Implementing Vague and Sarcastic Language Recognition for a Python Based Chatbot

JASON SUMMERS*, Swansea University

CCS Concepts: • **Human-centered computing → HCI theory, concepts and models**.

Additional Key Words and Phrases: datasets, neural networks, natural language processing

ACM Reference Format:
Jason Summers. 2020. Implementing Vague and Sarcastic Language Recognition for a Python Based Chatbot. 1, 1 (September 2020), 38 pages.

Summary.
This paper looks to investigate the problems present in the area of natural language processing (NLP) when dealing with sentiment analysis, specifically there will be a focus on analysing sarcastic and vague language. The analysis on vague language will be approached from the linguistic fundamentals rather than extracted real world data, unlike sarcasm, due to the specificity and limited data within the subject. Nevertheless both linguistic concepts will be analysed using a complex of neural networks capable of giving a prediction for the presence of sarcastic or vague language as well additional information such as sentence type. This paper will also investigate the implementation of these classifying neural networks into a basic python chatbot. The analysis conducted in this project utilizes many fundamental python based NLP tools, mostly from tensorflow, including tokenization, padding and high dimensional word embedding, in order to classify the sentiment of a phrase. There are some immediate issues present with this, both sarcastic and vague language rely on context and external factors such as tone and physical gestures, all of which are impossible to recognize when simply given one phrase in the form of a string. Both classifier models are instead intended to give a prediction of the sentiment of a phrase instead of an absolute classification. As mentioned the area of vagueness in natural language has limited resources, this means it is necessary to develop our own basic corpus for this project and although the corpus will be based on research in the field, it will undoubtedly come with many limitations. For this reason, it is important to state that this project does not look to revolutionize sentiment analysis but rather to explore NLP techniques and ponder over the implementation of the resultant models into a deep learning chatbot to improve conversational flow.

*Student Number: 903702

Author's address: Jason Summers, 903702@swansea.ac.uk, Swansea University, P.O. Box 1212, Swansea, 43017-6221.

Declaration.
This work has not previously been accepted in substance for any degree and is not being concurrently submitted in candidature for any degree.

Statement 1.
This dissertation is the result of my own independent work/investigation, except where otherwise stated. Other sources are acknowledged by giving explicit references. A bibliography is appended.

Statement 2.
I hereby give consent for my dissertation, if accepted, to be available for photocopying and for inter-library loan, and for the title and summary to be made available to outside organisations.

Signed: *Jason Summers*

Jason Summers, SN: 903702.
Swansea University

Acknowledgements.

I would like to thank Dr. Leigh Clark
for his counsel and guidance throughout
this project.

I would like to also thank my family and
friends for their insight throughout this
project and indeed their support during
the Covid-19 pandemic.

Contents

1

3

1 Introduction **4**

2 The Current State of Natural Language Processing and Sentiment Analysis **5**
2.1 Natural Language Processing with TensorFlow - Ganegedara 5
2.2 Sarcastic or Not: Word Embeddings to Predict the Literal or Sarcastic Meaning of Words - Ghosh et al 6
2.3 Tweet Sarcasm Detection Using Deep Neural Network - Zhang et al 7
2.4 Vague Language 7

3 The Language Model **7**
3.1 Vague Language 7
3.2 Sarcasm 9

4 Corpora **10**
4.1 Sarcasm Corpus 10
4.2 Vague Language Corpus 11
4.3 A Chatbot Corpus 13

5 Our Machine Learning Model **13**
5.1 Why Neural Networks? 14
5.2 The Basics of Neural Networks 14
5.3 Data Preparation 15

6 Classifying Models **16**
6.1 Sarcasm Classifier 16
6.2 VL Classifier 25

7 Analysis **29**
7.1 Sarcasm Analysis 29
7.2 VL Analysis 31

8 Chatbots **31**
8.1 Implementation 33

9 Conclusion and Future Studies **33**

References **35**

1 INTRODUCTION

The word *robotic* is a term often used to describe someone behaving in an unnatural stiff manner, to the point where it is considered an antonym to human-like. But why? What is it about the way computers interact that is so different to the way we interact? And do these differences limit the ease of computer interaction? For everyday people these questions may not seem relevant but the truth is computers have become an integral component of our day to day lives. Consequently, understanding human computer interaction (HCI) and looking into ways to lower this communication barrier has the potential to improve our productivity as a species.

HCI is an area of computing that refers to how we interface with programs and devices. In terms of hardware this includes inputs such as the keyboard, cameras and microphones, and outputs such as screens and speakers, but what can not be dismissed is how these components themselves interface with a program. The program will need to be able to break down any request into clear logical steps, this however does not reflect the way we naturally interact with each other, particularly regarding the most versatile interaction we employ, speech.

Natural language is the name given to the way us humans mostly interact, it is present in speech and text and is most evident in informal contexts. Its namesake characteristic of being naturally spoken without a great deal of attention being needed for syntax gives it a very interesting property, it is easily understood with the same lack of attention. This results in fast and efficient communication between both parties. This is not typically the case for interactions between humans and computers, these interactions are fundamentally based on and limited by code, the natural language for a computer. Code follows a strict rigorous syntax and is utterly unforgiving in the face of missing information. The result is a one way system where the human has to specify every single aspect of the code with perfect syntax, with limited or no involvement from the computer, in order for the computer to understand what is desired from the user. This is not convenient or efficient and dramatically slows the interaction down. This one way system would be analogous to you barking orders and questions to a friend, and they just shout back either the single answer or a big error. Moreover they would not show an understanding of the context or any hidden meanings behind the question or request. This is not considered to be a conversation, a conversation is a two way transfer of information where both parties understand the context and equally contribute to the subject.

A more conversational interaction between humans and computers would be a faster and more convenient one for the general public. As hinted to before, one factor that harms the use of code is how specific it has to be, there are many instances (that this paper will explore) where the full amount of information is not needed to understand the point or sentiment within a conversation. Another crucial factor of conversational language is the understanding of the sentiment itself, a more enjoyable and beneficial style of communication comes from both parties understanding the true meaning behind what is being exchanged.

There is a more literal sense to conversational HCI, while we have already briefly talked about code, specificity of syntax and the differences between them and natural language, there is a technology that marries the two ideas more than any other. This technology is agents, these are programs that carry out jobs for the user, much like a virtual assistant and in recent years often incorporate human like qualities such as speech. These jobs are casually requested in day to day life hence a good flow for the interaction would be highly valued. The most recognisable modern form of these agents is the Intelligent Personal Assistant (IPA), these may be programs on smartphones such as siri, or as a dedicated agent device, such as smart speakers (e.g. Amazon Alexa and Google Home). The interest and popularity of these technologies has been increasing at a rapid rate with the market for IPA technology being predicted to exceed $9 billion by 2023 [1]. This means that understanding and improving HCI and HAI (human agent interaction) is more relevant than ever.

So how may we improve the conversational flow of a language based computer interaction? The answer is through the use of Natural Language Processing (NLP), this area of computer science focuses on decoding, analysing and generating natural language. There are many forms of NLP but in this paper we will be focusing on the machine learning (ML) aspect, specifically using neural networks to classify and generate forms of natural language. These classifiers should be able to give us an insight into the nature of natural language by identifying key words contributing to particular sentiments. It is the objective of this paper to not only achieve this but to investigate the implementation of such analytical models.

2 THE CURRENT STATE OF NATURAL LANGUAGE PROCESSING AND SENTIMENT ANALYSIS

Before we begin our own investigation, we should have a look at what tools and methods are used in NLP today and a look into similar research in the field of sentiment analysis.

2.1 Natural Language Processing with TensorFlow - Ganegedara

Many basic techniques and tools for neural network (NN) based NLP have already been established, most notably the tensorflow library for python which provides many essential tools in the NN field. Ganegedara uses the tensorflow library to great effect in his book *Natural Language Processing with TensorFlow*[15]. This resource will provide much of information for some of the initial steps for the practical side of this project. First let us break down the possible types of analysis we can conduct on our data.

Ganegedara list 8 distinct acts that can be accomplished within NLP:

- **Tokenization** - separating a text corpus into atomic units, e.g. words.
- **Word-sense Disambiguation (WSD)** - identifying the correct meaning for a word.
- **Named Entity Recognition (NER)** - extracting entities (person, location, etc.)
- **Part-of-Speech (PoS) tagging** - assigning words their respective parts of speech.
- **Sentence/Synopsis classification** - categorizing content, e.g. political, technology.
- **Language generation** - training machine learning (ML) model
- **Question Answering (QA)**
- **Machine Translation (MT)** - language translation.

The quality or focus of the data we are able to find will dictate what processes we are able to perform on it. However, one of the many benefits of NNs is that their complex structure and learning process allows for the recognition of individual features and abstract patterns within a given input. This means that a NN could be used to identify sub-phrases and potentially even PoS, thus reducing the amount of data manipulation we have to do ourselves. One particularly relevant type of NN that Ganegedara outlines is a convolutional neural network (CNN). These NNs have pooling layers to help reduce dimensions and more confidently find strong features, some pooling layers even find patterns in variable locations. We will save the specifics of how NNs work and how we train and apply them for a future section.

2.2 Sarcastic or Not: Word Embeddings to Predict the Literal or Sarcastic Meaning of Words - Ghosh et al

This paper focuses on collecting a set of target words that can have either literal or sarcastic meanings depending on context, and given an utterance and a target word, how to automatically detect whether the target word is used in the literal or the sarcastic sense. Ghosh et al describes this as the Literal/Sarcastic Sense Disambiguation (LSSD) task [16].

Data Collection:
Ghosh et al proposes two techniques to collect these target words, the first being crowdsourcing and the second being unsupervised alignment techniques to detect semantically opposite words/phrases. Turkers were asked to re-phase a sarcastic message into what they imagined it to literally mean.

- Sarcastic message:
  - I am so *happy* that I am going back to the emergency room.
- Examples of Turker interpretations:
  - I *don't like* that I have to go back to the emergency room again.
  - I am so *upset* I have to return to the emergency room.
  - I'm so *unhappy* that I am going back to the emergency room.

This allowed Ghosh et al to detect that "happy" can be aligned to "don't like", "upset" and "unhappy", thus revealing that "happy" can be considered a target word. With this in mind 1000 sarcastic messages from twitter were given the same treatment, with Turkers (given clear instructions and examples) re-phrasing the entire tweet. The use of Turkers, if needed, is most likely not suited for this project due to both budget and time.

Ghosh et al also used a collection of algorithms to aid with target word identification. One core algorithm was a co-training algorithm for paraphrase detection developed by Barzilay an McKeown [4] which allowed them to acquire 367 pairs of paraphrases. They then used Moses software [20] to extract lexical translations on the 367 pairs and disregarded any that resulted in a lexical translation score lower than 0.8. This resulted in 80 semantically opposite paraphrases.

Analysis:
One of the more interesting approaches conducted by Ghosh was to treat the LSSD task as a binary classification task to identify the sarcastic or literal sense of a target word. They used the libSVM toolkit developed by Chang and Lin [7] and a *CMU Tweet Tokenizer* [12] to complete this task. SVM here stands for a Support Vector Machine, a ML algorithm that is typically used for supervised learning but can be used for clustering data. Using this method and by varying the word embedding technique used, they were able to reach a F1 score (a test of a models accuracy) of 96-97%.

