Neural Language Models for Data-driven Programming Support

by

Xin Rong

A dissertation submitted in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
(Information)
in the University of Michigan
2017

Doctoral Committee:

Associate Professor Eytan Adar, Chair
Assistant Professor Walter Lasecki
Associate Professor Satish Narayanasamy
Assistant Professor Stephen Oney
Professor Dragomir R. Radev, Yale University

Xin Rong
ronxin@umich.edu

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ABSTRACT

Neural Language Models
for Data-driven Programming Support

by

Xin Rong

Chair: Eytan Adar

Programming can be hard to learn and master. Search engines and social Q&A websites offer tremendous help to programmers, but great expertise (e.g., "Google-fu") is required to efficiently use these resources and successfully solve complex problems. An integrated system that can recognize a programmer's tasks and provide contextualized solutions is thus desirable, and ideally programmers can interact with the system using natural input channels, in a way similar to how they communicate with a human expert. To enable such an integrated system, neural language models constitute a promising solution. These models encode programming language in the same high-dimensional space with data of other modalities, and can be trained in an end-to-end fashion. By leveraging the massive data about programming knowledge that are available online, including social Q&A websites, tutorials, blogs, and open-source code repositories, we can train neural language models to support a variety of user intentions, including the long-tail ones.

We propose three studies related to using neural language models to solve programming problems in practice. First, we introduce CodeMend, an intelligent programming assistant that supports interactive programming. The system employs a bimodal embedding model to encode programming language and natural language in the same vector space. We demonstrate that this model can effectively understand the code context and associate it with user

input to suggest relevant code modifications. We also develop novel user interface to render search results in a way that makes the problem solving process more efficient.

Second, we propose a deep learning pipeline that converts data visualization images to source code. The pipeline is built by using computer vision techniques and recurrent neural networks, and it supports the user to get source code generated based on visual examples. We develop novel techniques that augment existing a limited set of training samples via code parameterization and random variation. We also propose strategies that can adapt the general-purpose neural language model to fit the task of predicting source code.

Third, we introduce LAMVI, a set of visualization tools for diagnosing issues with neural language models. It tracks the ranks of individual candidate outputs for user-selected queries, and supports the exploration of the corresponding hidden-layer activations. It also tracks influential training instances, and provides guidance for taking actions for tuning the model. The system is evaluated on simulated datasets facilitates the user to efficiently adapt mature neural language models to new datasets or new tasks.

Collectively, these three components form an integral solution to computer-assisted problem solving for programmers driven by big data, and may have impact on various different domains, including natural language processing, machine learning, software engineering, and interactive data visualization.

CHAPTER 1

Introduction

Language is of paramount importance to human civilization. It is essential to our communication, collaboration, and knowledge accumulation. The advent of this decade's new AI spring is marked, in part, by the new success of modeling natural language—various impressive improvements have been made to speech recognition [25, 53, 113], machine translation [6, 8, 18, 24, 104, 133], and many other important fields of natural language processing by leveraging neural language models on large-scale data [92].

However, we should not ignore that, in the world today, tremendous amount of human knowledge is also being generated and stored in another form of language—*programming language*. Without this form of knowledge accumulation, the speed of innovation and technology development would be dramatically lowered. Despite its importance, programming language remains an understudied field in the research communities of machine learning and human-computer interaction. This thesis takes a joint perspective from both communities to explore the possibilities of improving the development and user experience of programming tools with an intelligence machine at the backend.

1.1 Motivation

"*The user can do this without writing a single line of code.*" The avoidance of textual programming has become the major selling point of many software applications or research projects that support data analysis, visualization, and even app development. No matter how IT technology companies try to persuade the young population that programming is easy and fun, programming remains a forbidding task to many. The reason is quite apparent—programming generally involves very tedious work [90, 157]. One has to be concentrate highly while programming, engage in an unnatural way of thinking, and patiently handle bugs or unexpected errors. Indeed, programming can be hard to learn and master.

Yet all the existing efforts to *replace* programming can only deal with the innate complexities of the problems to a certain extent. To support a task or an operation, building a graphical user interface (GUI) is more costly than creating an API. Often the most common routines are supported in GUI, and to unlock the full potential of a complex software application, the user has to touch code in one way or another [127]. Sometimes, a complex GUI routine can be replaced with few lines of code [166].

For example, to create a diverging bar chart (useful for depicting answers to Likert-style survey questions), one has to perform a series of actions in Tableau, including the selection of Gantt chart, defining percentages, starting points, and heights of each bars. For an intermediate-level programmer, writing a few lines of code in Python can get the job done in a much more efficient way. For those R users, even better, there exists a package that allows them to create such a chart by invoking a single API call.

As much as it is appealing to reduce the effort of a complex routine down to typing a single line of code, it also comes with its own price—the user first needs to *know* such an API function exists, then she also needs to know how to *use* it. While APIs often come with documentations, it takes effort for a user to read and learn them. Especially when the tasks involve customizing the outcome so that it deviates from the default option to fit the user's varying intentions.

Alternatively the user can go to social Q&A sites, such as Stack Overflow, and seek answers, but then it can still be challenging to identify the truly useful part, since it may be buried deep down in a lengthy example. For an inexperienced programmer, this may take as much time as reading through the documentation and finding the right parameter to use [59, 128].

What if the user has a programming expert to help her? It can be expected that the expert quickly recognizes the user's issues and points out that she should use which function and which parameter. Occasionally the expert may also needs to consult information via search engines. In such cases, the expert is more capable of composing appropriate search queries than the user, and is quicker to digest documentation or code examples to identify the key component of the solution.

But why is the expert able to do such things well while a novice user cannot? Since a large part of expertise is intuition, which is nothing but quickly recognizing a situation based on the pattern that may be buried in very complex appearances [41, 129]. In a sense, the expert has a better *model* (or representation) of code in the user's hand, the user's information need, and the available resources to address the issue. Because of this superior model or representation, when a new situation emerges, it is not entirely *new* to the expert, and she or he can quickly associate the situation with past problem-solving experience, and

knows a route towards the end solution based on intuition and knowledge in memory.

Reproducing such expertise in a computer system is not at all an easy task. Modern IDEs with smart code auto-completion (e.g., *intellisense*) can help programmers write code in a more efficient way, but when it comes to hard-to-tackle problems, they still need to turn to search engines. But then search engines take the user's queries, and return result pages which the user still has to go through and find answers. Social Q&A sites, like Stack Overflow, allows the users to ask and answer questions, and the discussion threads are optimized to be searched by commercial search engines. Despite the recent progress in jointly modeling code and NL text, this is not always a good experience. Wouldn't it be nice if the computer can take one step further, and help the user complete the task, or provide assistance that more directly integrates the knowledge of what is found from the Web?

The barrier that stands in between reality and our desired capabilities of intelligent systems comes in the realms of both machine learning and human-computer interaction.

Unlike human experts, the computer system has not yet had a good representation of the code, the problem, and the available solutions. While it is able to index the web pages and code examples and retrieve them fast, it is incapable of *understanding* them. Recent advances in NLP do bring intelligence to search engines, enabling them to directly answer questions related to facts, such as "*When was Lincoln born?*", or procedures, such as "*How to cook salmon in the oven?*" [89, 158]. Behind recent improvement to such intelligence, deep learning models play an important role, in creating good representations for words, sentences, and documents, and enable language models to understand the semantics of natural language [176]. However, the domain of programming language remains an understudied one, and given the structural difference between programming language and natural language, many of the technologies developed for natural language do not directly transfer to handling programming language. Relatively little research has been done on modeling programming language with neural language models, and the research on associating natural language and programming language also remain in the preliminary stage. The vast amount of data available online represents huge opportunities for research advancement in the area.

On the other hand, from the HCI perspective, relatively little work has been seen on the presentation of code search results. The presentation of search engine results has been extensively studied, but presenting code search results is rather for a different purpose—the programmer wants to integrate it into the code, and also often wants to quickly identify the most meaningful part of the code example, and know alternative solutions.

Given these opportunities, in this thesis, we introduce three research projects on com-

bining machine learning and HCI to creatively deliver new programming experience. The overall theme of the projects lies in the exploration of solutions that take the input from different modalities, and generate output that can either be directly useful to the user or provide intermediate results from which the user can continue to modify or from which the user may draw insights. We explore how machine learning models and proper interface designs can interplay to generate synergy to help users narrow down options, explore alternative options, and making the whole process transparent. While the artifacts of the research projects are directly usable and can benefit the end users, the system framework can be reused by other researchers and developers, and the design implications can be adapted to further understand the best design strategies and inspire future technologies.

The first project revolves around helping users solve programming problems in the context of the IDE. We present an intelligent programming assistant that supports interactive programming. The system employs a bimodal embedding model to encode programming language and natural language in the same vector space. We demonstrate that this model can effectively understand the code context and associate it with the user input to suggest relevant code modifications. We also develop novel user interface to render search results in a way that makes the problem solving process more efficient.

The second project aims at helping users achieve programming tasks by finding templates given image input. We propose a deep learning pipeline that converts data visualization images to source code. The pipeline is built by using computer vision techniques and recurrent neural networks, and it supports the user to get source code generated based on visual examples. We develop novel techniques that augment an existing limited set of training samples via code parameterization and random variation. We also propose strategies that can adapt the general-purpose neural language model to fit the task of predicting source code.

