a title

Sean Bugeja

Supervisor: Mr. Mike Rosner

Faculty of ICT

University of Malta

enter a date

Submitted in partial fulfillment of the requirements for the degree of Master of Science in Computer Science

Faculty of ICT

Declaration

I, the undersigned, declare that the dissertation entitled:

a title

submitted is my work, except where acknowledged and referenced.

Sean Bugeja

enter a date

Acknowledgements

your acknowledgments

Abstract

Provides a short (typically 1 page) overview of the dissertation's contents including the tackled problem and high-level results/conclusions.

Contents

List of Figures

List of Tables

Acronyms

NLP Natural Language Processing

LSTM Long Short Term Memory

OOV Out of Vocabulary

IV In Vocabulary

SVM Support Vector Machine

ASR Automatic Speech Recognition

RC Reading Comprehension

LDA Latent Dirichlet Allocation

LSA Latent Semantic Analysis

CLI Command Line Interface

1. Introduction

1.1 Problem Definition

A definition of the problem being tackled and establishment of the research question(s).

2. Background and Literature Review

2.1 Targeted Sentiment Analysis

2.1.1 What is targeted sentiment analysis?

Targeted sentiment analysis is a fine-grained text-classification task which stems from the broader, more general, document, or sentence, level sentiment analysis. The former extends on the latter by taking into consideration a particular target or aspect within the context of the document, and aims to identify the sentiment with respect to this target or aspect [**?**], [**?**], [**?**].

It is often the case in the literature that when referring to a *target*, this would be a particular noun or subject within the phrase while an *aspect* can be a more general area or concept that the phrase touches on, without referencing in the literal sense. Consider a sentence such as, "The waiting times were long however the ravioli were simply to die for", a plausible *target* that could be considered is "waiting times" for which the statement conveys a negative sentiment. Alternatively, the phrase could be assessed with respect to an *aspect* such as "food quality", for which a positive sentiment is conveyed even though the precise term "food quality" is only implicitly implied.

It is evident that, separating itself from sentence-oriented sentiment analysis, target or aspect based sentiment analysis requires the careful consideration of the target or aspect in question along with its context. The extent of this fact was initially demonstrated by [**?**], whose work demonstrated that a staggering 40% of errors within the field of targeted-sentiment analysis could be attributed to the lack of consideration of the target or aspect [**?**].

2.1.2 What is the importance of targeted sentiment analysis?

Due to the proliferation of social media networks and online shopping, opinions voiced from users on specific topics, products, services and events have never been as readily available for data mining. The value in having the means to accurately gauge public interest and opinion of very specific topics of interest on such a phenomenal scale cannot be understated. From those in the public sector, such as electoral campaigns who seek to obtain a clearer picture of their constituents' strongest held opinions and expectations, to private businesses who wish to employ the most effective advertising campaign for their products and services, all of these objectives rely heavily on being as cognizant on public sentiment as possible. [**?**]

Over time the content of these online text sources has become more sophisticated and richer in information. Changes in social media platforms such as *twitter*'s decision to raise the character limit of tweets results in the same unit of data conveying up to twice the amount of information. As this availability increases, so to must the resolution at which this information is processed, so as to keep pace with the needs of both producers and consumers alike. This phenomenon further pushes the need to focus on opinion mining at a finer-grained level, perfecting the ability to discern varying sentiments towards separate targets within the same phrase.

2.1.3 What are the challenges of targeted sentiment analysis?

As with any task that requires a deeper understanding of the intricacies of language, there are many challenges that face target-oriented sentiment analysis. Many of these challenges are inherent to parsing the structure of a language such as sarcasm, where sentences such as "Nice perfume. You must shower in it." [**?**] are composed entirely of positive-sentiment bearing words while expressing a negative sentiment. These notwithstanding, the informal nature of the majority of text data found on social media platforms (upon which this task frequently focuses), supplements these challenges with its own.

Colloquialisms and social short-hands are a commonplace within social media networks where many users intend on conveying as much information in as little characters as possible, particularly in situations where this number is capped. This phenomenon also leads to intentional, as well as unintentional, spelling errors which further obscures that data for any prospective machine learning model that does not account for these circumstances.

Along with these challenges, the literature also presents a number of obstacles and particularly problematic instances that need to be taken into account when approaching the task of targeted sentiment analysis. Comparative opinions are one such circumstance where the sentiment being conveyed is obscured by another subject. [**?**] report challenges of this sort, with phrases such as "I've had better Japanese food at a mall food court".

Other common challenges that are pointed out in [**?**], are negation and conditional situations, citing the example "but dinner here is never disappointing, even if the prices are a bit over the top", where the sentiment towards the target cannot be easily deduced from the various syntactic structures present.

Moreover, the particular case of expressions which consist of multiple words needs to be given special care. Various approaches that employ word embeddings

operate on the word as the atomic unit of operation, and would therefore struggle to correctly model an expression such as "die for" in "the ice cream to die for" [?] from its constituents. [?] also also stress the significance of this issue, and argue that it has not been given sufficient attention, particularly when modelling *targets* that also consist of multiple words.

When considering the opinion of a sentence towards a specific target, it may be the case that the sentence will have opposing sentiments for different targets, this is another degree of complexity that targeted-sentiment analysis models need to account for as opposed to sentiment analysis of the sentence as a whole [?]. Phrases such as "great food but the service was dreadful!" convey different and opposite sentiments towards "food" and "service" [?]. Previous sentence oriented sentiment analysis approaches such as [?], [?] would be incapable of correctly distinguishing this level of granularity [?].

[?], [?] also call attention to the fact that there are several instances where the sentiment conveyed by a particular word is contingent upon the target or aspect that is being considered. An adjective such as "short" can have positive connotations with respect to "waiting times" for a restaurant, on the other hand the same adjective is assumed negative when describing something such as the "battery life" of a product.

2.1.4 What metrics are commonly used to measure performance?

	$y = A$	$y \neq A$
$\hat{y} = A$	true positive (tp)	false positive (fp)
$\hat{y} \neq A$	false negative (fn)	true negative (tn)

Table 2.1: Confusion Matrix for the binary case of some class label, A. y represents the true label while $\hat{y}$ represents the predicted label.

Two commonly extrapolated metrics from which other measures are typically derived are precision and recall. Given some class A, the former is a ratio of

correctly labelled instances to all instances labelled A whereas the latter compares the amount of correctly labelled instances to all instances of class A present in the data, which is analogous to the accuracy for class A. Formally, based on the definitions in table **??**, the two measures are given by:

$$precision_A = \frac{tp_A}{tp_A + fp_A} \tag{2.1}$$

$$recall_A = \frac{tp_A}{tp_A + fn_A} \tag{2.2}$$

For the case with C classes, the total number of instances for a class c, N_c is equal to $tp_c + fn_c$. The prevalent metric for accuracy that is reported in the literature, equivalent to the micro-averaged recall, is thus computed by:

$$accuracy = \frac{\sum_c^C tp_c}{N} = recall^{micro} \tag{2.3}$$

Where N is the total number of instances in the data. Using this micro-averaged metric, however, is not necessarily the most accurate indicator of a model's performance in a classification task, particularly when the dataset that is being utilized is heavily biased to one specific class. Care must be given in the training phase of any machine learning model to ensure that the model is exposed to all classes in question in a balanced way. This is because in computing the micro-average, the weighting scheme is distributed across all instances in the dataset, as opposed to the classes. A more sophisticated metric that is robust to this issue is the macro-averaged F1-score which equally distributes the weight across all classes as opposed to instances [**?**]. The macro-averaged F-measure is given by the harmonic mean of the, macro-averaged, precision and recall for a specific class. For some class A, this is given by:

$$F1_A^{macro} = \frac{2P_A^{macro}R_A^{macro}}{P_A^{macro} + R_A^{macro}} \tag{2.4}$$

Training a model heavily on one specific class, or not enough on another could

lead the model to classify the majority of test samples to the biased class or being unable to correctly classify the class that has been under-represented in training, since the model would not have gathered enough information to discern this class. In the case where the testing dataset would be imbalanced towards the same class, the overall accuracy would lack the sufficient information expected as a metric to illustrate the effectiveness of the model to classify samples into the correct class since the model would have been trained in a biased way towards the class that is prevalent.

As an example, a frequently cited benchmark dataset is presented in [**?**], this dataset consists of 6248 training and 692 test phrases collected from twitter, each annotated with a particular sentiment (negative, neutral or positive) towards a specific target that appears in the tweet. Within both the training and testing subsets of this dataset, there are twice as many neutral instances as there are positive and negative instances. Works such as [**?**], [**?**] correctly point out the shortcoming of accuracy as a valid performance metric in this situations such as this, and cite macro-averaged F1 scores in their results.

2.2 Manual Feature Engineering

2.2.1 What did initial approaches using manual features involve?

Initially, the conventional approach involved manually extracting the most expressive and information rich features from sentences that would subsequently be processed through some statistical model such as a Support Vector Machine (SVM) for classification.

This entailed the formulation of processes by which to obtain these features, and was normally preceded by some form of normalization of the original data before these features could be extracted. Typically many types of these features were

used in conjunction, each intended to extrapolate differing particularities about a specific aspect of the text, such as whether a specific token represented a noun or an adjective, or details about the words surrounding it to name a few.

2.2.2 What were some of the initial approaches using manual features?

The capacity of the SVM had been demonstrated on the general task of sentiment analysis in works such as [**?**], as well as other tools such as, bag-of-words, part-of-speech tags and other morphological and syntactical features and external resources such as linguistic parsers and sentiment lexicon, employed in works such as [**?**], [**?**], [**?**].

However, as [**?**] point out, these methods would implicitly impose an external dependency on the system. Moreover, within the context of social media, where conventional rules of language are often times regarded rather as guidelines, various studies question the applicability of dependency parsing techniques that rely on a particular degree of formality, or structure, within the content itself [**?**], [**?**]. Nevertheless these features have proven their worth when used in conjunction with powerful models such as the aforementioned SVM [**?**] [**?**], as well as neural networks [**?**], [**?**], in predicting sentiment polarity.

Even in the work that followed, focusing on increasingly autonomous feature extraction methods and more sophisticated deep learning architectures such as the Long Short Term Memory (LSTM) model, [**?**] make note of the competitive results obtained by the SVM approach in [**?**] when compared to their implementations.

2.2.3 What are the disadvantages of using manual features?

Although works such as [**?**], [**?**]) obtained encouraging results, much of the subsequent literature recognizes that these results where exceedingly contingent on the choice of features that were being utilized [**?**].

Although the manual feature-based approach fared well in their work, [?] suggest that features of this kind lack the required resolution of detail that would accurately capture the interplay between target and context. The features that had been used had sound rationales behind them, however devising these rationales was in itself becoming increasingly time consuming. One reason for this is scalability; with the increase of data that were available, this inevitably brings with it more considerations and specifics that must be accounted for when otherwise manually devising these feature.

As alluded to by [?], with the aforementioned increase in labor involved, these approaches were exhibiting diminishing returns, and could be regarded as a bottleneck in terms of performance of these models and the wealth of data available. A more autonomous solution that would accurately capture the intricacies of language from an expansive wealth of text at a deeper level, not contingent on a proportionally large amount of labor-intensive manual feature-engineering, was desired to further advance the field of targeted sentiment analysis.

2.3 Word Embeddings

2.3.1 What are word embeddings?

Word embeddings seek to tackle the non-trivial task of accurately capturing as much of the intricate details that are inherent in language, as possible. To model the intricacies of a word within the context of a language, or a particular subset thereof, sophisticated models are employed to construct continuous and real-valued vectors for each word. The resulting vectors are commonly referred to as word embeddings, and are meant to be numerical representations of contexts in which each word is used [?] [?] [?] [?].

By first learning a continuous word vector embedding from data [?] [?] [?], most approaches can take advantage of the principle of *compositionality* [?] to obtain a sentence or indeed a document level representation for a myriad of downstream

NLP tasks, including sentiment analysis.

Two of the most prominent word embedding models that are currently employed in many NLP tasks are *word2vec* [**?**] and *GloVe* [**?**]. More recently, an extension on the former, termed *fasttext* [**?**] is also garnering considerable attention within the field, distinguishing itself from the other models mentioned through the use of sub-word information.

2.3.2 What are the leading two approaches?

The two principal methods for learning distributional word representations are count-based and prediction-based, each with their own strengths and shortcomings, and neither being clearly superior to the other in every aspect. The former typically involves performing dimentionality reduction on a co-occurrence count matrix, whereas the latter is derives word representations from learning to correctly predict words or contexts within a bounded, moving, window. [**?**]

Both methods provide a powerful means of autonomously extracting expressive and meaningful features from a wealth of text to the degree that would be unfeasible through manual means alone due to the staggering complexity inherent in language itself as well as the sheer volume of data that is being constantly made available through various online platforms that allow the power of expression to an unparalleled, and ever-growing, number of people across a myriad of different domains and topics.

One of the benefits that is had from constructing a vector space with distributional representations of words is the ability to model features such as similarity between the constituent words and subsequently group words with similar meanings which expands a downstream model's comprehension of a language. [**?**] Matrix factorization methods, such as Latent Semantic Analysis (LSA) [**?**] as cited in [**?**], attempt to extrapolate significant statistical information pertaining to a specific corpus, from decomposed matrices that are typically obtained through low-rank approximations. Although techniques such as LSA do this effectively, [**?**] note a

lackluster performance in word-analogy tasks, which ultimately depend, in large part, on accurately capturing the aforementioned similarity between words in the vector space.

Models trained using matrix factorization methods are efficient at exploiting statistical information, and are typically easier to parallelize albeit at a higher initial memory investment involved in storing the co-occurrence matrix [**?**]. [**?**] work on the assumption that more related words will tend to appear closer to each other than not, making window-based approaches ideal for word analogy tasks that benefit from accurate word-similarity modelling.

Since predictive methods rely on a bounded context window of some specified width when making predictions a trivial disadvantage that presents itself immediately with this approach is the limited capacity to efficiently exploit redundancy in the data on at a macro-scale. This is due to the fact that these methods operate on the level of their current context window as opposed to the document as a whole [**?**].

2.3.3 How does word2vec do its thing?

In their work, [**?**] note that the tendency in the field of NLP was to regard words in an isolated fashion as opposed to considering each word within the scope of a distributed space where more in-depth information could be encoded regarding the relationship between words. Models, such as the popular n-gram model where prevalent in the literature due to their simplicity and effectiveness. [**?**] note that these methods had largely peaked as unlike distributed representations, they were limited in their capacity to model relationships, such as similarity, between words.

There was a need for more sophisticated, and scalable, techniques to address the bottleneck that was presenting itself in the lack of accurately labelled data that was being made available for training models in fields such as Automatic Speech Recognition (ASR) and machine translation. As these models emerged to tackle increasingly large data sets, simpler models, such as the aforementioned n-gram

model, were consistently outperformed by neural networks using distributed representations of words in the form of word embeddings. Moreover, neural networks were shown to capture the linear relationships between words more accurately than previous methods that used LSA, while maintaining a higher level of scalability when compared to LDA [**?**].

