Evaluating Word Embedding for System Reference Codes: Building Better Features for Error Code Classification

Matthew Pickering

Department of Computer and Information Science

University of Pennsylvania

Supervisor

Joseph Devietti Ph.D.

In partial fulfillment of the requirements for the degree of

Bachelor of Science in Engineering

April 29, 2023

Acknowledgements

I'd first like to acknowledge my mentor and friend Brett Benefield for his unlimited enthusiasm, mentorship, encouragement, and patience for all of the work done for this thesis. Each time I thought we'd reached the ceiling of what we could accomplish, he was always sure to raise it.

I must also acknowledge those IBM Z developers who've shown incredible patience and interest in assisting in the work done in this thesis, particularly Scott Swaney and Chris Hutton for getting the word out and being indispensable to evaluating my work.

Finally, I give thanks to Jeffrey Tracey, my manager at IBM and generally good human, and senior thesis program manager Jonathan M. Smith at the University of Pennsylvania for making this thesis possible. Even when I was confident there was too much red tape to cut through, we found support from both IBM and the University of Pennsylvania towards allowing this idea to come to life.

This thesis is dedicated to the coaches, teachers, professors, and friends that made learning fun. Thanks for building me up every step of the way.

Abstract

IBM has identified value in training classifiers to automate the detection of defects in the output stream of hardware and firmware events (known internally as System Reference Codes or SRCs) from its Z series of mainframe computers. Previous work has shown limited performance in large part due to scarcity of labeled data. We compare this limitation to that of Natural Language Processing (NLP) before the introduction of word vector embeddings. This thesis seeks to evaluate the performance of a word vector embedding over the language of SRCs to overcome the limitation as NLP has. We find the language of SRCs obeys the same assumptions of natural language that make these embeddings work. Using such an embedding, we are able to make a 1.57 times improvement over existing precision benchmarks in the task of defect detection. Our work suggests there is vast room for growth in a variety of machine learning tasks on SRCs using word embeddings.

Contents

Chapter 1

Introduction

1.1 IBM Z System Reference Codes

The IBM Z Series is a family of mainframe computers built to be a large enterprise solution to latency sensitive applications, especially designed for banks, insurance firms, retailers, and other clients to whom reliability is critical [1]. Towards guaranteeing this reliability, IBM employs a language of 8-digit hexadecimal strings identifying a unique event having occurred on the machine, which can represent a hardware or firmware event [2]. These strings are referred to as System Reference Codes (SRCs) or "refcodes" here-on. The utility of this language gives Z developers a common reference for debugging unexpected behavior on any Z system.

IBM keeps a number of internal Z Systems of any Z generation for support testing, each being equipped with a Support Element (SE). Any workload submitted to a system will be monitored by the SE on the hardware and software level, and the SE will stream the refcodes identifying important events to the system log. These logs are regularly retrieved from the SEs on each system and stored for developer reference in a service known as the System Test Event Logging and Analysis engine, or STELA [3].

1.2 Automated IBM Z Defect Detection

IBM offers continued developer support to its Z clients and is always in development of unstable builds for the next Z iteration. A client who discovers an issue in the field or an internal test technician of an unstable build may submit a support ticket to IBM's Z development team for investigation. We refer to any undesired behavior identified in a support ticket as a defect. IBM Z developers leverage the test systems described above and the data in STELA to fix defects as they arise.

To speak towards automation, it's important to outline a regular IBM Z developer's testing and development pipeline. One regular development cycle follows:

1. A test technician identifies a defect on the machine and submits a support ticket.
2. A developer receives the ticket and submits a workload to a test system to reproduce the behavior.
3. The developer investigates the results of the workload and identifies a fix.
4. The fix is deployed to a test system.
5. The fix is deployed to the next build.

When we refer to automated defect detection, we precisely want to automate step 1 of this pipeline: automatically classify sequences of refcodes on any system as symptomatic of a defect or benign.

McCain, E. et al identified several potential benefits to using Machine Learning (ML) to automate the IBM Z developer's testing pipeline: improved accuracy of defect detection, reduced time spent in data entry for ticket submission, and uniform ticket format for a consistent developer experience [3]. To yield these, McCain, E. et al designed a defect detection model, automating step 1, and an automatic ticket writing program, automating step 2. We are most interested in defect detection, so we will focus on McCain, E. et al's model for step 1, named AutoJup [3].

1.2.1 Limitations of AutoJup

To justify the tools researched in this thesis, we will examine the limitations of AutoJup here and discuss how they are ameliorated later. We highlight three limitations here:

1. AutoJup is trained on labels produced by a rule engine, not developer-identified defects.

2. AutoJup is trained on a one-hot-encoding of the SRC language.

3. Autojup predicts on individual refcodes, not sequences of refcodes.

Before its training, McCain, E. et al first designed a system of rules, or if-then checks, to determine if a refcode represented a defect. A collection of refcodes were then labeled according to this rule engine, and AutoJup was given the resulting data set as training data [3]. This approach implies AutoJup is not strictly predicting whether or not a refcode is a defect but if the underyling rule engine would classify a refcode as a defect. AutoJup is benchmarked at an accuracy of 85.85% [3], which suggests not that 85.85% of AutoJup's predictions align with whether or not the refcode is a defect but that 85.85% of AutoJup's predictions align with the rule engine's output. If the underlying rule engine is an excellent classifier, then AutoJup is always outperformed by simply substituting it for the rule engine it was trained on. If the rule engine is not an excellent classifier, then AutoJup is training towards a flawed hypothesis.

