Discrete Flag Xor (DFx)
Memory and Time Efficient Vector Search

by

Tobias Renwick

A thesis submitted in partial fulfillment of the requirements for the degree of

Master of Science

Department of Computing Science

University of Alberta

Abstract

Vector representations of text are heavily used in state of the art NLP tasks, however they require significant computing resources to build and use. In this work, we discuss and evaluate DFx, a system for representing vectors in a compressed form without compromising the similarity between those vectors, resulting in faster comparisons with less memory. Evaluation on a corpus of 1,000,000 tweets indicates that for the top 10 closest neighbors (KNN), dFX overlaps with cosine similarity for 95.4% of results while using as little as 12.5% of the original memory and completing the comparisons in 17.6% or less of the original time (assumes that the full size vector array fits in main memory). The KNN similarity rate decreases to only 90.2% when $K = 1000$. We further evaluate DFx on STS (Semantic Textual Similarity) benchmarks (STS-12 through 16) and demonstrate minimal performance loss (and occasional gains) due to discretization.

We also explore the notion of paraphrases built through a corpus wide similarity search. The generated paraphrases are evaluated on ROUGE, and BLEU for corpus similarity and for semantic similarity by using BERT-Score. The generated phrases achieve scoring as follows: BLEU 60.74%, ROUGE-1 67.91%, ROUGE-2 41.21% and BERT-Score F1 result of 77.92% on the WikiAnswers paraphrase corpus. We believe these results indicate that DFx can be used in practical applications and introduce computational gains without significantly compromising accuracy.

To my family

For letting me do these cool things everyday

Choose a job you love, and you will never have to work a day in your life.

– Confuscius

Acknowledgements

Before the beginning of my graduate studies, I did not think any of this would be possible for me. I love learning new ideas, and I love applying them in a meaningful way but in all honesty, I did not realize I had a chance to make a career out of learning. Fortunately for me, with the never-ending support of my lovely wife and family, I was allowed to pursue higher education. The following is not an exhaustive list, and in fact, I have not encountered a professor on this journey that was not encouraging of true engagement in learning.

I have found the department of Computing Science at the University of Alberta to be full of wonderful people who share my love for learning, and have introduced me to a passion for encouraging others to pursue novel ideas. It is on this note that I would like to acknowledge the support of my supervisor, Dr. Denilson Barbosa both for the knowledge he has given me and for the encouragement of true exploration and learning. I would also like to acknowledge Dr. Rick Szostak for encouraging me to explore ideas that have contributed to the field of knowledge organization, and I hope will help others to create future meaningful works. Dr. Paul Lu introduced me to the University, and through his Trellis group built my confidence as a researcher and inspired me to be a better academic. I also owe Spencer Killen, a colleague and excellent programmer, a large amount of thanks for his help in discussing the implementation of DFx.

I would also like to acknowledge two of my outstanding undergraduate professors at MacEwan University, Doctors Cameron MacDonnell and Nicholas Boers who made learning fun and engaging and opened the door to my graduate career.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

NLP (natural language processing) is a discipline centred on the idea of teaching computers how to read. If we imagine the cornerstones of human development, one of them is most certainly the ability to read and write. With the advent of the written word, every individual was no longer required to begin life with only the information provided by their caregivers, they could read and gain the expertise of distant or departed individuals, which gave them an important competitive edge. In the landscape of artificial intelligence, I believe that this is equally true — teaching computers to read will give them an understanding of the breadth of human knowledge, and current technology is advancing towards this goal.

In the current state of NLP, there is a fundamental challenge of attempting to represent the meaning of text using numerical input, which is generally referred to as embedding. An embedding can be thought of as the translation layer between human language and the binary code required by the computer. On its face it seems simple enough, we can turn any word into a unique number, but when we do that we lose the relations between them — their meaning. An example of this information loss is if we encode swim as `#A34465` and swimming as `#F12354` they lose all meaning of association, they are just fundamentally different values. Further to this, it is also challenging to represent homonyms as they map to the same number, and new words humans invent also need to be inserted as they are created. The goal of an embedding is to create a numerical representation that retains the meaning of the text — we want

swimming to be close to swim, but also diving, and maybe even butterfly (for the stroke, but preferably not the insect).

When humans read a text, we infer meaning to words quite fluidly and without much explicit information, as we relate them to our physical world and our prior experiences; this even applies to words we have never heard or used before. To illustrate this point I was walking with my colleagues one day in the quad and I said 'geeze it sure is smuggy today', which they were able to understand in context. If you (the reader) were a computer, you receive this phrase with solely the linguistic context in isolation from the physical context, which includes the temperature, the time of day, other entities, prior experiences and all 5 senses of a person, and it basically comes across as 'geeze it sure is `NEW_WORD` today', which could mean anything. I'm reasonably sure that most readers have by now developed some definition for 'smuggy', but it will likely help you define the word if I tell you it was a hot humid day, and there was a lot of smoke in the air from surrounding forest fires — 'smuggy' referred to the thick, smelly, moist air. The word 'smuggy' doesn't mean anything as far as I am aware, I could have replaced it with 'blick' and it might have gotten across the point, but I feel that 'blick' is a worse word to choose than 'smuggy' for that context, likely because of the close similarity to the word 'smog'. It is this inference of meaning through context that we need to teach computers, take a guess based on what you have experienced before and go with it until you know you are wrong.

An embedding, therefore, is a numerical definition of some chunk of text (character, token, word, phrase or larger) that is built up by reading text and accumulating statistics about that text. This initially includes words that happen frequently around a target word, so that their co-occurrence can be modelled, but it has grown to be so much more in recent transformer networks [45]. A transformer [45] is a newer application to word embeddings where the network effectively looks at all of the tokens in a fixed window and then assesses them all according to their surroundings, a notion of the linguistic context of the word. For now, this means that the surrounding

words can change the embedding of the current word and the current word also affects the surrounding words — the relationship of the embedding values is co-dependant.

Transformers, in modern NLP, are used to embed phrases or paragraphs as a whole to a single embedding. For example, we would expect the final embedding of the phrase 'I like what you said' to be quite close in latent space (the high dimensional space to which the embedding belongs) to another phrase 'I like your statement'. This closeness in latent space represents the semantic meaning of the phrases, the closer they are to each other, the more similar they are.

If we only consider a phrase to be a single embedding, however, we lose information about the words contained in that phrase, they are represented as one. For the sake of this work, we are interested in the journey through that space, how embeddings are created by a sequence of words. To examine this issue of moving through latent space, we conceptualize a phrase length transformer embedding as a path through latent space, where each constituent word in a phrase is a separate transformation from the previous embedding to a new one as the phrase grows. In Figure 1.1 the embeddings for all of the nodes are calculated separately with only the context of prior words, as a human would read. In other words, the embedding for 'I' is considered independently of 'I like', as the addition of the word like moves both the embeddings for 'I' and 'like'.

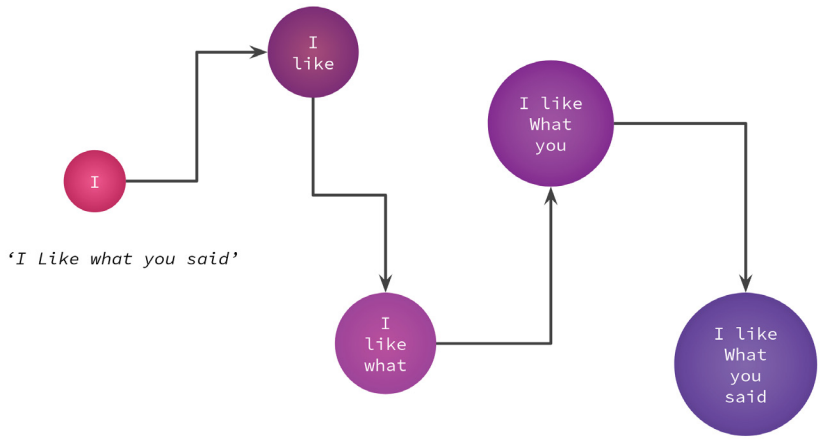


Figure 1.1: Vector paths in latent space

Concretely in Figure 1.1, we propose producing 5 distinct embeddings and retain the situational meaning of each word through its embedding. This retention of transformations between words allows us to examine portions of phrases independently, and consider their effect on the phrase embedding as a whole, as human writing in English would compose the phrase from left to right, word by word.

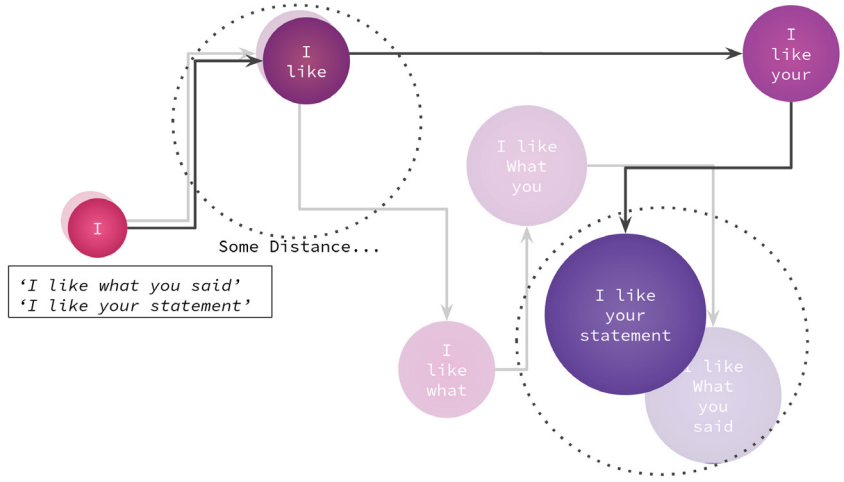


Figure 1.2: Vector paths collide in latent space

If we now compare the two phrases 'I like your statement' and 'I like what you said' we can see which parts of each statement are equivalent according to the network. In Figure 1.2 the two phrases are illustrated as they move in latent space, and wherever they approximately collide, they are considered

equivalent. This equivalency itself is interesting, as it represents a paraphrase — an alternate way of saying the same thing or more concretely *a different way to arrive at the same vector*. The central challenge of exploring this notion is one of computational complexity, we now need to store an embedding for each chunk of each phrase and with enough text, this is very costly.

DFx (the primary tool produced herein — Discrete Flag xor) is a vector-based search engine that is used to facilitate the exploration of a network's latent space. DFx was developed to combat the resource allocation required to reasonably explore the embeddings of an entire corpus of text. DFx empowers the user to rapidly query a corpus of text for similarity according to the embeddings created by any given network. Herein DFx is limited to testing and evaluation on a text corpus, however, there is no reason it could not be applied to any embedding — it is agnostic to the network used.

Once DFx has ingested a text corpus, the user can enter queries and DFx returns the embeddings that are closest to that point in latent space. By exploring a network in such a fashion, one can examine the separation of negations: e.g. are 'can' and 'cannot' close? — the separation of word senses — e.g. bank as a financial institution vs bank as a riverbank — throughout an entire corpus of unlabelled text, by expanding the query neighbourhood until the two phrases share neighbours in latent space. If for example, we start with the phrases 'I went to the bank to deposit money' and 'I went for a swim on the river bank' we can expand the query neighbourhood until the two notions overlap, this represents the number of interfering phrases in the corpus are between the two candidates. This measurement can allow you to look into a densely packed vector space and extract knowledge about the phrases.

The applications of a tool such as DFx are numerous, however herein I explore the generation of paraphrases, not through network inference, but by considering different embeddings that connect the same space as equivalent through substitution as in Figure 1.2. In this scenario, DFx is asked to repeatedly search a corpus of text for chunks of phrases that are semantically equivalent — chunks that collide, connecting two separate portions of latent

space. This illustrates the importance of efficiency for DFx, as a moderately sized text corpus can contain many millions of chunks.

1.1 Proof of Concept

The initial proof of concept for the work can be well illustrated with the following two phrases 'I went to the park with my dog' and 'I took my dog to the park'. Using the Universal Sentence Encoder [10], a transformer network created primarily for the task of embedding sentences, we generate Table 1.1 wherein we compare each phrase, word by word using cosine similarity, a measure of the angle between two high dimensional vectors. As Table 1.1 shows, USE is capable of identifying paraphrases as when the cosine similarity is high, the sentences are very close to semantic equivalency.

Phrase A	**Phrase B**	**Cosine Similarity**
I	I	**1.000**
I went	I took	0.787
I went to	I took my	0.745
I went to the	I took my dog	0.543
I went to the park	I took my dog to	0.749
I went to the park with	I took my dog to the	0.726
I went to the park with my	I took my dog to the park	0.817
I went to the park with my dog	I took my dog to the park	**0.978**

Table 1.1: The initial proof of concept example

The task herein is to take an unlabelled corpus of text and first embed it with USE [10] and ingest it with DFx and then use the search to locate the embeddings that represent paraphrases of a given query phrase. As the paraphrased chunks are located, the portions of the query text that are used are then replaced with the search results to create potentially novel paraphrases (though the text fragments are completely contained in the corpus, the combination is potentially novel). For example, if 'Phrase A' from Table 1.1 was present in the corpus, and the algorithm is given the phrase 'I took my dog to the park to play frisbee', an expected paraphrase would be 'I went to the park

with my dog to play frisbee', where 'Phrase B' was substituted with 'Phrase A'.

DFx is used to power this search method, which involves evaluating a vector for every unique sequence of chunks in a corpus, as in Table 1.1.

1.2 Contributions

The main contribution of this work is to design and evaluate DFx, a method of vector compression useful for storing and retrieving embeddings that maintains a notion of similarity for large-scale comparison. While other methods perform this task, DFx allows a true O(n) search to be completed on large arrays of vectors in a reasonable time with common hardware, meaning that there is no chance of missing a close neighbour in the search.

To validate DFx, I offer comparisons of the method across various tasks requiring the maintenance of semantic similarity. First I show that DFx's compression method maintains a similar level of semantic similarity as the raw vector by comparing on STS (Semantic Textual Similarity) benchmarks. Second I show that the increased speed of brute force comparisons can be used to power a paraphrase generation algorithm that competes with state-of-the-art neural systems.

Code for DFx is made available at github.com/qubies/DFx

Chapter 2

Background

This chapter introduces some background concepts that are essential to the work presented. It begins with an introduction to the foundational concepts of vector embeddings as they pertain to text, and how embeddings relate to tokenization. A factor to be aware of is that the concepts of embedding and tokenization are intertwined to a large extent as the embedding is the representation of the token and the methodology used for tokenization reciprocally affects the embeddings.

In this work, I primarily discuss how to embed tokens in such a fashion as to show how they are related to one another, i.e. their semantic similarity. To this end this section discusses how embeddings can be used to compare two tokens (or groups of tokens) of text for similarity measures — i.e we want to measure how alike these tokens are in terms of meaning.

The chapter then outlines some of the common neural methods (Word2Vec [35], Doc2Vec[28], RNNs[39], and LSTMs[25]) used to learn embeddings of text and their noted strengths, closing on transformer embeddings [10], [12], [45] which make up the current state-of-the-art (SOTA).