Some of the initial work making use of neural networks to learn word embeddings include [**?**], later refined in [**?**]. Their approach consisted of a feedforward neural network tasked with predicting the correct word within a context window of five words including the target word itself.

[**?**] introduced the Continuous Skip-gram and the Continuous Bag-Of-Words (CBOW) models in an effort to extract word embeddings from large corpora. As illustrated in figure **??**, the former aims to learn an adequate representation of a word by training to predict the words that are most likely to surround it while, inversely, the latter was trained to predict a specific word given its context, both using a feed-forward neural network with the non-linear hidden layer removed.

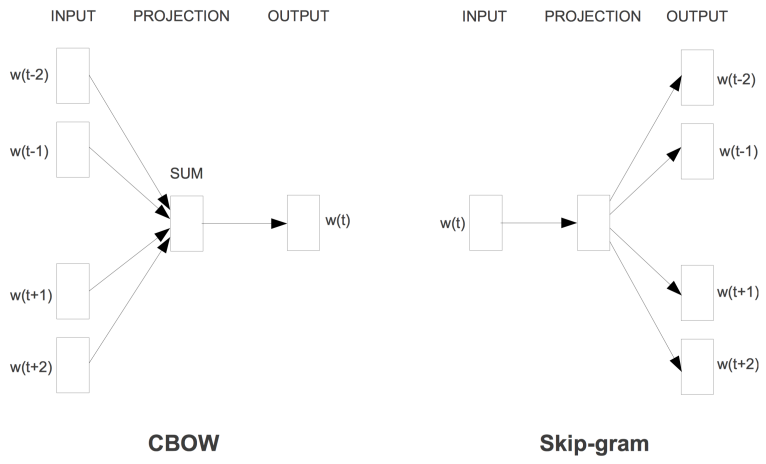

Figure 2.1: The differences between the model architectures proposed by Mikolov in [**?**]. The Continuous Bag-Of-Words (CBOW) approach predicts a word from its context and, conversely, the skip-gram model predicts the context from a word. [**?**]

Their work brought about a novel evaluation strategy which measures the expressive capacity of a word vector space through complex multi-dimensional com-

parisons that went over and above previous scalar metrics that were typically limited to the distance or angle between word vectors. As evidence of the level of sophistication obtained in the representations that were generated using the *skip-gram* model, the authors note that the composition of vectors for words such as "Germany" and "capital" results in a vector that closely resembles the word "Berlin". Additionally, more sophisticated linear translations could be modeled such as "Madrid" - "Spain" + "France" is resulting in a word vector closest to that of the word "Paris" [**?**].

While the skip-gram model required no dense matrix multiplications which made it substantially more efficient than most of the neural network implementations that had preceded it, in their experiments they note that the quality of the resultant word vectors could be improved by increasing the window size, however this would carry with it a corresponding increase in computational complexity. Nevertheless, [**?**] demonstrated that sufficiently expressive vector representations of words could be obtained from large corpora of data without the need for computationally expensive models.

Following their initial work, [**?**] later expanded on their *word2vec* models, introducing the negative sampling algorithm to improve the efficiency by which the model learned word vectors while proposing a method for accounting for phrases through a simple data-driven approach whereby particular phrases composed of multiple words were treated as a singular token, and trained-for as such.

As noted by [**?**], the performance of their models were contingent on a set of design choices, most critical of which include the training algorithm, vector size, sub-sampling rate and the size of the training window. They conclude that the optimal configuration for these parameters varies based on the task being tackled.

2.3.4 How is GloVe different to Word2Vec?

Using co-occurrence statistics to extract continuous representations of words within a large corpus of data has been explored in NLP in works as early as [**?**], as cited

by [**?**]

[**?**] argue that while the significance of word occurrence information within a text when learning word representations in an unsupervised manner is uncontested, further research is needed into the mechanisms by which these statistical data generate meaningful vector representations. In pursuit of this, they propose the *GloVe* model, characterized by its use of the global, that is to say at the level of the corpus in its entirety, co-occurrence information to produce aptly-called *"global vectors"*.

[**?**] point out that while count and prediction based methods are not fundamentally dissimilar, since both exploit co-occurrence statistics within a corpus to obtain accurate representations, they argue for the efficiency of the former approach over the latter.

A word-word co-occurrence matrix X is constructed from a vocabulary such that X_{ij} is representative of the number of times word j is found in the context of word i. This invariably makes X sparse in nature, since the substantial portion of words within a language cannot be expected to occur within an equally substantial number of words.

[**?**] develop *GloVe* by only considering non-zero elements of the co-occurrence matrix of the corpus as a whole and as opposed to the sparse matrix in its entirety. This provides a substantial increase in speed and moreover, as their work suggests, generates more expressive representations of each word when compared to a limited window-based approach. *GloVe* was evaluated on three separate tasks, specifically word analogy, word similarity and entity recognition tasks, achieving superior results over the previous literature in all. [**?**]

Furthering the case for the capacity of the *GloVe* model, the comparative study carried out on the task of reading comprehension by [**?**] demonstrated that pre-trained *GloVe* embeddings outclassed other embeddings, including *word2vec* [**?**]. From their experiments, [**?**] continue to suggest that these embeddings, in their off-the-shelf format, also surpassed embeddings trained on the target text itself as

well as, to their surprise, an expanded corpus of data extracted from a domain commensurate with that of the target text.

2.3.5 How is *fasttext* different?

Both *word2vec* and *GloVe* are considered as models operating on a word-level by regarding the word as the atomic operand, however [**?**] argue that this approach is possibly sub-optimal when considering languages, such as Turkish and Finnish, where single words can have multiple morphologies, or comprise of exceedingly large vocabularies, or both. In contrast, they argue that the use of sub-word information lends itself well to these languages where the multiple morphologies of a word follow some form structure, such as specific verb conjugations.

To tackle this issue they extend the *word2vec* skip-gram model to operate at a sub-word level, adopting a bag-of-characters n-grams approach for word representation. As opposed to having each word represented by a vector, a word is represented as the aggregate sum of its constituent character n-grams.

The authors also demonstrate how a vector for an OOV word can be constructed from its character n-grams with remarkable similarity to a comparable IV word. Some of their results can be seen in **??**, where an OOV word "microcircuit" shows positive cosine similarity (in red) to an IV word "chip" between its constituent character n-grams "micro" and "circuit".

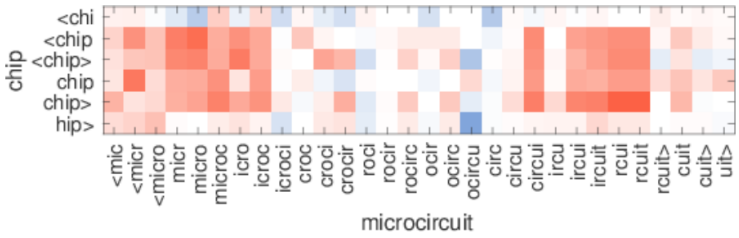

Figure 2.2: Character n-gram similarity between the OOV word "microcircuit" and IV word "chip". Positive and negative cosine similarity are denoted in red and blue respectively. Figure adapted from [**?**]

Need to include disadvantages of fasttext to make a contrast

2.4 Deep Learning

2.4.1 What are the fundamentals of a Neural Network?

Modelled on the human brain, neural networks typically involve a series of layers composed of neurons. Figure **??** illustrates one such simplified neural network architecture with a single hidden layer, L_2, preceded by the input layer L_1, and followed by the output layer L_3. (x_1, x_2, x_3) represents a three dimensional input vector, whereas the neurons, or hidden units, of the of the network are depicted as (h_1, h_2, h_3). The firing action of each neuron is expressed using a non-linear activation function. This action propagates from one neuron in a layer to all other connected neurons in the subsequent layer and is modulated by a particular weight that characterizes each intra-neural connection. At minimum, these networks will incorporate an input and an output layer that will encapsulate one, or more, hidden layers.

The network is able to learn, or model, a function by adjusting the weights to minimize a some measure of error towards a particular objective function. [**?**]

The hidden layers of a neural network architecture are able to extrapolate features at a level that cannot be carried out manually, thereby capturing more subtle details and generating richer representations of the data being learned. Foregoing this need for extensive manual feature engineering is one of the primary drivers of neural network models, even though these are approaches are typically black-box solutions, with obfuscated inner-workings.

2.4.2 What are the common activation functions?

Three of the most commonly used activation functions are sigmoid (eq. **??**), hyperbolic tangent (eq. **??**) and Rectified Linear Unit (ReLU; eq. **??**). Although the choice of activation function is typically heuristic, [**?**] as cited in [**?**] notes that the ReLU is easier to compute when compared to the other two and also tends to converge faster, all while maintaining similar or even better performance.

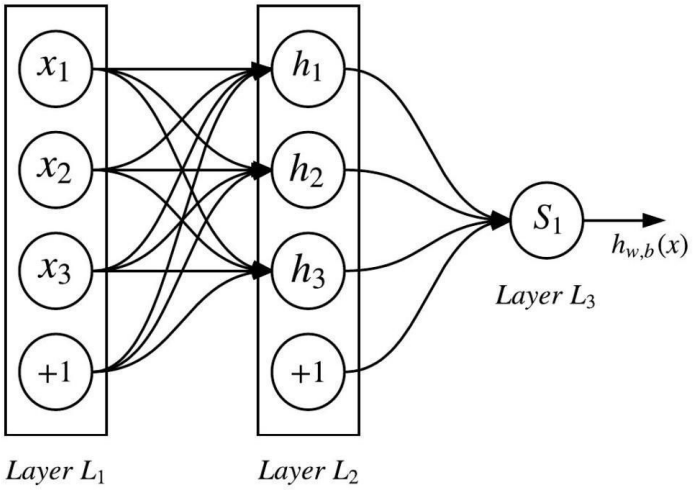

Figure 2.3: Standard feedforward neural network (FFNN) architecture [**?**]

$$f(W^t x) = sigmoid(W^t x) = \frac{1}{1 + e^{-W^t x}} \tag{2.5}$$

$$f(W^t x) = \tanh(W^t x) = \frac{e^{W^t x} - e^{-W^t x}}{e^{W^t x} + e^{-W^t x}} \tag{2.6}$$

$$f(W^t x) = ReLU(W^t x) = \max(0, W^t x) \tag{2.7}$$

2.4.3 How are Neural Networks typically trained?

Neural networks are trained by following the direction of the gradient that lessens the measured error rate of some objective function which is calculated through the repeated application of the chain-rule. The process can be carried out on the training set in its entirety, commonly referred to as batch learning, or in an on-line manner, carrying out updates after each training sample. The latter is known as stochastic gradient descent and is known to be more efficient and robust to local minima when dealing with large datasets [**?**] as cited in [**?**].

There will always be noise in the sampled training data that does not translate to the real world test data scenarios and that the model must therefore avoid learning. Training models to the point of becoming overly-sensitive to this noise

is referred to as over-fitting the data, and there are a number of regularization techniques often employed in the literature while training to counteract this.

One common approach is introducing some form of weight penalties, for instance, $L1$ and $L2$ regularization. Other techniques frequently used include *early-stopping*, whereby a fraction of the training data is used as a validation set that the model is tested against at regular intervals during training to test for a sustained improvement [**?**], and *dropout* [**?**], in which random neurons in a NN are ignored ("dropped") with some probability p, effectively training a diverse range of "thinned" versions of the original NN. An approximated average of those networks is subsequently used as final trained model by scaling its weights by that same dropout probability p.

2.4.4 How are Deep Neural Networks formed?

"Deep" neural networks are constructed by stacking a series of layers in such a way that salient features extracted from one layer are passed on as input data to the next layer, and so on. This stacking process, in theory, improves the capacity of the network to extract more abstract and expressive features that reside at a deeper level within the input data. [**?**]

2.4.5 Why is Deep Learning popular again recently?

In recent years, the astounding progress in computing resources such as GPUs and high performance distributed computing have made deep learning accessible on an unparalleled level. Consequently, this has driven interest in the applicability of deep learning architectures such as the CNN and RNN in a range of fields from computer vision to natural language processing [**?**], [**?**].

2.4.6 What are two common deep learning models used? (CNN and RNN)

Not least owing to their impeccable ability for effectively extracting highly expressive low-dimensional representations of text automatically, coupled with the aforementioned resurgence of deep learning due to the increase in available computing power, the use of neural network models such as [?] [?] [?] became increasingly widespread within the field of NLP, including targeted and aspect based sentiment classification tasks [?]; [?]; [?] [?].

The most prevalent architectures are the Convolutional Neural Networks (CNNs) and Recurrent Neural Networks (RNN) and the descendents thereof. Other deep learning models worth mentioning include the Recursive Neural Network (Rec-NN), which has been employed in works such as [?] and [?] for syntactic analysis and sentence sentiment analysis respectively. [?]

2.4.7 What are the fundamentals of a CNN (in brief)?

The layers typically comprise of filters, also referred to as kernels, of differing widths that slide over the input data to extrapolate various features. Each of these features would be representative of a diverse set of aspects of the data that would aid in defining it with respect to the task being tackled.

These convolution layers are coupled with a max-pooling layer with the intention of extracting the most salient values. These values can subsequently be forwarded to another convolution layer that presumably extrapolates deeper, more abstract features. This process can be repeated a number of times across multiple convolution layers to build a deep CNN that eventually produces a single feature vector representation of the original input data. Figure **??** illustrates this process using six filters with three difference widths and two diverse operations for each, the results of which are down-scaled using max-pooling, and subsequently passed through a softmax function for binary classification.

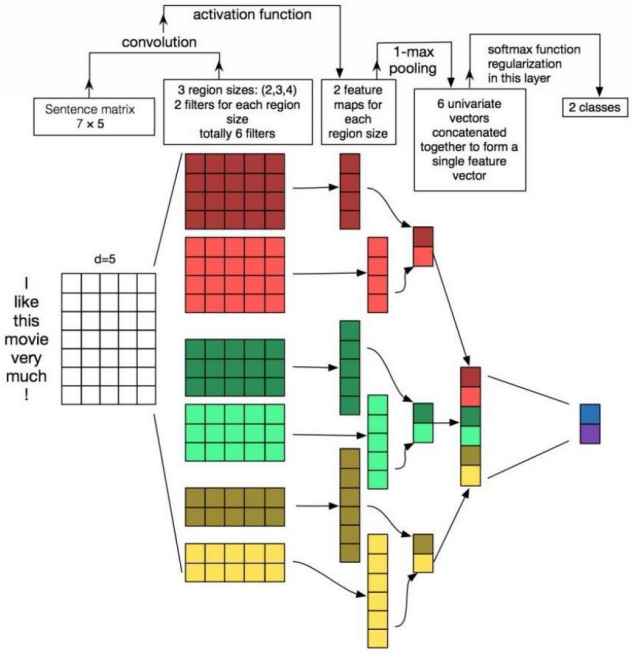

Figure 2.4: An example of a CNN architecture used for sentence modelling and subsequent binary classification [?] as cited in [?].