McCain, E. et al one-hot-encoded the refcode as as a feature to AutoJup [3]. By one-hot-encode, we refer to transforming text data to an integer-valued feature vector by creating a sequence of binary indicators for each unique string in the data that show 1 if the input instance is equal to the indicator's corresponding string and 0 otherwise. One-hot-encoded feature inputs impose a limitation on the system that the hypothesis must be learned for each unique input individually [4]. This considerably increases the volume of training data necessary to get strong results from the system [4]. In other words, unless AutoJup has already seen a particular refcode with a label, it can do no better than a guess one way or the other.

AutoJup is designed to predict whether or not a single refcode represents a defect [3], rather than a string of refcodes. This imposes the limitation that the hypothesis on any refcode is insensitive to the refcodes emitted before it: the system must declare every refcode either always represents a defect or never represents a defect. This is undesirable since all but the worst refcodes are only *conditionally* symptomatic of undesired behavior.

1.3 Word Embeddings

We refer to an m-dimensional word embedding as mapping of some language L to the space of m-dimensional real-valued vectors $\mathbb{R}^m$. That is, a word embedding maps any word $w \in L$ to some vector $[v_1, v_2, ..., v_m]$ where $v_i \in \mathbb{R} \ \forall \ 1 \leq i \leq m$. This understanding of word embeddings is borrowed from Natural Language Processing (NLP).

We briefly review the history of NLP's adoption of word embeddings towards a comparison to the state of IBM Z defect detection today. Before wide use of vector representations in NLP, Bengio, Y. et al described the state of NLP in 2003 as suffering from the *curse of dimensionality* [5]. The curse is particularly referring to the requirement of adding a new dimension to the input feature vector for each and every word in the language's vocabulary. In statistical terms, Bengio, Y et al point out modeling the joint distribution of each of 10 consecutive words using one-hot-encoded discrete random variables on a vocabulary V of size 100,000 gives $10^{50} - 1$ free parameters. Mikolov, T et al summarize the effect of this discrete representation, noting the performance of such systems is limited by the volume of data available to train on [4]. Towards dispelling the curse of dimensionality, Bengio, Y. et al pioneer a system of word representations as real-valued vectors, whose joint distribution has a number of parameters scaling linearly with the size of the vocabulary V [5]. The substantially reduced parameter space gives way to systems that generalize much better with less training data [5]. The training procedure for these vectors are then filtered through a series of improvements before reaching the degree of optimization provided by Mikolov, T et al in 2013 with their algorithm *Word2Vec* [4], which is the algorithm used to train the word

embeddings described in this thesis.

Mikolov, T et al provide two models for training NLP word embeddings: the Continuous Skip-gram Model and the Continuous Bag of-Words Model. We briefly describe the training procedure for Mikolov, T et al's Continuous Skip-gram model towards a better understanding of why it delivers results.

The training procedure can be understood as a shallow neural network with three layers: the input layer, hidden, layer, and output layer. Suppose we seek to embed a vocabulary V as vectors of m dimensions each. Let our input be one-hot-encoded as above, i.e. we create an input node $v \in \{0, 1\}$ for each word $v \in V$. For any particular word, say "app", we notate its input as v_{app}. Next, create a hidden node h_i with no activation function for each $1 \leq i \leq m$ for an embedding of m dimensions. Finally, create an output node α_i for each word in $i \in V$ with a softmax activation function $\phi(\alpha_i)$:

$$\phi(\alpha_i) = \frac{\exp \alpha_i}{\sum_{j \in V} \exp \alpha_j}$$

Altogether, this architecture takes a word $v \in V$ as input and outputs a probability distribution over all words in V representing the probability any $v' \in V$ appears in the context of v. Figure 1.1 shows a visual representation of the network.

For training, we collect a set of sentences S. The trainer defines some window size k and builds the training set by iterating over each sentence $s \in S$. If s has ℓ words, we'll construct a training instance for each word $w_i,\ i \in [1 \dots \ell]$. Let each w_i be the *focus word* of its instance, and build the instance by one-hot-encoding w_i to get the feature vector and computing the word frequencies of the focus word's *context*, $\{w_j | j \in [i - k \dots i + k]\} \setminus w_i$. These frequencies provide the probabilities that represent the labels of the training instance. With the training set built, we proceed with stochastic gradient descent. To retrieve the m-dimensional vector $[v_1, v_2, \dots, v_m]$ of any $w \in V$, assign v_i to be the weight of the connection between the input node for w and the hidden node h_i.

