Trinity College Dublin
Coláiste na Tríonóide, Baile Átha Cliath
The University of Dublin

School of Computer Science and Statistics

GUI Design and Results Visualisation of a Computational Linguistics Model of Word Sense Emergence

Eoin Brereton Hurley

Dr. Martin Emms April 17, 2023

Word Count: 11139

A Final Year Project Report submitted in partial fulfilment of the requirements for the degree of BA(Mod) in CSLL i.e. Computer Science, Linguitics & Language (French)

GUI Design and Results Visualisation of a Computational Linguistics Model of Word Sense Emergence

Eoin Brereton Hurley, BA(Mod) in CSLL i.e. Computer Science, Linguitics & Language (French)

University of Dublin, Trinity College, 2023

Supervisor: Dr. Martin Emms

Word Count: 11139

This Final Year Project is concerned with designing a Graphical User Interface for running and visualising the results of an Expectation Maximisation algorithm which infers the parameters of a model to detect diachronic word sense emergence. The generation of animations is facilitated and graphs may be displayed side-by-side. This allows an analyst wishing to run experiments using the model the ability to cross-reference different runs which is valuable for fine-tuning hyperparameters. A comparison of contemporary hyperparameter optimisation algorithms and applications is given before a discussion of the key challenges confronted when designing the GUI. The general requirements were successfully fulfilled and relevant further work on this project was suggested.

Contents

Abstract **i**

Chapter 1 Introduction **1**
1.1 Context: *Neologisms* & `EJ` Model 1
1.2 `EM`: Expectation Maximisation 3
1.2.1 Gaussian Mixture Model Example 4
1.2.2 Laplacian Mixture Model Example 6
1.2.3 Relation to `DynamicEM` 7
1.3 Issues Impeding Use of `DynamicEM` 10

Chapter 2 State of the Art **12**
2.1 Background 12
2.1.1 Development of the `EJ` model & `DynamicEM` algorithm 12
2.1.2 *Pseudoneologisms* 13
2.2 Hyperparameter Optimisation 15
2.2.1 Search Algorithms 16
2.2.2 Early-Stopping Methods 17
2.2.3 HPO Toolkits 19
2.3 Summary 20

Chapter 3 Design **21**
3.1 Overview of the Approach 21
3.2 Library Selection 21
3.3 Corpus Data 22
3.4 Use Case Scenario 23

Chapter 4 Implementation **25**
4.1 User-friendly Parameter Specification 25

4.2 `RInside` package 28
4.2.1 `RInside` directly within `Qt` 29
4.2.2 Visual Analysis 31
4.2.3 Blocking Issue 37
4.2.4 Multi-threading 37
4.3 Summary 38

Chapter 5 Evaluation **40**
5.1 Revisiting Requirements 40

Chapter 6 Conclusions & Future Work **42**
6.1 Future Work 42
6.1.1 Generic Parameter Specification 42
6.1.2 *Pseudoneologisms* 43
6.1.3 Extension of Features 44
6.2 Overall Conclusions 44

Bibliography **45**

List of Figures

1.1 Sample Graph of *Sense Probabilities* against *Year* for target *mouse* 3

2.1 An example of overfitting. The true relationship between x and y is a cubic relationship, but the training values have some noise added to y. A 15-degree polynomial fits the training data between, but it fails to capture the true relationship. 18

4.1 User-friendly Parameter Setting 26
4.2 File Dialogue Box prompting user to select graphs 35
4.3 Two Graphs Displayed Side-By-Side 36
4.4 Three Graphs Displayed Side-By-Side 36

6.1 Generic Parameter Setting Implementation Flowchart 43

List of Source Code Snippets

1 `App.cpp`: Error-checking for initial values of Sense Probabilities 27
2 `Animation.cpp`: Creating an Embedded `R` Instance in `C++` 29
3 `main.cpp`: Passing an Embedded `R` Instance to `Qt` Application 30
4 `App.cpp`: Running `R` Code from within `Qt` Application 31
5 `App.cpp`: Rendering File Dialogue Box to Prompt User to Select Animations 34

Chapter 1

Introduction

This Final Year Project is concerned with the development of a Graphical User Interface to assist analysts in using *Dr. Martin Emms* and *Dr. Arun Jayapal*'s Expectation Maximisation algorithm for inferring the parameters of their proposed model for detecting the emergence of *neologisms*. The `EM` (Expectation Maximisation) algorithm is named `DynamicEM` and the model will hereafter be referred to as the `EJ` model (*Emms-Jayapal* model).

1.1 Context: *Neologisms* & `EJ` Model

To contextualise the domain with which this Final Year Project is concerned, it is first important to understand the notion of *neologisms*. There are two types: *formal* neologisms and *semantic* neologisms. The former refers to the arrival of an entirely new word to a language, whereas the latter is used to describe an existing word which takes on a new meaning. From a *synchronic* standpoint (i.e. at a discrete point in time), a word may be associated with numerous senses. These senses have different probabilities, meaning some may be more likely to occur than others. We are interested in plotting the variation of these senses *diachronically* i.e. throughout time. Upon examining the different probability distributions as they vary from year to year, we can begin to link the emergence of specific senses to certain time periods. Indeed, we find that the cultural context of the time period influences the distribution of sense probabilities for a given word. Neologisms are likely to emerge as a result of culturally significant events. Emms and Jayapal (2015) use the example of the word *'strike'* which began to take on the new meaning of *'industrial action'* in the late 18th century.

As discussed in the literature review below, *Emms* and *Jayapal* aim to create a time-aware model to detect the emergence of new senses for a given target word. In Emms and Jayapal (2015), the authors produce an algorithm to infer the best values for their model parameters based on n-gram data i.e. sequences of words extracted from a corpus. They implement an `EM` algorithm which is used to choose the model parameters in such a way as to maximise the likelihood of the observed n-gram data. In Emms and Jayapal (2016), the authors develop the idea further, this time providing a new estimation approach, namely *Gibbs Sampling*.

As we mentioned earlier, the time period is linked to the distribution of sense probabilities. However, so too is the total number of possible senses that may be associated with a given word throughout time. This value cannot always be known, especially when it comes to *neologisms*. Due to the perpetual evolution of natural language, new meanings are constantly arising while old ones fall out of use. As such, the number of senses, (referred to as k), is an unknown *hyperparameter* which must be supplied by the analyst when running of the `DynamicEM` algorithm to infer the parameters of the `EJ` model.

The `EJ` model developed was successful in detecting sense emergence for a given target word based on various user-specified parameters including the number, k, of senses. The obtained data could be plotted as a graph of *Year* against *Sense Probabilities* for a given iteration of the algorithm, as in Figure 1.1. The figure shows that, in the first year of analysis (1950), there was one single dominating sense of the word 'mouse' which occurred with 90% probability, (see the red curve). Presumably, this is the sense of the word corresponding to 'animal', although the model says nothing about the word's meaning, except for associating the sense with certain n-grams. By the year 2000, another meaning of 'mouse', (corresponding to the blue curve), was now more likely to occur in the corpus. Presumably, this is the 'pointer device' meaning of the word that emerged in technology from the 1980s onward.

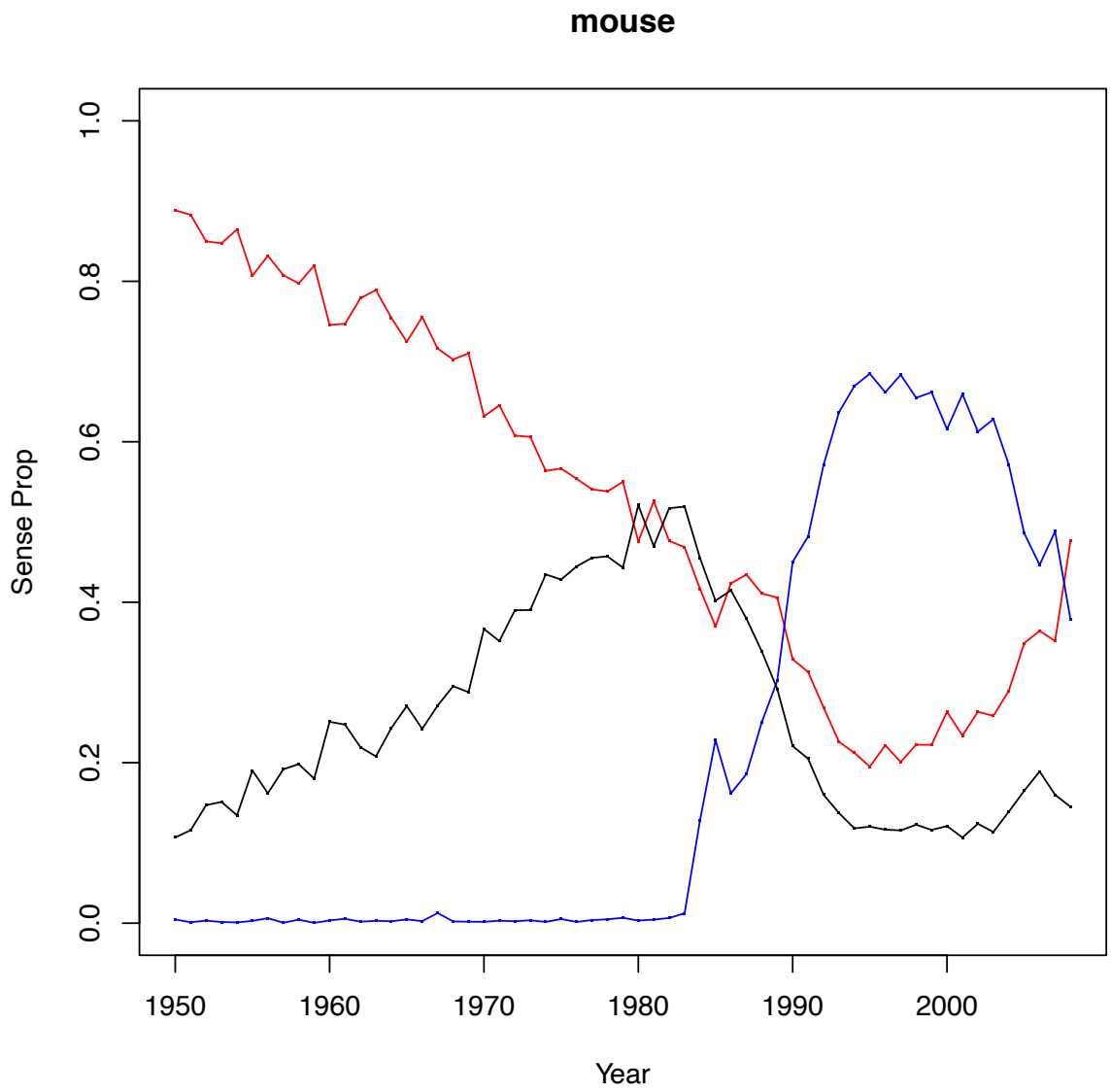

Figure 1.1: Sample Graph of *Sense Probabilities* against *Year* for target *mouse*

1.2 `EM`: Expectation Maximisation

The `EM` (Expectation Maximisation) algorithm first appeared in Dempster et al. (1977) and is a widely-applicable algorithm used to obtain maximum likelihood estimates of the parameters of statistical models from incomplete data. `EM` algorithms are often used in the context of *mixture models*, but can be applied to a wide range of models.