Overall Ghosh et al gives us an extensive insight into how to approach a sarcasm classifier and provides some thoughts on how to develop and handle our own corpus. The idea of using a binary classifier for sarcasm identification is a simple but seemingly effective one, and ma be a viable approach for this project too.

2.3 Tweet Sarcasm Detection Using Deep Neural Network - Zhang et al

Now we will take a very brief look at an example of how researchers are looking beyond simple corpus based classification models in an incredibly ambitious study. As we will later observe, sarcasm has a deep and fundamental connection to context, something that is often hard to incorporate into typical corpora, but Zhang et al look to integrate past tweets into its analysis of a subject tweet [27]. More specifically they look at the difference between a discrete model (using discrete feature vectors) and a neural model. Both models include "contextual tweets" in the inputs along with a target or subject tweet. Unlike the discrete model, the neural model explores low dimensional dense vectors as input and utilizes pooling layers. This is all with the aim of increasing accuracy, which it does, Zhang et al reports a final accuracy of 90.74% for the neural model compared to 79.29% for the discrete model.

This has only been a brief summary as this paper goes into very deep and complex discussions around model optimization, but it provides a different approach to the seemingly classic binary classification approach. The reader may wish to read the entire original paper as it really does examine one of the most important but also overlooked components of sarcasm.

2.4 Vague Language

There are many investigations into the area of sarcasm including the extraction and analysis of sarcastic comments on internet forums and social media, but the same level of investigation has not been given to VL. It appears that there is limited research into the area of VL identification from internet sources, or any other source for that matter. The majority (of what few studies were able to be found) of analytical research seemingly occurs in a multi-language context with scarcely available corpora for common use. Thus this means there is somewhat of a 'gap in the market' for a ML based classifying model for VL, hopefully this project can attempt to fill this gap (at a basic level) and begin to identify its potential in chatbots.

3 THE LANGUAGE MODEL

As previously mentioned this paper will focus on two important aspects of natural language, the lack of necessity for precise information and the understanding of subtle sentiments behind phrases. In particular this paper will focus on a key example from both aspects, vague language (VL) and sarcasm. In this segment, to help with our analysis, we will begin looking to develop an understanding for both key examples. It is not the objective of this paper to make significant new insights on the subject of linguistics, but merely to aid us for our analysis and to give us insight into the potential applications of vague and sarcastic language.

3.1 Vague Language

The first tool here is VL, this is a central feature of daily language, both spoken and written. It can be used to express something that has a meaning that arises from intrinsic uncertainty, this is because it relies on common idiomatic knowledge and a mutual understanding of the context of the conversation [10].

3.1.1 Some vague expressions in English, Dr J.M. Channell, 1983.

Channell delivers a deep look into the properties of VL in her 1983 thesis and speaks to its importance saying "a complete theory of language must have vagueness as an integral component" [8]. Channell states that there are many ways to be vague, one might employ "hedges" such as *virtually* or *it seems that*, or they may use agentless passives [8]. These phrases are seemingly a product of informality in that they expose the lack of required precision when simply conversing with a friend. In a formal environment, such as work, it is expected that one does not compromise a project or effort with imprecision resulting from VL. She goes on to list two distinct groups of common vagueness that lacks percision;

Type A: Number Approximations:

- *about n*
- *approximately n*
- *around n*
- *n or so*
- *n or m*

where n and m are real numbers.

Type B: 'Tag' Approximations:

- *X and things like that*
- *X and that*
- *X or anything/something like that*

where X (to be referred to as a tag for this paper) may be replaced with various verbs, nouns and descriptors such as, for example, gaming, fish and smart.

From these examples we can see that VL can revolve around delivering information in the form of ranges, this could be the result of genuine ignorance or, more significantly, a quick way of referring to multiple tags quickly. It is clear that no matter the cause, it is just the *gist* that the user wishes to impart rather than an exact meaning. Channell expands on this suggesting "that a vague utterance is one which cannot be assigned an exact meaning, even with recourse to context". This lack of an exact meaning present in VL suggest that a true analysis of VL requires a deeper look outside of semantics. Channell suggests that if a universal model for understanding VL exist it must include a look into the pragmatics of the phrase as well as the semantics. The analysis of the pragmatics within a conversation will be interesting as context will likely be a statistical propagation within the NN training. For now let us focus on understanding pragmatics in VL further before we tackle the implementation.

3.1.2 Pragmatic Competence: The Case of Hedging, B.Fraser, 2010.

Fraser helps to define the area of VL by investigating the area of hedging, this is still effectively VL but we shall still evaluate Fraser's outlook and compare it to our current model of VL. He gives the definition:

> "Hedging, a rhetorical strategy that attenuates either the full semantic value of a particular expression, as in *he's **sort of** nice*, or the full force of a speech act, as in *I must ask you to stop doing that*". [14]

This definition paints a more detailed picture on where and how VL affects a phrase. First of all a new variable arises from the idea of semantic attenuation, by how much is the meaning warped? and is it positive or negative attenuation? Fraser deals with this by introducing three categories to classify possible hedging:

- Propositional hedging
- Reinforcement
- Speech act hedging

Propositional hedging is the category that we have looked at the most in this section thus far. It has positive semantic attenuation, this means that the semantic value is weakened by hedging. Reinforcement is the opposite, it has negative attenuation, the semantic value of the phrase is strengthened. This warping of meaning is actually to the benefit of the semantic value, hence reinforcement is considered not to be a true category of hedging. Nevertheless we can still be mindful of its existence in our model. The final category here is speech act hedging, this is a case where hedging is used to show the sincerity of a speech act despite the speech act's actual semantic meaning. An example of speech act hedging is "**can you** please leave?", despite being phrased like a question it is used to convey a request. Speech act hedging is an interesting area of hedging as it is used to deliver a well defined pragmatic meaning of some importance. Propositional hedging on the other hand is used in scenarios where only the gist is required.

These categories will now be the centre of our VL model as they cover informal approximations and, most importantly, the idea of a hidden meaning (semantically speaking).

3.2 Sarcasm

Now for something that most people will likely knowingly experience on a daily basis, sarcasm. Sarcasm has a different effect within natural language than that of VL but shares the property of giving potentially hidden messages. Instead of providing someone with the gist of a sentiment like VL, sarcasm is simply known to be someone meaning the opposite of the literal meaning of a phrase. But like VL, let us first look into sources to confirm and strengthen our reasoning and develop a model that we may compare to our analysis.

3.2.1 Sarcasm, Pretence and The Semantics/Pragmatics Distinction, E. Camp.

Camp delivers a valuable discussion on the nature of the subject by evaluating both ours and the common definition of sarcasm, that being that it is the case of a speaker meaning the opposite of what they say, and develops it further. Camp looks into typical Gricean theory, which states that that in speaking sarcastically, a speaker exploits a mutually shared assumption that he could not plausibly have meant what he said [18][6]. Immediately this adds an element logical evaluation to the recognition of sarcasm, where the evaluation of how plausible the raw semantic sentiment is reveals the existence of sarcasm. Joshi et al expands on this idea, describing how one may evaluate how similar two sides of a comparative simile are, giving the example of "*A women needs a man like a fish needs a bicycle*"[19]. The components of the phrase can be analysed using word vector similarity scores, revealing (through its lack of similarity) that the chance is, a fish does not need a bicycle, hence we may now conclude that the sentiment behind the first relation was a sarcastic one. Camp creates her own model by outlining five core types of sarcasm:

Sarcasm and Verbal Irony. Firstly Camp looks into the relationship between sarcasm and verbal irony, particularly where they overlap. She states the best characterization of the broad genus of verbal irony derives from Kumon-Nakamura et al's (1995) "allusional pretense" theory [21]. Their view consists of two claims, firstly, ironic utterances are allusive, in the sense of "call[ing] the listener's attention to some expectation that has been violated in some way"; where this violation of expectation itself entails "a discrepancy between what is expected (what should be) and what actually is". Typically, a speaker draws attention to this discrepancy in order to communicate a negative evaluation of the actual circumstances; but as Kumon-Nakamura et al note, the expressed attitude may also be positive. Secondly, ironic utterances involve pretense, in the sense that the speech act is presented as not being straightforwardly genuine or sincere [6].

Propositional Sarcasm. Camp describes how the most straightforward cases of sarcasm are those in which the sarcasm's scope is directed toward some proposition to which a sincere utterance would have committed the speaker. Camp goes on to say that this proposition evokes a situation at one extreme of an evaluative scale,

typically the positive end, and by pretending to assert this proposition, the user implicates the contrary. An example of this form: “James must be a real hit with the ladies”.

Lexical Sarcasm. This form of sarcasm is described to be a more extreme version of propositional sarcasm through having a tighter connection to an evoked evaluative scale. Where propositional sarcasm may be pragmatically evoked, lexical sarcasm is evoked through the inclusion of the extreme end of a conventionally associated scale, again often being positive in nature, e.g. “genius” and “brilliant”.

Like-prefixed Sarcasm. As the name suggest, this form of sarcasm occurs when the phrase begins with the term “like” or even “as if”, Camp describes how these terms are used to target a proposition, again much like propositional sarcasm, but are specifically applicable to declarative sentences. An example is “Like Alan has any money”. Users of this form of sarcasm cannot pretend to have intended to fully claim the content of their proposition, and thus this becomes one of the easier forms of sarcasm to diagnose.

Illocutionary Sarcasm. Finally we have illocutionary sarcasm, Camp describes this form of sarcasm as targeting speech acts with an illocutionary force other than the assertion, a key example of this form is “thanks for holding the door”. Camp goes on to describe how “it can also include the full range of implicatures, including especially implicatures that express evaluative attitudes such as pity, admiration, or surprise”[6].

4 CORPORA

4.1 Sarcasm Corpus

Before we can begin to talk about the specifics of deep learning we should first observe what kind of data we are able to source and what we may be able to acquire from any analysis. This section will also look into what lead to the development of the corpora to be used, and how this may affect results. The plan will be to use neural networks to classify a phrase as either sarcastic or non-sarcastic, this means the most ideal corpus will contain a large selection of phrases with **labels** signifying if it as one or the other. Generally speaking, and assuming a credible source, the larger the dataset, the more flexible you can be around training your model and thus the greater the potential for a more accurate model. This is not only due to a larger coverage of our language but also because in this area, especially with the ability to remove as much data as required, it is definitely better to have too much rather than too little.