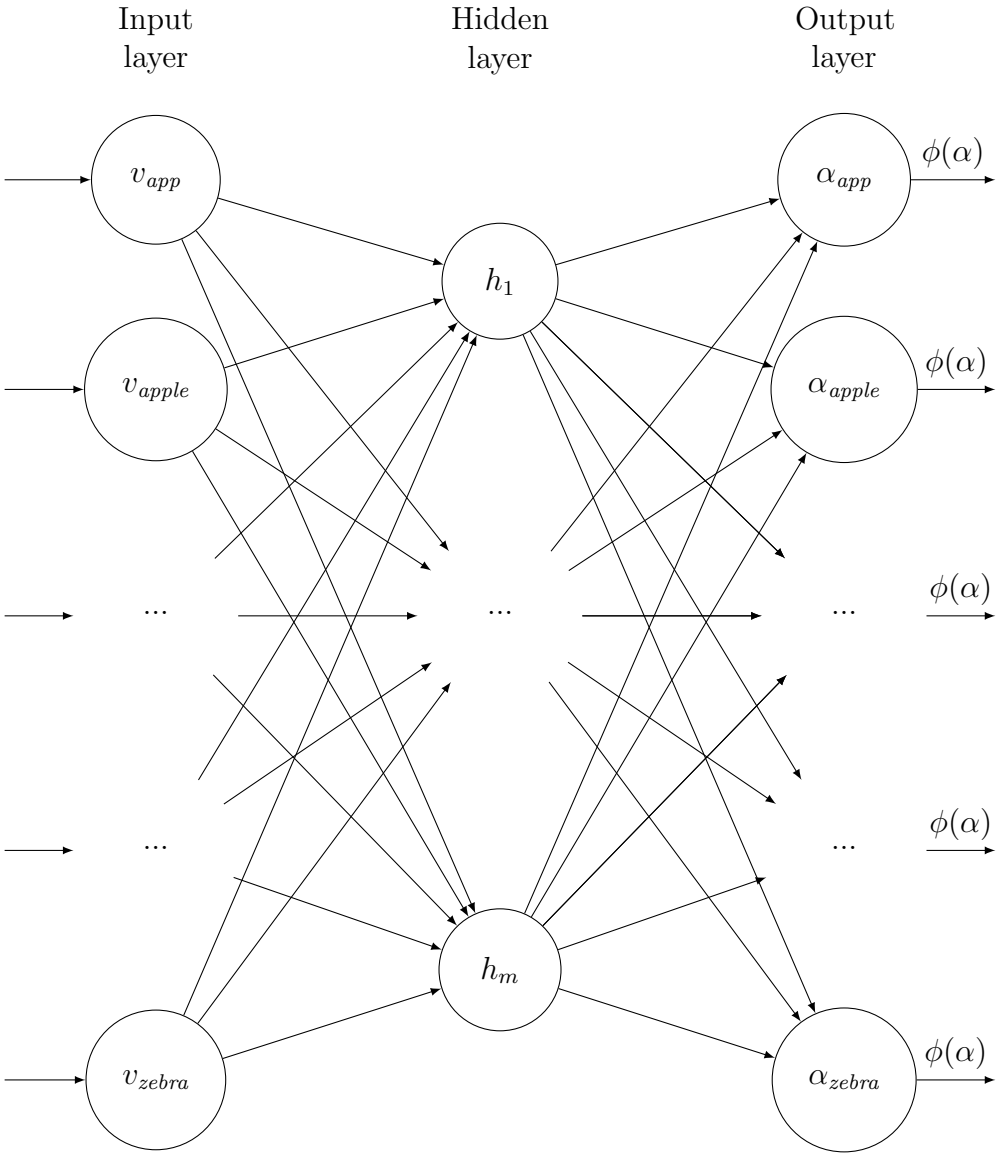

Figure 1.1 Skip-grams deep neural network architecture with input vocabulary $V = \{v_{app}, v_{apple}, \ldots, v_{zebra}\}$.

1.3.1 Why do they work?

Our objective is to train a word embedding satisfying the property that words appearing in similar contexts have similar vectors [5]. More formally, any two words w_1, w_2 that frequently share similar contexts should have vector representations v_1, v_2 such that their distance $\|v_1 - v_2\|$ is relatively small compared to the distance between v_1 or v_2 and some other third word w_3 that frequently appears in different contexts from w_1 and w_2:

$$\|v_1 - v_2\| \ll \|v_1 - v_3\|$$

$$\|v_1 - v_2\| \ll \|v_2 - v_3\|$$

How does the architecture described above achieve this? As above, suppose we have a pair of words w_1, w_2 that frequently appear in similar contexts and some other word w_3 that rarely appears in a context similar to w_1 or w_2. Since w_1 and w_2 have similar contexts, they will have a similar probability distribution over any word's chances of appearing in their contexts. Keeping in mind that any word's vector representation is equivalent to the activation of the hidden layer, the model's error is minimized by moving their vector representations closer together. The opposite can be said for w_1 or w_2 and w_3. Having very different probability distributions over their contexts, the model is rewarded by maintaining distant activations for the hidden layer and consequently keeping their vector representations distant. Thus, w_1 and w_2 have a small distance between their vector representations relative to their distance to w_3's vector representation.

To conclude, the critical feature of natural language that the *Word2Vec* family of algorithms assumes is that semantically similar words appear in similar contexts. This observation provides the lens through which we will evaluate the possibility of applying these techniques in the domain of Z refcodes.

1.4 Goals of the Thesis

The primary goal of this thesis is to evaluate the performance of word embedding techniques from NLP on the language of IBM Z refcodes. The objective is moti-

vated by parallels between the state of NLP before widespread usage of real-valued vector embeddings. IBM has identified value in using refcodes as model features [3], but such systems have been shown to be limited by the curse of dimensionality, just as NLP was before discovering vector representations of words [5]. Along the way, we want to test if the language of these refcodes obeys the same assumption as natural language which allows *Word2Vec* and its family of algorithms to work: do semantically similar refcodes appear in similar contexts?

Additionally, we seek to evaluate if refcodes embedded as vectors improve the performance of existing defect detection efforts. We will address the other limitations of AutoJup described above and attempt to provide a new performance benchmark to better understand how we are performing now, how we can perform with the work outlined in this thesis, and how we can branch out using this work to other machine learning tasks with refcodes as features.

Chapter 2

Methods

2.1 Data Collection & Organization

We begin by collecting a corpus of event data on which we can train the embedding. Our data was collected from STELA [3] and included all recorded events across 380 test machines spanning from January 1st, 2021 to November 10th 2022, excluding events manually designated as error injections This collection summed to a total 683,950,540 events, each having an associated timestamp, system ID, and refcode.