1.2.1 Gaussian Mixture Model Example

To illustrate how an `EM` algorithm functions, we will examine how it is applicable to carrying out maximum likelihood estimation (`MLE`) for Gaussian Mixture Models (`GMMs`). Afterwards, we will show how this example is comparable to the use of `DynamicEM` for *Dr. Martin Emms* and *Dr. Arun Jayapal's* time-aware model for detecting sense emergence. As this is the subject of this Final Year Project, it is relevant to explore an analogous example in this way.

Before discussing GMM, we will first start with a simpler case. Let us assume our observed data is $X = \{x_1, \ldots, x_n\}$, where each $x_i \in \mathbb{R}$. In a probabilistic model, we posit that this data was generated from some stochastic process, and we write down the probability that the particular samples that we observed, would have been observed among all other possibilities. This probability is our *model*. It is usually formed by making some assumptions about the generative process. The model is composed of *parameters* and *hyperparameters*.

Parameters are 'unknowns' which are internal to the model. The *inference* process involves using an algorithm to determine the optimal values of these parameters. There are different interpretations of the 'optimal' values. In maximum likelihood estimation (`MLE`), the optimal parameters are those which result in the observed data having been the most probable to occur. Hyperparameters, on the other hand, are variables whose values are not inferred from the data. Instead, the analyst must either supply these values themselves or determine them via testing the model's performance over a range of hyperparameters.

In our simple case of observed data $X = \{x_1, \ldots, x_n\}$, where each $x_i \in \mathbb{R}$, we want to devise a model describing how we think this data may have been generated. We could suppose that it was created from a Normal distribution. The formula for the probability of this observed data would then be given by:

$$p(x_{1:n}) = \prod_i \frac{1}{\sqrt{2\pi}\sigma} \exp(-\frac{1}{2}(x_i - \mu)^2/\sigma^2)$$

In this model, there are two unknowns: μ, the mean, and σ, the standard deviation. We could, for example, decide that μ should be a parameter of the model which will be inferred by an algorithm based on the observed data. On the contrary, we could decide that σ is a hyperparameter for which the analyst will plug in some reasonable value before running the inference algorithm. (Note that these decisions are at the discretion of the

model designers – we could easily have decided that σ should be also be an inferred parameter, for example).

Now, let us assume that we have a set of observed data points in $\mathbb{R}$. (Typically, we deal with vectors of values, for example: $[1.2, 3.4, 5.6] \in \mathbb{R}^3$). In many applications, when we plot real-world data, it is unlikely to be evenly distributed across $\mathbb{R}$. Instead, we often see clusters of data emerging, for example, a group of many points in the vicinity of 1.0 and another group of multiple points around -1.0, etc.

In a GMM, observed data points are described as belonging to a set of clusters using k normal distributions. Each data point comes from a single cluster, but since we are unsure which one, we must write down for each individual cluster the probability that it came from that cluster. There might be a normal distribution whose mean is 1.0 and another whose mean is -1.0, as in the example above, but we don't know these means. We must infer their values. We could decide that we have k clusters. (Since we are deciding this value, it is a hyperparameter). Therefore, the values of k means i.e. $\mu_{1:k}$ will have to be determined in the inference process.

Furthermore, we note the probability that a data point x_i will have come from a given cluster generated from its corresponding Normal distribution. We will write this as $\pi_{1:k}$. As such, π_j represents the probability that x_i came from the j^{th} cluster and μ_j is the mean of this cluster. The probability distribution is now given by:

$$p(x_{1:n} \mid \mu_{1:k}, \pi_{1:k}) = \prod_{i=1}^{n} \big(\sum_{j=1}^{k} \pi_j \frac{1}{\sqrt{2\pi}\sigma} \exp(-\frac{1}{2}(x_i - \mu_j)^2/\sigma^2) \big)$$

Once again, the observed data is represented by x_i, the parameters of the model to be inferred are π_j and μ_j and the hyperparameters that the analyst must decide are k and σ. We now need an algorithm for our inference process. One such algorithm is of course the `EM` algorithm. This is commonly used as it provides an `MLE`, as explained above. First, we attempt to simplify the expression by taking the *log* of both sides, (this allows us to turn products into sums: Π becomes Σ):

$$\log p(x_{1:n} \mid \mu_{1:k}, \pi_{1:k}) = \sum_{i=1}^{n} \log \big(\sum_{j=1}^{k} \pi_j \frac{1}{\sqrt{2\pi}\sigma} \exp(-\frac{1}{2}(x_i - \mu_j)^2/\sigma^2) \big)$$

We have now encountered a sum within a *log* on the RHS of this expression. Since this is difficult to simplify, we introduce a *latent* or *hidden* variable. This type of variable is

not directly observed in our data. A typical *latent* variable introduced in mixture models is the cluster from which a data point originated, namely z_i. In reality, our observed data does not include values for this variable. However, we can 'pretend' as though that is the case. We now have:

$$p(x_{1:n}, z_{1:n} \mid \mu_{1:k}, \pi_{1:k}) = \prod_{i=1}^{n} \pi_{z_i} \frac{1}{\sqrt{2\pi}\sigma} \exp(-\frac{1}{2}(x_i - \mu_{z_i})^2/\sigma^2)$$

and when we take the log we get:

$$\log p(x_{1:n}, z_{1:n} \mid \mu_{1:k}, \pi_{1:k}) = \sum_{i=1}^{n} \log(\pi_{z_i}) - \frac{1}{2} \sum_{i=1}^{n} (x_i - \mu_{z_i})^2/\sigma^2 - \frac{1}{2} \log(2\pi\sigma^2)$$

This expression helps us to calculate the conditional probability of z_i given x_i i.e. $p(z_i \mid x_i)$. Based on this, we can then calculate the expected log likelihood. The `EM` algorithm now iterates through two successive steps until convergence. Initially, the parameters and latent variables are set to random values. Then the steps are:

1. **Expectation:** The current values of the parameters are used to update the probability of the latent variable z_i.

2. **Maximisation:** The updated probability of z_i is now used to update the values of the parameters.

1.2.2 Laplacian Mixture Model Example

We now briefly digress to highlight that an `EM` algorithm can be applied to any type of mixture model, not just one involving Normal distributions. Below, there is a brief illustration of the process using a mixture model whose clusters are generated from the Laplacian distribution. Like the Gaussian distribution, the Laplacian distribution has many real-world applications. (For example, Maretic and Frossard (2018) propose a framework for inferring multiple graph Laplacians from mixed signals).

As before, our observed data is $X = \{x_1, \ldots, x_n\}$, where each $x_i \in \mathbf{R}$. We posit that this data was generated from a Laplacian distribution. Then the probability of the observed data is given by:

$$p(x_{1:n} \mid \mu, b) = \prod_i \frac{1}{2b} \exp\left(-\frac{|x_i - \mu|}{b}\right)$$

Here, μ is the mean of the distribution. It is also a location parameter as it essentially controls the position of the maximum on the x-axis. b is a scale parameter which controls the steepness of the function. Similar to the GMM, we could decide that μ is an inferred parameter and b is a hyperparameter for which the analyst will provide a reasonable value. Again, we have k clusters and $\pi_{1:k}$ being the probability that x_i came from the k^{th} cluster. The probability distribution is now:

$$p(x_{1:n} \mid \mu_{1:k}, \pi_{1:k}) = \prod_{i=1}^{n} (\sum_{j=1}^{k} \pi_j \frac{1}{2b} \exp\left(-\frac{|x_i - \mu_j|}{b}\right))$$

Then we can introduce the latent variable z_i as before:

$$p(x_{1:n}, z_{1:n} \mid \mu_{1:k}, \pi_{1:k}) = \prod_{i=1}^{n} \pi_{z_i} \frac{1}{2b} \exp\left(-\frac{|x_i - \mu_{z_i}|}{b}\right)$$

and when we take the log we get:

$$\log p(x_{1:n}, z_{1:n} \mid \mu_{1:k}, \pi_{1:k}) = \sum_{i=1}^{n} \log(\pi_{z_i}) + \sum_{i=1}^{n} \frac{-|x_i - \mu_{z_i}|}{b} - nlog(2b))$$

As before, this expression helps us to calculate the conditional probability of z_i given x_i i.e. $p(z_i \mid x_i)$. Based on this, we can then calculate the expected log likelihood. We would then try to find the parameter values that maximise this expression. This would be done by running our `EM` algorithm until convergence.

1.2.3 Relation to `DynamicEM`

The example of mixture models discussed above, both in relation to Gaussian and Laplacian distributions, is analogous to the `EJ` model. Having explained how `EM` algorithms are used to infer model parameters, the `EJ` model and `DynamicEM` will now be considered. To begin drawing parallels between our analogous examples and the `EJ` model, we will first ignore the *Year* data.

Ignoring Year

Instead of generating numbers, the `EJ` model generates sequences of words (n-grams). The observed data can be written in the form $\mathbf{W}_1, \mathbf{W}_2, \ldots \mathbf{W}_D$ where D is the number of observed samples and each $\mathbf{W}_i$ is a sequence of n words i.e. an n-gram. We write

down a probability function which represents the probability of observing the n-gram $\mathbf{W}$ among all possible n-grams that might have been observed. Since the word sense is not incorporated into the Google n-gram data used in Emms and Jayapal (2015) and Emms and Jayapal (2016), we introduce a latent variable S, (just like z in the mixture model), which represents the sense. Next, the complete likelihood is given as

$$p(\mathbf{W}, S) = p(\mathbf{W}|S)p(S) = \prod_{j=1}^{n} p(W_j|S)p(S)$$

where the second equality represents splitting the n-gram into its constituent words. An assumption is made that the words are all independent of each other, once the sense S is known. The size of the vocabulary of the entire corpus is represented as V. We use k to denote the number of possible senses. This number is unknown and therefore it is a hyperparameter which means the analyst will have to select some appropriate value for it.

Now we consider the expression $P(W|S)$, the probability of the word W given the sense S. There are V possible words and k possible senses, so there are $V \times k$ values to represent all these probabilities. We can call these values θ_{ij}, where i runs from 1 to V and j runs from 1 to k, and θ_{ij} is the probability of the i^{th} word given the j^{th} sense. Similarly, $P(S)$ has one value for each of the k possible values of S, and we can write $P(S = j) = \pi_j$. So we get to the probability of the observed data, where we have observed D samples of n-grams:

$$\begin{aligned} p(\mathbf{W}_1, \ldots, \mathbf{W}_D) &= \prod_{i=1}^{D} p(\mathbf{W}_i) = \prod_{i=1}^{D} \left(\sum_{j=1}^{k} p(S = j) p(\mathbf{W}_i | S = j) \right) \\ &= \prod_{i=1}^{D} \left(\sum_{j=1}^{k} p(S = j) \prod_{\ell=1}^{n} p(W_{i\ell} | S = j) \right) \\ &= \prod_{i=1}^{D} \left(\sum_{j=1}^{k} \pi_j \prod_{\ell=1}^{n} \theta_{w_{i\ell}, j} \right) \end{aligned}$$

and taking the log:

$$\log p(\mathbf{W}_1, \ldots, \mathbf{W}_D) = \sum_{i=1}^{D} \log \left(\sum_{j=1}^{k} \pi_j \prod_{\ell=1}^{n} \theta_{w_{i\ell}, j} \right)$$

The probability is written in terms of our parameters θ_{ij} and π_j, but, as for the mixture model, it doesn't simplify much. We need to use an EM algorithm to find the values for

the parameters that maximises the likelihood.