Oraby et al developed a subset of a corpus originating from a paper by Walker et al, named the Internet Argument Corpus (IAC) [26]. The resultant corpus, *The Sarcasm Corpus V2* [25], contains data representing three categories of sarcasm, we will be using the updated version for the purposes of this paper. The three categories are the general sarcasm subset, containing 3260 posts per class (sarcastic or non-sarcastic) and a hyperbole and rhetorical question subset, both containing 582 and 851 entries per class, respectively. The corpus details the category, class, an ID number and the text. The original IAC is built from internet debate forums on the site 4forums.com. In total there are 390704 posts in 11800 discussions which have been partially categorized into subsets and annotated with details such as a emotionality of argumentation and sarcasm. We will be using the efforts of Oraby et al to satisfy our ambition for a sarcasm corpus but the other categories here have the potential to contribute to a more extensive model that covers more emotions and sentiments. The subjects discussed in this corpus are heated recurring debates in society ranging from the legalization of marijuana and climate change all the way to Gun control laws and the death penalty. There may be an interesting consequence to these topics being so serious, people may be both unlikely to use humorous sarcasm and more likely to use propositional or lexical sarcasm as the extremities of these topics are too drastic for the majority of people to ever begin to agree with. For instance, any positive views directed towards topics generally viewed as being extremely nasty, such as

school shootings, would in turn be generally considered sarcastic. This conflicting potential quality of avoiding humour but exposing unpleasantness through sarcastic agreement can be seen throughout the IAC. Turkers (Mechanical Turk Workers) were asked to judge the response in a debate according to several measures, this resulted in a percentage of sarcastic (and other categories of) interactions in each topic. The percentages range from 0% for the topic of the death penalty and 16% for the topic of marijuana legalization with a mean average around 11.8% for all topics. The legalization of marijuana is arguably the least severe and thus it potentially opens itself up to humorous forms of sarcasm as well as other forms, whereas abortion has the second lowest at 9%, this could be due to the reverse of the effect. Either way this corpus hold plenty of phrases with ideal information to help train our classifier, however intense the topics within it may be.

Despite the presence of such detail in the IAC, our classifier will be specialized in sarcasm recognition and thus we will need a more focused corpus. Oraby et al relays the disadvantages of using an overly diverse dataset when looking at one particular aspect, saying:

> "It is difficult to efficiently gather sarcastic data, because only about 12% of the utterances in written online debate forums dialogue are sarcastic [26], and it is difficult to achieve high reliability for sarcasm annotation." [13][17][3]

Thus the use of a more focused corpus may increase the reliability of the model. Oraby et al uses several methods that filter non-sarcastic utterances in order to skew the distribution toward sarcastic utterances. As well as this, they replicated the pattern-extraction experiments of Lukin and Walker [22] using a weakly-supervised pattern learner to aid in generating a subset of **general** sarcastic and non-sarcastic utterances. Similar methods are used in combination with findings in theoretical literature to annotate further entries into two other categories, those being **hyperbolic** statements and **rhetorical questions**. The result is a dense and detailed corpus containing general, hyperbolic and rhetorical statements/questions with a 50/50 split of sarcastic and non-sarcastic phrases. Each phrase in the corpus has been labelled with a sarcastic label and a sentence type label, having more information will give us more options when classifying.

4.2 Vague Language Corpus

As mentioned, the majority of analytical research for VL seemingly occurs in a multi-language context with scarcely available corpora for common use. This has led to one of the most awkward components of this project, creating our own corpus. The idea of implementing our own VL corpus, partially formed from our own ideas of what VL is and then analysing it to confirm these ideas, is a poor one. So the focus here will the **method**, specifically how we can analyse a self made corpus, and whether we can retrieve useful information that we planted based on a language model supported by independent research.

For VL we will use a similar approach to that of sarcasm, in that the plan is to use neural networks to predict a classification of vagueness for a given phrase. This means the most ideal corpus will reflect the structure of the sarcasm corpus. The corpus will need to contain the phrase, possible types of VL and an indication of whether or not the phrase is vague. The first hurdle is types of VL, we identified two core types of VL, propositional and speech act hedging. Reinforcement will be excluded from our final model due to the demand of making our own corpus already present and the questionable relevance it has to VL as a whole. Speech act hedging propagates itself in a wide variety of forms and contexts but often implies hidden meanings and may rely on the context itself. This makes speech act hedging both deep and broad and thus presents a more daunting and challenging task in generating numerous examples for our corpus. Propositional hedging, on the other hand, is arguably the

simplest form of hedging and, as shown in our language model, has different aspects such as tag and number approximations that we can use to construct our corpus. For these reasons, and the fact that this component of the project is designed to provide a proof of concept for VL recognition, our corpus will be significantly smaller than the sarcasm corpus.

Type of VL		**Number of Entries**
Propositional	number	200
	tag	200
Speech Act		100

half of each category will be vague, the other non-vague

Number approximations: This first section of the corpus is the simplest, and as explored, it is exactly what it sounds like, the quantities or values in a conversation are given as approximations or rounded up/down. This includes subject matters like the time, a count of people or objects, money, etc. There may also be no subject matter at all, where the phrase is a simple response given to an unknown question. So how do we add to the vagueness of number based phrases? When writing the vague phrases we may add words that increase the imprecision of the overall phrase. This includes words such as "*like*", "*maybe*" and "*around*", but also comments such as "*I think*" and "*or so*" can be added to the end of a phrase to create uncertainty. We work in a decimal system and consequently we are drawn to what we often refer to as *round* numbers, typically multiples of ten. However there is an extension to this thought, humans think logarithmically [24] and thus the larger the quantity being rounded/approximated, the larger base unit we use internally for this process. We may use this reasoning when writing phrases discussing large numbers as the larger the number, the wider the range is that we may round up or down to. This, of course, works both ways meaning the smaller the number, the stricter we have to be in terms of vagueness. This idea helps to expose the irrelevance of specific details when considering gargantuan concepts or values. There is little significance in adding one sweet to a packet of 100 as oppose to adding one sweet to a packet of 4. We should aim to reflect this variety of base units for *roundness* in our corpus. Finally there are words we can include to add to the informality of the corpus content. Words like "*yeah*", "*well*" and "*ok*" may exist in a phrase without serving a great deal of surface level function, such as in the phrase "ok so it was around noon...".

Tag approximations. Similar to number approximations, this form of hedging focuses on everything but numbers, such as verbs, nouns and descriptors. For some of these tags it may be hard to imagine what is meant by 'approximations', but in actuality approximations often mean to hint at an area that the tag exists in, and it is the area itself that the user is referring to. With that in mind the rest follows the same rules as number approximations, the difference lies in what words fit where. Additions such as "*sort of*" and "*pretty*" affect descriptors, they may even be responses to unknown questions similarly again to number approximations. Following a tag with "*and that*" invites the receiver to continue thinking along the same theme of the tag stated, the same is true for the word "*stuff*". The final idea we can factor in our corpus is the idea of 'polite' propositional hedging where the user employs additions like "*kind of*" to imply hidden meanings or to simply be polite. With these concepts we can begin to form a corpus for propositional hedging

Speech act hedging. These forms of VL are somewhat only identifiable after taking a closer look at the literal meaning of a phrase. While our ability to ignore this literal sentiment supports our ideas for the relevance of Vagueness in natural language, it does mean that identifying or creating examples is slightly more tedious than the propositional case. Nevertheless, using the writings and examples provided by Fraser [14], we can strengthen our understanding and begin to write our corpus. Fraser gives the examples:

- I **should** apologize for running over your cat.
- I **can** promise that I will never again smoke grass.
- I **must** request that you sit down.

Fraser explains how certain words, such as those here in bold, are responsible for the attenuation of the semantic meaning of an expression. We can see that the verbs following the bold words provide the majority of the sentiment being attenuated. For instance, the first example is an apology, but the strength of that apology has been greatly affected by the word "*should*". This may be due to these words suggesting other sentiments such as the user possibly feeling that running over someone's cat is not a big deal or that it could not have been prevented. As with sarcasm, factors like tone may rectify this or indeed make it worse, so our machine learning model will again only provide a prediction. The last example provides us with a different form of vagueness, the literal meaning suggest the user has to request something but in actuality the user is just being polite. These cases are simpler to sense as there is little chance the user *must* do something. With a better understanding of the language used in speech act hedging we can now populate our corpus, this can be done using expressions utilizing words we have highlighted such as "*should*", "*can*", "*must*", "*have to*", "*could*" and "*would*". These words followed by a verb will be the foundations for this corpus.

Despite our best efforts the corpus will ultimately be flawed. By basing it off of examples and key ideas from our model, the corpus will be following a template, which will limit its content to exactly what we interpret from the theoretical landscape around the subject. On top of that the phrases are constructed by a single person attempting to think up a wide variety of scenarios, resulting in a questionable level of success. These limitations mean that the corpus does not reflect reality and thus the results do not reflect reality. However as mentioned, there may be other note worthy conclusions we may draw from this analysis, flawed as it may be.

In future experiments it may be possible to use this corpus in a partially supervised learning model to generate a larger more varied corpus ready for interrogation. However nothing will compare to a more official corpus such as our sarcasm one.

4.3 A Chatbot Corpus

Although we will not be developing a full chatbot in this project, we can still look at what is required to make one, this includes finding an ideal corpus. The Cornell Movie Dialogs Corpus [11] is a very commonly used corpus in the development of chatbots and for good reason, it is a large metadata-rich collection of fictional conversations extracted from movie scripts. It contains 220579 conversational exchanges from 10292 pairs of movie characters, it also contains metadata surrounding these characters, including genre, name and gender. This data opens it self up to a plethora of possible studies such as what words are most used in what genres and, possibly tying into our project, what genre or gender has been written with more instances of sarcasm (or any other language tool). What makes this corpus ideal for a chatbot is the inclusion of conversation data, where each conversation is described as a list of line IDs. Using these lists and their corresponding sentences, we can link a certain selection and order of a phrase to a particular response, this can then be repeated on the responses response until the end of the conversation.

It is also possible that movie dialogue is clearer and more formal than the language we hear daily, possibly harming the conversational flow. However the data in this corpus is worth the risk due to its large size across many genres and valuable metadata, all of which allow for a customizable genre-based chatbot.

5 OUR MACHINE LEARNING MODEL

In this section we will cover the foundations for the technical aspect of this project. This will involve investigating what the most appropriate possible machine learning techniques are, a dive into how our chosen technique functions, and finally how we will prepare the data for analysis.

5.1 Why Neural Networks?

It is hard to enter into this field and not be greeted with an immense collection of resources and documentation for NN algorithms. The reason for this is due to the versatility and capabilities of NN based approaches. Bengio et al describe the **curse of dimensionality** that arises when one wants to model the joint distribution between many discrete random variables (such as words in a sentence) [5]. NNs are able to reduce the impact of some potentially irrelevant words through its learning process, something we will cover later, and thus reduce the effect of the curse of dimensionality. NNs also allow for a great deal of customizability internally, something we will discuss in the next section. As mentioned there is much documentation on the subject of NNs, nowhere is this more apparent than in the dedicated libraries for python, such as Tensorflow. Finally our data has labels meaning we will be needing a supervised model. Supervised here means we can provide the model with feedback in the form of whether or not it made the right prediction. Although other algorithms are also supported by these libraries, the speed at which they allow you to build an effective NN, plus the previously mentioned reasons, means NNs are the clear choice.

