



# UNIVERSITY *of* LIMERICK

O L L S C O I L L U I M N I G H

## Text Classification/Analysis Using Neural Networks and Word2Vec: How To Use The Technique

**Author:** John Rey Juele

**Student ID:** 15167798

**Supervisor:** Farshad Toosi

### **Final Year Project Report**

B.Sc. Computer Systems

Department of Computer Science and Information Systems

Academic Year – 2018/2019

## Table of Contents

Acknowledgements.....	i
Declaration.....	ii
Project Summary.....	1
Chapter 1: Introduction .....	2
1.1. Context/Motivation .....	2
1.2. Objectives.....	3
1.3. Technologies .....	4
Chapter 2: Research.....	6
2.1. Artificial Neural Networks.....	6
2.1.1. The Inspiration .....	6
2.1.2. The Application .....	7
2.2. Word2Vec .....	11
2.3. Doc2Vec .....	14
2.3.1. Introduction .....	14
2.3.2. Vector Creation .....	15
2.3.3. Word2Vec vs. Doc2Vec .....	15
2.3.4. Distributed Memory Model (PV-DM) .....	16
2.3.5. Distributed Bag of Words (PV-DBOW) .....	17
2.3.6. Vectors For Unseen Documents .....	17
2.3.7. Doc2Vec Application .....	18
Chapter 3: Implementation .....	19
3.1. Introduction to the Implementation.....	19
3.2. Dataset Acquirement .....	19
3.3. Text Pre-processing.....	23
3.4. Doc2Vec & Word2Vec Implementation.....	24
3.5. Classifier .....	27
3.6. Software Architecture & Design Patterns.....	30
3.6.1. MVC (Model View Controller Architectural Pattern).....	30
3.6.2. Strategy .....	31
3.6.3. Factory Method.....	33
3.6.4. Command.....	35
3.6.5. Observer.....	38
Chapter 4: Product Evaluation .....	42

---

Chapter 5: Conclusion .....	48
5.1. Possible Improvements.....	48
5.2. Learning Outcome.....	49
References .....	51
Image References.....	53
Table of Figures.....	54
Dataset sources.....	54
Appendix .....	55
GUI Snapshots.....	55
Doc2Vec Tool .....	55
Doc2Vec Results GUI.....	56
Word2Vec tool .....	56

## Acknowledgements

I want to take this quick opportunity to express my thanks to everyone who was involved in this project. I am grateful for my supervisor, Farshad, who played a key role during the FYP as he provided the guidance and insight I needed to complete this project as well as his encouragement throughout the period. I also thank all my lecturers who helped me develop my knowledge and skillset throughout the four years in UL. I also want to express my appreciation for the friends I have made along the way, the ones who made my time at university an adventurous one.

Of course I wish to mention my loving family who remain by my side no matter where I am and what I am doing. It is them who motivates me to always keep doing my best in everything that I do.

## Declaration

The work described in this document is, except where otherwise stated, entirely that of the author and has not been submitted in any part for a degree at this or any other University.

Signed: \_\_\_\_\_

Dated: \_\_\_\_\_

## Project Summary

For my final year project, I developed an application that will take a piece of text or a set of texts as input and then labelled them with a topic or subject that best described them. This project made use of concepts and ideas that are brought out in the fields of machine learning and natural language processing and the objective was to use these to create software that fulfils a need in real life.

During the lifetime of this project I looked into developing an application that incorporated software models capable of performing classification tasks, of which I paid close attention to neural networks, the theory and the operations that make them work. I also included “Word2Vec” (Mikolov, et al., 2013), a relatively recent technique devised by a team of researchers at Google. This technique converts words into numerical representations which can in turn help the computer process and understand language while at the same time keeping the semantics of the words intact. Word2Vec can also be extended to cover not just words but entire documents with the help of “Doc2Vec” (Mikolov & Le, 2014).

As the machine learning algorithms that I used utilised a supervised style of learning, it was necessary to obtain a good set of training data (Brownlee, 2016). This could range from news articles, academic papers to non-formal texts sources such as Twitter and other social medias. There was no limit to the sources I could pull from as long as they were appropriately labelled due to the fact that the performance of the classification task greatly depended on the quality of the training data. The pre-processing of the text was also required in order to help the machine efficiently handle the text input that was fed into it. This included stemming the words, removing unnecessary punctuations and other pre-processing techniques on text data.

One of the goals of this FYP was to produce working software and therefore I also considered the characteristics and best practices to developing good software. I approached this project similar to how a business would approach such a development problem by keeping in mind the functional and non-functional

requirements, devising a plan of action and to incorporate design patterns so that the application remained in line with the “SOLID” design principles to software development.

## Chapter 1: Introduction

### 1.1. Context/Motivation

As we are living in a digital age, information has become widely distributed and therefore easily accessible. Even though we have not completely moved on from physical books and other paper-back publications, more emphasis has been placed on making information available electronically. 80% of the information available to us, both in physical and electronic form, is in text format (Korde and Mahender, 2012) .

Helping computers understand and process human language has been an active area of research in computer science especially in the field of natural language processing (NLP). One of the subareas of this field is in the classification of text. Computers are being used as tools to automate work especially in analysing and processing text with a view of classifying them. With this in mind, text classification is a contributory building block in many NLP applications. Following on from this, the technological environment today has encouraged researchers to place more attention in incorporating and implementing mechanisms of text classification in everyday, practical tasks. We can see this in action with the prevalence of machine learning today regarding web searching, sentiment analysis and data filtering (Aggarwal and Zhai, 2012), as well as in digital assistants such as Apple’s Siri and Amazon’s Alexa.

It has always been an interest of mine to dabble in the field of machine learning and to gain an understanding on how the resources of computers can be manipulated to mimic and simulate human-like intelligence and learning. Communication and language is an integral part of being human as well as of society as a whole. I was curious to discover the nuts and bolts that operate underneath the machine learning

technology of today, especially on how they perform human-like tasks such as understanding text.

## 1.2. Objectives

The main goal of this project was to develop an application that read the text provided to it and labelled each one an appropriate topic or subject. This goal could be broken down even further to three major milestone or sub-goals.

1. **A training and test dataset must be founded and readily available.** I used a supervised style of learning for my classification model. Therefore it was vital that the dataset be correctly labelled and was suitable for the task at hand. Upon obtaining the dataset, they should also be pre-processed. This includes but is not limited to stemming the words in the document, removing unnecessary punctuation and “stop words”.
2. **Develop the program using the SOLID design principles.** For the implementation of the program I intended to follow a 3-tiered closed architecture as well as including the MVC (Model View Controller) architectural pattern. It was also a goal of mine to add in necessary design patterns with a view of accommodating quality attributes such as maintainability and extensibility. The program will of course include a graphical user interface that would display the labelling. I planned to design the GUI as user friendly as possible by making it easy to use, clear and organised.
3. **Test and evaluate the end product.** Upon finishing the implementation, I had planned to create different test cases for the outputs of the program. I also aimed at evaluating the classification model that I would be incorporating into the program and to determine the accuracy of said model. Using the data from the tests I made it a goal to tweak the different parameters of the program with a goal of increasing and improving the accuracy of the classification model.



This FYP introduced new concepts and ideas that I had not encountered before. Therefore I had personal goals that I set for myself. Python was a programming language that I had no knowledge of prior to this FYP. I hoped to achieve a strong level of competency of the language by means of working through the project. Machine learning was an area of computer science that I found interesting, yet I had very little knowledge of the field. My means of the project, I intended to increase my current knowledge base in this area as I believed it would prove practical in the real world.

### 1.3. Technologies

The below are the main technologies that I used for the FYP.

#### **Python**

Python was the programming language that I predominantly used to implement the application. The main reason why I decided to use python was because I did not use python previously and the FYP was an opportunity to learn and utilise the language in a meaningful way. Another reason why I chose python was the many libraries available as well as the documentation and tutorials that were available online.

#### **PyCharm**

PyCharm was the IDE that I used for this project. It is one of the IDE's provided by JetBrains who also have an IDE available for the Java programming language called "IntelliJ". I was very familiar with IntelliJ and the features and settings in IntelliJ were also available in PyCharm and it was because of this that encouraged me to use it.

#### **Python Libraries**

- **Gensim**

Gensim was used to help create my Word2Vec and Doc2Vec models. According to the Gensim website its purpose is to "realize unsupervised semantic modelling from plain text" (Řehůřek, 2009). The library provided the functionality in creating the models, training the models, finding similar

words, obtaining document vectors as well as some useful utility functions such as text cleaning.

- **NumPy**

NumPy was the library that I used to perform the mathematical operations within my project. The library also made available a more efficient implementation of arrays and matrices which I found to be useful in my research and in my own implementation of a simple neural network.

- **Scikit-Learn**

Scikit-learn is a library that provides already made implementations of many machine learning algorithms. This library was particularly helpful in the creation of my classification models as well as the evaluation of such models.

- **Beautiful Soup**

Beautiful Soup was the library that I used to parse through HTML files and to extract data from them. I used this library alongside python's in built **urllib** package to automatically obtain information from websites and to create my dataset.

## Chapter 2: Research

### 2.1. Artificial Neural Networks

An artificial neural network (ANN) is an “information processing paradigm” that takes inspiration from the workings and the operation of the biological brain (Stergiou and Siganos, n.d.). ANN’s are not necessarily an algorithm; that is it follows a specific and concrete set of steps to solve a particular problem. They can be seen more as a paradigm or a framework as to how to structure and conceptualise the program.

#### 2.1.1. The Inspiration

Artificial neural networks are inspired by our brains which are comprised by a large network of neurons. A neuron is just a single nerve cell that transmit electrical signals from one end to the other.

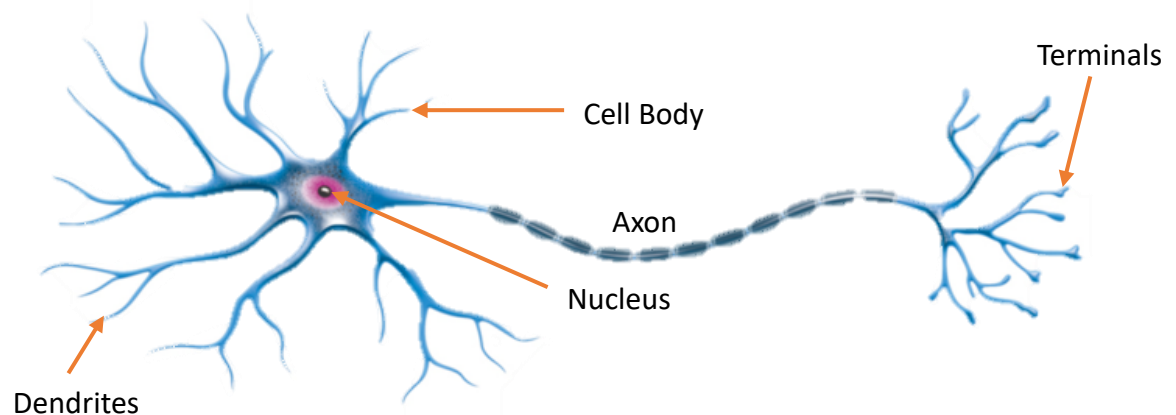


Figure 1: A Neuron (PNGImage, 2019)

In the case of the diagram above, the electrical signal would enter the neuron through the dendrites which will then be processed inside the nucleus. It is only upon reaching a certain threshold that an output signal would be produced which is then sent through the axon and exits through the synapses within the neuron’s terminals (Rashid, 2016, pp. 41-43).

### 2.1.2. The Application

We can model an artificial neuron like so:

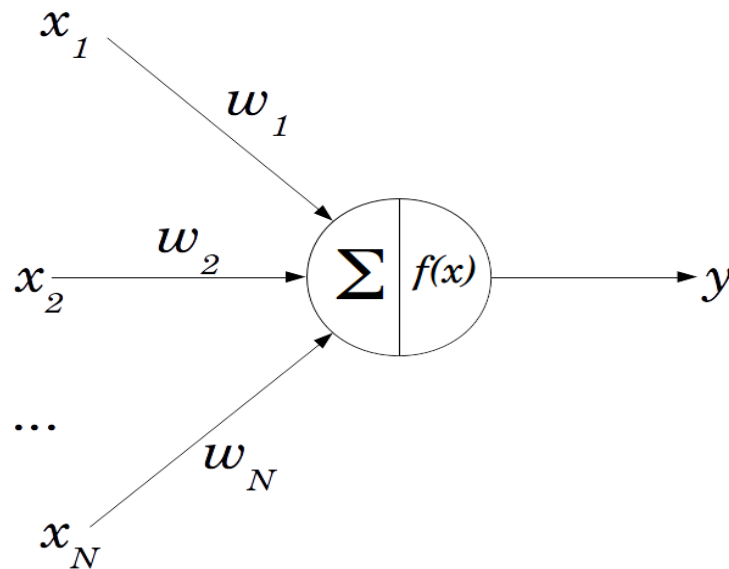


Figure 2: The Perceptron (Lucidarme, 2019)

The official name of an artificial neuron is a “perceptron” which can act as a single processing unit of an ANN (Rosenblatt, 1958). The main components of the perceptron are:

- **The weighted inputs**

There can be N inputs to this neuron which are also weighted. The weights on each connection signifies the strength of that connection and therefore influences the processing and ultimately the output of that perceptron. These weights are assigned random values initially. They can be seen as the dendrites from the biological neuron.