Let s_i be a latent variable representing the sense corresponding to the n-gram $\mathbf{W}_i$. Before commencing the `EM` algorithm, this latent variable s_i and the parameters are initialised randomly. Then, in the *expectation* step, we find $p_{ij} = P(s_i = j|\mathbf{W}_1, \dots \mathbf{W}_d)$, the probability that the i^{th} n-gram comes from the sense j, given the data and the current value of the parameters. In the *maximisation* step, we use p_{ij} to update the values of the parameters. We iterate over these steps until convergence to finally learn estimate of π_j and θ_{ij}.

Reintroducing Year

The `EJ` model is more complicated in so far as it also considers the year and assumes that the probability of a sense depends on the year in which the n-gram has been generated. So, instead of k values of π_j, we now have $k \times T$ values, where T is the total number of different years that we want to consider. Now π_{jt} is the probability of the j^{th} sense in the year t. The year is contained within the Google n-gram corpora used by the authors (discussed later) and so the observed data now extends to $\mathbf{W}_1, \dots \mathbf{W}_d$ and $Y_1, \dots Y_d$, the years in which the words were observed. We need to introduce an extra set of parameters $p(Y = j) = \tau_j$. The model is now:

$$
\begin{aligned}
p((Y_1, \mathbf{W}_1), \dots, (Y_D, \mathbf{W}_D)) &= \prod_{i=1}^{D} p((Y_i, \mathbf{W}_i)) = \prod_{i=1}^{D} \left(\sum_{j=1}^{k} p(Y_i) p(S = j|Y_i) p(\mathbf{W}_i|S) \right) \\
&= \prod_{i=1}^{D} \left(p(Y_i) \sum_{j=1}^{k} \pi_{j,y_i} p(\mathbf{W}_i|S = j) \right) \\
&= \prod_{i=1}^{D} \left(p(Y_i) \sum_{j=1}^{k} \pi_{j,y_i} \prod_{\ell=1}^{n} p(W_{i\ell}|S = j) \right) \\
&= \prod_{i=1}^{D} \left(p(Y_i) \sum_{j=1}^{k} \pi_{j,y_i} \prod_{\ell=1}^{n} \theta_{w_{i\ell},j} \right) \\
&= \prod_{i=1}^{D} \left(\tau_{y_i} \sum_{j=1}^{k} \pi_{j,y_i} \prod_{\ell=1}^{n} \theta_{w_{i\ell},j} \right)
\end{aligned}
$$

Once again, we have the probability written in terms of our parameters. As before, we can see the sum over the senses will cause difficulties if we try to directly maximise this expression over the parameters. The method is the same as before. First, we introduce the latent variable s_i representing the sense of the i^{th} n-gram. Then, we use the *expectation* step of the `EM` algorithm to estimate p_{ij}, the probability that the sense of the i^{th} n-gram

is j given the data and parameters. Next, we use p_{ij} in the *maximisation* step to update the parameters. Finally, we repeat until convergence.

1.3 Issues Impeding Use of `DynamicEM`

The current implementation of the algorithm does not easily facilitate its use. For an analyst, there are a multitude of hindrances which render using this tool a somewhat complicated task. Firstly, it may only be run via the command line. There is no interface to support changing what arguments are passed to the code. In fact, given the numerous parameters that may be adjusted, the command to run the program must include a manually typed-out list of arguments that the user wishes to specify. The following is an example:

```
-targ mouse \
-corpus_type 2 \
-expts <output_dir> \
-files ./mouse_list \
-left 4 \
-right 4 \
-whether_csv_suffix no \
-whether_pad no \
-num_senses 3 \
-senseprobs_string 0.35/0.55/0.1 \
-sense_rand no \
-word_rand yes \
-set_seed yes \
-rand_mix no \
-rand_word_mix yes \
-rand_word_mix_level 1e-05 \
-words_prior_type corpus \
-unsup yes \
-max_it 100 \
-time_stamp yes
```

Typing out this long string is impractical and not conducive to an analyst making full use of the tool. It also means that running the algorithm numerous times becomes tedious, even if arguments are only being varied in a minor way. The aim is to improve this user experience so that more analyses may be conducted without wasting time on

manual typing. We therefore seek to design a Graphical User Interface which streamlines parameter setting.

Whilst the `EJ` model whose parameters are inferred by the `DynamicEM` algorithm is a useful tool for detecting sense emergence, it cannot easily be utilised for conducting more complex analyses, such as comparing runs with different hyperparameter values. (For example: comparing runs with various values for k, the number of senses). The results depicted in Figure 1.1 represent a specific run of the algorithm based on discrete parameter values. An analyst wishing to use this tool might expect to view graphs which offer more information about parameter variation, such as how changes to k may influence the results. This calls for the implementation of some way to display multiple graphs side-by-side to aid with visually comparing results. To save even more time, this visual inspection should ideally be possible to carry out while the `DynamicEM` algorithm runs.

Another impractical drawback associated with the current implementation is the graph-generation process. This process is entirely separate to `DynamicEM` code and involves starting an `R` shell and manually typing in a series of commands. Since we have planned to implement a GUI, the question arises of whether it would be possible to consolidate this process within the same interface. Furthermore, it would be desirable to simplify the process into a single click.

Additionally, the static graph shown in Figure 1.1 provides no indication as to the progression of the `EM` algorithm. Since this type of algorithm involves numerous iterations, it would be useful to be able to identify what number of iterations produces a satisfiable result. A visual representation of progress could also quickly indicate if there is any unexpected behaviour occurring during a given run. One solution to this would be the generation of animations, which, in contrast to graphs, depict the sense probability values throughout *all* the iterations.

This Final Year Project will address the issues with the current implementation as outlined above. The goal is to provide an intuitive GUI equipped with the necessary tools for analyses to be smoothly conducted using the `EJ` model inferred by `DynamicEM`.

Chapter 2

State of the Art

2.1 Background

2.1.1 Development of the `EJ` model & `DynamicEM` algorithm

Emms and Jayapal (2015) explored the phenomenon of semantic neologisms, specifically in relation to determining their date of emergence. A model was proposed to condition the surrounding vocabulary (or context) of a target word on the sense of the target and to condition the target sense on the time of its occurrence. It was also assumed that the context vocabulary conditioned on the target was independent of time. This is largely true in reality and the assumption means that fewer parameters must be estimated. A second assumption was made that the context vocabulary conditioned on the sense of the target were independent of each other. The model produced was the following equation:

$$p(Y, S, \vec{W}) = p(Y) \times p(S \mid Y) \times \prod_i p(W_i \mid S)$$

where, for a given target word, Y represents the year of occurrence, S refers to one of its k senses and $\vec{W}$ is a vector describing its surrounding words (context).

The authors used the Google n-gram data set compiled by Michel et al. (2011), which contains values for Y and $\vec{W}$. However, it lacks the sense variable, S. This variable is a *latent* or *hidden* variable. To estimate the parameters of the model based on this data, an unsupervised `EM` procedure was run. The specifics of this procedure are given below:

Estimation: for each data item d and each possible value s of S, determine the conditional probability of $S = s$ given Y^d and $\vec{W}^d$, called $\gamma^d(s)$

Maximisation: use the $\gamma^d(s)$ values to derive new estimates of parameters via the

update formulae:

- $p(S = s \mid Y = y) = \frac{\sum_{d:Y^d=y}[\gamma^d(s)]}{\sum_{d:Y^d=y}[1]}$
- $p(w \mid S = s) = \frac{\sum_d(\gamma^d(s) \times \#(w, \vec{W}^d))}{\sum_d(\gamma^d(s) \times length(\vec{W}^d))}$

`EJ` model inferred with Gibbs Sampling

In Emms and Jayapal (2016), the authors again use the `EJ` model to detect diachronic word sense emergence. However, they adopt a new method for inferring the model parameters rather than utilising an `EM` algorithm. The method in question is *Gibbs Sampling*. Here, the iterations produce samples which all contribute to the calculation of a final mean. Essentially, there is a posterior density on the sense given year probabilities. The resulting output obtains the mean of this posterior by taking a mean over all the samples. The advantage of Gibbs sampling is that the final result represents a full distribution of values. One can plot the means of the sampled sense given year probabilities as well as the Highest Density Region (HPD). This is an interval within which the sense given year probabilities may fall. As such, the analyst can view a range of possible probabilities which affords them a degree of confidence in the inference process – if the HPD is a significantly large interval, the degree of confidence may decrease.

We have briefly mentioned the notion of *Gibbs Sampling* in order to convey that the `EJ` has been tested with multiple methods of inference. However, this is outside the scope of this Final Year Project as we focused instead on *Expectation Maximisation* to infer parameters, as described in Chapter 1. A future direction of work could be extending the GUI to allow for the relevant parameters to be specified for a run based on *Gibbs Sampling*.

We will now discuss the notion of *pseudoneologisms* explored in Emms and Jayapal (2016). This represents an important technique for testing sense emergence which is very relevant in the context of experimenting with the `EJ` model.

2.1.2 *Pseudoneologisms*

We will first look at the paper in which the *pseudoword* method was formulated: Schütze (1998). This concept gave rise to the development of *pseudoneologisms*. In the paper, the authors propose an algorithm called *context-group discrimination* for distinguishing between ambiguous words. Ambiguous words are grouped into clusters according to the similarity of their contexts. These clusters may be considered as the senses of the

word. A high-dimensional, real-valued vector space represents words, contexts and clusters. Second-order context vectors ($\mathbf{C}^2$) are used to capture this information, since they are considered to be more robust and less sparse than first-order context vectors ($\mathbf{C}^1$). As concluded by Maldonado and Emms (2012), in a supervised word sense disambiguation setting, $\mathbf{C}^1$ vectors perform better when compared to $\mathbf{C}^2$ vectors; but in an unsupervised word sense discrimination setting, the two context representations perform similarly. The *context-group discrimination* algorithm is unsupervised, so $\mathbf{C}^1$ would not provide any advantage in terms of performance.

The second-order context vectors are used to judge similar words and assign them to the same cluster. A word from a test text is disambiguated by determining the second-order representation of its context and assigning it to the cluster for which the centroid (average of cluster elements) is closest to that representation. In order to test this disambiguation algorithm, the notion of *pseudowords* is introduced. *Pseudowords* are artificially ambiguous words constructed from two or more words merged into a new type. Schütze (1998) gives the example of *banana* and *door*, which becomes *banana/door*. The occurrences of all the words used to form this pseudoword within the corpora are then replaced with the pseudoword. In our example, every *banana* and every *door* would be replaced with *banana/door*. Now, after testing the algorithm, it is easy to go back to the original text and judge whether the sense was correctly determined i.e. the word was correctly disambiguated. This also eliminates the tedious requirement of hand-labelling every occurrence of an ambiguous word within the corpora.