2.4.8 What are the advantages of using max-pooling?

Max-pooling is ideal for producing a fixed-length representation of the data, which is a common prerequisite for classification tasks, while at the same time preserving the most prominent features from the original data. [?]

2.4.9 What are some approaches using CNN?

Some of the first CNN-based approaches that paved the way for the growth of CNN architectures in the literature that followed were [?], [?], and [?]. [?] used CNN architecture for sentence classification tasks ranging from subjectivity and question type with promising results albeit in the face of a number of challenges that became evident to the authors. Not least of these was the limited capacity for the CNN architecture to capture syntactical dependencies in sentences that occurred over long distances. This challenge was the one of the foremost drivers for the work

that followed by [?] who developed the DynamicCNN (DCNN) which consisted of a series of hierarchical convolution and k-max pooling layers.

Other works cited in [?] include [?] that dealt with sarcasm detection on twitter data, noting a need for additional context information when dealing with short texts of this nature. This observation is also echoed in [?], who noted superior performance of CNN networks when dealing with longer text that would provide more contextual information as opposed to shorter text. Due to the vast amount of parameters that CNNs typically need to learn, scarcity of data is an often cited challenge [?].

Work carried out by [?] made use of a CNN to deduce the sentiment of the target based on the sentiment of the clause surrounding it. As noted by [?] however, this method still operated on the assumption that the most salient features of a word can be extracted from other words in close proximity.

A phrase such as "I bought a mobile phone, its camera is wonderful but the battery life is short, not particularly satisfied overall", challenges this assumption with respect to the "mobile phone", as the intended target since the most sentimentally-laden words appear at the opposite end of the sentence.

2.4.10 What are the fundamentals of an RNN based model (in brief)?

A Recurrent Neural Network (RNN) can be thought of as a chain of recurring modules which typically represent elements within a variable-length sequence, with an internal hidden state that represents the network's "memory". This state is passed forward from one module in the chain to the next.

Unlike a CNN, where each layer has its own set of trainable parameters that must be learned, a RNN uses a single set of parameters across all of the modules in the chain which significantly diminishes the total number of parameters that it must learn.

Through forwarding the hidden state from one time step in the chain to the

next, the network is able to "remember" information from previous elements of the sequence and use that information when generating a representation for the current element [**?**].

2.4.11 What makes the RNN more suitable for targeted sentiment analysis? (sequence)

The hidden state that characterizes RNNs acts as its "memory" element and makes these networks particularly effective in dealing with data that are sequential in nature. One of the most prominent examples of these data is language, where the significance of a word at one time step may be substantially altered by those that preceded it. Consider, for example, the word "dog", for which the meaning would shift entirely, from an animal to a popular American snack, should it be preceded by the word "hot" [**?**].

Moreover, the capacity of RNNs to model variable length sequences to a fixed length representation also makes them particularly practical when dealing with different units of resolution in languages including documents, sentences, and even words, all of which are naturally arbitrary in length. [**?**]

2.4.12 What is problem faced by RNN? (vanishing & exploding gradient)

In the setting of a traditional RNN, the tendency for a gradient to exhibit exponential change to the degree that prevents substantive learning increases with the length of a sequence [**?**] [**?**]. This phenomenon is what is implied by the terms "vanishing" or "exploding" gradients, where the changes observed over time steps often vanish with time or, although less frequently, but with equally devastating results, grow exponentially; hindering the network's capacity for learning.

2.4.13 What solutions exist to vanishing/exploding gradient problem? (LSTM and GRU)

One class of solutions to the vanishing or exploding gradient problem is to carry out particular modifications on top of the standard stochastic gradient descent algorithm, such as gradient clipping, whereby the norm of the gradient vector is *clipped*, or using second order derivatives, which may or may not be influences to a lesser degree. [**?**] The second, and more popular, class of solutions look instead to introduce further sophisticated, additive, non-linearities to the traditional RNN unit. These would selectively carry forward salient features and filter out irrelevant information from previous time steps, as opposed to overwriting the memory content at each time step.

Variants on the standard RNN network were proposed to address the vanishing and exploding gradient issue. The most popular of these being the Long Short Term Memory (LSTM) and, more recently, Gated Recurrent Unit (GRU). Other approaches include, but are not limited to, Residual Networks (Res-Net).

2.4.14 What are the fundamentals of the LSTM (in brief)?

LSTM [**?**] is an extension on the RNN model that addresses the issue of a vanishing or exploding gradients through the use of gates that control the flow of information from the past states to present states.

To do this, the LSTM introduces three adaptive gates that can be considered additional neural layers on top of the single neural layer that characterizes the typical RNN. The layers introduce an increased level of sophistication in the "remembering" process from one sequence point to the next.

These gates are commonly referred to as the input, forget and output gate. Moreover the LSTM also maintains a two inner states as opposed to one, namely the cell state and the hidden state.

The process undergone in each repeating module of the standard LSTM archi-

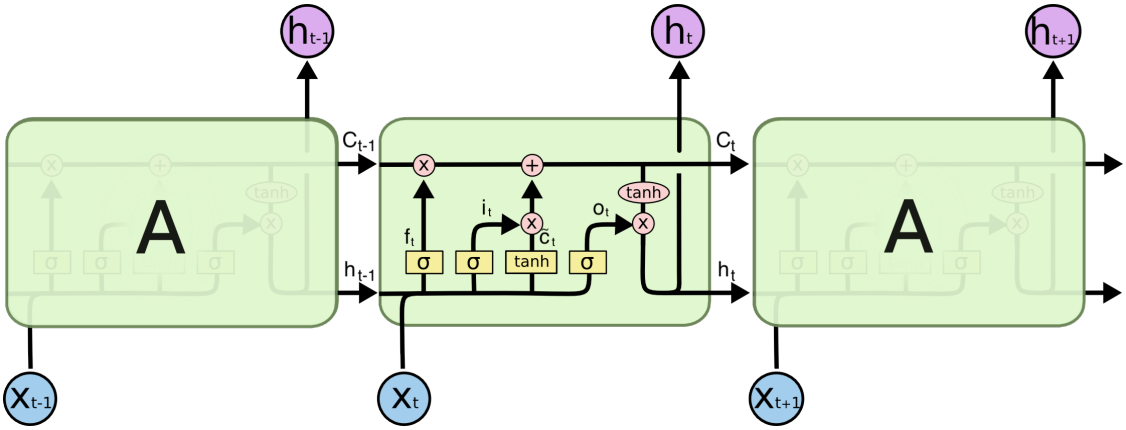

Figure 2.5: LSTM repeating module illustrating the four neural layers (Yellow Boxes) that comprise it. Point-wise vector operations are depicted in red boxes. Image adapted from [**?**]

tecture depicted in figure **??** starts at the forget gate. Here, the information to be removed from the cell state C_{t-1} is selected through the sigmoid layer (eq. **??**).

$$f_t = \sigma(W_f \cdot [h_{t-1}, x_t] + b_f) \tag{2.8}$$

Next, an update vector is produced through a point-wise multiplication operation between the input gate (**??**), which represents the values selected to be updated, and a vector of candidate values, $\tilde{C}$ (**??**). This update vector is added to the current cell state, from which the forget gate had previously removed information deemed redundant.

$$i_t = \sigma(W_i \cdot [h_{t-1}, x_t] + b_i) \tag{2.9}$$

$$\tilde{C} = \tanh(W_C \cdot [h_{t-1}, x_t] + b_C) \tag{2.10}$$

Finally, an output gate (**??**) regulates the amount of cell state information that is output to the rest of the network through the unit's hidden state h_t (eq. **??**).

$$o_t = \sigma(W_o \cdot [h_{t-1}, x_t] + b_o) \tag{2.11}$$

$$h_t = o_t * \tanh(C_t) \tag{2.12}$$

2.4.15 How does the LSTM model address the vanishing & exploding gradient problem?

The gating mechanism that is present in the LSTM network allows for the persistence of salient features that are encountered early in the sequence and which would otherwise be overwritten in a typical RNN architecture.

The input and output gates of the LSTM control the amount of memory content that is to be added to the memory cell and the memory content that is exposed to the rest of the network respectively.

Forget gates were later added to the architecture [**?**], which enabled the memory cells to reset themselves. It is important to note that the LSTM updates the memory cell independently from the forget gate, which is to say that, for the LSTM to "remember" a prior input in a sequence two conditions must be satisfied; the input gate must be closed, preventing the addition of new information and, the forget gate must be maintained open, as otherwise this would cause the existing memory content to be reset.

2.4.16 What are some approaches using LSTM?

[**?**] achieved results competitive with those obtained by [**?**] by using an LSTM, as opposed to CNN, to generate representations of tweets. [**?**] notes that their model was able to extract features over long distances at a fraction of the complexity of the DynamicCNN [**?**].

Another variant on the LSTM architecture involved the use of dependency parsing to build tree-structured LSTMs which operate on some specific tree-pattern that is extracted from the data, typically through the use of external parsing tools [**?**]. Approaches of this variety [**?**], [**?**], [**?**] have obtained promising results, however as [**?**] note, these are contingent on the data being well-formed structurally and grammatically, which is not guaranteed when dealing with micro-texts on social media platforms.

Two of the foremost extensions on the classical LSTM model when tackling target-based sentiment analysis, were proposed in [?] with the intent of accounting for the target information, these were termed the Target Dependent LSTM (TD-LSTM) and Target Connection LSTM (TC-LSTM). Their work showed that the changes proposed would improve the performance over a standard LSTM. [?] generate representations of a target's left and right contexts, where the target representation is concatenated to each. These two are then finally chained together to form the final, target-specific, representation of the sentence which is fed to an LSTM for sentiment classification [?].

TC-LSTM [?] was reported as one of the first neural network based methods to obtain state-of-the-art performance without the use of laborious feature engineering and external data sources. It is worth noting however that recent work [?] failed to reproduce the original results that were reported.

2.4.17 What are difficulties mentioned that LSTMs deal with?

While the issue of long-distance dependencies and CNN based approaches is often brought up, [?] remarks that LSTMs will still struggle with features that are located too far apart within a sequence, given the phrase "Except Patrick, all other actors don't play well", an LSTM would struggle to identify the positive sentiment on the opinion target "Patrick" due to the distance between the terms "Except" and "don't play well".

From the observation made by [?] when employing LSTM-based models in the field of machine translation, [?] make the case that the TD-LSTM would suffer from in instances where the most sentimentally salient word is located further away from the target being considered.

2.4.18 How does a Bi-LSTM differ from an LSTM?

When dealing with bounded sequences, a natural conclusion one might draw is that valuable information can be extrapolated from future contexts as well as past contexts. Bidirectional RNNs are employed with this specific intention in mind.

Building on top of the LSTM architecture, a Bi-LSTM is so called as it is the result of two stacked LSTMs, each responsible for processing and extracting features from a sequence in opposite directions. Features extracted from these two directions are then typically concatenated into a single, theoretically richer, and more expressive, representation of the sequential data.

It has been argued that the bidirectional nature of these models violates causality, which is justified to an extent, since a future context cannot be exploited if it is the same element that is being predicted. An example of this is predicting future stock market fluctuations. However, for tasks where this information is available, bidirectional variants that make use of this information have consistently been shown to outperform their uni-directional counterparts. [**?**]

2.4.19 What are some approaches using Bi-LSTM?

Sentence and document level sentiment classification tasks lend themselves well to the use of bi-directional RNN (BRNN) variants, since the boundaries of the sequence being classified are well-defined a priori. In their work, [**?**] suggest that using a BLSTM improves the ability of their approach to extrapolate phrase-like features when compared to a uni-directional sequential approach.

An improvement in results stemming, in part, from taking a bi-directional approach is also observed in [**?**], in both accuracy and macro F1 scores when compared to the work of [**?**], from which the former was inspired. More recent works (eg. [**?**], [**?**]) that have reported notable results in the field also take advantage of bi-directionality in their approach.

2.4.20 What are the fundamentals of the GRU (in brief)?

The GRU does away with the cell state that is found in the LSTM, maintaining only a single hidden state as its "memory". The second way in which the GRU differs from the LSTM is in its number of gating units, by foregoing the output gate and combining the input and forget gates into a single update gate, the GRU benefits from having less parameters to learn.

$$z_t = \sigma(W_z \cdot [h_{t-1}, x_t] + b_z) \tag{2.13}$$

As the name implies, when the reset gate (eq. **??**) nears 0, it allows the GRU unit to drop the previous information, which may be deemed inconsequential at a later time, and reset itself with the current input only. This information is then used by the GRU to produce a vector of candidate activation values (eq. **??**).

$$r_t = \sigma(W_r \cdot [h_{t-1}, x_t] + b_r) \tag{2.14}$$

$$\tilde{h}_t = \tanh(W \cdot [r_t * h_{t-1}, x_t] + b) \tag{2.15}$$

Instead of maintaining an internal memory cell, the GRU uses the update gate and carries out a linear interpolation function (eq. **??**) to control the amount of candidate activation, $\tilde{h}_t$, that is added to the previous activation. In this way the update gate serves as the primary mechanism preventing the GRU from overwriting a previously encountered salient feature.

$$h_t = (1 - z_t) * h_{t-1} + z_t * \tilde{h}_t \tag{2.16}$$

Whereas the LSTM utilizes the output gate to control the exposure of its internal memory to the rest of the network, the GRU does not have any means by which to control the exposure of its inner state, and therefore exposes its hidden state in its entirety to the rest of the network.

Each unit in a GRU model will have separate update and reset gates that will independently learn to capture long and short term dependencies in a sequence. Units that learn to capture longer-term dependencies will have active update gates whereas, for short-term dependencies, the reset gates will tend to be more active.

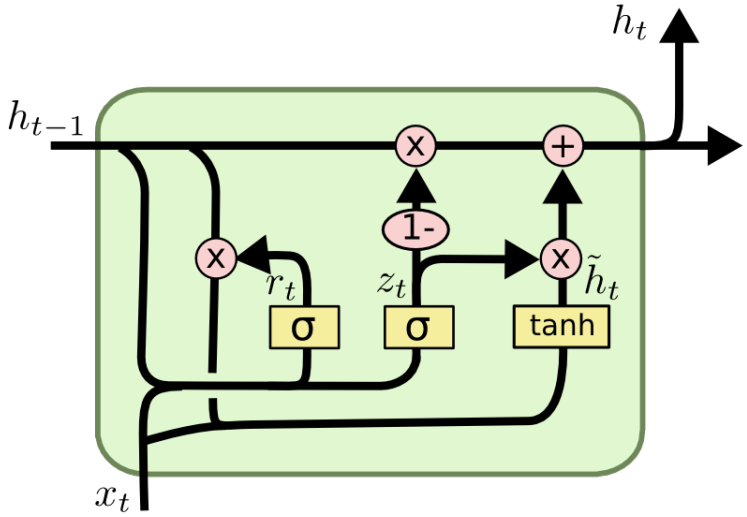

Figure 2.6: Internal gating structure of a GRU unit. [?]