Our third project centers around delivering a visual interface for diagnosing issues of neural embedding models. While a lot of existing work visualizes convolutional neural networks, relatively little effort has been paid in the realm of visualizing neural language models. Some existing work does offer visualization [79], but they offer little guidance on debugging the model for a new problem, or on a new dataset. Given our accumulated experience in debugging such models, we propose a project in visualizing the neural network training process for spotting patterns for model development and debugging. We introduce a set of visualization tools for diagnosing issues with neural language models. It tracks the ranks of individual candidate outputs for user-selected queries, and supports the exploration of the corresponding hidden-layer activations. It also tracks influential training instances, and provides guidance for taking actions for tuning the model. The system is evaluated

on simulated datasets and facilitates the user to efficiently adapt mature neural language models to new datasets or new tasks.

Collectively, these three components form an integral solution to computer-assisted problem solving for programmers driven by big data, and may have impact on various different domains, including natural language processing, machine learning, software engineering, and interactive data visualization.

In the rest of this chapter, we overview the structures of the thesis, and outline our claimed contributions.

1.2 Structure of Thesis

The structure of the rest of the thesis is as follows.

Chapter 2 provides an overview of the existing neural language models, including word embedding models, recurrent networks The chapter serves as a foundation for the techniques discussed in the later sections. We also have a special focus on the intuitive understanding of the models and common practices for model development, as this relates to the creation of visual debugging interfaces for these models in Chapter 5.

The next two chapters introduce two projects that relate to assisting programming using neural language models.

Chapter 3 introduces CodeMend, an intelligent programming assistant that supports interactive programming. The backend of the assistant is implemented using a bimodal embedding model, that encodes programming language elements and natural language words in the same embedding space. We demonstrate that such models can effectively understand the code context the user is currently in, and associate the natural language input prescribed by the user with the code context in order to find relevant code modification candidates. We also introduce novel user interface with nested-layer spotlight search, which renders code search results in a way that facilitates the user's problem solving process and the exploration of alternative solutions.

Chapter 4 introduces a project that converts images to source code based on computer vision and neural language models. The project complements CodeMend in a way that supports the user to get programming examples by using a different modality for input—images. A state-of-the-art image captioning pipeline is borrowed for the task, but we show the training data set must be augmented by parameterization of the code examples and sampling from the data and parameter space. We also show that the generated source code must be modified in order to fit in the user's work flow and supports user-initiated customization.

In Chapter 5, we introduce LAMVI, an integrated visual interface for debugging neural language models. Compared to existing tools that emphasize on the overall model performance, such as perplexity and metrics of the downstream tasks, LAMVI lets the user focus on individual test cases thought of based on domain knowledge, while offering a mechanism to generate and integrate additional ground-truth use cases to prevent over-fitting. The system is evaluated by user study and is expected to facilitate the developer to efficiently optimize the model performance when it is applied to a new dataset.

We conclude with remarks on design implications and future work in Chapter 6.

This thesis contains content drawn from the following publications written with my co-authors.

- Xin Rong, Shiyan Yan, Steve Oney, Mira Dontcheva, Eytan Adar, "CodeMend: Assisting Interactive Programming with Bimodal Embedding," in UIST (2016).

- Xin Rong, Eytan Adar "Visual Tools for Debugging Neural Language Models," in ICML Workshop on Visualization for Deep Learning (2016).

1.3 Contribution

The contributions that we have made or will make throughout the projects in this thesis range from the communities of natural language processing, machine learning, to software engineering, and human-computer interaction. In specific, our contributions include:

- A novel end-to-end solution that applies a neural network model trained on a large Web-mined dataset to suggest API functions, parameters, values, or lines of code for modifying the user's code snippets to achieve their tasks expressed in natural language;

- An innovative user interface design that supports the developer to efficiently search for code editing suggestions, browse parameter values, inspect live previews, and integrate suggested modifications to their working code without leaving the IDE;

- A set of evaluations of the CodeMend system that contributes a set of insights into the ways that code search results can be effectively presented to the end-user;

- An end-to-end pipeline that converts chart images to source code for facilitating programming by visual examples;

- A set of insights into what makes code synthesis works by training recurrent neural language models conditioned on information sources from a different modality;

- A set of evaluations of conducted on systems of programming by visual examples, and insights into the usefulness of such systems in general;
- A set of visual tools for diagnosing training issues of neural embedding models, and the strategies of providing debugging guidance to model developers based on a variety of model training anomaly.

CHAPTER 2

Overview of Neural Language Models

The contributions of this thesis are made based upon the recent advancement on natural language understanding and generation. These achievements are largely empowered by the improvement of neural network language models, which have generated promising results on a variety of natural language processing tasks, including document classification, speech recognition, and machine translation. More recently, these models have been applied to programming language understanding and generation, and have also achieved promising results. This section reviews these models.

2.1 Background

To begin with, we briefly review the definition of statistical language models. Statistical language models describe the likelihood of observing a particular sequence of words in a real-world sentence, formally,

$$p(w_1, w_2, \ldots, w_N) \tag{2.1}$$

where $\{w_1, w_2, \ldots, w_N\}$ is an arbitrary sequence of words. This prediction is closely related to the task of predicting the next word given the previous words in a sentence, i.e.,

$$p(w_n|w_1, w_2, \ldots, w_{n-1}) \tag{2.2}$$

Using the chain rule, one can convert the above two tasks from one to the other:

$$p(w_1, w_2, \ldots, w_N) = \prod_{n=1}^{N} p(w_n|w_1, w_2, \ldots, w_{n-1}) \tag{2.3}$$

Language models are essential to many statistical approaches to NLP tasks, such as machine translation, speech recognition, parsing, and information retrieval. For example, in speech recognition, the phrases “I saw a van” and “eyes awe of an” have nearly identical

sounds, but using language modeling, the speech recognizer can identify the candidate with the higher likelihood.

A common approach to language modeling is to use *n*-gram models. In an *n*-gram model, Eq. (2.3) is simplified based on the Markov assumption,

$$p(w_1, w_2, \ldots, w_N) = \prod_{n=1}^{N} p(w_n | w_{n-1}, w_{n-2}, \ldots, w_{n-k}) \tag{2.4}$$

where the strength of the assumption is directly determined by k, i.e., the number of words that the probability is conditioned upon in the above equation. The strongest assumption is achieved when each word in the sentence is independent ($k = 0$). Then the model becomes a unigram model. Increasing k makes the assumption weaker. If $k = 1$, then bigram model, $k = 2$ trigarm, and hence force. Using maximum likelihood estimation (MLE), one can obtain the probability in Eq. (2.4) as follows:

$$p(w_n | w_{n-1}, w_{n-2}, \ldots, w_{n-k}) = \frac{c(w_n, w_{n-1}, w_{n-2}, \ldots, w_{n-k})}{c(w_{n-1}, w_{n-2}, \ldots, w_{n-k})} \tag{2.5}$$

where $c(\cdot)$ is the count of the word sequence in the training text corpus.

In reality, this approach of estimating *n*-gram probability suffers from the data sparsity problem. Because of the limitation of a real-world text corpus, the count, $c(w_n, w_{n-1}, w_{n-2}, \ldots, w_{n-k})$ can be zero for legitimate English phrases. This will result in zero-probability estimations for *n*-grams encountered in testing data that are unseen in the training corpus, and thus will cause the model to be unfeasible to use in practice. In addition, *n*-grams that are observed for very few times tend to also have poor probability estimations.

To account for this sparsity issue, one can apply smoothing (e.g., Laplace smoothing, Good-Turing smoothing), interpolation, or backoff (e.g., Katz backoff). These methods operate by either adding artificial counts to unseen *n*-grams or computing the probability of *n*-grams by their length-reduced versions. However, the performance of these methods may still be limited, because they still use the so-called atomic word representation, which means each word is represented by its identity.

This way of representing words loses considerable amount of information in the corpus. Consider a bigram model: if "*eat apples*" is observed many times in the corpus, but "*eat mangosteens*" (a popular South Asian fruit) is unseen, and so is "*eat computers*", then the latter two will be assigned with equal probability. Clearly this is undesired, especially if there exists other signals in the corpus that indicate "*apples*" and "*mangosteens*" are *similar*. One such signal can be that they are both often observed within the neighborhoods of the same set of words (e.g., "*grow*", "*taste*", "*fruit*"). If this signal is used in estimating

the *n*-gram probability, then each word is no longer represented by its own identity—they are represented by the contexts in which they occur.

Various language models have been proposed to build word representations based on statistics information of word contexts in a large corpus [64]. Commonly used models include hard class-based models (e.g., Brown Clustering [16]), soft class-based models—topic models (e.g., PLSA [66], LDA [13]), and network-based models [112]. Recently a family of so-called *distributed representation models*, or *word embedding models*, have gained significant popularity because of both their appealing qualities as well as their compatibility with neural network models.[1] We review these models next.

2.2 Word Embedding Models

In word embedding models, each word is represented by a set of continuous levels of activations (i.e., a vector), hence distributed representation. This is analogous to how a human neural system reacts to different words or concepts—when we think about the word "apple", a certain set of neurons are activated[2]; when we think about the word "orange", a different (but still quite similar) set of neurons are activated; but when we think about "car", quite a different set of neurons will be activated compared to the previous two words. A well-trained word embedding model should preserve such kind of patterns.