In the context of natural language, word embeddings are trained on a collection of sentences and use punctuation to delimit sentences to determine the context of any focus word. So, a natural challenge to applying these techniques on event data is that we have no such punctuation and no clear definition of a "sentence". It was determined from qualitative discussion with domain experts that most events which share a cause-effect or common-cause relationship occur within fewer than five seconds of each other. So, we considered refcodes in the order in which they were emitted and delimited sentences by periods of ≥ 5 seconds where an event was not emitted. When delimited into sentences, our collection summed to 163,937,561 sentences.

2.2 Refcode Embedding Training

Training followed the algorithm faithful to *Word2Vec*'s skip-grams approach [4], as described in the introduction. We used PySpark's Word2Vec implementation

to distribute training across several worker nodes [6].

Several embeddings were trained across different hyperparameters to evaluate how sensitive the training process is to parameter changes. We chose to individually vary number of dimensions in the embedding, the window size k, and tried swapping each refcode out for a substring of the last six digits.

The final variation was motivated by the internal structure of each refcode. There exist groups of refcodes whose least significant six digits are identical and who represent the same error but whose two most significant digits vary to identify the generation of hardware it was detected on. Since these refcodes identify the same event on the machine, it's conceivable that the embedding can be improved by collapsing them into the same word.

2.3 Refcode Embedding Evaluation

Most pioneering work on word embeddings for natural language evaluated the performance of any embedding by considering any word and its nearest neighbor, then intuit whether or not the pair make sense as neighbors [4]. Mikolov, et al. evaluated their architectures on the more difficult task of considering three words at a time in the following form: *vector("big")* is to *vector("biggest")* as *vector("small")* is to x [4]. The vector x was evaluated by considering the nearest neighbor of the vector produced by *vector("biggest")* - *vector("big")* + *vector("small")*. If the embedding produced "smallest", then the embedding was considered high performing on that example [4].

These evaluation schemes make sense applied to natural language because natural language is constructed with many such word analogs as "big" and "biggest" and "small" and "smallest". This is not true of Z refcodes: two refcodes may identify similar events but there are no natural groups of word pairs who all share the same relationship. This is more difficult when considering that engineers working on embeddings for English natural language tend to have a great intuitive understanding of what words mean and which words should be similar. This is also not true of Z refcodes: few engineers have a great intuitive understanding of the event any refcode identifies and each engineer may have their own notion of what sim-

ilarity between refcodes means. The first of these concerns is addressed by using a different scheme for collecting expert qualitative testimony on the embedding performance and the second is kept in mind for later analysis.

One benefit of refcode embedding evaluation over natural language is the end uses of refcode embeddings are defined more narrowly: we precisely seek to make classifications over defect status. Word embeddings in the NLP context have a broad scope of uses and ethical implications, so the degree to which they improve the performance of other models is a poor performance measure. This is not so true for models that take Z refcodes as features. Hence, we will quantitatively evaluate our embeddings by the marginal improvement in performance over AutoJup, the existing standard for Z defect detection [3].

2.3.1 Expert Testimony Collection

We leverage expert testimony to evaluate our trained embeddings by designing an interface into which experts can plug in two refcodes, say A and B, and get back the cosine similarity of A and B as well as the cosine similarity of the 10th and 100th most similar words to A. Users then have an opportunity to provide feedback on the results, quantitatively on a Likert scale and qualitatively via a text box.

The Likert scale was in response to the question, "Do you agree with the following statement: "The provided similarity between the two refcodes aligns with my knowledge and experience of the two codes." Then, the user could select "Strongly Disagree", "Disagree", "Neither Agree nor Disagree", "Agree", "Strongly Agree". Users were identified using their email for follow-up questions and duplicate detection.

2.3.2 Defect Detection Model Training

We will evaluate how useful the embedding is when used as a feature for defect detection. To do so, we will begin by collecting a data set of developer responses to tickets created in AutoJup's current deployment. This data set will double as a quantitative benchmark for AutoJup and a new defect detection model's training and test data towards an improvement on AutoJup.

2.3.2.1 Data Collection & Organization

AutoJup was deployed for testing in January, 2021 where it has written up approximately 10 low-severity tickets for developer investigation each day on the previous day's refcode streams. We classified each of these tickets into one of three end behaviors: open, resolved, or rejected. We define an open ticket as not having received developer attention, a resolved ticket as one that a developer recognized as a defect and fixed, and a rejected ticket as one that a developer identified as non-representative of a true defect.

We acknowledge a possibility of non-response bias in the dataset described above. If a developer has identified some ticket as benign, then it's conceivable they would simply leave the ticket alone rather than expend the clicks necessary to mark it as a false alarm. Since our objective is to provide an upper bound on AutoJup's performance, we elected to exclude these instances from the data set.

We also note AutoJup classifies on individual refcodes instead of sequences. In order to surpass that limitation, we identify the refcode associated with each ticket and join it against the data set we collected for embedding training to retrieve its context. For labeling purposes, the sequence of refcodes that contains any labeled refcode will inherit the label of the refcode.

After filtering to only those tickets which received developer interaction, we finished with 1,333 total instances, 987 of which were classified as rejected and 346 of which were classified as resolved. Since the data set includes all tickets carrying labels that AutoJup flagged as a defect, this data set can provide a benchmark for AutoJup's precision, or the percent of flagged refcodes that truly represented a defect. Since the data set strictly includes flagged instances, AutoJup's true recall cannot be known, which we will keep in mind for analysis later.

