|  |
| --- |
|  |
| Software Development Report |
| Research and Development Project CSC4006 |

|  |
| --- |
| Ryan McCloskey |

Contents

[1. System Specification 3](#_Toc7471144)

[1.1 Requirements 3](#_Toc7471145)

[1.1.1 System Inputs 3](#_Toc7471146)

[1.1.2 System Outputs 3](#_Toc7471147)

[1.1.3 Machine Learning Models 3](#_Toc7471148)

[1.2 Document Model 3](#_Toc7471149)

[1.3 Components 4](#_Toc7471150)

[1.3.1 Document Collation and Storage 4](#_Toc7471151)

[1.3.2 Document and Text Processing 4](#_Toc7471152)

[1.3.3 Document Embedding Models 4](#_Toc7471153)

[1.3.4 User Interface 4](#_Toc7471154)

[2. System Design 4](#_Toc7471155)

[2.1 Document Collation and Storage 4](#_Toc7471156)

[2.2 Document and Text Processing 5](#_Toc7471157)

[2.3 Document Embedding Models 5](#_Toc7471158)

[2.4 System Architecture 5](#_Toc7471159)

[2.5 User Interface 7](#_Toc7471160)

[2.5.1 User Input Window 7](#_Toc7471161)

[2.5.2 Results Window 8](#_Toc7471162)

[3. Implementation 9](#_Toc7471163)

[3.1 Hardware 9](#_Toc7471164)

[3.2 Software 9](#_Toc7471165)

[3.3 Programming Languages and Libraries 10](#_Toc7471166)

[3.4 Database Implementation 11](#_Toc7471167)

[3.5 Document Embedding Models 11](#_Toc7471168)

[3.5.1 Architectures 11](#_Toc7471169)

[3.5.2 Model: Sequence of Words with ULMFiT 11](#_Toc7471170)

[Step 1: Pretraining a Language Model 11](#_Toc7471171)

[Step 2: Fine Tuning a Language Model 12](#_Toc7471172)

[Step 3: Use a fine-tuned language model for Classification 12](#_Toc7471173)

[3.5.3 Model: Sequence of Sentences with BERT Embeddings 13](#_Toc7471174)

[Step 1: Tokenising a document into sentences 13](#_Toc7471175)

[Step 2: Pre-embedding document with pre-trained embeddings 13](#_Toc7471176)

[Step 3: Classifying using the embedded document as input 13](#_Toc7471177)

[3.5.4 Model: Bag of Word with Random Forest 14](#_Toc7471178)

[Step 1: Tokenising a document into word tokens 14](#_Toc7471179)

[Step 2: Training a Random Forest Model. 15](#_Toc7471180)

[3.6 System Architecture 15](#_Toc7471181)

[3.6.1 Text Extraction Module 15](#_Toc7471182)

[3.6.2 Encoder Module 15](#_Toc7471183)

[3.6.3 Document Module 16](#_Toc7471184)

[3.6.4 Search Module 16](#_Toc7471185)

[3.6.5 Main Module 16](#_Toc7471186)

[4. Testing 16](#_Toc7471187)

[4.1 Unit Tests 16](#_Toc7471188)

[4.2 Integration Testing 17](#_Toc7471189)

[4.3 Validation Testing 17](#_Toc7471190)

[Works Cited 18](#_Toc7471191)

# System Specification

## Requirements

Below is a description of the requirements of the system as formulated as part of this research.

## System Inputs

* The system should allow a research paper file to be used as input to a search query.
* The system should allow a specification of which features from an input document to use in a search query, the features are defined as the documents Abstract or Main Text.
* The system should allow a selection of different text classifiers to compare results.

## System Outputs

* The system should provide a list of papers relevant to the search criteria.
* The system should show the relevance of each returned paper to the search criteria.
* The system should provide a means to validate the returned results.
* The system should provide information on each returned paper.
* The system should provide a link to the full version of the paper.
* The system should provide a means to compare the results of different classifiers.

## Machine Learning Models

* The system should test a model of a paper as a sequence of words
* The system should test a model of a paper as a sequence of sentences
* The system should test a model of a paper as a whole document
* The system should test modelling a paper using only its Abstract
* The system should test modelling a paper using its Main Text

## Document Model

Below is the description of the data model of a Bio-Medical paper as required for the document retrieval system.

|  |  |  |
| --- | --- | --- |
| **Section** | **Description** | **Purpose** |
| Document Id | A unique identifier for a paper. | Retrieval and storage |
| Category | The field of research the paper relates to | Labelling of a document |
| Title | Title of a paper. | Inference and description |
| Abstract | The Abstract section of a paper, a brief summary of a research article. | Feature for modelling a paper |
| Main Body | The entire written section of a paper, including abstract, introduction, technical sections, results and conclusions. Excluding sections such as references, acknowledgements, appendices, tables or image data within a paper. | Feature for modelling a paper |
| List of Abstract sentences | A list comprised of the individual sentences that make up the Abstract section of a paper. | Feature for modelling a paper |
| List of Main Body Sentences | A list comprised of the individual sentences that make up the Main Body section of a paper. | Feature for modelling a paper |
| Key Words | A list of keywords within a paper or index terms that are used to categorise a research article. | Feature for validation of results |
| References | A list of documents that a given paper cites. | Feature for validation of results |
| Full Document | An unprocessed version of a given research paper. | Inference and description, data extraction |

Figure 1: Document Data Model

## Components

Below is a description of components necessary to create a document retrieval system based on the requirements in *Section 2.1*.