In Emms and Jayapal (2016), the authors develop the *pseudoword* technique for use in testing diachronic sense emergence. In a given range of years t, two unambiguous words are selected, namely σ_1 and σ_2. σ_1 is a word used throughout the entire time period, whereas σ_2 is a word which first emerges at some point t_e. Now, every occurrence of σ_1 and every occurrence of σ_2 in the n-gram corpora is treated instead as an occurrence of the fake word 'σ_1-σ_2'. The result is that σ_1-σ_2 functions as an artificial semantic neologism where σ_2's sense begins to be exhibited from t_e onward. The term *pseudoneologism* is coined to describe this. The authors define $f_t(\sigma_i)$ which describes the actual observed probability of the target word σ_i in combined σ_1-σ_2 data for time t. Next, the result of the inference process for which the number of senses hyperparameter $k = 2$ is obtained. The expectation is that for each sense k, the plotted $\pi_t[k]$ data should be similar to the real observed probability given by $f_t(\sigma_i)$.

To test this theory, the time period from 1850-2008 was selected. σ_1 was assigned the word 'ostensible', which is known to have existed throughout this period. σ_2 was

assigned the possible words 'supermarket', 'genocide' and 'byte', as each of these three words first emerged at some point throughout the period. Plots of word sense probability distributions against year were produced for both the real observed data and the inferred data in the way described above. The authors indeed found that the trajectories of the empirical and inferred data did match.

2.2 Hyperparameter Optimisation

As explained, hyperparameters are not inferred throughout the machine learning process; rather, their values are given by the analyst. There are different methods that the analyst might use for choosing these hyperparameters ranging from simply deciding on values that seem reasonable to visually comparing runs with different values to relying on specific algorithms for determining the optimal values. Our GUI developed in this Final Year Project offers the analyst a degree of side-by-side visual inspection which can be used to cross-reference runs for different values of the hyperparameter k (the number of senses).

In a wider context, different machine learning models may incorporate large amounts of hyperparameters and the process of determining their optimal values is therefore imperative for improving the model's performance. Analysts may use their expertise and prior knowledge to determine the best hyperparameter values. To aid in this regard, they frequently perform visual analyses to compare different runs, (a method that has been facilitated in this Final Year Project). Additionally, numerous algorithmic-based approaches to fine-tuning hyperparameters have been developed and there are many tools and services that can be used to accelerate this process. We will give a brief overview of some of the commonly used hyperparameter optimisation algorithms as well as some of the applications available to analysts for this purpose.

Yu and Zhu (2020) provide a review comparing different hyperparameter optimisation (HPO) algorithms and applications. According to the authors, HPO is "a process of finding a set of hyperparameters to obtain minimum loss or maximum accuracy of an objective network". This objective function, as defined by Feurer and Hutter (2019), is the following expression:

$$\lambda^* = \arg\min_{\lambda \in \mathbf{\Lambda}} \; or \; \arg\max_{\lambda \in \mathbf{\Lambda}} \mathbb{E}_{(D_{train}, D_{valid}) \sim \mathcal{D}} \mathbf{V}(\mathcal{L}, \mathcal{A}_\lambda, D_{train}, D_{valid})$$

$\mathcal{D}$ represents the data set. $\mathcal{A}$ is a machine learning algorithm with N hyperparameters. The *hyperparameter configuration space* is given by $\mathbf{\Lambda} = \Lambda_1, \ldots, \Lambda_N$. $\lambda \in \mathbf{\Lambda}$ describes a vector of hyperparameters and therefore $\mathcal{A}_\lambda$ denotes a machine learning algorithm whose

hyperparameters are instantiated to λ. $\mathbf{V}(\mathcal{L}, \mathcal{A}_\lambda, D_{train}, D_{valid})$ measures the loss or accuracy (when minimised or maximised respectively) of a model generated by $\mathcal{A}_\lambda$ on training data D_{train} and evaluated on validation data D_{valid}.

The authors of Yu and Zhu (2020) then proceed to subdivide HPO algorithms into two main categories: search algorithms and trial schedulers. Search algorithms are used for sampling whereas trial schedulers essentially entail early-stopping methods for model evaluation. We will now give a brief overview of these different algorithms and compare their advantages and disadvantages.

2.2.1 Search Algorithms

Grid Search

In this method, the analyst must generate all hyperparameter candidates. Therefore, they must be able to decide on reasonable values for these candidates based on prior knowledge. Grid search entails performing an exhaustive search on this set of candidates i.e. all possible hyperparameter combinations are considered. It is a simple HPO algorithm which provides accurate predictions. It is also advantageous because it can easily be run in parallel. However, it suffers greatly when it comes to computational resources. The required resources increase exponentially the more hyperparameters need to be determined. The total computation budget B refers to the number of sampled candidates or sampling iterations. In Grid search, the budget for each hyperparameter set is $B^{1/N}$, where N is the number of hyperparameters.

Random Search

Random search is carried out over a number of candidates which are sampled from a hyperparameter space with a specified distribution. This method is an improvement on Grid search. Instead of a fixed budget, the analyst may assign an independent budget for each set of hyperparameters according to the distribution of the search space. Furthermore, the more time consumed, the more probable it is that the optimal hyperparameter set will be found. Random search lends itself well to being run in parallel, but intensive computation is still a significant drawback.

Bayesian Optimisation (BO)

Bayesian Optimisation is a well-known method which can be applied in many areas where global optimisation is required. This is a sequential method which involves balancing exploitation and exploration. Exploitation refers to making the best decision based on

current information whereas exploration refers to collecting more information. This avoids trapping into the local optimum. BO is more computationally efficient compared to both Grid search and Random search. BO comprises two main elements, namely a Bayesian probability surrogate model which models the objective function λ^*, and an acquisition function which determines the next sampling point. Initially, a prior distribution of the surrogate model is built. Then, the following steps are repeated until the optimal hyperparameters are obtained or the specified limit of resources has been reached:

1. Determine the best-performing hyperparameter set on the surrogate model.
2. Compute the acquisition function with the current surrogate model.
3. Apply the hyperparameter set to the objective function.
4. Update the surrogate model with new results.

2.2.2 Early-Stopping Methods

In many cases, machine learning can take a long time and consume a large amount of resources. As such, an analyst must often evaluate the model during training and use their expertise to manually fine tune the hyperparameters and narrow the search space. They can determine whether to stop or continue the machine learning process. Early-stopping algorithms use different methods to decide whether or not to halt the training process. There is similarity between early-stopping algorithms for both HPO and neural network training, but in the latter, they are applied to avoid *overfitting*. Overfitting is where the model gives accurate predictions for the training data, but not for new data. The Figure 2.1 below depicts an example of overfitting:

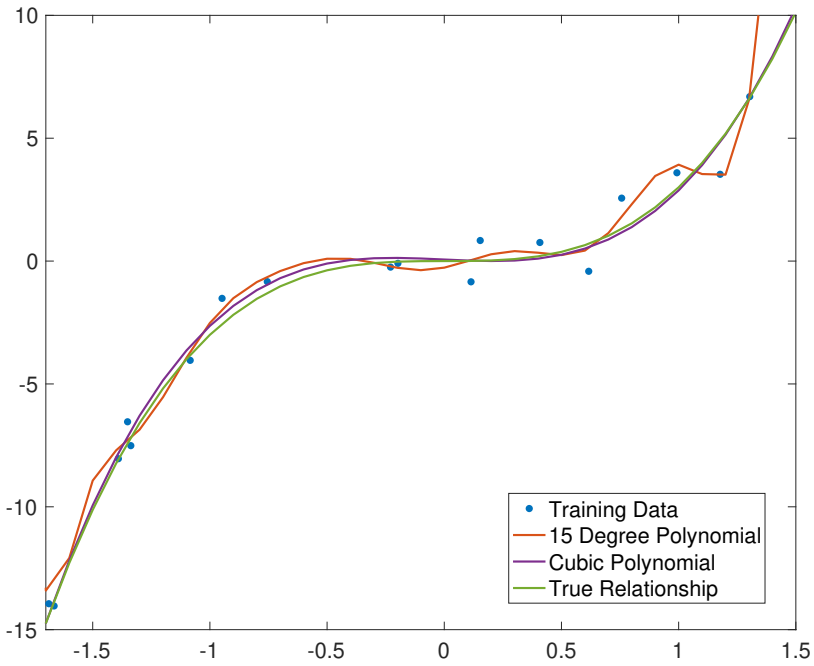

Figure 2.1: An example of overfitting. The true relationship between x and y is a cubic relationship, but the training values have some noise added to y. A 15-degree polynomial fits the training data between, but it fails to capture the true relationship.

Median Stopping

This is an early-halting strategy which computes running averages of primary metrics across all trials i.e. training runs. A trial is halted if its best performance by the current step is strictly worse than the median value of the running average. Since this is an early-stopping policy rather than a model, it is advantageous as the analyst does not need to determine hyperparameters.

Curve Fitting

Curve fitting is an algorithm which can be used in combination with the search algorithms mentioned above. A set of completed or partially completed trials are used to compute a performance curve to predict the final objective value, such as accuracy or loss. A given trial is halted if the prediction of the final objective value at the current step is worse than the tolerance value of the optimal in the trial history. HPO performance is significantly improved when this model is applied with search algorithms. Care must be taken to ensure that overfitting is avoided.

2.2.3 HPO Toolkits

Many tools have been developed to aid with the HPO process. One example is *'Neptune'*. This service offers a python API for logging and querying model-building metadata. There is also an accompanying web application for visualising and comparing hyperparameters. *'Comet'* is another service for helping to optimise hyperparameters. It facilitates combining and customising visualisations all in one platform. In terms of open-source software, there is *'TensorBoard'*. This is a toolkit extension to the popular *'TensorFlow'* library used for machine learning and artificial intelligence.

Yu and Zhu (2020) give an overview of some HPO toolkits and compare them based on a number of factors including *Ease of Use, Scalability, State-of-the-art, Availability* and *Flexibility*. Google *'Vizier'* and Amazon *'SageMaker'* are grouped together as they are both cloud-based services. This is very advantageous as computational resources are automatically managed and jobs are distributed across load-balancing infrastructure. These services are excellent in terms of scalability and running many trials in parallel. The advantage of Amazon *'SageMaker'* over Google *'Vizier'* is that it facilitates every step of training a neural network – HPO is only a module within this service. This means that an analyst could carry out all stages of model development from within the same tool. A host of open-source software is also available for HPO. Notably, Microsoft's *'Neural Network Intelligence'* (NNI).

Comparing open-source toolkits with paid services for HPO:

- *Ease of Use*: Paid services are easier for people starting off, whereas open-source toolkits can require prior knowledge on model training.
- *Scalability*: Paid services such as Google *'Vizier'* and Amazon *'SageMaker'* can leverage their long-established cloud-based infrastructure which offers easier and faster scaling options than open-source toolkits.
- *State-of-the-art*: The HPO algorithms offered differ from service to service and an analyst should consider what they require when choosing between open-source and paid services.
- *Availability*: There is a cost to be considered when utilising paid services, whereas open-source software is available for free.
- *Flexibility*: The infrastructure of paid services is closed-source which limits flexibility. On the other hand, everything can be modified when it comes to open-source software and therefore experienced users can avail of higher flexibility.