2.1 Embeddings

An embedding is a learned vector representation of text created to maintain the semantic relationship between different tokens, where that vector is an array of real numbers ($\vec{v} \in \mathbb{R}$). Embeddings are used throughout NLP as a

way to represent variable-length text (and some level of meaning) in a fixed-size latent space, the size of which is controlled by the dimensionality of the vector used as the embedding. Each dimension of the embedding vector is ideally trained to represent some concept or feature associated with the text which could be ideas like plurality, gender, part of speech etc. or it could also represent classes of objects such as 'is a fruit', 'is a vehicle' etc. The fixed-length representation is required for many different algorithms and the method with which language is transformed into the latent representation (many are discussed in further sections) has great bearing on the output. An embedding matrix E is then an array of vector embeddings such that they represent all of the tokens in text — the vocabulary of the embeddings.

The larger the dimensionality of the embedding, the more features it can represent, however, larger vectors are computationally expensive to maintain and also more difficult to resolve due to 'the curse of dimensionality' [7].

'The curse of dimensionality' is a phrase introduced by Bellman [7] used to define the idea that too many dimensions harm many metrics (particularly nearest neighbour search). The problem arises that as dimensionality increases, the distances between them all become approximately equal. An extreme example of why this phenomenon arises is as follows: given that each dimension represents a distinct feature, and each token also represents a distinct concept, if there are sufficient dimensions for each token to be represented by a unique dimension (the size of the vocabulary is equal to the dimensionality) then each token is a one-hot representation of itself (a vector with a single dimension set to 1.0 and all others set to 0), which has no additional meaning encoded, and all tokens are different from each other by the same amount in the embedding space. In a typical embedding scenario, the dimensionality of that embedding is a balance between too small to not represent enough distinct features, and too large as to enter the curse of dimensionality.

Creating an embedding is such a common, general-purpose task in AI that its meaning is somewhat blurry across the body of work in academia, but is generally used to reduce an input to a vector. Herein, to create an em-

bedding is the process of turning any text token (character, subword, word, n-gram, sentence, paragraph etc.) into a fixed-length vector, which is typically normalized to provide a common ground for comparison.

2.2 Tokenization

In NLP, the process of tokenization is the act of assembling text into a subsection of 'tokens'. A token can be anything from a single character to a group of characters (subword or n-gram), a word, or a group of words (n-gram). The level of tokenization is an important consideration, and while many initial algorithms [22], [35] rely on words, modern methods [12], [13], [34] use subword tokenization.

Tokenization methods affect embedding methods as the style of tokenization affects what meaning the embedding is trying to capture. For example, if the token is a single character, then the embedding matrix will be smaller than if the tokens are words (as there are fewer characters than words in English), however, more of the ability to extract meaning will be shifted to the network used with the embeddings, as there is not as much semantic content in a character as there is in a word.

On one hand, word-level tokenization, with an embedding for each word allows a more natural understanding of what the embeddings mean, but they hamper generalization, as plurality and verb conjugations are different surface forms of the same root. To work around these differences, techniques such as stemming and lemmatization were introduced to reduce the word to a single common form, at the expense of the loss of information about the plurality or the tense, among other linguistic processes.

Subword tokenization, on the other hand, allows a word to be broken into groupings of characters based on their occurrence in the text. The idea is that a word might get split into a root and a conjugation, so instead of 'run', 'runs' and 'running' being separate words, they might get split into ['run'], ['run','s'], ['run', 'ning'].

2.2.1 Word2Vec

The most significant embedding method of which I am aware is Word2Vec [35]. Word2Vec is significant because it is simple and effective, but mostly because it helped make main-stream the idea of transfer learning.

Word2Vec is based around the principle that 'You shall know a word by the company it keeps' introduced by Firth [16] in 1958, which can be summarized as the meaning of any given word can be determined by the words used in context around it. This process is natural as when we (as humans) encounter an unknown word, we can often decipher a likely meaning from the context in which it is used. There are a lot of factors that are built into what a human develops as context when they read, their individual history and exposure to language play a role, but also the text that occurred earlier in the current document, and even text that occurs in an entirely different document can affect the interpretation.

Figure 2.1: A word (red) and its context (blue) with a window size of 2

In Word2Vec, a word is embedded through the use of either a skip-gram or bag of words task, which is analogous to deciphering a word from its context. In the bag of words task, we provide to the algorithm a set of words from a window around both sides of a target word and attempt to maximize the probability that we choose the target word. In Figure 2.1 the bag of words task is to predict red given blue. For the skip-gram task we do the opposite, from red in Figure 2.1, predict blue.

Over a large volume of text, Word2Vec [35] creates word embeddings that model the co-occurrence of words and their contexts through a fixed-length vector by applying either the skip-gram or bag of words task to a set of word and context embeddings that are initially randomly created. The embedding

of a given word is selected, and the dot product of that embedding is taken across the entire context vocabulary, where the greatest value is taken as a prediction, through a sigmoid function that is used to normalize the output. The error is then used to update the embeddings attempting to maximize the probability of that word being the correct one.

For this work, the important part to understand is that this process (hopefully) creates neighbourhoods of words in latent space, where words that share a similar context, share a similar position in that space. If those words are often used interchangeably, then the context will be very similar, and the resulting embeddings will also be similar. From this similarity, the word vectors can now be related through mathematical operations with some intuitive implications.

With a Word2Vec embedding, one can attempt to determine the relationships between words through the addition and subtraction of the embeddings. If we subtract one embedding from another we effectively remove their shared context, and if we add two embeddings together we add that context. This idea leads to the seemingly natural understanding that the embedding of the capital city 'Paris' with the removal of the embedding of the country 'France' removes the association to words that were associated with the country, but not those associated with the capital, which can then have other capitals added to it to approximate the embedding for that country, e.g. +'Rome' = 'Italy' [15]. This phenomenon is of great interest, as despite not being trained to specifically model this behaviour, the embeddings have 'learned' something about the words and their greater meaning through context.

Word2Vec is a good example of transfer learning in language, it is trained on a general task(s) so that downstream tasks can make use of its prior learning. This enables organizations with access to larger computing resources to create and package embeddings for use and extension in other applications.

2.2.2 Vector Comparison

One of the goals of creating embeddings is to allow them to be compared to one another. In a continuation of the understanding of the relationships between vectors in latent space, many works have targeted the cosine similarity or dot product of language vectors to relate similar words by examining their neighbourhoods in latent space [22]. In this context, both dot product and cosine similarity are thought of as a measure of how close two vectors are to being in the same neighbourhood — or how far apart they are. On one hand, the dot product measures how similar two vectors are while allowing the length (or magnitude) to play a role in the measurement [20]. On the other hand, cosine similarity measures only the angle between two vectors, irrespective of magnitude. Dot product and cosine similarity are indistinguishable if the vectors are normalized, however, if magnitude plays a role, longer vectors generally have a greater similarity to all other vectors when using the dot product [20].

2.2.3 Multi-Word Embeddings

Word2Vec [35] was a very influential work, and its use has propagated to many applications, however, when reading, people consider more than a single word at a time to generate semantic meaning. To add document context to the equation, Doc2Vec was created, where a paragraph vector was concatenated to the word vector to provide a direct memory of past words [28]. Doc2Vec treats the paragraph embedding as another word, but one that is shared for the entire paragraph.

Word2Vec, and by extension Doc2Vec, consider context during training, but because the target is a single word or document embedding, the context is lost — all words that are spelt the same, are treated as the same, and all documents that contain the same words are also treated as the same since they ignore ordering. Order can be a key factor in semantics, take for example the word ‘bank’ in the sentence “ I went to the bank on the river bank ” in Word2Vec both banks here would share an embedding. This lack of word sense

disambiguation (WSD) can be tied back to the lack of attention to word order and contextual application.

2.2.4 RNNs

As we have identified the concept of word order and memory to be fairly central to language understanding, the recursive neural network is another way to create embeddings. An RNN [39] is a fundamentally different method than discussed thus far in that it allows consideration of a sequence of words as a whole. In a general form, an RNN used for language has an initial embedding layer, which may be pre-trained (Word2Vec) using the power of transfer learning, or may be created from scratch, but in either case, each word is transformed to an initial embedding. Unlike Word2Vec, this initial embedding is not intended to be used as such, rather for each token in a multi-word input, that embedding is now one of the network's inputs. An embedding built from an RNN is capable of being sensitive to word order, unlike Word2Vec, as the state created for each word is allowed to consider the words used up to that point (Figure 2.2).

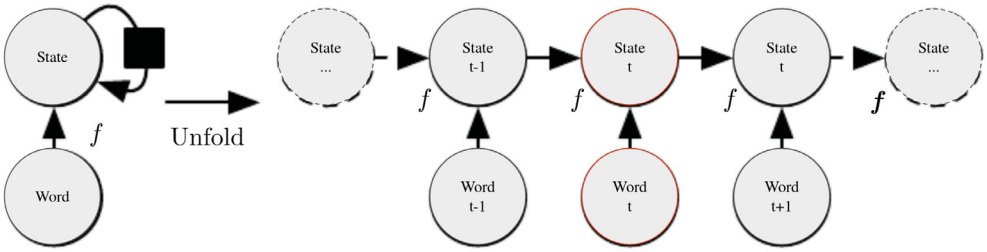


Figure 2.2: An unrolled RNN
adapted from Goodfellow, Bengio, and Courville [19]

It is important to note now as well that networks are not limited to the consideration of words. As discussed in Section 2.2 we can break language into any different level of token we choose. From this point on I will substitute the word 'word' with 'token' to signify this variability.

In an RNN, each input token (*t*) is mixed with the output of the prior timestep (*t-1*) which is in turn mixed with the token before it (*t-2*) recursively

to create the current state (Figure 2.2). In this fashion, an RNN is quite representative of how humans read, token by token in order, building a 'state' that allows the network to bring in information from the past. An RNN can be used to embed any length of phrase by continuing to feed in the input words, and then eventually extracting the hidden states from the network as an embedding. The issue that arises with an RNN is that it attempts to keep all information in its state, and since the state is a fixed size vector, it rapidly becomes overwhelmed and loses information.

Further, the longer the sentence the more significant the 'vanishing gradient effect' [24]. The essence of back-propagation (the method by which many neural networks learn from their mistakes) is that the nodes that were responsible for an output error are corrected for their hand in it. As an RNN essentially adds a layer for each token, more tokens mean more nodes to blame, and the gradient becomes vanishingly small as it is spread over those nodes. This makes it very hard for an RNN to learn the long-term dependencies of words.

2.2.5 LSTMs

To address the memory capacity issues of RNNs, the LSTM [25] presents a series of gates (input, forget, output) that allow the network to selectively forget or ignore portions of both the current input and of the prior state (Figure 2.3). These gates insulate the memory component of the RNN from the state with a series of sigmoid functions.

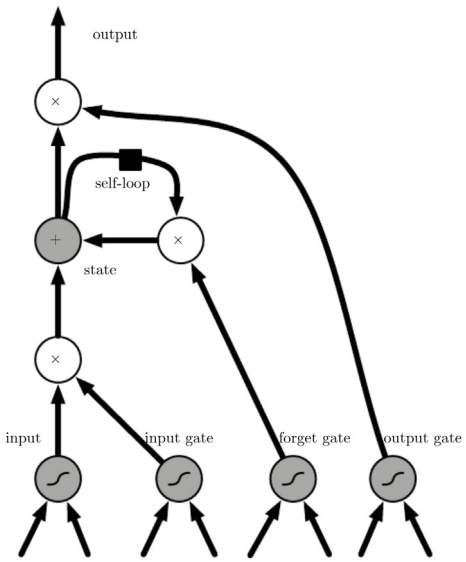


Figure 2.3: An LSTM cell [19]

The reason for using a sigmoid is that it strongly activates as either 1 (remember/use this dimension) or 0 (forget this dimension). The LSTM adapts the sigmoid to each input so that it can react to, and therefore learn to handle different inputs. Further these sigmoid functions, in combination with the additive nature of the network help to prevent the gradient from vanishing — the network can usually blame something for its mistakes.

2.2.6 Transformers

Transformer networks [45] have brought a large amount of change to the language processing landscape, and they currently represent the majority of the SOTA models. The key advantage that transformers have over LSTMs is that they are not burdened with maintaining a state, instead, the network looks at all of the input tokens (to a maximum length) at the same time, and uses an attention function to regulate which inputs are important for which other tokens. This attention function is derived from 3 sets of weights that create a query, key and value from an input embedding for a given token.

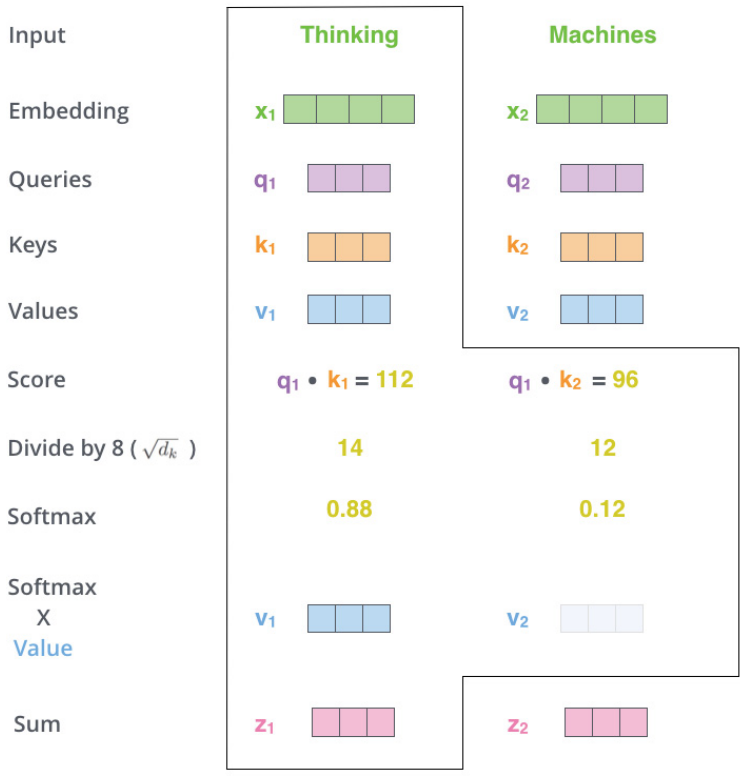


Figure 2.4: The self attention mechanism in a transformer. Image credit to Alammar [3]

The attention function (Figure 2.4) first uses the newly built query vector for a target word and computes the dot product with the key vector of every other word in the input, this produces a series of scalars, which is thought of as the attention score for the token [3]. The softmax is then taken of the scalars to normalize them and highlight the more important tokens — the ones we should pay attention to. Each scalar here is now multiplied by the value vector (created in the same way as the query and key) which produces the final embedding for that token. The scalar scores are very similar to the sigmoid of the LSTM, wherein a 0 allows the output from that token to ignore the effect of other tokens, and a 1 allows it to pay attention to only that token.