2.3.2.2 Model Architecture

We design our model to overcome each of the current limitations AutoJup: training on data labeled via a rule engine, one-hot-encoding the language of refcodes, and predicting on individual refcodes instead of sequences.

We first overcome the training data by using the data described above. Since instances are labelled manually by developers, we are certain the hypothesis we

are training towards is the ground truth classification defined by the developers themselves, as opposed to training to mimic a rule engine.

The next limitation of AutoJup is one-hot-encoding the input refcode. This is overcome by instead using the features provided by our trained refcode embeddings. This gives our mechanism for evaluation: if the embedding indeed embeds some sense of similarity between refcodes, then this allows our hypothesis to generalize between refcodes rather than individually training a prediction for each.

The final limitation we overcome is predicting on refcode sequences as opposed to individual refcodes. Given it's frequent that refcodes are only *conditionally* symptomatic of a defect, the context around any refcode is critical to determining if the refcode in question is a defect. We overcome this by instead classifying on sequences of refcodes. Inspired architectures for natural language, we use a recurrent network, specifically a Long-Short Term Memory (LSTM) architecture. For testing, we varied the size of the hidden layer between 16 and 128, the number of layers between 1 and 3, and the learning rate between 0.002 and 0.040. These boundaries were determined by stagnation in changes of performance when going beyond them. To avoid overfitting, dropout with probability $p = 0.5$ was added to the outputs of the LSTM layer.

Chapter 3

Results

We will consider qualitatively whether or not the notion of similarity captured by the embedding aligns with domain experts' sense of similarity between refcodes and then evaluate how the embedding can quantitatively improve the existing defect detection model AutoJup. Unless otherwise specified, results presented with respect to a particular embedding are measured from our best performing embedding `cut_refs0`, which was trained with a dimensionality of 50, window size 5, learning rate 0.025, and clipped the 2 most significant digits off refcodes as described in section 2.2.

3.1 Expert Testimony Results

We received responses from five seperate Z developers across different system domains. Of those developers, we interviewed two for greater detail on their responses. Provided the relatively small size of the sample and the complications of defining similarity with respect to refcodes, we will analyze these results qualitatively. Between all pairs of refcodes experts evaluated, we found groups of refcodes which seemed well-aligned with the expert's definition of similarity and pairs which seemed to defy an expert's expectation of similarity. Two examples are provided below:

Refcodes of the form $3031000X$ where $0 \leq X \leq 3$ are a family of refcodes that identify clock stop failures where X represents the number CPUs found in failure. By most notions of similarity, these refcodes ought to be very close together in

	30310000	30310001	30310002	30310003
30310000	1.000	0.761	0.673	0.671
30310001	0.761	1.000	0.699	0.802
30310002	0.673	0.699	1.000	0.853
30310003	0.671	0.802	0.853	1.000

Table 3.1 Pair-wise cosine similarity scores between unique clock stop events where the least significant digit identifies the number of CPU failures detected. All pair-wise similarity scores are relatively high, falling in the 99th percentile of most similar refcodes.

	L3 cache purged	L4 cache purged	L3 cache purged 2x
L3 cache purged	1.000	0.189	0.435
L4 cache purged	0.189	1.000	0.364
L3 cache purged 2x	0.435	0.364	1.000

Table 3.2 Pair-wise cosine similarity scores between events 3030B030, 3030B035, and 3030B130 respectively identifying L3 cache purged, L4 cache purged, and L3 cache purged twice on `cut_refs0`. L3 cache purged is embedded more similarly to L3 cache purged twice than L4 cache purged.

a high performing refcode embedding. In alignment, all evaluated embeddings showed very high pair-wise cosine similarity between all pairs. For `cut_refs0` in particular, all pair-wise similarity scores were in the 99th percentile of most similar refcodes. The pair-wise results of these refcodes for `cut_refs0` are summarized in table 3.1.

The refcodes 3030B030, 3030B035, and 3030B130 respectively identify the L3 cache purged, the L4 cache purged, and the L3 cache purged twice due to Chip Errors (CEs). With respect to these refcodes, all scores above 0.4 still fall above the 90th percentile in similarity despite smaller absolute values than the previous example. As shown in table 3.2, the event signifying the L3 cache is purged shows a much higher similarity to the L3 cache being purged twice over the L4 cache being purged. Qualitatively, this result would be high performing for a developer who is interested in events emitted from a particular cache but is low performing for a developer who is interested in caching failures in general.

These two examples characterize a general observation that events identifying more grave errors tended to show higher absolute similarity scores than those events that identify recoverable errors. That is, refcodes which signaled a fatal system event such as a clock stop tended to show similarity scores in the ranges of

0.65-0.9 with their closest neighbors meanwhile recoverable errors showed scores in the range of 0.35-0.45 with their closest neighbors.

3.2 Defect Detection Results

3.2.1 Benchmarking AutoJup

By nature of the data we sourced, the only sequences of refcodes we can access which are labeled with their defect status is the tickets AutoJup has wrote in its deployment. Therefore, we can get a sense of AutoJup's precision, or the number of defects AutoJup flagged that represent true defects, but we can't make any conclusions about its recall, or the number of true defects that AutoJup missed. Across 1,333 created tickets, 346 tickets being classified as 'Resolved' benchmarks AutoJup at a precision of 25.96% (see figure 3.1).