2.3 Summary

In this Section, relevant literature has been explored relating to the development of the `EJ` model and accompanying `DynamicEM` algorithm for inferring its parameters. Emms and Jayapal (2015) described the model with an *Expectation Maximisation* approach and tests were run using the Google n-gram data compiled by Michel et al. (2011). Then, in Emms and Jayapal (2016), the model was inferred using the *Gibbs Sampling* method. Additionally the concept of *pseudoneologisms* was introduced. A brief overview was given of the paper from which this notion was inspired.

Furthermore, contemporary methods of hyperparameter optimisation (HPO) were examined, including popular algorithms and applications as outlined in Yu and Zhu (2020). Two categories of HPO algorithm were discussed, namely search algorithms and early-stopping algorithms. The former included *Grid Search, Random Search* and *Bayesian Optimisation* (BO). The latter included *Median Stopping* and *Curve Fitting*. In terms of applications which facilitate HPO and visualisation, a number of paid services and open-source toolkits were compared. Examples of paid services were *'Neptune', 'Comet', 'Vizier'* and *'SageMaker'*. On the other hand, open-source software including *'TensorBoard'* and *'Neural Network Intelligence'* (NNI) were discussed and compared to the their paid service counterparts.

Chapter 3

Design

3.1 Overview of the Approach

Based on the main areas of improvement identified earlier, a list of requirements were formulated, as can be seen in the following checklist:

1. Provide a means for the user to easily specify parameters without needing to modify a long string. ☐
2. Allow analysis to be carried out while the machine learning process runs. ☐
3. Provide some way to seamlessly generate graphs from data obtained via the algorithm. ☐
4. Allow for smooth side-by-side visual analysis of graphs. ☐

Overall, these requirements call for the development of a GUI in which *visual analysis*, *running of the algorithm* and *graph generation* are all consolidated in the same platform.

3.2 Library Selection

There are many different libraries created for aiding programmers to develop Graphical User Interfaces. One such library, which was eventually selected, is `Qt`. This library was selected for a multitude of reasons.

- Firstly, it is well-established and is accompanied by detailed documentation. As such, one may become familiar with its functionality relatively quickly.

- It also integrates with a number of different languages, including `C++`. The original code supplied by *Dr. Martin Emms* was written in `C++` and furthermore, since the running of the `DynamicEM` algorithm itself could easily be implemented using the `system()` call provided by `C++`, this was naturally the default language of choice.
- It also contains numerous useful classes, including `QThread` for multi-threading.

One of our aims was to create animated graphs to represent the progression of the `DynamicEM` algorithm throughout all of its iterations. To this end, we required some way to generate animations from within `R` code. The solution was the creation of `gif` files wherein every frame represents the next iteration of the algorithm. This endeavour was accomplished with the aid of three `R` packages:

- `ggplot2` by Wickham (2016): This package is used for creating static graphs based on 'The Grammar of Graphics'.
- `gganimate` by Pedersen and Robinson (2022): This package extends the `ggplot2` package to facilitate the creation of animations.
- `gifski` (available at `https://github.com/r-rust/gifski`): This package is typically used to combine the frames of the animations created via `gganimate`.

3.3 Corpus Data

We have already seen that, within the `EJ` model, the sense parameter is dependent on the year. As such, the corpora of real observed data serving as input to the model must be time-stamped and ideally grouped by target word. In Emms and Jayapal (2015) and Emms and Jayapal (2016), the authors use the Google n-gram data set compiled by Michel et al. (2011) which contains time-stamped data. An n-gram is a length-n sequence of words extracted from a given corpus. The Google data set in question represents about 4% of all books ever printed and includes 1-gram, ..., 5-gram data for each year. An example 5-gram available in the data set, as given in Emms and Jayapal (2016), is shown below:

```
Enter or click the mouse 1990 9 7
```

The rightmost number (7) indicates the total number of publications from the given year (1990) which contain the 5-gram. The second number from the right (9) counts the total number of occurrences of the 5-gram in all publications from that year. All information in these 5-grams is utilised in the algorithm, except for the rightmost number. The data set

is neither arranged into year-specific files nor grouped by target word. Such organisation must therefore be carried out before the algorithm may be run. Furthermore, the sheer size of the Google data set means that sub-corpora must be extracted for a given target word. This extraction process can take a very long time – between 2 and 3 days – which is a significant disadvantage associated with using the Google corpora.

An alternative is the COHA (Corpus of Historical American) compiled by Mark Davies (2022). In contrast to the Google n-gram data set, the entirety of this corpus can be run through in about 10 minutes, which means tests may be carried out on this data relatively quickly. The format of this corpus is slightly different to the Google data set in that text is not arranged into n-grams. Instead, it is a collection of samples of text comprising 385 million words derived from 115,000 texts which were produced between the years 1820 and 2019. The `DynamicEM` code was implemented to read in corpus data from both the Google and COHA corpora. Specifically, the *'Corpus Type'* parameter indicates which of these corpora is to be read in and processed. (Google: `corpus_type = 2`, COHA: `corpus_type = 3`).

Some sub-corpora had already been extracted from the Google data set by *Dr. Martin Emms* for specific target words. The words in question were: {*mouse, gay, jet, tank, tanks, plane, planes, transcribe, transcribes, transcribed, transcribing*}. Additionally these sub-corpora were organised into specific years. Therefore, seeing as this initial work had already been carried out, we decided that our GUI implementation would facilitate the running of `DynamiEM` based on these sub-corpora. Potential further work on this Final Year Project could involve extending the possible corpus data that could be specified as training input for the `DynamicEM` algorithm. The code already accepts some Google sub-corpora data, (as well as COHA data), but it would be interesting to see if the process of extracting sub-corpora for a given target word and organising them according to specific years could be streamlined or done automatically.

3.4 Use Case Scenario

We will now describe a use case scenario to illustrate the desired functionality of our GUI. The scenario involves the analyst running the `DynamicEM` algorithm and, in the meantime, interacting with graphs which were generated at a prior time. Finally, when the execution of `DynamicEM` has terminated, the analyst generates a new graph from the resulting data and compares it to other graphs.

1. The analyst starts the GUI and is presented with the parameter-setting display.

2. Different parameters are modified in order to specify a run for the target word 'mouse' with 4 senses and 100 iterations. However, the analyst incorrectly initialises the sense probabilities by not ensuring that they sum to 1.

3. The *'Run Algorithm'* button is pressed and the parameter selections are processed. Since the sense probability values do not sum to 1, the code instead inserts default values specifying that each sense has an equal initial probability, in other words, '0.25/0.25/0.25/0.25' is the argument passed for the `senseprobs_string` parameter. The algorithm begins processing the corpus and going through its iterations in a separate thread.

4. During processing, the analyst checks how many graphs are currently available for inspection by clicking a button *'Which Graphs'*. An alert box displays the appropriate information. (In this scenario, it is assumed that two graphs have already been generated, namely `animation-mouse-2-sens-100-iter.gif` and `animation-mouse-3-sens-100-iter.gif`).

5. Next, the analyst presses the *'Start Animation'* button. They are prompted with a box listing the two available graphs to select. Both of these graphs are selected and are rendered in the window of the interface side-by-side. (They are animating on loop).

6. The analyst maximises the animation speed using a slider.

7. The *'Display Final Iteration'* button is pressed to halt the animations at their final frame which essentially displays static graphs representing the results of the runs.

8. Once the `DynamicEM` code has terminated, the *'Generate Animation'* button is pressed. This requests an animation to be generated based on the same parameter settings the analyst just used for the run. (The parameters have not been modified since the algorithm was started). The animation generation begins in a new thread.

9. Once the animation has been rendered, the analyst presses *'Start Animation'* once more and this time selects all three graphs i.e. the same two graphs as before plus the new one just generated, namely `animation-mouse-4-sens-100-iter.gif`.

10. The *'Quit'* button is pressed and the GUI closes.

Chapter 4

Implementation

4.1 User-friendly Parameter Specification

As mentioned, the original implementation of the `DynamicEM` algorithm requires the user or analyst to pass a long string of arguments to the program. To streamline this process, we aimed to implement a Graphical User Interface allowing for parameters to be easily modified. Additionally, a certain level of error-checking would be required to prevent illegal or inconsistent arguments being passed to the program.

According to the original documentation for the `DynamicEM` algorithm written by *Dr. Martin Emms*, there are many possible parameters that may be set. Some of these parameters are required and some are not. Furthermore, some parameters depend on others having already been specified. When designing the GUI, the goal was to account for as many of these parameters as possible, however, not every single one was implemented. Further work on this Final Year Possible could therefore extend to modifying the code to allow for all possible parameters to be specified.

Two different types of widgets were required for parameter specification, namely: `QComboBox` and `QLineEdit`. Figure 4.1 depicts the interface consisting of these two widget types:

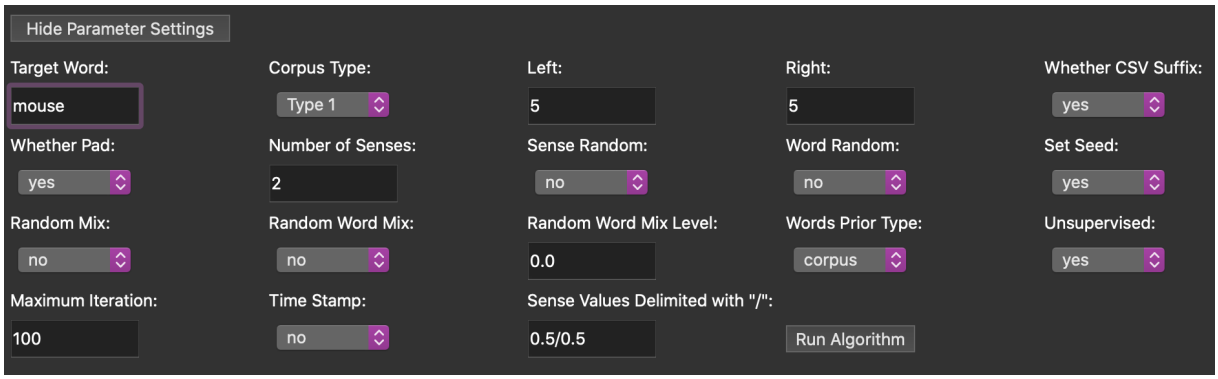

Figure 4.1: User-friendly Parameter Setting

The `QComboBox` widget is used for allowing the user to pick from a finite list of options. For example, a number of parameters have binary *'yes-no'* options. Others, such as *Corpus Type* and *Words Prior Type*, have only a few possible choices and so the `QComboBox` is an appropriate widget.

On the other hand, the `QLineEdit` widget is used for parameters which can't be specified by means of a finite list, for example: numbers, words, etc. This widget takes the form of a simple text box into which the user may type whatever they wish. This means that a large amount of error-checking is required. The value inside this box is a `QString`[1] which must be parsed and checked for validity. One of the more complex cases is the *Sense Values* parameter.

