Sheffield Hallam University

College of Business, Technology and Engineering

Department of Computing
Project (Technical Computing)
[55-604708]
2022/23

Author:	**Thomas Hodgkins**
Student ID:	**30009574 / c0009574**
Year Submitted:	**2023**
Supervisor:	**Syafiq, Zolkepy**
Second Marker:	**Abayomi, Otebolaku**
Degree Course:	**Computer Science**
Title of Project:	**Developing a deliverable chrome extension that gives easy access to research papers relevant to an input text**

Confidentially Required?

YES / NO **NO**

I give permission to make my project report, video and deliverable accessible to staff and students on the Project (Technical Computing) module at Sheffield Hallam University.

YES / NO **YES**

Introduction and Aims

Imagine yourself scrolling through social media. You come across a post stating that the world is flat. This post has millions of views and hundreds of thousands of likes with positive comments supporting the message. You are about to scroll away when you realise the post has stated a fact without any references to back up the claim. This puzzles you and makes you think for a while leading you to go down a research rabbit hole into the correct shape of the earth. You come across a youtube video stating that the earth was round. Luckily this video has citations allowing you to easily confirm that the original flat earth post was wrong and spreading mis-information.

Mis-information is a prevalent issue in today's society especially on social media. This is especially the case within the health sector as this survey shows (Suarez-Lledo & Alvarez-Galvez, 2019). This survey reviewed 69 studies relating to mis-information about a number of health topics on social media sites. It concluded that the lowest rate of mis-information was in the topic of medical treatments with 30% being labelled as mis-informarmative.

It is easy to say mis-information is harmful but harder to define and enforce its absence, especially online. This can be shown in the recent push of the online safety bill (UK Government, 25 C.E.), which tries to quantify mis-information to allow it to be punished. This has been met with backlash (Trueman, 2022) with a point being that it infringes on freedom of speech. (Chin, 2022) goes into more depth about the challenges faced by this bill such as blocking a person's freedom of expression and how hard it would be to regulate different social media sites. All this shows that this topic is hotly debated in parliament and solutions are needed.

Instead of mis-information being punished, factual information should be more accessible. This would allow people to easily fact check posts and articles online as well as find out more information about certain topics that they come across.

This project aims to allow users to more effectively reach potentially truthful but more importantly factual information in a fluid manner without intruding on the users current experience. This will be achieved by allowing the user to highlight a given piece of text and scan it through a chrome extension returning related papers about the highlighted text using AI transformers and text classification. This will hopefully push people to check what they are reading especially when it has limited to zero sources referenced.

Research

This section takes a dive into methods used to classify text into topics and compare the similarity of two blocks of text. These two methods are potential engineering solutions to solve the problem of semantically defining text in a way it can be used to compare to other pieces of text. These techniques are critical for this project as a user's inputted text needs to be semantically compared accurately to provide relevant information.

Topic Modelling

Topic modelling is an area of research aimed at creating algorithms and techniques to semantically define text into topics. This area of research has been around for decades using Bayesian methods such as latent dirichlet allocation proposed by (Blei et al., 2003). These models used probabilistic methods to define topics based on the words contained in a corpus (set of documents).

Specifically the LDA equation has two arguments. The first being a dirichlet distribution of documents in a space in which each axis is a topic. The second being a word space with each word being an axis with the topics placed within that space to show how words are weighted between topics. These arguments are then refined to try to get the best probability of reproducing the original documents. These refined arguments can then be used on a new document to place it into the space mentioned above and assign a probability of being a certain topic.

The downfall of this method is that it does not take into account the context of words or semantic meaning in any way, instead mapping all words present in the documents individually (excluding words cleaned out such as stop words). This method also requires the number of topics to be defined beforehand leading to large amounts of manual effort to tune the right amount of topics as well as cleaning the data to remove stop words.

To combat the downfalls of LDA new methods of topic modelling started to arise using the encoder section of the transformer model (Grootendorst, 2022) to harness the power of embeddings created by the encoder. This would allow a topic model to see the context between words and use that to define topics in a more semantic manner.

Embeddings

Embeddings are a way to translate text into numbers/vectors so a machine learning model can more effectively understand what is being inputted into it. This can be achieved in a variety of ways, some of which will be explained below.

One-Hot Encoding

One hot embedding creates a vector equal to the size of the vocabulary in your given data. When a word is input to be converted into a vector a 1 is placed in the index that matches the word being input. This allows the model to understand which word is being input as each word will have a unique vector. The downfall of this method is that it creates sparse vectors scaling with the size of the vocabulary. This is inefficient and also provides no context about the word within the embedding.

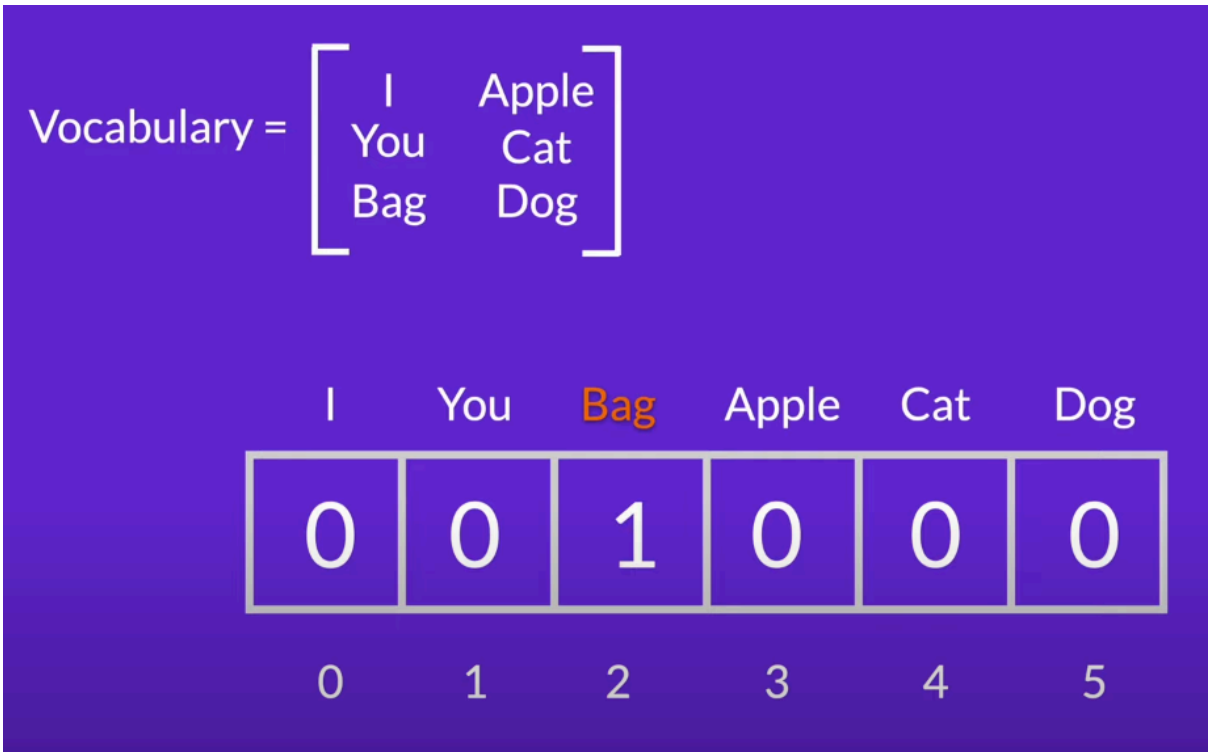

Figure 1.

Bag-Of-Words

In this approach a sentence is input into the bag of words encoders. Each word in the vocabulary is given an index in the vector and assigned to 0. Then each time the word occurs in the sentence the number associated with that word is incremented by 1. This is a useful approach especially when trying to find common words in certain topics but falls short as no context is given to the words being processed. Also as with the previous method sparse vectors are created using this approach leading to inefficiencies.

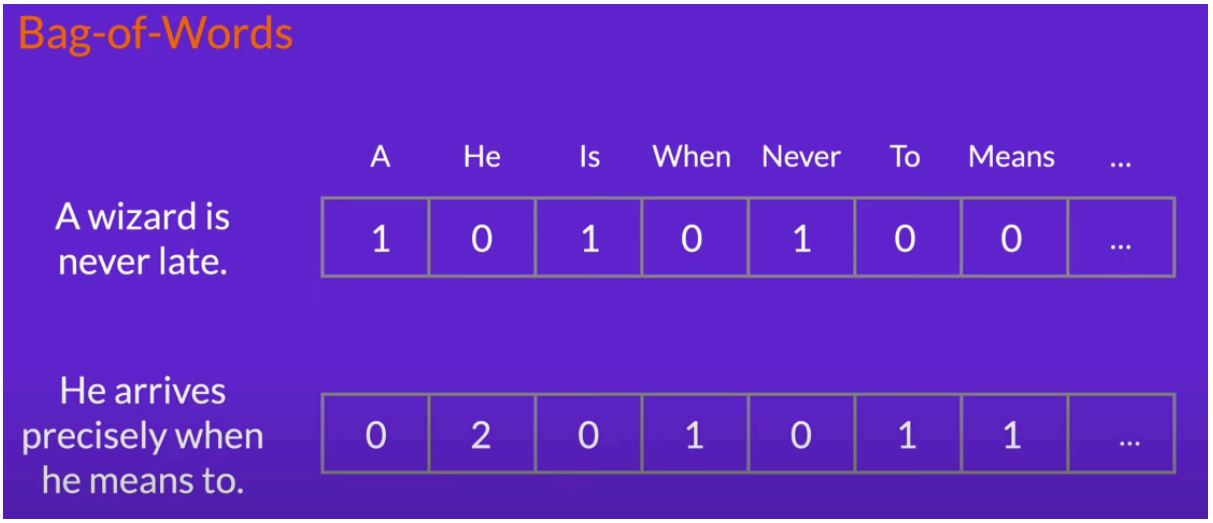

Figure 2.

Word Embeddings

There are many ways to produce word embeddings such as word2vec, BERT and SBERT to name a few. I will be focusing on SBERT (Reimers & Gurevych, 2019) as it is a sentence transformer. This architecture is useful for semantic similarity searching where two embedded vectors can be compared to gauge how similar they are.

SBERT is a model founded to combat the issue that BERT has with semantic textual similarity. This issue arises because both sentences have to be fed into the network creating massive computation overhead . This overhead causes BERT to take 65 hours and around 50 million computations to find the most similar sentence within a dataset consisting of 10,000 sentences. The SBERT model was able to reduce this to 5 seconds without reducing accuracy making it suitable for semantic textual similarity searches.

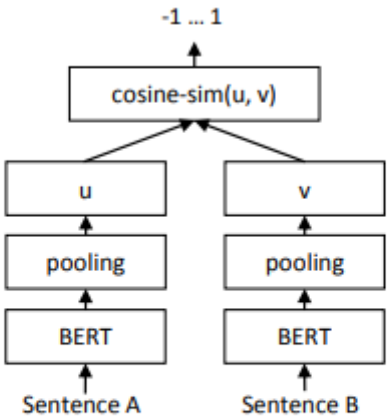

Figure 2: SBERT architecture at inference, for example, to compute similarity scores. This architecture is also used with the regression objective function.

Figure 3.

The model consists of two identical BERT models (Siamese models) with tied weights. The models are fed differing sentences producing differing vectors for each sentence. Pooling is applied to the output of the BERT models to ensure the vectors are of the same dimensional size. These two vectors can then be compared with a cosine similarity function to find out how similar the original sentences are.

MiniLM Models

MiniLM models are distilled versions of large transformer models. Knowledge distillation in machine learning is an efficient technique to produce smaller models using bigger models as teachers while maintaining the bigger models performance. This is done by training the smaller model (student) in conjunction with the larger model (teacher) in a way that allows the student model to see and learn to mimic the outputs of the teacher model. This results in a minor loss in performance while significantly reducing model size. Using this method (Wang et al., 2020) the Mini LM model was able to outperform other 66 million parameter models and its average accuracy rating was 1.1% below the base BERT model which consisted of 109 million parameters. This shows the power of knowledge distillation when applied to large transformer models. Also as this is a encoder only transformer can be downsteamed to word embeddings tasks like semantic textual similarity tasks.