A number of word embedding models have been proposed, many of which are based on the neural language model proposed by Bengio et al. [10]. The model is a feedforward network with a linear projection layer (i.e., a hidden layer with linear activation function) and a non-linear projection layer. Follow-up work has shown this model can be further simplified and that the learned word vectors can be used to reduce the complexity of many NLP tasks and improve their performance [30, 31, 113, 114, 162]. One way of improving the quality of the word vectors even further is to train them on a larger dataset, desirably a billion-word corpus with a million-word vocabulary. But the then existing models can be hardly scaled to that level due to the high computational complexity of their training process.

To minimize this computational complexity, Mikolov et al. propose two simplified neural network models, namely the *continuous bag-of-words model (CBOW)* and the *skip-gram* (SG) model [115]. Both models are log-linear models, meaning that there is only a

[1] Word embedding models can also be considered as a kind of soft class-based models or topic models. But topic models generally take word-document co-occurrence as input, whereas word embedding models, as we shall later discuss, are factorizing a word-word co-occurrence matrix.

[2] This is a very weak analogy. The neural system in a real human brain operates in a much more complex way.

linear projection layer (without any non-linear projection layer), and the output is softmax. They show that the two models can train word vectors of a higher quality with less time. The implementations of the two models are bundled in a well-known software package, word2vec, and has gained tremendous amount of popularity over the past several years.

In the rest of this section, we review word2vec and its successors, and then we summarize the existing theoretical and empirical analyses on the relationships between neural network-based word embedding models with traditional distributional semantic models. These discussions can help us understand the reason behind the superior performance achieved by word2vec and the related neural embedding models.

2.2.0.1 Word2vec

As stated above, word2vec implements two models, continuous bag-of-words (CBOW) and the skip-gram (SG) model. The architectures of the two models are very similar. They are both log-bilinear models—a feed-forward network with a single hidden-layer that has linear activation function. The input-layer takes one-hot encoding vectors representing words as input, and the output-layer is a softmax layer, yielding a multinomial distribution among all the words in the vocabulary. What is different is the setup of the predictive task, or, how the training data are consumed. We review the two models respectively and then review the computational optimization strategies.

Continuous Bag-of-word Model (CBOW) - Figure 2.1 shows the network model under a simplified assumption, where the context is only one word, and the predictive target is also only one word—just like a bigram model.[3]

In our setting, the vocabulary size is V, and the hidden layer size is N. The weights between the input layer and the hidden layer can be represented by a $V \times N$ matrix $\mathbf{W}$. Each row of $\mathbf{W}$ is the N-dimension vector representation $\mathbf{v}_w$ of the associated word of the input layer. Formally, row i of $\mathbf{W}$ is $\mathbf{v}_w^T$. Given a context (a word), assuming $x_k = 1$ and $x_{k'} = 0$ for $k' \neq k$, we have

$$\mathbf{h} = \mathbf{W}^T \mathbf{x} = \mathbf{W}^T_{(k,\cdot)} := \mathbf{v}^T_{w_I}, \tag{2.6}$$

which is essentially copying the k-th row of $\mathbf{W}$ to $\mathbf{h}$. $\mathbf{v}_{w_I}$ is the vector representation of the input word w_I.

From the hidden layer to the output layer, there is a different weight matrix $\mathbf{W}' =$

[3] In Figures 2.1, 2.2, 2.3, and the rest of this note, $\mathbf{W}'$ is not the transpose of $\mathbf{W}$, but a different matrix instead.

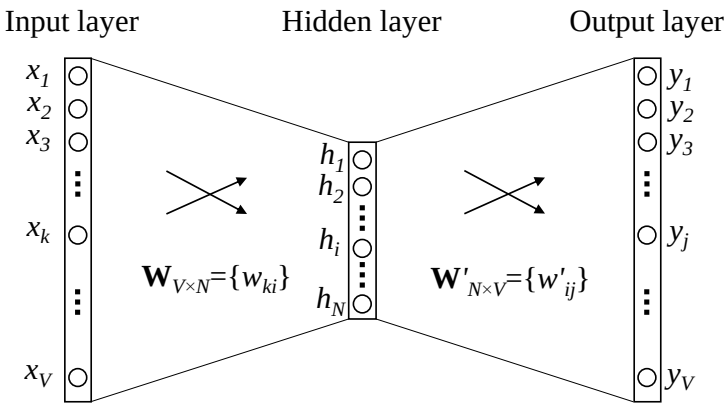


Figure 2.1: Simplified continuous bag-of-word model (CBOW) with single-word context

$\{w'_{ij}\}$, which is an $N \times V$ matrix. Using these weights, we can compute a score u_j for each word in the vocabulary,

$$u_j = {\mathbf{v}'_{w_j}}^T \mathbf{h}, \tag{2.7}$$

where $\mathbf{v}'_{w_j}$ is the j-th column of the matrix $\mathbf{W}'$. Then we can use softmax to obtain the posterior distribution of words, which is a multinomial distribution:

$$p(w_j|w_I) = y_j = \frac{\exp(u_j)}{\sum_{j'=1}^{V} \exp(u_{j'})}, \tag{2.8}$$

where y_j is the output of the j-th unit in the output layer. Substituting (2.6) and (2.7) into (2.8), we obtain

$$p(w_j|w_I) = \frac{\exp\left({\mathbf{v}'_{w_j}}^T \mathbf{v}_{w_I}\right)}{\sum_{j'=1}^{V} \exp\left({\mathbf{v}'_{w_{j'}}}^T \mathbf{v}_{w_I}\right)} \tag{2.9}$$

Note that $\mathbf{v}_w$ and $\mathbf{v}'_w$ are two representations of the word w. $\mathbf{v}_w$ comes from rows of $\mathbf{W}$, which is the input→hidden weight matrix, and $\mathbf{v}'_w$ comes from columns of $\mathbf{W}'$, which is the hidden→output matrix. In subsequent analysis, we call $\mathbf{v}_w$ as the "**input vector**", and $\mathbf{v}'_w$ as the "**output vector**" of the word w.

The training objective (for one training sample) is to maximize (2.9), the conditional probability of observing the actual output word w_O (denote its index in the output layer as j^*) given the input context word w_I with regard to the weights. We take the logarithm of

the conditional probability and use it to define our loss function.

$$\log p(w_O|w_I) = \log y_{j^*} \tag{2.10}$$

$$= u_{j^*} - \log \sum_{j'=1}^{V} \exp(u_{j'}) := -E, \tag{2.11}$$

where $E = -\log p(w_O|w_I)$ is our loss function (we want to minimize E), and j^* is the index of the actual output word in the output layer. Note that this loss function can be understood as a special case of the cross-entropy measurement between two probabilistic distributions.

Figure 2.2 shows the CBOW model with a multi-word context setting. When computing the hidden layer output, instead of directly copying the input vector of the input context word, the CBOW model takes the average of the vectors of the input context words, and use the product of the input→hidden weight matrix and the average vector as the output.

$$\mathbf{h} = \frac{1}{C}\mathbf{W}^T(\mathbf{x}_1 + \mathbf{x}_2 + \cdots + \mathbf{x}_C) \tag{2.12}$$

$$= \frac{1}{C}(\mathbf{v}_{w_1} + \mathbf{v}_{w_2} + \cdots + \mathbf{v}_{w_C})^T \tag{2.13}$$

where C is the number of words in the context, $w_1, \cdots, w_C$ are the words the in the context, and $\mathbf{v}_w$ is the input vector of a word w. The loss function is

$$E = = -\log p(w_O|w_{I,1}, \cdots, w_{I,C}) \tag{2.14}$$

$$= -u_{j^*} + \log \sum_{j'=1}^{V} \exp(u_{j'}) \tag{2.15}$$

$$= -\mathbf{v}'_{w_O}{}^T \cdot \mathbf{h} + \log \sum_{j'=1}^{V} \exp(\mathbf{v}'_{w_j}{}^T \cdot \mathbf{h}) \tag{2.16}$$

which is the same as (2.11), the objective of the one-word-context model, except that $\mathbf{h}$ is different, as defined in (2.13) instead of (2.6).

Skip-gram Model - The skip-gram model is introduced in [115, 116]. Figure 2.3 shows the skip-gram model. It is the opposite of the CBOW model. The target word is now at the input layer, and the context words are on the output layer.