While the mechanism of attention has a similar action to the gates of the LSTM (zeroing out values deemed forgettable or unimportant), Transformer networks are faster to train for the same amount of parameters than an RNN as

they have no dependencies between each state, i.e. all tokens can be calculated on their own, at the same time. This makes the transformer architecture highly parallelizable, and as such its limits are scalable with more hardware. This computational flexibility has resulted in networks with a very large number of parameters, trained by huge computing resources.

The initial breakthroughs in transformers were brought into the mainstream by BERT [12], a transformer network with 340 million parameters in its 'large' model (figure 2.5). Microsoft recently announced Turin-NLG, a massive 17+ billion parameter transformer network, which lays claim to SOTA in many tasks but cannot be trained on common hardware [34]. Microsoft states that any network with more than 1.3 billion parameters cannot fit on a single GPU (even with 32GB memory) [34].

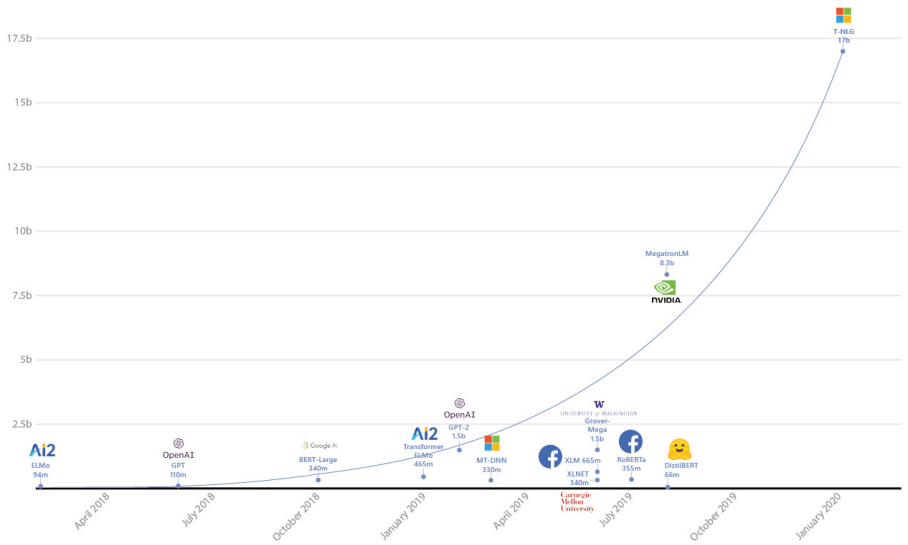


Figure 2.5: The growing size of transformer networks [34]

Most recently, OpenAI has announced GPT-3, a 175 billion parameter transformer-based language model that possesses a very captivating ability to generate text, answer questions and do simple arithmetic [9]. The ever-increasing size of networks produces interesting results, however, the amount of computing required to train and even run inference on them is out of the reach of most academics, which could restrict future advances to large corporations.

Despite their training resource requirements, transformers have greatly increased the power of many transfer learning applications by creating token order sensitive embeddings of token sequences, often considered sentence or paragraph embeddings [12]. The primary difference between a transformer embedding and a word embedding in this context is that the phrase 'philosophy of history' has a different embedding than 'history of philosophy' [38], [42]. This importance of order allows for a larger chance at Word Sense Disambiguation (WSD) [13], as the resultant sentence vector is dependant on both the constituent tokens of the context and their order. As a result, embeddings extracted from Transformer networks are no longer a simple affair, as each sentence has to be passed through the network, and has a likely unique output.

Chapter 3

Related Work

There are 3 principal components at use in this work: vector compression, vector comparison, and paraphrase creation. DFx performs the comparison and compression, while the paraphrase generation is made possible by the former. While paraphrasing is only one application of search in this domain, it is an interesting demonstration of the capabilities of DFx.

3.1 Vector Compression

The first issue in storing a large number of vectors is the size of that storage. Vectors are typically represented as an array of 32bit floats, therefore the storage requirement for each vector is $32\ bits \times number\ of\ dimensions$. The level of precision provided by a 32-bit float is rarely required for machine learning applications, therefore one method of compression is to reduce the precision of the floating-point representation to either 16 or 8-bit floats or an 8-bit int (weight quantization) [21], [32]. In these applications, the quantized weights are used inside the model itself rather than in the representation it creates, and can both be learned through training (quantization aware) and applied after training (dynamic & static quantization) [37]. Not surprisingly, these methods are applied in large-scale machine learning deployments to speed up inference and reduce storage costs [32], and are being incorporated into GPU frameworks to offer mixed-precision computations [23]. DFx tries to be agnostic from the initial implementation and is based on a more general method of

vector compression wherein it operates after training on the representations created, and has no interaction with the model itself. I refer to this generically as vector compression, irrespective of where or when it is applied, the principle is the same.

There are two main methods for vector compression: binary embeddings and vector quantization, which also give rise to hybrid methods that use both techniques [8], [17], [18], [40], [46].

3.1.1 Binary Embeddings

Binary embedding methods concentrate on turning a vector into a representation with reduced dimensionality that maintains a low hamming distance for similar vectors [17], [18], [46]. The idea is that such a binary embedding can then also be rapidly compared using the popcount instruction (Population count) which counts the number of 1s in a binary representation.

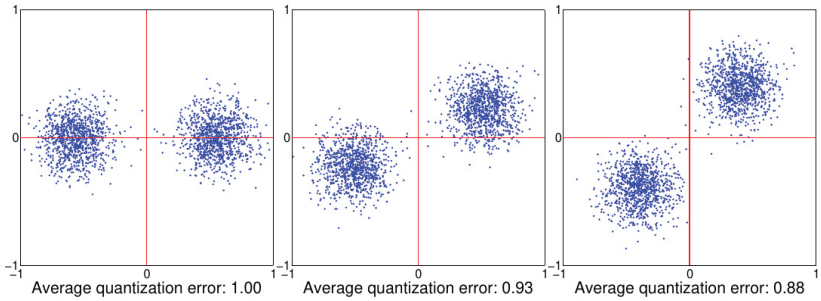
Introduced by Salakhutdinov and Hinton [40] as one of the first methods of binary embedding — semantic hashing is the process of using a neural net to produce a binary code rather than a vector. The work completed by Salakhutdinov and Hinton [40] is largely before, or on the cusp of the deep learning revolution, and before the modern word embedding and transfer learning, therefore the binary encoding is their embedding method. A graphical model is used to generate the binary encoding from what is essentially word counts of documents which are fed through an auto-encoder stack of Restricted Boltzmann Machines (RBM), fine-tuned by gradient search [40]. In this setting the RBM is used as an encoder, trained without ground-truth (unsupervised), just to avoid reconstruction loss of the word count from the embedding. As an important note, this task is a document-level embedding rather than a sentence or word level task.

Modern embeddings are surprisingly similar, however, rather than using a word count as input, modern networks typically use an embedding layer for each token in a vocabulary, and further they can account for word order and unseen words through various methods. DFx is an extra layer to any

vector embedding and does not require specific training, however, in future work I would like to explore the notion of creating the binary embedding from the encoding stage to see if it improves performance or allows dimensional reduction and comparable performance.

PCA (Principal Component Analysis) [26] is a frequent first step in binary encoding methods [18]. The goals of PCA are to reduce a given matrix to its principle components, keeping only the important information, thereby reducing the size of the representation [1].

PCA is completed by first condensing dimensions that have a high co-occurrence, as that information is considered redundant [1]. From the co-occurrence matrix, PCA considers the variance across a vector $v_j^2 = \sum_{i,j}^{|i|} x_{i,j}^2$ to be the inertia of the vector from which is subtracted the mean of the vector, i.e. the amount the vector changes [1]. Then, through linear combinations of the original variables, new vectors that represent the maximal variance of that space — its inertia — are constructed as the principle components [1]. This process continues for each dimension, leaving out the prior dimensions which had greater inertia, resulting in a series of factor scores that represent a projection of each value to the principle [1]. PCA, by its nature, does not directly reduce dimensionality, however, it loads the importance to the top — the vector with the greatest inertia. To end with reduced dimensionality, the less important components are omitted from the final projection — i.e. the components with the smallest variance are not considered. Further, as noted by Gong, Lazebnik, Gordo, *et al.* using the same number of bits to encode these higher value dimensions is likely to result in loss — termed quantization error. To address this issue, Weiss, Torralba, and Fergus [46] attempt to assign more bits to more important dimensions, while Gong, Lazebnik, Gordo, *et al.* [18] develop a method they dub iterative quantization, which attempts to minimize quantization loss by optimizing the rotation of a binary hypercube to quantize each point to a vertex of that cube (Figure 3.1).



(a) PCA aligned. (b) Random Rotation. (c) Optimized Rotation.

Figure 3.1: Taken from [18], this figure illustrates the rotation of the binary hypercube to suit the clusterings created by PCA

DFx performs a simplistic decomposition, where each vector is considered equal and has a centroid that is $\frac{\max - \min}{2}$. The idea behind this is that the important parts of the vectors for an embedding are the variance of groups, not the variances between them. As an extreme example, a normalized vector that consists of an equal number of ones and zeroes has high inertia, but a low interest from a learning perspective, this vector is effectively binary. DFx assumes that an insignificant value is middling in a min-max sense. In the extreme case presented above (all 0s and 1s) DFx will not allocate any more or less importance to that vector, it encodes them as what is effectively high and low.

3.1.2 Vector Quantization

Vector quantization attempts to learn a mapping of the entire vector field and reduce vectors to a similar representation by k-means (or other clustering methods) [8]. An approximate nearest neighbour is then returned by comparing a candidate to the 'codebook' of possible vectors generated by the clustering. In general, this means that vector quantization does not need to lower dimensionality, as each cluster membership can have the same dimensionality, however it saves resources by checking only the representative from the cluster, rather than each member. The result of this is that there is a tradeoff between the number of clusters (the size of the codebook) and efficiency — the larger the codebook the better the result, but the more expensive it is to compute.

Additionally, vector quantization has a lesser ability to extrapolate to unseen data, as if a cluster is not represented in the input set, it cannot be properly represented in the quantized output.

Jégou, Douze, and Schmid [27] introduce their method of product quantization, which attempts to augment k-means by splitting the vector into subvectors and then applying a separate quantizer on each subvector. The resultant clustering/search method is quite computationally expensive; therefore Jégou, Douze, and Schmid [27] implement a coarse quantizer (effectively k-means), and a fine quantizer (product quantization), wherein the fine quantizer is learned from results of the coarse quantizer, and on search, an 'inverted file' is produced according to the coarse quantizer then the results are further processed with the fine quantizer. Further, for each result, it is now possible for the nearest neighbour to not be in the inverted file specified by the coarse quantizer, therefore, multiple coarse indices are checked.

The fundamental challenge with vector quantization is to create a representative field without overburdening future search initiatives. While DFx is most similar to a binary embedding, it uses a simple discretization across dimensions, which could be replaced by a product quantization [27] like approach — clustering across each dimension to find relevant clusters.

3.1.3 Paraphrasing

Bannard and Callison-Burch [4] suggest that paraphrases can help create a more fluent and varied generated text. The central idea is that you could generate text multiple ways by expanding with paraphrases, and then either select the best or provide them all. Bannard and Callison-Burch [4] also suggests that paraphrasing can help in evaluating translations as there may be many ways of saying the same thing — BERT-score [47] can be seen as an attempt at exactly this.

In early work, paraphrases were drawn from monolingual parallel corpora — phrases that were translated more than once into a target language often contained different translations of the same idea [5], [6]. Bannard and

Callison-Burch [4] consider a foreign language as a pivot, where an input English sentence is aligned to the translated version, then other instances of that same foreign language phrase are located, and their English translations are returned. This method attempts to make use of the translation errors that accumulate with every translation to create paraphrase examples (Figure 3.2).

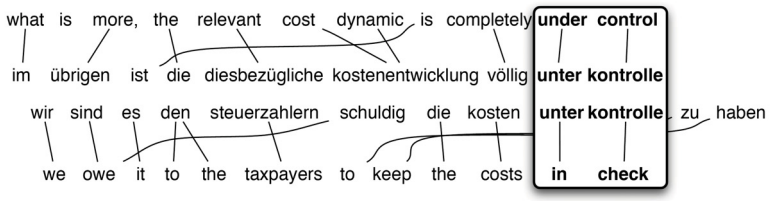


Figure 3.2: a German language paraphrase for 'under control' [4]

A paraphrase is then calculated as the sum of the probabilities that the translation model can create from the initial phrase to the foreign language then back again — i.e. the sum of all of the probabilities of the translations of the initial phrase being translated back into the new paraphrase.

For evaluation, Bannard and Callison-Burch [4] substitute the candidate paraphrases into 10 sentences that contained the initial phrase, which was then judged by 2 human annotators.

One of the challenges with foreign language paraphrasing methods is the creation of the corpus, i.e. how to match the foreign and target phrases in their respective texts. This task is called alignment and is difficult as it must be performed twice in bilingual paraphrasing methods, once from the input phrase to the foreign language, then back again to the input language. The issue at play is how best to align phrases, since ordering in many languages is different, and even if it is similar it can still be challenging to determine which words are part of the target phrase. In Figure 3.2, the alignment between the examples is straight, however other alignments in the phrase (shown with lines connecting them) are mismatched. If for example, the phrase was 'completely under control' this maps to 'vollig unter kontrolle', however, 'completely' has no connection in the second phrase. It is easy to see that the longer the phrase becomes, the more difficult alignment is. Bannard and Callison-Burch [4]

conclude that using manual alignments is best, and automatic alignments are improved by the addition of a language model to score the resultant phrases due to frequent grammar errors.

The paraphrasing with DFx employed here is, in a sense, an automated alignment task, and has even fully adopted the idea of using a language model to score the candidates generated, due to frequent grammatical errors. The difference is that DFx's paraphrasing method does not compute an alignment per-say, it checks a subset of possible alignments as split around the verbs in the phrase over a large corpus. Because DFx operates as a scalable search, there is also no need for a translation, the corpus is exhaustively searched for comparable phrasings. This last point is a benefit, also highlighted by Liu, Mou, Meng, *et al.* [30].

In their more recent and related work, Liu, Mou, Meng, *et al.* [30] propose Unsupervised Paraphrasing by Simulated Annealing (UPSA), a search-based method for the creation of paraphrases. Provided with a target phrase, UPSA attempts to alter it via a stochastic search algorithm driven by an objective function to alter the text by word replacement, insertion and deletion. The objective function attempts to measure the semantic similarity, diversity, and linguistic fluency of candidate phrases. Each candidate that is proposed by UPSA is scored, with the temperature controlling the range of acceptable deviation below the current best phrase — this temperature control is what prevents UPSA from getting stuck and returning either uninteresting paraphrases or very similar phrases to the original text. As the search progresses, the temperature is cooled, and UPSA is allowed to converge.

Both UPSA and DFx are unsupervised in the sense that neither requires a training set of example paraphrases to be used in this fashion, rather they search for valid replacements by using the text that they have seen before. UPSA employs stochastic mechanisms to find potential replacements, then scores them, while DFx exhaustively searches its input corpus for similar phrases and replaces them.