The *Sense Values* parameter represents the initial sense probabilities for each sense, before the algorithm begins its iterations. This is strictly dependent on the number of k senses, (specified in the *Number of Senses* parameter). The amount of probabilities specified must equal k and the values of all of these probabilities must sum to 1. The user is asked to input the values using '/' as a delimiting character. This allows the input to be easily parsed and also happens to be the format expected by the `DynamicEM` code. If any error is detected, (such as an incorrect number of probabilities, non-number values, sum not equal to 1, etc.), the input is overridden with a default. This default is a `QString` of k '/'-delimited equal sense probability values which sum to 1. For example, if $k = 5$, the default input is '0.2/0.2/0.2/0.2/0.2'.

The code snippet Listing 1 illustrates the necessary parsing and error-checking required for this case:

[1] `QString` is `Qt`'s version of string implementation. At times, there must be conversion between the `QString` type and other more familiar types, such as the `C`-style `char*` or `std::string` from the standard library.

```
QString sense_probs = "";
QString vals = sense_vals->text();
QStringList delim = vals.split("/");
bool valid_str = true;
float total = 0.0;
if (delim.length() != curr_num_senses) {
    //Check if different length
    valid_str = false;
} else {
    for (int i = 0; i < delim.length(); i++) {
        float curr_val = delim[i].toFloat(&ok);
        if (!ok) {
            //Check if curr_val is not a float
            valid_str = false;
        } else {
            total += curr_val;
        }
    }
    if (total != 1) {
        //Check if total probabilities != 1
        valid_str = false;
    }
}
if (valid_str) {
    sense_probs = vals;
} else {
    //Invalid string:
    //Set to default (all values are equal and sum to 1)
    double default_value = 1.0 / curr_num_senses;
    for (unsigned int i = 0; i < curr_num_senses; i++) {
        sense_probs += QString::number(default_value);
        if (!(i + 1 == curr_num_senses)) { // if not last sense
            sense_probs += "/";
        }
    }
    sense_vals->setText(sense_probs);
}
```

Listing 1: `App.cpp`: Error-checking for initial values of Sense Probabilities

4.2 RInside package

Once the DynamicEM algorithm has finished running, a series of files are generated and stored in a specified output directory. The following list indicates an example of these output files:

1. accuracy_details_<DATE>
2. cat_details_<DATE>
3. em_senseprobs_final_<DATE>
4. generated_sym_table_<DATE>
5. accuracy_details_<DATE>
6. param_details_<DATE>
7. sense_distribs_<DATE>
8. version_dynamicEM_<DATE>

The original way in which analyses were conducted on data generated from the DynamicEM algorithm was using R code. R is a programming language used for statistical computing and graphics. A file named plotting_functions.R was supplied to facilitate these analyses, but the way in which it was used involved a number of steps. Firstly, an R shell would be started, then the path to this plotting_functions.R file would be given as a source and finally, a function within this file would be called, for example, plot_senses_on_iteration_frm_file(File, Iteration, Title). The data used for graph generation was contained within the sense_distribs_<DATE> file, (item (7) above).

Even though there are only a few steps, it could possibly prove tedious for the analyst to repeat multiple times whenever they want to generate multiple graphs. Therefore, we sought to streamline this process by eliminating these steps and providing a more user-friendly interface for graph generation. To this end, the first consideration was finding a means to merge two fundamentally separate processes: the main GUI program execution in C++ and the running of R code within a shell. This was accomplished via the RInside package.

RInisde was created by *Dirk Eddelbuettel, Romain Francois* and *Lance Bachmeier*. (Available at: https://github.com/eddelbuettel/rinside, license: GPL >= 2). This package allows for Embedding R inside C++. The code snippet Listing 2 illustrates how this embedding is created:

```
RInside R(argc, argv); // create an embedded R instance

//String Declarations:
R["plot_funcs_src_path"] = "./Animation/animate_plots.R";
R["data_src_path"] = src;
R["word"] = word;
R["iteration"] = iteration;

//R Code:
std::string RCode =
    "source(plot_funcs_src_path); "
    "p <- plot_sens_on_iter(data_src_path, word, iteration); ";

R.parseEvalQ(RCode);
```

Listing 2: `Animation.cpp`: Creating an Embedded `R` Instance in `C++`

4.2.1 `RInside` directly within `Qt`

The `RInside` package also provides a means for embedding `R` code directly into a `Qt` application. This may be particularly useful as it would grant access to `R` functionality throughout the entire GUI. In the current implementation however, we opted not to use this approach for running `R`. All the `R` code is contained within `animate_plots.R` which is called from the `animation.cpp` file. The executable, `animation`, is simply run from the GUI code via `C++`'s `system()` call. As such, passing an embedded `R` instance into our entire `Qt` application, as briefly illustrated below, was deemed unnecessary.

However, this method could certainly be advantageous in different contexts. For example, although `C++`'s `system()` call is appropriate for our needs, a fully-fledged GUI to be released with the aim of being cross-platform compatible would likely refrain from using this call. This is because inconsistent behaviour may occur depending on the given system running the code. In this case, it could be preferable to have all the `R` code contained within the `Qt` application. This would be a relatively simple adjustment to make, thanks to `RInside`'s feature of allowing embedding of `R` within a `Qt` application.

The modifications to `main.cpp` would be as shown in Listing 3.

```
#include ...

int main(int argc, char *argv[]) {

    RInside R(argc, argv); // create an embedded R instance

    QApplication a(argc, argv);

    App app(R); // pass R instance by reference

    app.show();

    a.exec();

}
```

Listing 3: `main.cpp`: Passing an Embedded `R` Instance to `Qt` Application

Thus, after passing the `R` instance to `App.cpp`, we could simply use the code as is currently written in `animation.cpp`. This would eliminate the need for `animation.cpp` and avoid having to use a `system()` call, as shown in Listing 4.

```
#include ...

// Initialise member variable with R instance
// before constructor body executes:
App::App(RInside & R) : m_R(R) {
    ...
    //String Declarations:
    m_R["plot_funcs_src_path"] = "./Animation/animate_plots.R";
    m_R["data_src_path"] = src;
    m_R["word"] = word;
    m_R["iteration"] = iteration;

    //R Code:
    std::string RCode =
        "source(plot_funcs_src_path); "
        "p <- plot_sens_on_iter(data_src_path, word, iteration); ";

    m_R.parseEvalQ(RCode);
    ...
}
```

Listing 4: `App.cpp`: Running `R` Code from within `Qt` Application

As stated before, we decided not to opt for this method of integration since using `C++`'s `system()` call is not a problem for the purposes of this Final Year Project. In its current form, keeping the `animation.cpp` and `animate_plots.R` files separate from the GUI code provides a convenient level of abstraction.

4.2.2 Visual Analysis

A key requirement for this GUI was to facilitate graph analysis from within the same interface used to run the `DynamicEM` algorithm. Now that we had overcome the challenge of running `R` code from within `C++`, plotting graphs could be easily accomplished. A `QPushButton` widget could simply be programmed to execute some `R` code when pressed. Thereafter, another `QPushButton` widget could be programmed to display a number of graphs side-by-side within the interface window. Each of these endeavours entailed their

own unique challenges.

Graph Generation

As previously mentioned, the creation of animations representing the progression of the probability distributions was favoured over the static graphs generated in the original implementation. Therefore, an entirely new piece of `R` code (`animate_plots.R`) was written to replace the original file (`plotting_functions.R`) which could only generate static graphs. This new file was designed in such a way as to be able to utilise the `DynamicEM` algorithm's output data in its original format. In this way, no modification of the internals of `DynamicEM` would be required. For creating plots of *Sense Probabilities* against *Year*, the output file `sense_distribs_<DATE>` was used.

A `QPushButton` labelled *'Generate Animation'* was created for the purpose of generating non-static graphs. There were a few considerations relating to programming this widget.

- In order to ensure that relevant data exists, the user generates an animation via a request which can be thought of as a vector containing three parameters. As such, a request R would take the form of $R = [w, k, i]$ where w is the target word, k is the number of senses and i is the number of iterations. For example, a user might request an animation for $R = [mouse, 3, 100]$. There should be a check to determine whether data exists for a run of `DynamicEM` wherein those three parameters were set in this way.

- The current implementation obtains the request information from the same widgets in the parameter-setting interface. This means that the analyst can easily generate an animation immediately after a run as the relevant parameters will already be set. However, it could be argued that utilising the same widgets for these two functions (setting parameters for `DynamicEM` and requesting an animation) is non-intuitive. Perhaps future work on this project could extend to creating separate widgets to specify what animation to generate.

- The animation code outputs an animated `gif` in the form of "`<target word>-<number of senses>-sens-<number of iterations>-iter.gif`". When generating a new animation, there is a check for whether a `gif` already exists whose name matches the request information. If so, the animation does not need to be generated and the user is thusly informed. If not, then there is a check for whether relevant data exists.

- The GUI code ensures that a new directory for outputting data is automatically created when running `DynamicEM`. The name of this directory takes the form of "`<target word>-<number of senses>-sens-<number of iterations>-iter`". So, when processing a request for an animation, the code only needs to check for the existence of a relevant directory whose name matches the request information. If no matching data is found, the user in alerted. Otherwise, the execution of the `R` code is started to generate the graph. (The generation process tends to take some time which gives rise to a blocking issue. This was solved via *multi-threading*, as explained later).

Rendering Graphs

Having implemented the functionality for user-specified graphs to be generated from within the GUI, there was then a need to render these graphs onscreen. This involved displaying animated `gif`s within the window according to the user's request. Firstly, a `QPushButton` labelled *'Which Graphs?'* was created. Its function was to simply display a `QMessageBox` with a list of all graphs generated so far. This allows the analyst to quickly check what graphs are already available for inspection.

Next, a `QPushButton` labelled *'Show Graphs'* was created to allow the analyst to specify which graphs to display within the interface for visual comparison. The maximum number of graphs that may be selected is 5 in the current implementation because rendering any more than this may make the graphs too small to be inspected. Prompting the user to select files required displaying the graph directory (provided it is non-empty) in such a way that files within this directory may be selected and processed. To this end, the `QFileDialog` class was utilised. The code snippet Listing 5 illustrates how the file dialogue box was rendered and how the user's input was subsequently processed.

```
void App::show_graph() {
    //Check if any animations exist
    ...
    bool no_graphs_selected = false;
    int orig_num_graphs = curr_num_graphs;
    QStringList file_names = QFileDialog::getOpenFileNames(this,
        tr("Open File"), "./Graphs", "All files (*.gif)");
    if (file_names.length() == 0) {
        no_graphs_selected = true;
    } else if (file_names.length() <= 5) {
        curr_num_graphs = file_names.length();
    } else {
        curr_num_graphs = 5;
        QString txt = "Too many files." + "\n" +
            "Defaulting to maximum number of files = 5";
        info->setText(txt);
        info->exec();
    }
    if (!no_graphs_selected) {
        //Stop previous gifs and render the new ones
        ...
    }
}
```

Listing 5: `App.cpp`: Rendering File Dialogue Box to Prompt User to Select Animations

A screenshot of the resulting file dialogue box is depicted in Figure 4.2.