5.2 The Basics of Neural Networks

NNs are named as such because of their slight resemblance to the biological neural systems in our own brains, they are formed of connected components that interact with each other in some way, some connections are stronger, signifying some relationship. In a basic sense the artificial NNs have two components, nodes and edges. Edges carry a signal from one node to the other, this signal will be manipulated within the node mathematically, and the resultant signal will be relayed to other nodes, and so on. Each node will belong to a particular layer of nodes, the first layer is the input layer, and the last is the output layer, between these we have *hidden layers*, the more of these we have, the deeper the learning will be. In a Feedforward neural network (FNN) these layers are only connected to the layer directly before and after it, and each node in the layer is connected to every other node in neighbouring layers.

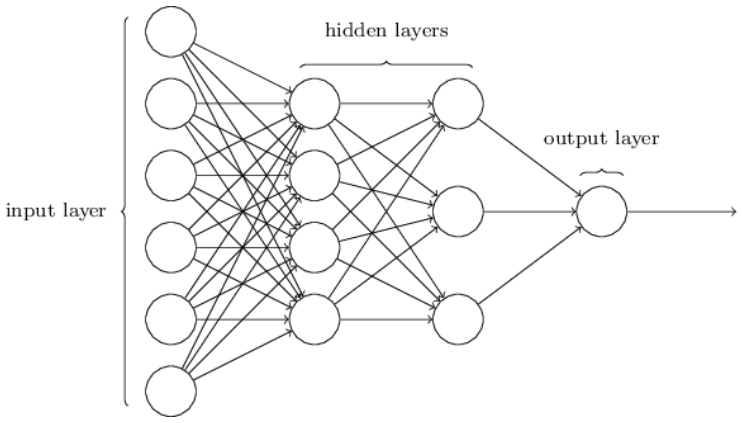


Fig. 1. An example of a simple FNN with two hidden layers. [23]

The real magic of this algorithm happens internally between the layers.

An example: Image recognition provides possibly the simplest explanation for the inner workings of a NN. Imagine taking a black and white image with a set number of pixels, each pixel can be viewed as one variable (or dimension) to be inputted into the NN, hence the number of input nodes will match the number of pixels. How

white the pixel is describes the activation of the respective input node, which we will denote with an a_n. The activation of a node will be relayed to the next node via an edge, every edge will have a **weight**, denoted as w_n, that is multiplied by the activation before it reaches the next node. When the signal reaches the next node, the next node will itself have an activation value formed from the sum of all edge values. Each node has another variable to lower or heighten this activation, this is called the *bias*, denoted as b_n. Finally each node will compress its activation so it exists as a value between 1 and 0, the most popular functions used to achieve this are the sigmoid and relu functions. The final formula for a node in the first hidden layer will look like this:

$$a_0^{(1)} = \sigma(w_{(0,0)}a_0^{(0)} + w_{(0,1)}a_1^{(0)} + ... + w_{(0,n)}a_n^{(0)} + b_0) \tag{1}$$

Where the top index for a represents the layer, the lower index (for both a and b) represents the subject node and the lower indices for the weights signify the receiver and the sender node respectively.

5.2.1 Gradient Descent. These values manipulate a given input as it travels down the NN until a particular output node is activated, generally each output node represents a specific class such as an animal or a number. Our hope is that given a specific input, the correct class will be the most activated output node. For our previous example, given an image of a number between 0 and 9, the most activated output should be the one we have designated as the expected number. These values are responsible for the internal makeup of the NN and contribute to a form of abstract pattern identification, thus the key to training a NN lies in how these values are manipulated.

Having first set random values for each weight and bias, the NN has to run an input, find the outputted activation (representing the predicted class) and make adjustments to these values, it does this using gradient descent. This process involves using a cost function to calculate how well our model does at predicting a classification. This can involve calculating the square difference between the outputted activation and the desired activation, reducing this value will mean our predicted label (or class) is closer to the actual label. The details behind finding the minimum of a cost function, and the specifics behind recursively performing gradient descent through previous layers (back propagation), are extensive and not required by the reader to know for this project.

5.3 Data Preparation

Before we can fully apply this reasoning to our data we need to prepare it for computational analysis, as mentioned in Ganegedara's book of *Natural Language Processing with TensorFlow*, the first step to do this is **Tokenization**. This involves encoding our sentences in a form more appropriate for mathematical based operations such as a number value that can be inputted into the input layer of nodes. One way to do this would be to transform every letter into a number but this presents an issue, the occurrence of anagrams means words may share the same set of numbers without sharing the same meaning. There is no intrinsic meaning to a single letter, hence analysing sentences at this level may produce muddled results, a more logical way would be to encode at the word level, where the majority of the semantics comes from. Consider the examples:

$$\text{I went to the station} \implies [001, 002, 003, 004, 005]$$
$$\text{I went to the shop} \implies [001, 002, 003, 004, 006]$$

When tokenizing a group of texts, for instance these two sentences, we assign each word to a given token, or value. We can see here that the two sentences share the same tokens, excluding the last one, possibly hinting at

their semantic similarities. When doing this process we create a log of what words we have encoded, but what if we need to encode a word in the future that we have not yet logged? Say we add an undocumented word "yesterday" on to the end of one of these sentences, what would happen? One very useful way to handle this is to use an out of vocabulary (<OOV>) token assigned to a value such as 007. The hope is that the dataset is large enough to cover a wide variety of words, particularly popular words, meaning we should not often see the <OOV> token. In the case for the word "yesterday" not being logged in the tokenizer, python will see the sentence "*I went to the shop yesterday*" as:

$$\text{I went to the shop <OOV>} \implies [001, 002, 003, 004, 006, 007]$$

In this case some of the major semantic meaning has been lost and this is an issue that contributes massively to the inaccuracy of a NLP model, unfortunately there is little to be done except to train the model on as large a dataset as possible. It is important to note that in order to be as reliable as possible the tokenizers lexicon has to be filled from the training sentences. Otherwise the model will have an unfair advantage when analysing words it should not have seen before, giving it a false boost in effectiveness.

There is another issue we have to tackle before we are able to input our encoded expressions into our NN. Sentences can obviously be different lengths, but basic NNs have a set number of input nodes, in order for the sentence (encoded into as a list) to be registered by the NN it needs to have the same number of elements as the input layer has nodes. In order to achieve this we effectively make every sentence the same length. We choose the length we want every sentence to be extended to and fill in the empty space using blank values, for instance 000. This process is called padding and is a vital part of transforming our sentences into appropriate data. We can choose to have the blank values placed before the encoded words or after, for this project we will set the padding to *post* (after the encoded words) for aesthetic reasons but either way will work as long as the padding is consistent. Say we set the max length of our sentences to 10, our phrase "*I went to the shop yesterday*" will now look like this:

$$\text{I went to the shop <OOV>} \implies [001, 002, 003, 004, 006, 007, 000, 000, 000, 000]$$

Once these steps have been fulfilled, we may begin to create and utilize a NN for text analysis.

6 CLASSIFYING MODELS

In this section we will look at the thought process and execution behind developing classifiers for both sarcastic and vague language. This paper will spare the reader the chore of reading through the entire code and instead we will focus on the key libraries, functions and the logic behind these classifiers.

6.1 Sarcasm Classifier

We will start with the more straightforward language tool, the majority of this work will be done in a single jupyter notebook using python with tensorflow version 2.3. For this program we will first need to import the important libraries:

Basic imports

```
import numpy as np                                                    basic operations
import pandas as pd                                                   dataframes
import os                                                             directory manipulation
```

Machine learning imports

```
import tensorflow as tf
from tensorflow import keras
from keras.models import Sequential
from keras.layers import Dense
from keras.preprocessing.text import Tokenizer
from keras.preprocessing.sequence import pad_sequences

import sklearn
from sklearn.model_selection import train_test_split
```

6.1.1 Data manipulation functions.
Before importing the data we will define functions `clean_text`, `split_data` and `make_sequences`. These functions will be called later on in order to tidy the data, decreasing abnormalities within the data and increasing accuracy, split the data into test and train data, and make tokenized padded sequences.

`clean_text`: In order to clean the text we will use the `re` library. The function works by taking some given text, de-capitalizing it, replacing contractions with the full word and removing punctuation, e.g.

$$\text{I'm sure I would've helped!} \implies \text{i am sure i would have helped}$$

This is achieved with the function:

`clean_text`

```
def clean_text(text):
    text = text.lower()
    text = re.sub(r"i'm", "i am", text)
    text = re.sub(r"\'s", " is", text)
    text = re.sub(r"\'ll", " will", text)
    text = re.sub(r"\'ve", " have", text)
    text = re.sub(r"\'re", " are", text)
    text = re.sub(r"\'d", " would", text)
    text = re.sub(r"n't", " not", text)
    text = re.sub(r"n'", "ng", text)
    text = re.sub(r"'bout", "about", text)
    text = re.sub(r"\?", "", text)
    text = re.sub(r"\!", "", text)
    text = re.sub(r"\.", "", text)
    text = re.sub(r"\,", "", text)
    text = re.sub(r"\r\n", "", text)
    return text
```

`split_data`: This function makes use of sklearn to split both the text and labels into four separate lists, then the function converts them into four arrays, the percentage split between test and train data needs to be specified.

split_data

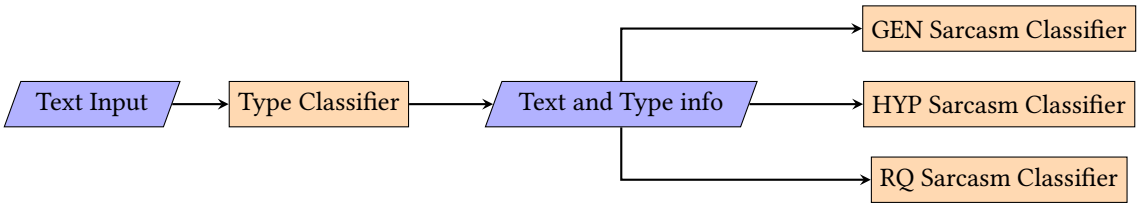
```
def split_data(text, labels, split_size):
    sentences_train, sentences_test, label_train, label_test = train_test_split(
        text, labels, test_size = split_size, random_state = 42)

    sentences_train = np.array(sentences_train)
    label_train = np.array(label_train)
    sentences_test = np.array(sentences_test)
    label_test = np.array(label_test)

    return sentences_train, sentences_test, label_train, label_test
```

`make_sequences`: Finally we have a function that takes sets of training and testing data, along with a respective pre-made tokenizer, and encodes and pads both sets of data. The data and tokenizer should be related, e.g. the test and train data for the general subset should have the general tokenizer.

make_sequences

```
max_length = 200

def make_sequences(tok, train_text, test_text):
    training_sequences = tok.texts_to_sequences(train_text)
    training_padded = pad_sequences(training_sequences, maxlen=max_length,
                                    padding = 'post', truncating = 'post')

    testing_sequences = tok.texts_to_sequences(test_text)
    testing_padded = pad_sequences(testing_sequences, maxlen=max_length,
                                   padding = 'post', truncating = 'post')

    return training_padded, testing_padded
```

As hinted at throughout this paper there was an immediate discovery found when designing a sarcasm classifier. The initial accuracy of a model working on the whole corpus was around 60% which was far from ideal, however, if we trained the model on purely general sentence type data, it was much more accurate, this increase was found when evaluating the other types as well. So it was decided that the best design for this model would be to have one NN that classified the type, then three specialized classifiers (one for each sentence type) to determine whether or not the text was sarcastic. The layout of this system will be as follows:



6.1.2 *TYPE Classifier:*

Using pandas we can import the three sarcasm corpus subsets into three distinct dataframes called `GENdata`, `HYPdata` and `RQdata`, representing the general, hyperbolic and rhetorical question subsets. We can then briefly join the datasets and update the ID numbers for the entries, giving us a big dataframe of the whole corpus. This allows for a nice organised dataframe from which we can quickly extract any sets we need as lists. We can then extract our sentences and labels, clean our text and finally transform the labels into values. This is done with a quick function that generates a new empty list and fills it with a 0, 1 or 2 when reading each element in the label list, 0 for general, 1 for hyperbole and 2 for rhetorical. We can then split the data using our `split_data` function, our resultant arrays are labelled as `Type_label_train`, `Type_label_test`, `nl_Type_train` and `nl_Type_test` (where `nl` refers to number labels). We may now create our tokenizer using the following code:

Type Classifier Tokenizer

```
Type_tok = Tokenizer(oov_token="<OOV>")                    generate tokenizer
Type_tok.fit_on_texts(Type_train)                          fill it from train data
Type_word_index = Type_tok.word_index                      define our lexicon
Type_vocab_size = len(Type_word_index) + 1
```

The last step to do before we can build the actual NN is the padding, this is easily done with the function we defined earlier, the resultant arrays are called `Type_train_padded` and `Type_test_padded`. A splitting ratio of 50/50 was used and a max padding length of 200 was used, thus the resultant shapes of these arrays are both $(4693, 200)$.