### Document Collation and Storage

To create and train models for text classification a relatively large collection of input papers is required. This collection is also necessary for the output of the system, where a subset of relevant documents should be returned from a search. These documents need to be stored in such a way to provide relatively quick retrieval for real-time searches.

### Document and Text Processing

For document input, the contents of a paper must be extracted into text and processed into the required format. Certain aspects of the text need to be extracted such as the individual sections referred to in *Figure 1.*

### Document Embedding Models

The system is intended to research the accuracy of applying certain machine learning models to text classification of a given input. Different models are required to test different features of a paper for classification, as mentioned in the requirements in Section 2.1.3.

### User Interface

An interface is required for a user to interact with and gather results from the system. The interface will allow input to the system and controls to select paper features and classification models. The interface will meet the requirements of input and output described in Section 2.1.

# System Design

## Document Collation and Storage

To retrieve and store a large volume of biomedical papers a process will be created to collate documents from a specified source and stored into a database. Metadata of each document will be collected from the source and stored alongside each document in the database. *Figure 2* visualises this process in a flow diagram.

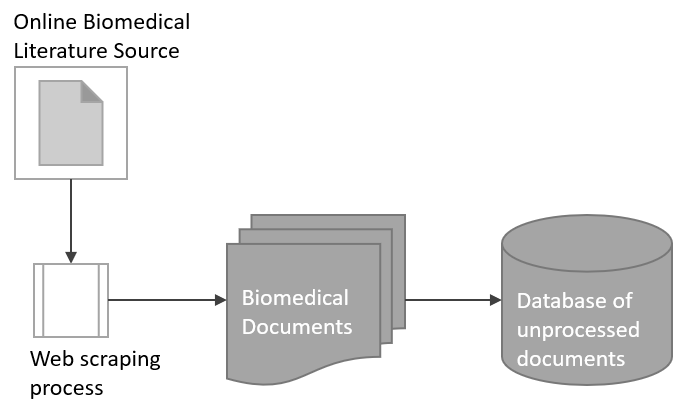


Figure 2: Document Collation Diagram

## Document and Text Processing

Documents will be collected and stored in a file format such as a PDF. To use the contents of a document a process will be defined to convert the format of the file into a text format that can be used for further processing, the converted data will be stored in a database with the original document metadata.

The text extracted from a file will need to go through another processing step to clean and remove unwanted sections or artefacts stored in the file, to extract features described in *Figure 1****.*** Theprocessed text will be stored in a database with the original document metadata. *Figure 3.* visualises the document processing flow.

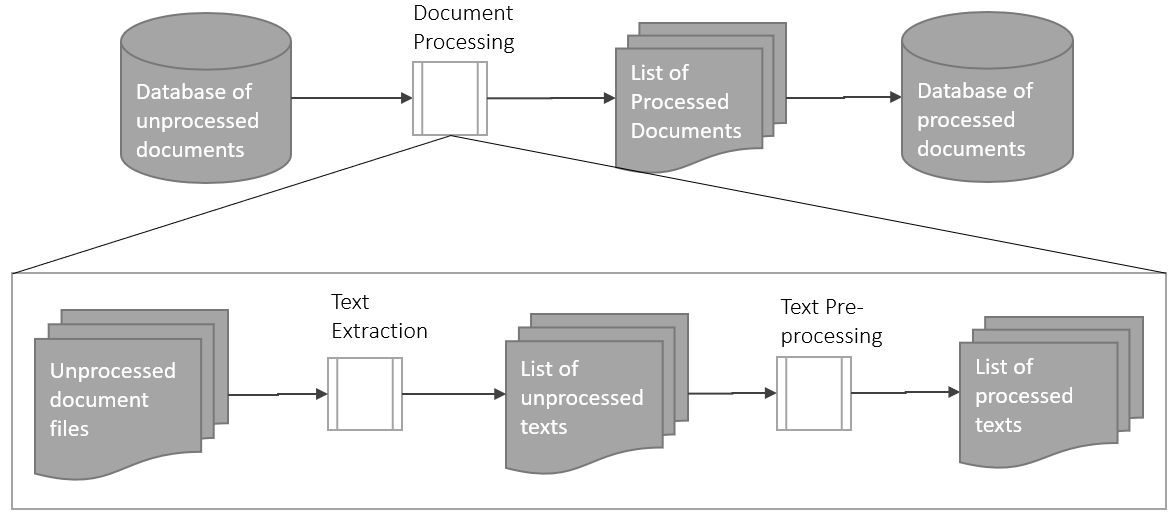


Figure 3: Document Processing Flow Diagram

## Document Embedding Models

For the process of creating a document embedding, text classification models will be created to convert a portion of text into an embedding vector representing the text. These embeddings will be stored along with the document in a database for retrieval and comparison.

*Figure 4.* shows the general process for embedding a document.

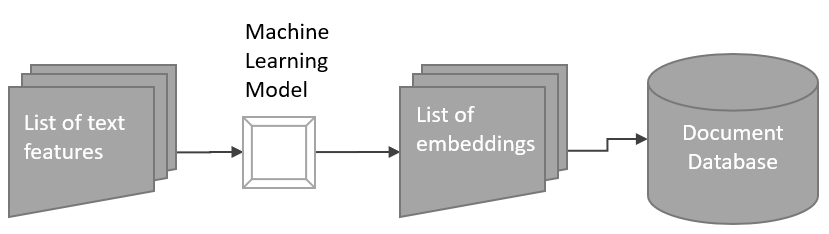


Figure 4: Document Embedding Flow Diagram

## System Architecture

With a database of documents collected and various data extracted from papers such as text features and embeddings, a system will be developed to provide an interface to experiment with the models created. The design of the system is laid out as a flow diagram in *Figure 5*, showing the pipeline from input to results. When interacting with the system, a user makes selections for the inputs described in *Section 2.1.1.* Shown in the diagramas blue arrows.