Topic Modelling With Neural Topic Models

BERTopic (Grootendorst, 2022) is a neural topic model (NTM's). This topic model takes advantage of the BERT-based encoders to produce embedding vectors in order to more accurately classify topics. BERTopic is a modular system allowing you to choose different algorithms for each step in the topic modelling process. This means you can tune the algorithm stack to fit your specific task. On top of this NTM defines topics dynamically as well as negates the need for stop words to be removed (as they can be important for understanding context). This allows you to insert a large corpus with minimal effort and analyse the topics within it.

The downfall of this method is it does not fit well for data retrieval. It works well for data analysis being able to visualise the topics of a large corpus, however when inserting a new document and wanting to find a neighbouring document with a similar semantic value a state of persistence is needed to quickly compare and fetch these related documents.

Vector Databases

Vector databases solve the persistence problem of BERTopic while still utilising the power of vector embeddings. This type of database enables you to create embedding vectors for text and store those vectors in a space where they can all be query easily. This is suitable for my problem as I will be able to insert research papers into the database and query them using an input vector as a query. This input vector can be created by processing the query text through the same embedding model the database contents went through and then finding similar vectors within the database returning the closest ones.

Hierarchical Navigable Small Worlds (HNSW)

HNSW is the main vector searching algorithm used by vector database services like Weaviate and Pinecone. This search algorithm combines the probability skip list with the navigable small worlds data structure to make an efficient nearest neighbour search algorithm.

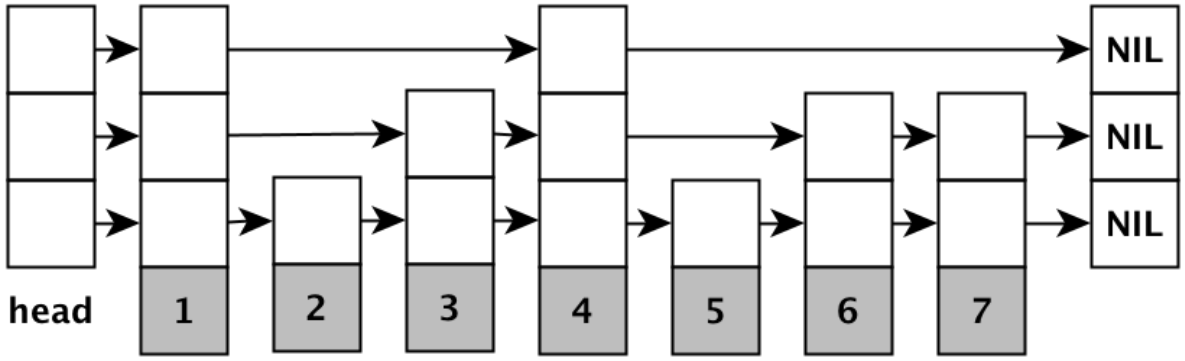

Figure 4.

Probability skip lists were suggested by (Pugh, 1990) as a probabilistic alternative to binary trees. Figure 4 shows the structure of a probability skip list. This structure is constructed using probability. Firstly, a linked list is initialised with the stored elements. Each element is then given a 50/50 chance to be added to the next layer. This 50/50 chance is recursively called adding a layer to that element each time until the max layers are reached or the coin flip returns false. Each of these layers can be thought of as its own separate link list with the elements within it chaining to the other linked lists around it. Due to the nature of probability the higher the layer the fewer the elements present in that layer. This enables you to search from the top layer allowing you to skip past elements that you would otherwise have to iterate through when using a standard linked list. This can be shown if we were to search for the number 6 in figure 4. We start at the highest layer's head which points to element 1. This is smaller than 6 so we move to the next element in the highest layer 4. As this element is also smaller we carry on along the linked list to the end. As we reached the end without finding element 6 we backtrack to element 4 moving down a layer. Now on layer 2 we move across to find element 6. This ability to skip over elements using probability reduces search time complexity from O(n) for linked lists to O(log n).

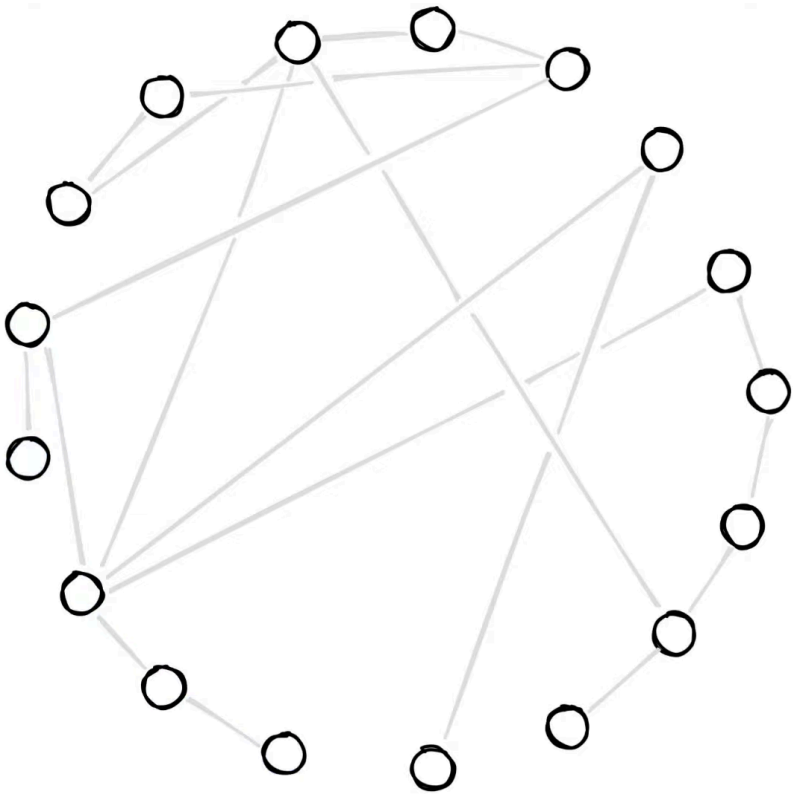

Figure 5.

The other ingredient of HNSW is the navigable small world model (Kleinberg, 2000) as shown in figure 5. This model consists of a set of nodes assigned vertices connected to other nodes. These connections are defined by a variable of randomness leading to some long vertices and some short vertices. A node with a small number of connections is called a low-degree vertex and a node with lots of connections is called a high-degree vertex. These high degree vertices can be used as highways to move to different parts of the vector space quickly while low degree vertices can be used to move smaller distances to close in on a specific point in the space.

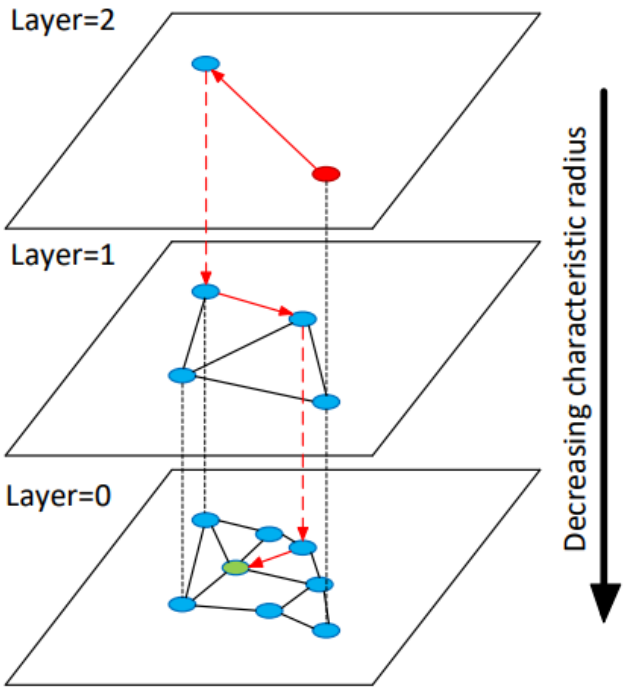

Figure 6.

Figure 6 is the image taken from the original paper that came up with HNSW (Malkov & Yashunin, 2020). The image clearly shows how navigable small world models and probability skip lists have been integrated together to create an efficient structure for nearest neighbour searching. Each layer is a new navigable small world model consisting of sparser and spaser connections between nodes and fewer nodes when moving up the layers. This allows for efficient navigation across a vector space using higher layers with the ability to drop down through layers to gain increasingly more precise movements around the vector to close in on a nearest neighbour depending on the queried vector. This structure allowed HNSW to surprise top PQ algorithms such as Faiss. Although the query time is fast, construction time for these data structures is long with a 1 billion sized dataset taking 5.6 hours to construct.

Similar Applications

Google Scholar

Google Scholar is a search engine for research papers. It allows you to input a search term and view related papers based on the search term. It can take in long passages and return related results unlike other research paper search engines allowing users who do not know about a subject to paste in found passages and find related results easily. However, for some search criteria multiple searches are required fine tuning the search term each time until the desired results are found. This is a problem especially if the person researching is unfamiliar with the research area.

Connected Papers

Connected papers (Connected Papers) is a search engine specifically for research papers. It allows a user to search, similar to Google scholar, and find related research papers based on the search terms. Once a paper is found it can be selected leading to a graph displaying the selected paper with a number of papers surrounding and linking back to it. This graph is shown in figure 7. These papers are related to the original paper in certain ways. They use the concept of Co-citation and Bibliography Coupling to find similar papers. This concept presumes that papers with overlapping citations and references will cover a similar topic and are therefore semantically similar placing the papers on the graph with a linking line based on the strength of the similarity score.

This product is extremely useful when starting research. I was able to use it to easily find all papers relating to topic modelling. The graph is user friendly and the paper's pdfs are easily accessible with two clicks, one on the paper and one on the pdf icon. However, the terms still need to be searched for by the user meaning some knowledge of the subject even if it's just a brief title is needed to find relevant papers. Also similar to Google search a certain method is needed to search efficiently with the product meaning a piece of text from a passage cannot be easily inputted and result in related papers.

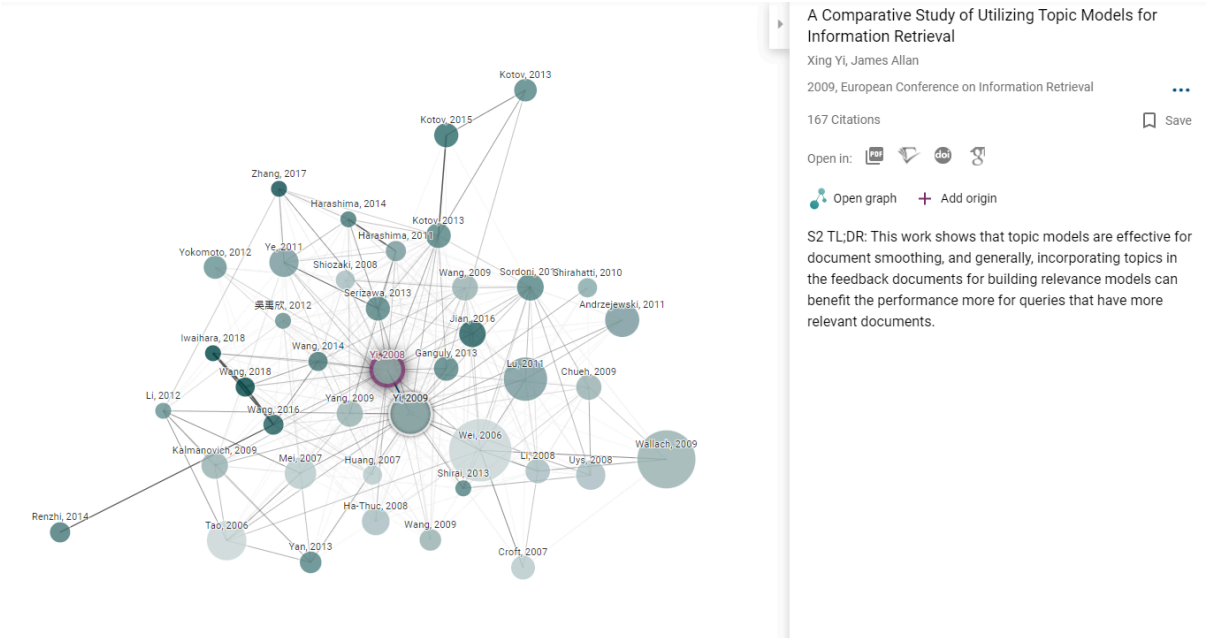

Figure 7.