We may now finally construct the NN. Our first layer will be an embedding layer to transform all the words into vectors of a given dimension, after experimentation, 32 embedding dimensions were chosen. This is followed by a global average pooling layer which will add these vectors into something that can be that can be interpreted as the total semantic content of the inputted sentence. We then proceed with a more standard 24 node hidden layer using the relu activation function. Finally we have a 3 node output layer using the sigmoid activation function. Each node in the output layer represents a sentence type. The `sparse_categorical_crossentropy` loss function (another term for cost function) and the Adam optimizer also proved to be the most effective. The full classifier code is as follows:

Type Classifier NN code

```
Type_embedding_dim = 32

Type_model = tf.keras.Sequential([
       tf.keras.layers.Embedding(Type_vocab_size, Type_embedding_dim, input_length = max_length),
       tf.keras.layers.GlobalAveragePooling1D(),
       tf.keras.layers.Dense(24, activation = 'relu'),
       tf.keras.layers.Dense(3, activation = 'sigmoid'),
   ])

Type_model.compile(loss = 'sparse_categorical_crossentropy', optimizer = 'adam', metrics =
    ['accuracy'])

Type_model.summary()

Type_history = Type_model.fit(Type_train_padded, nl_Type_train, epochs = 30,
                           validation_data = (Type_test_padded, nl_Type_test), verbose = 2)
```

The final line here allows us to see the progress of our models training by testing our model on the test data, this is done over 30 iterations (or epochs). The final accuracy for this NN is 71% when analysing unseen test data. One way to improve this accuracy is by fine tuning the models layers, this can be a rather arbitrary process often achieved with repeated testing, something that can be quite time consuming with little payoff. 71% is still impressive, given the size of the dataset, and with an even bigger dataset, this accuracy could massively increase.

6.1.3 The Specialized Classifiers.

Now that we have hopefully found the type of sentence inputted we can get to classifying whether or not it is sarcastic. To do this we first have to repeat many of the steps in the last section, most notably cleaning, splitting and padding our individual subsets. The only major addition is a quick function to transform a list of strings containing “sarc” or “nonsarc” into an array of binary values, with 1 being sarcastic, these arrays are denoted with a “bl” prefix to mean binary from list. After we may again create specialized tokenizers filled with words from a respective training set and proceed to our model creation.

General Classifier Model

```
GEN_embedding_dim = 16

GEN_model = tf.keras.Sequential([
       tf.keras.layers.Embedding(GEN_vocab_size, GEN_embedding_dim,
                                 input_length = max_length, name = 'GENembed'),
       tf.keras.layers.GlobalAveragePooling1D(),
       tf.keras.layers.Dense(16, activation = 'sigmoid'),
       tf.keras.layers.Dense(24, activation = 'relu'),
       tf.keras.layers.Dense(32, activation = 'sigmoid'),
       tf.keras.layers.Dense(12, activation = 'relu'),
       tf.keras.layers.Dense(1, activation = 'sigmoid')
   ])

GEN_model.compile(loss = 'binary_crossentropy', optimizer = 'adam', metrics = ['accuracy'])

GEN_model.summary()
GEN_n_epochs = 18

GEN_history = GEN_model.fit(GEN_train_padded, blGEN_train, epochs = GEN_n_epochs,
                            validation_data = (GEN_test_padded, blGEN_test), verbose = 2)
```

The sarcasm classifier models have some key differences to our type classifier, the most important is the output layer. Instead of three categories we have a binary choice, 1 or 0, despite this, the most appropriate output layer will not have two nodes for these two classifications. As sarcasm and non-sarcasm are complete opposites, a classification of “sarcastic” would eliminate any chance for it to be any other classification. Thus it is more appropriate for our model to have one output node, simply classifying it as sarcastic when a high activation is present, if there is a low activation then it is likely to be not sarcastic. This change in output also allows for a different loss function, the binary cross-entropy function, this is specifically designed for such a model, and thus maximises our learning capabilities.

The general model is comprised of the same first two layers as our type classifier, with an embedding dimension on 16, then four hidden layers of varying node content, and finally a single sigmoid activated output node. As with all the models in this project, different sizes of layers and embedding dimensions were tested to maximise the validation accuracy (models accuracy on test data). Again there may be room for improvement in terms of the models design, but the potentially more beneficial area to improve is the data, this is especially important in later models.

Hyperbole Classifier Model

```
HYP_embedding_dim = 32

HYP_model = tf.keras.Sequential([
       tf.keras.layers.Embedding(HYP_vocab_size, HYP_embedding_dim,
                                 input_length = max_length, name = 'HYPembed'),
       tf.keras.layers.GlobalAveragePooling1D(),
       tf.keras.layers.Dense(16, kernel_regularizer=keras.regularizers.l2(0.001), activation =
           'relu'),
       tf.keras.layers.Dense(16, kernel_regularizer=keras.regularizers.l2(0.001), activation =
           'relu'),
       tf.keras.layers.Dense(1, activation = 'sigmoid')
   ])

HYP_model.compile(loss = 'binary_crossentropy', optimizer = 'adam', metrics = ['accuracy'])

HYP_model.summary()
HYP_n_epochs = 50

HYP_history = HYP_model.fit(HYP_train_padded, blHYP_train, epochs = HYP_n_epochs, validation_data =
    (HYP_test_padded, blHYP_test), verbose = 2)
```

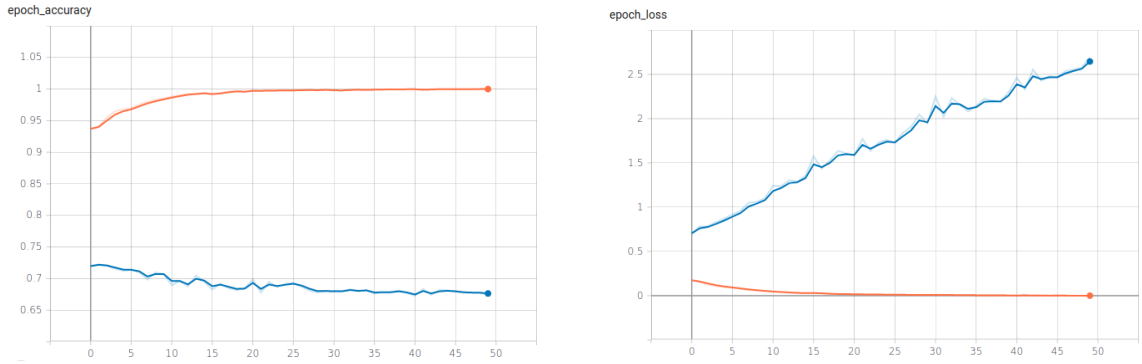
Rhetorical Classifier Model

```
RQ_embedding_dim = 3

RQ_model = tf.keras.Sequential([
       tf.keras.layers.Embedding(RQ_vocab_size, RQ_embedding_dim, input_length = max_length, name =
           'RQembed'),
       tf.keras.layers.GlobalAveragePooling1D(),
       tf.keras.layers.Dense(16, kernel_regularizer=keras.regularizers.l2(0.001), activation =
           'relu'),
       tf.keras.layers.Dense(16, kernel_regularizer=keras.regularizers.l2(0.001), activation =
           'relu'),
       tf.keras.layers.Dense(1, activation = 'sigmoid')
   ])

RQ_model.compile(loss = 'binary_crossentropy', optimizer = 'adam', metrics = ['accuracy'])

RQ_model.summary()
RQ_n_epochs = 50

RQ_history = RQ_model.fit(RQ_train_padded, blRQ_train, epochs = RQ_n_epochs, validation_data =
    (RQ_test_padded, blRQ_test), verbose = 2)
```

These models follow the same structure as the general classifier but with different hidden layers and embedding dimensions, all again designed to maximise accuracy. This time however, the effect of questionable dataset size becomes even more evident.

6.1.4 *Model Performance - Sarcasm.*

Overall these models managed to score a validation accuracy of 73.09% after 15 epochs, 64.10% for 20 epochs and 70.48% for 50 epochs for the GEN, HYP and RQ models respectively. The amount of epochs allowed for each model to train was the result of an observed over-fitting, most notably with the general classifier as seen here:



(a) GEN model accuracy over 50 epochs

(b) GEN model loss over 50 epochs

Fig. 2. Visualization of the over-fitting occurring in the GEN model in one particular training session (with orange for train data and blue for validation data)

As we can see in Fig 2a there is a clear decrease in validation accuracy over the course of 50 epochs, in some cases the use of dropout layers may reduce over-fitting but in this case no noticeable result was found. Later we will look into what specific details and results we can draw from this system with regards to our language model.

6.1.5 *The Kappa Module.*

Here we outline a brief code to tie all these models up into our desired final model. This complex of NNs has been named the Kappa module, this module will take a text input and output the sentence type prediction, the sarcasm prediction and the activation of the final node. First we use and format the type classifier output as part of a new function.