- **The neuron**

Within the neuron itself is where the processing takes place. Two main processes take place. The “Σ” or sigma signifies the summation of the inputs in the form:

$$sum = (x_1 * \omega_1) + (x_2 * \omega_2) + \dots + (x_N * \omega_N)$$

As stated earlier under the biological brain, a certain threshold has to be met before the output is fired out of the neuron. This is modelled using an activation function:  $f(x)$  where we pass the sum into that function. There are many types of activation functions such as the step function and the sigmoid/logistic functions (Rashid, 2016, pp. 44-45). The result of that function will be that neuron's output or in other words its *activation value*.

Just like the brain is a network of neurons, we can model a network of perceptrons like so:

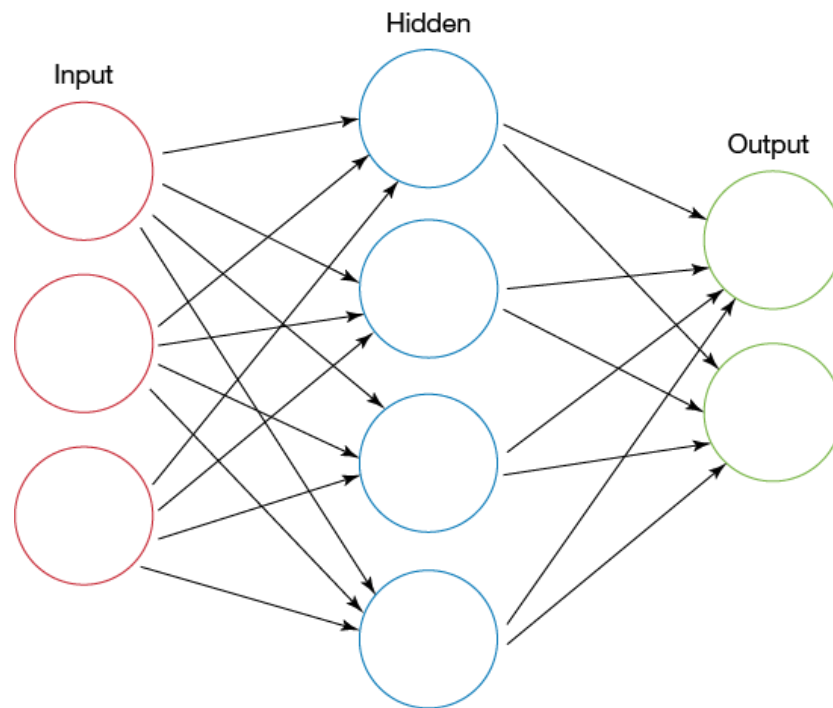


Figure 3: A feed forward neural network (Perry, 2019)

Each circle is a neuron/perceptron where the output of that neuron will act as the input of the next neuron. I should also note that connections between neurons are weighted (as weights are not included on the above diagram). In a typical neural network architecture, there are three layers: the input, hidden and output layers. The number of nodes in the output and input layers are dependent on the problem domain while the number of nodes in the hidden layer is arbitrary.

In order to produce a guess out of the neural network, it must perform a feed forward mechanism through the network. Once executed, the activation value of each node in each layer is calculated until processing reaches the output nodes. The activation value of the nodes of the output layer is the neural network's guess. I should also point out that there is a more efficient way of performing the feed

forward algorithm. Everything on the neural network can be represented as matrices (Nielsen, 2018).

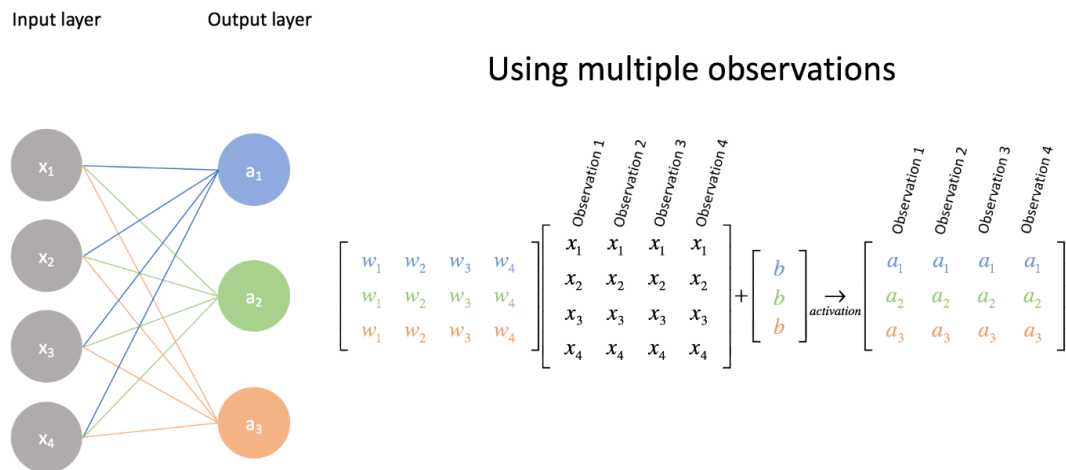


Figure 4: Matrix representation of the NN (Jordan, 2019)

In order to find the activation values of the nodes in the next layer, a dot product operation can be done between the matrices of each layer. With the above diagram, we perform the operation between the matrix representing the weights between the input and hidden layer and the matrix representing the input layer nodes in order to produce the activation values in the hidden layer. The diagram above has not included the activation function however. So upon obtaining each values from performing the multiplication, we apply the activation function on each of the values in the result in order to find activation value of the nodes. We can do the same operation in order to find the output layer values. Through linear algebra operations we can simplify the feedforward algorithm especially programmatically.

However running the feed forward on a neural network for the very first time may not operate and produce a result that is expected. That is due to the fact that it is untrained. The neural network must learn from already existing data in order to help it produce a more accurate guess. As I have pointed earlier, the weights between each layer greatly influence the activation values of the next neuron. We can also

see them as the decision makers of the network. As the weights are initialised with random values, they will likely produce inaccurate activation values in the next node. So from that idea, when training the neural network, what really is happening in the background is that the weights are being adjusted towards a more accurate value.

A concept found in supervised learning is the idea of errors. This is very similar to how humans learn from making mistakes. We compare our work or guess with the actual answer and to use the mistakes we have learnt to adjust our current knowledge. So in the case of neural networks, we make it produce a guess for a particular set of inputs and to compare its guess with the actual answer that is correlated to that set of inputs. Let's put this mathematically:

$$error = answer - guess$$

Upon discovering the error of the output layer (as the error produced from the overall guess of the NN is the error of the output layer), we use those errors to traverse or propagate back towards the input layer of the NN while at the same time finding the errors of each node in each layer. This is where the mechanism of "backpropagation" comes in (Klein, 2011).

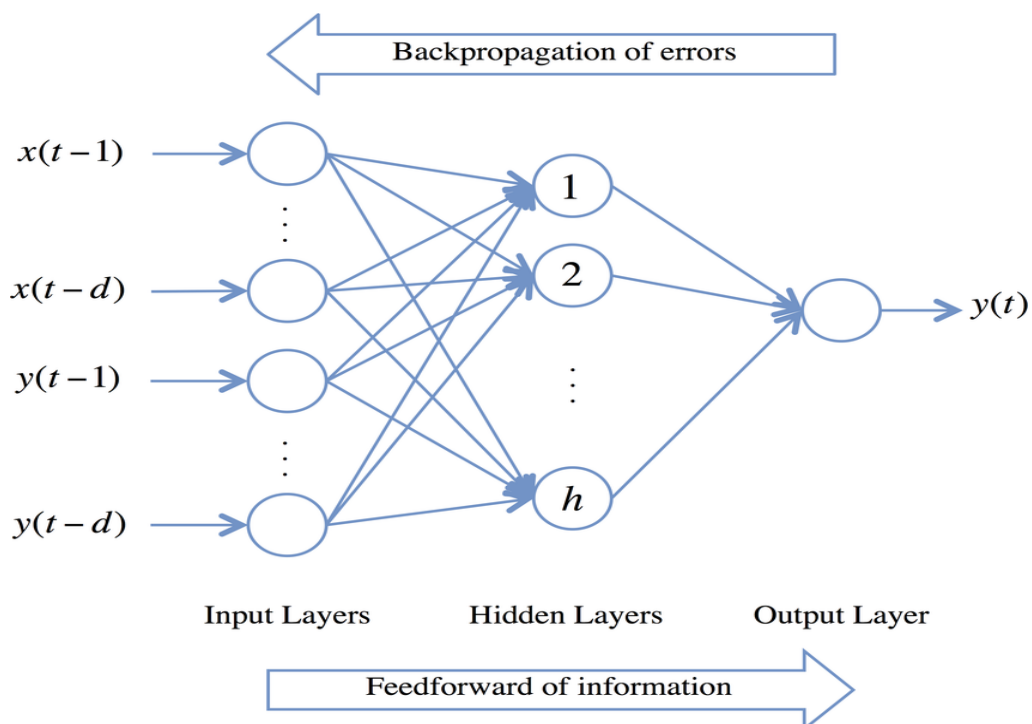


Figure 5: Feed forward & Backpropagation (Chan Phooi M'ng and Mehralizadeh, 2016)

Once we have obtained all the errors of each node in the neural network it will help us to figure out how much we should adjust each of the weights in the networks. The goal of the learning process is to minimize the errors and get them as close to zero as possible. This brings up a new idea of “gradient descent” (Rashid, 2016, pp. 83-93). From this discussion we can conclude that the effectiveness of a neural network is not determined fully by the efficiency of the implementation but by the quality of the training data. As this is a supervised model, the guesses produced by neural networks depend fully on the data it learnt from.

Gaining an understanding on the functionality of a neural network was very important for this project as it was the foundation of many of the core features of the project application. We can see one in use in the next section with regards to Word2Vec.

## 2.2. Word2Vec

Word2Vec is a mechanism that converts words into “word embeddings” or numerical representations of words (TensorFlow, 2018). There are many different ways of creating word embeddings such as TF-IDF vectorisation, count vectors and co-occurrence matrices. However, what is great about Word2Vec is that it maintains the semantical relationships that exist between words. We can illustrate this by performing matrix calculations between word vectors. For example, the word vector produced by:

$$\text{vector}(\text{"King"}) - \text{vector}(\text{"Man"}) + \text{vector}(\text{"Woman"})$$

is a vector that is very similar to the word vector of “Queen” (Mikolov, et al., 2013).



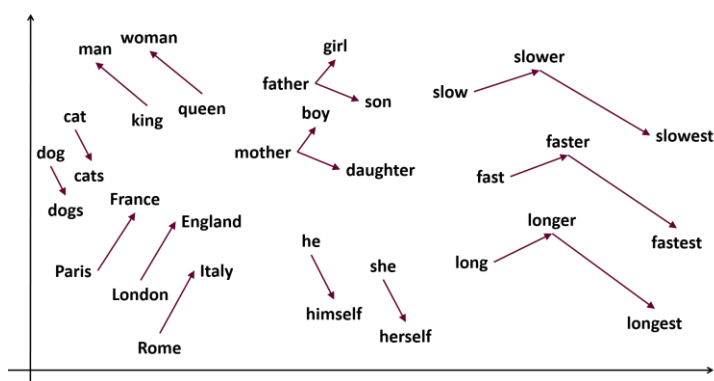


Figure 6: Vector relationships (Samyza.com, 2019)

Word2Vec makes use of shallow neural networks (NN with only one hidden layer) that are trained to do a specific purpose. The two main purposes are described by the two architectures of Word2Vec: **Continuous Bag of Words** and **The Skip-gram model**.

### The Skip-gram model

The skip-gram model is a neural network that is trained to predict the neighbouring word (context words) given a specific word in the text corpus.

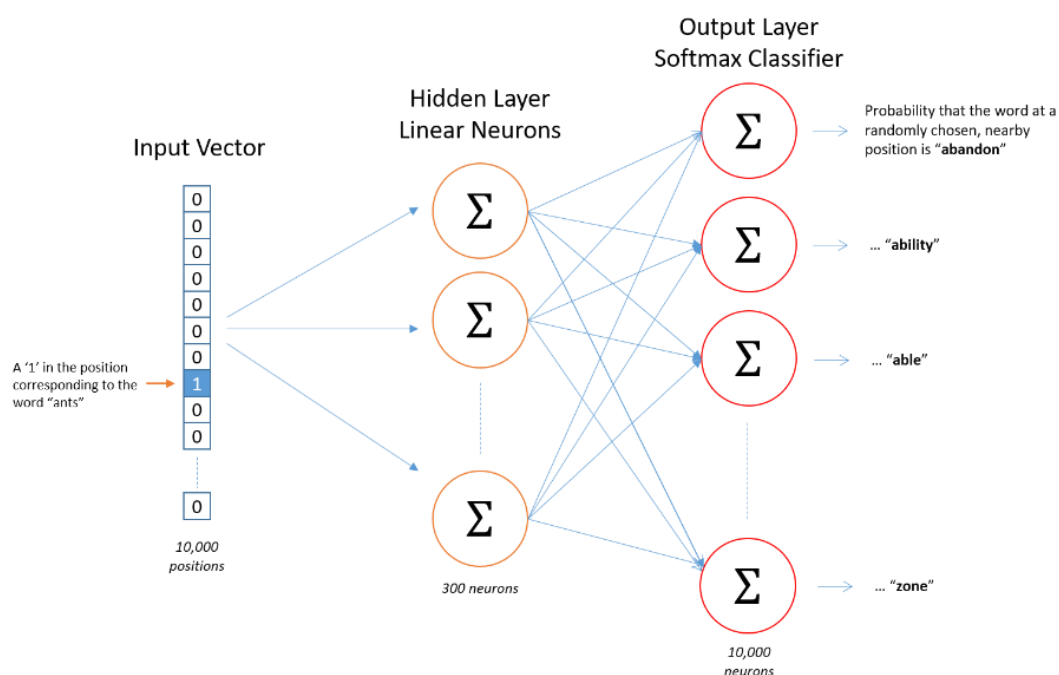


Figure 7: The skip-gram architecture (Chablani, 2019)

### Continuous Bag of Words (CBOW)

Continuous Bag of Words is a neural network that is trained to guess a particular word given the word(s) that surrounds that word.