The general design of the system is:

* Input Document is uploaded by the user, processed and embedded.
* Document embedding is used to compare paper with other documents and their embeddings.
* A list of documents with similar embeddings is returned to the user from the database.

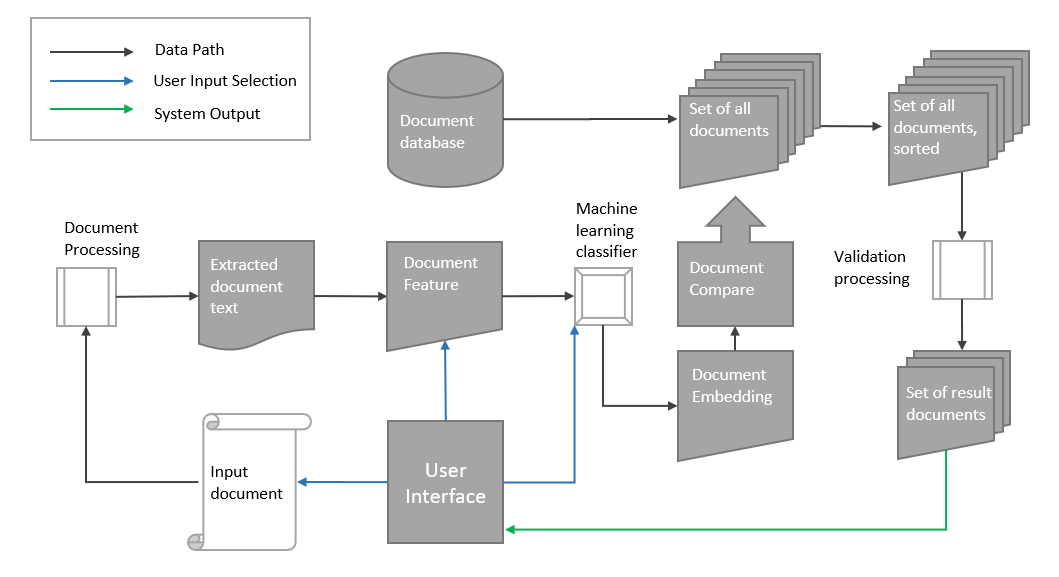


Figure 5: System Architecture Design Flow Chart

## User Interface

The following section details the elements of the User Interface to allow the system to provide meaningful input and outputs. The Interface will be comprised of two main windows, one manages user input *(Figure 6)* and the other will output the results of a search *(Figure 7).*

## User Input Window

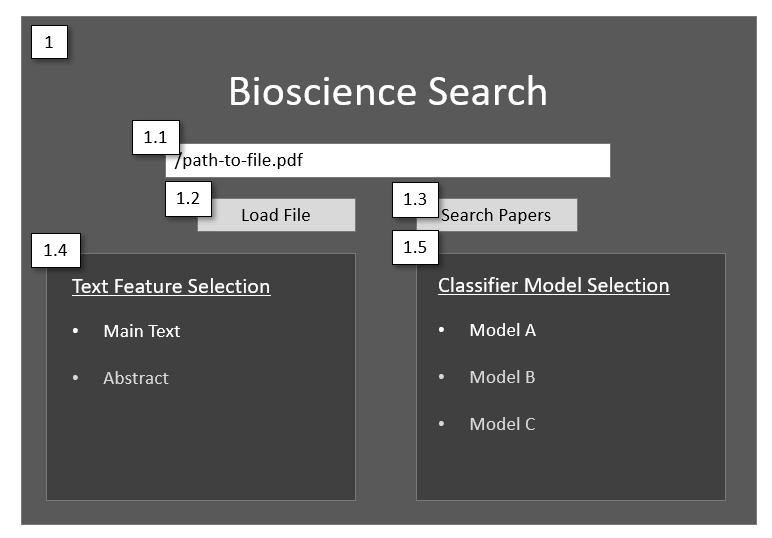


Figure 6: User Input Design

|  |  |  |  |
| --- | --- | --- | --- |
| **Element Id** | **Type** | **Name** | **Description** |
| **1.1** | Text Box | Input file text box | Text Box showing file path of input chosen by a user |
| **1.2** | Button | Load File button | A button that allows a user to search for an input document within their file system |
| **1.3** | Button | Search Paper button | A button that allows a user to search the system for relevant documents to the search criteria. This button will open the results window (*Window 2).* |
| **1.4** | Radio Buttons | Text Feature Selection | This allows a user to select which part of the text to use, either the Abstract or Main Body of the input document. |
| **1.5** | Radio Buttons | Classifier Selection | This allows a user to select the model for which to classify the given input document, this also decides which embeddings to return from the database |