Type Classifier Function

```
def type_classifier(txt):
    text = clean_text(txt)
    input_sequence = Type_tok.texts_to_sequences([text])
    input_padded = pad_sequences(input_sequence, maxlen=max_length, padding = 'post', truncating =
        'post')

    if text == "": return

    prediction_list = Type_model.predict(input_padded)
    prediction_location = np.argmax(prediction_list)

    type_list = ['general', 'hyperbole', 'rhetorical']

    return(type_list[prediction_location])
```

This code simply returns the type of sentence given an input. This involves cleaning, tokenizing and padding the text then using the `Type_model` to predict the sentence type through indexing a list. Notice how it will return nothing in the event of an empty input. To follow this we use a similar function called the `sarcasm_classifier`, this again cleans, tokenizes and pads the text, if the input is not empty then it will then use the appropriate model to make a prediction. An example of how the right model is chosen and used is as follows:

The general classifier component of the sarcasm classifier

```
    elif(Type == 'general'):
        input_sequence = GENtok.texts_to_sequences([text])
        input_padded = pad_sequences(input_sequence, maxlen=max_length,
                                padding = 'post', truncating = 'post')

        predicted_probability = GEN_model.predict(input_padded)

        if(predicted_probability <= Gthreshold):
            prediction = "not sarcastic"
        elif(predicted_probability > Gthreshold):
            prediction = "sarcastic"
```

In an ideal world the model would give values close to 0 or 1 when outputting its prediction. However many inputs (from user testing) result in an output of around 0.5, this may highlight a level of uncertainty present in this model, but what is more immediately disruptive is the difficulty of drawing actual readable predictions. So our next step would be to find some way of reading this value as an actual prediction. Of course functions exist that do this, such as the sigmoid functions within our models, but we shall use another method that has the potential to humanize this Kappa module. In giving our model a threshold for classifying an activation value as sarcastic or not, that *we* can alter, we can change the sensitivity and potentially the "personality" of the module. This is done through the `Gthreshold` value seen in the previous component code. This ties in with our initial theme of advancements in chatbots as implementing this module into a chatbot would allow it to recognise and possibly produce sarcastic comments. Furthermore we may be able to change how sensitive it is to sarcasm, and possibly even change how sarcastic the chatbot is.

The final function `Kappa` calls these two functions to print all the information listed. While training the model can be computationally expensive, the models we have formed are able to give a prediction almost instantly, we are also able to export all of these models as a collection of weights to any other program we want (using the `pickle` library).

The final function utilizing all of our work

```
def Kappa(text):
    return sarcasm_classifier(text, type_classifier(text))
```

Listing 1. An example of an output from Kappa being effective

```
in:
Kappa("The weather is nice today!")

out:
INPUT: the weather is nice today
Type: general statement
prediction: not sarcastic
0.007712513 activation
```

Listing 2. An example of an output from Kappa being not so effective

```
in:
Kappa("Hello my name is Jason")

out:
INPUT: hello my name is jason
Type: general statement
prediction: sarcastic
0.99082863 activation
```

6.2 VL Classifier

This section will draw form much of the technical side of what we have done already for sarcasm, but now there are some new challenges for us to tackle due to some key differences. Firstly and more fundamentally VL is a very different language tool to sarcasm, sarcasm is also often more obvious to us than VL, possibly due to VL being more varied in language and application. There is a more immediate difference present (in terms of this project), the corpora. The corpus for VL that has been constructed for this project has many flaws, it has not been drawn from real world data, it is likely formed of phrases that unintentionally follow a template, and worst of all it is very small in size. The only saving grace of this corpus is that this potential template would have been based on the findings made when developing our language model. This gives us another opportunity to investigate the efficacy of our methods and allows us to hypothesize applications for a truly realized system.

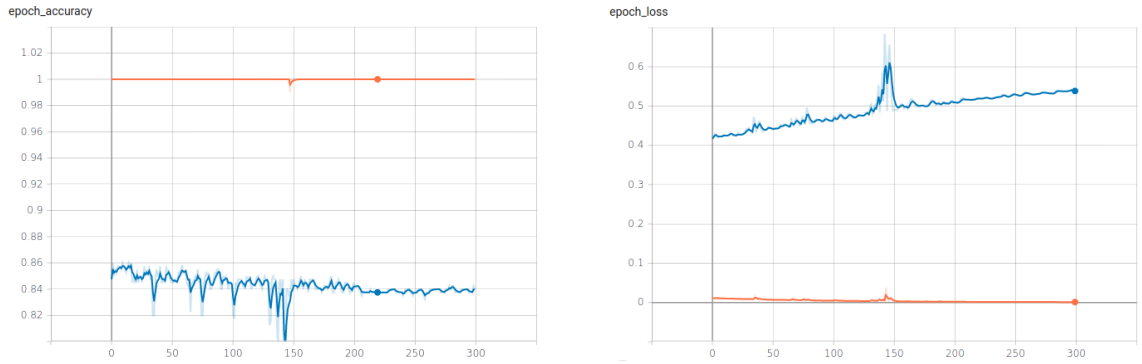
6.2.1 Type Classifier.

We will import the same libraries and functions used in the sarcasm document, and adapt them for a system with two types of VL, propositional and speech act.

Type Classifier for VL

```
type_embedding_dim = 128

type_model = tf.keras.Sequential([
       tf.keras.layers.Embedding(type_vocab_size, type_embedding_dim,
                                 input_length = max_length, name = 'TYPEembed'),
       tf.keras.layers.GlobalAveragePooling1D(),
       tf.keras.layers.Dense(512, activation = 'sigmoid'),
       tf.keras.layers.Dense(2, activation = 'sigmoid'),
   ])

type_model.compile(loss = 'sparse_categorical_crossentropy',
                   optimizer = 'adam', metrics = ['accuracy'])

type_model.summary()

type_history = type_model.fit(type_train_padded, nl_type_train, epochs = 300,
                              validation_data = (type_test_padded, nl_type_test),
                              verbose = 2)
```

This model shares many aspects with the sarcasm type classifier but due to a smaller dataset size we can experiment with larger layers. The most effective set up was found to be a system with one hidden 512 node sigmoid layer and an embedding dimension of 128. Using a small training data size of only 100 (20% of the corpus), this model achieved a validation accuracy of 84.50%. This did not come easily however, the model proved to be very inconsistent in its accuracy between epochs, but with such a small dataset we may increase the number of epochs until a more stable value is established. Increasing the epochs to 300 produced such results.



(a) VL type model accuracy

(b) VL type model loss

Fig. 3. Accuracy and loss changes over 300 epochs for the VL type classifier (with orange for train data and blue for validation data)

6.2.2 *Specialized Classifiers For VL.*

This pattern repeats to an extent for the other classifiers, validation values behave unpredictably as their average slowly increases over many epochs. Nevertheless we proceed forward with a similar tokenization system to that of the sarcasm specialized classifiers.

Propositional VL Classifier

```
prop_embedding_dim = 128

prop_model = tf.keras.Sequential([
       tf.keras.layers.Embedding(prop_vocab_size, prop_embedding_dim, input_length = max_length,
           name = 'Pembed'),
       tf.keras.layers.GlobalAveragePooling1D(),
       tf.keras.layers.Dense(256, activation = 'sigmoid'),
       tf.keras.layers.Dense(1, activation = 'sigmoid')
   ])

prop_model.compile(loss = 'binary_crossentropy', optimizer = 'adam', metrics = ['accuracy'])

prop_model.summary()
prop_n_epochs = 100

prop_history = prop_model.fit(prop_train_padded, prop_label_train, epochs = prop_n_epochs,
                          validation_data = (prop_test_padded, prop_label_test), verbose = 2)
```

Speech Act VL Classifier

```
speechact_embedding_dim =500

speechact_model = tf.keras.Sequential([
       tf.keras.layers.Embedding(speechact_vocab_size, speechact_embedding_dim, input_length =
           max_length, name = 'SAembed'),
       tf.keras.layers.GlobalAveragePooling1D(),
       tf.keras.layers.Dense(64, activation = 'sigmoid'),
       tf.keras.layers.Dense(1, activation = 'sigmoid')
   ])

speechact_model.compile(loss = 'binary_crossentropy', optimizer = 'adam', metrics = ['accuracy'])

speechact_model.summary()
speechact_n_epochs = 100

speechact_history = speechact_model.fit(speechact_train_padded, speechact_label_train,
                                    epochs = speechact_n_epochs,
                                    validation_data = (speechact_test_padded, speechact_label_test),
                                    verbose = 2)
```

These models reached validation accuracies of 80% and 90% over 100 epochs for propositional and speech act hedging respectively. While these appear to be good accuracies the model is only as good as the data. Again we will look deeper into this models effectiveness later in this project. Like with the sarcasm model, we can create a compact function, this time named the `Krieger` module, to access all models for VL in a similar fashion to Kappa.

Listing 3. An example of an output from Krieger being effective

```
in:
Krieger("there are around six of them in
    there")

out:
INPUT: there are around six of them in there
Type: propositional
prediction: vague
0.7513188 activation
```

Listing 4. An example of an output from Krieger being not so effective

```
in:
Krieger("Could you open this for me?")

out:
INPUT: could you open this for me
Type: propositional
prediction: not vague
0.17318392 activation
```

7 ANALYSIS

In this section we will take a deeper look into our models and attempt to extract words involved in both language tools.

7.1 Sarcasm Analysis

The function `Count_Correct` was constructed to generate a dataframe containing valuable information about each word in the dataset. This function requires an input of the model, test data (both padded and unpadded), test labels, word index and the threshold. With this information it produces predictions for each sentence in the data, constructs a dataframe and fills it with each word in the word index. It will then identify if the prediction was correct and then updates the dataframe with the total occurrences of that word, how many instances of a correct prediction and how many times it was both correct or incorrect in its prediction and the prediction was sarcastic.

Count_Correct

```
def Count_Correct(model, test_data, test_padded, test_label, word_index, thres):
    predicted_labels = model.predict(test_padded)
    words = list(word_index.keys())

    df = pd.DataFrame(columns = ['Words', 'Correct', 'Total', 'Rightly Sarcastic'])
    df['Words'] = words
    df['Correct'] = [0]*len(word_index)
    df['Total'] = [0]*len(word_index)
    df['Rightly Sarcastic'] = [0]*len(word_index)
    df['Incorrectly Sarcastic'] = [0]*len(word_index)

    for i in range(len(test_data)):
        boo = (binDecode(predicted_labels[i], thres) == binDecode(test_label[i], thres))
        element = test_data[i]
        for word in element.split():
            if (word in (df.Words.values)):
                word = word
            else: word = "<OOV>"
            df = update1(df, word, 'Total')
            if boo:
                df = update1(df, word, 'Correct')
                if (binDecode(predicted_labels[i], thres) == 1):
                    df = update1(df, word, 'Rightly Sarcastic')
            else:
                df = update1(df, word, 'Correct')
                if (binDecode(predicted_labels[i], thres) == 1):
                    df = update1(df, word, 'Incorrectly Sarcastic')
    return(df)
```

Internally this function uses two other functions, `binDecode` is designed to decide if the prediction is 1 or 0 given a threshold and `update1` is designed to update the dataframe by 1 in a given column for a given word. This function is not optimized, it loops over every sentence and every word in that sentence meaning it has a complexity of at least $O(n^2)$, consequently it is a slow process for larger datasets. Nevertheless we perform this function on the three specialized models and export the results. Now we can take each individual dataframe and

manipulate it to produce results. One such manipulation is the introduction of three new columns. The columns "TrueSarc%" and "FalseSarc%" are constructed by "Rightly Sarcastic" and "Incorrectly Sarcastic" values being divided by "Total" values respectively and the column "success%" is the result of "Correct" values divided by "Total" values. This will give us an insight into what percentage of the words are incorrectly identified and cause confusion for our models and we will be able to see what words are most commonly used in sarcastic phrases. In order to do this we need a quick function to reduce our dataframe to the top 15 words for "TrueSarc%" for instance.

reduceDF, produces the top words in a category

```
def reduceDF(df, total, topn, topcat):
    df = df[df['Total'] > total]
    df = df[~df.Words.isin(StopWords)]

    return df.nlargest(topn, topcat)
```

Here we can see two additional rules being applied, the total has to be above a certain number, reducing one-off words being seen, and the words can not be within the NLTK (another python library) stopwords corpus. This is a group of very frequently used short words which occur in nearly every sentence. Through being so common, they may not contribute massively to the sarcasm of a phrase, examples of words in the list include in, no, if, or, the, be, etc. The threshold for total occurrences is adjusted with respect to the dataset size for a given category as there are cases, like the hyperbole category, where only a few words are used more than 50 times. We therefore set the total threshold to 50, 15 and 50 for the GEN, HYP and RQ models, tables 1, 2 and 3 are the results of this dataframe reduction.