We still use $\mathbf{v}_{w_I}$ to denote the input vector of the only word on the input layer, and thus we have the same definition of the hidden-layer outputs $\mathbf{h}$ as in (2.6), which means $\mathbf{h}$ is simply copying (and transposing) a row of the input→hidden weight matrix, $\mathbf{W}$, associated

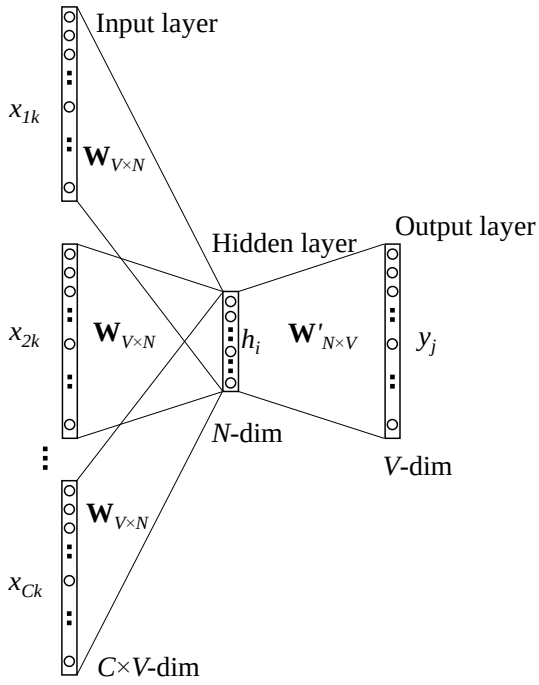


Figure 2.2: Continuous bag-of-word model (CBOW)

with the input word w_I. We copy the definition of $\mathbf{h}$ below:

$$\mathbf{h} = \mathbf{W}^T_{(k,\cdot)} := \mathbf{v}^T_{w_I}, \tag{2.17}$$

On the output layer, instead of outputting one multinomial distribution, we are outputing C multinomial distributions. Each output is computed using the same hidden→output matrix:

$$p(w_{c,j} = w_{O,c}|w_I) = y_{c,j} = \frac{\exp(u_{c,j})}{\sum_{j'=1}^{V} \exp(u_{j'})} \tag{2.18}$$

where $w_{c,j}$ is the j-th word on the c-th panel of the output layer; $w_{O,c}$ is the actual c-th word in the output context words; w_I is the only input word; $y_{c,j}$ is the output of the j-th unit on the c-th panel of the output layer; $u_{c,j}$ is the net input of the j-th unit on the c-th panel of the output layer. Because the output layer panels share the same weights, thus

$$u_{c,j} = u_j = {\mathbf{v}'_{w_j}}^T \cdot \mathbf{h}, \text{ for } c = 1, 2, \cdots, C \tag{2.19}$$

where $\mathbf{v}'_{w_j}$ is the output vector of the j-th word in the vocabulary, w_j, and also $\mathbf{v}'_{w_j}$ is taken from a column of the hidden→output weight matrix, $\mathbf{W}'$.

For both CBOW and skip-gram models, the training is done by backpropagation with

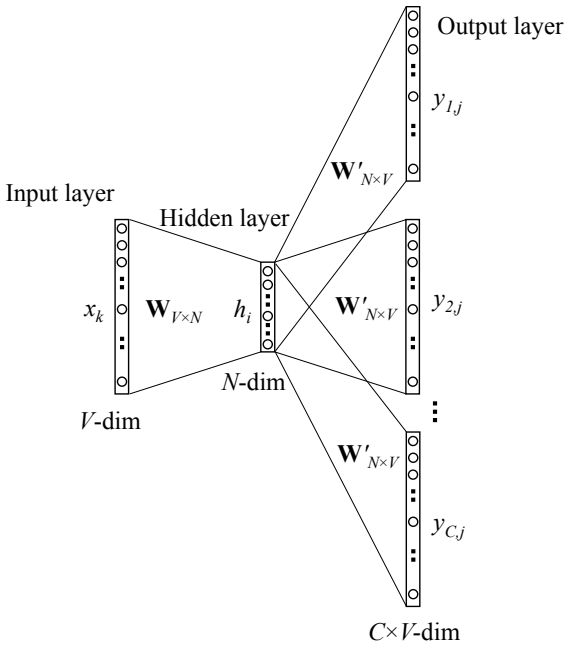


Figure 2.3: Skip-gram model

the stochastic gradient descent (SGD) algorithm. Learning the input vectors is cheap; but learning the output vectors is very expensive. The output vector of each word in the vocabulary has to be updated for each training instance, making it impractical to scale up to large vocabularies or large training corpora. To solve this problem, an intuition is to limit the number of output vectors that must be updated per training instance. Hierarchical softmax and negative sampling are two of the strategies to solve this. Both tricks optimize only the computation of the updates for output vectors.

Hierarchical Softmax - Hierarchical softmax is an efficient way of computing softmax [123, 122]. The model uses a binary tree to represent all words in the vocabulary. The V words must be leaf units of the tree. It can be proved that there are $V-1$ inner units. For each leaf unit, there exists a unique path from the root to the unit; and this path is used to estimate the probability of the word represented by the leaf unit. See Figure 2.4 for an example tree.

In the hierarchical softmax model, there is no output vector representation for words. Instead, each of the $V-1$ inner units has an output vector $\mathbf{v}'_{n(w,j)}$. And the probability of a word being the output word is defined as

$$p(w=w_O) = \prod_{j=1}^{L(w)-1} \sigma\left(\llbracket n(w,j+1) = \text{ch}(n(w,j)) \rrbracket \cdot {\mathbf{v}'_{n(w,j)}}^T \mathbf{h}\right) \tag{2.20}$$

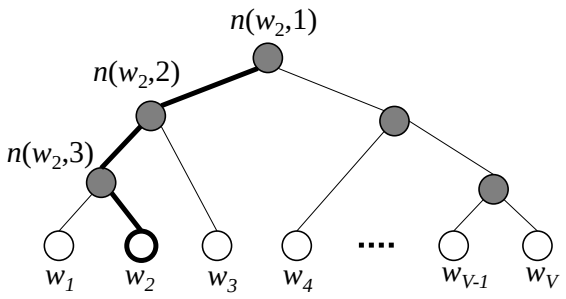


Figure 2.4: An example binary tree for the hierarchical softmax model. The white units are words in the vocabulary, and the dark units are inner units. An example path from root to w_2 is highlighted. In the example shown, the length of the path $L(w_2) = 4$. $n(w, j)$ means the j-th unit on the path from root to the word w.

where ch(n) is the left child of unit n; $\mathbf{v}'_{n(w,j)}$ is the vector representation ("output vector") of the inner unit $n(w, j)$; $\mathbf{h}$ is the output value of the hidden layer (in the skip-gram model $\mathbf{h} = \mathbf{v}_{w_I}$; and in CBOW, $\mathbf{h} = \frac{1}{C}\sum_{c=1}^{C} \mathbf{v}_{w_c}$); $[\![x]\!]$ is a special function defined as

$$[\![x]\!] = \begin{cases} 1 & \text{if } x \text{ is true;} \\ -1 & \text{otherwise.} \end{cases} \tag{2.21}$$

Negative Sampling - The idea of negative sampling is more straightforward than hierarchical softmax: in order to deal with the difficulty of having too many output vectors that need to be updated per iteration, we only update a sample of them.

The output word (i.e., the ground truth, or positive sample) should be kept in our sample and gets updated, and we need to sample a few words as negative samples (hence "negative sampling"). A probabilistic distribution is needed for the sampling process, and it can be arbitrarily chosen. We call this distribution the noise distribution, and denote it as $P_n(w)$. One can determine a good distribution empirically.[4]

In word2vec, instead of using a form of negative sampling that produces a well-defined posterior multinomial distribution, the authors argue that the following simplified training objective is capable of producing high-quality word embeddings:[5]

$$E = -\log \sigma({\mathbf{v}'_{w_O}}^T \mathbf{h}) - \sum_{w_j \in \mathcal{W}_{\text{neg}}} \log \sigma(-{\mathbf{v}'_{w_j}}^T \mathbf{h}) \tag{2.22}$$

[4] As described in [116], word2vec uses a unigram distribution raised to the $\frac{3}{4}$th power for the best quality of results.

[5] [50] provide a theoretical analysis on the reason of using this objective function.

where w_O is the output word (i.e., the positive sample), and $\mathbf{v}'_{w_O}$ is its output vector; $\mathbf{h}$ is the output value of the hidden layer: $\mathbf{h} = \frac{1}{C}\sum_{c=1}^{C}\mathbf{v}_{w_c}$ in the CBOW model and $\mathbf{h} = \mathbf{v}_{w_I}$ in the skip-gram model; $\mathcal{W}_{\text{neg}} = \{w_j | j = 1, \cdots, K\}$ is the set of words that are sampled based on $P_n(w)$, i.e., negative samples.

Besides the computational optimization methods, other detailed measures have also been shown to be very important [96]. These include: sub-sampling frequent words; random shrinking of window size (implicitly assigning higher weights to contextual words that are closer to the target word); the initialization of the word vectors, the learning rate, and the decay of learning rates are important as well.

The trained vectors of word2vec have been demonstrated to possess appealing qualities. In addition to achieving superior performances in NLP tasks stated earlier, a well-received quality is that one can perform word analogies using these vectors. For example, if one takes the vector of "*king*", subtracts it by "*man*", and adds the vector of "*woman*", then the resulted vector is closest to the vector of "*queen*". Therefore, the learned vector can encode the *analogy* using algebraic equation king-queen=man-woman. In a sense, the gender and/or royalty relations are encoded in the vector dimensions. Other relations can be encoded in a similar way, such as capital city and country, or, morphological relations between words (e.g., he-his=she-her). The authors of word2vec propose word analogy as a way of evaluation. Note that earlier models can also achieve similar analogy results [117].