Planning and Design

At first when planning this project I was set on using the BERTopic model. My plan was to fine-tune this model on the Arkvix research paper dataset. This would enable me to input new passages and classify them into a category and fine papers from this category. With initial testing and planning I realised that the model would be too broad as one category could contain up to 150,000 papers. This would mean that the results would have to be sifted through added code or users would get unrelated results. This problem led me to research vector databases allowing me to find a path forward using them instead of an end to end topic model like BERTopic. The use cases for these databases aligned well with my project's needs so I moved forward with the plan of using a vector database to store papers from the Arkvix dataset.

Hardware Limitations

Training embedding models is time consuming and resource intensive as shown by the most downloaded sentence encoder on hugging face, MiniLM-L6-v2. This encoder is a fine-tuned version of MiniLM-L6-H384-uncased using 1,170,060,424 tuple pairs as its dataset on top of the pre-trained model. This was done using 7 TPU's and efficiency intervention from a Google cloud team member. This is clearly not replicable with the resources I have on hand and the time frame given. For this reason I will be taking advantage of the multi-qa-MiniLM-L6-cos-v1 (Reimers & Gurevych, 2020) model which was fine tuned for similarity semantic search.

Database Platform

There are many vector database platforms available such as Pinecone, Weaviate and Milvus. All offer certain advantages and disadvantages. I will be choosing Weaviate as it is open source and offers easy instructions for use in a local environment with Docker. This local environment is needed as the scale of the data I am adding is too large for most free tier cloud services to be a viable option. This is shown as Weaviate's free tier cloud storage has a max storage capacity of 10,000 objects.

Dataset

At the beginning of the research process I tried to web scrape useful paper from the web to use as a dataset for my project. I did this because I thought it would offer a wide range of papers from different sites. This did not go as planned as web scraping some sites became increasingly difficult due to dynamic page elements. Due to these issues I decided to research a pre-existing dataset to use and found the arXiv dataset (arXiv, 2023) consisting of 2.7 million research papers converting a variety of research topics. This dataset also had categories labels for each paper making it suitable for my use case.

Libraries

Chrome API

The main library being used within the chrome extension is the built-in chrome api. This is accessible on content and background scripts which can be defined in the manifest file of the extension. This api allows the extension to interact with many chrome tools such as storage, tab information and bookmark utilities.

Weaviate-ts-client

Weaviate has a typescript library to allow connecting and querying on a Weaviate database. Although this is typescript based I will be using it within a javascript node js server as I find javascript easier to use for less complex servers like this one. The javascript library for Weaviate was deprecated meaning certain features and support are not available within it leading to this decision.

Dask

Dask extends and adds functionality to many libraries including the well known python library Pandas. It has many use cases with one being able to fit your dataset into memory by utilising disk space instead of RAM only. This is essential for my project as I found early on that loading the 2.7 million papers with vanilla Pandas lead to memory overload even with 20 Gb's of RAM. Dask allowed me to easily load in the dataset and manipulate it without risking the code breaking due to RAM being full.

System Context Diagram

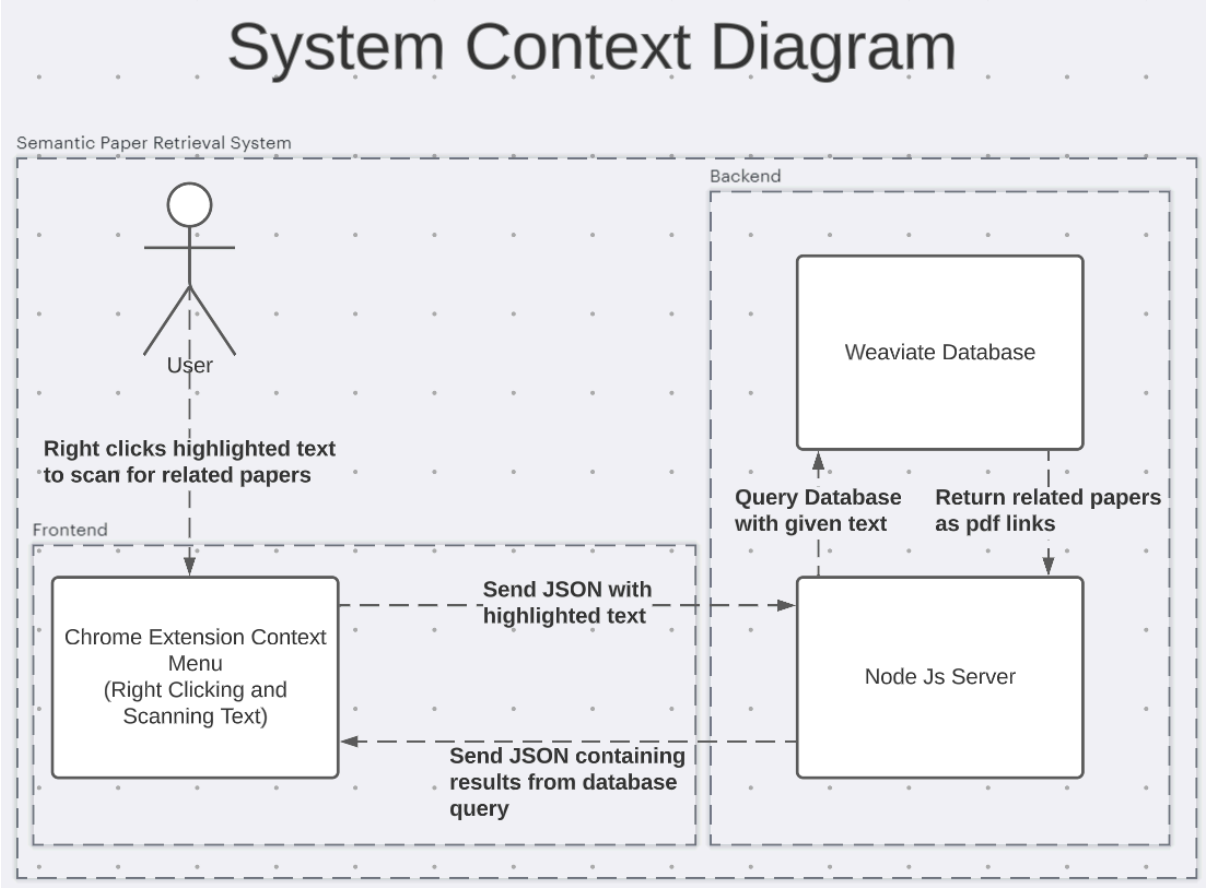

Figure 8.

Figure 8 shows the system context design for the deliverable. The user can interact with the chrome extension which will communicate with the Node js server. This server communicates with the database returning query results which will then return to the chrome extension front end to display results to the user.

Sequence Diagram

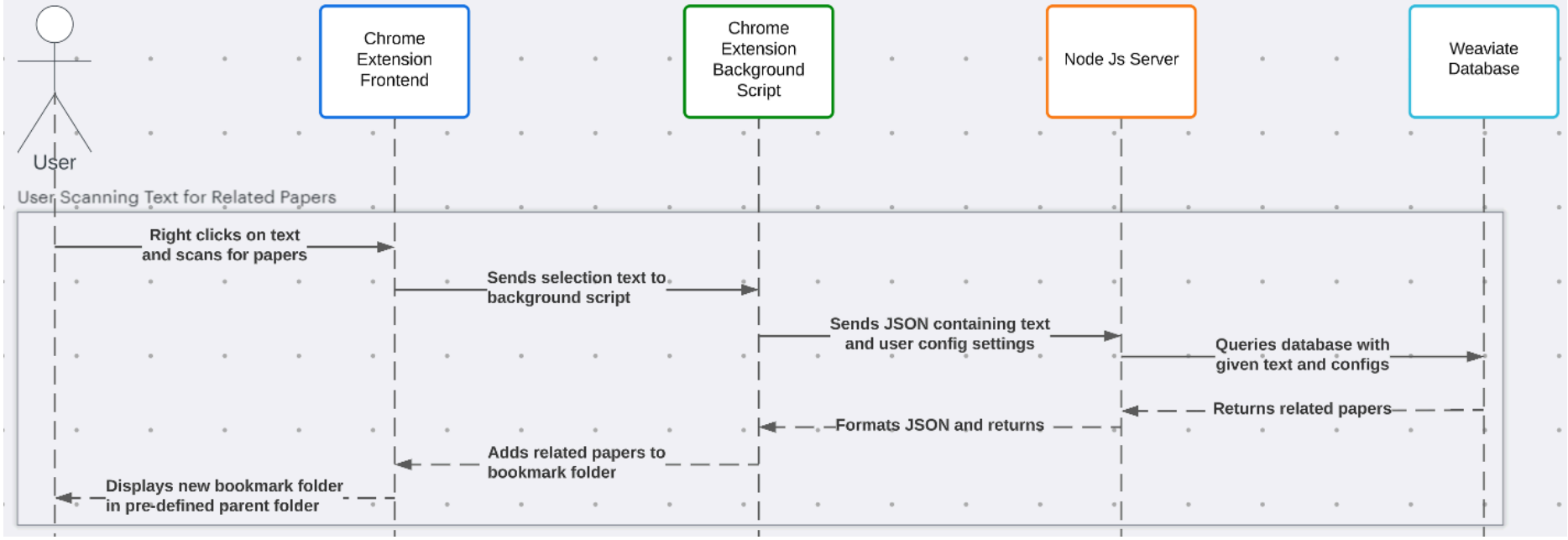

Figure 9. shows the sequence of events for a user scanning a section of text and the chain of events to eventually query the database and return related papers to the user.

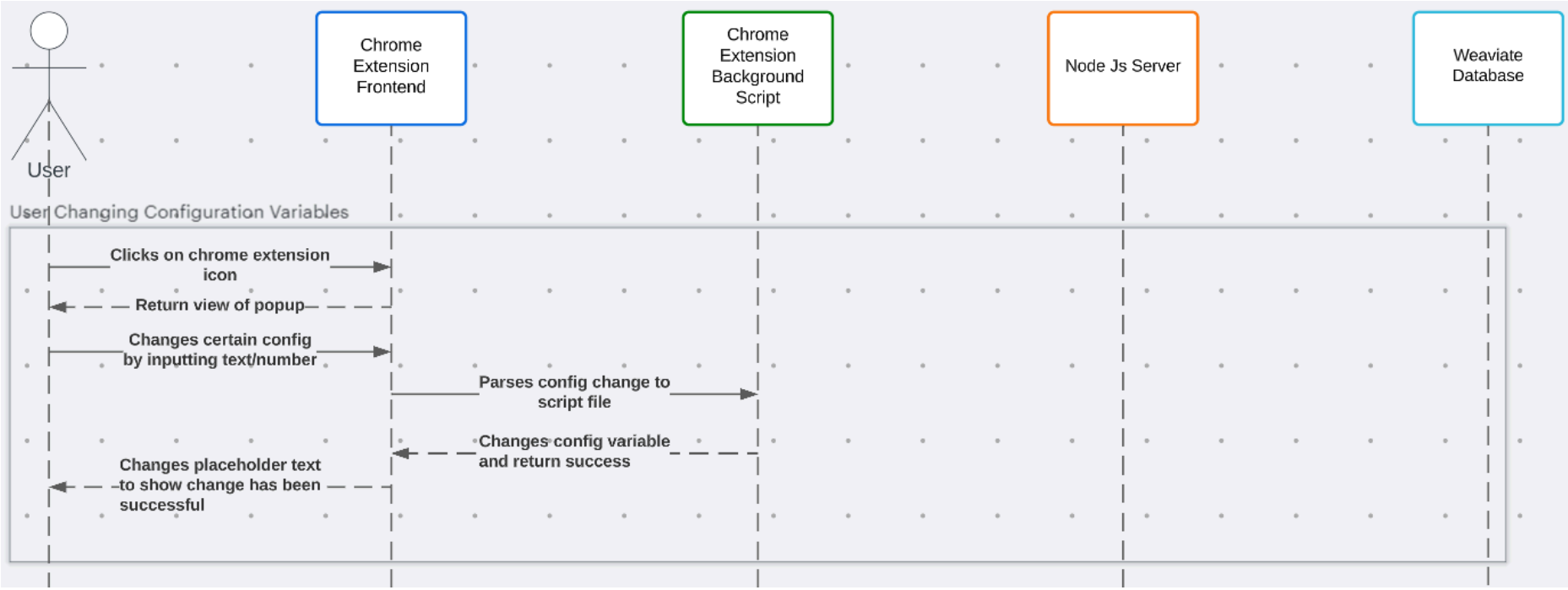

Figure 10. Shows the sequence of events for a user changing configuration variables using the chrome popup

UI Design

I designed a rough arrangement of the UI (figure 11) that would be present in the chrome extension using Figma. This design will allow me to easily map hex values for colours and provide a template of structure when implementing the deliverable's UI in the development phase.