Figure 7: User Input Window Elements

## Results Window

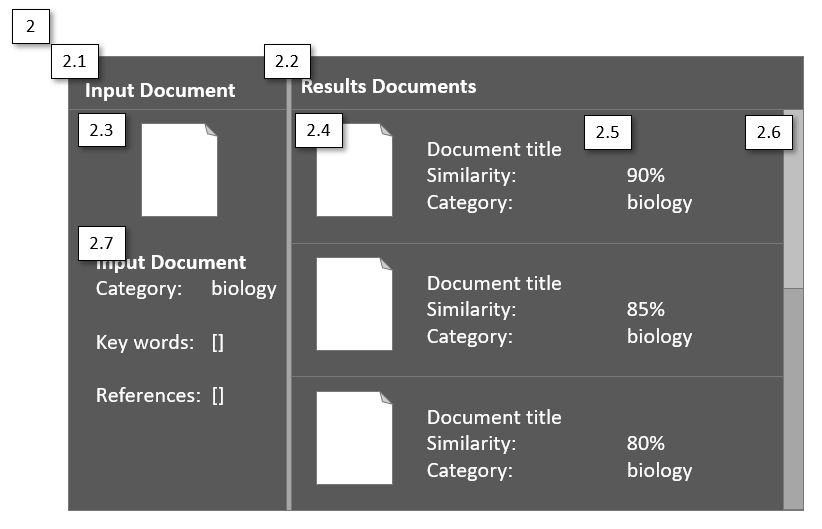


Figure 8: Result Window Design

|  |  |  |  |
| --- | --- | --- | --- |
| **Element Id** | **Type** | **Name** | **Description** |
| **2.1** | Section | Input Document Link | A section that shows information on the input document that was used as the search criteria. |
| **2.2** | Section | Output Document Link | A section that displays the list of result documents along with some information about each. |
| **2.3** | Image, hyperlink | Input Document Link | An image that when clicked, takes the user to the original document file. |
| **2.4** | Image, hyperlink | Output Document Link | An image that when clicked, takes the user to the original document file. |
| **2.5** | Text Information | Input Document Information | Information and metadata about the input provided as the search criteria, e.g. predicted class, keywords and reference lists extracted from the text. |
| **2.6** | Scroll Bar | Results Scroll Bar | A scroll bar that allows navigation through results list. |
| **2.7** | Text Information | Output Document Information | Information and metadata about the result document, e.g. predicted class, keywords, reference list and other data stored about the document. This will also show the similarity of the embeddings between it and the input text. |

Figure 9: Result Window Elements

# Implementation

## Hardware

Development of the system was conducted on a range of workstations and platforms, to suit the needs of certain software, libraries and processes. Certain cloud-based services such as The Google Cloud Platform [1] and Google Colaboratory [2] were used to leverage GPU accelerated hardware.

|  |  |  |
| --- | --- | --- |
| **Hardware Type** | **Specifications** | **Purpose** |
| Linux based workstation | **CPU**: Intel i7 2.40GHz  **Memory**: 11.4GiB  **OS**: 64-bit Ubuntu 16.04 | Required for certain libraries and software that the system requires. (See *Section 4.2 & 4.3*)  Main development workstation |
| Windows-based workstation | **CPU**: Intel i5 2.30GHz  **Memory**: 7.86 GB  **OS**: 64-bit Windows 10 | Used for connecting to cloud services such as google drive and google Colaboratory. |
| GPU accelerated Hardware | **Google Cloud Platform (Colab)**  **CPU**: Intel Xeon 2.30GHz  **Memory**: 13.00Gib  **Disk**: 358 Gb  **GPU**: 1xTesla K80, 2496 CUDA cores, 12GB GDDR5 VRAM | Required for training and experimenting with machine learning models. |
| External Storage | **Google Cloud Platform (Drive)**  **Disk**: 15GB | Required for loading data into cloud-based hardware such as Google Colaboratory. |
| External Storage | **External HDD**  **Disk**: 1TB | Required for data transfers and backups. |

Figure 10: Hardware Description

## Software

This section details the software used to develop the document retrieval system, the programming languages and dependencies required for development and for the final system to function.

|  |  |  |  |
| --- | --- | --- | --- |
| **Name** | **Version** | **Type** | **Purpose** |
| Atom | 1.36.0 | Text Editor | Primary text editor for development |
| MongoDB | 2.4.1 | Database Framework. Includes mongo shell | Database framework used to store and query document data for document retrieval system. |
| Robo3t | 1.3 | Database User Interface | GUI tool to view records and query database |
| Python | 3.6.7 | Programming Language | Primary programming language for development |
| Microsoft Excel | 16.23.0 | Spreadsheet software package | Used for results of the system |

Figure 11: Software Description

## Programming Languages and Libraries

The system was primarily developed on and implemented using Python 3.6.7. This section details the libraries and dependencies and their usage.

|  |  |  |
| --- | --- | --- |
| **Library Name** | **Description** | **Usage** |
| **Machine Learning Modules** | | |
| **TensorFlow** | Machine Learning framework | Used to create machine learning models for language modelling and text classification of documents |
| **Fastai** | Machine Learning framework with ULMFiT Implementation [3] | Primarily used for the ULMFiT implementation |
| **BERT-as-service Client and Server** | Framework for converting sentences to BERT [4] embeddings | Used to convert sentences of a paper to embeddings, for use in downstream tasks such as text classification |
| **NLTK** | Natural Language Processing framework | Used for extracting words and sentences from strings in text data, as well as cleaning and manipulating text data |
| **SKitLearn** | Machine Learning framework | Used for machine learning models and helpers such as test-train split functions |
| **Data Manipulation Models** | | |
| **Numpy** | Python array and matrix library | Used for data manipulation |
| **Pandas** | Data manipulation framework | Used for data manipulation, loading and saving |
| **csv** | Data manipulation and io framework for csv format | Used for data manipulation, loading and saving |
| **json** | Data manipulation and io framework for json format | Used for data manipulation, loading and saving |
| **pickle** | Framework for saving and loading python objects | Used for data loading and saving |
| **Textract** | PDF to text parsing framework | Used for converting downloaded pdf files to text |
| **Re** | Regex library within python | Used for cleaning and manipulating text data |
| **HTML and Request Modules for Web Scraping** | | |
| **Beautiful Soup** | HTML Parsing framework | Used for web scraping of BioArxiv |
| **urllib** | URL handling framework | Used for web scraping of BioArxiv |
| **Requests** | HTTP request handling framework | Used for web scraping of BioArxiv |
| **Database Interface Modules** | | |
| **PyMongo** | Interface for mongo database | Used for loading and storing document data |
| **GUI Development Modules** | | |
| **Tkinter** | GUI development library for python | Primary library for developing the user interface of the system |