2.2.0.2 GloVe

Global Vectors for Word Representation (GloVe), proposed by Pennington et al. [135], is a commonly used substitute of word2vec. While word analogy is not explicitly encoded in the objective function of word2vec, it is specially included in the training objective of GloVe, which is defined as

$$J = \sum_{i,j=1}^{V} f(X_{ij})(w_i^T w'_j + b_i + b'_j - \log X_{ij})^2 \tag{2.23}$$

where V is the number of words in the vocabulary; w_i and w_j are the word vectors of two words indexed i and j; b_i and b'_j are "biases" specially defined for the two words, which are useful in providing extra flexibility in reconstructing the PMI; and X_{ij} is the co-occurrence count between words w_i and w_j, or more precisely, the number of times word w_i appears in the window of word w_j; $f(X)$ is a "clipping" function that is designed to

prevent "spamming" of over-popular co-occurrence pairs. $f(X)$ is defined as:

$$f(x) = \begin{cases} (x/x_{\max})^{\alpha} & \text{if } x < x_{\max} \\ 1 & \text{otherwise.} \end{cases} \tag{2.24}$$

where α is empirically assigned to be 0.75, and $x_{\max}$ is empirically chosen to be 100 by the authors.

The intuition of the training objective is that word embedding vectors should be aimed to preserve the relative similarity rather than absolute frequency.

2.2.0.3 Understanding Embedding Models

The original papers presenting word2vec offer few explanations as to why the performance is desirable. Several pieces of follow-up work employ both theoretical and empirical analyses to understand why word embedding models can generate desirable results.

First, people want to verify whether neural embedding models in fact outperform traditional methods under more comprehensive and scrutinized comparison. Baroni et al. [9] compare word embedding models (prediction-based models) with traditional distributional semantic models (DSM, or, count-based models), including latent semantic analysis (LSA) [36] or applying non-negative matrix factorization to word co-occurrence matrices. Their experiment result indicates that prediction-based models are superior to count-based models in all benchmarks, including correlating with human-labeled semantic relatedness between word pairs, detecting synonyms, categorizing words into concepts, and word analogy.

But this is hardly the whole picture. Levy et al. [95] reveal that neural word embedding models are in fact deeply linked to matrix factorization methods. They show that the skip-gram model with negative sampling (SGNS) in word2vec is in fact factorizing a matrix of co-occurrence measures between context and target words. They show that the measure is point-wise mutual information (PMI) shifted by a constant. Levy et al. also demonstrate that traditional distributional semantic models (DSMs) can achieve comparable performance as word2vec, for various benchmarks including word analogy tasks, when certain hyperparameters are properly chosen, esp. the co-occurrence metrics that are used to generate the matrix that is to be factorized [96].

While demythifying the "superior" performance achieved by neural embedding models, these papers also provide explanations as to what each carefully calibrated system design decision in a word embedding model corresponds to from a matrix factorization point of view. For example, adjusting the number of negative samples correspond to the magnitude

of the constant in shifted PMI; negative sampling from a noise distribution with 0.75 power corresponds to smoothing the context distribution, thus alleviating the bias caused by rare words.

2.2.0.4 Recent Advancement

Since the introduction of word2vec and GloVe, word embedding methods have been extensively explored and expanded. One area of improvement that draws a great amount of attention is the embedding of higher-level linguistic structures, such as phrases, sentences, paragraphs, or documents. Since the basic neural embedding models operate on the word level, in order to get the representation of a sequence of words, one needs to take the sum or mean of the word vectors in the sequence. This approach, in fact, is still an bag-of-word approach, albeit a continuous one instead of discrete. Le et al. [91] show that by adding a *paragraph vector* as context in the word prediction task, one can achieve improved performance on document classification and sentiment analysis tasks.

The methods we have reviewed so far use atomic word representation in the training input of the word embedding model. However, subword information can also be helpful, since common character sequences shared by similar words can be used to enhance the similarity modeling or learning vectors about rare words. The fastText [14, 75] system uses sub-word information to enhance word embedding training. This model also innately supports document classification. In standard tasks, like sentiment analysis and document classification, fastText has been shown to outperform word2vec and other competitors.

2.2.0.5 Adapting Embedding to Other Data Modals

Embedding models have also been adapted to model data in other domains other than language. Mainly this is achieved by either framing the target problem as matrix factorization, or as some metaphor of "sentences." DeepWalk [136] embeds nodes of a network into lower dimensional space by treating random walks on the network as "sentences" and directly applies word2vec for learning. They show that the learned vector representations can be used for network classification that outperforms competing baselines by up to 10% when labeled data is sparse. LINE [160] targets at the same task using a different approach. They consider first-order co-occurrence and second-order co-occurrence between nodes and devise skip-gram-style objective functions to preserve such metrics directly. They show that vectors learned by LINE outperform word2vec on tasks like document classification, when considering word co-occurrence network. LINE can also be applied to other types of networks.

Dalvi et al. [34] develop an embedding model for entities in semi-structured data on the web. The data can be columns in a web table, or, hyponym pairs derived from Hearst patterns, like "A such as B." They demonstrate that the embedding model can be effectively applied to tasks like entity set expansion [146] and entity class prediction.

In summary, we have reviewed word2vec, GloVe, several related theoretical and empirical analyses, and the latest advancement of word embedding models. While word embedding models themselves are useful in many ways, they have limited utility in modeling sequences—one cannot retain sequential information by taking the mean or sum of all the words in a sequence. For sequence-processing pipelines like machine translation, one needs to use recurrent neural networks (RNN) to model the dependencies of words in sequence. We review RNN models in the next section.

2.3 Recurrent Neural Networks

A recurrent neural network (RNN) is a powerful tool to encode sequences. It can be used to predict the next word in a sentence given all the previous context, where the context is encoded as its hidden state.

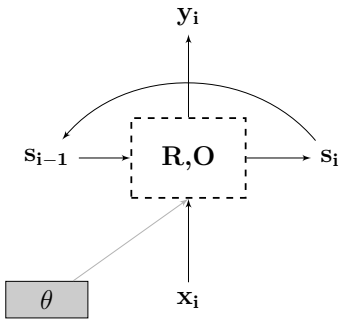


Figure 2.5: Recurrent neural network. Adapted from [49].

The vanilla form of an RNN model (Figure 2.5) is almost identical to a feed-forward neural network with one hidden layer, except that its hidden-layer value s_i is computed not only based on the current input x_i, but also the hidden-layer value of the *last time stamp*, s_{i-1}, i.e.,

$$s_i = R(s_{i-1}, x_i) = g(x_i W^x + s_{i-1} W^s + b) \tag{2.25}$$

where W^x and W^s are weight matrices; b is a bias term; and $g(\cdot)$ is a non-linear activation function. Common choices for g include hyperbolic tangent function (tanh) and rectifier linear unit (ReLU).

Intuitively, the hidden-layer value serves as the *memory*, i.e., it encodes all the past words encountered during a sentence, and this, combined with the current input x_i, is then used to predict the next output y_i,

$$y_i = O(s_i) = s_i \tag{2.26}$$

The inputs are one-hot encoded word vectors; and the output layer is often a softmax, that is

$$p(e = j|x_{1:i}) = softmax(y_i W + b)[j] \tag{2.27}$$

Note that the LHS of the above equation is equivalent to Eq. (2.2).

An RNN that consumes a finite-length input can be unrolled over time (Figure 2.6) for easier interpretation of its feedforward and backpropagation computations. Once unrolled, it is easy to see that an RNN is essentially equivalent to a multi-layer (*deep*) feed-forward network, except that the weights on the links are shared across all time steps. This attribute of shared weight will impact the training (in particular, gradient computation) as we shall discuss later.

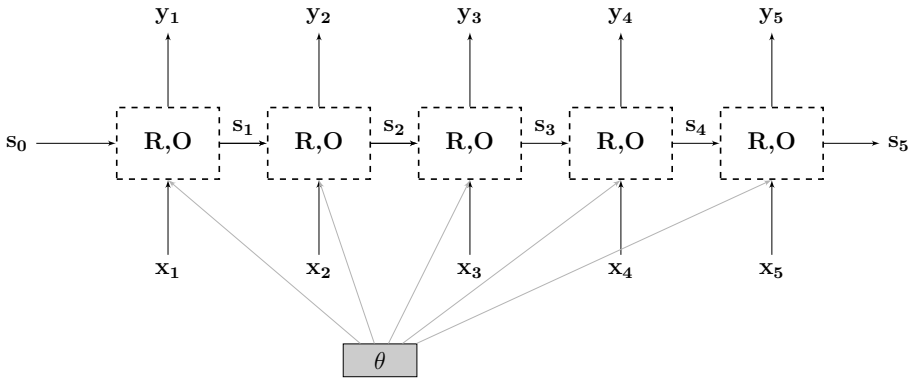


Figure 2.6: Recurrent neural network unrolled over time. Adapted from [49].

RNNs can be stacked to form a deep structure—added layers of non-linearity can achieve a higher level of expressiveness. RNNs are commonly trained by *backpropagation through time* using mini-batch gradient descent algorithm. The learning rate can be updated adaptively using Adagrad [42], RMSProp [35], Adam [81], etc.