Chapter 4

Problem Definition

DFx attempts to address the problem of semantic phrase similarity search through vector embeddings created by a transformer network. The problem with exploring this latent space is fundamentally an issue of scale. The space occupied by a normalized 512 dimension vector embedding is difficult to fully conceptualize but the full space occupies a number of bits equal to 32^{512} different combinations (disregarding specific float implementation factors). To more effectively explore the space, DFx compresses vectors through discretization and accelerates comparison computations.

To demonstrate the capability of DFx, we use it to generate paraphrases from a corpus of text by searching a latent space populated with a pre-embedded corpus.

4.1 Vector Search

DFx is primarily addressing the problem of vector search, in the context of a vector that represents an embedding. The goal of DFx is given an unlabelled corpus of text, produce an indexed database of embeddings for that text, such that the embeddings maintain their semantic relatedness in terms of the search while reducing the resource consumption so that the entire corpus can be contained in memory. In other words, DFx aims to maintain an approximation of cosine similarity between any two embeddings while reducing their resource

allocations:

$$DFx(\vec{e}_1, \vec{e}_2) \sim \frac{\vec{e}_1 \cdot \vec{e}_2}{||\vec{e}_1|| \times ||\vec{e}_2||} \tag{4.1}$$

The purpose of this operation is to allow time and memory efficient searches against the corpus to locate text, not by keyword, but by vector similarity.

4.2 Paraphrasing

The problem of paraphrasing text for phrase expansion is one where we desire to predict and use alternate forms of a given phrase that have the same (or approximately the same) resultant meaning. This process can be thought of as a phrase expansion, where a single meaning is mapped to multiple phrases. Phrase expansion could be useful for expanding existing datasets to increase the generalizability of results, or it could be used as a writing tool that suggests alternative wordings. The ability to paraphrase represents the potential for a more abstract notion of meaning in a phrase.

In this work I propose that a sentence, or paragraph, can be viewed as not a single vector embedding, but rather a path through the vector space, where each additional word applies a transformation to the existing vector, resulting in the matrix E. This transformation t is viewed as the movement of the meaning of the sentence S as it grows by words (w).

$$\boldsymbol{E} = t(w_1 ... w_n) \forall w, n \in S, |S| \tag{4.2}$$

The task of paraphrasing is reduced to finding alternate paths between a start and an end vector. This means effectively altering the rows of $\boldsymbol{E}$ by switching them with other similar embeddings that already exist in a given corpus to maintain the property that the final embedding of the initial phrase ($\boldsymbol{E}_n$) is approximately equal to the final embedding of the generated phrase ($\boldsymbol{E'}_m$):

$$\boldsymbol{E}_{n,*} \approx \boldsymbol{E'}_{m,*} \tag{4.3}$$

In DFx, candidate replacements are generated via the vector search and may contain a different number of words than the initial phrase (n is not

necessarily equal to m).

4.3 Evaluation Methods

4.3.1 Vector Similarity

To demonstrate the maintenence of similarity, a comparison of DFx against cosine similarity is completed for up to 1000 results, over 1,000,000 vector embeddings. The goal of this test is to have a significant overlap with cosine similarity, particularly when the number of results is lower, i.e top 10.

As a secondary measure of comparison, the similarity score of DFx vectors is compared to cosine similarity using Facebook's SentEval framework [11], and the STS-12 through STS-16 benchmarks. Starting in Sem-Eval 2012 and running until 2016, the STS benchmarks are a series of tasks which attempt to measure the semantic relatedness of machine generated vectors. STS tasks examine the correlation between various sentence pairs and test that the measurements rise an fall with human judgements on the same pairs. In this benchmark, we compare DFx's discretized vector representations to standard vector representations and expect to show minimal degradation in performance, despite the decrease in resource consumption.

4.3.2 Paraphrases

A significant issue in the task at hand is the evaluation of a successful paraphrase. Paraphrases, by their very nature, may bear little or no resemblance to the initial text, though they maintain the spirit of the statement. This property has the disadvantage that a correct answer cannot be predetermined reasonably, as there are likely many correct answers, which are all equally correct. Therefore, a test corpus cannot contain all possible paraphrases, or even all the most likely paraphrases, meaning that any evaluation short of human judgement is lacking.

Difficulties aside, some methods have been created to allow the automatic evaluation of paraphrases, namely ROUGE [29], BLEU [36] and Bert-Score

[47]. ROUGE and BLUE are n-gram based models that test for identical words used in a candidate paraphrase and a list of references. ROUGE focuses on recall, while BLEU focuses on precision, and adds a penalty for word omissions. Either ROUGE or BLEU can be run on many different levels of n-gram (consecutive words considered).

BERT-score [47] attempts to use embeddings from Google's BERT transformer encoder to detect similarity as a primary measure. Unlike ROUGE and BLEU, BERT-score is not as n-gram sensitive, as it matches each token to its most likely embedding both from the candidate to the reference and in reverse.

Chapter 5

Implementation

5.1 Code Infrastructure

5.1.1 Overview

At the core of this work is DFx (Discrete Flag Xor), named for a method of discretization which maintains the approximate relationships between vectors, while decreasing their resource requirements.

DFx is composed of a vector search engine tied to a datastore which allows for retrieval of phrases in detail at a low cost. The search engine and storage implementations are carried out in the Rust programming language. Rust was chosen because of its modern emphasis on memory safety without garbage collection, resulting in high performance and high-quality end products.

To make the library approachable and useable, DFx uses a Python front-end API, which is tied to a multithreaded Rust-based back end for performance. The Python front-end provides the user with the flexibility to define their embedding in Python with the efficiency of Rust. As a result, the user can choose any of the many embedding methods available in Python for text (BERT [12], Distilbert [41], USE [10], RoBERTa [31] etc.) to embed the input data, however, the user is required to provide a file that contains the maximum and minimum spans for each dimension of that embedding. The front-end Python API provides various hyperparameter and helper options to fine-tune storage and search options — these are listed in Figure 5.1.

DFx has two modes of operation, which are mutually exclusive: `add mode`

```
-h, --help            show this help message and exit
--add_json_file ADD_JSON_FILE
                      add a json file with line by line objects
--paraphrase_json_file PARAPHRASE_JSON_FILE
                      create a new json file with paraphrases of this
                          one. The input is a json file with line by
                          line objects
--generate_max_min_json_file GENERATE_MAX_MIN_JSON_FILE
                      create a max_min from a json file, the input is
                          a json file with line by line objects
--json_text_field JSON_TEXT_FIELD
                      the json field id as a string
--batch_size BATCH_SIZE
                      The size of a batch in the file input processor.
--add_phrase ADD_PHRASE
                      a phrase to add to DFx
--num_buckets NUM_BUCKETS
                      The number of buckets in the search server
                          store. DO NOT CHANGE after initial creation.
--search_limit SEARCH_LIMIT
                      The number of results to return with each search
--language_model_weight LANGUAGE_MODEL_WEIGHT
                      The amount to count the input of the language
                          model (higher is more)
--grammar_weight GRAMMAR_WEIGHT
                      The amount to count the input of the grammar
                          checker (higher is more)
```

Figure 5.1: DFx Python API help documentation

and `query mode` — the transition between the modes is not controlled directly, rather it is done according to the CLI options entered (Figure 5.1) where any add operation is processed before any query operation. Add mode is used to add entries to the datastore, while query mode is used to search that datastore. For this section, python methods are shown in conjunction with their back-end counterparts to illustrate typing requirements (Rust is strongly typed).

5.2 Component Layout

DFx consists of 3 main components that operate partially separately: The python front-end, the Rust back-end, and the PostgreSQL datastore. The python front-end chooses the encoding, the rust back-end performs encoding and compression, and PostgreSQL provides the storage.

In all of the tests for this work, the datastore operates from a docker container, as are the front-end and back-end. The front and back communicate via the PyO3 FFI inside the same container, while the database operates off of a socket-based connection pool (R2D2) in a separate container.

Hardware used either consists of the author's laptop — A Dell xps13 9370 - 12 cores/16GB ram/M.2 SSD (laptop) — or a cloud VM hosted on Amazon Web Services — 8 cores, 32 GB ram, and an SSD for storage (cloud).

5.3 The API

5.3.1 Instantiation

```
#Python
dfx = DFx.Vector_Search_Server(NUM_BUCKETS, SEARCH_LIMIT,
    "add")

#Rust
fn new(num_buckets: usize, search_limit: usize, mode:
    String) -> Self
```

A DFx instance is created using two essential hyper paramters: `NUM_BUCKETS` — the number of buckets to split each dimension — and `SEARCH_LIMIT` —

the number of results to return per query. These functions return a vector_search_server object, which loads all existing data from the Postgres store. `SEARCH_LIMIT` may be altered after DB creation, however, altering `NUM_BUCKETS` causes the datastore to be invalidated as the encodings need to be recalculated. Since DFx is a lossy compression, there is currently no way to patch the datastore without recalculating the vectors. As an operational consideration, it would be a future improvement to store the raw vectors in the DFx datastore so that they could be re-calculated.

5.3.2 Add Mode

```
#Rust
Add(&mut self, phrases: Vec<String>, embeddings:
    Vec<Vec<f32>>, phrase_ids: Vec<i32>)
#Python
dfx.add(phrases, embeddings, ids)
```

In add mode, DFx first discretizes the vector produced by the input network (USE [10] in this work), compressing it into an integer based sequence which aligns the hamming distance of the input vectors with their approximate similarity (full details of the discretization are in Section 5.4). This is done to reduce memory consumption and to collapse near-identical vectors into one another.

In add mode, each input is cached in a hash map to speed up ingestion and avoid duplication of work, however this conflicts with the demands of query mode in terms of memory usage and retrieval efficiency, but typically allows O(1) lookups of existing data by a vector.

For an add operation, the discretized vectors are pushed to a PostgreSQL datastore along with their phrases via a connection pool (Figure 5.2). Each add and discretization operation is sourced to a worker (number of workers is the number of cores), however, in practice, all workers wait for the datastore on add operations. This is because additions are synchronized around unique discretizations, therefore the store needs to be kept consistent as it is updated. While the add speed is slower than simply calculating the embeddings, it is a

single step that is not repeated.

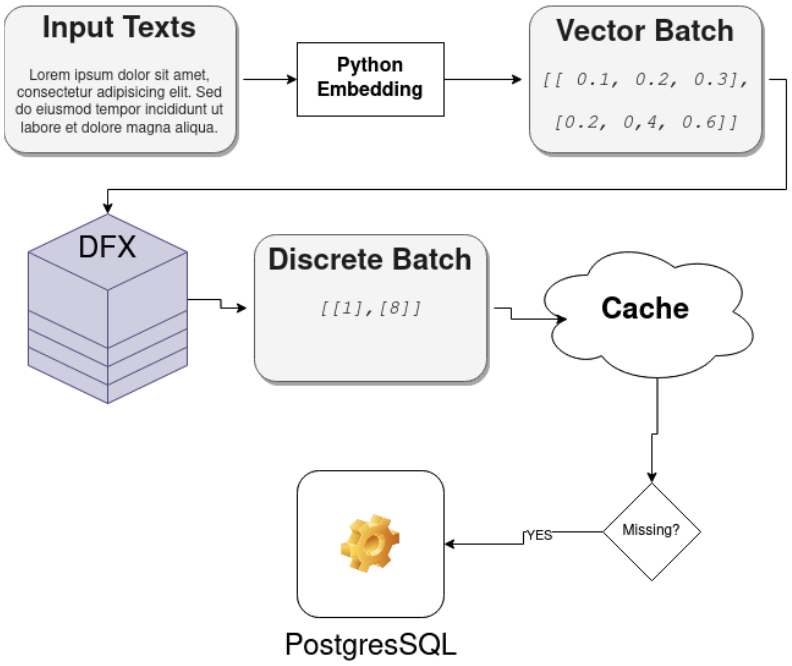


Figure 5.2: DFx Initial Text Processing Path

5.3.3 Query Mode

In query mode, DFx is designed to optimize an O(N) operation (a brute force search on all vectors), therefore the data is moved into a contiguous memory array to allow faster iteration. In query mode, additions are not allowed as the append operations may cause expensive reallocations. The modal nature is solely for this reason, as the performance objectives of DFx surround optimizing the speed of the query processor, rather than the speed of ingestion. On instantiation, all discretized vectors are loaded into memory in this contiguous array.

When a query string is received, the query is embedded and then discretized (Figure 5.3). DFx then performs a comparison across the array by slicing it to each available core, with each core returning the k lowest score discretized vectors — in DFx a lower score is a higher similarity as the score represents the divergence between the discretized vectors. The individual top k scores are collected for each core, sorted and returned as the query result as a list of

discretized vectors.

To turn the discretized vectors back into language, DFx stores the phrases in an external database, which are retrieved by querying the database for phrases represented by those K vectors. This query can return more than K candidate phrases, as vectors that collide are grouped.

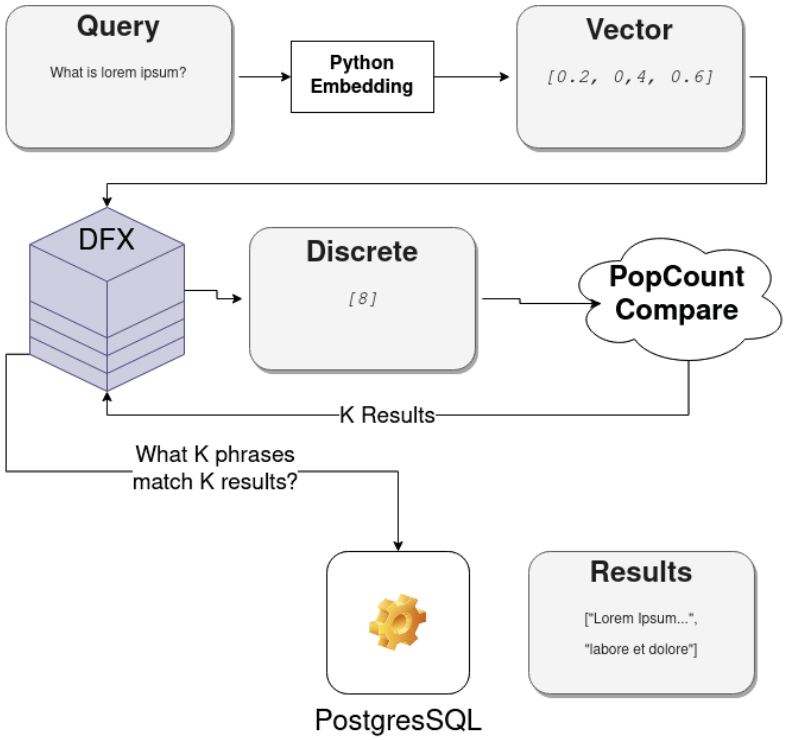


Figure 5.3: DFx Query Path

5.3.4 Datastore

The design works with a PostgreSQL database as a backing store, which was initially configured with the schema in figure 5.4. The prototyping of the datastore led to some interesting conclusions about the default PostgreSQL configuration.