2.4.21 What are some approaches using GRU?

[?] model the interplay between targets and their contexts through gated mechanism operating on the a representation of the original sentence split into three components using a gated-RNN.

[?] use a GRU as part of their approach, serving as a gating mechanism to determine whether to update a specific part of memory based on past activations. They claim state of the art results in aspect detection and sentiment classification, outperforming SenticNet [?], and foregoing the need of external knowledge.

[?] use gated tanh-ReLU units to control flow of features from convolutional neural networks to max pooling layer. They deal more with the advantages that a CNN offers, since it is not time-dependent and is therefore easier to parallelize.

[?] generate a sentence representation by stacking two GRU based networks,

similar to the BLSTM configuration, opting to go with the GRU architecture instead since it has less parameters to learn, noting similar results. A continuous vector representation of the target is sandwiched between the two GRU direction outputs, and fed to softmax classifier. Authors report accuracy and macro-f1 scores better than state of the art obtained on a twitter dataset [**?**].

2.4.22 What advantages/disadvantages does the GRU have over the LSTM if any?

Since the architecture of the GRU is less complex than that of the LSTM, there may be circumstances where the former may be more efficient than the latter since it requires less parameters to learn [**?**], [**?**].

That being said, there is no clear winner between the two variants, as demonstrated by work conducted in [**?**]. The results obtained, shown in figure **??**, concluded that the only demonstrable advantage is that of both variants over a standard RNN architecture. The authors go on to say that the performance of the two variants themselves is contingent on the particular nature of the tasks being addressed. Although it is worth noting that the work carried out by [**?**] was not specifically in the field of NLP, the deciding factor between these variants in NLP literature still tends to be heuristic [**?**].

2.4.23 Are there any advantages of CNN over RNN models?

In spite of various studies illustrating the fact that CNN networks are data heavy in nature and often require auxiliary data when dealing with micro-texts such as those obtained from social media networks such as twitter, [**?**] note that assuming the superiority of RNNs for sequential data is inaccurate.

They proceed to cite studies where CNNs performed competitively, and even surpassed RNN-based architectures in tasks for which the latter would, in theory,

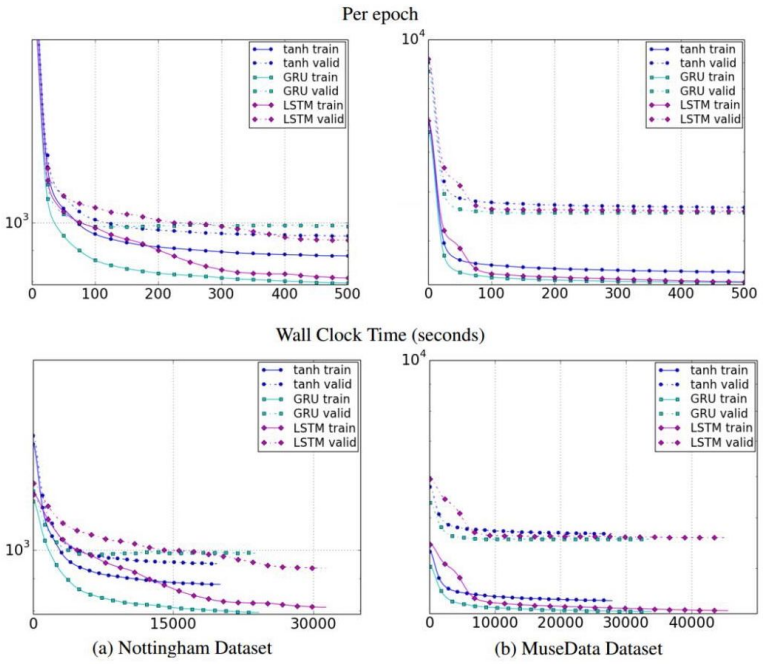

Figure 2.7: Results obtained by [?] during training and validation of different RNN variants illustrating the superiority of LSTM and GRU units over the traditional RNN.

be better suited, such as language-modelling [?] as cited in [?].

It is worth keeping in mind that when dealing with language modelling, RNNs and CNNs approach the task from separate avenues; while RNNs, and their descendents, have no context boundaries when processing sequential data to generate a representation, CNN-based approaches seek to extrapolate the most meaningful n-grams from bounded-contexts from that sequence, to produce their final representation. [?]

2.4.24 Are there any approaches that use both CNN and RNN models?

There have been works in the field of sentiment analysis, targeted and otherwise, with the intention of taking advantage of the benefits of both CNN and RNN based models in an ensemble approach. *Finki* [?] and *BB_twtr* [?] are two solutions of this nature, dealing with sentiment analysis of tweets. These models involve the

coupling of LSTM or GRU models with CNN models to carry out both binary, and fine-grained (five degrees) sentiment analysis on tweets. The works were part of the submission made to the yearly SemEval conference, and each performed impressively in their respective rounds.

While [**?**] and [**?**] were not addressing *targeted* sentiment analysis, recent works such as [**?**] and [**?**] also marry GRU or BLSTM with CNN models. Interestingly, both works cite the simplicity of their approaches when compared to more complex attention mechanisms as a driving factor, while still achieving comparable results in their experiments.

2.4.25 What does an attention mechanism seek to model?

The intuition behind an attention mechanism is that, when dealing with sequential data, different parts of the sequence contribute to varying degrees towards a specific task or goal. This is abundantly apparent in the field of machine translation, where the attention mechanism made its debut [**?**]

When translating a particularly long sequence, for example, it is natural to focus principally on specific regions related to the current element being translated, as opposed to the source sentence as a whole, as this would result in the most relevant parts possibly being shadowed by unrelated material.

2.4.26 Where was attention first applied?

The remarkable power of attention mechanisms for machine translation [**?**] sparked an interest in investigating their applicability in a range of other tasks, including target, and aspect, based sentiment analysis [**?**] [**?**] [**?**] [**?**], image captioning [**?**] and question answering [**?**], where the notion of the most salient features being dispersed unevenly across the source data also holds true.

2.4.27 What are the fundamentals of the standard attention mechanism?

In its simplest form, an attention mechanism involves a layer which produces a weight vector that represents a distribution of salience across an original feature vector, which would otherwise be considered evenly in its entirety, with respect to some target. The nature of the target varies depending on the downstream task, in targeted or aspect based sentiment analysis this is typically the target or aspect in question, whereas in fields such as machine translation this could be the last hidden state that was output.

This process typically takes the form of some scoring function, such as a simple FFNN, which can be trained alongside the rest of the model, followed by a softmax layer. Given a series of representations $[h_1, h_2, ..., h_n]$ and some target t, the scoring function a calculates the salience of each h_i with respect to t. Subsequently, the softmax layer squashes the attentions scores into a valid distribution vector α with values in the range $(0, 1)$.

$$\alpha_i = \frac{\exp(a(h^i, t))}{\sum_{j=1}^{n} \exp(a(h^j, t))} \tag{2.17}$$

This weight vector is then used to produce a context vector c, as the weighted sum of the original feature vector. This process will amplify features with high attention weight values (approaching 1) while attenuating features with low attention weight values (approaching 0).

$$c = \sum_{i=1}^{n} \alpha_i h_i \tag{2.18}$$

2.4.28 What are some approaches that implement attention?

ATAE-LSTM [?] was one of the first to incorporate attention into an LSTM model for the purposes of aspect-based sentiment analysis. A vector representation of the target aspect is used as the subject of attention, allowing the model to attend to different parts of a sentence with respect to the aspect.

[?] make use of multiple attention layers to extrapolate the most salient, and sentiment-bearing words with respect to a target, suggesting that through more than a single attention layer, the model would be more effective at extrapolating features over long distances. They subsequently aggregate these attention results non-linearly using a GRU. The authors attribute their preference of a GRU over an LSTM for this stage in the process due to the former requiring less parameters than the latter.

[?] argue that prior works had focused primarily of the representation of contexts and not on targets themselves. When considering targets that consist of multiple words, the idea that words should not necessarily contribute equally to the final representation of that target is a valid assumption to make. They cite the example of "picture quality" as a target and argue that in such a case the word "picture" would play a more important role that "quality". To address this, their Interactive Attention Network (IAN) incorporates two attention networks to model both the target and the context interactively, to obtain a representation of the effect each had on the other.

More recent work, by [?], build on the intuition of producing better target representation as well as context representation. Their LCR-Rot model is characterized by a novel "rotary attention mechanism" that attempts to better capture the interplay between a target and its context as well as the contexts on the target. They argue that left and right contexts affect the target representation to a degree that merits a separate representation of the target for each, one which is

"left-aware" and another that is "right-aware". [?] demonstrate the effectiveness of their rotary engine approach citing state-of-the-art results in accuracy on three distinct datasets, suggesting that properly modelling the effect of the target on the context may be as important as that of the context on the target.

2.4.29 What is the rationale behind memory networks?

It can be argued that one of the most prominent contributions of the attention mechanism covered in the previous section is the ability to selectively read information, typically from the internal state of a model, in a differentiable manner by reading from all of the data, to varying degrees. This, however, has more profound implications: the same process can be applied to selectively write in a differentiable way. Memory networks are so-called as they exploit this fact, providing an external memory store that a model can learn to refer to and update over time.

2.4.30 What were the first works that employed memory?

This concept was first actualized in two distinct, yet coincident, studies: [?], who proposed their Neural Turing Machine (NTM) and [?] who put forward their Memory Neural Network (MemNN) framework.

2.4.31 What are the fundamentals of memory networks?

In their seminal work, [?] describe their memory network framework as comprising of some tangible memory component, which is represented as an encoded continuous matrix. This external memory is updated through the use of neural network operations which selectively read and write to and from it. They proceed to conceptualize these operations in the form of four fundamental components.

The first component, the input component, I, is tasked with mapping incoming data to an internal representation. Pre-processing and embedding look-ups are two examples of operations that may be entailed at this stage. A generalization compo-

nent, G, follows, which uses the data from I to update the memory. G is so-called as it allows for the potential of generalization of existing memory towards some future goal. In its simplest form however, this component simply stores incoming data in the next available memory slot, leaving existing memory unchanged. Using the input data from I and the current memory state an output component, O, produces an output feature vector which typically involves inferring the most relevant memories. This output is finally interpreted by a response component, R, to the desired format.

Each component that makes up this process may represent any trainable model such as an RNN or SVM, and trained accordingly. In their original approach, [**?**] use "hard" attention when probing for the most relevant memory evidences, where the number of highest scoring evidences is a tunable parameter. [**?**] build upon this idea by opting instead to use a softmax operation, which is conceptually comparable to using a "soft" attention mechanism over the external memory store. Moreover, unlike its "hard" counterpart, this makes the process differentiable, making the model trainable in an end-to-end fashion, requiring less supervision when compared to [**?**].

This framework put forward in [**?**] and subsequently extended by [**?**] served as the foundation from which most memory-based targeted sentiment analysis approaches emerged.

Furthermore, [**?**] show that the performance of their model can be enhanced by having the O component repeatedly attending to memory for a number of consecutively stacked "hops". Indeed, this observation is echoed in subsequent studies inspired by their work in the field of targeted sentiment analysis (eg. [**?**], shown in figure **??**), the intuition being that through these consecutive hops the model is able to attend to richer, more abstractive, features than those existing solely on the surface level of the data.

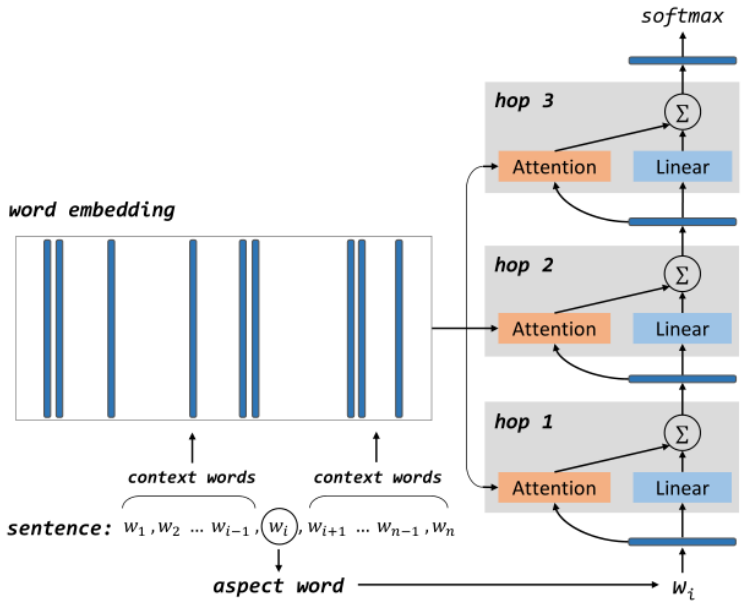

Figure 2.8: The deep memory network approach for targeted sentiment analysis, with 3 hops. The model stored the context of a target in memory and repeatedly attends to that memory with respect to the target word w_i. [?]

2.4.32 What are some targeted sentiment analysis approaches that use memory networks?

Memory networks were first put forward towards the goal of targeted sentiment analysis by [?] (figure ??). Their approach was inspired by [?], with some differences in the attention function that was used, opting instead to use the method put forward in [?]. The authors reported results outperforming the state-of-the-art SVM-based model by [?], which required extensive manual feature engineering, absent from their proposed memory network, as well as the far more complex LSTM based models [?] which required substantially more time to train. Similar to [?], the authors noted a marginal increase in performance as they increased the number of computational hops, which capped at around 8 hops.

[?] construct a location-weighted memory module from the hidden states of a BLSTM placed between the input and the attention modules so as to better capture information from phrases comprising of multiple words, such as “not wonderful

enough". Moreover, unlike the [?], they combine the results of their recurrent attention layers non-linearly. These improvements led to a performance boost over [?]. The proposed model architecture is illustrated in figure ??. It is also worth pointing out that the authors failed to reproduce the results originally cited in [?].

