories of how they work.

SOURCES

[i] https://huggingface.co/fse/word2vec-google-news-300
[ii] https://huggingface.co/docs/transformers/en/main_classes/tokenizer
[iii] https://www.geeksforgeeks.org/python-word-embedding-using-word2vec/
[iv] https://mayurdhvajsinhjadeja.medium.com/jaccard-similarity-34e2c15fb524
[v] https://igraph.org/r/html/1.2.7/neighbors.html
[vi] https://www.sciencedirect.com/topics/computer-science/degree-centrality
[vii] https://igraph.org/r/doc/degree.html
[viii] https://igraph.org/r/html/1.3.2/page_rank.html
[ix] https://www.positional.com/blog/pagerank
[x] https://scikit-learn.org/stable/modules/generated/sklearn.feature_extraction.text.TfidfVectorizer.html
[xi] https://medium.com/@abhishekjainindore24/tf-idf-in-nlp-term-frequency-inverse-document-frequency-e05b65932f1d
[xii] https://scikit-learn.org/stable/modules/generated/sklearn.decomposition.TruncatedSVD.html
[xiii] https://www.ibm.com/think/topics/latent-dirichlet-allocation
[xiv] https://scikit-learn.org/stable/modules/generated/sklearn.decomposition.LatentDirichletAllocation.html
[xv] https://medium.com/@davidlfliang/intro-getting-started-with-text-embeddings-using-bert-9f8c3b98dee6
[xvi] https://medium.com/@Roy.Wong/step-by-step-guide-how-to-use-bert-word-embeddings-in-python-ac7b621771d8
[xvii] https://www.geeksforgeeks.org/machine-learning/node2vec-algorithm/
[xviii] https://igraph.org/r/doc/transitivity.html
[xix] https://igraph.org/r/html/1.2.4/eigen_centrality.html
[xx] https://igraph.org/r/html/1.3.0/authority_score.html
[xxi] https://igraph.org/r/html/1.3.3/hub_score.html
[xxii] https://igraph.org/r/html/1.3.5/coreness.html
[xxiii] https://igraph.org/r/html/1.3.5/constraint.html
[xxiv] https://igraph.org/python/tutorial/0.9.7/analysis.html
[xxv] https://scikit-learn.org/stable/modules/generated/sklearn.preprocessing.StandardScaler.html
[xxvi] https://stackoverflow.com/questions/40758562/can-anyone-explain-me-standardscaler
[xxvii] https://www.quora.com/Which-machine-learning-algorithms-dont-require-feature-scaling
[xxviii] https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.RandomForestClassifier.html
[xxix] https://grsahagian.medium.com/what-is-random-state-42-d803402ee76b
[xxx] https://scikit-learn.org/stable/modules/generated/sklearn.calibration.CalibratedClassifierCV.html
[xxxi] https://scikit-learn.org/stable/modules/generated/sklearn.linear_model.LogisticRegressionCV.html
[xxxii] https://scikit-learn.org/stable/modules/generated/sklearn.svm.SVC.html
[xxxiii] https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.GradientBoostingClassifier.html
[xxxiv] https://scikit-learn.org/stable/modules/generated/sklearn.neighbors.KNeighborsClassifier.html
[xxxv] https://medium.com/@bengikoseoglu/why-log-loss-metric-shouldnt-be-used-to-evaluate-nearest-neighbour-classification-1fe314f460a2