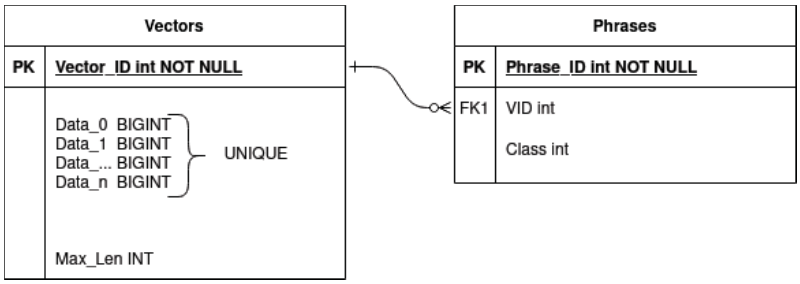


Figure 5.4: The first database schema

Indices

In the initial configuration, indices were added to the Data_n fields, with the intention that only unique vectors could be allowed. This worked in early tests that used only a small number of buckets (most initial tests were completed with 3 buckets) as this consumes fewer resources and is faster, however as the testing was scaled up the number of buckets was increased, this schema was not viable. By default PostgreSQL only allows 32 columns to be included in an index, therefore DFx could only function up to 5 buckets — 32 columns of u64 is 2048 bits of storage, spread over 5 buckets uses exactly 2048 bits of storage.

Aside from the limitation of the index column count, the keeping and construction of the index are problematic as well. The default indexing scheme in PostgreSQL is the B-tree because in a typical scenario both lookup and insertion are $O(log(n))$. PostgreSQL offers many methods for maintaining indices, but the one worthy of discussion here is the hash index. Since DFx's optimizations are all centred around query speed, the hash index is the ultimate choice. Once a hash index is built, the query time is expected to be $O(1)$, however during building a hash index may take extra time to rebuild when it encounters too many collisions. This is one of the reasons that Hash is not the default index, as insertions may be randomly slowed down to reorganize the index. The other reason that Hash indices may be inferior is that they do not sort items, so if a lookup needs to cover a range of values, hash indexes face difficulty as each lookup may be in a different block, therefore there is no benefit from prefetching. Neither of the drawbacks is significant for DFx, therefore

a significant speedup (approx $4x$ on insertions) was achieved by converting to Hash indices. The final schema is detailed in Figure 5.5.

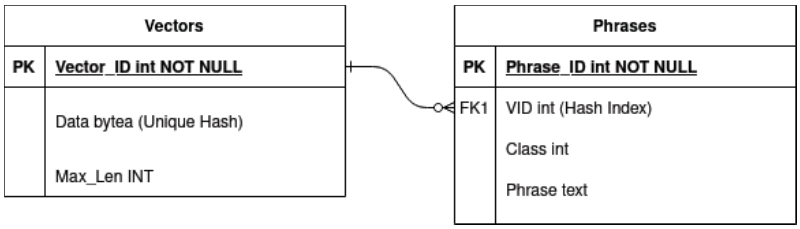


Figure 5.5: The final database schema

Hash indices in PostgreSql only support a single column key, however, so to gain that advantage, the vectors had to be serialized to a binary format to allow their storage in a single field wherein the 64-bit uints are concatenated together in a `bytea`.

5.4 Discretization

One of the core contributions of DFx is the discretization function. The goal of the discretization function is to reduce the size of each vector by allowing only a discrete number of choices for each dimension. In this work, this limit is called the bucket count. The larger the bucket count the more variablility in each dimension, however the more resources the algorithm consumes.

The discretization function in this work attempts to take a continuous function — the vector embedding $f(x)$ — and turn it into a discrete value set $d(f(x))$, where d is applied element-wise and k is typically small and represents the number of discrete buckets for each dimension in a vector produced by the input network ($f(x) = \vec{v}$).

$$d(\vec{v}_i) \to y;\ y \in \mathbb{Z},\ 0 \leq y < k \tag{5.1}$$

The basic concept of DFx is maintenance of near outliers, the assumption is that what makes a vector special are its distinct values, not those that it shares with other embeddings. To this end, DFx uses a bucket type discretization

which relies on the maximum and minimum values for each dimension. Knowing the exact values for maximum and minimum is not required (and likely not desirable), we only strive to identify which values are likely outliers.

Given the max and min, the bucket centroids are calculated separately for each max/min pair:

$$bucket\_centroids = \{min + (\frac{max - min}{2k}) + (\frac{max - min}{k}) * x\}_{x=0}^{k-1} \quad (5.2)$$

In this work, the max and min values were computed by embedding 1,000,000 phrases in the target set and taking the max and min for each dimension of those embeddings. The resultant discretization is then determined as the closest centroid to the initial value with ties going to the lower centroid (Fig 5.6).

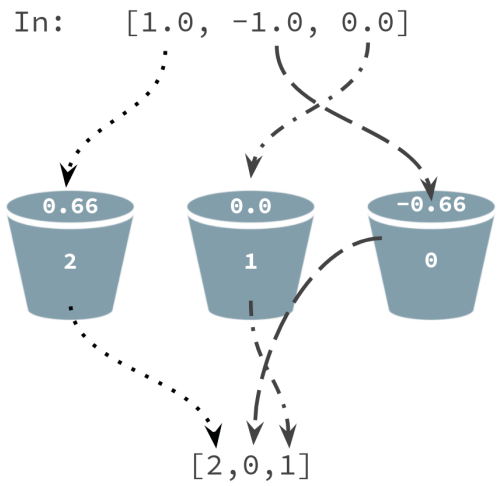


Figure 5.6: here a 3 bucket discretization is presented where each bucket has a max of 1.0 and a min of -1.0. The centroids are labelled on the 'opening' of the buckets. The closest value to each input is selected as the representative value

5.4.1 Encoding

The result of discretization is $B = \{\mathbb{Z}_{<\texttt{NUM\_BUCKETS}}\}$, which represents the bucket each dimension was assigned to. In DFx, each member of B is encoded as a binary flag format where 0 is `b0`, 1 is `b1`, 2 is `b11` and so on, where each 1 is a flag that represents the bucket number assigned to that value. The number

of bits required to represent a vector is $dim * (\texttt{NUM\_BUCKETS} - 1)$. In DFx, a 512d vector with 3 buckets is compressed from 16,384 bits ($512d \times 32bit$) to 1024 bits — a savings of 16x without considering overheads.

Figure 5.7 illustrates how the memory consumption of DFx scales with the bucket count relative to a more standard floating-point representation.

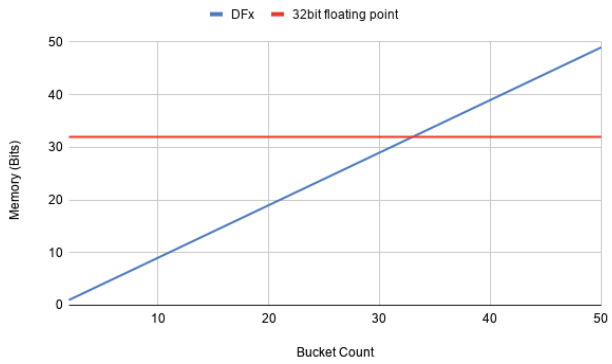


Figure 5.7: Comparison of the number of buckets per dimension to a 32-bit floating-point value.

5.4.2 Comparison

A comparison in DFx is then computed by taking the xor of the 2 discretized, encoded vectors and summing the 1s in that value, where the 1s represent that the vectors differ at that point (a higher score is a lower match). For example with `NUM_BUCKETS` set to 5, a minimal value is encoded as `b0000`, while a maximal value is encoded as `b1111`. The xor of these two points is `b1111`, therefore the distance is 4. If we were to compare two middling values, say 2(`b0001`) and 3(`b0011`) our distance is only 1 (Table 5.1 for full examples).

DFx's comparison is computationally simple, and also offers an additional speed advantage (Figure 5.8) in most bucket counts over both cosine similarity and dot product as implemented in the ndarray package in Rust — ndarray. To arrive at a fair comparison, DFx's compare method was examined using the Rust toolchain's benchmark utility, which performs many iterations of the method until a stable measurement is obtained. Figure 5.8 outlines the relative performance of the comparisons on a single 512dim vector randomly created.

	0,0 b0000	**0,1** b0001	**0,2** b0011	**1,0** b0100	**2,0** b1100	**2,1** b1101	**2,2** b1111
0,0	0	1	2	1	2	3	4
0,1	1	0	1	2	3	2	3
0,2	2	1	0	3	4	3	2
1,0	1	2	3	0	1	2	3
2,0	2	3	4	1	0	1	2
2,1	3	2	3	2	1	0	1
2,2	4	3	2	3	2	1	0

Table 5.1: The distance returned from DFx comparison for various discrete representations of a 2 dimensional vector using 3 buckets as in figure 5.6

In many instruction sets, there is a special low-level instruction named 'popcount', short for population count, it is specifically designed to optimize the counting of 1s in a binary representation [44]. Popcount is often the target of binary encoding schema, simply because it is faster than the standard methods of bit-shifting. Comparisons with different bucket counts with and without popcount are displayed in Figure 5.8.

Note that when popcount is enabled, a DFx compare has a minimum timing of 5ns per iteration at 2 buckets, compared to 68 ns for dot product and 210 for cosine similarity — a reduction of 93% vs Dot Product. At the default bucket count of 9, DFx obtains a benchmark of 25ns, a 63% reduction.

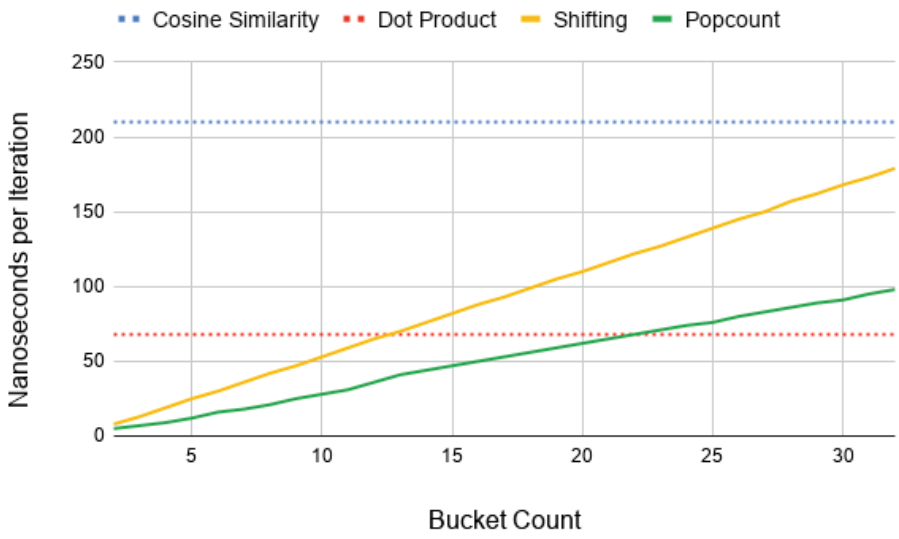


Figure 5.8: DFx compare across bucket settings with and without popcount optimizations
*completed on laptop

The reason that DFx's compare and discretization functions were designed and selected is that they are not an approximation of the search space, but an approximation at the level of each vector. A search in DFx is truly a search over all of the values, and has no chance of missing a comparison that is considered close by DFx. Other methods of searching a large vector space either rely on clever segmentation of the search space (vector quantization, PCA, LSH), or require a specially trained vector embedding. The tradeoff of segmentation methods is that they run the risk of not returning the best matches as the segments may occur in less than optimal locations. The tradeoff of binary embedding methods is that the training phase needs to be redone to create the embedding. DFx provides a generic method that can be applied after training to any series of embeddings, as it only requires a maximum and minimum value for each dimension, which also only needs to be approximate to obtain reasonable results.

5.5 Creating Paraphrases

With an understanding of how DFx works and the goal of exploring a latent space, this section explores a method to use DFx to create paraphrases. As stated in the introduction, the idea here is to create a vector embedding following the flow of a sentence. If we consider a sentence as a combination of separate subphrases, then those are exchangeable for semantically equivalent ones and given a large enough corpus, there are sure to exist valid substitutions. In this application, a chunking method is initially applied to the entire corpus to break each input sentence into chunks, which are initially stored in DFx.

The first step in creating a paraphrase is to ingest a corpus into DFx using a method of chunking the phrases. The phrases are chunked from the input, keeping the sentence together by adding further chunks and re-embedding, step by step (chunk by chunk). This is intended to be the equivalent of a recursive network outputting an embedding for each token as it is ingested, but with the contextual understanding of the transformer. A single phrase is turned into many different chunks which are all embedded separately. The method works this way because transformer embeddings are context sensitive, therefore chunks are best compared with their context attached.

Once the corpus has been ingested, in inference mode, a query phrase can be chunked into candidate chunks for replacement using the same chunking method as with which the vectors were stored. For each candidate chunk DFx then returns the top k most salient matches. Each candidate chunk in the input phrase is replaced with all k DFx results for that chunk to create a minimum $k * \texttt{num\_chunks}$ new phrases. To make this more clear, if chunks were words, this is the exact method used in Table 1.1, where each chunk is replaced with the k most likely chunks.

Because DFx creates many different phrases, and many of them are not grammatically correct, there are 3 additional selection measures that are applied to this smaller corpus to improve the generated text — similarity, lan-

guage modelling, and grammar correction.

5.5.1 Chunking

While for the sake of examples, until now we mostly had word based chunking, for paraphrasing DFx actually uses a verb-phrase based method. The theory for this is that verbs are seen as idea transitions, moving the vector to a new location. While all tokens move the vector, some are relatively insignificant (stop words, articles, determiners etc.) and others, such as nouns, are harder to gauge in effect, particularly proper nouns, as some will be significant (Barack Obama) and others less so (Tobias Renwick) as in training USE has likely encountered a president of the US more frequently than an Edmonton based computer science researcher.

To create an input chunk verb phrases — verbs plus any adverbs — are detected using Spacy and the phrase is split around the verb, and the verb phrase itself is also kept. For example, given the input phrase, 'I went to the park', the chunker would create the chunk list ['I', 'went', 'I went', 'I went to the park']. Note that while the verb chunks are extracted, the complete sentence is also included for comparison, this way DFx can get a holistic view as well. It should be noted that because of the way the chunker works, replacements near the beginning of a phrase are more likely than those at the end of a phrase i.e DFx is more likely to have good matches for 'I went' than for the full phrase 'I went to the park' simply because the former is more common.

5.5.2 Queries

DFx is queried for each chunk created by the chunker separately against the entire corpus. This operation has a time complexity of $O(mn)$ where m is the number of chunks in the query phrase, and n is the number of chunks in the corpus. Additionally, the quantity of paraphrases returned is a minimum of $m * k$ – For the example phrase, initial paraphrases are returned where each token is substituted for its returned matches — these are the candidate paraphrases. Note that DFx is prevented from returning an exact embedding

match for the input query, this is important as it requires the text to be different than the input.