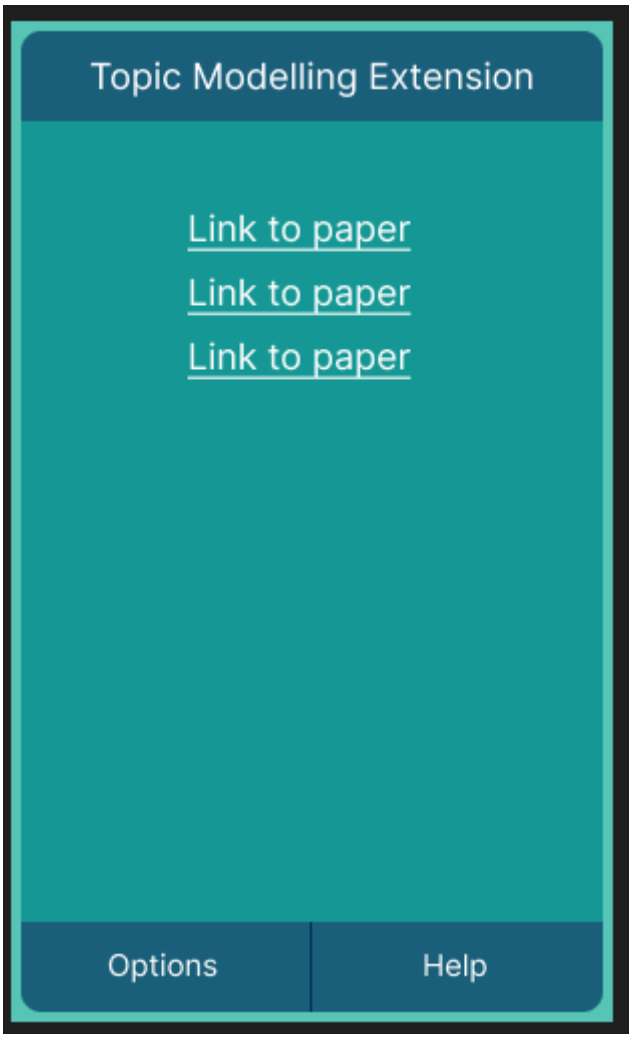

Figure 11.

Software Engineering Approaches

Development Approach

I planned to use agile development during this project. I did not initially design the structure of the first implementation attempt I had as I did not have the knowledge about the technologies I was using to accurately portray what the end result would look like. My second approach was more structured with a system designed to match the structure of the project as well as dynamic designs showing how the user would interact with the system. This could be argued as agile development as I planned and implemented my first prototype then evaluated that it needed to be changed as it had many problems. Once these problems had solutions I then implemented my second prototype being much more successful. These could be seen as two cycles of agile development.

Version Control

For version control I will be using GitHub. It is easy to use and comes integrated in popular software applications. One of these applications is visual studio code. This is the IDE I will be using throughout this project as it has a large community of developers which produce useful extensions. One of these extensions implements all of github's version control functionality allowing you to manage your commits and pull from within the IDE. This extension will be used as the main interaction with my repository.

Library Management

Conda

I will be using MiniConda for my python library management. This software allows for Conda environments to be created which can independently install libraries and then be activated and deactivated when needed. This makes python library management easy. One added benefit is if library conflicts happen separate environments can be made and run on separate python scripts to easily avoid conflict issues. On top of this an environment can be assigned a python version leading to easy changes in python versions if needed. MiniConda will be used with pip to install all libraries.

Node Package Manager

Like all great JavaScript based web projects I will be utilising NPM as my package manager. This will be used to easily install some of the millions of libraries available on NPM as well as easily manage libraries installed within a project. This package manager makes managing libraries easy and as all libraries get installed into a single node_modules folder it makes git ignore file configuration easy. Also sharing NPM projects with other people is stress free as they just have to type npm i into the terminal when pointing at the projects directory to install all related libraries. The package json file also allows for condensed cli commands allowing you to easily reduce a script run command to commands like "npm run start". This makes project workflows easier to understand and document.

Testing Method

Testing my deliverable is tough as it is based on the semantic similarity between an input text and resulting texts. I will be extrinsically testing the false positive rate of the deliverable by inputting a decided text and determining whether the results are related to the input text. This should yield a good understanding of the rate of false positives present when using my deliverable.

Deliverable

Initial Plan

The original approach for this deliverable was to fine-tune a BERTopic model using the arxiv dataset. This would allow that model to classify input documents as a topic and I could then search for similar topics on the web using web scrapers or within the dataset. Although this sounded like a solid approach I soon found that BERTopic did not define topics specifically enough for precise searching and as a lot of papers in the same area used similar vocabularies papers were not separated enough in semantically to allow for precise similarity textual searching to occur. This method also seemed too slow for a deliverable product as BERTopic would often take seconds to define a document in a topic. Due to these shortcomings I decided to research a different approach to solving the main problem. This failed implementation arised due to tunnel vision when researching. I found topic modelling and instantly accepted it as the solution putting all my research hours into it. However, I slowly realised topic modelling is used as more of a data analysis tool especially BERTopic meaning speed and precision were not apparent enough to apply to the final project.

Although this section of my development was scrapped due to it being unsuitable I managed to learn a significant amount about chrome extension development. One of the big takeaways from this development period was the lack of ability to run native scripts on a chrome extension. I tried using the chrome api's native messaging protocol to be able to run a python bat file containing code to run the BERTopic model and return results, however, I could not get this working due to errors I could not fix even with days of debugging. I also tried using the node js function child_process as this allows you to run scripts with a specific python interpreter. However, this was also unsuccessful as child_process is a server side function that can't be run on a chrome extension background script as it is browser side. This led me down the path of potentially scraping the idea of a chrome extension all together and just using a native python application with PyQt5 being used for the GUI. While I made some progress with this Idea I soon came across a Youtube video (Soft, 2022) showcasing web sockets being used with chrome extensions to allow for server communication. This was a potential solution that led me down the path of using a Node Js server with sockets to communicate and run scripts on a local backend server. Around the same time I discovered vector databases and their use cases. Seeing this I did more research surrounding the area and found I could easily run a server and vector database locally. This combined with web sockets solved all barriers met during the development of the original chrome extension and BERTopic model integration. The development section will deep dive specifically into the

development of the chrome extension using a node js server and a vector database with a BERT style word embedding model.

Development

The first act in development was to design and implement the chrome extension front end. As I had partially developed an extension trying to implement the previous solution I was confident in my ability to quickly develop the extension. On top of this confidence I discovered a chrome extension template repository on github which utilised webpack 5 (Salunkhe, 2023). This was useful and it set out the boiler plate code for a chrome extension and allowed me to focus on more important features instead of integrating webpack 5 (which I had struggled with in the previous implementation).

Building the chrome extension was made easier by my choice to use bare HTML, JavaScript and CSS instead of a bulky JavaScript framework such as Vue or React. I believe the frameworks are useful especially when building complex websites, however as I did not plan to have many dynamic features present on my chrome extension's popup I decided against using a framework to reduce complexity and boiler plate code. On top of this JavaScript frameworks are specifically designed for web pages not chrome extensions so certain features such as routers are not compatible or need a significant work around to become functional. All these factors resulted in my choice to use HTML, CSS and JavaScript.

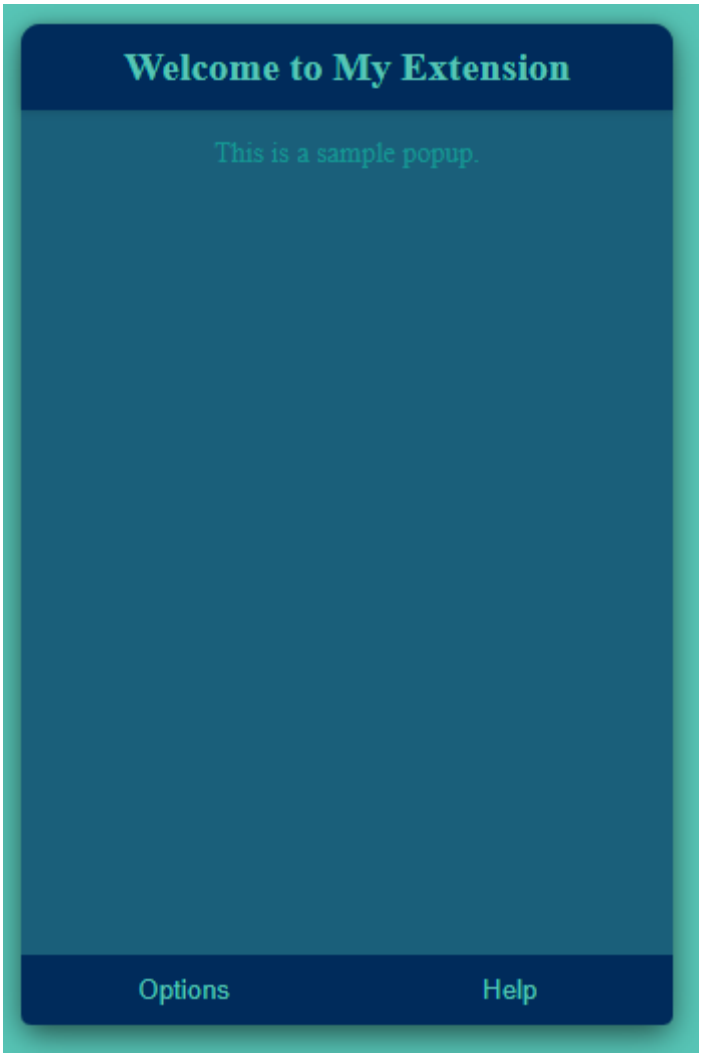

Figure 12.

As seen in figure 12 I was able to implement the design created in Figma. The two buttons on the bottom only had test functionality outputting the cursor position in the console on click.

Next I wanted to configure the Weaviate database. This could be done using different platforms such as Kubernetes and Docker. I had not used either before this project. With some research I realised Docker was more fit for my purpose as it was smaller and provided all the functionality needed for the database to work. The main configuration I chose for my docker file was the sentence transformer to use to convert text to a vector embedding. I decided to use the default option multi-qa-MiniLM-L6-cos-v1. I did research into the possible options and this option seemed the most suited to my application. It was fine-tuned for semantic similarity tasks and used a cosine similarity function to compare vectors semantic similarity.