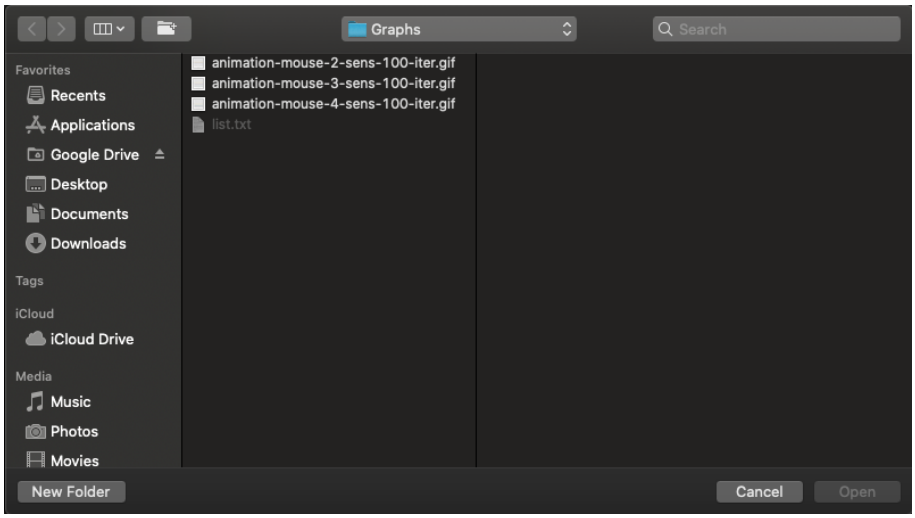

Figure 4.2: File Dialogue Box prompting user to select graphs

For rendering the animated `gif`s, the `QMovie` and `QLabel` classes were used. A `QMovie` object is initialised by passing the `gif` file. Then a `QLabel` object is initialised and instructed to expect contents of type `QMovie`. The two widgets can now be linked together. The position of the animation within the interface is dictated by the geometry of this `QLabel`. The `QMovie` class is particularly useful as it provides important functionality for controlling the animations, such as starting, stopping, setting the speed, etc.

To afford the analyst more control over how the graphs are presented, a `QPushButton` widget labelled *'Display Final Iteration'* was created. The function connected to this button sets the current frame of the `gif` to its final frame and then stops the animation. This essentially produces a static graph, like the original implementation. As such, the analyst can now decide whether they want to see the animation displaying the progress of `DynamicEM` or just inspect a static graph representing the result of the `EM` process, (the final iteration).

Furthermore, a `QSlider` widget labelled *'Control Speed of Graphs'* was created. The `QSlider` class is used to render a sliding widget within the window which allows the user to fine-tune a value between a given range. In the case of our GUI, the `QSlider` object is a horizontal sliding widget used for setting the speed of the graphs to a value between 0 and 200. The speed value is a percentage, so the maximum speed = 200% i.e. two times faster and the minimum speed = 0% i.e. stopped. Figure 4.3 is a screenshot showing how two graphs can be viewed side-by-side. The graph-related buttons are visible on the left hand side of the window as well as the speed control slider and the quit button, which exits the GUI. The graphs in question plot the *Sense Probabilities* against *Year* for the

target word, 'mouse'. The graph on the left represents the final (100^{th}) iteration of a run of `DynamicEM` for which the number of senses (k) hyperparameter was assigned the value of 2. By contrast, the graph on the right represents the exact same run, except that the number of senses (k) hyperparameter was assigned the value of 3.

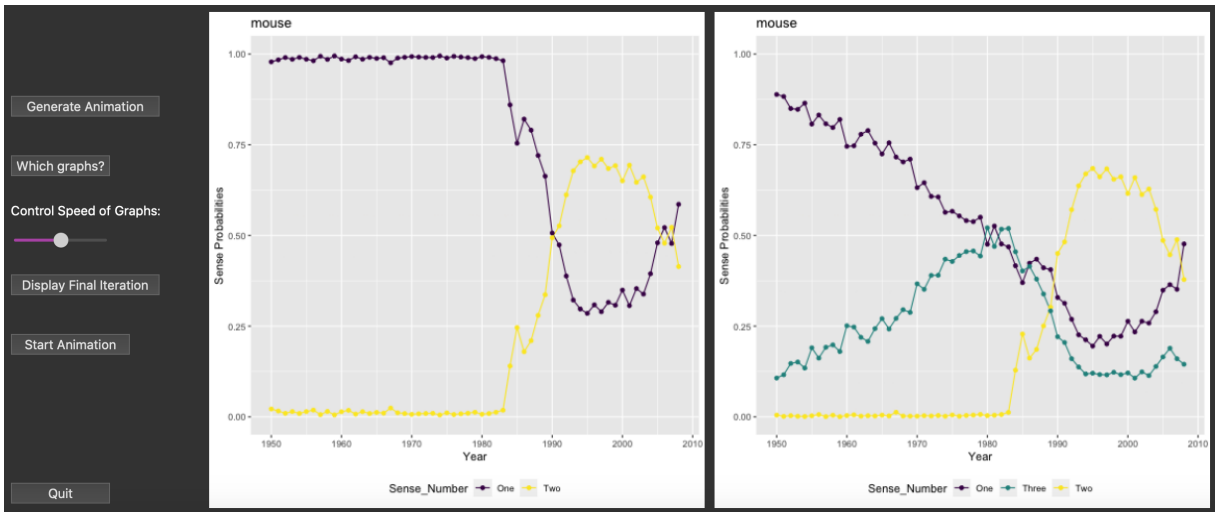

Figure 4.3: Two Graphs Displayed Side-By-Side

Figure 4.4 depicts three graphs displayed side-by-side. It is the same as Figure 4.3, except that the rightmost graph is an additional graph representing a run for which the number of senses (k) hyperparameter was assigned the value of 4.

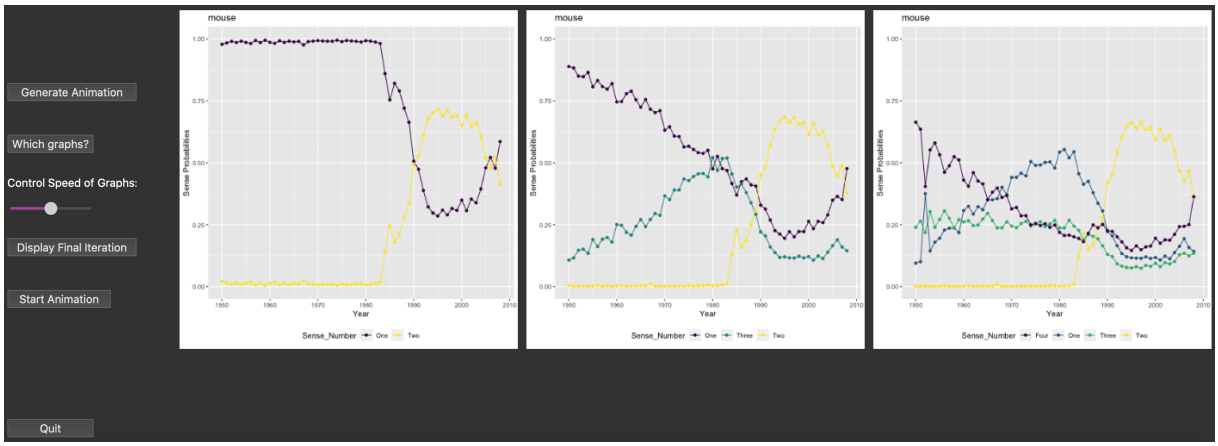

Figure 4.4: Three Graphs Displayed Side-By-Side

Figures 4.3 and 4.4 illustrate the advantage of the system in relation to analysing the hyperparameter space. Side-by-side comparison of graphs representing similar runs with different hyperparameter values offers one the ability to determine which hyperparameter

values are most suitable. In this case, an analyst might decide that $k = 3$ is a good value for the number of senses since $k = 4$, (the rightmost graph), is not particularly clear.

4.2.3 Blocking Issue

In order for a Graphical User Interface to function, it relies on looped execution. This means that the program can constantly detect changes to the interface by listening for events. Once an event is triggered, an appropriate function may be called to run specific code and/or to update the interface. In `Qt`, the execution of this loop is started by the call `a.exec();` in `main.cpp`. Events are known as *signals* and the functions they may be connected to are called *slots*. For example, note the following line:

```
connect(animate,SIGNAL(clicked()), this, SLOT(gen_anim()));
```

This line specifies that if, at any point throughout the execution of the program, the *'Animate'* button is clicked, (triggering a *SIGNAL*), the function `gen_anim()` will be called, (the associated *SLOT*).

Having highlighted the importance of the looped execution in relation to GUI functionality, we now address the issue of blocking. This problem refers to the pausing of the execution loop and is a significant consideration for any GUI design. It commonly arises because of the program having to wait for intensive computation to terminate. The user is entirely unable to utilise the interface because *signals* cannot be detected until the execution loop resumes.

The `DynamicEM` algorithm must process large corpora and go through numerous iterations in order to obtain acceptable results. This means that it will often take a significant amount of time to run. As such, if the blocking issue is not properly addressed, our GUI would risk being idle for long periods of time and the benefits of any user-friendly features would essentially be rendered negligible.

4.2.4 Multi-threading

One of the solutions to this blocking concern is *multi-threading*. As described by (Blanchette and Summerfield, 2006, p. 381):

"In a multithreaded application, the GUI runs in its own thread and the processing takes place in one or more other threads. This results in applications that have responsive GUIs even during intensive processing."

Following this idea, we decided to integrate the necessary framework to allow tasks to be run in separate threads. In order to accomplish this, the `QThread` class was utilised. The files `MyThread.h` and `MyThread.cpp` were created inheriting from `QThread`. Whenever there was a call to start running heavy computation, (such as the `DynamicEM` algorithm):

```
system(command);
```

It was replaced with the following:

```
mThread[x] = new MyThread(const char * command);
mThread[x]->start();
```

The `start()` function is inherited from the `QThread` base class and automatically calls the `run()` function. The `run()` function calls `exec()` by default which begins execution in a new thread. When `run()` has completed, the execution of the thread is terminated. As such, the default `run()` function is typically reimplemented to handle the specific code to be run in a separate thread. This is the case with our `MyThread` class:

```
void MyThread::run() {
    system(command);
}
```

4.3 Summary

In this section, we have discussed the challenges that were faced in designing our GUI.

Firstly, to become acquainted with the `Qt` library, different widgets were explored, such as `QPushButton`, `QLineEdit`, `QLabel`, etc. Functionality was implemented for an analyst to specify parameters before running the `DynamicEM` algorithm. A subset of all possible parameters were accounted for, which required a significant amount of parsing and error-checking to ensure that the user avoids passing impermissible arguments to the code.

Next, we encountered the challenge of running `R` code from within `C++`. The original method for generating graphs required starting an `R` shell and one of our aims was to streamline this process into one click. To this end, the `RInside` package was utilised. The `system()` call was employed to run a piece of `C++` code (`animation.cpp`) from the command line and it was inside this code that the `R` code was run. We briefly discussed

the possibility of embedding the R instance directly into the GUI and the potential benefit that this could provide. However, we did not opt for this approach.