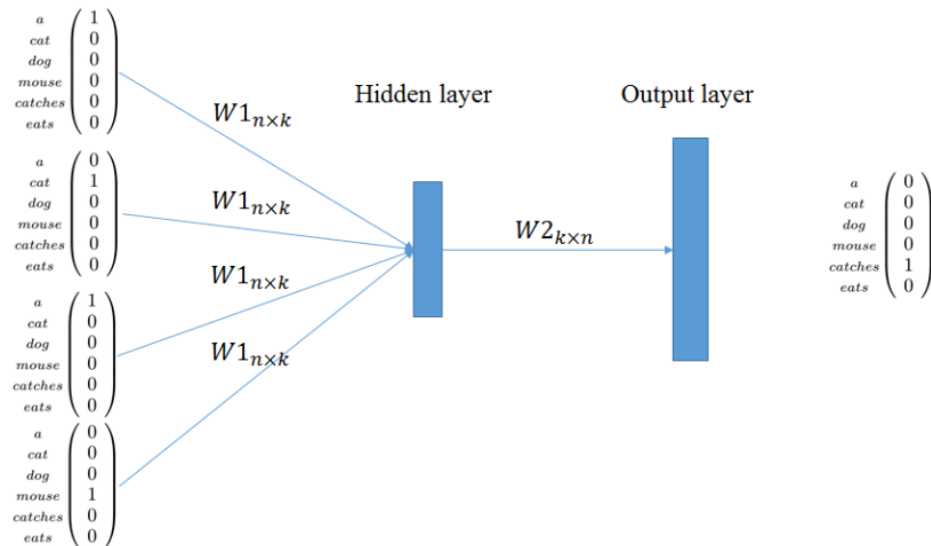


Figure 8: The CBOW architecture (Khalid, 2019)

As these neural networks are being trained to fulfil their target purpose, the weights connecting each layer in their networks are being adjusted towards a more accurate value. It is from these weights that we can derive the word vectors for the words in the text corpus (McCormick, 2016). As I have pointed earlier everything on the neural network can be represented as matrices.

$$\begin{bmatrix} 1.1 & -1.1 & 5.5 \\ 6.3 & 0.02 & 4.2 \\ 2.3 & -1.6 & 4.2 \\ 3.2 & 7.2 & 0.6 \\ 8.3 & 4.9 & 4.1 \end{bmatrix}$$

Figure 9: The weight matrix representing the weights between the input and hidden layer of a hypothetical Word2Vec model

We can represent the input word to the NN like so:

$$[0 \ 0 \ 1 \ 0 \ 0]$$

Figure 10: A one-hot representation of the input word “Computer”.

In order to find the word vector for the word “Computer” (As an example) we perform a simple dot product between the two matrices.

$$[0 \ 0 \ 1 \ 0 \ 0] * \begin{bmatrix} 1.1 & -1.1 & 5.5 \\ 6.3 & 0.02 & 4.2 \\ 2.3 & -1.6 & 4.2 \\ 3.2 & 7.6 & 0.6 \\ 8.3 & 4.9 & 4.1 \end{bmatrix} = [2.3 \quad -1.6 \quad 4.2]$$

By doing such an operation, what we’re really doing is that we’re selecting the word vector from all the stored weights representing all the possible words in the corpus and this selected word is projected into the hidden layer. So from the above example the word embedding for the word “computer” is:  $[2.3 \quad -1.6 \quad 4.2]$

## 2.3. Doc2Vec

### 2.3.1. Introduction

In the previous section, we saw how word vectors are created through the use of a shallow neural network while simultaneously capturing the semantical relationships between words through the use of Word2Vec. Thomas Mikolov and Quoc Le had a vision of extending the functionality of Word2Vec to create vectors for groups of words. This piece of text can be of variable length, ranging from sentences to entire books (Mikolov and Le, 2014). They called this mechanism “Paragraph Vector”. However this mechanism is commonly known as “Doc2Vec. Please note that as a

result of Mikolov and Le's definition of "text", I will be using the words "paragraph" and "document" interchangeably throughout this section.

### 2.3.2. Vector Creation

To re-iterate Word2Vec makes use of 2 specific models: "Skip-gram" and "Continuous Bag of Words". These two models are shallow neural networks that are trained to perform a particular task. In summary:

- **Skip-gram:** A neural network trained to guess the neighbouring words (the context) of a given target word in a specified window size.
- **Continuous Bag of Words:** A neural network trained to guess the target word given the neighbouring/context words in a specified window size.

As we have learnt previously, during the training process, the weights between the input and hidden layers are being adjusted to more precise values. At the end of training, a matrix of weights are produced and can be seen as a table containing all the vectors for the words.

### 2.3.3. Word2Vec vs. Doc2Vec

Doc2Vec is an extension to Word2Vec. Doc2Vec uses the same models from Word2Vec and performs the exact same training process of those models. In fact, the creator of the gensim library Radim Řehůřek stated that the Doc2Vec class actually extends the Word2Vec class in the library (Řehůřek, 2014). However, where it differs is that Doc2Vec makes use of "paragraph ids" or "documents ids". By introducing the document ids, small changes are introduced to the Word2Vec models as a new matrix is included into the system that actually stores the document vectors. The ultimate goal of these two models is to produce a matrix containing document vectors.

### 2.3.4. Distributed Memory Model (PV-DM)

Distributed memory is Doc2Vec's version of Word2Vec's "Continuous Bag of Words" model. The task of the neural network of distributed memory is to predict the next word that follows after a given sentence. The words of the sentence is passed in through the input layer with the expectation that the neural network would produce the expected word that follows that sentence. I should point out that all words that are involved in this prediction task belong to the same document which is denoted by a document id.

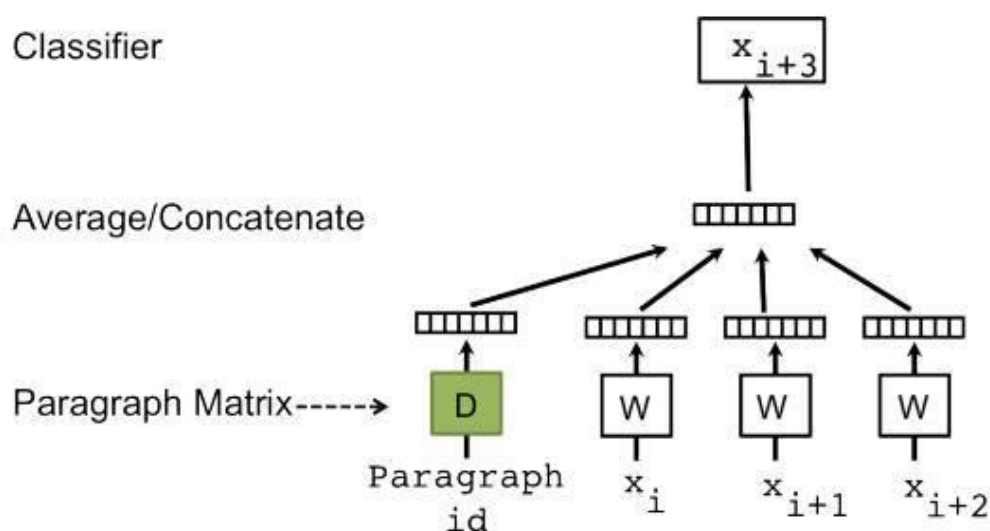


Figure 11: Distributed Memory model (Irene, 2019)

This document id is also passed in as input along with the involved words. Two matrices are involved in the training process, the document matrix and the word matrix. During the training, the words and the document id are processed within the hidden layer. There is a choice of either concatenating or averaging the vectors involved. As stochastic gradient descent and backpropagation is involved, the weights are adjusted in order to help increase the accuracy of producing the correct prediction. The weights involved in producing a document vector are found in the document matrix while the weights involved in producing the word vectors are found in the word matrix. In order to obtain the vector for a document, the same process for obtaining a word vector in Word2Vec can be used. However in this case,

the document matrix is used as it contains the vectors for all the documents involved in the training.

### 2.3.5. Distributed Bag of Words (PV-DBOW)

DBOW can be viewed as Doc2Vec's take on the Skip-gram model. The task of this neural network is to predict words that are randomly sampled from a given document.

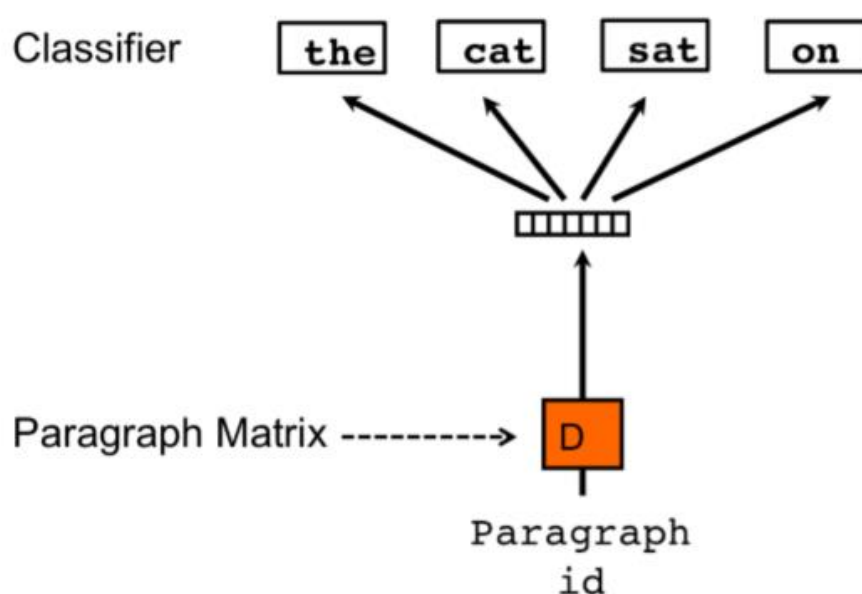


Figure 12: Distributed Bag of Words model (Budhiraja, 2019)

Precisely speaking, only a fixed number of words are sampled from that document. Then a random word from that list of words is chosen and it is that word the neural network will be trying to predict given that document's id. This model saves more memory than distributed memory has there is no need to store the weights/vectors for the words.

### 2.3.6. Vectors For Unseen Documents

Due to the fact that Doc2Vec utilises an unsupervised approach to learning, a trained Doc2Vec model can be used to produce vectors for documents that it has not

encountered before. This unseen document is augmented as another row into the document matrix. Initially, this unseen document would be in a one-hot encoded format. All other weights/vectors in the neural network are frozen or kept fixed. In order to transform the one-hot encoding, several iterations of feed-forward and backpropagation cycles are performed. In other words, training takes place for a number of iterations where the goal of the training depends on what Doc2Vec model was chosen. Just to reiterate, no weights are being adjusted during this training process other than the weights associated with the newly added document. As a result, a vector is produced from the one-hot encoding that best represents the newly unseen documents.

### 2.3.7. Doc2Vec Application

The Doc2Vec model in the project will act as a document vector factory/producer. My intention of using this mechanism was to produce vector representation of documents. Unlike humans, computers struggle in understanding text, especially the meaning and context behind that piece of text. As pointed out in the previous section, Word2Vec captures the meaning of words effectively. Doc2Vec extends this by capturing the context behind chunks of text and this can be seen through the relationships between the document vectors. Converting documents to vector representation through Doc2Vec helps the computer process these documents while at the same time, maintaining the meaning of these documents. From the outset, the Doc2Vec model is fed with training documents. Once training is complete, the model should be ready to produce vectors for unseen documents. These vectors produced by the Doc2Vec model will act as the input for the classification model that will perform the actual labelling of the documents.

## Chapter 3: Implementation

### 3.1. Introduction to the Implementation

In this chapter, I will be talking about the main components of the software application I created. The main components of the implementation are:

- Dataset acquirement
- Text pre-processing
- Doc2Vec & Word2Vec
- Classifier

For each of these components, I will be talking about in detail what actions I took in order to fulfil the goals of each component. I will then discuss how I joined these components together. In that particular section, I will be detailing the MVC architectural pattern as well as the software design patterns I included into the implementation in order to incorporate some of the SOLID design principles into my software application.

### 3.2. Dataset Acquirement

As stated earlier, the quality of the performance of the application depends heavily on the quality of the dataset I use to train it with. So the first step I took was to find a reliable source of text data that is appropriately and correctly labelled. The ideal dataset for me is when the text corpus:

- Is labelled
- Contains documents such that the context of that document accurately describes the label that is linked to it.
- Is to the point and exact. In other words, the text should revolve around its subject matter while simultaneously trying to avoid mixing other topics/subject into itself.
- Has little to no spelling mistakes or other grammatical errors.



I should point out however that documents may have correlations between each other with regards to subject matter or topic and therefore avoiding the mixture of different topics in the same document cannot be 100% avoided. For example, a document that talks about “Google” can dabble between the topics of “Tech” and “Business”. I embraced this fact while searching for a dataset source. However complications arose when this is accepted too much and I will talk about this more in depth in the testing and conclusions chapter of this report.