After I copied the file it was time to initialise the docker images and run the docker containers. I ran docker compose up in the terminal building the images for the database and sentence transformer being used. This was a time consuming process as the sentence transformer was 4GB's in size and took around 10 minutes to completely download and initialise. After this finished I was able to query the database using Weaviates python library.

```
client = weaviate.Client(
    url="http://localhost:8080"
)
```

```
print(client.schema.get())
```

```
{'classes': [{'class': 'Paper', 'description': 'Articles', 'invertedIndexConfig': {
```

Figure 13.

I wrote this test code in a .ipynb file as these Jupyter notebook style files work well for writing modular code that can be run in blocks separately. As seen in figure 13 I was able to connect to the locally running database via port 8080. The figure also shows schema outlined in the database called Paper. This was defined by me using the code below.

```
client.schema.create_class({
    "class": "Paper",
    "description": "Articles",  # description of the class
    "properties": [
        {
            "dataType": ["string"],
            "description": "The id",
            "name": "pdfId"
        },
        {
            "dataType": ["text"],
            "description": "The title of the paper",
            "name": "title"
        },
        {
            "dataType": ["text"],
            "description": "The abstract",
            "name": "abstract"
        },
        {
            "dataType": ["string[]"],
            "description": "The categories",
            "name": "categories"
        },
        {
            "dataType": ["int"],
            "description": "The year of the most recent version of the paper",
            "name": "version"
        },
    ],
    "vectorizer": "text2vec-tranformers"
})
```

Figure 14.

Figure 14 shows the code used to initialise the schema class for the papers I would insert into the database. I only included essential information such as the id of the papers pdf as well as the abstract so that the vectors could use it to produce an appropriate vector. The categories were also added to influence the vectorization as well as the most recent year that the paper was published.

Now that the schema was configured for the data It was time to clean the dataset to allow it to be inserted into the database without issues. I used .ipynb files to clean the data as it allowed for easy debugging when running functions as well as clean modular code that can be run in blocks.

I managed to find code on Kaggle which showed how to effectively clean the arXiv dataset using the dask library and lambda functions (COLINDRES, 2023). I was able to edit this code to fit my purpose allowing me to extract only the fields relevant for the database schema.

```
import dask.bag as db
import pandas as pb
import json
```
[1] ✓ 2.3s

```
loaded_data = db.read_text("PATHTODATASET//arxiv-metadata-oai-snapshot.json").map(json.loads)
```
[2] ✓ 0.0s

```
print("The dataset has {} entires.".format(loaded_data.count().compute()))
print("Data Structure:")
loaded_data.take(1)
```
[3] ✓ 30.7s

```
The dataset has 2223467 entires.
Data Structure:

({'id': '0704.0001',
  'submitter': 'Pavel Nadolsky',
  'authors': "C. Bal\\'azs, E. L. Berger, P. M. Nadolsky, C.-P. Yuan",
  'title': 'Calculation of prompt diphoton production cross sections at Tevatron and\n  LHC energies',
  'comments': '37 pages, 15 figures; published version',
  'journal-ref': 'Phys.Rev.D76:013009,2007',
  'doi': '10.1103/PhysRevD.76.013009',
  'report-no': 'ANL-HEP-PR-07-12',
  'categories': 'hep-ph',
  'license': None,
  'abstract': '  A fully differential calculation in perturbative quantum chromodynamics is\npresented f
  'versions': [{'version': 'v1', 'created': 'Mon, 2 Apr 2007 19:18:42 GMT'},
   {'version': 'v2', 'created': 'Tue, 24 Jul 2007 20:10:27 GMT'}],
  'update_date': '2008-11-26',
  'authors_parsed': [['Balázs', 'C.', ''],
   ['Berger', 'E. L.', ''],
   ['Nadolsky', 'P. M.', ''],
   ['Yuan', 'C. -P.', '']]},)
```

Figure 15.

```
trim_attributes = lambda x: {'pdfId': x['id'],
                             'title': x['title'].replace("\n", " "),
                             'categories': x['categories'].split(" "),
                             'abstract': x['abstract'].replace("\n", " "),
                             'version': int(x['versions'][-1]["created"].split(" ")[3]),
                            }
```
1] ✓ 0.0s

```
trimmed_loaded_data = (loaded_data.map(trim_attributes).compute())
```
]

Figure 16.

As seen in figure 15 many fields in this dataset are not needed for my task so these will be removed using the code in figure 16. This code returns each object with the specific fields required by the database schema. Both the abstract and title fields contain “\n” characters to represent new lines, however these will not be needed as they apply no context to the text and could confuse the model when constructing vector embeddings. The latest version field

is extracted and the categories field is converted into an array of values instead of the original string of words separated by spaces. The array conversion of the category field was decided on as I felt a string of words with spaces e.g “Category1 category2” would confuse the model more than having array the values separated in an array.

```
with open("ArxivTrimmedDataset.json", "w") as f:
    for i in range(0, len(trimmed_loaded_data)):
        json.dump(trimmed_loaded_data[i], f)
        if(i < (len(trimmed_loaded_data)-1)):
            f.write('\n')
```

Figure 17.

After the data was processed I dumped the newly formed objects into a new json file as shown in figure 17. New line characters had to be added on the end of objects to enable json file readers to read objects line by line.

```
def loadFromJson():
    dataset = pandas.read_json("ArxivTrimmedDataset.json", lines=True)
    dataset = dataset.reset_index()

    for index, row in dataset.iterrows():
        returnRow = row.to_dict()
        del returnRow["index"]
        returnRow["pdfId"] = str(returnRow["pdfId"]).replace(".", "-")
        yield returnRow
```

Figure 18.

Now that the trimmed data is in a json file it is time to insert the data into the Weaviate database. Figure 18 shows the function that will load the data from the json using Pandas. The rows are iterated through removing an index field which is added by pandas and changing the “.” in the pdf id field to a “-”. This change occurred as the first few times I tried to insert objects without this Weaviates insert function automatically converted the string value to a float and then threw a type mismatch exception as it was now no longer compatible with the Paper schema. This change seemed the most simple as I reverse the change easily when returning the pdf ids later on.

```python
def check_batch_result(results: dict):
  # Check batch results for errors.

  if results is not None:
    for result in results:
      if "result" in result and "errors" in result["result"]:
        if "error" in result["result"]["errors"]:
          print(result["result"])
```

```python
client.batch.configure(
  batch_size=1000,
  dynamic=True,
  num_workers=1,
  callback=check_batch_result
)

with client.batch as batch:
    for obj in loadFromJson():
        print(obj["pdfId"])
        batch.add_data_object(
            obj,
            class_name="Paper"
        )
```

Figure 19.

```
# Batch Efficiency Testing

batch-size num-workers time-to-complete 10000 inserts

5000        1           1m 51.1s
5000        2           1m 50.5s

1000        1           1m 48.7s
1000        2           1m 48.3s

500         1           1m 50.9s
500         2           1m 51.4s

10          1           1m 51s
10          2           1m 53s
```

Figure 20.

Figure 19 shows the code for inserting the data loaded from pandas into the database. It uses the batch inserting feature that Weaviate supports. As you can see multiple configurations can be made to the batch function including the numbers of workers and

batch size. I tried to change these values and record their speed to see if I could speed up inserting but value changes seemed to make little difference as shown in figure 20. For this reason I stuck with 1 worker and a batch size of 1000 as it seemed the fastest in the test cases.

Now with the batch configurations figured out I started inserting into the database. The first full try of this was run overnight as after 4 hours I realised it would take a long time to complete. I checked my pc the next day to find it had automatically reset in some way causing the code to interrupt. This was frustrating as it had been working well for the 6 hours I was awake with it running. Due to this issue I created a new json with the first 100,000 objects from the trimmed json. I inserted this json instead so that I could have a functioning database to query from so that other functionalities could be completed with the goal of uploading all data at some point. This worked well and took around 2 hours to complete.

```
    volumes:
      - dbData:/var/lib/weaviate
  t2v-transformers:
    image: semitechnologies/transfo
    environment:
      ENABLE_CUDA: '1'
      NVIDIA_VISIBLE_DEVICES: 'all'
    deploy:
      resources:
        reservations:
          devices:
          - capabilities:
            - 'gpu'
volumes:
  dbData:
...
```

Figure 21.

After getting the database functional I closed and deleted the docker container as well as the images as they consumed around 8 GB's of ram. Once I generated the images again using docker compose, the database was gone. It was my first time using docker and I didn't realise that volumes needed to be mounted onto your containers to store persistent data. I found that Weaviate has amazing documentation on how to implement this into your docker compose file easily as seen in figure 21. I added a volume to the Weaviate environment mounting it into a specific file path within the containers directory as Weaviate's documentation specified. This was tested and data was now persistent within the container.

```python
from websocket_server import WebsocketServer
import json
import weaviate
import asyncio

class Websocket_Server():

    def __init__(self, host, port):
        self.server = WebsocketServer(host, port)
        self.weaviateClient = weaviate.Client(url="http://localhost:8080")

    def new_client(self, client, server):
        print("New client({}) connected.".format(client["id"]))

    def client_left(self, client, server):
        print("Client({}) disconnected.".format(client["id"]))

    def message_received(self, client, server, message):
        print("Client({}) Message: {}".format(client["id"], message))

    def run(self):
        self.server.set_fn_new_client(self.new_client)
        self.server.set_fn_client_left(self.client_left)
        self.server.set_fn_message_received(self.message_received)
        self.server.run_forever()

ws_server = Websocket_Server("localhost" , 3000)
ws_server.run()
```

Figure 22.

Next I set up the server which for testing purposes was set up in python as seen in figure 22. This code was taken from this Youtube video and modified to connect to the Weaviate database. After verifying the server could connect to the database the client side of the connection on the chrome extension needed setting up.

```javascript
let connection;

chrome.runtime.onInstalled.addListener(() => {
  console.log("background.js installed!")

  // Connect to backend server to fetch db data
  connection = new WebSocket("ws://localhost:3000/")

  // Add event listener to run when messages are received from the server
  connection.addEventListener("message", (event) => {

    // Parse incoming message into object
    let data = JSON.parse(event.data)
    console.log(data.response)
  }
```

Figure 23.

The code in the figure 23 shows the client side code running on the background script on the chrome extension. The connection is defined as a global variable so that it can be accessed anywhere in the file scope. The connection then is initialised as a web socket connecting to port 3000. An event listener for the message event which will run when a message is received by the client. For now this message is parsed as a JSON into an object and then logged in the console for testing purposes.

After trying to implement more server side functionality such as adding new functions to the server and formatting the return objects correctly as a json I realised that I didn't like using python for the server side code. Due to this personal preference I decided to switch the server side code to JavaScript using Node Js. This was more familiar to me and meant that all functional code relating to the chrome extension and server was written in JavaScript.

```
const express = require('express');
const app = express();
const server = require('http').createServer(app);
const WebSocket = require('ws');
const weaviate = require('weaviate-ts-client');

const wss = new WebSocket.Server({ server:server });

const databaseClient = weaviate.client({
    scheme: 'http',
    host: 'localhost:8080',
});

wss.on('connection', (ws) => {
    console.log('A new client Connected!');

    ws.on('message', async (message) => {

        var data = JSON.parse(message);
        console.log(data);

    });
});

app.get('/', (req, res) => res.send('Hello World!'));

server.listen(3000, () => console.log(`Lisening on port :3000`));
```

Figure 24.

Figure 24 shows the converted python server rewritten in JavaScript with Express and Node js. I thought this code was more readable than the python server making it easy to understand for myself and people who might view my code.

```
async function queryDatabase(queryBody) {
    // Query database with given arguments

    // Creates an object in the correct format to query near text with Weaviate
    nearTextArgs = {
        "concepts": [queryBody]
    }

    // Calls graphql query returning only pdfId and title fields
    result = await databaseClient.graphql.get().withClassName("Paper").withFields("pdfId title").withNearText(nearTextArgs).withLimit(5).do();

    // removes unnecessary object layers
    result = result.data.Get.Paper;

    // Logs cleaned resulting objects from database query
    for (let i = 0; i < result.length; i++) {
        console.log(result[i]);
    }

    return result;
}
```

Figure 25.