3.2.2 Model Performance

Across all parameter permutations, we found our best-performing LSTM using one layer with a hidden size of 32. Using the data set formed by developers' responses to AutoJup's tickets, the data was split 80/20 into the training and test set respectively and trained over 25 epochs at a learning rate of $\alpha = 0.01$. For any sequence of refcodes, the RNN output a probability p that the input sequence is symptomatic of a defect. On the test set, thresholding p on the interval [0, 1] achieved an ROC curve with area 0.7182 (see figure 3.2).

To provide a more immediate comparison against AutoJup's benchmark, we computed the particular threshold t on the ROC curve that maximized the distance from the random classifier, which came out to $t = 0.33$. This particular threshold's model produced an overall accuracy of 68.16%, a precision of 40.91% (see figure 3.3), and a recall of 52.17%.

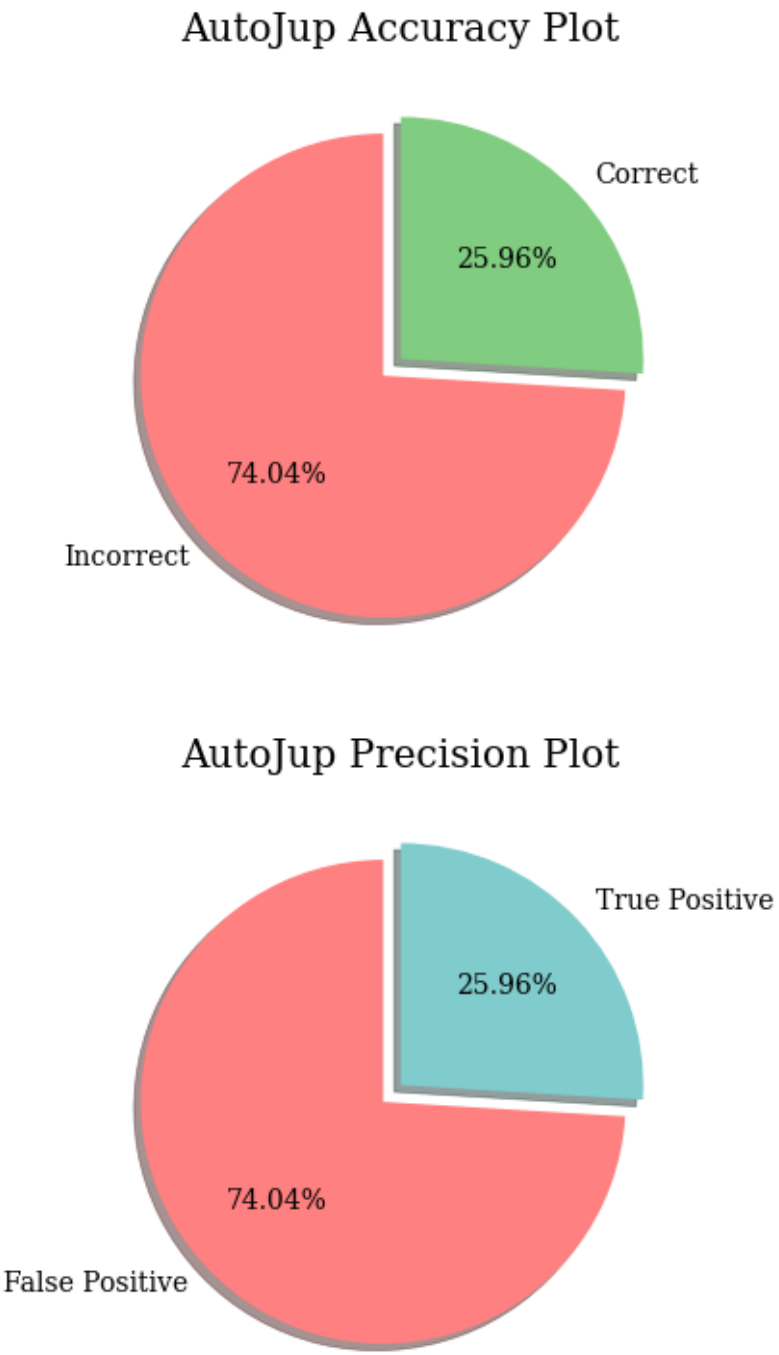

Figure 3.1 Of the tickets that saw developer attention, 25.96% were resolved as true defects. Note precision and accuracy are equal since data could only be collected for those events AutoJup classified as a defect.