I wanted to have a large supply of text data as a large corpus can lead to better output accuracy. My initial plan was to use academic papers from sources such as “Nature.com”, “IEEE”, “ACM” as the papers precisely stick to their subject matter, are labelled and contain very little to no grammatical errors as they are typically proof-read. Unfortunately, it was not at all simple to acquire documents from these types of sources. As I needed a large amount of documents, I was obliged to automate the acquirement of the dataset as downloading or acquiring the documents manually would be an impossible task to do. However, this approach could not be done for sources I mentioned above as payment was required to get the documents. It was possible to obtain these documents for free as a student of a university but due to the fact that I needed to automate the process, it was not possible to login as a student by means of a python script.

I was provided with a dataset from a member of the department that contained labelled news articles from BBC (Greene & Cunningham, 2006). This was close to what I wanted but it only contained approximately 2,000 documents which is not enough for what I wanted. I stumbled upon another dataset on Kaggle (Misra, 2018). This dataset contains a JSON file which had about 200,000 lines of objects in this format:

```
{
  "category": "CRIME",
  "headline": "There Were 2 Mass Shootings In Texas Last Week, But Only 1 On TV",
  "authors": "Melissa Jeltsen",
  "link": "https://www.huffingtonpost.com/entry/texas-amanda-painter-mass-shooting_us_5b081ab4e4b0802d69caad89",
  "short_description": "She left her husband. He killed their children. Just another day in America.",
  "date": "2018-05-26"
}
```

*Code 1: A json object in the HuffPost dataset*

These json objects represent articles from “Huffpost”. This json file appealed to me as it mentioned the link to the main article as well as it being labelled. The dataset contained many topics but I only chose 6 of them: “BUSINESS”, “POLITICS”, “SCIENCE”, “SPORTS”, “TECH” and “TRAVEL”. I wrote a python script that parsed through each json object, obtained the link and its category (topic) and scraped the website for its text using the obtained link which made use of the “urllib” package as well as the Beautiful Soup library. The below code (*code 2*) is a snippet of my Scraper class which helped in obtaining the main text content of a website. The URL was set using the “update\_web\_client( )” function and the text of the website was pulled using the “get\_text( )” function.

```

from bs4 import BeautifulSoup as Soup
from urllib.request import Request, urlopen as uReq

class Scraper:

    def __init__(self):

        # Data members
        self.web_client = None

    def update_web_client(self, url: str):

        req = Request(url, headers={"User-Agent": "Mozilla/5.0"})
        self.web_client = uReq(req)

    def get_text(self) -> str:

        result = ""

        html_page = self.web_client.read()
        self.web_client.close()

        page_soup = Soup(html_page, "html.parser")
        text = page_soup.findAll("div", {"class" : "content-list-component yr-content-list-
text text"})

        try:

            for t in text:
                try:
                    result += t.find("p").getText()
                except:
                    try:
                        result += "\n"
                        result += t.find("h3").getText().upper()
                        result += "\n"
                    except:
                        result += "\n"

            result += "\n"

        except Exception as e:
            result = ""
            print(str(e))

        return result

```

*Code 2: Scraper class that was used to scrape the text from a website.*

My script then saved the text in a .txt file and placed it in the folder whose name is equivalent to the category of the current json object. In this manner, I did not have to manually copy paste the text from the news articles by hand. I should also point out that the name of a folder was the topic associated with the documents contained in that folder. This made it easier to label the documents as well as adding new topics to the dataset in the future. At the end of the script, I had 6 folders named after the topics that I chose containing about 23,000 .txt files. This then acted as my dataset for this project. Once I have obtained my dataset, I decided to divide my dataset into training and testing. In this regards, I can use the training dataset to train my Doc2Vec model and to use my testing dataset to evaluate the performance

of my classification model. I will discuss this in more detail in the testing chapter of this report.

### 3.3. Text Pre-processing

Once the dataset was gathered, a mechanism was needed in order to “clean” the text that was to be fed into the software system. There are multiple techniques that can be used to pre-process text and these include (Ganesan, 2019):

- Lowercasing
- Stemming
  - “Chopping off” the ending of words in the hopes of obtaining the root word. For example, “-ing”, and “-ed” would be chopped off from words such as “ending”, “ended” to get the word “end”.
- Lemmatization
  - Similar to stemming such that the goal is to get the root word. However where it differs is that it also encompasses the special cases. For example the word “better” would be reduced to the word “good”.
- Stop-word removal
  - Removal of words that do not carry any a lot of meaning. These words/letters include “a”, “I”, “is”, “are” etc.
- Normalization
  - Transforming text to a standard format. Different situations may call for different format standards.
- Noise removal
  - Removing characters or text that is unwanted. Example of this are HTML or XML tags, “hashtag” symbol in tweets etc.

The techniques above do not have to be implemented together and the techniques used will depend on the situation. In my case I used the Gensim library that provided Word2Vec and Doc2Vec functionalities. The library also included text cleaning functions. I used the function called “`simple_preprocess()`” which converted all the words to lowercase, removed HTML tags, numbers, stop-words and all

punctuations. Once all of that was done, the function tokenized the text, or in other words, the text became an array/list of words. For example the text: “I was too <a>little</a> to reach 50cm above me, oh well!” converted to: ['was', 'too', 'little', 'to', 'reach', 'cm', 'above', 'me', 'oh', 'well']. The main reason why the text is tokenized was that the “TaggedDocument” class (the class that represents documents going to be used for the gensim Doc2Vec model) expected the text to be a list or array format (token format). By tokenizing the text, I simultaneously performed normalization of the text by converting all text fed into my program to token format. These text cleaning techniques used by Gensim proved to be sufficient for my use in the project.

### 3.4. Doc2Vec & Word2Vec Implementation

As I have stated before, I used the Gensim library as my Doc2Vec and Word2Vec implementation of the project. This library provided all the functionality that one could possibly perform with a Doc2Vec and Word2Vec model. The main reason why I chose to use the gensim library rather than creating my own Doc2Vec and Word2Vec model or to use other tools such as TensorFlow was because gensim was built to focus on topic modelling and to process plain text. As well as that, the library has been optimised to perform with efficiency in mind. For example, the training corpus does not need to be stored in RAM for training to take place and makes use of optimised math calculations in order to perform its algorithms. As well as that, trained models can be saved in the computer’s hard drive and loaded back into the program (Řehůřek, 2009).

As the gensim library provides a large array of processes you can perform with a Doc2Vec/Word2Vec model, I only made use of the functionality that was necessary for my software application by creating a wrapper class around the classes provided by the gensim library. The purpose of creating a wrapper class for the already existing gensim classes was to limit what functions that could be called onto the Doc/Word2Vec class. The code snippet below (Code 3) is my Doc2Vec wrapper class. In that class, I only made available the following functionality:

- Train the model
- Obtain the vector for a particular document
- Infer a vector for a new and unseen document
- To get the labels of all documents fed into the model
- Save a trained model
- Load a persisted model from disk

```

from gensim.models import Doc2Vec
import logging
import multiprocessing
import numpy as np

logging.basicConfig(format='%(asctime)s : %(levelname)s : %(message)s', level=logging.INFO)

class D2V:

    def __init__(self):
        cores = multiprocessing.cpu_count()
        self.__model = Doc2Vec(vector_size=300, min_count=2, epochs=70, workers=cores - 1)

    def train_model(self, documents):
        try:
            self.__model.build_vocab(documents)
            self.__model.train(documents, total_examples=self.__model.corpus_count,
                               epochs=self.__model.epochs)
            return 1
        except Exception as e:
            print(str(e))
            return -1

    def infer_new_document(self, doc):
        new_document = np.array(self.__model.infer_vector(doc), ndmin=2)
        return new_document

    def get_doc_vec(self, identifier: str):
        return self.__model.docvecs[identifier]

    def get_labels(self):
        return list(self.__model.docvecs.doctags.keys())

    def save_model(self, save_path):
        self.__model.save(save_path)

    def load_model(self, load_path):
        self.__model = Doc2Vec.load(load_path)
        print("Loaded model:\n-----")
        print("Vector size:\t" + str(self.__model.vector_size))
        print("Epochs:\t" + str(self.__model.epochs))

    def refresh(self):
        del self.__model
        self.__init__()

```

Code 3: My Doc2Vec wrapper class

In the context of my program, Doc2Vec could alternatively be viewed as a vector factory. It took a piece of text as its input and then produced a vector representation of that particular piece of text. In more accurate terms, the Doc2Vec model took in an iterable object of “TaggedDocument” objects as input and the model used these

objects during the training process. The TaggedDocument object needed a tokenized version of the text as well as a label for that text. This iterable object could be a list, an array or any object that could be looped. During the training process, the vectors for all the documents passed into it was formed. A vector for a particular document could simply be selected by passing in the label for the document. As discussed earlier in the research chapter of this report, the Doc2Vec model could also create the vector for unseen documents. This proved particularly useful when the user typed in a new piece of text. That text was then passed into the trained Doc2Vec model where a vector for that text was then produced (pre-processing would have been performed for that piece of text of course). The vectors were then used as input to the classifier. I will discuss this in more detail in the “Classifier” section.

The below code (Code 4) is the class I wrote for my Word2Vec class (“W2V”). This class is smaller than my Doc2Vec class as Word2Vec plays smaller part in my project. As my project’s main goal was to classify text, I initially intended to not have Word2Vec a part of my application as I did not intend to perform any word processes or similarity checking and that Doc2Vec performed the actual vectorization of entire documents. However, it was recommended to me to include Word2Vec in order to demonstrate its functionality. So I decided to add a smaller feature in my application where users could enter a word and the program would produce a list of similar words (within the documents provided during training) to that entered word.

```

from gensim.models import Word2Vec
import multiprocessing
import logging

logging.basicConfig(format="%(levelname)s - %(asctime)s: %(message)s", datefmt= '%H:%M:%S',
                    level=logging.INFO)

class W2V:

    def __init__(self):
        cores = multiprocessing.cpu_count()
        self.__model = Word2Vec(iter=5, min_count=10, size=300, workers=cores - 1, window=10)

    def train_model(self, sentences):
        try:
            self.__model.build_vocab(sentences)
            self.__model.train(sentences, total_examples=self.__model.corpus_count,
                              epochs=self.__model.epochs)
            return 1
        except Exception as e:
            print(str(e))
            return -1

    def get_similar_words(self, word: str):
        return self.__model.wv.most_similar(positive=word)

    def refresh(self):
        del self.__model
        self.__init__()

```

*Code 4: Word2Vec wrapper class*

In my Word2Vec class above, I only make available two main functionality of the gensim Word2Vec class: training the model and finding similar words using the “get\_similar\_words()” function. That function took in a word as a string format and sought for words with the largest cosine similarity to that word. The cosine similarity is the mathematical way of determining the closeness of two vectors. It should be noted though that the input word must also be part of the vocabulary that was stored in the trained Word2Vec model. The function then returned a list of words as well as their cosine similarity to the input word.

### 3.5. Classifier

In order to perform the actual classification process of the project, it was necessary to have a machine learning model in place, in particular, classification algorithms. There is a large number of classifiers available such as logistic regression, Naïve Bayes, k-nearest neighbours etc. However because the project title focused on neural networks, I used the neural network classifier for this project. However I coded the application so that it did not limit the classifier to that of a neural network. I will discuss this deeper in the design patterns section.



While I did create my own neural network as part of my research of the classifier, I ultimately decided to use the “Scikit Learn” library for its implementation of the neural network classifier (Buitinck, et al., 2013). The main reason why I chose to utilise the scikit learn library is due to its simplistic interface to the main functionality. The creators of the library are also seen to be experts in the field of machine learning and therefore it increased my confidence in the performance of the classifiers. Another great aspect about using the scikit learn library was that it made it very accessible to modify the parameters into the classifiers. In my case, I could change what activation function to use in the hidden layer of the neural network, the learning rate, the amount of training iterations to perform etc. The parameters could be adjusted by changing the values passed in through the constructor when creating the neural network model object.

Very similar to my approach with the gensim models, I decided to create a wrapper class of the necessary class. In my case, I made use of the scikit learn class called “MLPClassifier” which was part of the “neural network” package of the library. I created a class called “NeuralNetwork” that wrapped around the scikit learn class in order to limit access to it.

```
class NeuralNetwork(Classifier):
    def __init__(self):
        self.__model = MLPClassifier(activation='logistic', learning_rate="constant",
learning_rate_init=0.001)

    def train(self, x, y):
        self.__model.fit(x, y)

    def predict(self, x):
        val = self.__model.predict_proba(x)

        return val
```

*Code 5: Neural Network wrapper class*

In my approach, I only made available the training and predict functions. The name of the functions are self-explanatory in themselves. I should point out that I instead of calling the typical “predict()” function of the MLPClassifier class, I used the “predict\_proba()” function. By doing so, it calculated the probability that a

certain class was the correct guess. I felt that it was more appropriate to display all the topics and their probabilities of correctness rather than outputting a single topic.