5.5.3 Similarity

To consider the entire composition of the phrase, all of the candidates are first re-ranked by cosine similarity to an embedding method on the original phrase i.e. how much the paraphrase changed the final embedding. The embedding method used for evaluation here is USE [10], however, any other embedding method could be used as a secondary measure. USE was chosen as it is least likely to compromise the measurements of BERT-Score [47] as it is a different base network, based on encoder-decoder rather than masked language modelling. The result of the similarity phase is the list of $k * num\_chunks$ candidate paraphrases scored and ranked by their cosine similarity to the input.

5.5.4 Grammar Correction

If the `GRAMMAR_WEIGHT` is greater than 0, each candidate phrase initially has its grammar corrected by language tool (a popular grammar corrector in Python) [43] before being added as a candidate. Language tool can fix many simple mistakes, such as missing punctuation and capitalization. Once the candidates are corrected, they are again checked for errors, and persistent errors are punished by multiplying their count by $-$`GRAMMAR_WEIGHT`.

$$-1 * \texttt{num\_mistakes} * \texttt{GRAMMAR\_WEIGHT} \tag{5.3}$$

This modelling is quite simple and heavily punishes phrases that are less grammatical than the rest at default settings.

5.5.5 Language Modelling

Current SOTA generative models are based on decoder-only style transformers, which are constructed by asking the network to predict the next word and

adjusting the output probabilities based on the cross-entropy loss of that prediction. Generative networks are seen as language models, which is to mean that they have a statistical understanding of the overall language. In this work, I use GPT-2 as a language model to score the likelihood of the returned phrases being proper English.

To create a score from GPT-2 I measure perplexity, which is essentially how confused the model was when it sees what the phrase was given the inputs. In order to transform the cross-entropy loss from GPT-2 to perplexity we sum the loss and divide by the tokens (n): $e^{\sum_0^n loss_n/n}$. To make perplexity useful for our scoring function it is inverted, therefore the language score is calculated:

$$\texttt{LANGUAGE\_MODEL\_WEIGHT} * \frac{1}{e^{\sum_0^n loss_n/n}} \tag{5.4}$$

The perplexity score is then multiplied by `LANGUAGE_MODEL_WEIGHT` to create a factor which is added to the final score.

5.5.6 Final Scoring

At the end, all the scores are modified according to the weights, the candidate list is re-sorted and the top-k are returned as valid paraphrases:

$$\textbf{cosine}(\mathit{initial}, \mathit{candidate}) \ + \ \textbf{lanuage}(\mathit{candidate}) \ + \ \textbf{grammar}(\mathit{candidate}) \tag{5.5}$$

5.5.7 Paraphrase Method Summary

Starting with a corpus of example text, each phrase in the corpus is chunked into embeddings around the verb-phrase chunks and stored using DFx. Given a query text, the query is also chunked, and each chunk is separately searched in DFx returning k possible replacements for that chunk. The matching chunk in the query text is replaced with each of the k results from DFx to generate different combinations of phrasings – paraphrases. The paraphrases are then scored by their cosine similarity to the input text, their lack of grammatical errors, and their predictability according to a language model to generate the predicted paraphrase.

Chapter 6

Results

6.1 Search Examples

DFx is first and foremost a search tool that focuses on maintaining the semantics of embeddings. In this section, DFx was given a large corpus (28.4 million) of tweets on climate change, and then various search queries were run to see the results. Once ingested, at a setting of 9 buckets, the dataset consumed only 4.5 GB of memory and a corpus wide search could be conducted in 0.65 seconds using a Ryzen 2700x processor.

Many topics, if they are prominent in the corpus, appear much like a keyword search – if there are K results that match the keywords, those are likely the results returned. However, for topics that seem a little obscure, the interesting features of embedding search can be shown. Below are the search results for a query about dogs causing global warming,

Dogs cause global warming
Puppies cause global warming.
Dogs contribute to man (animal) made global warming scam
unpopular opinion: dogs are the root cause of global warming
Cats and dogs are one of the main cause of global warming
Well its about time. Dogs cause global warming. Everyone knows that.
Dogs now just referring to the weather as global warming.
Global warming dog
Global Warming Makes Dogs Bite <LINK>
Just clarifying..Hot Dogs caused climate change..got it !

The first result is that Puppies cause global warming (note the omission of

the keyword and the similarity of the phrase and its structure). If we repeat the search with a different house-pet:

Cats cause global warming
Cats and dogs are one of the main cause of global warming
Are cats to blame for 'Global Warming'?
Cats not taking global warming seriously
Cats are pro global warming, they enjoy the heat (and watching humans suffer)!
2021: cats caused global warming <LINK>
And cats cause climate change. Idiots.
Absolutely–it's not cats that are causing climate change!
Global warming. The cat dragged in global warming.
Puppies cause global warming.

In this section the results appear mostly sarcastic, however, for the last results, 'cat' is replaced with 'puppy', which is still a mammal and a house-pet, but not the right one. A potential negative about DFx's search is that it always returns k results, even if they are not what we would consider a good match, it is the closest one in the corpus.

With DFx's search, a synonym is close enough. For example, if the user wants to search for a mammal, they can just use that word and still get reasonable results:

Mammals cause global warming
humans cause global warming
Humans cause Global Warming. <LINK>
#TriggerAConservativeIn4Words Humans cause global warming
The effects of global warming. It's hot and mammals are pissed <LINK>
Global Warming causes this..hehe
Humans caused global warming #justsaying
Global warming causes cooling
Humans are the leading cause of global warming #exterminatethehumanrace
HUMANS ARE THE REASON FOR GLOBAL WARMING

The results contain some interesting claims about global warming, but more importantly, they illustrate how semantic search is different from keyword searches. In many results, synonyms are preferred over results with more keyword matches that have a different meaning, for example, the text 'humans cause global warming' ranks more highly than 'The effects of global warming.

It's hot and mammals are pissed' despite the inclusion of the keyword 'mammals'. Because of this increased reliance on semantics, it is sometimes possible to use foreign language phrases and get the translation – though it is important to note that this is primarily an effect of how USE was trained, and DFx only facilitates the large scope of the search.

In French 'Je t'aime' means 'I love you':

Je t'aime
I love you. I love you. I love you. <LINK>
I luv u <LINK>
i love u <LINK>
I love you <LINK>
i lov u!!! <LINK>
I love you 😘 <LINK>
I LOVE YOU 😭 <LINK>
I love u 😍 <LINK>
I love u 😘😘😍😍 <LINK>

And in French 'Je te deteste' means 'I hate you':

Je te deteste
i loathe you <LINK>
i despise you <LINK>
I dislike you. <LINK>
I h8 you <LINK>
I h8 u. <LINK>
i dislike you so much <LINK>
I HATE YOUU <LINK>
I really loathe you. <LINK>
I really do hate you <LINK>

While all of these results are anecdotal, they provide a feeling of how the search works. The algorithm does like phrasings with the same keywords, but it sorts them somewhat differently depending on the context. Synonyms can be substituted at any point in the results, which is exactly the behaviour we want to exploit when we generate paraphrases.

Phrase 1	**Phrase 2**	**USE Cosine**	**Distilbert Cosine**
I went to **the** park	I went to **the the** park	0.99663705	0.9785236
I went to **the** park	I went to **the the the** park	0.98901767	0.9509182
I went to **the** park	I went to **a the** park	0.9770059	0.9694521
I **like** to wear hats	I hate to wear hats	0.8288758	0.9799255
riding	ridding	0.87408507	0.8724389
swim	swimming	0.9424287	0.9536646

Table 6.1: A comparison of cosine similarity with multiple, incorrect stopwords and poor results from BPE encoding

6.2 Notes about Transformer Encodings Examined

One of the most noteworthy effects of exploring a latent space with a tool like DFx is that you can observe certain patterns within the spaces. It is important to note that the following are simply observations obtained from investigating the latent spaces while learning to create paraphrases.

For the task of paraphrasing, it is an interesting connection that neither Distilbert [41] nor USE [10] consider stop words important. This means that both networks will accept doubled stop words, and score them reasonably well as a measure of similarity (Table 6.1). In early iterations of the paraphrase generation, the results would often incorporate doubled stop words, usually from the paraphrase being spliced in after a stop.

While USE is intended to be an embedding method, masked language models such as BERT [12] and by extension Distilbert [41] do not strictly return an embedding. For comparison, a Distilbert embedding is formed by averaging the hidden states of the second last layer over the output tokens.

Table 6.1 demonstrates that the addition of stop words appears insignificant to either method.

6.3 Validation

As DFx is, to our knowledge, a novel approach to word and sentence vector similarity it is reasonable to compare its retrieval to the most common vector comparison method used in NLP, cosine similarity. In brief, the cosine similarity of 2 vectors is a comparison between their angles, which does not account for magnitude, only direction. We expect that cosine similarity is a relatively efficient method, that can produce reasonable results, though this is heavily dependant on the input data.

To examine each method in kind, validation tests were completed on a subset of 1,000,000 tweets sampled from the corpus on climate change.

6.3.1 Cosine Comparison

The data here was selected as its metered length makes it suitable and clean to process. For preprocessing only links were replaced with 'URL', while hashtags, @ mentions, and other Twitter data (retweets, emojis etc.) were not altered. The search results are mostly data-agnostic but are sensitive to the vector embedding used. For the sake of expediency, we use the Universal Sentence Encoder (USE) [10] version 5, which has a 512dim output vector for up to 1024 tokens. USE was chosen as it is a simpler transformer type network, with reasonable performance on STS benchmarks.

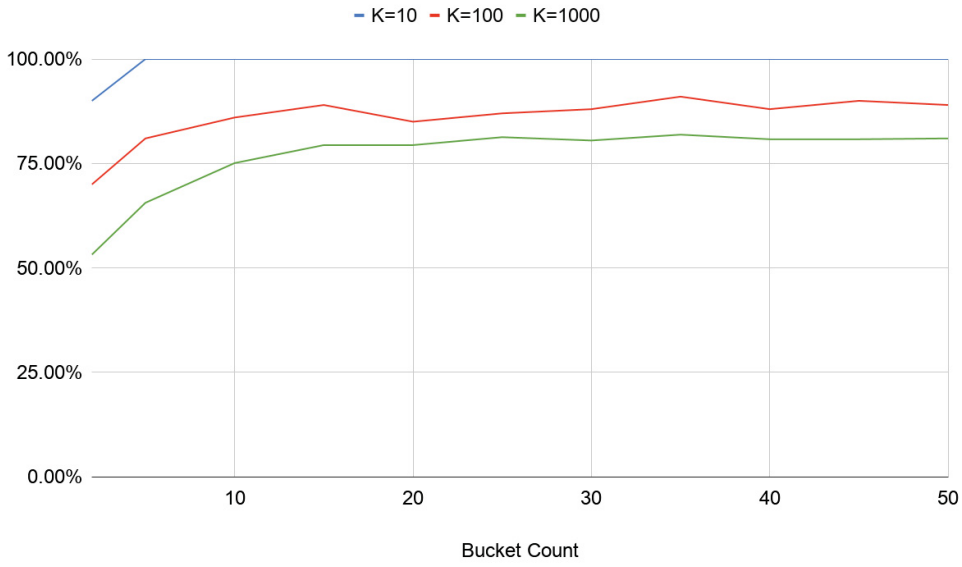


Figure 6.1: DFx agreement with cosine similarity as compared to bucket count for a single query

Both cosine similarity and DFx were used to embed the data which was then queried for the K nearest neighbours to a query phrase. The results of the queries were added to sets, where the overlap of those sets is considered a measure of agreement. In DFx, the initial input of 1,000,000 tweets was reduced to $\approx 269,000$ unique vectors, with some variance by bucket count (higher bucket counts result in fewer collisions, though at this dimensionality the difference is not striking). Also worthy of note is that in DFx it happens frequently that there are ties at the end of the search since the scores are discrete. Ties are important in DFx, as the algorithm is not ordered (it is multi-threaded) the final score values included will be effectively random. For example, if $K = 5$ and the options have scores of 1, 2, 3, 4, 4, 4 …which two of the three 4s that are included is whichever is added first during the search as there is no meaningful way to break the tie.

To determine the approximate bucket setting that would yield the best results, DFx was run with many different bucket settings across the corpus for three values of k: 10, 100, 1000 (Figure 6.1). The approximate knee of this curve, a bucket count of 16, was selected for the following larger scale

comparison.

To investigate the comparison between DFx and cosine similarity, the same set overlap methodology was conducted over 10 different queries. The results of this comparison are in Figures 6.2 & 6.3. Each Figure is from the same data, with figure 6.2 added to allow more visibility into the top 100 results of the chart, as they are likely to be much more important than the bottom 900. DFx generally aligns quite well with cosine similarity, typically providing the first 10 results without any discrepancies, however, some queries did have errors in the top 10, while the single highest agreement reached without error was at $K = 55$. At $K = 10$ DFx produced a mean of 95.4% overlap, while a mean of 91.5% overlap was obtained at $K = 100$, and 90.3% at $K = 1000$.

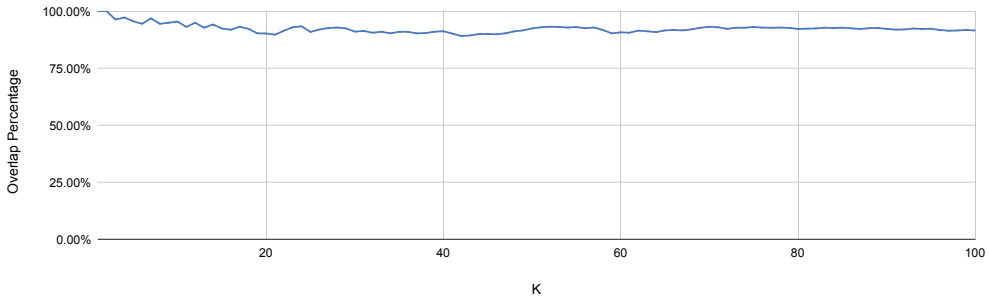


Figure 6.2: DFx mean error vs cosine similarity up to $k = 100$

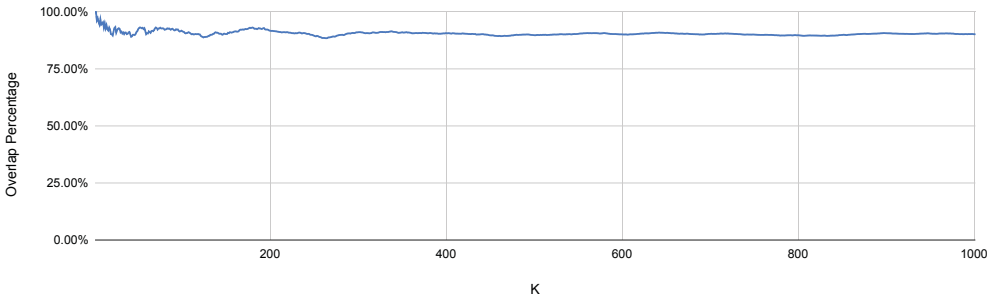


Figure 6.3: DFx mean error vs cosine similarity up to $k = 1000$

DFx's performance decreases with K because as K grows, and the difference between vectors becomes less significant, DFx begins to disagree more with cosine similarity as it does not respect small differences, it only measures

to its discrete thresholds. This means that while DFx concentrates on looking for significant differences, cosine similarity considers many small differences as more important. Additionally, as the count of vectors grows, the number of tied vectors at the end of the search grows.