In theory, an RNN can encode dependency (i.e., memorize context) of arbitrary length, but in practice, due to the *gradient explosion* and *gradient vanishing* problem, which happens to all deep network structures, its capability of encoding long-range dependencies is limited. Therefore, RNNs are said to have only a short-term memory. While gradient explosion can be addressed by gradient clipping during backpropagation, gradient vanishing

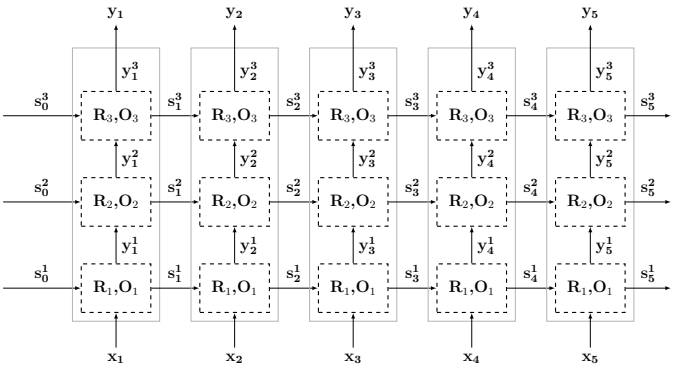


Figure 2.7: A stacked RNN with 3 layers. Adapted from [49].

is not as easily resolved. The consequence is similar to a Markov assumption of the *n*-gram model, except that there is not a clear cut-off point.

The *long short-term memory network* (LSTM) is designed to alleviate the long-range dependency problem by adding an additional memory unit (called an *LSTM cell*) that allows selective access. The computation of the hidden-layer value is changed to:

$$s_j = R(s_{j-1}, x_j) = [c_j; h_j] \tag{2.28}$$

$$c_j = c_{j-1} \odot f + g \odot i \tag{2.29}$$

$$h_j = \tanh(c_j) \odot o \tag{2.30}$$

where $\odot$ means element-wise product. i, f, and o are three gates, namely input gate, forget gate, and output gate. They are soft (instead of binary) switches that control the access to the memory unit c. The gates have values between 0 and 1, and are calculated based on the input x_j and the hidden-layer value of the previous time step h_{j-1},

$$i = \sigma(x_j W^{xi} + h_{j-1} W^{hi}) \tag{2.31}$$

$$f = \sigma(x_j W^{xf} + h_{j-1} W^{hf}) \tag{2.32}$$

$$o = \sigma(x_j W^{xo} + h_{j-1} W^{ho}) \tag{2.33}$$

where $\sigma(\cdot)$ is the sigmoid function. g is the "update candidate", a value that can be potentially (controlled by input-gate i) used to update c_j. It is given by

$$g = \tanh(x_j W^{xg} + h_{j-1} W^{hg}) \tag{2.34}$$

In summary we have briefly reviewed the vanilla RNN architecture and its LSTM version. There are a variety of techniques that improve the performance based upon these basic

models, such as bidirectional RNN [150] and Gated Recurrent Unit (GRU) [24], which are beyond the scope of this review.

CHAPTER 3

Assisting Interactive Programming with Bimodal Embedding

3.1 Overview

Modern programming libraries present complex APIs that are both powerful and potentially overwhelming. Python packages such as *matplotlib* and *pandas* are used in "traditional" programming environments but have also become critical for *interactive computing environments*. These environments enable end-users to perform data analysis in instantaneous read-eval-print loops (REPL). Driven by the demand of data scientists and analysts, interactive environments such as IPython/Jupyter Notebooks, Mathematica, and R, have grown in popularity [51]. Though in some ways programming in interactive environments is "easier"—with relatively short code blocks and architectures—they are nonetheless difficult to learn and master due to the significant scale of functions and parameters provided through the APIs. Additionally, end-users for interactive environments are broader than professional programmers and the development environments themselves are often less feature-rich. Search engines may help, but using them successfully still demands a great amount of knowledge and skill. Specifically, the end-user must formulate the query, identify good matches, interpret the APIs' use in example code, and correctly integrate this information into her own code.

Take a user who is using a graphing library to generate a bar-chart but doesn't like the default location of the legend. She searches for: "*move the legend in Bokeh*".[1] She is likely to find lengthy documentation of the library, or possibly a long example where the correct function and parameter is buried inside. If she is really lucky (or an experienced searcher), she may find an answer on Stack Overflow that clearly matches the question and describes the function and parameter, but even then she would need to figure out where exactly in

[1] Bokeh is an increasingly popular Python plotting library.

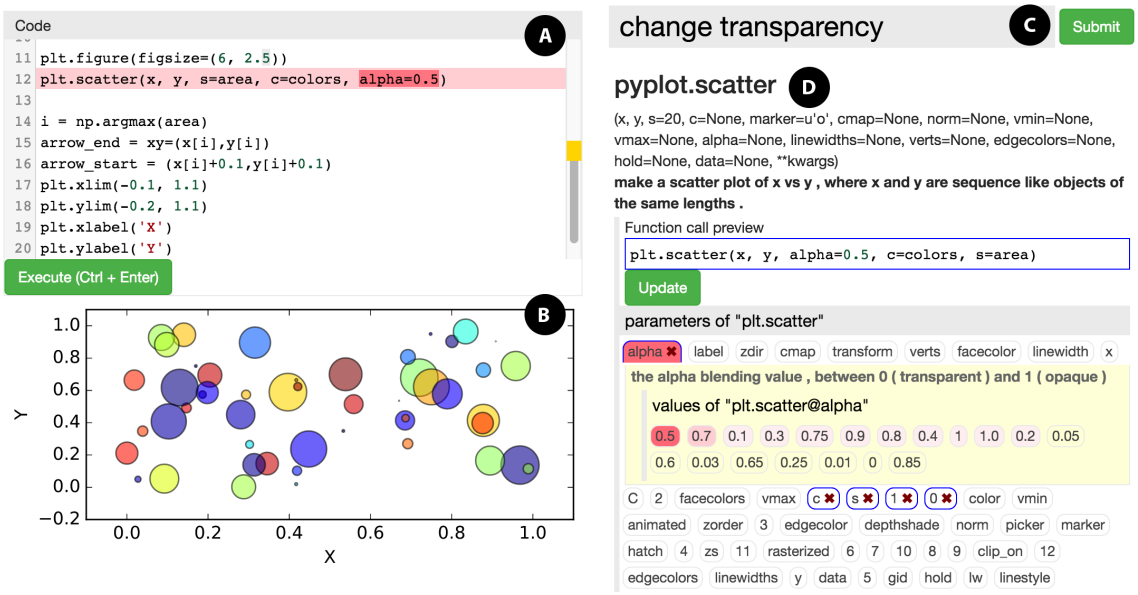


Figure 3.1: CodeMend system screenshot: (A) code editor: highlights relevant lines and columns based on the user's natural language query; (B) preview window: shows the image generated by the code; (C) query box: allows the user to type in natural language queries; (D) multi-function information box: provides code summary and nested-layer suggestions of possible code modifications.

her code the function or parameter should go. We propose instead that IDEs should short-circuit this and provide *in-situ* code modification recommendations in response to natural language (NL) queries. Suggestions should be contextualized by inspecting the end-user's current code context and supported by a large database of code examples.

In this chapter, we present CodeMend, an intelligent assistant for interactive programming environments. CodeMend is aware of what code the end-user has already written and can respond to their NL requests directly in the context of that code. For the example above, the system can understand that the user has already created a legend, and can associate to a large collection of code examples and understand what people usually do after creating a legend. By combining this contextual information with the user's query, "*move the legend,*" CodeMend can more precisely recommend the correct function and parameter than a search engine would. Even better, CodeMend not only indicates to the user *where* in the code the modification should be made by highlighting the relevant lines and parameters, but also displays the *suggested code changes* through a novel interactive UI.

To build CodeMend, we adopt a neural network model, namely a *bimodal embedding model* [2], to jointly model code and NL. This model is based on n-gram representations of code and NL. It learns the distributed representations of code and words in the same vector space by consuming large text and code corpora. In CodeMend, the corpora include

data ranging from API documentations, Stack Overflow pages, GitHub repositories, and other webpages. After training, the model can be applied to different tasks, including code prediction, code captioning, as well as the primary function of CodeMend, contextualized code modification suggestion based on NL queries.

We trained CodeMend's first model on *matplotlib* [72], a popular Python library for plotting scientific figures, that is frequently used in IPython or Jupyter Notebook environments. We targeted *matplotlib* initially as it is representative of a broad set of additional libraries (e.g., *numpy*, *pandas*, *networkx*, and *scikit-learn*). To evaluate CodeMend, we tested our model against a set of collected user queries, demonstrating that our model can accurately understand a significant portion of the queries while handling many instances of vocabulary mismatches. We also conducted a user study with the full CodeMend interface to show an improvement in end-user productivity.

CodeMend contributes a novel end-to-end solution that applies a neural network model trained on a large Web-mined dataset to suggest API functions, parameters, values, or lines of code for modifying the user's code snippet to achieve their tasks expressed in NL. Beyond the "back-end" models, CodeMend provides an innovative UI design that supports the developer to efficiently search for code editing suggestions, browse common parameter values, inspect live previews, and integrate suggested modifications to their working code without leaving the IDE. Finally, our evaluation of the system contributes a set of insights into the ways that code search results can be effectively presented to the end-user.

3.2 Related Work

We briefly touch upon related solutions including both UI-focused approaches as well as novel "code mining" backends.