The inputs to the classifier were document vectors. Another input that had to be included specifically for training were the topics corresponding to each document vector. However the string version of the topic could not be passed in to the classifier. The topic string must also be converted to a numerical format.

```
def add_topic(self, t: str):
    if t not in self.__topics:
        self.__topics.append(t)

def get_topics(self):
    return self.__topics

def get_topic_vector(self, t: str):
    topic_vec = list()
    for topic in self.__topics:
        if t == topic:
            topic_vec.append(1)
        else:
            topic_vec.append(0)

    return topic_vec
```

*Code 6: Functions in the DataStorer class that stores the topics as well as converts the topics to one-hot representation*

I found inspiration from my research of Word2Vec where the input word to the model were one-hot vectors. I decided to also use a one-hot encoding of the topics. For example, if the topics were “BUSINESS”, “SCIENCE”, and “TECH”, the one-hot vector of the topic “SCIENCE” would be [0, 1, 0] (The two zeros represents the business and tech topics). The previous code snippet (Code 6) highlights my code for creating one-hot vectors of the stored topics. During the training process of the classifier, the document vectors as well as their corresponding topic vectors are passed in. Once the classifier was done training, it was ready to perform the actual classification work. When a user entered a piece of text (text that was not encountered previously by the program) the trained Doc2Vec model created a vector representation of it. This vector was then passed into the trained classifier so that it would classify the unseen text.

## 3.6. Software Architecture & Design Patterns

When preparing for my implementation, I played with the thought that my tool could be used in a business setting or environment. This then motivated me to incorporate certain qualities with a large focus on extensibility and maintainability. Python is a dynamically-typed programming language, meaning that types are not as enforced as much compared to statically types languages such as Java. With this thought in mind, it was not necessary to define interfaces and other abstract classes. However I wanted to implement them anyways as I have an interest in software development and its best practices. I did so by using the “abc” package in python which provided mechanisms to creating interfaces in Python. In the previous sections I talked in detail the main components of the software application. I will now be discussing the design aspect of the project and how I joined the components together.

### 3.6.1. MVC (Model View Controller Architectural Pattern)

The architecture of software can be seen as the skeleton of the program. It defines how the system would be organised as well as the relationships between the components within that system (Eeles, 2006). For this project I decided to utilise the MVC architectural pattern. MVC is particularly useful when a graphical user interface is involved in the software system (Brainvire, 2016). This architectural pattern divides up the system in 3 main sections:

1. The model
2. The view
3. The controller

The model is the section that holds the “business logic” of the system. This is where the main processing of data takes place, or in other words, the brain of the software system. The view is the part of the system that displays the data as well as the interaction point of the user to the application. The controller section deals with connecting the view with the model. Whenever data is entered into the program by the user, the controller will process the data and will pass it on to the appropriate

model component. The controller will also send data produced by the model to the appropriate view component or GUI. One of the biggest reasons why I implemented the MVC pattern was that it helps separate the system's concerns. Setting roles for different aspects of the system helps in the organisation of the overall system, making the project easier and clearer to approach and to work with. MVC also modularises the entire system, therefore making it easier to extend in the future as well as using parts of the system for other already existing systems and in effect fulfilling what I wanted for my application to be in terms of extensibility and maintainability

The "user\_interfaces" package contains the GUI classes and is treated as the view part of the architecture. The "business\_logic" package contains both the controller and model components. The "managers" package contains the Word2Vec and Doc2Vec managers which handles the data passed into the program and called the appropriate classes to further process the data. This package is the controller part of the architecture. Multiple packages inside the "business\_logic" package acts as the model part of the system. Examples of the models are the "models" package, "reader" package and the "text\_processor" package.

### 3.6.2. Strategy

The intent of the strategy design pattern is to allow the interchange of algorithms and to abstract these algorithms by means of an interface (SourceMaking, 2019). The strategy design pattern is a good example of the "open-closed principle" which is part of the SOLID design principles of software. The open-closed principle states that software entities should be open for extension while at the same time, closed to modification (Janssen, 2018). This improves the maintainability and the extensibility of the application.

In my code base, I made use of the strategy design pattern more than once. The first instance is with the file readers. Currently I only have one file reader in my program which is a reader ("TxtReader" class) that manipulates and takes in ".txt" files. However, I envisage my program to have the capability of reading in files contained in different types of folders such as zip files, rar files etc. In order to accommodate

the possibility of change, I decided to utilise the strategy design pattern. Different folder types means different manner of reading in files, yet the variation can be encapsulated using an interface. Below is the “Reader” interface which abstracts a typical file reader.

```
import abc

class Reader(metaclass=abc.ABCMeta):

    @abc.abstractmethod
    def add_path(self, file_path): pass

    @abc.abstractmethod
    def clear_paths(self): pass

    @abc.abstractmethod
    def process_text(self, text): pass

    @abc.abstractmethod
    def yield_line(self): pass

    @abc.abstractmethod
    def yield_documents(self): pass
```

*Code 7: Reader interface*

The Reader interface defines what functionality a typical reader should have. Readers that implement this interface will have the same functionality yet their implementations may differ. Due to the abstraction of these readers, the dependency between the code using the readers and the readers themselves are loose as very little is known about the concrete readers.

The second instance I used the strategy pattern was with the text processors. My thinking behind this was that in the future, someone may want to add a set of text pre-processing techniques that will differ with what I have currently. As I discussed in the text pre-processing section, different text pre-processing techniques depends on the situation and because of this fact, I wanted to make way for the possibility of variance. Here is the text processor interface:

```
import abc

class TextProcessor(metaclass=abc.ABCMeta):

    @abc.abstractmethod
    def process_text(self, text): pass
```

*Code 8: TextProcessor interface*

The third instance of the strategy was with the classifier. There is a possibility that multiple classifiers could be used for the tool. The below is the code for the classifier interface:

```
import abc

class Classifier(metaclass=abc.ABCMeta):

    @abc.abstractmethod
    def train(self, x, y): pass

    @abc.abstractmethod
    def predict(self, x): pass
```

*Code 9: Classifier interface*

### 3.6.3. Factory Method

The main reason to use the factory method is to defer instantiation and the creation of objects (SourceMaking, 2019). The factory method encapsulates how an object is created. In other words, the client code knows very little about the specific objects being created or how it is being created. The client code only needs to know what type of object it is going to obtain. Due to the fact that the client code and the concrete classes are divided by a factory class, the relationship between these two are loose, therefore making it easier to extend in the future.

For this project, I decided to join my factory method with my strategy pattern. I felt that the factory method would help in creating the proper concrete strategy class and at the same time, making it easier to add new strategy classes in the future. I made use of the factory method in the same places the strategy pattern was implemented.

The first area I used the factory was with my reader classes. Just as I stated in the strategy design pattern section, new readers could be added in the future to handle

different folder types. By adding in the factory method, it decouples the dependency between the manager classes (“D2VManager & “W2VManager”) and the concrete reader (“TxtReader”), therefore making it easier to add new readers without modifying too much code. The below code is the ReaderFactory class (Code 10) and the code where the factory was used in the Doc2Vec manager constructor (Code 11):

```
from .txt_reader import TxtReader
from .reader import Reader

class ReaderFactory:

    TEXT = "TEXT"

    @staticmethod
    def create_reader(reader_type: str, manager_type: str) -> Reader:
        reader = None

        if reader_type == 'TEXT':
            reader = TxtReader(manager_type)

        return reader
```

*Code 10: ReaderFactory class*

```
class D2VManager(Publisher):

    def __init__(self, data_retriever: DataRetriever):
        self.__observers = list()
        self.__model = D2V()
        self.__file_reader = ReaderFactory.create_reader(ReaderFactory.TEXT, "D2V")
        self.__classifier = ClassifierFactory.create_classifier(ClassifierFactory.NEURAL_NET)
        self.__data_handler = DataStorer()
        self.__info_retriever = data_retriever

        self.__state = dict()
```

*Code 11: D2VManager using a factory class to obtain the file reader object*

If a new reader were to be introduced, the only change to make is to add a new “if” condition within the “create\_reader( )” method of the factory method.

I used to the factory method in all other locations where the strategy is used with the exact same mindset that I had for the readers. I utilized the pattern in text pre-processor and classifier classes. Below are the factory classes for them:

```

from .gensim_processor import GensimTextProcessor
from .text_processor import TextProcessor

class ProcessorFactory:
    @staticmethod
    def create_processor(processor_type: str) -> TextProcessor:
        processor = None

        if processor_type == "GENSIM":
            processor = GensimTextProcessor()

        return processor

```

Code 12: ProcessorFactory class

```

from .classifiers import *

class ClassifierFactory:
    NEURAL_NET = "NEURAL_NETWORK"
    KNN = "KNN"

    @staticmethod
    def create_classifier(classifier_type):
        classifier = None

        if classifier_type == "NEURAL_NETWORK":
            classifier = NeuralNetwork()
        elif classifier_type == "KNN":
            classifier = KNearestNeighbour()

        return classifier

```

Code 13: ClassifierFactory class

### 3.6.4. Command

The intent of the command design pattern is to encapsulate requests as objects and therefore treating requests like objects such that they can be parameterized, added to queues or stored in other data structures so that they could be executed later. By doing so, operations could be undone (SourceMaking, 2019). One of the greatest things about the command design pattern is that the client or in my case, the GUI classes do not know anything about the business logic. The GUI's do not call any of the functions in the business logic directly nor do they know the exact functions/components that resides in the business layer. The only thing the client/GUI knows is the request it wants the business logic to consume. It will just have to trust that their request will be handled appropriately by the logic of the application.



I made use of the command design pattern to decouple the GUI layer with the business logic layer of the application. Any GUI with different window widgets and different styling can be plugged into my program and interact with the business logic layer as long as it utilises the program's "Invoker". The invoker makes available the different commands that a GUI can make to interact with the program. In my implementation, the invoker stores the different commands in a python dictionary where each command is mapped to a certain key tag. The invoker selects and triggers a specific command depending on what function/method was called in the invoker itself. Below is a snippet of my invoker class. As the class is quite large, I only included a couple of the function calls.

```
class Invoker:

    def __init__(self):
        self.__commands = dict()

    def store_command(self, name, command):
        self.__commands[name] = command

    def add_directory_doc2vec(self):
        """
        Triggers the action to add a file path to a directory.
        :return: None
        """
        command = self.__commands["ADD_DIR_D2V"]
        command.execute()

    def add_dataset_doc2vec(self):
        command = self.__commands["ADD_DATASET_D2V"]
        command.execute()

    def load_doc2vec(self):
        command = self.__commands["LOAD_D2V"]
        command.execute()
```

*Code 14: Invoker class*

Command objects are stored in a dictionary through the "store\_command()" function. In order to trigger a piece of functionality in the business logic layer, an invoker method is called and this in turn triggers the appropriate command. I also included a snippet of the function in my GUI class that plugs itself to the program's business logic (Code 15).

```
def set_button_commands(self, button_listener: Invoker):
    """
    Sets the listener for button events.
    :param button_listener: The object that will handle the button presses.
    :return: None
    """
    self.add_directory.configure(command=button_listener.add_directory_doc2vec)
    self.add_dataset.configure(command=button_listener.add_dataset_doc2vec)
    self.load_doc2vec_model.configure(command=button_listener.load_doc2vec)
    self.train_doc2vec.configure(command=button_listener.train_doc2vec)
    self.train_classifier.configure(command=button_listener.train_classifier)
    self.classify_new_document.configure(command=button_listener.classify_doc)
    self.add_w2v_directory.configure(command=button_listener.add_word2vec_folder)
    self.train_word2vec.configure(command=button_listener.train_word2vec)
    self.search_similar.configure(command=button_listener.search_similar_words)
```

*Code 15: Linking the invoker object with the GUI*

The “set\_button\_commands()” function assigned the invoker class to handle the different mouse click events of my program. A reference to the invoker object is passed into the function. Each of the buttons on the GUI are configured to call the appropriate functions in the invoker object. I have designed the program so that it is not limited to function with a GUI. In theory, any piece of software can make use of the core functionality of my program as long as it utilises the program’s invoker class. The invoker class can also be viewed as my application’s API.

```

import abc
from business_logic.commands.command import Command

class Command(metaclass=abc.ABCMeta):

    @abc.abstractmethod
    def execute(self): pass

class AddDirectoryCommand(Command):

    def __init__(self, receiver):
        self.__receiver = receiver

    def execute(self):
        self.__receiver.add_directory()

class AddDatasetCommand(Command):

    def __init__(self, receiver):
        self.__receiver = receiver

    def execute(self):
        self.__receiver.add_dataset()

```

*Code 16: A subset of the Command classes*

The above code (Code 16) displays the Command interface that all requests/commands will implement. The snippet also includes some of the concrete command classes that implement the interface. These concrete commands have a reference to the actual class (receiver) within the business logic layer that would handle the called request. The execute method of each command calls the appropriate method in the receiver.

### 3.6.5. Observer

The purpose of the observer design pattern is to notify dependent objects (observers) of state changes within the main object (subject). The patterns is mainly used when there is a one-to-many relationship between objects (SourceMaking, 2019). A good way of knowing whether or not to use this pattern is when there is “Subscriber – Publisher” relationship in the system. The observer is very useful in architectures such as MVC as concerns and roles and separated.