Looking at table 1 we can see the word with the highest percentage of sarcastic use is "oh" with a correct sarcastic prediction being made on 57.14% of sentences in the corpus. Furthermore it shows the lowest incorrect sarcastic prediction percentage (in this selection) at just 5.36%, as an extension of this, the model was correct 73.21% of the time, the second highest in this selection. To an extent, this matches our language model. "Oh" is an interjectional word used to express or emphasise spontaneous feelings, emotions or reactions, this gives it the capacity to reinforce illocutionary sarcasm. Camps insight of how illocutionary sarcasm can include implicatures that express evaluative attitudes such as pity, admiration, or surprise contribute to the idea that interjections can play a large role in some forms of sarcasm.

One of the easier targets to evaluate is the word "like", this being a very common word with multiple meanings, it is unlikely that the majority of its use is sarcastic. This is reflected in table 1 with it coming lowest in TrueSarc% with 35.34% and through it having a relatively high FalseSarc% of 18.97%. Despite this, there is something to be said about its occurrence in all three sarcasm result tables and with the non-general tables having lower FalseSarc% values, it may be our models reveal evidence of like-prefixed sarcasm. There is also evidence of propositional and lexical sarcasm through words like "really" (which is present in both tables 2 and 3) as it amplifies a words meaning to the extreme end of an associated scale.

There are some words appearing on these tables that 'stick out', most notably words like "gay" and "god", these words are likely the result of the origin of the training corpus. The original corpus used for the sarcasm classifiers included many discussions on topics such as religion and homosexuality, topics that likely had many discussions using such words and thus, many sarcastic sentences happen to have them in. Although the majority of words in these tables are more common and generic than the words "gay" and "god", the fact that such specific words can still slip through highlights a potential issue with this corpus. The fact that these discussions are so focused on particular topics means that the data will inherently have a small focus on these topics too, thus reducing the

generality of this corpus, and the subsequent models based off of it. Thankfully this issue has only produced two anomalies in the data, any other words relating to specific topics have either not been involved in enough sarcastic cases or have been so specific they are unknown and placed in the *out of vocabulary* section.

Unfortunately due to time constraints and the computational cost of running the `Count_Correct` function, we were only able to evaluate data on each single word, a more interesting evaluation would include analysis on sub-phrases as well as single words. This would allow us to more easily relate our findings to our model, but as it stands, the results mostly consist of relatively generic words that have a limited connection to our language model.

7.2 VL Analysis

Now to analyse our VL models, the method for table production mirrors that of the sarcasm analysis but this time, with such a small dataset, we have to reduce the threshold for occurrences and adopt a more cautious attitude towards the applicability of our results. This is not only due to the corpus being small, but to the corpus being constructed specifically for this project. Keeping this in mind, the reader should be reminded that the purpose of this section will be to analyse our models ability to detect patterns and key words that comply with our language model. In no way will any of our results suggest that these ML models are well equipped to identify real world examples of VL, the strength of these models instead depend on the strength of our language model.

One would be forgiven for immediately being sceptical of such perfect success rates in both tables 4 and 5 but keeping in mind all that was just mentioned, the words picked out as the top contributors to propositional VL massively relate to our language model. Words such as “things”, “kind” and “stuff” all directly relate to tag approximations and words like “around” and “many” are heavily present in number approximations. On top of this, the model has found these patterns/words with a high level of confidence. This is not incredibly surprising, propositional hedging has the simplest definition of the two hedging types in our model and it has a far simpler format, incorporating particular *’fuzzy’* words such as the ones seen in table 4.

Speech act hedging on the other hand is comparatively a mess, the dataset is so small that the overlap between documented words and test words is so small that we are only to extract 7 words of interest from the top 15 for TrueVague%. Of these seven words we can see some components of speech act hedging. “Need” and “cannot” are often used in sentences where their literal meaning is not used, e.g. “you cannot go out tonight”. The model was also able to pick up the importance of the word “sorry”, although not directly related to VL, apologies are a result of politeness, as is some VL, something potentially encoded in our classifying model.

8 CHATBOTS

Now that we have developed classifiers for aspects of natural language (of varying validity and effectiveness), we can now look at how they may be implemented into a chatbot. It was the original purpose of this project to build a chatbot from scratch (and using python libraries such as `NLTK`) and possibly even perform a participant based study on its effectiveness, however, building such a chatbot proved to be difficult to accomplish. This along with the difficulties finding ideal corpora for VL and the limited time available for this project meant a fully developed chatbot was not achievable. Nevertheless we can still look into how one could make a chatbot, the possible techniques we could use in the future and most importantly, where there may be room to implement our classifiers.

First of all it is important to remind ourselves that there are two main approaches to chatbots, the first one is the fixed response approach, where as many queries as possible are given a number of appropriate fixed responses. This approach does very little to mitigate the problem present in typical HCI today, namely the requirement

for specific inputs, having such fixed responses is as far away from a human conversation as it's possible to be. It is however, good for a select number of applications and is very easy to create, but nothing beats a human conversation in terms of text based interactions. The second approach is the more interesting one, and the one with more potential for development, a ML chatbot. A chatbot such as this, using tools like NNs, can give a greater range of responses based off of real world conversations and possibly allow for information retrieval for later use.

As mentioned, NNs are often used to achieve this but the NNs we have looked at so far only take an input of text, how is one supposed to generate text? A recurrent neural network (RNN) feeds a previous input into the processing of the following input, i.e. it takes the history of a sequence and uses it to analyse the next step in a sequence. Cho et al phrases it as being able to learn a probability distribution over a sequence by being trained to predict the next symbol in a sequence [9]. This can apply to a sequence of words, either in the form of a sentence or even an entire book, meaning we could make a RNN that, given a single word or sentence, will continually generate the next word until we had something like a book. The reader may want to imagine a chain of NNs acting on each word in a given sentence with each NNs producing a typical output and a second value, the hidden state, this hidden state is carried to the next NN *cell* and thus the analysis on the next word incorporates analysis on the previous word. This however only produces the next word in sequence, how would you adapt an NN of this type to give a full response to a given sentence instead? One common way is to use an encoder-decoder system, Cho et al describes this as a NN architecture that learns to *encode* a variable-length sequence into a fixed-length vector representation and to *decode* a given representation back into a variable-length sequence [9]. Within this system lies the opportunity for text classification, hopefully effecting the decoding process to produce variable semantic attenuation.

This system is not perfect, during back-propagation the adjustments to the weights get progressively smaller the further back in the model you go, meaning the adjustments made to the beginning cell, when analysing a long sentence, are negligible. This problem is known as the vanishing gradients problem, and is the cause for loss of semantic influence from words near the beginning of a sentence. To solve this a new NN was developed called the Long Short-Term Memory (LSTM) network.

The specifics of the following models are complex and again, beyond the scope of this paper. LSTMs use similar structures to standard RNNs but implement new internal functions to help carry sentiments from early words further up the chain. This is done through another hidden state, LSTMs produce another state value to be passed on to the next cell, this state can be updated with new information, allowing for the LSTM to vary its focus on old or new information. This is all done with internal functions called gates. Transformers take this the a new level with the idea of attention allocation, these very complex forms of encoder-decoder systems learn to focus on particular words with a large semantic contribution to the sentence.

8.1 Implementation

Finally let's look at where our classifiers may be able to take part in generating responses. Abdul-Kader et al outline chatbot design techniques in their 2015 paper [2], and what is particularly interesting is the key steps happening within a encoder-decoder system. They state that a chatbot may be divided into three parts: Responder, Classifier and Graphmaster. They go on to define each component:

- Responder: it is the part that plays the interfacing role between the bot's main routines and the user. The tasks of the responder are: transferring the data from the user to the Classifier and controlling the input and output.
- Classifier: it is the part between the Responder and the Graphmaster. This layer's functions are: filtering and normalising the input, segmenting the input entered by the user into logical components, transferring the normalised sentence into the Graphmaster, processing the output from the Graphmaster, and handling the instructions of the database syntax (e.g. AIML).
- Graphmaster: is the part for pattern matching that does the following tasks: organising the brain's contents, storage and holding the pattern matching algorithms.

We can see a familiar system here, a classifier, used to segment the input into logical components for the graphmaster to then be analysed for patterns. This is where an additional system could be placed to analyse the user input and produce additional information for the graphmaster to consider when pattern matching. In doing this, the chatbot could be more knowledgable about the use of sarcasm or VL and adjust the sentiment extraction process to account for any semantic attenuation. This is a two way process meaning the responses will also be tested for sarcasm and vagueness, this means that during training the chatbot should identify additional information about the response. It is then possible that the chatbot will be trained to sarcastically or vaguely reply to a sentence input that was similar to a sarcastic or vague interaction in the training data. For the most part this is speculation, a hypothesis that would set a foundation for a future project.

9 CONCLUSION AND FUTURE STUDIES

Overall this project saw some successes and some failures, the language model developed covered a lot of different areas of both language tools, and was supported by multiple reputable papers including the works of Channel and Fraser. The research leading up to the language model enhanced my personal understanding of both a common speech component, sarcasm, and a underappreciated speech component, VL, whose importance I was previously unaware of.

The sarcasm classifier was among the successes as it not only achieved a validation accuracy of 73.09% but it was also able to identify components of multiple sarcasm types within our language model. With a larger and more diverse dataset a more perceptive classifier might be possible, a classifier that is able to identify more sarcasm keywords. The VL classifiers on the other hand were riddled with issues in the forms of unstable learning (especially for VL type classifier) and poor keyword identification from the speech act hedging classifier. A key suspect for these errors is of course the self made VL corpus. It was always known that we could not take the results of the VL analysis literally, but there was hope that we could gain some validation for the NNs ability to detect patterns by having it detect keywords known, through our language model, to be important in VL. The truth, however, is that the corpus was too small to do this correctly, at least for the speech act model.