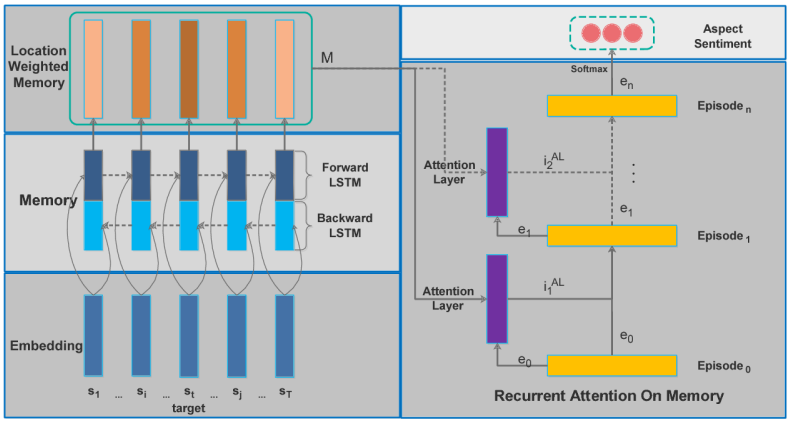

Figure 2.9: The recurrent attention model architecture showing the placement of the location-weighted memory module with respect to the input and attention layers. As opposed to [?], the attention values are combined from one episode to the next non-linearly, using a GRU. [?]

[?] maintain a memory chain of entities that are encountered while processing a phrase and develop a gating mechanism that determines how those memory chains should be updated. The gating mechanism accounts for the content and the location of the memory chains compared to the current input, as well as the past activations, through the use of a GRU unit, when choosing to update a particular memory element.

2.4.33 Why is it imperative to attend to targets as well as contexts?

Recently, work carried out by [?] showed an interesting performance boundary on approaches to targeted sentiment analysis that make use of attention mechanisms. In particular, [?] note that when context words have diametrically opposing sentimental bearings based on the target being considered, this cannot be modelled by

improving by attending to the context alone.

To illustrate this, [**?**] consider two different targets, *price* and *resolution*, in four different phrases; "high price", "low resolution", "high resolution" and, "high price". Some sentiment score, s, for these phrases, where $s < 0$ implies a negative sentiment and $s > 0$ implies a positive sentiment accordingly (eq. **??**). The attention weight α will have a value of 1 since the context consists of a single word which is represented by h. v represents the target and W is the weight matrix the model must learn.

$$s = W(\sum_{i=1}^{n} \alpha_i h_i + v) = W(h + v) \tag{2.19}$$

From equation **??**, the following inequalities can be obtained,

$$W(h_{high} + v_{price}) < 0$$
$$W(h_{low} + v_{price}) > 0$$
$$W(h_{high} + v_{resolution}) > 0$$
$$W(h_{low} + v_{resolution}) < 0$$

Expanding these inequalities results in $Wh_{high} < Wh_{low} < Wh_{high}$, which is a contradiction, and can therefore not be learned by the model. [**?**] point out that this condition cannot be rectified through further attending to the context but rather ameliorating the representation of the targets v to better capture the complex relationship they have with the context in a way that possibly reverses the polarity of the final sentiment score s.

Towards this end, [**?**] experiment using a series of techniques, and note an improvement over previously cited results as well as a direct improvement on the RAM model [**?**] when these are coupled with their optimally performing target-sensitive context modelling strategy.

2.5 Out-of-vocabulary Words

2.5.1 What are OOV words?

A common theme in the deep learning models that have been discussed herein is the uncanny similarity between the techniques that these models employ to understand information and the same techniques that the human brain takes advantage of, both consciously and unconsciously, to produce an understanding of the information it is presented, with respect to a particular goal.

Traits such as properly identifying the most important and informative elements of a sequence, and referring to past events when making such decisions as well the fundamental process by which the intricacies of the networks are tuned when errors are made in training, in an effort to lessen such errors in the future.

In this analogy, when dealing with language-related tasks such as targeted sentiment analysis, the word embedding matrix from which different words' are represented numerically can be regarded as the working vocabulary that the model has prior to tackling the task at hand.

It stands to reason that due to the ever changing and ever evolving nature of language, it is impossible to account for the entire vocabulary of a particular language when constructing word embeddings. Regardless of the amount of text that is initially used to construct the word embeddings there shall be words that are not encountered within that text and therefore a continuous vector representation of that word could not be produced. While the probability of covering the most commonly occurring words increases with the size of the original text and the variety within it, encountering new words is an inescapable eventuality. These previously unseen words are consequently referred to as out-of-vocabulary (OOV) words.

The magnitude of the challenge that OOV words present, and the potential repercussions thereof, become evident in the landscape of this analogy. These represent words which the models essentially have no understanding of and can be

of little help to it when attempting to extract any information it may convey to the task at hand. While the model can be expected to learn more about these words through repeated encounters in different contexts, the fact that these words, by their own nature, tend to occur infrequently in language substantially diminishes the efficacy of this learning process.

2.5.2 What problems for generative tasks do OOV words create?

OOV words are of particular concern when dealing with tasks that are generative in nature, such as ASR. The toll of OOV words on the performance of approaches to these tasks is two-fold. Firstly, OOV word may be substituted with an incorrect IV word. Secondly, the OOV word has a direct effect on the neighbouring IV words [**?**].

A common approach to this problem is clustering OOV words into groups that would are sufficiently expressive of their constituents. Various techniques have been employed towards this goal such syntactic and morphological features, part-of-speech tag information, online resources, and subword-level models to name a few, [**?**] does a good job of outlining these approaches.

2.5.3 What problems for sentiment analysis do OOV words create?

Since sentiment analysis is classification task, where words are provided as input and subsequently used as keys when looking up the relevant embedding vector, substituting an OOV word for an IV word is of no concern. The effect of an OOV word on its neighboring words, however, is prone to undermine a model's ability to generate an accurate representation of the content as a whole.

This sort of phenomenon is not particularly difficult to imagine since it is often times the case in languages that a single word can have drastic effects on the

meaning of a phrase, particularly in situations expressing negation. Consider a phrase such as "It avoids all the predictability found in Hollywood movies.", where "predictability" conveys a negative sentiment, which is subsequently negated by the verb "avoids".

Moreover, OOV words obviously make the process of comprehending a phrase more difficult by introducing elements that the model has no knowledge of. If the word embedding model that is being used is analogous to the model's understanding of a language, an OOV word is effectively a word the model does not understand, and therefore has limited means by which to gauge the effect of that word on the overall sentiment of the phrase, if any.

2.5.4 How are OOV words typically regarded in Sentiment Analysis?

A typical approach to this OOV challenge within the field of sentiment analysis is the use of a particular singular token that is meant to represent low frequency words during the training phase, and subsequently model all OOV words encountered in the test phase. The vector for this token is often times initialized using some bounded random uniform distribution.

2.5.5 Why is this sub-optimal?

As far back as [**?**], before the popularity of pre-trained word embeddings such as *GloVe* and *Word2Vec*, it was pointed out that using a single token is somewhat crude. It could not possibly encompass the wealth of linguistic information expressed by every OOV word that is encountered; consider that an OOV word can be anything from a spelling mistake to proper noun, such as the name of an entity, and anything in between.

When training an n-gram model, the use of a single $<UNK>$ label for all OOV words will lead to a substantial inconsistency in the frequency of OOV words

between training and test datasets [**?**]. This inconsistency is comparable to the possibly counter productive training that is carried out on the singular $<UNK>$ vector across different samples within the scope of sentiment analysis and word embeddings.

In their work, dealing particularly with OOV tokens within the field of Reading Comprehension (RC), [**?**] note a considerable drop in performance when taking this approach in some cases and suggest that a unique OOV token would lack the desired level of detail to correctly generate a correct answer.

2.5.6 Why are classes better for handling OOV words?

It is not necessarily useful to approach the OOV word challenge at the word-level. It is assumed that OOV words are scarce, which substantially limits the occasions for a prospective model to learn any discerning information about that word. Attempting to model clusters of OOV words instead, would benefit each member of the cluster by the accumulated frequency of all members [**?**].

Moreover, within the scope of sentiment analysis, the intuition for clustering words under classes characterized by some particular sentimental value, or a lack thereof; as in the case of registered trademarks, would be evidently beneficial.

2.5.7 What is the trade-off between too many and too few classes?

Too few classes may not possess a sufficiently fine level of detail in their discerning characteristics and cluster together words which are unrelated and subsequently erroneously trained together. This can be seen from the extreme of this case, where only a single token is used, and the issues that have been reported for this approach.

Conversely, if an excessive number of classes are used, this would naturally decrease the amount of OOV words within each class, and consequently the frequency

of words appearing in a particular sample. This hinders a model's ability to learn any distinguishing characteristics of a class. Taken to the extreme, if each class were to contain only a single word, this would effectively render each class as a randomly initialized vector for this word which is rarely encountered, and trained. This undermines the purpose of a word vector, which is to convey as much information about the word as possible.

2.5.8 Why do i think there are benefits to be had from better OOV handling in SA?

Within the scope of a RC task, [**?**], carry out a study to accurately measure the effects that different embeddings and OOV approaches can have on the final result of two benchmark models.

They outline the typical approach to RC problems as initially generating a representation of the source document, possibly through the use of pre-trained word embeddings such as *GloVe* in conjunction with statistical models such as the LSTM [**?**] which may employ an attention mechanism (eg. [**?**]). The result of this process is a contextual representation of the document from which a valid answer can be extracted.

It is worth noting that this process is not dissimilar from the majority of approaches that have been adopted recently within the field of sentiment analysis. Both employ similar techniques and maintain the same characteristic order of events in generating a substantive representation of the source, differing only in the objective and hence the final product to be extracted from that representation. While this is by no means an insignificant difference, on a macro level this can be seen merely as adjusting the variables and parameters that are input to the system, as opposed to the system as a whole.

In their work, [**?**] suggest that there are notable effects on the downstream results of models when comparing the use of different word embeddings, pre-trained or otherwise. Specifically, as an out-of-the-box solution, they recommend the use of

GloVe [?] 200-dimension pre-trained embeddings. Moreover, for their benchmark RC models, they recommend assigning random unique vectors for OOV tokens at test time, possibly due to the fact that subjects in generated responses are likely to be OOV token and proper nouns.

Based on these findings, the aforementioned similarity in the process of tackling RC tasks and SA tasks, along with the challenges that OOV words pose in the field of SA as previously outlined, the study of word embedding choice and OOV approaches and their effects therein is something that we believe merits further investigation.

2.6 Reproducibility

2.6.1 What is reproducibility and why is it important?

The ability to reproduce experiments is the integral basis upon which all disciplines of science are founded. Within many fields, NLP among them, this typically entails adherence to three integral guidelines, namely, (a) the provision of sufficiently detailed methodologies, (b) the release of operational code-bases and (c) access to the dataset(s) with clear details pertaining to any processing and/or stratification strategies used. These guidelines ensure that results can be easily reproduced, evaluated for generalizability and compared to other methods in the field, thereby fostering growth.

2.6.2 What is the state of reproducibility in the field of Targeted sentiment analysis?

In [?], the authors underscore the significance of reproducibility of approaches as well as the generalizability of the results that are reported, and proceed to argue that adherence to the aforementioned tenets has been lacking in recent years, paying particular attention in their work to the field of targeted sentiment analysis.

The authors also draw attention to the fact that a multitude of studies report results on different datasets which often stem from diverse sources that could be composed of language that is centered around a particular topic. Notable still, these datasets also exhibit consequential statistical differences such as the average length of, and/or amount of targets in, a sentence. Occasionally, studies also carry out particular alterations to existing datasets or adopt a specific strategy for merging one or more datasets (eg. [**?**]). These factors substantially limit the possibility of effectively comparing the novel approaches as they emerge in the field.

Replication studies such as [**?**] can remedy this issue by attempting to reproduce the studies in a comparative setting, however they outline challenges in this regard as well. The authors note that a number of approaches in the field fail to outline the precise pre-processing steps they adopted, which may have substantial effects on the downstream performance of a model, and thus make reproducing the results difficult. In some situations they also note model settings mentioned that are not necessarily self-evident, such as a "softmax clipping threshold" [**?**], which, in attempting to reproduce the study, [**?**] were forced to ignore as they were unfamiliar with the term.

One key observation that [**?**] also make with regards to deep learning and neural network based approaches such as [**?**] is the influence that an initial random seed has on the final performance results, particularly when using smaller word embeddings ([**?**] as cited in [**?**]). They mention this as the probable reason for other studies [**?**],[**?**] (including their own), not being able to recreate the original results reported in [**?**]. The authors remark that in situations such as this, when dealing with models of this nature, it would be well-advised to gauge a model's performance over a number of experiment runs as opposed to one.

2.7 Objectives

Following the principles outlined in [**?**], and motivated by the observations made by [**?**] in the field of reading comprehension and the effect of OOV embedding strategies thereof, the objectives of this study are two-fold.

1. We intend to extend the work carried out by [**?**] in the field of TSA to cover a wider range of studies, including those that employ techniques that have since emerged focusing on attention mechanisms and memory networks. This entails the attempt to reproduce these models based on the detail provided in the original studies and subsequently carrying out a comparative evaluation of these approaches across a wide range of datasets from varying domains using a number of different pre-trained word embeddings.

2. Inspired by the work and findings of [**?**], we shall endeavor to investigate the effect that different OOV embedding strategies and pre-trained word embeddings have on the downstream performance of models with respect to TSA. To our knowledge, at the time of writing, this study will be the first to investigate this issue in detail.

To achieve these goals, we make three principal contributions through this work:

- A publicly accessible framework that provides access to a range of frequently cited datasets that have been used in the field of TSA. This framework shall be used to obtain robust performance metrics, such as macro-f1 scores for all models, which have been hitherto unreported for a subset of the models, as well as other informative measures that shed light into the inner workings of the models implemented, where applicable (such as attention heat-maps).

- A comparative evaluation of models across different domains and datasets, using different pre-trained word embeddings to ascertain the degree to which results obtained are reproducible and generalizable.

- Detailed reports on a series of experiments using different OOV embedding strategies, across all implemented models, the results of which will allow us to deduce the degree to which these affect downstream performance and whether an optimal approach can be found that proves to be generally beneficial.

Finally, the proposed framework shall also serve as groundwork for future experimentation into alternative, more sophisticated, OOV embedding approaches while also providing a means of rapidly carrying out comparative evaluations of TSA models across different datasets and pre-trained word embeddings.

3. Design

3.1 Data Module

The primary task of the data module is to produce the features data for the experiment module. We sought to adopt a modular design paradigm since this allows different responsibilities to be delegated to the appropriate sub-component.

Each sub-component of the data module is responsible for writing and indexing data to an external file-system using a specific hashing policy when this data is first generated. In subsequent experiments that require this same data, the sub-component is tasked with querying the file-system for its availability and importing it, instead of having to reproduce it.

Maintaining this indexed data store on the external file-system has the benefits of speeding up computation for subsequent experiments that use the same data with parametric alterations, facilitating investigation of the data produced at intermediary steps, aiding in debugging and allowing the framework to isolate the minimum requirement of data that to upload for experiments carried out through online services.

3.1.1 Dataset Import Script

It is imperative to evaluate the performance of TSA models on datasets that have different characteristics, as a measure of the applicability of the model under different scenarios, or for the purposes of fine-tuning a model to a specific domain. Datasets from different platforms and domains vary in their vocabulary and other vernacular features, such as colloquialisms. Different platforms also provide samples of varying lengths and which may express different sentiments towards multiple targets in the same sample.