For my application there is just a one-to-one relationship existing which resides between the GUI class and the manager classes. However what is useful about the observer is that it provides a facility to add new observers in the future. New observers can just register to the subject in order to receive notifications from it. The main use of the observer in my program is to pass data from the Doc2Vec/Word2Vec managers to the GUI while simultaneously maintaining loose coupling between the two classes (as the pattern makes use of interfaces). Due to this reason, new GUI's or other type of receivers can once again plug themselves in by extending/implementing the "Observer" interface and to provide an implementation of the "update()" function. The managers on the other hand extend/implement from the "Publisher" interface and therefore must provide implementations to registering new observers as well as notifying the registered observers.

```
import abc

class Publisher(metaclass=abc.ABCMeta):

    @abc.abstractmethod
    def attach(self, observer): pass

    @abc.abstractmethod
    def notify_observers(self): pass

class Observer(metaclass=abc.ABCMeta):

    @abc.abstractmethod
    def update(self, args): pass
```

*Code 17: The participants of the Observer design pattern*

```

def update(self, args):
    state = args[D2VResultKeys.STATE]

    if state == D2VManagerStates.ADD_DIR:
        self.__output_selected_directory(p=args[D2VResultKeys.FILE_PATH],
        topic=args[D2VResultKeys.TOPIC],
        files=args[D2VResultKeys.FILES])
        self.train_doc2vec["state"] = "normal"
    elif state == D2VManagerStates.ADD_DATASET:
        self.__display_dataset()
    elif state == D2VManagerStates.LOAD_MODEL:
        self.__model_loaded(args[D2VResultKeys.TOPICS])
    elif state == D2VManagerStates.TRAIN_D2V_STATUS:
        if args[D2VResultKeys.STATUS] == "SUCCESS":
            self.__display_d2v_training_status(1)
        elif args[D2VResultKeys.STATUS] == "ERROR":
            self.__display_d2v_training_status(2)
    elif state == D2VManagerStates.TRAIN_CLASSIFIER:
        messagebox.showinfo("Training Complete", "Successfully trained classifier")
    elif state == D2VManagerStates.CLASSIFIER_RESULT:
        self.__output_results(args[D2VResultKeys.TOPICS], args[D2VResultKeys.RESULTS])
    elif state == W2VManagerStates.W2V_FILES:
        self.__display_w2v_directory(args[W2VResultKeys.FILES])
    elif state == W2VManagerStates.TRAIN_W2V_STATUS:
        if args[W2VResultKeys.STATUS] == "SUCCESS":
            self.__display_w2v_training_status(1)
        elif args[W2VResultKeys.STATUS] == "ERROR":
            self.__display_w2v_training_status(2)
    elif state == W2VManagerStates.SIMILAR_WORDS:
        self.__display_similar_words(args[W2VResultKeys.WORDS])
    else:
        print("main_gui: Invalid state passed!")

```

Code 18: The update function found in the GUI class. Handles state changes in the manager classes.

The above function (Code 18) is located in the GUI class. It receives the data that was sent out by the two managers and will handle the data depending on the manager state.

```

def add_directory(self):
    filepath = filedialog.askdirectory(initialdir=".")

    if filepath == '':
        print("No file was selected")
    else:
        split_path = filepath.split("/")
        topic = split_path[len(split_path) - 1]
        self.__data_handler.add_topic(topic)
        files = self.__file_reader.add_path(filepath)
        print("D2VManager: stored topics: " + str(self.__data_handler.get_topics()))

        self.__state.clear()
        self.__state[D2VResultKeys.STATE] = D2VManagerStates.ADD_DIR
        self.__state[D2VResultKeys.TOPIC] = topic
        self.__state[D2VResultKeys.FILE_PATH] = filepath
        self.__state[D2VResultKeys.FILES] = files

        self.notify_observers()

```

Code 19: An example of state changes in the D2VManager class.

The above function (Code 19) resides in the D2VManager class. It adds a directory location to the program. It also retrieves the topic associated with that directory as well as the file names within that directory. The file names and topic are stored in a python dictionary. Using the “notify\_observers( )” function, the dictionary is passed to the observers (see Code 20).

```
def notify_observers(self):  
    for observer in self.__observers:  
        observer.update(self.__state)
```

*Code 20: Function that notifies the observer of state changes*

## Chapter 4: Product Evaluation

Upon completing my product, the initial action I took was to see if it classified correctly and asked my peers to perform user testing. We ran a couple of executions of the program using a very small dataset of only 36 files. This smaller dataset consisted of Wikipedia articles divided into the topics: “Literature”, “Food”, “Finance” and “Technology”. Each of the folders only had approximately eight to ten files in them. For the Doc2Vec model, these were the following parameters that I set:

- Vector\_size = 3 (size of the document vectors)
- Epochs = 50 (training iterations)

I entered this text I found on the intel website:

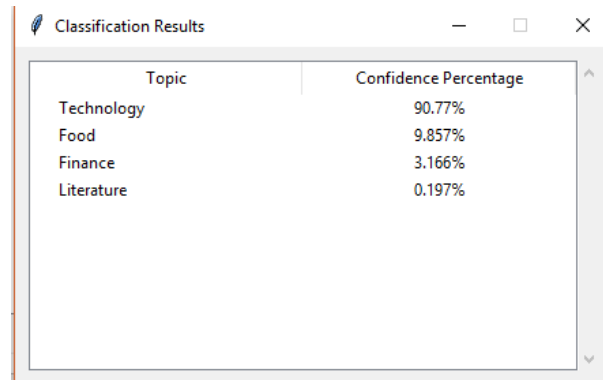
*“Computers with the new 9th Generation Intel® Core™ desktop processors are packed with performance for mainstream and competitive gamers. With up to 8 cores, 16 threads, 5.0 GHz, and 16 MB cache the 9th Generation Intel® Core™ desktop processors are built for gaming. Overall system performance is boosted up to 15% vs. 8th Gen Intel® Core™ processors and you can Game + Stream + Record with up to 40% more FPS on Total War: WARHAMMER II vs. a 3 yr. old PC” - (Intel, 2019)*

This was the result of my program with the Doc2Vec parameters I had above:

Topic	Confidence Percentage
Technology	35.199%
Food	18.511%
Literature	8.384%
Finance	3.058%

I was quite surprised that it guessed the topic correctly the first time around especially when I had a tiny dataset! However I felt that the “Food” category should not have had a large confidence percentage as the text above did not have any correlation to food at all.

I wanted to see for myself whether or not increasing the training iteration value would increase the accuracy of the program. So I increased the iteration to 1000 (due to the fact that my dataset only has 36 files). These were the results:

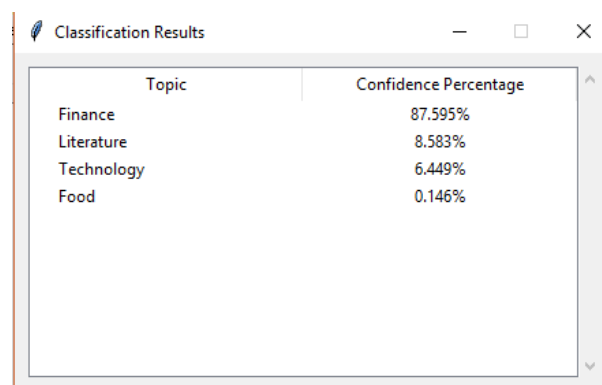


Topic	Confidence Percentage
Technology	90.77%
Food	9.857%
Finance	3.166%
Literature	0.197%

With the training iteration increased, the confidence of the program's guessed also increased. However the food topic still came second place which was not optimal. The finance topic though increased in confidence which was getting closer to was I wanted. I wanted to try a different piece of text with the same number of training iterations. I entered this text:

*"A corporation is a legal entity that is separate and distinct from its owners. Corporations enjoy most of the rights and responsibilities that an individual possesses: enter contracts, loan and borrow money, sue and be sued, hire employees, own assets and pay taxes. Some refer to it as a 'legal person.'"* - (Kenton, 2018)

The result:



Topic	Confidence Percentage
Finance	87.595%
Literature	8.583%
Technology	6.449%
Food	0.146%

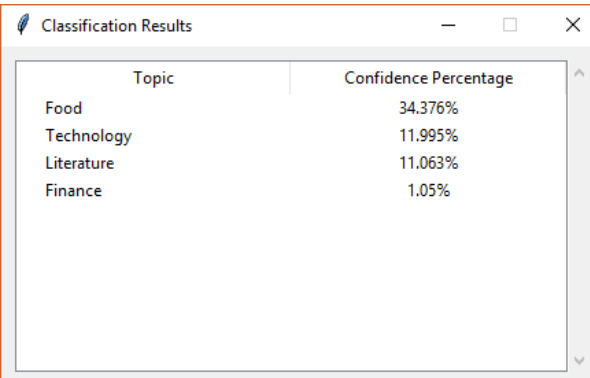
In this case the program guessed the topic correctly again. I was still quite shocked that it was classifying unseen text somewhat correctly with a tiny dataset. From this part of the evaluation, I learnt that increasing the training iteration does increase



accuracy of the model. I also came to experience the power of Doc2Vec as it demonstrated how it provided a satisfactory guess even with limited training data due to its effectiveness in detecting the context of text. However the smaller dataset means that there was still a strong likelihood that it would guess incorrectly if a piece of text that was harder to decipher was introduced. In this case here I entered a piece of text (quoted below) that had a small relationship with technology yet that text belonged to a news article belonging in the tech section.

*“van den Berg is a musician who performs wearing motion-tracking gloves and a full-body suit covered in sensors, which, during this SXSW performance, not only control a projection of a digital avatar that appears behind her, but also control nearly every instrument and effect in the music and her voice. As she moves across the stage, her avatar, floating in space, moves in sync. When she stretches her arms above her head, grains of audio slow to a grind and stutter. Every hand and body movement has cause and effect, crafting a pop-infused dreamscape that’s mesmerizing to watch.” - (Deahl, 2019)*

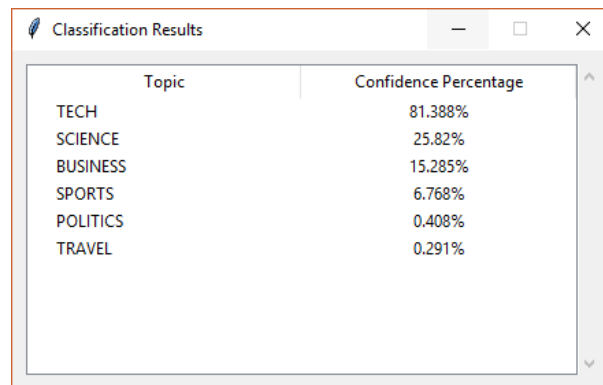
These were the results:



Topic	Confidence Percentage
Food	34.376%
Technology	11.995%
Literature	11.063%
Finance	1.05%

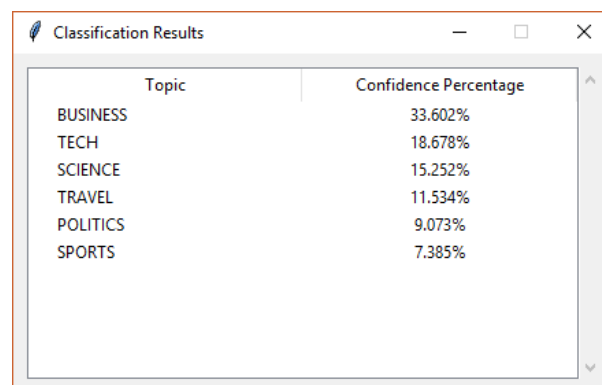
The results above demonstrate that the program failed to produce the correct topic which was the technology topic.

The next step I took was to use my larger HuffPost dataset where each folder had more than 2,000 files and see the performance of that dataset. The topics of that dataset were “POLITICS”, “BUSINESS”, “SCIENCE”, “TECH”, “TRAVEL”, “SPORT”. Due to the fact that I had a larger dataset, I decreased the training iteration down to 50 but with the same vector size. I used the same text from the Intel website and this was the result:



Topic	Confidence Percentage
TECH	81.388%
SCIENCE	25.82%
BUSINESS	15.285%
SPORTS	6.768%
POLITICS	0.408%
TRAVEL	0.291%

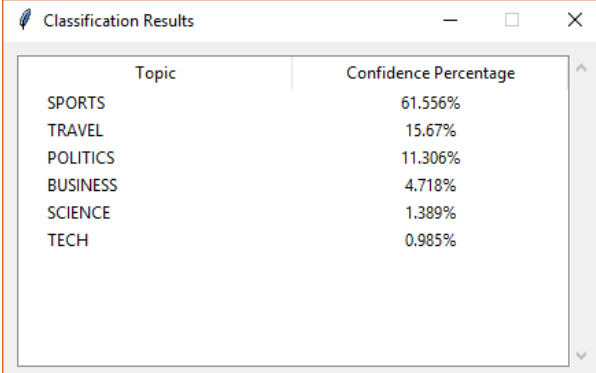
With these results, I felt that it gave more accurate results than the smaller dataset. Following on from the tech topic, science and business came just below tech. This made a lot of sense to me as the text did have tints of science and business jargon in it. I then tried the text that defines “corporation” and here was the result:



Topic	Confidence Percentage
BUSINESS	33.602%
TECH	18.678%
SCIENCE	15.252%
TRAVEL	11.534%
POLITICS	9.073%
SPORTS	7.385%

However a big disappointment that I encountered was the politics topic. I tried to enter even the most obvious of political texts yet the program refused to classify it being politics. For example this quote that I took from one new article:

*“President Donald Trump on Friday night announced he will nominate current Ambassador to Canada Kelly Knight Craft as his next United Nations ambassador. ‘Kelly has done an outstanding job representing our Nation and I have no doubt that, under her leadership, our Country will be represented at the highest level,’ Trump wrote on Twitter. The announcement came about an hour after the president met face-to-face with Craft along with Secretary of State Mike Pompeo and national security adviser John Bolton in the Oval Office to finalize the nomination, according to a source familiar with the conversation.” - (Panetta, et al., 2019)*



Topic	Confidence Percentage
SPORTS	61.556%
TRAVEL	15.67%
POLITICS	11.306%
BUSINESS	4.718%
SCIENCE	1.389%
TECH	0.985%

From the above results, we can see that the guesses are incorrect. There was very little references to sport and travel in that piece of text. I tried changing the parameters of the Doc2Vec as well as the classifier to see if it was a problem with the models. However the results did not change and the politics section still remained a problem. I then came to the conclusion that the dataset for the politics folder may have been the issue. I cannot be too certain as I had no way of validating whether or not the fault was in the actual text dataset. However from looking through a number of the files, I felt that the politics files were noisy. In other words, the files touched on different topics way too much from my point of view. It also had relatively small reference to actual political topics. For example, one file talked about Donald Trump's conversation with an Olympian athlete on Twitter while another talked about how the FBI scrutinized through a taxicab business. Just to re-iterate, I was not 100% sure why the political text could not be classified. After looking through the dataset however, I came to the conclusion that the documents I had saved for politics were not good enough for training.