A recurring issue in this project was unideal datasets, while the sarcasm corpus did produce pleasing results there is always room to improve and of course, there is a lot of room for improvement when it comes to the VL corpus. There is still so much to do in this field and if this project were to be repeated then obtaining an actual VL corpus, created using real world data, would be the first thing to carry out. The scope of this project was nowhere near big enough to merit the use of mechanical Turks but should this investigation be repeated in a larger study,

Turks could be a viable option. Secondly there was room for deeper analysis of the models, such as analysis on word combinations and sub-phrases which would give us a better idea of sarcastic and vague word combinations. A disappointment within this project was not being able to build a working example of how the classifier models would work within a chatbot, again if this study was to be repeated, then a larger focus would be put on researching chatbot design. As an extension of this, the Cornell movie corpus and the metadata it contained help form another idea about chatbot design. What if we could provide metadata to a specially designed chatbot to give it a profile with an accompanying "personality", which would include age, gender, name, etc.? Would the isolation of particular words relating to a speakers profile be possible? And would a NN be able to recall such information about itself? All of these are questions for a bigger and deeper study that I would like to attempt in the near future.

Finally I am happy to report that although this project was conducted in a strange and stressful time for both myself and the world, regarding the pandemic, I had a lot of fun researching in this wonderful area of computer science and I hope to continue developing my interest in such a fertile subject.

REFERENCES

[1] [n.d.]. Global Intelligent Virtual Assistant Market 2018-2023:. https://www.businesswire.com/news/home/20180723005506/en/Global-Intelligent-Virtual-Assistant-Market-2018-2023-Market.

[2] Sameera A Abdul-Kader and JC Woods. 2015. Survey on chatbot design techniques in speech conversation systems. *International Journal of Advanced Computer Science and Applications* 6, 7 (2015).

[3] Silvio Amir, Byron C Wallace, Hao Lyu, and Paula Carvalho Mário J Silva. 2016. Modelling context with user embeddings for sarcasm detection in social media. *arXiv preprint arXiv:1607.00976* (2016).

[4] Regina Barzilay and Kathleen McKeown. 2001. Extracting paraphrases from a parallel corpus. In *Proceedings of the 39th annual meeting of the Association for Computational Linguistics*. 50–57.

[5] Yoshua Bengio, Réjean Ducharme, Pascal Vincent, and Christian Jauvin. 2003. A neural probabilistic language model. *Journal of machine learning research* 3, Feb (2003), 1137–1155.

[6] Elisabeth Camp. 2012. Sarcasm, pretense, and the semantics/pragmatics distinction. *Noûs* 46, 4 (2012), 587–634.

[7] Chih-Chung Chang and Chih-Jen Lin. 2011. LIBSVM: A library for support vector machines. *ACM transactions on intelligent systems and technology (TIST)* 2, 3 (2011), 1–27.

[8] Joanna Mary Channell. 1983. *Vague language: Some vague expressions in English.* Ph.D. Dissertation. University of York.

[9] Kyunghyun Cho, Bart Van Merriënboer, Caglar Gulcehre, Dzmitry Bahdanau, Fethi Bougares, Holger Schwenk, and Yoshua Bengio. 2014. Learning phrase representations using RNN encoder-decoder for statistical machine translation. *arXiv preprint arXiv:1406.1078* (2014).

[10] Joan Cutting. 2007. *Vague language explored.* Springer.

[11] Cristian Danescu-Niculescu-Mizil and Lillian Lee. 2011. Chameleons in imagined conversations: A new approach to understanding coordination of linguistic style in dialogs.. In *Proceedings of the Workshop on Cognitive Modeling and Computational Linguistics, ACL 2011*.

[12] Bhatia et al. 2020. Twitter Natural Language Processing - Noah's ARK. http://www.cs.cmu.edu/~ark/TweetNLP/

[13] Elena Filatova. 2012. Irony and Sarcasm: Corpus Generation and Analysis Using Crowdsourcing.. In *Lrec*. Citeseer, 392–398.

[14] Bruce Fraser. 2010. Pragmatic competence: The case of hedging. *New approaches to hedging* 1534 (2010).

[15] Thushan Ganegedara. 2018. *Natural Language Processing with TensorFlow: Teach language to machines using Python's deep learning library.* Packt Publishing Ltd.

[16] Debanjan Ghosh, Weiwei Guo, and Smaranda Muresan. 2015. Sarcastic or not: Word embeddings to predict the literal or sarcastic meaning of words. In *proceedings of the 2015 conference on empirical methods in natural language processing*. 1003–1012.

[17] Roberto González-Ibánez, Smaranda Muresan, and Nina Wacholder. 2011. Identifying sarcasm in Twitter: a closer look. In *Proceedings of the 49th Annual Meeting of the Association for Computational Linguistics: Human Language Technologies*. 581–586.

[18] Herbert P Grice. 1975. Logic and conversation. In *Speech acts*. Brill, 41–58.

[19] Aditya Joshi, Vaibhav Tripathi, Kevin Patel, Pushpak Bhattacharyya, and Mark Carman. 2016. Are word embedding-based features useful for sarcasm detection? *arXiv preprint arXiv:1610.00883* (2016).

[20] Philipp Koehn, Hieu Hoang, Alexandra Birch, Chris Callison-Burch, Marcello Federico, Nicola Bertoldi, Brooke Cowan, Wade Shen, Christine Moran, Richard Zens, et al. 2007. Moses: Open source toolkit for statistical machine translation. In *Proceedings of the 45th annual meeting of the ACL on interactive poster and demonstration sessions*. Association for Computational Linguistics, 177–180.

[21] Sachi Kumon-Nakamura, Sam Glucksberg, and Mary Brown. 1995. How about another piece of pie: The allusional pretense theory of discourse irony. *Journal of Experimental Psychology: General* 124, 1 (1995), 3.

[22] S Lukin and M Walker. 2013. Really? Well. Apparently bootstrapping improves the performance of. In *Proc. Workshop on Language in Social Media*.

[23] Michael A. Nielsen. 2015. Neural Networks and Deep Learning. http://neuralnetworksanddeeplearning.com/chap1.html

[24] John E Opfer and Robert Siegler. 2012. Development of quantitative thinking. (2012).

[25] Shereen Oraby, Vrindavan Harrison, Lena Reed, Ernesto Hernandez, Ellen Riloff, and Marilyn Walker. 2017. Creating and characterizing a diverse corpus of sarcasm in dialogue. *arXiv preprint arXiv:1709.05404* (2017).

[26] Marilyn A Walker, Jean E Fox Tree, Pranav Anand, Rob Abbott, and Joseph King. 2012. A Corpus for Research on Deliberation and Debate.. In *LREC*, Vol. 12. Istanbul, 812–817.

[27] Meishan Zhang, Yue Zhang, and Guohong Fu. 2016. Tweet sarcasm detection using deep neural network. In *Proceedings of COLING 2016, The 26th International Conference on Computational Linguistics: Technical Papers*. 2449–2460.

Words	Correct	Total	Rightly Sarcastic	Incorrectly Sarcastic	FalseSarc%	TrueSarc%	success%
oh	41	56	32	3	5.36	57.14	73.21
guess	36	52	27	9	17.31	51.92	69.23
read	59	83	41	11	13.25	49.4	71.08
go	68	97	44	18	18.56	45.36	70.1
maybe	49	76	34	14	18.42	44.74	64.47
post	45	71	31	13	18.31	43.66	63.38
gay	58	84	36	18	21.43	42.86	69.05
good	66	96	39	18	18.75	40.63	68.75
old	48	73	28	10	13.7	38.36	65.75
understand	43	58	22	8	13.79	37.93	74.14
let	29	53	20	16	30.19	37.74	54.72
get	107	170	63	38	22.35	37.06	62.94
yet	45	65	24	8	12.31	36.92	69.23
must	55	77	28	12	15.58	36.36	71.43
like	156	232	82	44	18.97	35.34	67.24

Table 1. General classifier analysis table (at least 50 occurrences)

Words	Correct	Total	Rightly Sarcastic	Incorrectly Sarcastic	FalseSarc%	TrueSarc%	success%
know	15	25	10	3	12	40	60
people	15	23	8	2	8.7	34.78	65.22
really	13	16	5	2	12.5	31.25	81.25
oh	16	23	7	1	4.35	30.43	69.57
get	14	18	5	2	11.11	27.78	77.78
love	8	16	4	2	12.5	25	50
like	16	21	5	1	4.76	23.81	76.19
<OOV>	456	647	128	83	12.83	19.78	70.48
one	21	29	5	1	3.45	17.24	72.41
surprised	19	28	4	3	10.71	14.29	67.86
would	28	37	5	5	13.51	13.51	75.68
think	11	21	1	0	0	4.76	52.38

Table 2. Hyperbole classifier analysis table (at least 15 occurrences)

Words	Correct	Total	Rightly Sarcastic	Incorrectly Sarcastic	FalseSarc%	TrueSarc%	success%
go	40	58	20	4	6.9	34.48	68.97
get	53	83	28	12	14.46	33.73	63.86
way	45	55	17	5	9.09	30.91	81.82
mean	37	56	16	1	1.79	28.57	66.07
want	48	67	19	6	8.96	28.36	71.64
like	75	114	32	10	8.77	28.07	65.79
one	103	142	38	10	7.04	26.76	72.54
right	51	79	21	9	11.39	26.58	64.56
<OOV>	2891	4347	1146	402	9.25	26.36	66.51
god	60	92	24	10	10.87	26.09	65.22
really	51	77	19	7	9.09	24.68	66.23
even	51	76	18	9	11.84	23.68	67.11
could	47	69	16	5	7.25	23.19	68.12
see	46	70	15	11	15.71	21.43	65.71
much	40	60	12	9	15	20	66.67

Table 3. Rhetorical classifier analysis table (at least 50 occurrences)

Words	Correct	Total	Rightly Vague	Incorrectly Vague	FalseVague%	TrueVague%	success%
something	4	4	4	0	0	100	100
things	7	7	7	0	0	100	100
many	4	4	4	0	0	100	100
wonder	3	3	3	0	0	100	100
around	5	5	5	0	0	100	100
dozen	3	3	3	0	0	100	100
think	14	14	13	0	0	92.86	100
stuff	4	5	4	0	0	80	80
like	28	32	25	2	6.25	78.13	87.5
kind	3	4	3	1	25	75	75
well	5	7	5	2	28.57	71.43	71.43
yeah	5	7	5	0	0	71.43	71.43
still	5	7	5	1	14.29	71.43	71.43
get	6	7	5	0	0	71.43	85.71
sure	2	3	2	0	0	66.67	66.67

Table 4. Propositional classifier analysis table (at least 2 occurences)

Words	Correct	Total	Rightly Vague	Incorrectly Vague	FalseVague%	TrueVague%	success%
say	2	2	2	0	0	100	100
get	1	1	1	0	0	100	100
need	1	1	1	0	0	100	100
tell	1	1	1	0	0	100	100
sorry	1	1	1	0	0	100	100
got	1	1	1	0	0	100	100
<OOV>	13	14	9	0	0	64.29	92.86
cannot	1	2	1	0	0	50	50
go	1	1	0	0	0	0	100
tonight	1	1	0	0	0	0	100
going	1	1	0	0	0	0	100
tea	1	1	0	0	0	0	100
making	1	1	0	0	0	0	100
grass	0	1	0	0	0	0	0

Table 5. Speech act classifier analysis table (at least 1 occurence)