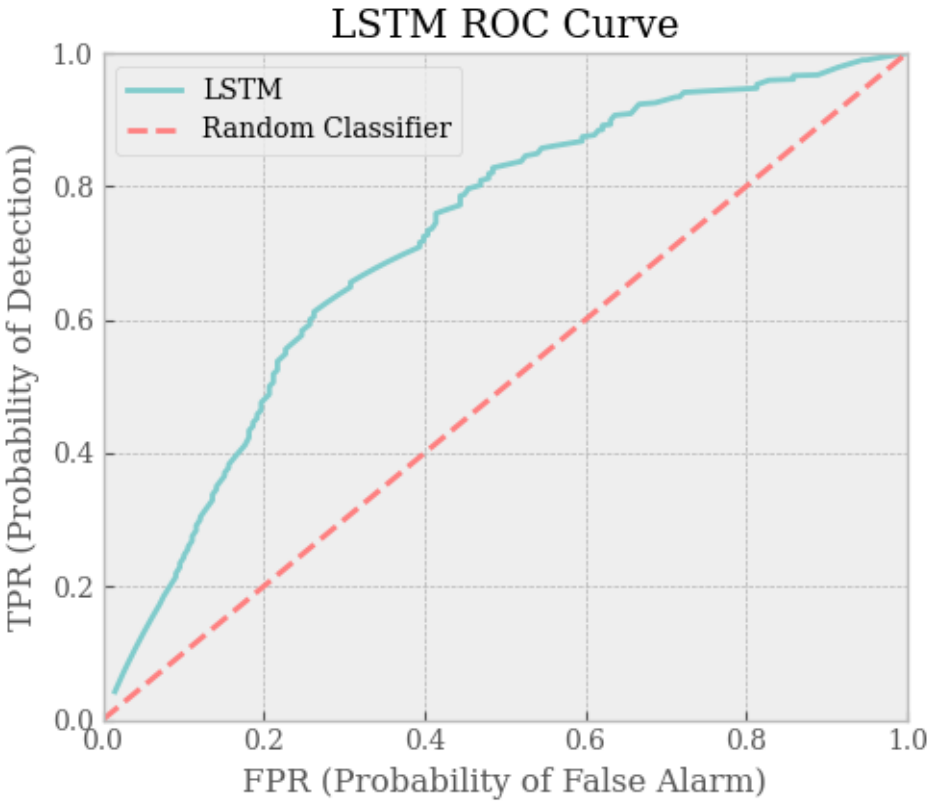

Figure 3.2 Best performing LSTM's ROC curve provided an area under curve of 0.7182.

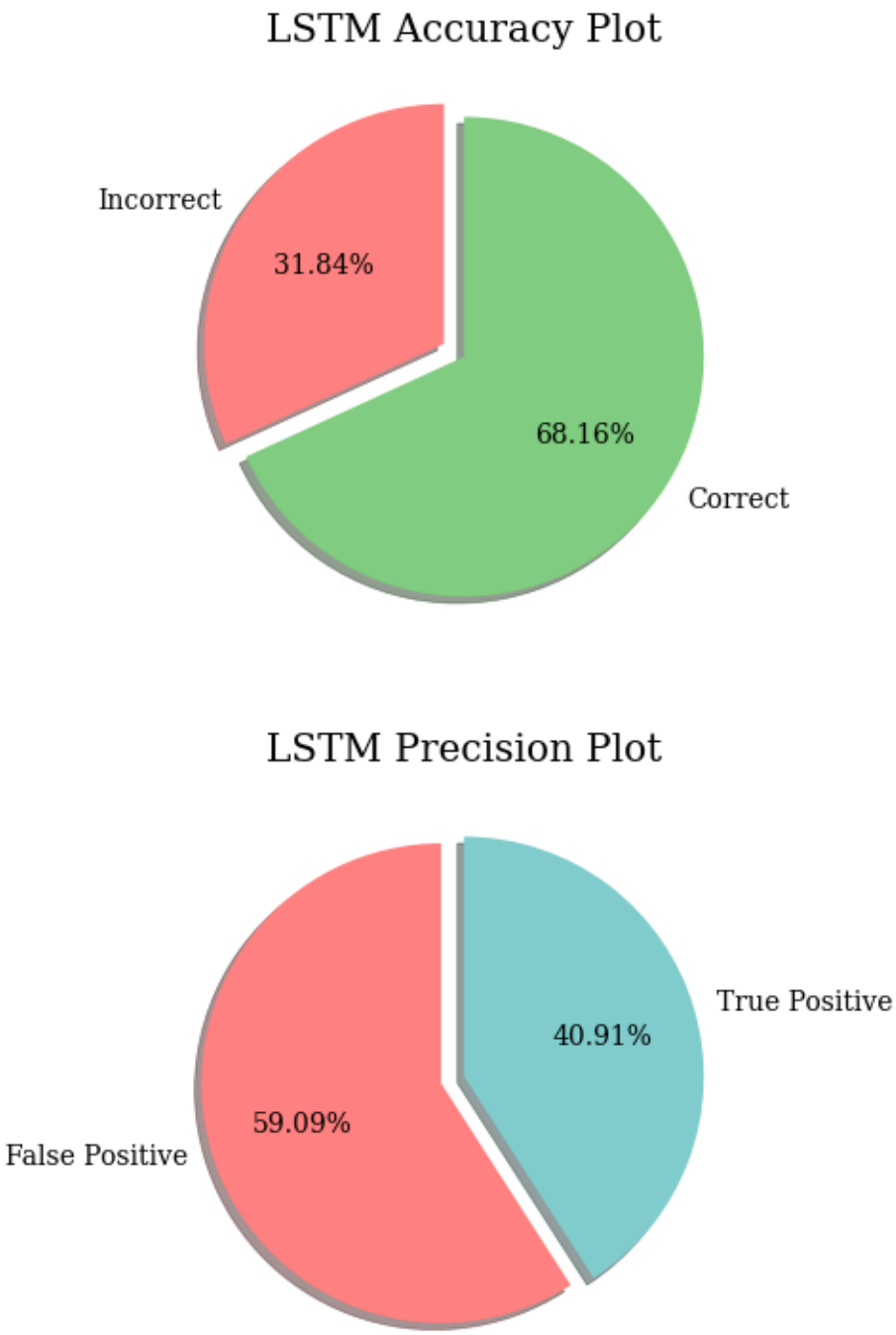

Figure 3.3 The LSTM trained on the refcode embedding achieved an accuracy of 68.16% and precision of 40.91%.

Chapter 4

Discussion

4.1 Z SRCs & Properties of Natural Language

Our results suggest the subset of refcodes that identify critical errors obeyed the property of natural language that semantically similar refcodes show up in similar contexts. We found fatal error events in particular showed relatively large pairwise similarity values that aligned well with domain experts' understanding of the refcodes, whereas results were generally modest or inconsistent for benign or less serious error events.