After my initial execution tests, I wanted to discover the optimal parameters to the Doc2Vec and neural network such that they produce the best guess possible. However when performing this process, I discovered that there were too many parameters to tweak and with the time that I had left, there was not enough time for me to do this rigorously. From past testing, the program performed in a satisfactory manner so I decided to keep the parameters of the Doc2Vec constant. These were the parameters of the Doc2Vec model that I had:

- Vector\_size = 300
- Epoch = 50

I should also point out that at this point, I have removed the politics folder from both my testing and training datasets as it had been found to be faulty. I kept an excel spreadsheet to record the changes I made to the neural network parameters as well as the accuracy score that resulted from executing the program with those parameters. Here are some of the results I got:

activation	learning rate	learning_rate_init	solver	max_iter	accuracy
logistic	adaptive	0.0001	adam	500	75%
relu	adaptive	0.0001	adam	500	70%
tanh	adaptive	0.0001	adan	500	65%
logistic	constant	0.0001	adam	500	62%
logistic	constant	0.001	adam	500	69%
logistic	adaptive	0.001	adam	500	69%
relu	constant	0.001	adam	500	71%
relu	adaptive	0.001	adam	500	70%
tanh	constant	0.001	adam	500	62%
tanh	adaptive	0.001	adam	500	64%

From these results I obtained, it was clear to me that the first row is what I should use to set the parameters of the neural network in order to obtain the optimal accuracy.

## Chapter 5: Conclusion

### 5.1. Possible Improvements

Once I have completed the implementation of the project and performed the testing, I felt that the program performed in a level that I found satisfactory. Other than the politics topics, the program classified unseen text correctly (most of the time). However during the 7 months I had in working with this FYP I felt that there was not enough time to do what I wanted to do. During the process of researching and implementing my tool, I discovered that there was a lot of information to take in as well as a lot of different actions to perform in order to reach my end goal. It is only now when I have accomplished my project that I have the time to reflect upon the project as a whole and to look back on what I could have done better. If I am to work on the project again, I would do the following:

- **Perform more research on more machine learning terminologies and techniques.** There was a lot of machine learning terms that I was exposed to during this FYP. Yet I felt that I did not have enough time in understanding more of these terms. For example, I am yet to fully comprehend the meaning behind terms such as “cross validation”, “regularisation”, and “under & overfitting”. Even though I have a grasp on these terminologies, I do not have deep knowledge on these terms to be confident in explaining them. I also felt that I could have explored one or two more classification techniques and algorithms such as “K-Nearest Neighbour” and “Naïve Bayes” and I could have done a side by side comparison between these algorithms.
- **Perform deeper research on the practicality of Word2Vec/Doc2Vec.** During my research on Word2Vec & Doc2Vec I stumbled upon a paper that compared them to other word embedding techniques. It displayed a table of the accuracy score between the different techniques. However during that moment, I prioritised trying to understand the operation of the Word/Doc2Vec and therefore I did not look too much into the material. I felt that if I did do this research, it would have

helped me to appreciate Word2Vec more as well as the power that this word embedding technique has in capturing word semantics and relationships.

- **Create a more intricate software design of the application.** As I have stated earlier, I wanted to design my project with the idea that my tool could be used in a corporate scenario. Though I definitely have spent time planning the design of my project with the goal of incorporating quality attributes such as maintainability and extensibility, my program still has room for a better design implementation. The time I had for this FYP was quite limited so I was not in a position to create a detailed approach to my program design. If I had another opportunity and with more time, I would spend time creating sequence diagrams, CRC cards, class diagrams as well as incorporating more advanced yet appropriate design patterns such as the “memento”, “façade” and the “decorator”.
- **Plan and perform a more systematic approach to testing.** I believe that I could have done better in the testing area of my project also. Though I did perform user testing, I did not have enough time to properly evaluate the accuracy of my Doc2Vec model as well as the classification model by tweaking the different parameters. By visiting the scikit learn documentation on the neural network class, you would see a large list of different parameters to the neural network class. The timeframe I had could not allow me to adjust all of these parameters in order to reach the highest possible accuracy. However this work is by no means impossible and if I were to have another chance, I would work on the testing and evaluation more.

## 5.2. Learning Outcome

From the previous section I outlined what I would have done if I were to do this project again. I do believe that doing the above would definitely improve my project more. On the other hand, I also believe that I have fulfilled my goals with this project.

During the research part of my project, I had the wonderful opportunity of learning about neural networks and how it can be seen as the computer’s brain. I gained

knowledge on how the entire system worked both from a high level perspective to the intricate and mathematical low level view of it. I came to understand how neural networks learn from their mistakes by means of backpropagation and gradient descent and I was exposed to calculus and matrix operations that drive the training processes.

I found Word2Vec a very interesting piece of research. It was fascinating to find out how this technique captures and retains the meaning behind words as well as the relationships between words when creating word vectors. To add to the excitement, doing the exact same but with entire documents was a very interesting discovery. I never knew that this was something that computers and mathematics could achieve and even perform!

Of course I found it a joy in programming the application. I had the privilege of learning Python, a programming language that I had no experience before. Though it was a challenge learning the language alongside the new information that I was being exposed to, it was something that I enjoyed doing. I also have a big interest in software engineering and best practices and being able to incorporate design patterns and implementing software design principles was a pleasure.

Before doing the FYP I had little to no knowledge of machine learning and artificial intelligence. Now that I have accomplished my project, I can confidently say that I have increased my knowledge base with my new machine learning experience. From this moment on, whenever I encounter a piece of technology that makes use of machine learning or A.I, I would have an understanding as to how it functions. Especially in today's times, machine learning is becoming more prevalent and I believe that the knowledge that I have now would come in great use.

## References

Aggarwal, C. C. & Zhai, C., 2012. A Survey of Text Classification Algorithms. *Mining Text Data*, January. pp. 163-222.

Brainvire, 2016. *Six Benefits of Using MVC Model for Effective Web Application Development*. [Online]

Available at: <https://www.brainvire.com/six-benefits-of-using-mvc-model-for-effective-web-application-development/>

[Accessed 05 April 2019].

Buitinck, L. et al., 2013. *API design for machine learning software: experiences from the scikit-learn project*. s.l., s.n., pp. 108-122.

Deahl, D., 2019. *Electronic Music Has A Performance Problem, And This Artist Is Trying To Solve It*. [Online]

Available at: <https://www.theverge.com/2019/4/5/18277345/chagall-van-den-berg-performance-sensors-gloves-motion-tracking-suit>

[Accessed 06 April 2019].

Eeles, P., 2006. *What is a software architecture?*. [Online]

Available at:

<https://www.ibm.com/developerworks/rational/library/feb06/eeles/index.html>

[Accessed 05 April 2019].

Ganesan, K., 2019. *How to Perform Text Preprocessing for Machine Learning & NLP? | Kavita Ganesan*. [Online]

Available at: <http://kavita-ganesan.com/text-preprocessing-tutorial/>

[Accessed 15 January 2019].

Greene, D. & Cunningham, P., 2006. *Practical Solutions to the Problem of Diagonal Dominance in Kernel Document Clustering*, Pittsburgh: ICML.

Intel, 2019. *INTEL® CORE™ PROCESSOR FAMILY*. [Online]

Available at: <https://www.intel.co.uk/content/www/uk/en/products/processors/core.html>

[Accessed 03 February 2019].

Janssen, T., 2018. *SOLID Design Principles Explained: The Open/Closed Principle with Code Examples*. [Online]

Available at: <https://stackify.com/solid-design-open-closed-principle/>

[Accessed 10 February 2019].

Kenton, W., 2018. *Corporation*. [Online]

Available at: <https://www.investopedia.com/terms/c/corporation.asp>

[Accessed 05 February 2019].

Klein, B., 2011. *Backpropagation in Neural Networks*. [Online]

Available at: [https://www.python-course.eu/neural\\_networks\\_backpropagation.php](https://www.python-course.eu/neural_networks_backpropagation.php)

[Accessed 16 Oct 2018].

Korde, V. & Mahender, N., 2012. Text Classification and Classifiers: A Survey. *International Journal of Artificial Intelligence & Applications (IJAI)*, 3(2), p. 85.



McCormick, C., 2016. *Word2Vec Tutorial - The Skip-Gram Model*. [Online]  
Available at: <http://mccormickml.com/2016/04/19/word2vec-tutorial-the-skip-gram-model/>  
[Accessed 20 October 2018].

Mikolov, T., Chen, K., Corrado, G. & Dean, J., 2013. *Efficient Estimation of Word Representations in Vector Space*. Arizona, Proceedings of Workshop at ICLR.

Mikolov, T. & Le, Q., 2014. *Distributed representations of sentences and documents*. Beijing, JMLR.org, pp. 1188-1196.

Misra, R., 2018. *News Category Dataset*. [Online]  
Available at: <https://rishabhmisra.github.io/publications/>

Nielsen, M., 2018. *Using neural nets to recognize handwritten digits*. [Online]  
Available at: <http://neuralnetworksanddeeplearning.com/chap1.html>  
[Accessed 15 Oct 2018].

Panetta, A., Toosi, N., Johnson, E. & Griffiths, B. D., 2019. *Trump taps Kelly Knight Craft as U.N. ambassador*. [Online]  
Available at: <https://www.politico.com/story/2019/02/22/trump-taps-kelly-knight-craft-as-un-ambassador-1182282>  
[Accessed 27 February 2019].

Rashid, T., 2016. *Make Your Own Neural Network*. Germany: CreateSpace Independent Publishing Platform.

Řehůřek, R., 2009. *gensim: Introduction*. [Online]  
Available at: <https://radimrehurek.com/gensim/intro.html>  
[Accessed November 2019].

Řehůřek, R., 2014. *Doc2vec tutorial*. [Online]  
Available at: <https://rare-technologies.com/doc2vec-tutorial/>  
[Accessed 15 December 2018].

Rosenblatt, F., 1958. The Perceptron: A probabilistic model for information storage and organization in the brain. *Psychological Review*, 65(6), pp. 386-388.

SourceMaking, 2019. *Strategy Design Pattern*. [Online]  
Available at: [https://sourcemaking.com/design\\_patterns/strategy](https://sourcemaking.com/design_patterns/strategy)  
[Accessed 10 February 2019].

Stergiou, C. & Siganos, D., n.d. *Neural Networks*. [Online]  
Available at:  
[https://www.doc.ic.ac.uk/~nd/surprise\\_96/journal/vol4/cs11/report.html#References](https://www.doc.ic.ac.uk/~nd/surprise_96/journal/vol4/cs11/report.html#References)  
[Accessed 24 December 2018].

TensorFlow, 2018. *Vector Representations of Words*. [Online]  
Available at: <https://www.tensorflow.org/tutorials/representation/word2vec>  
[Accessed 17 October 2018].

## Image References

### UL LOGO:

[http://www.mmrrc.ul.ie/dotnetnuke/portals/6/UL\\_Logo\\_1ver.png](http://www.mmrrc.ul.ie/dotnetnuke/portals/6/UL_Logo_1ver.png)

### Fig. 1:

Pngimage.net. (2019). *Nerve cell image*. [online] Available at: <https://pngimage.net/nerve-cell-png-5/> [Accessed 8 Apr. 2019].

### Fig. 2:

Lucidarme, P. (2019). *Simplest perceptron update rules demonstration • Robotics, Teaching & Learning*. [online] Robotics, Teaching & Learning. Available at: <https://www.lucidarme.me/simplest-perceptron-update-rules-demonstration/> [Accessed 8 Apr. 2019].

### Fig. 3:

Perry, J. (2019). *Build an ANN with the Java language and Neuroph*. [online] Ibm.com. Available at: <https://www.ibm.com/developerworks/library/cc-artificial-neural-networks-neuroph-machine-learning/index.html> [Accessed 8 Apr. 2019].

### Fig. 4:

Jordan, J. (2019). *Neural networks: representation..* [online] Jeremy Jordan. Available at: <https://www.jeremyjordan.me/intro-to-neural-networks/> [Accessed 8 Apr. 2019].

### Fig. 5:

Chan Phooi M'ng, J. and Mehralizadeh, M. (2016). Forecasting East Asian Indices Futures via a Novel Hybrid of Wavelet-PCA Denoising and Artificial Neural Network Models. *PLOS ONE*, [online] 11(6), p.e0156338. Available at: [https://www.researchgate.net/publication/303744090\\_Forecasting\\_East\\_Asian\\_Indices\\_Futures\\_via\\_a\\_Novel\\_Hybrid\\_of\\_Wavelet-PCA\\_Denoising\\_and\\_Artificial\\_Neural\\_Network\\_Models](https://www.researchgate.net/publication/303744090_Forecasting_East_Asian_Indices_Futures_via_a_Novel_Hybrid_of_Wavelet-PCA_Denoising_and_Artificial_Neural_Network_Models) [Accessed 8 Apr. 2019].