Figure 12: Python Libraries Description

## Database Implementation

Documents were collated and stored using a No-SQL database implementation with MongoDB, MongoDB allows data manipulation and transfer using the JSON format. The structure of the Document model as stored as a record in the database is described in Figure 13. PyMongo is the python library used to interface the database with the system

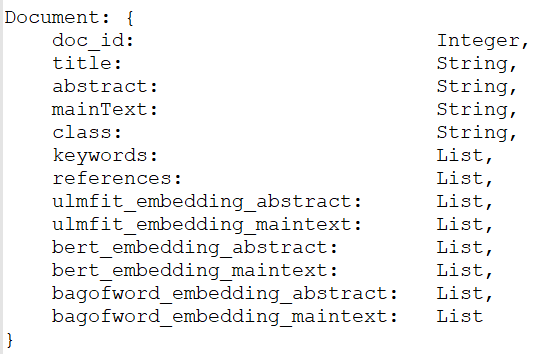


Figure 13: MongoDB Record Structure

## Document Embedding Models

## Architectures

## Model: Sequence of Words with ULMFiT

For this model, a document was represented as a sequence of word tokens, this model was implemented using Universal Language Model Fine Tuning. [3]. The model was implemented in a three-step process.

### Step 1: Pretraining a Language Model

Training a language model using a large general domain corpus of text

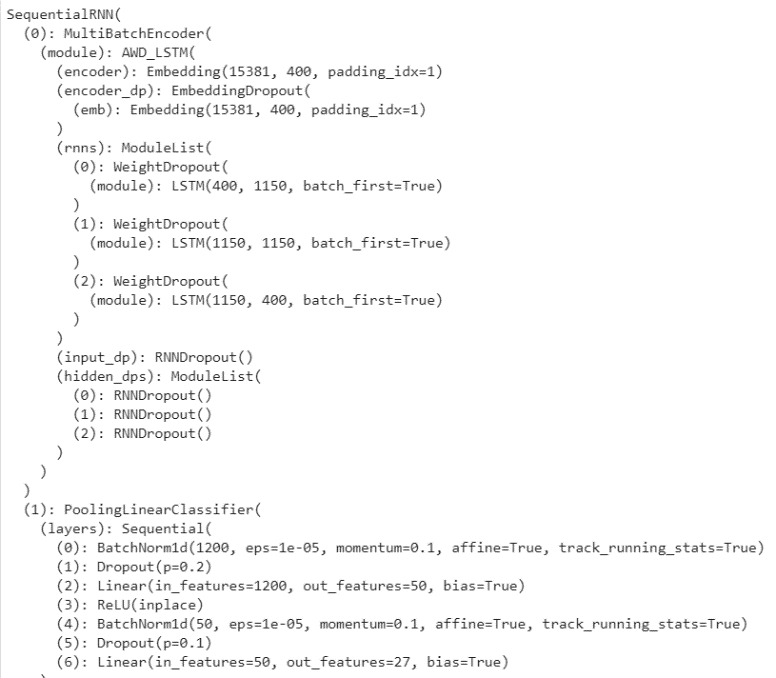
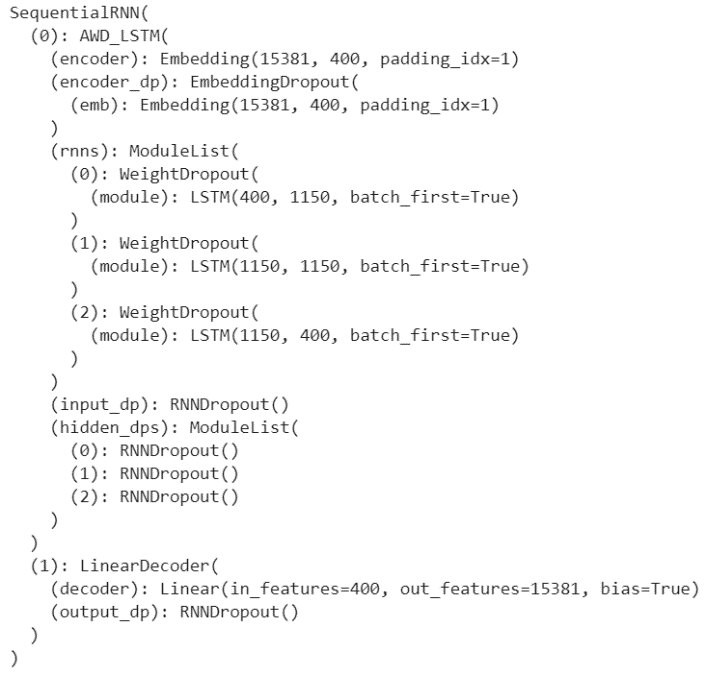
The language model is a sequential Recurrent Neural Network (RNN) based on the AWD LSTM [5] architecture. The layers of the model are represented in *Figure 14.*

Figure 15: Architecture of ULMFiT Classifier Model

Figure 14: Architecture of ULMFiT Language Model

The Model implements transfer learning to ‘learn’ something about the document using a pre-trained language model, trained on a large general corpus of data. For this model, the pre-trained corpus used is *Wikitext-103* [6], a model trained on the majority of Wikipedia articles in English. The tokens of the bio-paper vocabulary are compared to tokens that exist in the *Wikitext* model’s vocabulary. Any that exist in both are mapped to weights in the embedding matrix of the language model. Any tokens that don’t appear in the *Wikitext* model are given an average weight from all weights in the *Wikitext* model embedding matrix. The embedding vector is 400 dimensions.