Raw Data Parsing Function

The import script simplifies the process of importing new datasets to only requiring a, typically short, parsing function which reads the raw data files and mutates the data structure to records consisting of four fields: sentence, target, sentiment label and, offset[1]. The parsing function is an inevitable requirement since it is unfeasible to account for all the possible data formats of prospective datasets, the specifics of this function are explained in §**??**.

Once imported, the dataset is stored and indexed on the file-system for future use under a user-defined name. The imported data is stored in two dictionaries, for training and testing, and exported as binary pickle[2] files, which provides the fastest access times at minimal storage cost.

3.1.2 Dataset Component

The dataset component is primarily tasked with loading dataset files which have been imported as well as generating the accompanying information files used at other points in the process. Moreover, the component is also responsible for creating and exporting re-distributed variants of datasets, serving as a wrapper around these

[1] required to identify the correct target in situations where it occurs more than once in the same sentence

[2] python serialized object file

operations with which other sub-components in the data module can interact.

The corpus file that is generated by the dataset component provides a record of the occurrence count of each unique term in the dataset and is subsequently used to filter the vocabulary of an embedding to the minimal set of required terms. The *.json* file that is generated provides information on the distribution of the dataset, later used by the experiment component in producing a visual representation of this information. Two variants of each file are created, for the training and testing subsets of the dataset respectively.

Dataset Re-Distribution

It is often the case in datasets typically employed for TSA that one or more sentiment labels are over-represented in the data, indeed, this phenomenon is common across all the datasets gathered for this work. This is accounted for at the output stage by using appropriately weighted performance metrics, as detailed in §5 .

This notwithstanding, such an imbalance in classes causes the model to be over-exposed to a particular sentiment at the cost of others, which may be detrimental to the learning process. To address this issue, while also providing a means of investigating the effects of these imbalances, the framework provides a means of re-sampling the data, either to balance the distribution across all classes, or to any custom distribution as specified by the user.

The implementation details of the algorithm developed for this purpose are provided in §**??**.

3.1.3 Embedding Component

As with the dataset component, the goal of the embedding component is to provide a wrapper around reading and writing to and from an indexed embeddings directory by specifying a singular embedding type that exposes all the required data to allow the other sub-components to interact with it, facilitating the integration of additional embedding sources later in the development stage.

The commonly used embeddings are identified using a pre-defined shorthand, which the embedding component takes as input, along with a comma-delimited list of filters to apply on the embedding vocabulary. Function filters are specified using a custom name that is assigned when creating the filter. The "corpus" keyword is reserved within this scope and applies a set filter on the embedding vocabulary using the corpus of the datasets being used for the experiment. The embedding variable is initially obtained using the *GenSim* tool, which provides a means to download some commonly used embeddings. Once the embeddings are downloaded, the data files are stored and indexed within the embeddings directory of the framework, enabling offline access.

Filtering Embeddings

If one or more filters are specified, the embedding component extracts the vocabulary of the source embedding, unifies the requested filters into a single pipeline, runs the source vocabulary through this pipeline, and finally re-constructs the embedding on this filtered vocabulary. The resulting data is exported and indexed, along with a JSON file with the details of the filtration process and, in the case of function filters, a CSV report detailing which terms were filtered and what condition was met that resulted in this term being filtered.

The first type of filters that the embedding component supports are set filters, which simply define the set of terms to include from the entire embedding vocabulary. These filters are primarily used to limit the size of the embedding to contain only terms that occur in a dataset corpus, substantially reducing the file-size overhead of the embedding. The other type of filters that are supported are function filters, which refer to a function that, for each term, returns true or false as to whether to keep or discard it. These filters can be used to remove vocabulary terms based on particular POS tags and were implemented in an effort to investigate additional methods of reducing embedding sizes with a minimal cost to downstream performance.

Processing an entire embedding vocabulary to obtain NLP tags can be significantly time-consuming due to the size of these vocabularies. In an effort to optimize this process, a base function-filter is implemented from which the end-user extends, to construct custom function-filters. This base class automatically exposes the NLP attributes requested by the user while also allowing the framework to gather all filters into a singular pipeline, used to determine the minimum amount of processing needed to obtain the attributes that this pipeline requires. The implementation details of this procedure are detailed in §**??**.

In order to run multiple experiments using cloud computing services, all of the required data for these experiments need to be uploaded for each job. Apart from maximizing the ability to share data between experiments, reducing redundancy, the final size footprint of this data needs to be minimized to obtain feasible upload times. Since embedding variables were the largest bottleneck in this regard, set filters were used to reduce the size of embeddings to the bare necessity. Although the full embedding vocabulary would still be used at a production level, this reduction in embedding size also provided a marked speedup in look-up operations when indexing datasets, since the size of the look-up table was drastically reduced, resulting in faster experimentation times during development. Function filters were developed as an exploration of other techniques for further size reduction and their effect on downstream performance. The intuition behind this is that the contribution of subsets of vocabulary, such as adjectives and nouns, would suffice within the scope of TSA.

It is essential that the filtering mechanism be intuitive so as to enable extensibility by the end user. The framework grants access to NLP attributes of each particular term within the scope of a function filter. This design allows for a simple means of constructing custom filters that can operate on POS tags and other language characteristics without the need of manually extracting this information from the vocabulary.

Filtered versions of an embedding are stored under a single directory based on

the source embedding from which the filter originated. Indexing these directories using a unique hash that is generated from the filter specification enables the embedding component to load a filtered embedding using its hash identifier if it already exists.

3.1.4 Feature Provider Component

The feature provider is detached from the other components of the data module as it is responsible for generating data that is mostly conditional on the specific experiment being carried out, which limits its re-usability, unlike the dataset and embedding components. Aside from managing an indexed directory of this data, the feature provider is primarily responsible for producing both the feature data and number of auxiliary files. The purpose of the latter includes saving the state of data at different stages of the process, providing valuable information used when creating some of the output visualizations, as well as logging diagnostics on the process, for debugging and profiling purposes.

The steps that the feature provider takes to generate the required features are as follows: first, in the case of multiple datasets, these are merged and encompassing corpora are constructed for training and testing, second, text pre-processing and tokenization are carried out producing string tokens to be mapped to integer indices. A look-up-table is finally constructed from the vocabulary of the embedding and a TensorFlow graph of look-up operations is built to map these tokens to their respective indices within the embedding. Upon completion, the results of the graph are converted and exported into a proprietary TensorFlow Record (*TFRecord*) format to be consumed by the experiment module.

As inputs, this component requires an embedding object, one or more dataset objects, and the specific OOV policy that is to be adopted when constructing mapping tokens into features for the experiment module. The indices mapped to each token are contingent on whether OOV tokens are considered at all, solely during training, or whether a particular bucketing strategy is used. If OOV tokens

are ignored, this needs to be addressed at the tokenization stage by excluding these tokens. In the case where OOV tokens are only considered at the training stage, the process is similar, excluding only OOV tokens found in the test data, whereas when using a bucketing strategy, no tokens are excluded, and the mapping process is handled accordingly by the look-up table operation. Moreover, the mapped values for OOV tokens vary based on the OOV policy that is adopted, including the number of OOV buckets that are used, since this directly alters the hashing function that assigns the bucket to each token. All of these factors are taken into consideration and handled internally by the framework's own hashing mechanism, creating new data when required and re-using any computationally expensive data where possible.

Although trivially simple, the mapping process can become increasingly time-consuming for larger look-up tables, as the look-up ops would be required to traverse more data as a consequence. The initial approach to the problem was to optimize the python code responsible, through the use of generators and parallel processing libraries. These benefits from this approach were quickly exhausted however, and the approach proved to be unreliable, requiring fine-tuning based on the nature of the data being processed. To address this, the process was instead implemented using TensorFlow itself which is particularly adept at optimizing parallelizable operations by first expressing them in graph form, and subsequently executing handling resources and parallelizing operations wherever possible during graph execution, providing more consistent performance. Furthermore, the framework also generates a *filtered* vocabulary file containing only the tokens that occur in the datasets being used, and their corresponding index in the larger embedding vocabulary which is used for building a look-up table at a fraction of the size of the original embedding vocabulary, for the exclusive purpose of mapping the tokenized datasets being considered.

Merging Datasets

As previously alluded to, different datasets are characterized by the different attributes and vocabulary that they provide based on the limitations of the platform from which they are sourced, such as the character limit on Twitter, or a specific domain around which they center, such as reviews of a specific type of product. A mechanism by which these datasets could be merged allows us to capture a wider range of these characteristics in both training and testing data and evaluate the performance of different TSA approaches across these variances.

Text Pre-Processing

Spacy[3] was used for all text-processing and tokenization tasks. One of the primary reasons for using this tool was its native and intuitive support for parallelization. Secondly, its proprietary language model format enables selectively loading sub-modules based on run-time requirements, improving performance by avoiding superfluous processing.

Text normalization is kept to a minimum since the exploration of these techniques is beyond the scope of this work. First, for embeddings with case-insensitive vocabularies, all tokens are lower-cased accordingly. Furthermore, a default filter function is implemented which omits URLs and email addresses, and which is detached from the feature provider, facilitating extensibility by the end user.

Output Data

The most salient outputs of the feature provider are the *TFRecord* files containing the feature data for the experiment module. These features are exported to separate directories for training and testing data and further partitioned, following TensorFlow guidelines for ensuring integrity when batching and randomizing this data through their data pipeline API. Other important outputs of the feature provider are the vocabulary files that can be used to visualize the embedding using

[3]Public NLP library for python, accessible at https://spacy.io/

TensorBoard and a JSON reference file detailing the datasets and embeddings as well as OOV policy employed. The latter includes details on the OOV initialization function and a list of the OOV buckets and their constituent tokens, providing insights into the behavior of the bucketing function[4].

Finally, similar to other components, data at multiple stages are exported to pickle files for two reasons. Firstly, this foregoes the need for carrying out repetitive processing. Secondly, by loading these files at different stages of the feature generation process in an online setting, where computationally expensive tasks are kept to a minimum, the feature provider can maintain its internal state, preserving the value of all of its class members. Indeed, this is desired for all components; maintaining an identical internal state in both offline and online environments reduces the risk of costly online experiment failures resulting from run-time errors.

3.2 Experiment Module

At a high level, all operations that take place in the experiment module are governed by the experiment class, which simultaneously serves the purpose of maintaining an indexed directory of experiments while also preparing and dispatching commands to run those experiments. In this context, preparing an experiment implies setting up the directory for a new experiment, bridging the feature provider and model being trained and, loading any environment configuration settings for TensorFlow and scaffolding the experiment with the necessary code for tools that are included in the framework, such as comet-ml.[5]

The experiment class automatically organizes the experiment directory first by model name, and then either by the specific name for an experiment which is provided by the user, or using a name automatically generated from the experiment parameters provided. Enforcing this approach has two benefits, first, experiment directories are self-documenting since they convey the details of the experiment,

[4]The default is a basic string hashing function defined internally by TensorFlow
[5]For more information on this tool refer to §**??**

secondly, the framework will use the directory and continue training for an existing experiment with the same parameters by default.

Finally, since each experiment also serves as a self-contained model, the experiment class is also responsible for exporting trained models from an experiment. Once exported, trained models can be evaluated using real-world data. Using this data as a means of ensuring the validity of the process as a whole was the motivation behind implementing this functionality.

3.2.1 TSA Model Base Class

The Targeted-Sentiment-Analysis (TSA) Model base class serves as the abstraction layer between the end-user and the internal infrastructure of the framework. The implementation adheres to the principle of least privilege, exposing only the core constructs of a TSA model while handling the integration of that model with the rest of the frameworks' features internally. The core constructs for TSA model development are the default parameters of the model and the tensor operations it performs. A third, optional construct, covers any feature pre-processing where feature mutations can be specified that take place before invoking the model definition function. Exposing only these core constructs greatly simplifies development, enabling the user to focus on the model being developed. Moreover, this ensures that the rest of the framework components, particularly those concerning evaluation remain pristine and consistent across different model implementations.

The parameter definition function is the simplest of the three, only required to return a dictionary of default parameter values which may be overridden at runtime before being made available within the scope of the model definition function.

The default feature set is a dictionary of literal string tokens and their respective embedding IDs for target and their contexts (left and right separately). An optional pre-processing function can be specified to mutate the samples to prior to consumption by the model function so as to obtain the features desired by the end-user. This design choice was made to decouple the pre-processing logic from

the model definition, making it possible to inspect and test the process in isolation. Typical pre-processing operations may include concatenating the target to the left and right contexts.

The feature pre-processing function must return a dictionary of features, which the framework internally re-couples with the label data. For this reason, the pre-processing function may only mutate the structure of each sample, maintaining the length and order of the features as a whole, otherwise this could result in features being assigned incorrect labels. Token string data is required for certain downstream visualizations, such as attention heat-maps, where token IDs are not sufficiently informative. It is worth noting that the string literals are not automatically mutated, and maintaining integrity between IDs and string literals must be handled by the end-user.

Finally, before exposing these features to the model definition function, the framework automatically appends `*_emd` and `*_len` entries for each `*_ids` key on the dictionary, representing the embedded sequences and their length, respectively.

The model definition function defines the tensor operations to carry out on the feature data, producing logits for each sample. It exposes four parameters within its scope: a dictionary of features, their corresponding labels, a flag denoting whether the model is being trained, evaluated or running as a prediction service, and finally, a dictionary containing the resolved set of model parameters. Using the produced logits in conjunction with a loss function and optimization technique, which are also defined in this scope, the function must return a call to an in-built class method that generates the "estimator spec" object which encapsulates the model's behavior.

Adhering to the design specifications described herein, the framework is able to internally plug into the TSA model and handle the necessary code infrastructure adding functionality such as online collaboration, various downstream evaluation visualizations and introspection tools, automatic embedding of sequence data, support for Google cloud services and finally, a streamlined process for exporting the

trained model as a prediction service.

Certain features, specifically those centering around evaluation methods, such as attention heat-maps, are only applicable to a subset of model types. To address these situations efficiently an add-on mechanism that could isolate particular functionalities and automatically make them accessible to all TSA models was developed.

3.2.2 Model Add-On API

The principal motivation behind the add-on API is to enforce a design pattern where a TSA model implementation consists of code concerned solely with the definition of the model. Moreover, due to the importance of visualizations for evaluation purposes, it became apparent that this process would need to be streamlined to meet our objective of facilitating rapid experimentation. The add-on mechanism achieves this by allowing code concerned with specific functionality to be isolated, preventing repetitive code, reducing development time and facilitating testing. Furthermore, while the TSA model class provides out-of-the-box functionality, in some aspects it could be considered a black-box approach, limiting the end user. The add-on API aims to address this by providing an entry point for customization at a deeper level while still maintaining a layer of abstraction between the user and the framework's internals.