Towards a potential explanation, we return to an example from natural language: if "The dog ran around the house" and "The cat ran around the house" both appear in the training set, then the model can generalize "dog" and "cat" appear in the same context and thus are embedded as relatively similar. The key property is that both "dog" and "cat" shift the distribution of the words in their contexts towards each other which rewards the model for embedding their vectors close together. We suggest that more severe error events exhibit this property to a greater magnitude: by nature of being more severe events, they will coincide more frequently with the system emitting other severe errors as various systems on the machine fail. In contrast, more benign refcodes are by nature less serious and affect fewer other systems on the machine. Intuitively, this would lead such events to exert less influence on their context and consequently reduce the frequency they co-occur with the same set of other words.

4.2 SRC Embeddings & Machine Learning

Our results show a strong improvement over AutoJup on the particular set of tickets AutoJup has written during its deployment. However, the low volume of data used for training and sampling bias stop the results from providing a decisive improvement to the defect detection task at large. We will discuss the limitations of the training data and the generalizations we are able to make from the results.

We acknowledge there is inherent sampling bias in the data we had available to train our LSTM on for the defect detection task. Naturally, a sequence of refcodes labeled with its defect status only reached our data if AutoJup flagged the sequence as containing a defect. Thus, our data is biased towards the subset of tickets AutoJup flags as defects. We cannot make any assumptions about which qualities the data is biased towards except for considering the underlying rule engine AutoJup was trained to mimic, which we don't have access to in great detail [3]. As a result, we cannot make any assumptions about how our LSTM would perform on the defect detection task when exposed to all data leaving the machines.

These biases in our training data shed light upon the greater barrier to complex model design: we argue the most critical limitation of model design in the space of refcodes is a lack of high quality data in large quantities. Particularly for defect detection, the nature of the task is that the defect status of a ticket can only be determined by a relatively small population of subject matter experts whose priorities are software development before data curation. The only realistic avenues for data collection require methods that record data from the work they are already completing, which tends to bias itself towards the issues that already receive developer attention. Therefore, the most successful models will be ones that can take take inherent features of the Z SRC language and generalize with relatively few training samples.

Our results suggest that word embeddings of refcodes offer one solution to meet this gap and design models that generalize on fewer labeled training instances. With an Area Under Curve (AUC) of 0.7182 (see figure 3.2), our LSTM showed modest but considerable results on the task of defect detection, in spite of the relatively small data set of 1,333 training instances. We reason this performance

is a consequence of the same benefits word embeddings provide in the context of natural language: by designing features that encode the underlying semantics of the words they represent, models can generalize with much fewer training instances [4]. We are led to believe the refcode embeddings described in this thesis offer a strong foundation for training high performing models in spite of the scarcity of labeled refcode data.

Chapter 5

Conclusions

Machine learning techniques applied over Z SRCs, or refcodes, have presented a ripe automation opportunity to improve tools assisting developers and catch trends in refcodes that a human might miss. The first attempt at one such task, defect detection, exposed the underlying limitations of data quantity and quality of labeled refcode instances.

We explore a novel approach to resolving the data scarcity described above by applying word embeddings from NLP to the language of Z refcodes. Just as natural language models rely on word embeddings to encode the underlying semantics of the language, we sought out a refcode embedding that assigns vectors to refcodes that keep their underlying semantics intact. To evaluate the efficacy of these methods, we looked to identify whether or not the language obeys the same critical property of natural language that embedding training algorithms take advantage of: words with similar meanings appear in similar contexts. Our qualitative results suggest pairs of IBM Z refcodes identifying similar events indeed tend to appear in similar contexts, particularly so for more severe errors.

Quantitatively, we find modest but promising results on the defect detection task with LSTMs built on top of trained refcode embeddings, achieving an AUC of 0.7182 on a training set of just 1,333 instances. These results suggest vector embeddings provide one solution to the missing foundation necessary to see strong performance in a variety of machine learning tasks on Z refcode data.

Future work in the space of IBM Z defect detection that could be particularly insightful would apply model architectures from the NLP domain such as the LSTMs described in this thesis to production environments. Given the limitations

of data used in this thesis, we could build a more complete view of the quantitative performance of defect detection models built on top of refcode embeddings by exposing the model to the complete volume of data available on IBM's test systems and labeling predicted defects by domain experts.

In the particular domain of training refcode embeddings, the techniques utilized in this thesis were limited to skip-gram architectures. Vectorization using matrix factorization techniques has also shown promising results in other domains, suggesting a similar exploration of them on IBM Z refcodes could improve upon our work.

References

(1) Jacobi, C.; Webb, C. *IEEE Micro* **2020**, *40*, 50–58.

(2) IBM System reference codes `https://www.ibm.com/docs/en/i/7.1?topic=problems-system-reference-codes` (accessed March 20, 2023).

(3) McCain, E. et al. *IBM Journal of Research and Development* **2020**, *64*.

(4) Mikolov, T.; Chen, K.; Corrado, G.; Dean, J. Efficient Estimation of Word Representations in Vector Space, 2013.

(5) Bengio, Y.; Ducharme, R.; Vincent, P.; Jauvin, C. *Journal of Machine Learning Research* **2003**, *3*.

(6) Feature Extraction and Transformation - RDD-based API `https://spark.apache.org/docs/2.2.0/mllib-feature-extraction.html#word2vec` (accessed March 22, 2023).