3.2.1 Context-based Code Search and Code Synthesis

Code-search solutions have long-existed to support the needs of developers [7, 80, 83, 101, 110]. Though many search engines are context-free, some have leveraged the end-user's current code to enhance search performance [15, 36, 67, 119, 173]. CodeBroker [173], for example, uses comments and function definitions in the user's current code to match code examples. Strathcona [67], PRIME [119], and SWIM [139] extract class types and function calls as context. Some systems go further by helping adapt and integrate found code to the user's current code [60, 131, 168]. A few solutions can even *synthesize* new code blocks unseen by the system [21, 46, 57, 102, 139, 144]. To this end, the local variables defined in

the user's current code (and even their runtime values) are leveraged, which offers greater flexibility in terms of adapting the code example to the current codebase and helping the user navigate through complex APIs [106, 147, 161]. While most of these existing systems present results as a ranked list of synthesized code snippets or directly modify the user's code, CodeMend takes an innovative approach that directs the user's attention to the part of the code that is most relevant to the query, and uses a nested-layer spotlight search interface to help the user select suggested code changes. To the best of our knowledge, CodeMend is unique in offering an end-to-end solution that combines mining massive Web resources, joint modeling of text and code, support for long-tail NL queries, and a novel UI to present code modifications with interactive visualizations.

3.2.2 Associating Code with NL

Many code-search systems use keyword matching to process NL queries [21, 80, 109]. Although keyword matching can be effective in catching lexical features (e.g., comments, variable and function names), it fails when there is a vocabulary mismatch between the query terms and the indexed terms. To address this issue, a number of systems employs query expansion [58, 110, 139] or topic models [7, 173]. For example, AnyCode [58] uses WordNet [118] to perform query expansion, while SWIM [139] leverages a proprietary commercial search engine's click-through logs. Broker [173] and SSI [7] use variants of Latent Semantic Analysis (LSA) [36]. In comparison, CodeMend uses a neural embedding model, which has several advantages: it can be trained on openly available domain-specific corpora and thus is not limited by the coverage of WordNet or proprietary data; it can be trained more efficiently than topic models; and it can easily consume a larger amount of data and gain better performance.

A second line of research focuses on synthesizing programs that perform small and repetitive tasks (e.g., text editing) based on NL instructions [38, 108, 143, 174]. These systems can achieve very high accuracy in composing domain-specific programs but have relatively strict requirement on the syntax of the NL query. In comparison, CodeMend focuses on handling more open-ended task expressions.

3.2.3 Statistical Code Modeling

Statistical code modeling, or *big code analysis* [142], captures regularities in a code corpus and distills useful knowledge about APIs or the underlying programming language [12, 63, 130]. While such models are often used to enhance applications like plagiarism detection [68] or code completion [141], they can also be used to enhance the modeling of code

context for suggesting code modifications as in CodeMend.

A popular choice for code modeling are *n*-gram models. Hsiao et al. [68] use *n-grams* of code tokens to represent a program, and show that *tf-idf* (term frequency–inverse document frequency) weights, a common NL corpus statistic, can effectively improve code similarity measurement (leading to better performance in plagiarism detection). SLANG [141] leverages an *n*-gram model of API call sequences, and allows the user to write programs with placeholders which the system will automatically fill in with appropriate API calls. While CodeMend also leverages the *n*-gram representation to model the code context, our objective is different and our model is also dependent on the NL context.

3.2.4 Distributed Representation Models

Neural embedding models that learn distributed representations (vectors) of NL words have recently gained a great amount of attention [91, 115, 116]. Such models are fast to train and can take advantage of large-scale unlabeled training data. They are shown to be able to learn representations of words that carry deep semantics [116].

Several recent studies adapt the embedding approach to modeling tasks [1] and programs [2, 134, 137]. Our model is inspired by Allamanis et al. [2]. Their solution frames the process of code generation as searching for plausible children tuples to be attached to a partially grown abstract syntax tree (AST). The searching is contextualized based on the bag-of-word representation of an NL utterance. As a result, their model supports code retrieval by NL and also caption generation for code snippets. Our model stems from this approach, but is based on a simpler code representation and employs a novel method to generate training data from code examples, so that the model can learn plausible code modifications.

Other work on recurrent neural networks (RNNs) and convolutional neural networks (CNNs) also show promising results when adapted to modeling programming language [77, 124]. CodeMend may benefit from these techniques in the future.

3.2.5 Exploratory Programming Interfaces

Exploratory programming, or live programming, is a paradigm that is centered around the REPL experience [62] Mathematica, MATLAB, R, IPython, and Jupyter Notebook are all examples in this space. Recent work has focused on augmenting these tools to support collaboration among data scientists. For example, Tempe [45] is an integrated system that supports collaborative analysis on temporal and streaming data via REPL experience. The system can manage persistent research notebooks and make the results of analysis more re-

producible than tools like Jupyter Notebooks by tracking workflow provenance. Although CodeMend has a similar motivation—augmenting the REPL experience for data scientists in general—our focus is on optimizing individual users' experience of navigating through complex APIs.

3.3 System Overview

3.3.1 Sample User Experience

Figure 3.1 displays the interface of our system. Suppose Alice wants to create a scatter plot. She opens CodeMend with an empty editor. The system displays a list of functions that are likely to be called first (Figure 3.2). Although she can browse the suggested functions, Alice types in a search query "*create a scatter plot.*" As she types each word, the system updates the ranked list of functions, while using color to encode the likelihood of each candidate. When she stops typing, she finds that `scatter` is highest ranked, and clicks on it. CodeMend shows a set of previews of scatter plots generated by different code examples mined from the Web, ranked by the simplicity (length) of the code. Alice clicks on one of them and the system populates the editor with its code. Alice can then replace the dummy data variables in the code with her real data.

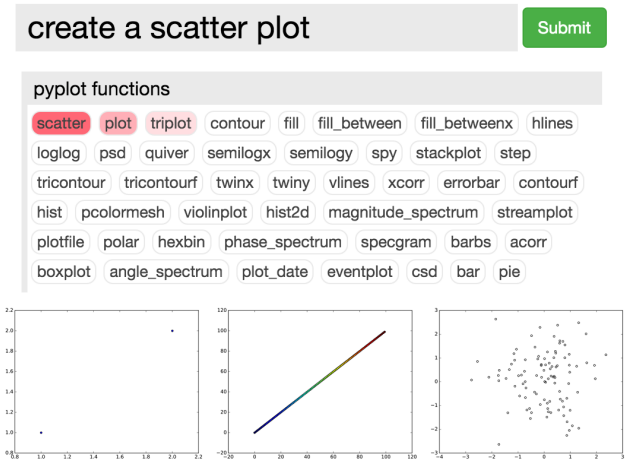


Figure 3.2: CodeMend showing a ranked list of suggested functions based on the query "*create a scatter plot.*" Clicking on a suggested function will make the system display a number of images generated by the code examples using the function.

Unfortunately, the look of the chart still isn't quite right. Alice wants to change the transparency of the points on the plot, but she does not know how to do this. She presses ESC to focus on the query box, and types a query: "*change transparency of the points.*" As

Alice types, the system computes how likely each line is associated with this modification task and highlights the line with the highest likelihood. In this case, it is the line that has the `plt.scatter()` function (Figure 3.1 (A)). After she clicks on the line, the system shows a suggestion box with a ranked list of parameters (Figure 3.1 (D)), and the top ranked one is `alpha`. She clicks on the parameter and the API documentation of the parameter shows up, as well as a number of example values for this parameter. These values are extracted from tens of thousands of code examples online, and are ranked by the frequency of their usage. Alice clicks on a suggested value `0.5`, and CodeMend updates the code and the image preview immediately. Alice checks the image and is satisfied with the change.

In summary, CodeMend has supported Alice's editing process via: (1) query-dependent code highlighting; (2) multi-layer suggestion (functions, arguments, and values); and (3) instant preview of the effects of code changes.