Each add-on functions as a pass-through for the estimator spec object generated by the model definition function, enabling the user to inject code before running the model. These commonly take the form of wrappers around hooks, a native Tensorflow construct. Hooks enable the execution of custom code at different points in the model life-cycle such as specific steps during training or evaluation. The applications for these hooks include producing custom visualizations, logging additional metrics and enabling techniques such as early stopping, to name a few. Within the scope of an add-on function, the user has access to all of the same variables available in the model definition function, including the model instance

itself, as well as the incoming estimator spec object, which must be returned, so as to either be passed on to the next add-on or execute the model. There are no requirements on the internals of the add-on or any custom hooks that the add-on wraps around, however, it must conform to a function signature to integrate with the framework. A number of examples of this signature are provided in the `addons.py` file along with custom hooks implemented specifically for the purpose of our work in the `hooks.py` file.

An addon decorator is provided which takes as input an array of one or more add-ons to attach to the model definition function it *decorates*. Add-ons run in the order they are attached, after the model definition function is called and an initial estimator spec object has been produced. Additionally, each add-on can be instructed to run only in a subset of scenarios, specifically: training, evaluation, and prediction. Finally, each add-on can be deactivated at runtime through when invoking a new task using the CLI. This is useful when submitting batch jobs to cloud service where there is no access to the codebase in between tasks and add-ons can only be deactivated programmatically between tasks.

4. Implementation

4.1 Importing Datasets

The framework provides a CLI command as an entry point for importing new datasets. The command signature requires a path to a directory containing three files at a minimum: a "train" and "test" file and an accompanying parser Python[1] script. This API additionally exposes arguments for specifying a custom identified under which to index the imported data, as well as a specific parser in situations where multiple parsers are provided.

The parsing script will typically consist of two stages: reading and mutating the raw data. The first is contingent on the format of the raw data being imported and can vary in complexity from simply reading from a text file to more complex operations on proprietary data formats. The second stage generally involves the use of generators and looping structures to iterate through raw samples and produce the arrays the framework expects. This stage also presents an opportunity for the user to incorporate any conditional filtering or transformations, such as excluding "conflict" labels from raw datasets. The function must produce three arrays consisting of the sentences, targets, and labels. A fourth array, of character offsets of each target, may also be returned. This is optional for datasets where each target appears only once in each sentence, but required otherwise, as the framework would

[1] Python3 syntax is expected

have no means of differentiating between multiple target occurrences in a sentence.

The signature that is enforced by the framework on the parsing function stipulates that it expects solely one parameter without a default value, which is reserved for the path of the file to be parsed. For most cases a single function will suffice, however in some instances, multiple functions may be required for improved separation of concerns. In these situations, both the parsing script and the function that serves as the primary entry point within it must be named according to the provided `parser-name` CLI argument.

Initially, the framework scans the directory for the relevant source files which contain the words "test" and "train" in their names. The parsing script is loaded based on the provided parser name argument or the default parser file-name. Once all required files are located, the framework inspects the parsing script and ensures that the parsing function has the expected signature. Provided all these tests are passed, the framework proceeds to run the parsing script on the raw data then stores the result in its own hashed index of datasets under the user-provided identifier for future use.

This mechanism for importing datasets provides the widest range for support of various data types while maximally extrapolating the code responsible for this operation, thereby simplifying the process for the end-user. Furthermore, the framework is abstracted from the dataset importing process thereby limiting the scope of errors that may occur during the process to the parsing script itself, which is easier to debug by the end-user as opposed to the internal code of the framework.

Finally, the structure of this approach allows for the possibility of defining multiple parsing functions, each of which may carry out processing or filtering operations prior to importing the dataset, while also delegating the responsibility of storing and accessing these different versions of a dataset to the framework.

4.2 Dataset Re-Distribution

The primary consideration when re-distributing a dataset is minimizing the number of original samples that are discarded to maintain as much of the size of the original dataset as possible without selecting duplicate samples from the source.

Algorithm 1: Dataset Re-Distribution

Input: Source Class Label Counts → **counts**

Input: Target Class Label Distribution → **targets**

Result: Redistributed Class Label Counts

`// ...argument validation`

1 **counts**′ ← **counts** ⊙ **targets**

2 **counts** ← (counts′ = 0) ? ∞ : counts

3 **while** min(*counts*) ≠ ∞ **do**

4 **smallest** ← (counts = min(*counts*)) ? counts : 0

5 **totals** ← (smallest ≠ 0) ? $\lfloor \frac{smallest}{target} \rfloor$: ∞

6 total ← min(*totals*)

7 **candidates** ← ⌊ total ⊙ **targets** ⌋

8 **counts** ← (candidates > counts) ? counts : ∞

9 **end**

10 **return candidates**

Ternary operations are applied element-wise

The first stage of the process validates the arguments provided, ensuring that a valid distribution is provided.[2] Since the limiting factors in this process are the smallest (most scarce) count numbers, an infinitely large marker must be used to remove elements from consideration. The algorithm(**??**) starts from the smallest count number as a reference point, from which candidate counts are calculated, and subsequently compared to the original number of samples for each class. If all candidates are less than the number of samples available, the redistribution is

[2]Omitted from the algorithm for brevity

considered valid and the algorithm terminates. Otherwise, the process is repeated using the class for which there are insufficient samples as the new reference until all candidate counts are valid.

One drawback of this approach is the substantial reduction in the magnitude of the original dataset that can occur in heavily skewed datasets that contain a small number of samples for some particular class. The amount of samples that will be discarded is directly proportional to the class that is the most scarce in the dataset and its corresponding target distribution in the new dataset.

4.3 Filtering Embeddings

The Spacy NLP library defines three different pipes that each are responsible for obtaining POS, NER and Dependency-Parsing tags respectively. A POS pipe, for example, assigns a *pos_* attribute to each token with its corresponding POS tag which can subsequently be used by the end-user as a filtering criterion. These piping operations can carry a heavy computational cost when running on exceedingly large embedding vocabularies. This motivated the development of a filtering process that is intuitively extendible by the end-user while adhering to the core tenant of the framework of maintaining as small a computational overhead as possible to facilitate rapid experimentation.

First, set filters are applied to reduce the size of the embedding, thereby reducing the amount of data each pipe would need to process. All the requested function-filters are loaded using a lookup registry which allows the user to specify a custom string identifier for a filter. The framework then encapsulates all function-filters into a single function and injects code to enable logging throughout the filtering process. Throughout the process, the framework deduces which pipes are required based on the collective set of filtering criteria specified by all filter-functions, and only runs these pipe operations to obtain the requisite NLP tags and filter tokens accordingly.

Partial functions are provided that conform to the signature that it expects from a function-filter to integrate these functions into a single function and deduce the corresponding pipes required. This makes the process of defining custom function-filters as simple as extending these partial functions, providing the attributes or tags the user intends to filter on. The framework expects an array of tags or attributes that are to be included by default, unless prefixed with a "!" character which conversely implies exclusion based on the same character or tag.

The sample code below illustrates the implementation of two filters, each requiring the POS pipe and therefore extending the `_pos_pipe_filter` partial. The first excludes all proper noun tokens whereas the second stipulates that only adjectives, adverbs, nouns and, verbs should be allowed.[3]

```
no_proper_nouns = partial(_pos_pipe_filter, tags=["!PROPN"])
pos_set_one = partial(_pos_pipe_filter, tags=["ADJ", "ADV", "NOUN", "VERB"])
```

During the filtering process, the framework tracks all tokens that are being filtered along with the corresponding user-provided filter-function. This information is tabulated and exported as a CSV file that is stored in the framework's hashed index of embeddings alongside the filtered embedding, for future reference. The primary motivation behind the filter report file is to enable the end-user to inspect the filtering process at a granular level, providing insight into the tokens being filtered along with the function-filter responsible.

4.3.1 OOV Policies

The OOV policy of an experiment describes the mechanism used to deal with OOV tokens during feature generation. This policy is controlled by two parameters, an `oov` flag, and a `num_oov_buckets` parameter, which is an integer that instructs the framework to use a bucketing strategy and how many buckets to use.

There are three valid configurations for these parameters described as follows,

[3]The complete annotation specification can be found at https://spacy.io/api/annotation

1. `oov` = `false`

   In this instance, all OOV tokens in both training and testing datasets are discarded.

2. `oov` = `true`, `num_oov_buckets` = 0

   When using no bucketing strategy, distinct vectors are initialized using a customizable initialization function for all OOV tokens in the training dataset. OOV tokens from the testing dataset are discarded to simulate real-world environments.

3. `oov` = `true`, `num_oov_buckets` $\geqslant$ 0

   All tokens, in both training and testing datasets are considered, with OOV tokens being assigned a bucket using a native Tensorflow hashing function. Similar to (2), vectors for each bucket are initialized using a customizable initialization function.

A list is maintained of which tokens to consider during tokenization, which controls which OOV tokens are discarded. Lookup operations during experimentation requires a vocabulary file which is generated with the tokens and their respective indices in the embedding, including any bucket entries. To speed up experimentation, a second, filtered, vocabulary file is generated containing only indices of tokens relevant to the experiment, thereby reducing the look-up operations' computation time.

This approach implies that any changes made to the `num_oov_buckets` or the `oov` flag will alter the content of the aforementioned files and require the process to be repeated. Changes to the OOV initialization function, however, do not, since vector initialization for OOV tokens and buckets takes place within the scope of the experiment module.

4.4 Datasets

The following section presents a brief overview of the datasets that were chosen to be integrated into the framework. We focus on the principal factors that were considered when choosing these datasets with the intention of maximizing the diversity in these attributes. These include the mechanism by which the datasets where originally obtained and annotated, the respective platform from where they were sourced, any specific domain around which they center and finally we present any salient statistical information that characterizes the datasets.

4.4.1 Dong Twitter Dataset

[?] is one of the most frequently cited datasets for TSA, used in works such as [?], [?] and [?]. This dataset comprises tweets focusing on a diverse range of topics that were obtained through the twitter streaming API by querying for targets such as "bill gates" and "xbox". This implies that, in each instance, the target string occurs explicitly in the tweet. The respective sentiment of each tweet towards its target is manually annotated as "positive", "neutral" or "negative", and the authors report a fair inter-annotator agreement score of 0.82. The final dataset consists of 6248 training samples and 692 testing samples, each having a distribution consisting of twice as many neutral samples as positive and negative combined. As some studies [?] [?] that followed point out, a notable property of this dataset is that each sample limited to a single target. It is also worth noting that due to the skewed nature of the class distribution, it is imperative that a metric that accounts for this is reported, such as the Macro-F1, when evaluating performance on this dataset.

4.4.2 Saeidi Yahoo Answers Dataset

In their work, [?] combine the tasks of detecting the existence of a particular aspect as well as the sentiment with respect to that aspect within pieces of text obtained from "Yahoo! Answers" concerning specific neighborhoods in or around

the city of London. The authors suggest that this text tends to be less constrained than that originating from social media platforms, and that this enables them to obtain reviews that mention multiple neighborhoods per instance with differing sentiments. The final dataset is limited to samples that contain up to two targets and these are manually annotated using binary labels ("positive" and "negative") across eleven separate aspects such as "dining" and "shopping". These aspects do not necessarily occur within the text itself and are therefore inconsequential within the scope of TSA. This notwithstanding, a "general" aspect category is also considered, used to denote a *general* sentiment towards a particular entity or target, and is the most frequently annotated category. We consider only this subset of samples from this dataset and only use the train and test portions, discarding the development portion. It is important to note that all original target mentions within this dataset are obfuscated through the use of "location markers" which eliminates any potential context information conveyed by the original target. The final datasets consists of 1770 sentences, 78% of which are labelled positive. To the best of our knowledge, the baselines implemented in this work have not previously been evaluated on this data.

4.4.3 Wang Political Twitter Dataset

A point that [**?**] stresses on in developing their dataset is the importance of considering an approach's ability to discern different and possibly opposing sentences towards multiple targets in the same phrase. With this same goal in mind, [**?**] build a dataset consisting of tweets centered around UK politics, labelled according to the sentiment expressed towards a specific target entity that occurs in the tweet, using a 3-point scale. Across 4077 tweets, a total of 12587 targets are identified with just under half (6015) of those expressing a negative sentiment and an average of 3.09 targets per sentence. As pointed out in [**?**], there are a number of samples in this dataset where the target locations are not specified, which makes tweets where the target occurs more than once unusable since there is no way of knowing

which instance is being considered. We exclude these cases for our experiments, resulting in a reduced dataset which contains 11899 target-sentiment pairs in total; with 2190 unique targets and an average of 2.94 targets per tweet.

4.4.4 SemEval Laptop & Restaurant Reviews Dataset

Another dataset commonly cited in the field of targeted sentiment analysis and aspect-based sentiment analysis is the Restaurant and Laptop Reviews dataset [**?**]. This dataset comprises of a total of 7686 sentences from two distinct domains, namely technology and dining, from which specific aspect terms and the respective polarity towards them were manually annotated. Sentiment polarity was determined on a 3 point scale with the addition of a "conflict" label which we do not consider in our experiments. Compared to data sourced from micro-text platforms the samples in this dataset are typically lengthier and more syntactically sound. The authors note some interesting differences in the nature of the reviews obtained from the two domains such as the restaurant subset containing substantially more aspect terms and a stronger bias towards the positive sentiment compared to the laptop reviews. The majority of targets in both domains consist of a single word. Recently, [**?**] used this dataset for their evaluation purposed and included results for three of the five baselines implemented in our work, namely, IAN [**?**], RAM [**?**] and TD-LSTM [**?**].

4.4.5 SemEval 2015 Twitter Dataset

This dataset was specifically proposed for the task of sentiment analysis on twitter as part of the SemEval challenge [**?**]. A subset of the dataset which we are interested in within the confines of this study consists of a series of tweets which were collected and manually annotated with a 3-scale sentiment polarity score directed towards a particular entity term occurring in the tweet. The annotation process was carried out using Amazon's Mechanical Turk system with 5 turks, employing a majority

vote in cases where an agreement could not be obtained. For the purposes of our study we make use of the training and testing subsets of the dataset and interchange these two subsets so as to have a larger training dataset as opposed to testing set, as in the previous datasets outlined. Moreover, in its original form this dataset consisted only of tweet IDs from this the original tweet can be obtained however only within a specific window. In out experiments we were able to obtain a downloaded version of this data, which included the original tweet text, from a later SemEval submission [**?**] from their github repository. The final dataset consists of 2872 samples, the majority of which are labelled "positive".