### Step 2: Fine Tuning a Language Model

Once the embedding matrix of the pre-trained model is modified using the biomedical corpus vocabulary, the model can be trained and fine-tuned. For the language model, this involves continuously trying to predict the next word in a series of words.

The data is split into ‘Batches’. Totalling 64 batches over the number of tokens present in the concatenated corpus. Each batch is transposed to become a matrix of shape 64 columns and (total tokens/64) rows. Each batch is taken one at a time and a sample of the column (a sequence of tokens) is fed to the model to predict the next token. This sample indicates the sequence length for Back Propagation Through Time (*bptt*) for the Recurrent Neural Network. The size of the *bptt* isn’t fixed and is changed at random (roughly 5% of the time) as to not overfit predicting on a certain sequence length. The output layer mirrors the input matrix as it will be a prediction of a certain token from the entire corpus.

### Step 3: Use a fine-tuned language model for Classification

The classifier model is a sequential Recurrent Neural Network (RNN) based on the AWD LSTM architecture, similar to the Language model. With a Linear Classifier used as an output. The layers of the model are represented in *Figure 15.*

Once the fine tune language model is trained to a good degree of accuracy the embedding matrix can be transferred to a classification model. The classifier model has the same basic structure as the language model, with a linear classifier model that uses max and average pooling layers to aggregate weights into a final decision layer. The structure of this model can be seen below in figure 2. The initial Embedding layer is the same embedding matrix as the language model, with each token represented by a 400-dimension matrix. The last layer of the model acts as the classifier. The document is passed through the same way as the language model, in batches except for the sequence length that's passed in is larger encapsulating every token in the document with the target set to one of the labelled classes. The same *bptt* is used but a ‘max sequence length’ is defined which indicates that the last number of those activations are considered.

The output layer acts as a probability distribution for the input document, a vector of 27 dimensions with the highest value corresponding to the predicted class of that document.

The embedding process for this model is represented by the diagram in *Figure 16.*

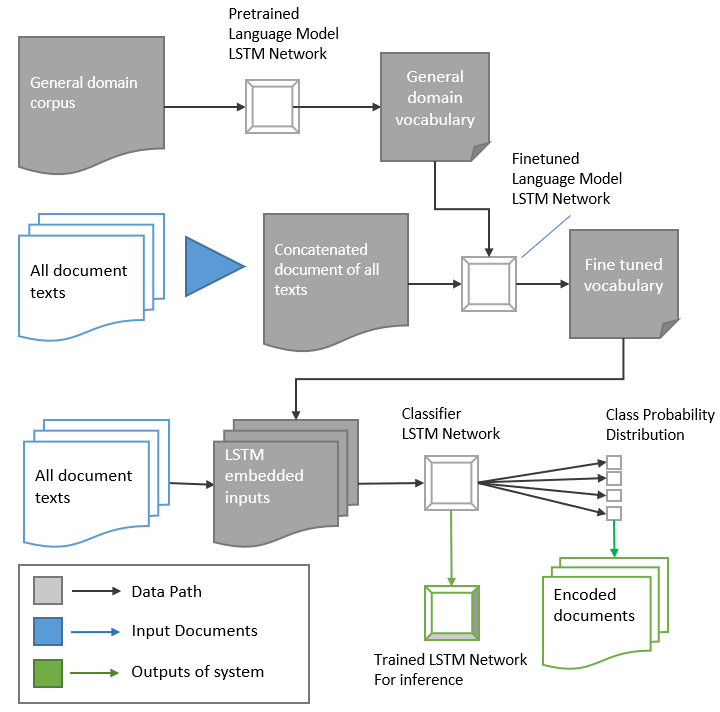


Figure 16: Diagram of ULMFiT Embedding Model

## Model: Sequence of Sentences with BERT Embeddings

The next approach at embedding a document was to create a model based on representing a document as a sequence of sentences. For this, the BERT (Bidirectional Encoder Representations from Transformers) [7] language model was implemented. Namely the BERT as a service software [4] which allows mapping of variable length sentences to fixed length embeddings. The embedding process is defined in three steps.

### Step 1: Tokenising a document into sentences

To tokenise a document the NLTK python library was implemented. Namely the Punkt sentence tokenizer module. [8] which is an implementation of an algorithm [9] developed by Kiss et al. (2006). A text was split into a set of sentence strings.

### Step 2: Pre-embedding document with pre-trained embeddings

Once a document was represented as a set of sentences, each sentence was mapped to a fixed length (768) vector using the bert-as-service software and a pre-trained language model created by the BERT team. The language model used was BERT-Base, Cased [10]. Once embedded a document would be padded to a fixed length depending on the feature type. It was found that the average length of *Abstracts* was around 10 sentences and *Main Texts*, 100 sentences long. Short documents were padded with arrays of zeros while long documents were trimmed to length by using the first x sentences.

### Step 3: Classifying using the embedded document as input

A sequential LSTM classifier model was implemented to embed the documents. The architecture of the model is represented in *Figure 16.* Similar to the ULMFiT classifier, the output layer acts as a probability distribution for the input document, a vector of 27 dimensions with the highest value corresponding to the predicted class of that document. The model did not require an embedding layer as input due to the pre-embedded nature of the document.