Consolidation of both training the EJ model and visualising its results in the same interface was a key goal of our implementation. Now that we had added the necessary framework for running R code, we could allow for graphs to be generated with the press of a QPushButton. Checking that the generation request was valid was another challenge – the data had to be matched based on the target word, the number of senses and the number of iterations. Next, functionality was implemented for the analyst to be able to check how many graphs are available and select some for inspection. Furthermore, a new piece of R code (animate_plots.R) was written to generate animated gifs from the DynamicEM algorithm's output data. Rendering these animations onscreen was accomplished via the QMovie and QLabel widgets. Additionally, a QSlider widget was created to afford the analyst control over the graph speed.

Finally, we addressed the issue of blocking wherein the GUI becomes unusable when awaiting termination of intensive computation. To overcome this issue, we utilised a multi-threading framework available via the Qt class QThread. The files MyThread.h and MyThread.cpp were created to handle the running of processes in separate threads. Once integrated with the code, this framework was used to allow DynamicEM and animation.cpp to be run in separate threads, as both of these pieces of code can take some time to terminate. This resulted in the GUI always being usable, even during intensive computation. The analyst can carry out a visual comparison of graphs representing different runs, (for example to determine the best value for the number of senses k hyperparameter), while more trials are occurring in the background.

Chapter 5

Evaluation

5.1 Revisiting Requirements

After implementation, we now revisit our requirements checklist:

1. Provide a means for the user to easily specify parameters without needing to modify a long string. ☑
2. Allow analysis to be carried out while the machine learning process runs. ☑
3. Provide some way to seamlessly generate graphs from data obtained via the algorithm. ☑
4. Allow for smooth side-by-side visual analysis of graphs. ☑

These general requirements have been successfully implemented as discussed in Chapter 4. There are however some implicit requirements that one could argue have not been addressed, for example, status reporting of the `DynamicEM` algorithm within the GUI, (as opposed to the console).

From examining the use case scenario in Section 3.4, one can see that the analyst must wait for intensive computation to terminate. Even though the application does not block and is fully functional during this time, (thanks to multi-threading), there must nevertheless be some sort of indication of when a process has finished. In the current implementation of our GUI, all processing output is displayed in the console. In fact, there is nothing within the GUI that informs the user as to the progression of a process. Although the analyst may check the console, this is not an ideal situation as it renders the GUI less useful. One of the key aims of this Final Year Project was to integrate multiple

processes into the same interface and remove the need for typing commands into a shell.

Providing updates within the GUI about the progression of processes could be a direction of further work. This endeavour may not be entirely complex due to the functionality associated with the `QThread` class. There are functions intended to allow threads to be queried, such as `isRunning()` and `isFinished()`. These two methods are rather similar. The former returns a Boolean value indicating whether a given thread is running or not, whereas the latter returns a Boolean value indicating whether the thread in question is terminated or not.

Furthermore, since the `DynamicEM` algorithm already outputs updates to the console, this information could be directly used to inform the analyst of its progression in a more graphical way. For example, one could potentially implement a progress bar showing the percentage completion of the algorithm. `DynamicEM` prints to the console at every iteration, so perhaps this progress bar could also be updated at every iteration according to the simple rule: `curr_progress = (curr_iteration/max_iteration)*100`. An animated `gif` of a loading symbol could also be implemented fairly easily following the same methodology used for displaying animated graphs in the window. Again, since the use case scenario implies that the analyst is informed when a process has terminated, one could argue that this implicit requirement was not fulfilled. The console log style of progression updates are not necessarily expected in the context of a GUI.

Chapter 6

Conclusions & Future Work

6.1 Future Work

6.1.1 Generic Parameter Specification

The current implementation of the GUI allows the analyst to pass a finite number of arguments to the code. As mentioned earlier, this is not an exhaustive list – it was decided to not implement every single possible parameter. Extending the GUI to account for these unimplemented parameters is a clear direction for further work on this project. This endeavour could be accomplished in a fairly straightforward manner by mimicking the implementation of the current parameters i.e. making use of `QComboBox` and `QLineEdit` as well as appropriate error-checking.

However, the question arises as to whether one could go about creating a more generic interface allowing the user to specify any number of parameters of any type. This would be applicable to algorithms which have been created in a similar way to `DynamicEM` in that they expect a long string of arguments passed via the command line. The core method of running the algorithm from the GUI could remain the same: using `C++`'s `system()` call, (from within a new thread to avoid blocking). All that would be required is to pass the appropriate command based on valid user-specified data obtained from the `QComboBox` and `QLineEdit` widgets.

Of course, the analyst should be able to select the number of parameters required and provide relevant error-checking information. Based on this, the GUI would ideally render the appropriate number of widgets to be interacted with and be able to parse information from them before passing a command through the `system()` call. Figure 6.1 below is

a flowchart indicating the general steps that might be expected in a GUI allowing for generic parameter-specification.

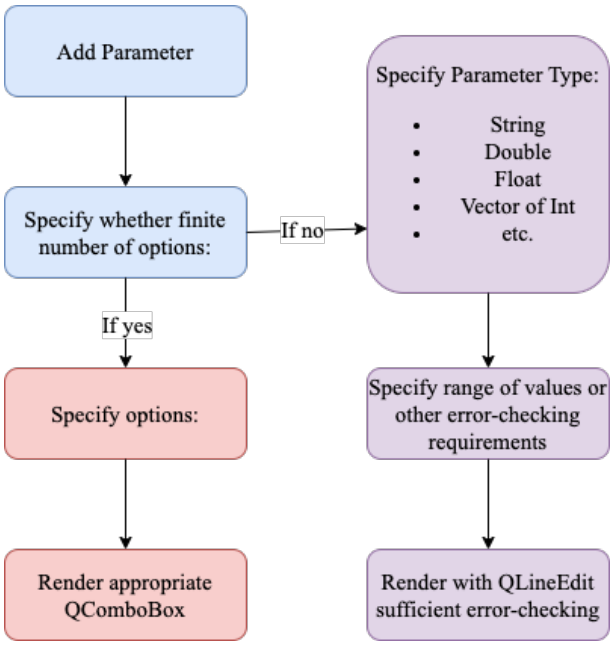

Figure 6.1: Generic Parameter Setting Implementation Flowchart

6.1.2 *Pseudoneologisms*

Emms and Jayapal (2016) introduced the notion of pseudoneologisms for testing word sense emergence. This is an important technique which currently cannot be carried out in our GUI. The reason for this is that the list of possible words only extends to {*mouse, gay, jet, tank, tanks, plane, planes, transcribe, transcribes, transcribed, transcribing*} and also the set of parameters that can be specified for the `DynamicEM` algorithm is not fully implemented. Further work could aim to allow for pseudoneologisms to be incorporated. Specifically, it would be interesting to see if one could develop a way for an appropriate *pseudoword* to be selected for a given time period t. As discussed in Section 2.1.2, for a given range of years t, two unambiguous words are selected, namely σ_1 and σ_2. σ_1 is a word used throughout the entire time period, whereas σ_2 is a word which first emerges at some point t_e. After this, the fake word 'σ_1-σ_2' is used to replace these original words in the corpus. Appropriately selecting these two words σ_1 and σ_2 and replacing their occurrences with 'σ_1-σ_2' could potentially be implemented automatically.

6.1.3 Extension of Features

As mentioned in the previous Section, the analyst is not informed of the progress of intensive computational tasks in any graphical way. They must instead check the console to gain an insight into whether such a task has terminated. One might imagine that graphical progression representation would be a feature of our GUI, so further work could include implementing this. Furthermore, as discussed in Section 4.2.2, the GUI currently obtains the request information from the same widgets used for the parameter specification. This may not be entirely intuitive since running `DynamicEM` and generating graphs from its output are fundamentally separate processes. One could perhaps view these as separate modules. Then, within the interface, there could be some functionality allowing the analyst to toggle between these modules. Additionally, each module might be separated into different windows.

6.2 Overall Conclusions

In this Final Year Project, a Graphical User Interface was created in order to allow an analyst to specify parameters of an `EM` algorithm. This algorithm infers the parameters of a model for detecting diachronic word sense emergence as presented in Emms and Jayapal (2015) and Emms and Jayapal (2016). This model has been referred to as the `EJ` (*Emms-Jayapal*) model based on the names of the authors and the `EM` algorithm is named `DynamicEM`. The background development of the model was discussed and the context of word sense disambiguation and *Expectation Maximisation* was explored. An overview was given of the contemporary algorithms and applications used for hyperparameter optimisation (HPO).

The intended design of the GUI was proposed with an accompanying use case scenario before discussing the various details of the implementation and the challenges they presented. General requirements were satisfied, including allowing the analyst to specify parameters and run `DynamicEM` as well generate animated graphs through `R` code in one click (without launching a shell). Side-by-side graph visualisation was also facilitated and the necessary framework was integrated to ensure that the GUI avoids blocking. The implicit requirement of updating the user on the progression of processes in a graphical way was not addressed. Further work was suggested to improve the system, including generic parameter specification, allowing for the incorporation of pseudoneologisms and other extra features.

Bibliography

Blanchette, J. and Summerfield, M. (2006). *C++ GUI programming with Qt 4*. Prentice Hall.

Dempster, A. P., Laird, N. M., and Rubin, D. B. (1977). Maximum likelihood from incomplete data via the em algorithm. *Journal of the Royal Statistical Society. Series B (Methodological)*, 39(1):1–38.

Emms, M. and Jayapal, A. (2015). An unsupervised EM method to infer time variation in sense probabilities. In *Proceedings of the 12th International Conference on Natural Language Processing*, pages 89–94, Trivandrum, India. NLP Association of India.

Emms, M. and Jayapal, A. K. (2016). Dynamic generative model for diachronic sense emergence detection. In *Proceedings of COLING 2016, the 26th International Conference on Computational Linguistics: Technical Papers*, pages 1362–1373, Osaka, Japan. The COLING 2016 Organizing Committee.

Feurer, M. and Hutter, F. (2019). *Hyperparameter Optimization*, pages 3–33. Springer.

Maldonado, A. and Emms, M. (2012). First-order and second-order context representations: geometrical considerations and performance in word-sense disambiguation and discrimination. In *JADT 2012 : 11es Journées internationales d'Analyse statistique des Données Textuelles*, pages 676–686.

Maretic, H. P. and Frossard, P. (2018). Graph laplacian mixture model. *CoRR*, abs/1810.10053.

Mark Davies (2022). Corpus of Historical American English (COHA).

Michel, J.-B., Shen, Y. K., Aiden, A. P., Veres, A., Gray, M. K., Team, T. G. B., Pickett, J. P., Hoiberg, D., Clancy, D., Norvig, P., Orwant, J., Pinker, S., Nowak, M. A., and Aiden, E. L. (2011). Quantitative analysis of culture using millions of digitized books. *Science*, 331(6014):176–182.

Pedersen, T. L. and Robinson, D. (2022). *gganimate: A Grammar of Animated Graphics*. https://gganimate.com, https://github.com/thomasp85/gganimate.

Schütze, H. (1998). Automatic word sense discrimination. *Computational Linguistics*, 24(1):97–123.

Wickham, H. (2016). *ggplot2: Elegant Graphics for Data Analysis*. Springer-Verlag New York.

Yu, T. and Zhu, H. (2020). Hyper-parameter optimization: A review of algorithms and applications.