4.4.6 SemEval 2016 Twitter Dataset

Similar to the previously mentioned dataset, this was also constructed for the purposes of the SemEval sentiment analysis on twitter challenge [**?**], with the difference of being graded on a 5-point scale as opposed to a 3-point and also comprising substantially more samples and topics than the former. For the purposes of our work, we reduce the resolution of this dataset from a 5-point scale to a 3-point scale by grouping "very negative" and "weakly negative" samples under a single "negative" label, and similarly for "very positive" and "weakly positive", to obtain "positive" samples. The annotation process that was employed is similar to that outlined in the [**?**] with minor differences in the majority voting system for conflict resolution around label disagreements due to the different point-scale (the process is detailed in [**?**]). The actual tweet data for this dataset was also obtained from the Github repository of [**?**] due to the same limitation around this data expiring after a particular time-window. The test and train splits of the dataset were similarly interchanged to obtain a larger training dataset. The final dataset includes over 20000 training samples covering 100 topics, with the overwhelming majority being either "neutral" or "positive". Whereas the testing dataset consists of 6000 samples with entirely different topics and is skewed towards the "positive" label.

Dataset	Domain	Type	Split	Vocabulary Size	Unique Targets
Dong	General	Social Media	Test	3682	82
			Train	15037	113
			Total	**16047**	**118**
Saeidi	General	Review	Test	1231	2
			Train	1804	2
			Total	**2257**	**2**
Wang	Politics	Social Media	Test	4385	755
			Train	9706	1855
			Total	**11211**	**2190**
Pontiki - R	Restaurants	Review	Test	1945	520
			Train	3635	1168
			Total	**4292**	**1500**
Pontiki - L	Technology	Review	Test	1408	389
			Train	3174	945
			Total	**3510**	**1181**
Rosenthal	General	Social Media	Test	3243	44
			Train	9800	137
			Total	**11575**	**180**
Nakov	General	Social Media	Test	19374	60
			Train	43859	100
			Total	**54034**	**160**

Table 4.1: Dataset domains, vocabulary sizes, and the number of unique targets.

Dataset	Target Length (tokens)		
	1	**2**	**3+**
Dong	2080	4851	9
Saeidi	1770	0	0
Wang	9702	1937	260
Pontiki - R	4449	977	528
Pontiki - L	2143	982	410
Rosenthal	1554	1145	170
Nakov	11450	13684	1366

Table 4.2: Number of targets of different lengths, in tokens.

Dataset	Split	Samples	Negative	Neutral	Positive
Dong	Test	692	173	346	173
	Train	6248	1560	3127	1561
	Total	**6940**	**1733**	**3473**	**1734**
Saeidi	Test	588	139	-	449
	Train	1182	246	-	936
	Total	**1770**	**385**	**-**	**1385**
Wang	Test	2541	1206	957	378
	Train	9358	4377	3615	1366
	Total	**11899**	**5583**	**4572**	**1744**
Pontiki - R	Test	1357	258	279	820
	Train	4597	1128	926	2543
	Total	**5954**	**1386**	**1205**	**3363**
Pontiki - L	Test	743	153	218	372
	Train	2792	1013	633	1146
	Total	**3535**	**1166**	**851**	**1518**
Rosenthal	Test	486	56	288	142
	Train	2383	260	1256	867
	Total	**2869**	**316**	**1544**	**1009**
Nakov	Test	5868	749	1623	3496
	Train	20632	2339	10081	8212
	Total	**26500**	**3088**	**11704**	**11708**

Table 4.3: Dataset sizes and class distributions.

Dataset	Average Targets /Sentence	Sentence Length (tokens)		Distinct Sentiments /Sentence		
		Average	/Target	1	2	3+
Dong	1	21.1	21.1	6934	0	0
Saeidi	1.24	15.2	12.4	1416	23	0
Wang	2.94	26.3	9	2163	1640	242
Pontiki - R	2.31	21.4	7.4	2177	379	20
Pontiki - L	1.92	22.7	10	1665	193	9
Rosenthal	1.01	23	22.8	2842	5	0
Nakov	1.01	23.2	23.1	26355	40	0

Table 4.4: Dataset sentence information

4.5 Executing Tasks

A task represents the start of a new experiment or a continuation thereof and is invoked using the `tsaplay.task` submodule:

```
python -m tsaplay.task single \
--batch-size=25 \
--steps=1000 \
--model="lstm" \
--embedding="wiki-50[corpus,only_adjectives]" \
--datasets 'dong[33/34/33,15/70/15]' wang \
--contd-tag="example-experiment-task" \
--model-params hidden_units=50 oov=true num_oov_buckets=100 \
--aux-config logging=false debug=cli \
--run-config save_checkpoints_steps=1000 \
--comet-api="XXXXXXXXXXXXXXXX" \
--verbosity="INFO"
```

This section covers the CLI arguments available on this command when executing a single task.

- `--steps` The number of steps to train for.
- `--batch-size` The number of samples to use in each batch.
- `--model` The name of the model being run.
- `--embedding` The name of the embedding to use for the experiment. A comma-delimited list of filters may be included, as well as the `corpus` keyword to filter on tokens occurring in the datasets.
- `--datasets` One or more datasets specified by the name used to import them. Each dataset argument may be postfixed with a comma delimited list describing of redistribution percentages for training and testing. Each redistribution must have the same number of elements as the amount of unique classes in the dataset, separated using a `/` character. If only one set of values are provided this is applied to both training and testing datasets.

- `--model-parameters` Model hyperparameters which may override the default specified in its parameters function. The OOV policy parameters are also supplied through this parameter set.
- `--run-config` Values which are passed on to the Tensorflow run configuration. These include options such as setting a random seed for an experiment and how often to save model summaries during training.[4]
- `--aux-params` Auxiliary model parameters, separate from its hyperparameters, include programmatic switches for add-ons and a flag for enabling debug mode.
- `--contd-tag` An optional name to index the experiment under. If the name already exists, that experiment will continue training up to the provided `--steps` argument.
- `--comet-api` Comet.ml API key which enables real-time remote logging of the experiment (requires a `contd-tag` argument to be set).
- `--verbosity` Controls the logging level on `stdout` during execution.

[4]https://www.tensorflow.org/versions/r1.12/api_docs/python/tf/estimator/RunConfig

A batch option was also developed to allow multiple tasks to be run in sequence on the google cloud platform without the need to re-build and re-submit a job for each task. This command is also invoked on the same `tsaplay.task` submodule:

```
python -m tsaplay.task batch "batch_tasks.txt"
```

The batch-file syntax allows the user to specify multiple tasks using the single task CLI format where each task is delimited by a new-line character. Additionally, comments may be included in the file using either the '`#`' or '`;`' characters at the start of a new line.[5]

4.5.1 Google Cloud Support

Making use of a cloud computing service provides additional computational power through hardware such as GPUs which reduce experimentation times through increased parallel computing capabilities. Additionally, by running experiments in a remote environment, local machines can be used to continue development on future experiments. The Google cloud platform was chosen as it integrates well with the Tensorflow library, itself being a product of the same company. Moreover, this platform also provided us complimentary credits with which we could carry out our work.

While the framework facilitates integration with the Google cloud AI platform, some pre-requisites must be met on the host machine. These include setting up Google cloud CLI tools, as well as a Google cloud project and cloud storage bucket. The specifics of this process are beyond the scope of this work and can be reviewed in the relevant Google cloud API documentation.[6]

Since the framework must be packaged as a module before submitting a job, an external script must be used to programmatically invoke this process:

```
python submit_job.py \
--job-id="example_job" \
```

[5] An example batch file is included in the source code for reference.
[6] https://cloud.google.com/ml-engine/docs/

```
--job-dir="experiments_data" \
--job-labels type=example \
--machine-types masterType=standard_gpu workerType=large_model workerCount=1 \
--stream-logs \
--show-sdist \
--task-args batch "batch_tasks.txt"
```

- `--job-id` A unique string which identifies the job being submitted. This is also used as a job-dir if one is not provided.
- `--job-dir` The name of the parent directory which will house experiment data using the same structure described in §**??**
- `--job-labels` Optional labels which may be attached to a specific job for organizational purposes.
- `--machine-types` Used to specify the specific machine configuration for the job, as offered by the google cloud platform.[7]
- `--stream-logs` A flag which, when set, continues to stream log data to the stdout as the job runs.
- `--show-sdist` A flag which, when set, logs the build process that takes place before submitting a job to the stdout.
- `--task-args` The arguments that are forwarded to the `tsaplay.task` submodule which follows the same syntax described in §**??**.

The `submit_job.py` script is responsible for packaging the framework with the requisite assets then submitting the job request to the Google cloud platform. This process must be repeated for every job that is submitted. Tasks that fall within the scope of the data module are carried out on the host machine, before packaging the framework and requisite assets. In this way, the framework can minimize the size of

[7] https://cloud.google.com/ml-engine/docs/machine-types

the assets that need to be uploaded to the cloud while storing this data for future reuse. The internal assets directory is used to store the requisite embeddings, datasets, and features for a particular job. It is packaged with the rest of the `tsaplay` directory as a distributable python module and uploaded to the Google cloud platform. This module is subsequently installed on the platform to invoke the `tsaplay.task` submodule.

At the time of writing, task execution across multiple machines in a distributed environment is considered experimental as some features, such as comet.ml logging, are known to malfunction in this setting. This is likely a consequence of machine instances submitting data to the comet.ml service concurrently. Attempts to rectify this were made, however the issue was ultimately deferred as it was beyond the scope of this work.

5. Evaluation Methods

5.1 Tensorboard

Tensorboard is a development tool, included with the Tensorflow library, used primarily for visualizing the transient performance and behavior summaries of a model. A summary is a representation of a model at a particular checkpoint, encapsulating all the information about that model at that step in training. At each checkpoint, the model is evaluated on the test dataset consisting exclusively of unseen samples adding the respective metrics and visualization to the summary object of that checkpoint. The add-on mechanism wraps around hooks which allow custom metrics and visualizations to be added to the summary object.

The checkpoint frequency of an experiment is an adjustable run-configuration parameter. While more frequent checkpoints increase the granularity of the transient data for a model, this also incurs a cost on experimentation times, particularly when dealing with visualizations which may need to be generated multiple times. In certain situations however, such as employing early stopping, more frequent checkpoints may help minimize the risk for overfitting.

The Tensorboard service is started using a CLI command pointing to an experiment directory, stored locally or on a google cloud storage bucket, and is subsequently accessible on a local webserver.

5.1.1 Scalar Metrics

Model loss is plotted during training and evaluation to identify possible cases of overfitting. Although not an ideal metric, accuracy is recorded to compare results with papers that report it exclusively. Mean per-class accuracy is also monitored as it is more robust to class imbalances, however, unlike the macro-F1, it does take into consideration false positives.

5.2 Comet-ML Integration

The primary motivating factor behind integrating the comet-ml service into the framework is to facilitate collaborative work by making experiment details and results remotely accessible while speeding up development by automatically setting up the requisite code infrastructure for this service.

The top-level scope, encompassing all experiment data, organized by the name of the models used, is the comet-ml workspace. At a lower level, a dashboard for each model lists all the respective experiments and allows the user to define custom visualizations comparing metrics across multiple experiments. Finally, each experiment may be individually accessed to view all of the data that has been, or is being, uploaded.

The default behavior of the framework is to upload all images and scalar metrics, all CLI parameters used to start the experiment and, their values as they are resolved during runtime so as to ensure the experiment was run with the desired configuration. These parameters also serve as a useful point of reference when referring back to the experiment at a later date.

In addition to these, the framework also uploads the following resources for each experiment:

- Dataset distribution figure
- Model graph definition file

- A log of the `stdout` stream during execution
- The model source code
- All parameters used in the experiment
- Any embedding filter details and corresponding report
- Description of the host environment used to run the experiment
- A list of installed Python packages used for the experiment

5.2.1 Embedding Filter Report

The embedding filter report tabulates embedding tokens filtered as a result of a function-filter. The first three columns describe the function-filter responsible, while the latter two contain the token and its corresponding NLP properties.

The insights provided by the embedding filter report may serve to inform further refinement of a filtering scheme, for example, if a less stringent filtering approach might be beneficial, or if a stricter rule-set is feasible without sacrificing performance.

5.3 Visualizations

5.3.1 Model Graph

The model graph provides an intuitive way of visualizing the architecture and data flow of a model. Nodes are used to represent single operations or groups thereof, and connections represent the tensor data that flows from one operation to the next.[1]

[1] https://www.tensorflow.org/guide/graph_viz

5.3.2 Histograms

Histograms illustrate the evolution of tensor distributions with training. Internally, the framework attempts to produce histograms for all tensors identified as subject to training.[2]

5.3.3 Embedding Projections

When preparing feature data for an experiment, the framework also generates a tab-separated-value (tsv) file which lists all the tokens, including any OOV buckets, in the order they appear in the embedding. This file can be used in conjunction with the Tensorboard projector tool to attach labels to points in the embedding matrix projected in $\mathbb{R}^3$ space using principal component analysis (PCA). This is useful when embedding vectors are trained alongside the model and may provide additional insights into how the representation of OOV buckets evolves with respect to other tokens by visualizing neighboring points.

5.3.4 Dataset Distribution Figure

Two pie-charts are generated for the training and test datasets which depict the class distribution of the each dataset. Moreover, in the case where merged datasets are used, the contribution of each constituent dataset and their respective class distributions are also included in this figure.

This type of visualization intuitively highlights class imbalances as well as their sources while also efficiently conveying the contribution of each dataset relative to the others in a merged dataset.

[2]https://www.tensorflow.org/guide/tensorboard_histograms

5.3.5 Confusion Matrix

Matplotlib[3] was used to render most of the visualizations offered by the framework as it offers an extensive tool-set. The confusion matrix is generated using a custom-built hook and provides a broad overview of the performance of the model. It is rendered as a heat-map spread across all samples of the evaluation dataset. Although this information may be condensed into equally informative scalar metrics, such as the macro-F1, the confusion matrix may convey more insight into the per-class transient performance of the model. Ideally, with training, the heat-map should consolidate across the diagonal of the matrix.

5.3.6 Attention Heat-maps

Attention weight vectors are obtained using custom implementations of attention units which expose these data and their corresponding string tokens. These are used by the add-on to produce attention heat-maps for a select subset of samples from each batch during evaluation checkpoints. The size of this subset can be adjusted using the auxiliary configuration parameters to avoid excessive heat-maps from being generated for increasingly large datasets.

The attention heat-maps use a color-map to contrast the different weights being placed on each token. Over time, as the model learns to distinguish which tokens maximally contribute to the sentiment of a sentence, these tokens are expected to be highlighted relative to their surroundings which signifies a stronger attention weight. Two attention heat-maps are generated for each sample, the first illustrating the color-coded attention weight distribution corresponding to each string token whereas the second supplementing the first with the precise real-valued weights.

For models that employ multiple hops, such as memory networks, the process is repeated for each hop. These models are expected to concentrate their attention on the most salient tokens with each hop, as well as improving this process on the

[3]https://matplotlib.org/

whole ov er multiple evaluation checkpoints.

6. Results and Observations

7. Conclusion

7.1 Future Work

A. Appendix A

A.1 These are some details

```
this is some code;
```

Make sure to use this template.