Next I added a database query function to the server (figure 25) to allow it to query data using an argument as the query body. This function returned the pdf id and title of the 5 results. I realised when logging the results to the console that the pdf ids that had been converted before to avoid type mismatch issues needed to be converted back.

```
async function convertPdfLink(pdfId){

    resultPdfLink = pdfId;

    // Checks if pdfId includes / and replaces - with . if false
    // This is needed as the valid pdf url requires a . instead of a -
    if (!pdfId.includes("/")) {
        resultPdfLink = pdfId.replace("-", ".");
    }

    // Add surrounding url
    resultPdfLink = "https://arxiv.org/pdf/" + resultPdfLink + ".pdf";

    return resultPdfLink;
}
```

Figure 26.

Figure 26 converts the input pdf to a valid pdf link that can be used to get to the papers pdf. The if statement was used as some pdf id's did not include a "." originally but did include a "-". This was a mistake made by me when inserting the documents as now I have to filter out the pdf id's that contain "-" but did not originally contain ".". Luckily, this was easy as all links with an original "-" contained a "/" character which the original "." links did not. This allowed me to easily change only the pdf ids that needed to be changed. This could have been easily avoided by properly evaluating the dataset's objects to find a potential unique character to replace "." instead of assuming a "-" would do the job.

```
wss.on('connection', (ws) => {
    console.log('A new client Connected!');

    ws.on('message', async (message) => {
        // On client message parse message to object and query database with arguments parsed

        var data = JSON.parse(message);
        console.log(data);

        responseBody = await queryDatabase(data.body, data.amount);
        response = JSON.stringify({response: responseBody});

        // Send response to client socket with returned query results
        ws.send(response);
    });
});
```

Figure 27.

Figure 27 shows the updated message event. Once the message is decoded into an object two arguments are parsed from it "body" and "amount" into the query database function to be used as query parameters. I added an await on the query database function to make sure the result would be returned before the socket responds to the client.

Next was to write the client side code responsible for sending messages to the server and receiving responses back.

```
// Create right click context menu for text selection
chrome.contextMenus.create({
    id: "scan",
    title: "Scan for related papers",
    type: "normal",
    contexts: ["selection"]
})
```

Figure 28.

```
chrome.contextMenus.onClicked.addListener(function(info, tab){
  // When the custom context menu button is pressed take selected text and send to server with amount of papers variable

  console.log("Selected Text: ", info.selectionText)
  connection.send(JSON.stringify({body: info.selectionText, amount: amountOfPapers}))
});
```

Figure 29.

Firstly, using the chrome api and documentation I added a context menu when the background script is installed as seen in figure 28. This adds a button press to the menu that appears when you highlight text and right click on it. I also added a function for when this button is pressed as seen in figure 29. This sends a json formatted string to the server containing the amount of papers to return and the text highlighted by the user.

Next I added a function that received the server's response. During the development process I was hesitant on how to format the returning links. Adding them to the chrome extension popup was relatively easy for one scan. Just show the returning links and that's it. However, this meant that users trying to look back at results from older scans would not be able to. A massive part of research and links in general is the ability to save them and go back to them when you need to. This would be quite tough and time consuming to implement using bare CSS and HTML. One method I thought of was to have a scrolling box full of sessions. Each session would be one scan containing the resulting links from that scan. Although this was a suitable solution I was not confident in my HTML/CSS ability and thought the end result would look too messy and most likely result in a scroll box with lots of scans being saved meaning the user would lose track of which session contained which results. Time was also a concern with this method as I didn't want to spend too much time on the front end to run out of time when implementing other more important features. The second solution that I ended up implementing consisted of using chrome bookmarks. The chrome api gives the ability to add/rename and delete bookmark folders as well as bookmarked links. My idea was to create a root folder for all sessions to be stored in on the bookmarks bar and store a new folder in this root folder for each session. Each of these session folders would contain the resulting links from that session. I felt this solution would be more compatible for most user experiences as they would most likely be used to using the bookmark bar to save important links they need to use. On top of this, bookmark folders and links can be easily renamed by the user after the session folder has been created allowing for users to easily keep track of sessions through personal naming conventions. This also freed up space on the extension's popup as it could now be used for other functionality.

```
chrome.bookmarks.getTree(function(bookmarkTree){

    console.log(bookmarkTree[0].children[0])

    let bookmarkBar = bookmarkTree[0].children[0]

    let folderFound = false
    for (let child in bookmarkBar.children) {
        if (child.title == "Papers Found - Chrome Extension") {
            folderFound = true
        }
    }

    if(!folderFound) {
    chrome.bookmarks.create(
        {'parentId': bookmarkBar.id, 'title': "Papers Found - Chrome Extension"},
        function(folder) {
            console.log("New folder created on bookmarks bar: " + folder.title)
        }
    )
    }
})
```

Figure 30.

Figure 30 shows the beginning of my bookmark implementation. This function uses the chrome api to access the bookmark tree. This is a directory tree containing all bookmark folders and links within that folder. The bookmark bar portion of the tree is saved. The bookmark bar is then checked for folders with the title “Papers Found - Chrome Extension”. This was my temporary root folder name for all sessions to be saved in. This function ensured that if this folder was not present it would be created so there would be a place for sessions to be stored.

```
connection.addEventListener("message", (event) => {
    console.log(event.data)
    let data = JSON.parse(event.data)
    console.log(data.response)

    chrome.bookmarks.getChildren(workingFolderId, function(result) {
        console.log("Paper Folder Children:")
        console.log(result)
        sessionID = result.length

        let title = "Session " + sessionID

        chrome.bookmarks.create({'parentId': workingFolderId, 'title': title}, function(folder) {
            console.log("Created new folder for session called: " + folder.title)

            for (let i = 0; i < data.response.length; i++) {
                chrome.bookmarks.create({
                    'parentId': folder.id,
                    'title': data.response[i].title,
                    'url': data.response[i].pdfId
                })
            }
          }
        )

        sessionID++
    })
})
```

Figure 31.

Figure 31 shows the next feature implemented on the client side. This added session creation functionality when the socket received a message from the server. This message was parsed to an object then used the globally assigned folder id, which was equal to the folder id of the root folder, to create a session folder inside the root folder and add links to that folder. These links would be titled according to the returned title from the database and contain the url received from the database. The sessionID variable is a global variable that keeps track of how many session folders are created. This sessionID is assigned to the session folders title so session folders are not titled the same. This is the main functionality added and worked well with a paragraph on LDA returning 5 related research papers about LDA.

After this I wanted to use the space available on the chrome extension's popup. I also wanted the user to be able to personalise the name of the root folder that is automatically created if the user does not already have a folder with that exact name. At the moment if you rename the root folder it will stay but the next time chrome is opened the extension will create a brand new folder. As chrome extension bookmarks have unique ids the extension will not break when scanning text after the root folder is renamed as session folders are created within it based on its id not its name. I also wanted to add the ability for the user to choose the amount of papers returned by the database.

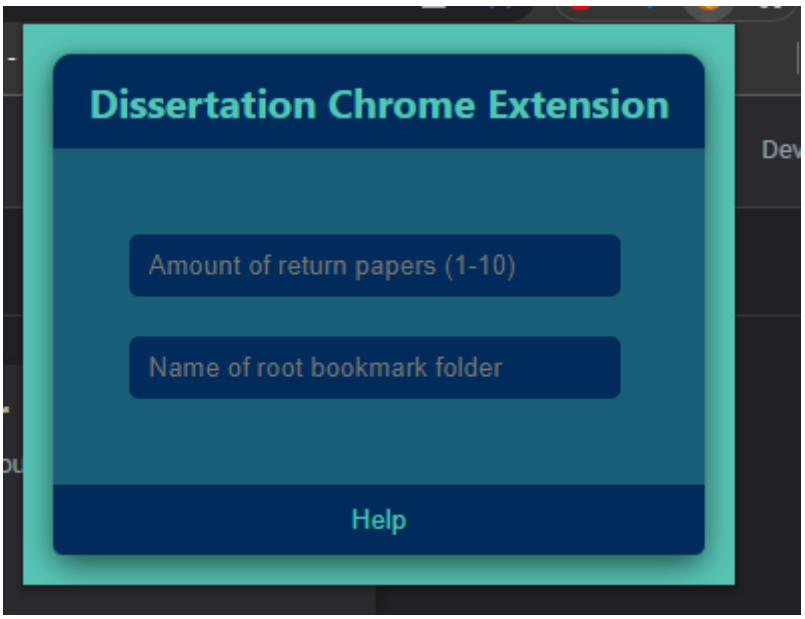

Figure 32.

Figure 32 shows the new design of the chrome extension with the front end of the functionality for renaming the root folder and changing the amount of returning papers. I chose to limit the value to 10 as more than 10 seemed to bloat the session folder and return papers further away from the original input text. With this change I also removed the options button from the bottom bar of the extension as the chrome extension popup had become the options menu.

```
numberInput.addEventListener('change', async (event) => {
    // When number input is changed (amount of papers config) send input value to background.js

    console.log('input for return paper amount: ', event.target.value);

    const response = await chrome.runtime.sendMessage({head: "amount", value: event.target.value});

    // Wait for response and display appropriate placeholder text depending on if errored or successful
    console.log(response);
    if(response.result){
        console.log("success")
        numberInput.placeholder = "successfully changed amount";
    } else {
        console.log("Error")
        numberInput.placeholder = "Error changing amount, try again";
    }
});
```

Figure 33.

```
textInput.addEventListener('change', async (event) => {
    // When text input is changed (folder rename config) send inputed text to background.js

    console.log('input for folder rename: ', event.target.value);

    const response = await chrome.runtime.sendMessage({head: "folderRename", value: event.target.value});

    // Wait for response and display appropriate placeholder text depending on if errored or successful
    console.log(response);
    if(response.result){
        console.log("success")
        textInput.placeholder = "successfully changed folder name";
    } else {
        console.log("Error")
        textInput.placeholder = "Error renaming folder, try again";
    }
});
```

Figure 34.

Next up was implementing the functionality for these new configurations. I first added two event functions to trigger on each of the text boxes change events. This event is called when an input is within the box and the user removes their focus from that box (clicks elsewhere on the popup). Figure 33 shows the functionality for the amount box. This function takes in the data and sends it to the background script through the chrome api's messaging library. The head of the argument defines what the data is meant to be used for so the server knows what to do with the incoming arguments. The result is awaited for and then the placeholder text is changed to show if the change was successful or unsuccessful. The same function was implemented for the folder rename text box as shown by figure 34.

```
chrome.runtime.onMessage.addListener( (request, sender, sendResponse) => {
  // Recieves messages from popup.js decodes command and execute with given arguments

  console.log("Received: ", request);

  switch (request.head){
    case "amount":
      // If command is amount make sure the value is between 0 - 10

      if(request.value > 10){
        request.value = 10
      }
      if(request < 0){
        request.value = 1
      }

      // Set the parsed config value to storage and if succesful assign global variable to it and return success
      chrome.storage.sync.set({amount: request.value}).then(() => {
        console.log("New amount " + request.value + " saved to storage")
        amountOfPapers = request.value
        sendResponse({result: true})
      })

      break;
```

Figure 35.

The background script file was then changed to implement a message listener function that receives the messages from the popup file. Figure 35 shows the message listener function as well as the method for figuring out what to do with the arguments parsed. The head is

used as an argument in a switch statement to figure out if the intended parsed data is meant for changing the amount of returned papers or renaming the root folder.