6.3.2 Semantic Textual Similarity

Because there is a significant overlap between cosine similarity and DFx does not necessarily mean that DFx's discretization does not affect the semantic relatability of the vectors. To quantify that difference, we ran DFx's discretizer on top of USE [10] against the STS (Semantic Text Similarity) benchmarks from 2012-2016 as implemented in Facebook's SentEval framework [11].

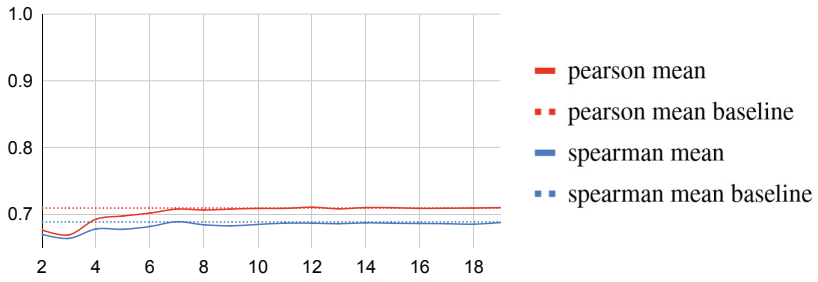
The STS task was a task within SemEval from 2012 to 2016 to illustrate how well an embedding captures the semantic differences between two phrases [2]. The task presents the algorithm with two sentences and asks for a score from 0 to 5, where 0 means that the sentences are completely different, and 5 means they are equivalent (Table 6.2). The phrases are extracted from different corpi from datasets of image captions, news headlines and user forums [2] and results reported (Figure 6.4) consist of a mean of all corpi in each test.

Phrase 1	**Phrase 2**	**Score**
The bird is bathing in the sink.	Birdie is washing itself in the water basin.	5
In May 2010, the troops attempted to invade Kabul.	The US army invaded Kabul on May 7th last year, 2010.	4
John said he is considered a witness but not a suspect.	"He is not a suspect anymore." John said.	3
They flew out of the nest in groups.	They flew into the nest together.	2
The woman is playing the violin.	The young lady enjoys listening to the guitar.	1
John went horse back riding at dawn with a whole group of friends.	Sunrise at dawn is a magnificent view to take in if you wake up early enough for it.	0

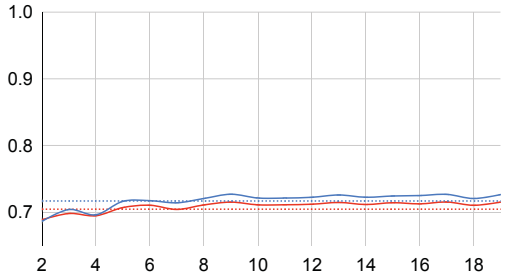
Table 6.2: STS Sentence comparisions — Adapted from Agirre, Banea, Cer, *et al.* [2]

The main goal of the STS task was to create embeddings that show a high correlation with these changes in semantic meaning, therefore scores are measured using both the Spearman and Pearson correlation coefficients to show similarity, where the Pearson coefficient is considered the primary measure. If the Spearman correlation is high while the Pearson is lower, this likely means that the relationship between one or more of the different brackets is less represented by values than by rank. For example, if an algorithm was successful at differentiating between 0 and 5, but ranked middling scores(1-4 inclusive) as between 3.0 and 3.4, it may have a good Spearman score, but a poorer Pearson score. If the Pearson is higher it means that there is a higher correlation between the actual values than the rank.

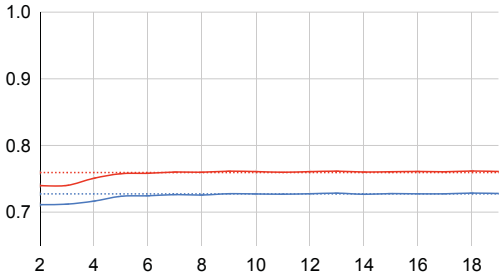
Figure 6.4 shows the results obtained by bucket size when DFx is overlaid on USE [10]. In general, DFx stops harming performance between 6 to 10 buckets (dependant on STS year). There is a trade-off between resource consumption and accuracy which is well illustrated, where lower bucket settings consume fewer resources, and higher bucket settings show better alignment with the baseline algorithm. Figure 6.4b illustrates that at points discretization can improve results, where beyond 5 buckets, DFx outperforms the baseline. I hypothesize that this is due to the insignificance of small embedding changes on this particular dataset.



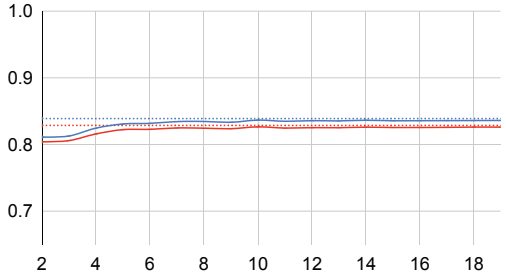
(a) STS-12



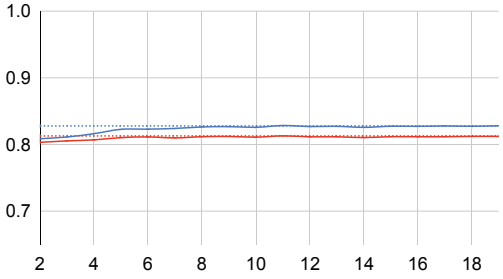
(b) STS-13



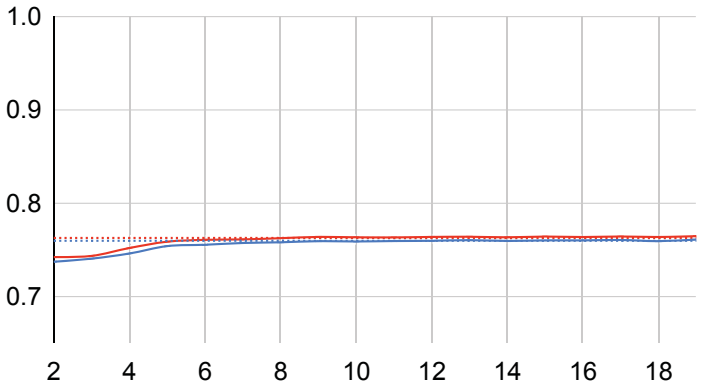
(c) STS-14



(d) STS-15



(e) STS-16



(f) STS-12 — 16 Combined

Figure 6.4: STS Benchmark correlation with semantic similarity across DFx bucket count.
Baseline performance is USE version 5 [10]

6.4 Paraphrasing

To illustrate the usefulness of a tool like DFx, it was examined for its ability to generate paraphrases. Paraphrasing a corpus can be a way to expand a low data corpus or can be used as a way to verify a network's understanding of language. It is common to ask a person to restate an idea in different words if it is not well understood by the listener, this demonstrates an attempt to re-establish communication along similarly understood terms. In this way, paraphrasing (or more exactly the ability to map similar phrases together) is central to many NLP tasks, such as question answering, information retrieval and dialogue [30].

6.4.1 Dataset

The Wikianswers corpus contains 18 million word-aligned question pairs and 2.5 million original questions, which were identified as identical questions by the WikiAnswers community [14]. For the following tests, DFx is allowed to ingest the original, un-lemmatized corpus of 2.5 million questions for search, however, no alignment with the paraphrases is made. In other words, DFx is asked to identify the paraphrases from a series of chunks of the original text and reassemble them into coherent phrases. DFx is used in this application does not create original text, only potentially original combinations of chunks of it, and to do that, it needs to be shown the possible options.

For evaluation 10,000 paraphrase sets are selected at random from the 18 million aligned sets. The candidate paraphrases are lemmatized after selection for comparison using Spacy.

6.4.2 Metrics

By its very nature, paraphrasing is difficult to evaluate. If an idea is well represented in different words, how can we empirically, and automatically, measure a paraphrase? There are potentially infinite ways to restate any given phrase, therefore corpi built for evaluating paraphrases generally contain

many different paraphrases of a given statement (references), against which a candidate paraphrase is evaluated.

The most accurate way to evaluate a generated paraphrase is likely human judgement with a large number of annotators, however, due to time constraints, I have not been able to complete this style of evaluation here. For automated evaluation of results similar metrics to translation and summarization are employed — ROUGE (Recall-Oriented Understudy for Gisting Evaluation) [29] and BLEU (Bilingual Evaluation Understudy) [36], and the recent BERT-Score [47].

6.4.3 BLEU

BLEU [36] attempts to measure the precision of a paraphrase to several candidates using n-gram similarity. What this means is that BLEU counts the matching words (BLEU-1), the matching groups of 1 and 2 words (BLEU-2), the matching groups of 1,2, and 3 words (BLEU-3) and the matching groups of 1,2,3, and 4 words (BLEU-4). BLEU is primarily a measure of precision where precision is the ratio of tokens in the generated paraphrase that also appear in the reference — $\frac{\#\ of\ overlapping\ tokens}{Total\ tokens\ in\ paraphrase}$.

A potential deficiency of a precision-based score is that high precision can be obtained at the expense of recall — if we don't guess we are never wrong. Imagine if our paraphrase consists of a single word 'the'. The word 'the' will appear in many references, and will probably net a reasonable precision score. To combat this, BLEU introduces a brevity penalty, where if a generated phrase is shorter than the reference, the final score is multiplied by that ratio. When the phrases are equal length, or the candidate has a greater length, the brevity penalty is 1. A negative consequence of the brevity penalty is that BLEU penalizes concise paraphrasing — a paraphrase must be as long or longer than the original. This is far from ideal as a more practical application of paraphrasing would be to improve concision.

For the task of paraphrasing, BLEU is not ideal, however, it is commonly reported [30], [33]. Table 6.3 represents the BLEU score for DFx generated

paraphrases on the Wiki-Answers corpus. Note that the larger k is, the better the result for DFx, however, the larger k is the more expensive the paraphrase is to generate.

DFx k	**Metric**	**Score** (%)
1	BLEU	34.51
1	BLEU-1	74.97
1	BLEU-2	59.91
1	BLEU-3	46.51
10	BLEU	55.45
10	BLEU-1	85.84
10	BLEU-2	75.54
10	BLEU-3	65.96
50	BLEU	60.74
50	BLEU-1	87.12
50	BLEU-2	77.93
50	BLEU-3	69.86

Table 6.3: Mean BLEU results on 10,000 phrases randomly selected from the Wiki-Answers Corpus
Created using corpus-bleu from the NLTK python library

6.4.4 ROUGE

ROUGE is a series of metrics that measure the number of n-grams that overlap between a generated and a reference paraphrase. Reported in Table 6.4 are the unigram ROUGE scores (ROUGE-1), bigram ROUGE scores (ROUGE-2) and ROUGE-l which is the longest common subsequence. Unlike BLEU, ROUGE, as per its namesake, involves recall in the scoring rather than solely precision.

Scores for ROUGE are reported in Table 6.4 are means for F1, precision and recall. ROUGE recall scores measure the amount of a given reference summary that overlaps with the generated paraphrase: $\frac{\#\ of\ overlapping\ tokens}{Total\ tokens\ in\ reference}$ while the precision, like BLEU, measures the fraction of tokens in the paraphrase that were needed to make the reference: $\frac{\#\ of\ overlapping\ tokens}{Total\ tokens\ in\ generated\ paraphrase}$. ROUGE F1 is the harmonic mean combination of precision and recall.

To arrive at a comparison with current work on paraphrase generation, I present Table 6.5. The above results are summarized for DFx compared to

k	Metric	Precision (%)	Recall (%)	F1 (%)
1	ROUGE-1	53.37	57.66	54.06
1	ROUGE-2	22.88	24.56	23.11
1	ROUGE-l	65.13	55.09	52.06
10	ROUGE-1	66.48	68.13	65.94
10	ROUGE-2	38.33	37.90	37.22
10	ROUGE-l	65.13	65.83	64.26
50	ROUGE-1	67.19	71.54	67.91
50	ROUGE-2	42.43	42.06	41.21
50	ROUGE-l	66.00	68.70	66.13

Table 6.4: Mean ROUGE results on 10,000 phrases randomly selected from the Wiki-Answers Corpus
Created using the rouge python library

other paraphrase generating algorithms as listed in Liu, Mou, Meng, *et al.* [30]. DFx_{10} outperforms the other candidates, however, it should be noted that part of the reason for this is that if a given reference had a paraphrase included in the training set, DFx is very good at selecting that phrase because of the way it is built.

6.4.5 BERT-Score

BERT-Score is a different measurement as it attempts to use the cosine similarity between BERT embeddings to evaluate the semantic relation of two phrases [47]. The candidate is compared to the references by greedily matching the highest cosine similarity tokens with each-other. This implies that order is not as significant, as long as the embeddings are consistent, and the true meaning should be more readily represented.

Precision and Recall for BERT-Score are slightly different than for n-gram based approaches. For BERT-Score both precision and recall have the same meaning as before (candidate to reference, reference to candidate), except that each token is matched to another to maximize its potential score, rather than considering the actual order directly. In this way, the intent is that if tokens have the same meaning that they are matched together in latent space.

An example where BERT-Score outperforms ROUGE and BLEU is where

Method Class	Model	BLEU	ROUGE-1	ROUGE-2
Supervised	ResidualLSTM	27.36	48.52	18.71
Supervised	VAE-SVG-eq	32.98	50.93	19.11
Supervised	Pointer-generator	39.36	57.19	25.38
Supervised	Transformer	33.01	51.85	20.70
Supervised	Transformer+Copy	37.88	55.88	23.37
Supervised	DNPG	41.64	57.32	25.88
Domain Adapted	Pointer-generator	27.94	53.99	20.85
Domain Adapted	Transformer+Copy	29.22	53.33	21.02
Domain Adapted	Shallow fusion	29.76	53.54	20.68
Domain Adapted	MTL	23.65	48.19	17.53
Domain Adapted	MTL+Copy	30.78	54.10	21.08
Domain Adapted	DNPG	35.12	56.17	23.65
Unsupervised	VAE	24.13	31.87	12.08
Unsupervised	CGMH	26.45	43.31	16.53
Unsupervised	UPSA	32.39	54.12	21.45
Search	DFx_1	34.51	54.06	23.11
Search	DFx_{10}	55.45	65.94	37.22
Search	DFx_{50}	**60.74**	**67.91**	**41.21**

Table 6.5: Wikianswers Corpus Comparison — Adapted from Liu, Mou, Meng, *et al.* [30]

the meaning is changed by a small number of words. For example, since ROUGE and BLEU are n-gram based, a paraphrase of "I'm going to the lake" could be "I'm not going to the lake". This has no brevity penalty, near-perfect precision, and perfect recall, but it means the opposite (negation) and is not a proper paraphrase. A better paraphrase — "I'm headed to the lake" — receives lower ROUGE and BLEU scores, but a better BERT-Score (Table 6.6). As a more extreme example, Zhang, Kishore, Wu, *et al.* provide the input phrase: "people like foreign cars" where semantically incorrect candidate of "people like visiting places abroad" achieves a higher score than "consumers prefer imported cars" as there are no n-grams shared [47].