### Fig. 6:

Samyazaf.com. (2019). *word2vec*. [online] Available at: <https://www.samyazaf.com/ML/nlp/nlp.html> [Accessed 8 Apr. 2019].

### Fig. 7:

Chablani, M. (2019). *Word2Vec (skip-gram model): PART 1 - Intuition..* [online] Towards Data Science. Available at: <https://towardsdatascience.com/word2vec-skip-gram-model-part-1-intuition-78614e4d6e0b> [Accessed 8 Apr. 2019].

### Fig. 8:

Khalid, S. (2019). *Understanding Word Embeddings*. [online] Towards Machine Learning. Available at: <https://towardsml.com/2018/06/12/understanding-word-embeddings/> [Accessed 8 Apr. 2019].

**Fig. 11:**

Irene (2019). *NLP 05: From Word2vec to Doc2vec: a simple example with Gensim*. [online] Night Café. Available at: <https://ireneli.eu/2016/07/27/nlp-05-from-word2vec-to-doc2vec-a-simple-example-with-gensim/> [Accessed 8 Apr. 2019].

**Fig. 12:**

Budhiraja, A. (2019). *A simple explanation of document embeddings generated using Doc2Vec*. [online] Medium. Available at: <https://medium.com/@amarbudhiraja/understanding-document-embeddings-of-doc2vec-bfe7237a26da> [Accessed 8 Apr. 2019].

## Table of Figures

Figure 1: A Neuron (PNGImage, 2019).....	6
Figure 2: The Perceptron (Lucidarme, 2019) .....	7
Figure 3: A feed forward neural network (Perry, 2019) .....	8
Figure 4: Matrix representation of the NN (Jordan, 2019) .....	9
Figure 5: Feed forward & Backpropagation (Chan Phooi M'ng and Mehralizadeh, 2016).....	10
Figure 6: Vector relationships (Samyazaf.com, 2019) .....	12
Figure 7: The skip-gram architecture (Chablani, 2019) .....	12
Figure 8: The CBOW architecture (Khalid, 2019) .....	13
Figure 9: The weight matrix representing the weights between the input and hidden layer of a hypothetical Word2Vec model .....	13
Figure 10: A one-hot representation of the input word "Computer". .....	14
Figure 11: Distributed Memory model (Irene, 2019).....	16
Figure 12: Distributed Bag of Words model (Budhiraja, 2019).....	17

## Dataset sources

Misra R. (2018) *News Category Dataset* [dataset], V.2, Kaggle, available:

<https://www.kaggle.com/rmisra/news-category-dataset/metadata>

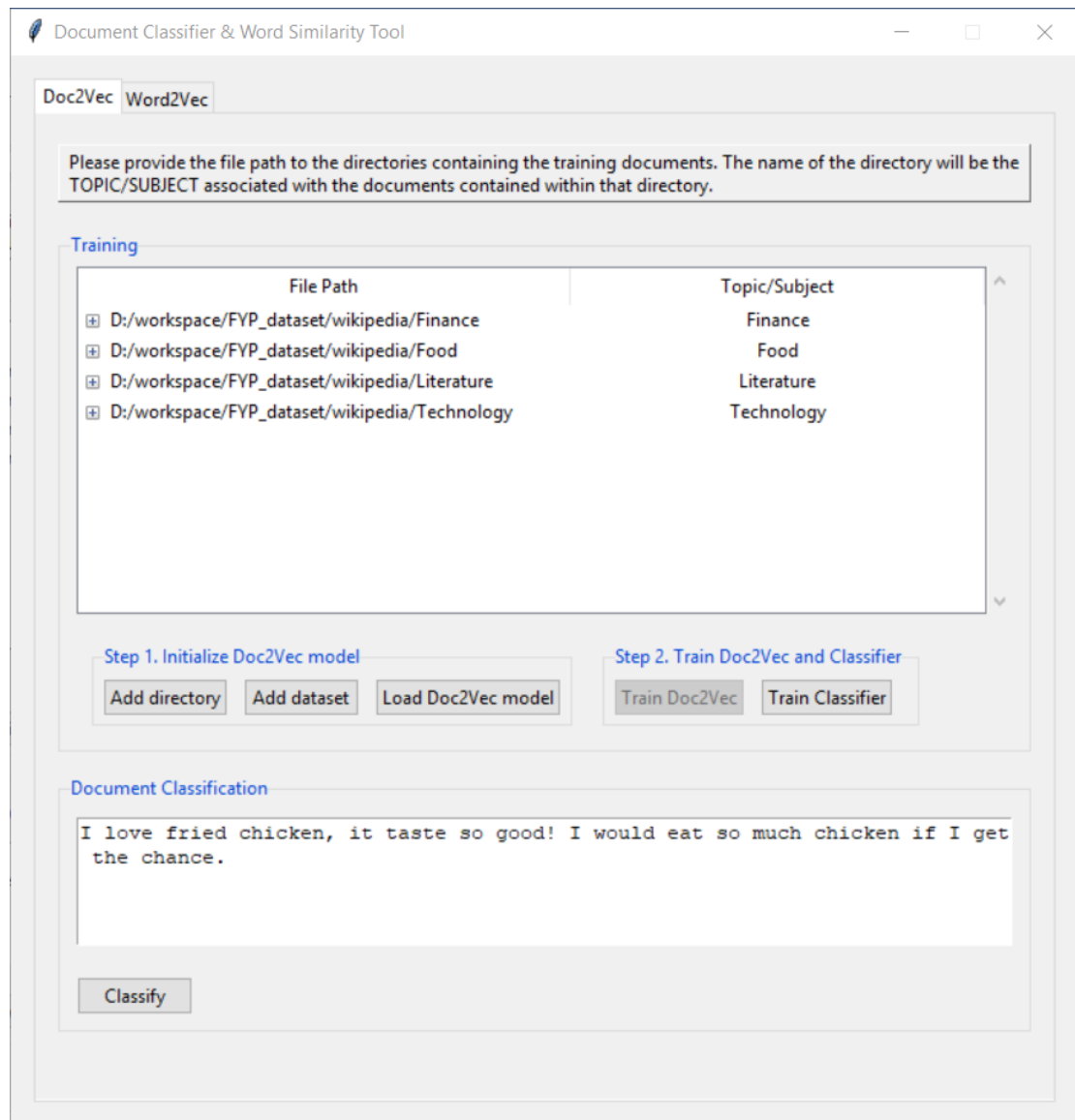
Greene, D. and Cunningham, P. (2006) *Practical Solutions to the Problem of Diagonal Dominance in Kernel Document Clustering* [dataset], Proc. ICML 2006, available:

<http://mlg.ucd.ie/datasets/bbc.html>

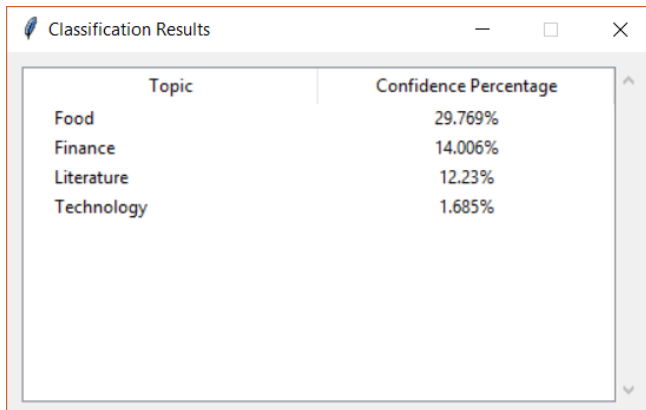
## Appendix

### GUI Snapshots

#### Doc2Vec Tool



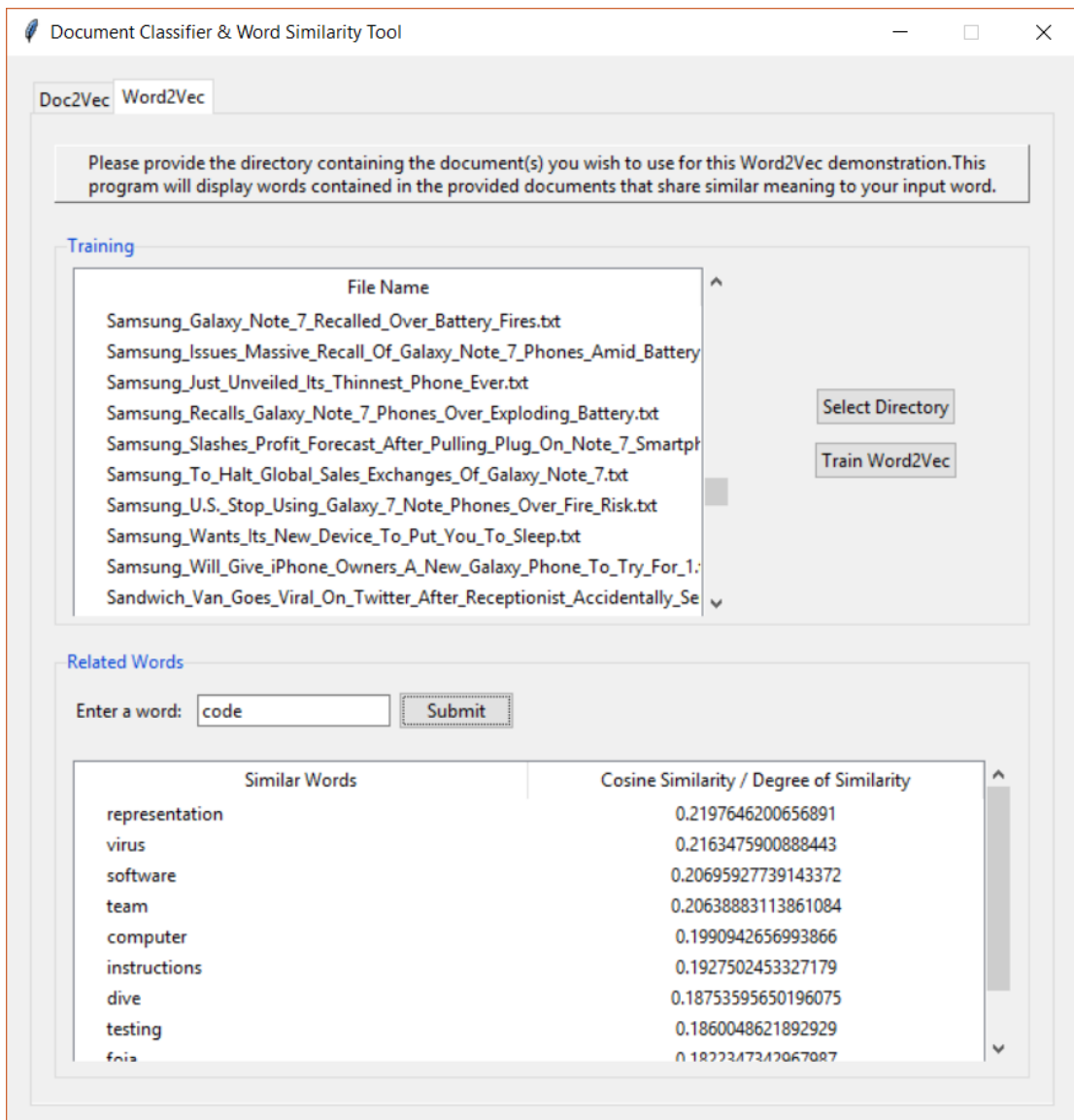
## Doc2Vec Results GUI



Classification Results

Topic	Confidence Percentage
Food	29.769%
Finance	14.006%
Literature	12.23%
Technology	1.685%

## Word2Vec tool



Document Classifier & Word Similarity Tool

Doc2Vec Word2Vec

Please provide the directory containing the document(s) you wish to use for this Word2Vec demonstration. This program will display words contained in the provided documents that share similar meaning to your input word.

Training

File Name
Samsung_Galaxy_Note_7_Recalled_Over_Battery_Fires.txt
Samsung_Issues_Massive_Recall_Of_Galaxy_Note_7_Phones_Amid_Battery
Samsung_Just_Unveiled_Its_Thinnest_Phone_Ever.txt
Samsung_Recalls_Galaxy_Note_7_Phones_Over_Exploding_Battery.txt
Samsung_Slashes_Profit_Forecast_After_Pulling_Plug_On_Note_7_Smartph
Samsung_To_Halt_Global_Sales_Exchanges_Of_Galaxy_Note_7.txt
Samsung_U.S._Stop_Using_Galaxy_7_Note_Phones_Over_Fire_Risk.txt
Samsung_Wants_Its_New_Device_To_Put_You_To_Sleep.txt
Samsung_Will_Give_iPhone_Owners_A_New_Galaxy_Phone_To_Try_For_1.
Sandwich_Van_Goes_Viral_On_Twitter_After_Receptionist_Accidentally_Se

Select Directory

Train Word2Vec

Related Words

Enter a word:

Similar Words	Cosine Similarity / Degree of Similarity
representation	0.2197646200656891
virus	0.2163475900888443
software	0.20695927739143372
team	0.20638883113861084
computer	0.1990942656993866
instructions	0.1927502453327179
dive	0.18753595650196075
testing	0.1860048621892929
foia	0.1822217217067087