Figure 35 shows the case for the head argument “amount”. At this point in implementing this feature I realised through constantly reloading the extension to refresh changes to code that there was no persistence for these user configuration variables. This means the code has no way of knowing the user’s configuration settings when starting the extension. To solve this issue I used the chrome api’s storage library allowing me to set and get from the user’s chrome browser local storage. The sync part of the function means the storage is available on any browser where the user is signed in so different devices will have the same configuration values as long as the same user is signed in. This storage implementation is shown in figure 35 as the storage key “amount” is set to the value sent as an argument. When this function is complete the global amount variable is changed and the success response is sent back to the popup.

```
    case "folderRename":
      // If command is folder rename update the folder to new title with argument pa

      chrome.bookmarks.update(workingFolderId, {title: request.value}).then(() => {
        workingFolderName = request.value
        console.log("folder renamed to " + request.value)

        chrome.storage.sync.set({title: request.value}).then((e) => {
          console.log("New title " + request.value + " saved to storage")
          sendResponse({result: true})
        })
      })

      break;
    default:
      // Fell into unknown command return false
      console.log("fell into default, Unknown command")
      sendResponse({result: false})
  }

  // This return true statement tell the listener on popup.js to keep listening for
  return true;
});
```

Figure 36.

Figure 36 shows the implementation of the “folderRename” case. This function updates the folder title and assigns the folder title variable to that new folder name. The function then runs similar code to the amount function setting the title variable in storage instead. The default case is also shown returning a false response to show the function failed to understand the head argument parsed by the front end.

Figure 36 shows a return true statement at the end of the function even though all cases return to the popup with the send response function. Following chrome api’s documentation on the message listener function this return statement is needed to keep the channel on the

other end of the connection (the popup) open until a send response function is received. If this return statement was not present the popup would throw an error about no response being received from the server before the server can send a response.

```
// Tries to get saved variables for folder title and paper amount
chrome.storage.sync.get(["amount", "title"]).then((e) => {
  console.log("amount variable from storage: " + e.amount)
  console.log("title variable from storage: " + e.title)

  // If title is not found assign default name otherwise assign stored name
  if(e.title != undefined){
    workingFolderName = e.title
  } else {
    workingFolderName = "Papers Found - Chrome Extension"
  }

  // If amount is not found assign default amount otherwise assign stored amount
  if(e.amount != undefined){
    amountOfPapers = e.amount
  } else {
    amountOfPapers = 5
  }
```

Figure 37.

The next logical step is to implement a get function to get the saved variables from storage when chrome extension starts. Figure 37 shows the get function. It is within the on installed function meaning it will be run when the chrome extension is installed onto chrome (everytime it is opened). This function gets the two values from storage and assigns them to the respective global variables as long as they are not undefined. If the values in storage are undefined the variables are defined by default values that I set.

```
async function queryDatabase(queryBody, limitAmount) {
    // Query database with given arguments

    // Creates an object in the correct format to query near text with Weaviate
    nearTextArgs = {
        "concepts": [queryBody]
    }

    // Calls graphql query returning only pdfId and title fields
    result = await databaseClient.graphql.get().withClassName("Paper")
    .withFields("pdfId title").withNearText(nearTextArgs).withLimit(limitAmount).do();

    // removes unnecessary object layers
    result = result.data.Get.Paper;

    // Loops over pdf title and converts them to valid urls
    for (let i = 0; i < result.length; i++) {
        result[i].pdfId = await convertPdfLink(result[i].pdfId);
    }

    // Logs cleaned resulting objects from database query
    for (let i = 0; i < result.length; i++) {
        console.log(result[i]);
    }

    return result;
}
```

Figure 38.

The amount configuration needs to be implemented on the server side code so that affects the actual database query. This was easily done by adding another argument to the query database function and replacing 5 with this argument in the with limit section of the database query. This is shown in figure 38.

```
{
    "dataType": ["text"],
    "description": "The abstract",
    "name": "abstract",
    "moduleConfig": {
        "text2vec-transformers": {
            "skip": False
        }
    },
    "indexInverted": True
},
{
    "dataType": ["string[]"],
    "description": "The categories",
    "name": "categories",
    "moduleConfig": {
        "text2vec-transformers": {
            "skip": True,
            "options": {
                "waitForModel": True,
                "useGPU": True,
                "useCache": True
            }
        }
    },
    "indexInverted": False
},
```

Figure 39.

Finally, I wanted to upload the entire arXiv dataset to the database, however this would not be the case as I could not get through the entire run time due to the code interrupting when attempting the insert. I only attempted it once more as it took so long and ended up deciding to upload only papers that contained the “cs” category. This decision was made as it meant I could still show a wide range of papers from a familiar research area while still maintaining a broad range of research papers. The cs category contained around 700,000 papers showing that the database still has a significant amount of papers to compare even when removing all other categories.

As shown in figure 39 I changed the Paper schema on top of reducing the number of objects I would insert into the database. This would speed up the vectorizing process of the objects as I applied the skip boolean to all fields except the abstract. This skip boolean removes that field from the vectorizing process if it is set to true. This speeds up the process of inserting data as only the abstract (the most relevant part of the data when comparing semantic similarity) needs to be vectorized.

```
removeUnwantedPapers = lambda x: x["categories"].__contains__("cs") == True and x["categories"].__contains__("physics") == False
```

```
trimmed_loaded_data = (loaded_data.filter(removeUnwantedPapers).map(trim_attributes).compute())
```

Figure 40.

Figure 40 shows how I changed the original trim data lambda function to now filter only by categories that contained the category “cs”. I additionally added the other condition removing papers that contained “physics” as the category field was one continuous string of categories so the function would allow physics papers through as the word contained the string “cs” within it. Figure 41 shows the function that writes these newly trimmed values to a json. I then uploaded this new json to the database using the batch insert function shown in figure 19.

```
with open("ArxivCSOnlyDataset.json", "w") as f:
    for i in range(0, len(trimmed_loaded_data)):
        json.dump(trimmed_loaded_data[i], f)
        if(i < (len(trimmed_loaded_data)-1)):
            f.write('\n')
```

Figure 41.

Testing

I will be testing the false positive rate of my deliverable. This will be done by inputting a piece of text from a computer science related Wikipedia page. This text will be decided by myself. The database will be queried with this text and 10 results will be returned. I will then look through the 10 results and determine if they are relevant to the original input text's concept as a whole. This will be conducted 5 times on 5 different input texts.

"In natural language processing, Latent Dirichlet Allocation (LDA) is a generative statistical model that explains a set of observations through unobserved groups, and each group explains why some parts of the data are similar. The LDA is an example of a topic model. In this, observations (e.g., words) are collected into documents, and each word's presence is attributable to one of the document's topics. Each document will contain a small number of topics." https://en.wikipedia.org/wiki/Latent_Dirichlet_allocation		
Link	Certainty	True/False Positive (T/F)
https://arxiv.org/pdf/1507.06593.pdf	0.923	T
https://arxiv.org/pdf/2112.03101.pdf	0.902	T
https://arxiv.org/pdf/1711.04305.pdf	0.899	T
https://arxiv.org/pdf/1401.6169.pdf	0.897	T
https://arxiv.org/pdf/1804.04749.pdf	0.895	T
https://arxiv.org/pdf/2207.14687.pdf	0.883	T
https://arxiv.org/pdf/2104.07969.pdf	0.880	T
https://arxiv.org/pdf/2110.08591.pdf	0.878	T
https://arxiv.org/pdf/2102.04449.pdf	0.877	T
https://arxiv.org/pdf/1511.03546.pdf	0.873	T

Table 1.

“In natural language processing (NLP), a **word embedding** is a representation of a word. The embedding is used in text analysis. Typically, the representation is a real-valued vector that encodes the meaning of the word in such a way that words that are closer in the vector space are expected to be similar in meaning.” https://en.wikipedia.org/wiki/Word_embedding		
Link	Certainty	True/False Positive (T/F)
https://arxiv.org/pdf/2301.00709.pdf	0.899	T
https://arxiv.org/pdf/2208.08386.pdf	0.894	T
https://arxiv.org/pdf/1811.11002.pdf	0.886	T
https://arxiv.org/pdf/1809.02765.pdf	0.885	T
https://arxiv.org/pdf/1901.07176.pdf	0.884	T
https://arxiv.org/pdf/2007.07287.pdf	0.884	T
https://arxiv.org/pdf/1711.00331.pdf	0.884	T
https://arxiv.org/pdf/1911.00845.pdf	0.881	T
https://arxiv.org/pdf/2209.10583.pdf	0.879	T
https://arxiv.org/pdf/1911.04975.pdf	0.878	T

Table 2.

“In computer science, **concurrency** is the ability of different parts or units of a program, algorithm, or problem to be executed out-of-order or in partial order, without affecting the outcome. This allows for parallel execution of the concurrent units, which can significantly improve overall speed of the execution in multi-processor and multi-core systems. In more technical terms, concurrency refers to the decomposability of a program, algorithm, or problem into order-independent or partially-ordered components or units of computation.” https://en.wikipedia.org/wiki/Concurrency_(computer_science)		
Link	Certainty	True/False Positive (T/F)
https://arxiv.org/pdf/1406.0184.pdf	0.890	T
https://arxiv.org/pdf/1203.4751.pdf	0.869	T
https://arxiv.org/pdf/1406.3485.pdf	0.857	T

https://arxiv.org/pdf/1710.07588.pdf	0.853	T
https://arxiv.org/pdf/0810.1316.pdf	0.849	T
https://arxiv.org/pdf/1511.01779.pdf	0.848	T
https://arxiv.org/pdf/1803.10067.pdf	0.839	T
https://arxiv.org/pdf/1705.02851.pdf	0.835	T
https://arxiv.org/abs/1909.11644	0.829	T
https://arxiv.org/pdf/1812.06011.pdf	0.828	T

Table 3.

"In computer science, a **linked list** is a linear collection of data elements whose order is not given by their physical placement in memory. Instead, each element points to the next. It is a data structure consisting of a collection of nodes which together represent a sequence. In its most basic form, each node contains: data, and a reference (in other words, a *link*) to the next node in the sequence."

https://en.wikipedia.org/wiki/Linked_list

Link	Certainty	True/False Positive (T/F)
https://arxiv.org/pdf/0908.3089.pdf	0.797	T
https://arxiv.org/pdf/1305.6757.pdf	0.761	F
https://arxiv.org/pdf/2207.12942.pdf	0.761	F
https://arxiv.org/pdf/0905.2214.pdf	0.760	F
https://arxiv.org/pdf/1403.0764.pdf	0.760	F
https://arxiv.org/pdf/cs/0308041.pdf	0.759	F
https://arxiv.org/pdf/1009.0929.pdf	0.759	F
https://arxiv.org/pdf/1504.00785.pdf	0.757	T
https://arxiv.org/pdf/1004.1001.pdf	0.756	F
https://arxiv.org/pdf/1710.08748.pdf	0.755	F

Table 4.

"In mathematics, computer science and network science, **network theory** is a part of graph theory. It defines **networks** as graphs where the nodes or edges possess attributes. Network theory analyses these networks over the symmetric relations or asymmetric relations between their (discrete) components." https://en.wikipedia.org/wiki/Network_theory		
Link	Certainty	True/False Positive (T/F)
https://arxiv.org/pdf/0907.3099.pdf	0.912	T
https://arxiv.org/pdf/1710.05660.pdf	0.880	T
https://arxiv.org/pdf/1609.09739.pdf	0.867	T
https://arxiv.org/pdf/1003.3629.pdf	0.866	T
https://arxiv.org/pdf/2209.08294.pdf	0.865	T
https://arxiv.org/pdf/1511.04785.pdf	0.860	T
https://arxiv.org/pdf/2010.16366.pdf	0.860	T
https://arxiv.org/pdf/2104.11329.pdf	0.859	T
https://arxiv.org/pdf/2102.10014.pdf	0.857	T
https://arxiv.org/pdf/2006.16860.pdf	0.855	T

Table 5.

Findings and Conclusion

This was an interesting test resulting in a significant amount of true positives through my own assessment of the relevance of the papers. I determined if the paper was relevant by taking the overall concept of the text into account. This meant that papers that mentioned words within the input text but didn't fit the concept of the text would be treated as a false positive.