In a fashion, BERT-Score is very similar to DFx, but in reverse. A phrase is broken down into its separate tokens to create a matrix of contextual embeddings, which is then compared to a candidate, broken down in the same way. It is important to note that DFx does not directly share any embeddings with BERT, it uses USE [10] as a base for all reported work. Results of

Candidate	**Metric**	**F1** (%)
“im not going to the lake”	BLEU	55.72
“im not going to the lake”	ROUGE-1	90.01
“im not going to the lake”	ROUGE-2	66.67
“im not going to the lake”	ROUGE-l	90.91
“im not going to the lake”	BERT-Score	78.82
“im headed to the lake”	BLEU	0.00
“im headed to the lake”	ROUGE-1	80.00
“im headed to the lake”	ROUGE-2	50.00
“im headed to the lake”	ROUGE-l	80.00
“im headed to the lake”	BERT-Score	87.38

Table 6.6: Metrics on the paraphrases of “im going to the lake”

BERT-Score evaluation are in Table 6.7.

k	**Metric**	**Precision** (%)	**Recall** (%)	**F1** (%)
1	BERT-Score	63.25	67.26	64.40
10	BERT-Score	77.52	78.51	77.61
50	BERT-Score	77.88	78.93	77.92

Table 6.7: Mean BERT-Score results on 10,000 phrases randomly selected from the Wiki-Answers Corpus [47]

The most significant disadvantage of ROUGE is shared with BLEU, and that is that the reference still must capture the same words used by the generator, including spelling mistakes which are frequent in Wikianswers. The fourth phrase in our random selections was “who deliverd the getty burg address ?”, which DFx_1 paraphrased as ‘Who delivered the Getty burg address ?’, which does not net a perfect score in any measure (Table 6.8).

If the generator produces a completely novel paraphrase that was unaccounted for by the authors of the corpus, the ROUGE score will be 0. This is what BERT-Score is attempting to correct.

6.5 Paraphrasing Summary

DFx can be used to create paraphrases that in terms of metrics, rival state-of-the-art supervised systems. I note, however, that DFx is incentivized to make the most simple paraphrases possible, and it draws upon an existing

Metric	**F1** (%)
BLEU	64.35
ROUGE-1	85.71
ROUGE-2	66.67
ROUGE-l	85.71
BERT-Score	81.87

Table 6.8: Metric error on the paraphrase 'who delivered the getty burg address ?' of reference 'who deliverd the getty burg address ?'

corpus. The paraphrases it generates are limited to different combinations of the chunks in the corpus, and while that number is extremely large, it is not generative. There is some argument to be made that current generative language models also perform this same task, however with a higher level of obfuscation. By this I mean, they too can only re-combine phrases they have previously seen and do not truly create. For example, if an LM was trained without ever having read Dr. Seuss's The Cat in The Hat, it would never come up with the hat part of that phrase. If asked to complete the phrase 'the cat in the <>' it would likely choose box or house and never a hat. In this regard DFx is the same, it recombines what it is given without the ability to create.

Chapter 7

Conclusion

I presented DFx, a vector search system which through a discrete encoding facilitates an exhaustive search of latent space. DFx primarily works by compressing the vectors through discretization and accelerating comparison through a binary encoding which allows the hamming distance to be used as a measure of similarity. To examine the uses of the system, evidence was provided that DFx maintains results comparable to cosine similarity (Figure 6.1), and is effectively identical at $k = 10$. To further demonstrate that DFx does not harm semantic relatability it was used as an overlay to the Universal Sentence Encoder [10] on the Semantic Textual Similarity Benchmark, where between 5 to 10 buckets, performance is near identical (Figure 6.4).

In a test of its ability, DFx was used to create potentially novel paraphrases by replacing verb phrase chunks in the WikiAnswers dataset [14] through an exhaustive search of that corpus. Results are comparable to state-of-the-art supervised systems (Table 6.5), however, DFx needs to be shown the data and is not truly generative, but more of an identification tool that is used in a compositional way.

References

[1] H. Abdi and L. J. Williams, "Principal component analysis," *Wiley interdisciplinary reviews: computational statistics*, vol. 2, no. 4, pp. 433–459, 2010. 22

[2] E. Agirre, C. Banea, D. Cer, M. Diab, A. Gonzalez-Agirre, R. Mihalcea, G. Rigau, and J. Wiebe, "SemEval-2016 Task 1: Semantic Textual Similarity, Monolingual and Cross-Lingual Evaluation," in *Proceedings of the 10th International Workshop on Semantic Evaluation (SemEval-2016)*, Association for Computational Linguistics, 2016, pp. 497–511. DOI: 10.18653/v1/S16-1081. [Online]. Available: http://aclweb.org/anthology/S16-1081. 54

[3] J. Alammar, *The Illustrated Transformer*, 2018. [Online]. Available: http://jalammar.github.io/illustrated-transformer/. 17

[4] C. Bannard and C. Callison-Burch, "Paraphrasing with bilingual parallel corpora," in *Proceedings of the 43rd Annual Meeting of the Association for Computational Linguistics (ACL'05)*, 2005, pp. 597–604. 24, 25

[5] R. Barzilay and L. Lee, "Learning to paraphrase: an unsupervised approach using multiple-sequence alignment," *arXiv preprint cs/0304006*, 2003. 24

[6] R. Barzilay and K. McKeown, "Extracting paraphrases from a parallel corpus," in *Proceedings of the 39th annual meeting of the Association for Computational Linguistics*, 2001, pp. 50–57. 24

[7] R. E. Bellman, *Adaptive control processes: a guided tour*. Princeton university press, 2015, vol. 2045. 9

[8] D. W. Blalock and J. V. Guttag, "Bolt: Accelerated data mining with fast vector compression," in *Proceedings of the 23rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, 2017, pp. 727–735. 21, 23

[9] T. B. Brown, B. Mann, N. Ryder, M. Subbiah, J. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell, *et al.*, "Language models are few-shot learners," *arXiv preprint arXiv:2005.14165*, 2020. 18

[10] D. Cer, Y. Yang, S.-y. Kong, N. Hua, N. Limtiaco, R. S. John, N. Constant, M. Guajardo-Cespedes, S. Yuan, C. Tar, *et al.*, "Universal sentence encoder," *arXiv preprint arXiv:1803.11175*, 2018. 6, 8, 31, 34, 45, 50, 51, 5

[11] A. Conneau and D. Kiela, "Senteval: An evaluation toolkit for universal sentence representations," *arXiv preprint arXiv:1803.05449*, 2018. 29, 54

[12] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, "BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding," *arXiv:1810.04805 [cs]*, May 2019, arXiv: 1810.04805. [Online]. Available: `http://arxiv.org/abs/1810.04805`. 8, 10, 18, 19, 31, 50

[13] J. Du, F. Qi, and M. Sun, "Using BERT for Word Sense Disambiguation," *arXiv:1909.08358 [cs]*, Sep. 2019, arXiv: 1909.08358. [Online]. Available: `http://arxiv.org/abs/1909.08358`. 10, 19

[14] A. Fader, L. Zettlemoyer, and O. Etzioni, "Paraphrase-Driven Learning for Open Question Answering," in *Proceedings of the 51st Annual Meeting of the Association for Computational Linguistics*, 2013. 57, 64

[15] G. Finley, S. Farmer, and S. Pakhomov, "What Analogies Reveal about Word Vectors and their Compositionality," in *Proceedings of the 6th Joint Conference on Lexical and Computational Semantics (*SEM 2017)*, Association for Computational Linguistics, 2017, pp. 1–11. DOI: `10.18653/v1/S17-1001`. [Online]. Available: `http://www.aclweb.org/anthology/S17-1001`. 12

[16] J. R. Firth, *Papers in Linguistics, 1934-1951*. Oxford University Press, 1958. 11

[17] Y. Gong, S. Kumar, V. Verma, and S. Lazebnik, "Angular quantization-based binary codes for fast similarity search," in *Advances in neural information processing systems*, 2012, pp. 1196–1204. 21

[18] Y. Gong, S. Lazebnik, A. Gordo, and F. Perronnin, "Iterative quantization: A procrustean approach to learning binary codes for large-scale image retrieval," *IEEE transactions on pattern analysis and machine intelligence*, vol. 35, no. 12, pp. 2916–2929, 2012. 21–23

[19] I. Goodfellow, Y. Bengio, and A. Courville, *Deep Learning*. MIT Press, 2016, `http://www.deeplearningbook.org`. 14, 16

[20] Google, *Measuring Similarity from Embeddings - Clustering in Machine Learning*, 2020. [Online]. Available: `https://developers.google.com/machine-learning/clustering/similarity/measuring-similarity`. 13

[21] S. Han, H. Mao, and W. J. Dally, "Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding," *arXiv preprint arXiv:1510.00149*, 2015. 20

[22] A. Handler, "An empirical study of semantic similarity in WordNet and Word2Vec," p. 23, 2014. [Online]. Available: `https://scholarworks.uno.edu/td/1922/`. 10, 13

[23] M. Harris, *Mixed-Precision Programming with CUDA 8*, Aug. 2020. [Online]. Available: https://developer.nvidia.com/blog/mixed-precision-programming-cuda-8/. 20

[24] S. Hochreiter, "The vanishing gradient problem during learning recurrent neural nets and problem solutions," *International Journal of Uncertainty, Fuzziness and Knowledge-Based Systems*, vol. 6, no. 02, pp. 107–116, 1998. 15

[25] S. Hochreiter and J. Schmidhuber, "Long short-term memory," *Neural computation*, vol. 9, no. 8, pp. 1735–1780, 1997. 8, 15

[26] H. Hotelling, "Analysis of a complex of statistical variables into principal components.," *Journal of educational psychology*, vol. 24, no. 6, p. 417, 1933. 22

[27] H. Jégou, M. Douze, and C. Schmid, "Product Quantization for Nearest Neighbor Search," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 33, no. 1, pp. 117–128, Jan. 2011, ISSN: 0162-8828. DOI: 10.1109/TPAMI.2010.57. 24

[28] Q. Le and T. Mikolov, "Distributed representations of sentences and documents," in *International conference on machine learning*, 2014, pp. 1188–1196. 8, 13

[29] C.-Y. Lin, "Rouge: A package for automatic evaluation of summaries," in *Text summarization branches out*, 2004, pp. 74–81. 29, 58

[30] X. Liu, L. Mou, F. Meng, H. Zhou, J. Zhou, and S. Song, "Unsupervised paraphrasing by simulated annealing," *arXiv preprint arXiv:1909.03588*, 2019. 26, 57, 58, 60, 61

[31] Y. Liu, M. Ott, N. Goyal, J. Du, M. Joshi, D. Chen, O. Levy, M. Lewis, L. Zettlemoyer, and V. Stoyanov, "Roberta: A robustly optimized bert pretraining approach," *arXiv preprint arXiv:1907.11692*, 2019. 31

[32] Machynist and Kippinitreal, *How We Scaled Bert To Serve 1+ Billion Daily Requests on CPUs*, May 2020. [Online]. Available: https://blog.roblox.com/2020/05/scaled-bert-serve-1-billion-daily-requests-cpus/. 20

[33] N. Miao, H. Zhou, L. Mou, R. Yan, and L. Li, "CGMH: Constrained Sentence Generation by Metropolis-Hastings Sampling," *arXiv:1811.10996 [cs, math, stat]*, Nov. 2018, arXiv: 1811.10996. [Online]. Available: http://arxiv.org/abs/1811.10996. 58

[34] Microsoft, *Turing-NLG: A 17-billion-parameter language model by Microsoft*, Feb. 2020. [Online]. Available: https://www.microsoft.com/en-us/research/blog/turing-nlg-a-17-billion-parameter-language-model-by-microsoft/. 10, 18

[35] T. Mikolov, I. Sutskever, K. Chen, G. S. Corrado, and J. Dean, "Distributed Representations of Words and Phrases and their Compositionality," p. 9, 2013. 8, 10, 11, 13

[36] K. Papineni, S. Roukos, T. Ward, and W.-J. Zhu, "BLEU: a method for automatic evaluation of machine translation," in *Proceedings of the 40th annual meeting of the Association for Computational Linguistics*, 2002, pp. 311–318. 29, 58

[37] PyTorch, *Quantization*, Aug. 2020. [Online]. Available: `https://pytorch.org/docs/stable/quantization.html`. 20

[38] T. Renwick and R. Szostak, "A Thesaural Interface for the Basic Concepts Classification," p. 5, 2020. 19

[39] D. E. Rumelhart, G. E. Hinton, and R. J. Williams, "Learning internal representations by error propagation," California Univ San Diego La Jolla Inst for Cognitive Science, Tech. Rep., 1985. 8, 14

[40] R. Salakhutdinov and G. Hinton, "Semantic hashing," *International Journal of Approximate Reasoning*, vol. 50, no. 7, pp. 969–978, 2009. 21

[41] V. Sanh, L. Debut, J. Chaumond, and T. Wolf, "DistilBERT, a distilled version of BERT: smaller, faster, cheaper and lighter," *arXiv preprint arXiv:1910.01108*, 2019. 31, 50

[42] A. Šauperl, "Precoordination or not?: A new view of the old question," *Journal of Documentation*, vol. 65, no. 5, pp. 817–833, Sep. 2009, ISSN: 0022-0418. DOI: `10.1108/00220410910983128`. 19

[43] L. Tool, *Spell and Grammar Checker*, 2020. [Online]. Available: `https://languagetool.org/`. 45

[44] v. sagar vaibhav, *You Won't Believe This One Weird CPU Instruction!* 2019. [Online]. Available: `https://vaibhavsagar.com/blog/2019/09/08/popcount/`. 41

[45] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. Kaiser, and I. Polosukhin, "Attention Is All You Need," *arXiv:1706.03762 [cs]*, Dec. 2017, arXiv: 1706.03762. [Online]. Available: `http://arxiv.org/abs/1706.03762`. 2, 8, 16

[46] Y. Weiss, A. Torralba, and R. Fergus, "Spectral hashing," in *Advances in neural information processing systems*, 2009, pp. 1753–1760. 21, 22

[47] T. Zhang, V. Kishore, F. Wu, K. Q. Weinberger, and Y. Artzi, "BERTScore: Evaluating Text Generation with BERT," in *International Conference on Learning Representations*, 2020. [Online]. Available: `https://openreview.net/forum?id=SkeHuCVFDr`. 24, 30, 45, 58, 60–62