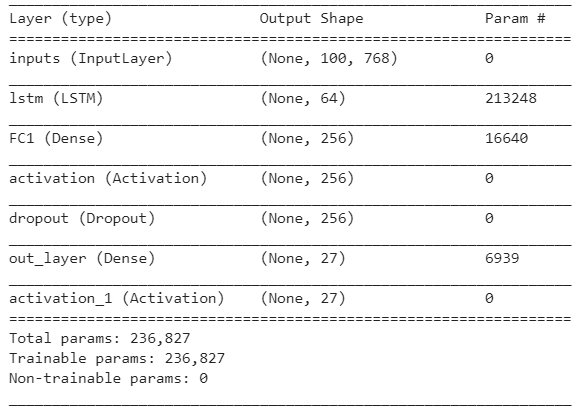


Figure 17: Architecture of BERT LSTM classifier Model (using Main Text feature)

The embedding process for this model is represented by the diagram in *Figure 17.*

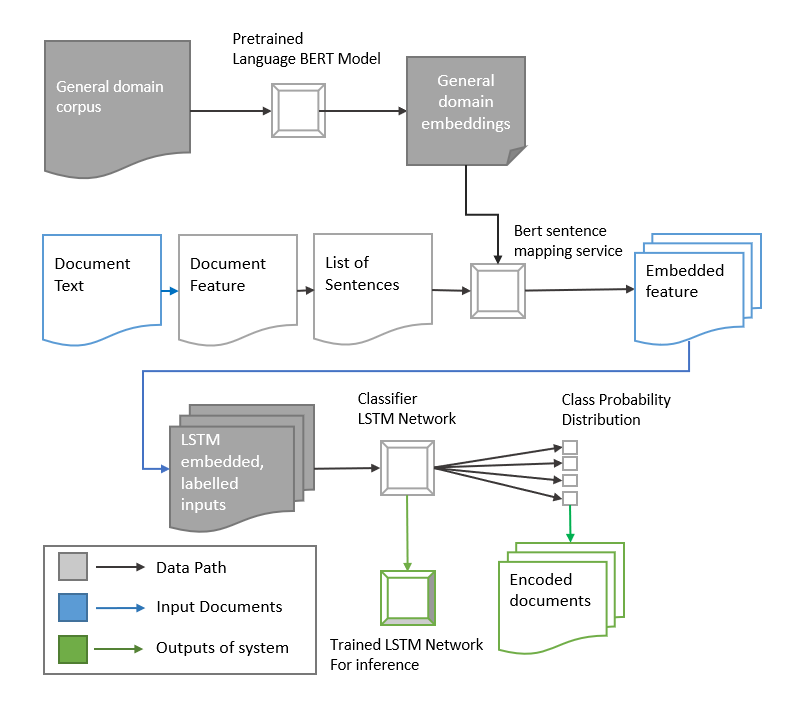


Figure 18: Diagram of BERT Embedding Model

## Model: Bag of Word with Random Forest

The final embedding of a document is a more basic model, designed as a benchmark to compare the results of using the deeper sequence features of the other models. This model is implemented by classifying a document based on the frequency and importance of single word tokens in a text. The embedding process is defined by the following steps.

### Step 1: Tokenising a document into word tokens

Using the NLTK Punkt tokenizer module as discussed in Section 4.6.3, as well as the CountVectorizer module from the SKLearn python library [11]. a vocabulary of words was created for the document corpus. The top 2000 word-tokens were used to represent the vocabulary. The vocabulary was saved separately for the inference model.

### Step 2: Training a Random Forest Model.

With a vocabulary created, a Random Forest Classifier was trained and used to create document embeddings. The classifier uses 100 trees in the forest to estimate, this value was found to be a reasonable balance of accuracy and complexity for training the model.

The embedding process for this model is presented in Figure *19.*

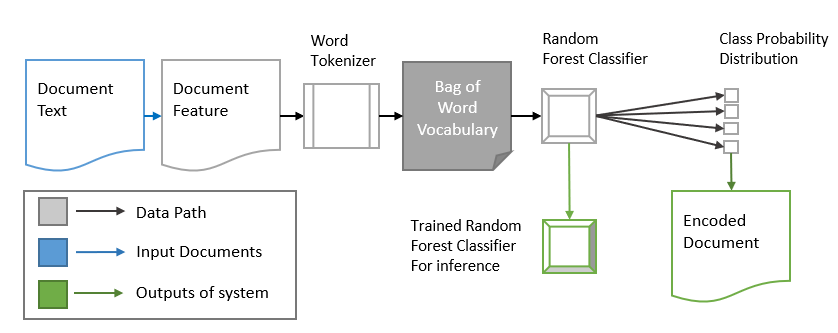


Figure 19: Diagram of Bag of Word Embedding Model

## System Architecture

To implement the requirements and design of the proposed system, a number of modules were created.

## Text Extraction Module

Module primarily used for document input and pre-processing.

|  |  |
| --- | --- |
| **Class** | **Description** |
| Extract\_Text | Extracts various sections (as per document model in *Section 2.2*) of text from a string representation of a document |
| PDF\_to\_Text | A class that implements ***Textract*** to attempt to convert a PDF encoded file to text data, the class also implements a function to clean text of artefacts left from conversion (Image data, tables etc) |

## Encoder Module

Module primarily used for embedding documents.

|  |  |
| --- | --- |
| **Class** | **Description** |
| Encoder\_ULMFIT | Class to load and implement models created using the Fastai library. Model is loaded from a pickle file |
| Encoder\_BERT | Class to load and implement models using *Tensorflow*, this class also implements the ***bert-client*** and ***bert-server*** for converting sentence strings (representing sentences of the text) into vector embeddings before encoding a document. Model is loaded from a .*h5 file*. |
| Encoder\_BagofWord | Class to load and implement models created using the *SKLearn****.*** For this encoder, a classifier model was loaded from a *pickle* file as well as an *Encoding Vectorizer*, which is used to embed a String to an embedded vector. Model is loaded from a pickle file |