The overall mean cosine similarity value of all tables was 0.851 and the overall false positive rate was 16%. However, There was one clear outlier in the test results. Table 4 produced an 80% false positive rate out of the 10 results. This is drastically different to all other tables as they produced 0% false positive results. An indicator of this table being an outlier can be shown through the cosine similarity ratings of all papers in all tables. The mean cosine similarity ratings of all papers excluding Table 4 was 0.829. The mean cosine similarity score of Table 4 was 0.762. This is a 8.79% degree of difference between the cosine similarity scores of Tables except Table 4 and Table 4. This shows that this table was an outlier.

The reason for Table 4 being an outlier is hard to pinpoint. I propose it is due to the fact that the linked list definition has a lot of components to it such as sequences, nodes and data structures. These words most likely match more closely to research papers than a linked list would as this data structure is simple and does not need intense research to expand the understanding of it. This proposal can be inverted to explain the results of the other tables as they contain broad definitions of rich subjects such as concurrency, graphs, networks and topic models. These areas have large amounts of research data compared to linked lists.

In conclusion I think the deliverable performed well in the tests achieving impressive results when given text that had rich research areas available in the dataset. However, the outlying table shows that the deliverable needs improvements. Time constraints don't allow these improvements to be implemented but one theorised method would be to include more readable articles from trusted sources that cover more basic topics such as linked lists. This would give true positive results for these kinds of topics without hindering the more complex topics results.

Completeness

I think the deliverable is complete. I accomplished all the functionality goals I set out to achieve. I believe there could definitely be improvements added to the project such as adding the whole dataset into the database. Even though not all of the dataset data was added I observed that the computer science papers offered enough of a range of topics to display the applications functionality.

Critical Reflection

Planning

I feel that planning was successful when implementing my second prototype, however this cannot be said for my first implementation. Although this first implementation was beneficial for my overall understanding of the technologies I was using, I feel it could have been avoided entirely had I planned my system more thoroughly from the start. On the other hand I feel I learned from these mistakes quickly planning my second implementation and executing on a function by function basis reducing the amount of issues and the severity of issues I encountered. Overall I think I should have developed smaller prototypes that tested individual functionalities, such as native messaging within chrome extensions, as this approach would expose weaknesses in my plan allowing me to get around issues much faster leading to more time for implementation of functionalities.

Throughout this project my time management skills have been lacking. This can be shown as certain features like my bookmark functionality were used partially to save time with the project. This was because I was afraid I would not be able to finish implementing the important features within the time frame given. Also researching took a long time and I often got sidetracked by papers that yielded little information about the subject I was trying to look into. This was especially apparent when researching word embeddings as I struggled to

understand how they were structured at first and took a while getting my head around it as well as finding good papers to reference from. This time researching could have been shortened with effective researching strategies and more research into similar real world implementations as these could have supplied me with information that can't easily be found in research literature. Although there are definitely some downsides to my researching methods I do feel I covered the academic area of my deliverable well. I feel I tapped into the areas at a low enough level to extract confident understanding of the area without going so deep that the content becomes too dense to get through. This approach allowed me to cover all the areas that I researched such as vector databases, topic modelling and word embeddings.

Professionalism, Ethics and Personal Development

Although I didn't face official ethics in the form of human participants and surveys I feel there is still an important ethical element to this project. Mis-information is an important topic and ethics can be applied to it. On the one hand freedom of speech is important and needs to be protected, however blatant manipulation of facts to favour your point of view is malicious and can take advantage of people especially if they are vulnerable. Ethics can also be applied to the deliverable itself. Does the training data that the word embedding model trained on make it biassed? In some way it does as It learns what texts are similar from potentially biassed sources within its training datasets. This could affect resulting links given to people at the inference stage of my deliverable making it a biassed source of information. Although this argument can be made I feel the deliverable is a net positive as it provides easy access to published papers containing deep dives into the statistical analysis of subjects allowing the user to form an opinion on their own using the papers provided by the deliverable.

Testing

The testing used within this paper lacklustre and does not hold much scientific value as it used subjective reasoning from one party to decide whether a paper was relevant to the original input text. This could have been greatly improved by using more data and using a wide range of participants to label the results as true or false positives.

Although this testing method is not valid for proper comparison of the deliverable I do think it offers a good insight into how the deliverable would function with users and the kind of results and usability to expect with it.

Conclusion

In conclusion, I believe I fulfilled my project goals. I managed to develop a chrome extension that allows a user to scan some text and receive links to related research papers. I learnt a significant amount about the topics covered in this paper and have found a new interest in NLP tasks as a whole. With that being said I do think I could have taken certain aspects of the project further If I had more time such as the testing method as well as expanding the data used within the database.

References

Reimers, N., & Gurevych, I. (2020). Making Monolingual Sentence Embeddings Multilingual using Knowledge Distillation. *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing*. https://arxiv.org/abs/2004.09813

Government, U. (25 C.E., April). *Online Safety Bill*. UK Parliament. https://bills.parliament.uk/bills/3137

Blei, D., Ng, A., & Jordan, M. (2003). Latent Dirichlet Allocation. *Journal of Machine Learning Research*, *3*, 993–1022. https://www.jmlr.org/papers/volume3/blei03a/blei03a.pdf

Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, L., & Polosukhin, I. (2017). *Attention Is All You Need*. ArXiv.org. https://arxiv.org/abs/1706.03762

Zhai, M., Tan, J., & Choi, J. (2016). Intrinsic and Extrinsic Evaluations of Word Embeddings. *Proceedings of the AAAI Conference on Artificial Intelligence*, *30*(1). https://doi.org/10.1609/aaai.v30i1.9959

Zhao, H., Phung, D., Huynh, V., Jin, Y., Du, L., & Buntine, W. (2021). Topic Modelling Meets Deep Neural Networks: A Survey. In *arxiv.org*. https://arxiv.org/pdf/2103.00498.pdf

Reimers, N. (2022). *Pretrained Models — Sentence-Transformers documentation*. www.sbert.net. https://www.sbert.net/docs/pretrained_models.html

Reimers, N., & Gurevych, I. (2019). Sentence-BERT: Sentence Embeddings using Siamese BERT-Networks. In *arxiv.org*. https://arxiv.org/pdf/1908.10084.pdf

Pugh, W. (1990). Skip lists: a probabilistic alternative to balanced trees. *Communications of the ACM*, *33*(6), 668–676. https://doi.org/10.1145/78973.78977

Malkov, Y. A., & Yashunin, D. A. (2020). Efficient and Robust Approximate Nearest Neighbour Search Using Hierarchical Navigable Small World Graphs. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, *42*(4), 824–836. https://doi.org/10.1109/tpami.2018.2889473

Kleinberg, J. M. (2000). Navigation in a small world. *Nature*, *406*(6798), 845–845. https://doi.org/10.1038/35022643

Salunkhe, D. (2023, March 30). *Chrome Extension CLI*. GitHub. https://github.com/dutiyesh/chrome-extension-cli

Weaviate. (2023). *Docker Compose | Weaviate - vector database*. Weaviate. https://weaviate.io/developers/weaviate/installation/docker-compose

Grootendorst, M. (2022). BERTopic: Neural topic modelling with a class-based TF-IDF procedure. *ArXiv:2203.05794 [Cs]*. https://doi.org/10.48550/arXiv.2203.05794

COLINDRES, R. (2023, March). *Topic-Modeling with the Arxiv-Dataset*. Kaggle.com. https://www.kaggle.com/code/ricardocolindres/topic-modeling-arxiv-dataset

Suarez-Lledo, V., & Alvarez-Galvez, J. (2019). Prevalence of health misinformation in social media: a systematic review (Preprint). *Journal of Medical Internet Research*, *23*(1). https://doi.org/10.2196/17187

Trueman, C. (2022, December 6). *What you need to know about the UK's Online Safety Bill*. Computerworld. https://www.computerworld.com/article/3681832/what-you-need-to-know-about-the-uks-online-safety-bill.html

Chin, C. (2022, June 11). *The United Kingdom's Online Safety Bill Exposes a Disinformation Divide*. Www.csis.org. https://www.csis.org/analysis/united-kingdoms-online-safety-bill-exposes-disinformation-divide

arXiv.org submitters. (2023). *arXiv Dataset* [Data set]. Kaggle. https://doi.org/10.34740/KAGGLE/DSV/5490982

Connected Papers | Find and explore academic papers. (n.d.). www.connectedpapers.com. Retrieved March 26, 2023, from https://www.connectedpapers.com/

Soft, N. (2022, May). *How to make Chrome Extension 40 WebSocket Client*. www.youtube.com. https://www.youtube.com/watch?v=xlJddufkgJg

Wang, W., Wei, F., Dong, L., Bao, H., Yang, N., & Zhou, M. (2020). MiniLM: Deep Self-Attention Distillation for Task-Agnostic Compression of Pre-Trained Transformers. *ArXiv:2002.10957 [Cs]*. https://arxiv.org/abs/2002.10957

Appendix

PROJECT SPECIFICATION - Project (Technical Computing) 2022/23

Student:	**Thomas Hodgkins**
Date:	**08/02/2023**
Supervisor:	**Zolkeply Syafiq**
Degree Course:	**Computer Science**
Title of Project:	**Developing a deliverable chrome extension that gives easy access to research papers relevant to an input text**

Elaboration

My project consists of taking a pre-trained NLP such as BERT and fine-tuning it on a personalised dataset of research articles and their related keywords (used as labels). This dataset will be gathered using web scraping, finding articles in the machine learning sector from various scholar based websites.

The keywords generated by the model can then be fed into a separate web scraping script that will find related articles based on the keywords passed to it. The results would be shown to the user via a GUI displayed to the user via a chrome extension.

Project Aims

- Produce adequately sized labelled dataset consisting of research papers in the computer science field and their keywords
- Fine-tune a pre-trained NLP model using the dataset outlined above as training data.
- Test this fine-tuning with differing hyperparameters to determine the best performing model for my task.
- Create a web-scraping algorithm that uses the classifications from my fine-tuned model to scrap links for related articles off of the web.
- Curate a GUI that allows users to interact with my fine-tuned model and web scraping script.

Project deliverable(s)

I will produce a chrome extension which will take in a block of text and use this to search for and display relevant research papers based on the predicted keywords from the input.

This chrome extension will be produced using a web stack consisting of a Vue.js frontend and a Javascript/Python backend. The javascript backend will mainly be used for dynamic frontend and the python scripts will be used for all machine learning related computation and code. Backend requests can be called to run specific python scripts and get their return values.

The Machine Learning models will be produced using TensorFlow, Keras and hugging face specifically their Transformer library. These 3 libraries will allow me to get pre-trained transformer models (using hugging face) and add on top of them to fine-tune (using TensorFlow and Keras).

The web scraping scripts will be using BeautifulSoup4, a popular web scraping library. This will be applied to many respected research paper sites such as the ACM Digital Library and IEEE Xplore.

Action plan

- Project Specification and Ethics Form - 10th February
- Information Review - 14th February
- Contents Page - 23rd February
- Draft Evaluation - 9th March
- Gather training data using research paper sites and web scraping - 16th March
- Load pre-trained model and fine-tune it using training data - 20th March
- Test hyperparameters to better fine-tune model for task - 27th March
- Create web scraping script to gather papers based on output of model - 3rd April
- Create skeleton chrome extension to implement scripts into - 7th April
- Finalise documentation for final hand-in - 18th April
- Final Touches - 20th April

BCS Code of Conduct

I confirm that I have successfully completed the BCS code of conduct on-line test with a mark of 70% or above. This is a condition of completing the Project (Technical Computing) module.

Signature: Thomas Hodgkins

Publication of Work

I confirm that I understand the "Guidance on Publication Procedures" as described on the Bb site for the module.

Signature: Thomas Hodgkins

GDPR

I confirm that I will use the "Participant Information Sheet" as a basis for any survey, questionnaire, or participant testing materials. The participant information sheet form is available on the Bb site for the module and as an appendix in the handbook.

Signature: Thomas Hodgkins

List Of Images

List Of Tables