## Document Module

Module primarily used for generating the list of results.

|  |  |
| --- | --- |
| **Class** | **Description** |
| Doc\_Compare | Contains functions for comparing document embeddings, implements the cosine distance algorithm. Also, validation functions for comparing keyword and reference matches. |
| Doc\_Finder | Provides an interface between the document database and the system. Maps database records to objects that can be used within the system and be returned to the user. |
| Doc\_Sort | Provides functions for sorting results based on distance, keyword matches or reference matches, providing a user with control on what |
| Search\_Doc | Class for input document representation. Contains fields for different required sections of the input *(Section 2.2)* |
| Result\_Doc | Class for result document representation, contains fields representing different fields of records in the database, in addition to fields for document compare results (cosine distance) |

## Search Module

A module used for executing a search using input document and variables.

|  |  |
| --- | --- |
| **Class** | **Description** |
| Search\_Documents | A function that searches for relevant documents based on the input document, classifier and a result limit variable.  The function retrieves all document embeddings saved in the database and compares them to input. Then returns a sublist of result sorted by embedding distance. The list consists of document ids and embedding tuples. |
| Results | Used to convert results of the search function to documents by querying the database with a list of paper id's generated from a search. |

## Main Module

The Entry point and GUI manager of the system

|  |  |
| --- | --- |
| **Class** | **Description** |
| Main | The main module acts as the entry point for the system. The main loads the necessary modules and classification models. It also handles the Graphical User Interface. The GUI is implemented using the Tkinter python library, as listed in *Figure 12.* |
| Search | Searches system based on input parameters and input document, once a search is complete a new window is drawn to display the results of the search. |

# Testing

A test harness was created to ensure the various components of the system function as intended. The tests are described in the following sections.

## Unit Tests

As the system is built using a number of distinct modules, the functionality of each was tested in isolation to ensure correct execution.

|  |  |  |
| --- | --- | --- |
| **Module** | **Test** | **Description** |
| Text Extraction | Extract from PDF | Using a test PDF file, the various functions of the text extraction module were tested to ensure the correct strings are extracted from the document. |
| Document Module | Document Compare | Compare two mock vectors. Assert distance between 0 and 1.  Compare two mock vectors with the same values, assert distance of 0 |
| Document Module | Document Sort | Assert that the list of results is sorted from lowest to highest distance. |
| Document Module | Document Finder | Assert that there is a successful connection to the database and return a document count. |
| Encoder Module | Encoding Test | using a given document stored in the database, encode the document via the various models and compare the embedding vectors. Assert that the vectors match. |
| Encoder Module | Sentence Tokenize | Using a mock paragraph to convert to sentences and embed using bert-as-service. Compare to a premade list of sentences encoded using bert. Assert that encoding vectors match. |

## Integration Testing

To ensure the system works as intended, a test PDF document is passed through the system a series of test cases are checked for each encoder (ULMFiT, BERT, BagOfWord) and feature (Abstract, Main-Text)

1. Text is correctly extracted from a file
2. Text is converted to Feature
3. The feature is converted to embedding using selected Encoder
4. Set of result documents are returned

## Validation Testing

To validate the results and ensure the system is returning documents relevant to the input, a single paper from the document set was passed through the system and embedded using the various classification models, the top 1000 most similar documents were returned. The class and similarity value were exported for each document to a csv. The data stored in the csv was then visualised to observe the quality of documents returned for a given input. This also allows a comparison of the various embedding methods.

# Works Cited

|  |  |
| --- | --- |
| [1] | Alphabet, “Google Could Platform,” Google, [Online]. Available: https://cloud.google.com/products/. [Accessed 2019]. |
| [2] | Google, “Google Colaboratory,” [Online]. Available: https://github.com/jupyter/colaboratory/blob/master/README.md. [Accessed 2019]. |
| [3] | J. a. S. R. Howard, “ Universal language model fine-tuning for text classification.,” *arXiv preprint ,* 2018. |
| [4] | X. Han, “bert-as-service,” 2018. [Online]. Available: https://github.com/hanxiao/bert-as-service. [Accessed 2019]. |
| [5] | S. N. S. K. a. R. S. Merity, “Regularizing and optimizing LSTM language models,” *arXiv preprint,* no. arXiv:1708.02182, 2017. |
| [6] | C. X. J. B. a. R. S. 2. Stephen Merity, “Pointer Sentinel Mixture Models,” 2016. |
| [7] | J. M.-W. C. K. L. a. K. T. Devlin, “Bert: Pre-training of deep bidirectional transformers for language understanding,” 2018. |
| [8] | S. B. E. L. J. N. a. A. D. willy csse, “NLTK Punkt Tokenizer Module,” [Online]. Available: https://www.nltk.org/\_modules/nltk/tokenize/punkt.html. |
| [9] | T. a. J. S. Kiss, “Unsupervised multilingual sentence boundary detection,” *Computational Linguistics,* no. 32.4, pp. 485-525, 2006. |
| [10] | Google, “BERT, or Bidirectional Encoder Representations from Transformers,” 2018. [Online]. Available: https://github.com/google-research/bert. [Accessed 2019]. |
| [11] | Scikit-Learn, “CountVectorizer Module,” [Online]. Available: https://scikit-learn.org/stable/modules/generated/sklearn.feature\_extraction.text.CountVectorizer.html. [Accessed 2019]. |