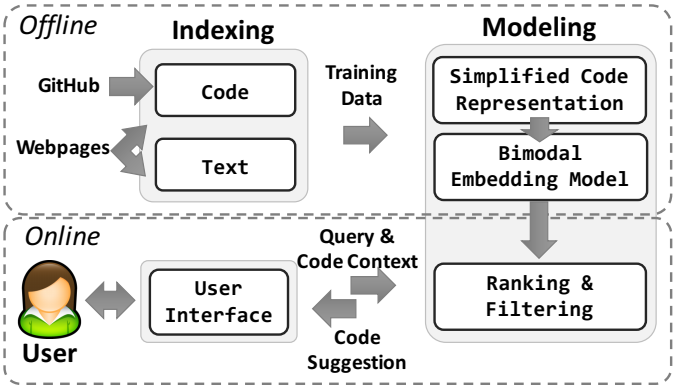


Figure 3.3: Overview of the CodeMend architecture

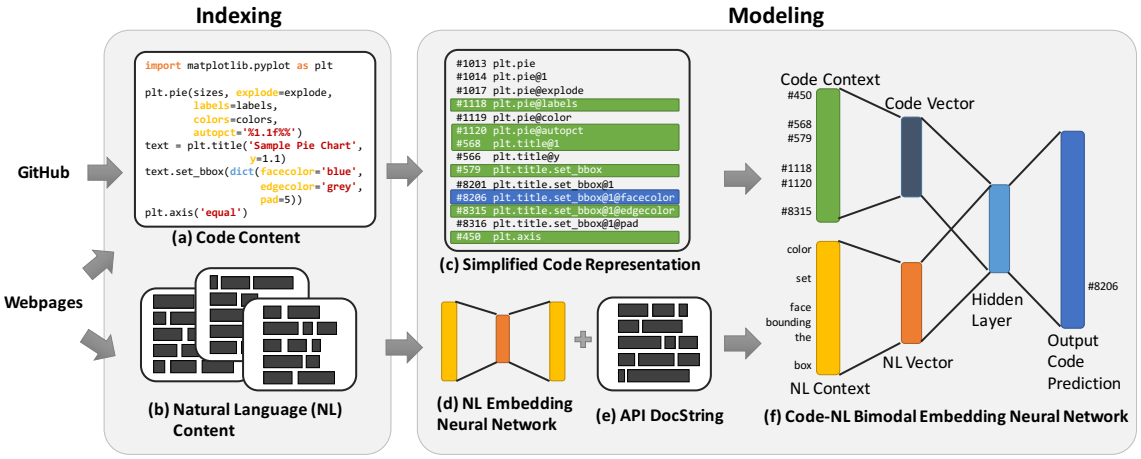


Figure 3.4: Pipeline for training a bimodal embedding model

3.3.2 System Architecture

CodeMend contains three major components: an indexing module, a modeling module, and a front-end user interface component (Figure 3.3). The indexing module crawls GitHub repositories and webpages (such as Stack Overflow discussion pages), to collect code examples and NL text content. This data is then fed into a bimodal embedding model, which is a statistical model that can automatically learn the associations between code and NL and handle vocabulary mismatch by scanning large collections of data in both modes. Once training is complete, the back-end model is used to provide code suggestions. It takes both code context and the user query as input, and generates ranked lists of code elements (functions, parameters, and parameter values) for the front-end to display. While the user only interacts with CodeMend's UI, she benefits from the knowledge extracted from a large collection of Web resources. Below, we detail each of the CodeMend's components.

3.4 Data Preparation

To train a neural embedding model to learn high-quality representations, one has to supply an extensive amount of data. In our case, we need both a large collection of code examples that *use* matplotlib and a large text corpus that *talks about* the same library. To create a natural language corpus with reasonably good coverage on how developers describe tasks and problems in Python programming in general, we extracted all the NL content from Stack Overflow threads that were tagged "Python". This resulted in a corpus of 90 million words extracted from 397,197 threads. All the text content was tokenized, lower-cased, and lemmatized.

To collect *code examples*, we searched GitHub for repositories that contained the string "matplotlib", and cloned the first 1,000 repositories in the search results. We extracted code from Python source files as well IPython Notebooks. After discarding duplicates and unparseable files, we obtained 8,732 useful code examples. We also downloaded the entire Stack Overflow data dump[2] and extracted code examples from positively rated answers in all the threads tagged "matplotlib", which resulted in an extra 15,570 code examples. To obtain more code, we prepared a list of 1,428 queries using frequent keywords of matplotlib and appended "matplotlib" to the end of each query (e.g. "*Axes matplotlib*"). We submitted these queries to the Yahoo! Boss Search API and collected 38,590 URLs. Using these URLs and their one-hop outgoing links, we retrieved a total of 1,921,890 webpages. We then built a classifier using character sequences as features to identify the code examples

[2]https://archive.org/details/stackexchange

in these webpages. This step gave us 99,424 code examples, of which 21,993 were useful (examples shorter than 3 lines were discarded). We also included all the code examples of a textbook about matplotlib [39]. In total, we collected 46,495 useful code examples.

In addition to training the model, the code examples are also used to generate sample plots of functions (see Figure 3.2). For this purpose, we consulted the documentation of matplotlib, and identified 47 functions that can directly generate figures (as opposed to adjusting a figure). For each function, we found the code examples that used the function and filtered out those that failed to execute. We ranked the remaining code examples by simplicity (measured by number of characters in code), and kept at most 20 shortest code examples per function. Finally, a total of 405 code examples and their generated plots were used for real-time previewing. Note that we did not support automatic mapping from data to example plots in the current version. Data transformation is beyond the scope of CodeMend but is a very interesting direction for future work.

3.5 Modeling Code and Natural Language

Figure 3.4 illustrates our pipeline for training the CodeMend model. Below, we detail our modeling techniques, including creating simplified representations of code, modeling NL, and jointly modeling code and NL (i.e., bimodal modeling).

3.5.1 Simplified Code Representation

We use a set of *n*-grams $\{x\}$ to represent a piece of code C. Each *n*-gram x is a concatenation of a subset of tokens $\{t\} \in C$, and each token t can be a module, a function, a parameter, or a keyword. The tokens in x must follow a specific order, which obeys the dependencies between the tokens. For example, a module token t_M must be followed by a function token t_F, while t_F may be followed by another function token t'_F if t'_F is a member function of the returned value of t_F, or alternatively t_F may be followed by a parameter token, t_P.

Figure 3.4(c) shows an example set of *n*-grams extracted from a given piece of code. One of the *n*-grams is `plt.title.set_bbox@1`, which consists of four tokens: t_M=`plt`, t_{F_1}=`title`, t_{F_2}=`set_bbox`, and t_P=`@1`. It represents the first positional argument of the `set_bbox()` method of the returned variable of the function `title()`, which belongs to the module `plt` (short for `matplotlib.pyplot`).

Generating these *n*-grams from code and counting them is relatively cheap and convenient, which enables us to leverage the large collection of code examples quickly without

losing much representational power. It also reduces the complexity of the downstream bimodal embedding model. However, the use of *n*-grams as code representation does limit the model's capability of understanding the sequential ordering of code elements or more complex code structure. We describe these limitations further in the Discussion section as well as alternative designs.

To convert code examples to *n*-grams, we parsed found code using Python's built-in *ast* module. We obtained 177,033 unique *n*-grams. Filtering those *n*-grams that did not relate to matplotlib or occurred fewer than 10 times. After filtering, we retained 9,569 unique *n*-grams as the vocabulary of program tokens. In subsequent processing, each code example, as well as the user's code context, were abstracted as a "bag" of these *n*-grams.

3.5.2 Modeling Natural Language

To handle the potential vocabulary mismatch between the user's query and the documentation of the library, we used the *word2vec* package [115] to train a word embedding model (see Figure 3.4(d)) by consuming the previously collected text corpus. We used the continuous bag-of-word (CBOW) model with a vector size of 150, a window of 10 words, and negative sampling of 5 samples per instance. Discarding rare words which occurred less than 20 times in the corpus resulted in a vocabulary of 28,872 words. We ran 10 iterations over the corpus for training.

Since the text corpus we collected was from all Python-related Stack Overflow threads, we were curious whether it had good precision in modeling terms specific to matplotlib. We looked at the terms that are most "similar", as measured by cosine similarity, to the parameter names in matplotlib, and inspected whether these terms are indeed relevant to the concept of the parameters. Though anecdotal, this initial inspection shows that a number of terms relevant to matplotlib are precisely captured in the model. For example, among the most similar terms to "alpha" are "opacity", "lightness", "saturation", and "transparency"; and among the most similar terms to "rotation" are "angle", "orientation", "clockwise", and "counter-clockwise". These term associations are important for handling vocabulary mismatch in the subsequent bimodal modeling process.

3.5.3 Bimodal Modeling

Thus far we have introduced how we model the code context and NL, but these two models are still separate. In this section, we describe the techniques to jointly model code and NL in a single unified model—a neural embedding network. The reason we favor a single unified model over two separate ones is that the predictive outputs of the separate models often

need to be merged based on human heuristics, whereas a unified model can automatically capture the associations between data of two domains with much less human intervention and is thus more robust than models heavily involved with heuristics.

The unified bimodal model we use is illustrated in Figure 3.4(f). The model is inspired by [115] and [2], but is specifically tailored for our code modification task. It takes a code context C and a user's query Q as input, and generates a likelihood prediction $p(X|C, Q)$ over all possible code *n*-grams $X = \{x\}$ as output. In our specific context, it takes a subset of 9,569 code *n*-grams and a subset of 28,872 NL tokens as input, and produces a score for each of the 9,569 code *n*-grams (see Figure 3.4). By training the model, we want the score $p(X|C, Q)$ to align well with the relevance of a code *n*-gram to the user's task expressed by Q in the context of C.

For example, if C contains the `plt.pie` (generate a pie chart) function and Q contains the term "*rotate*", then we want the correct parameter, `startangle` of the function `pie`, to have a very high ranking among all $x \in X$. Since many functions have a parameter related to rotation, the code context, `plt.pie`, serves to disambiguate the user query. Conversely, Q disambiguates all related API functions.

To train the model, we need to supply a series of training instances, (C, Q, x^*), where C is the code context, Q is the query, and x^* is the expected output. These training instances are generated by going through all the code *n*-grams of a code example, selecting one code *n*-gram as x^* at a time, and using the surrounding code elements within a window (e.g., 5 lines of code on each side) as C, and using the docstring of x^* as Q. When we construct C, we randomly drop the *n*-grams within the window with a 50% chance, so as to simulate the situations with incomplete code.

Internally, the model holds three sets of vector representations: (1) V_C, the vectors of the code *n*-grams in the context; (2) V_X, the vectors of code *n*-grams in the output layer; and (3) V_Q, the vectors of NL words. For each training instance (C, Q, x^*), the model fixes the positions of V_Q, which are learned previously using *word2vec*, and adjusts the positions of V_C and V_X in the vector space to optimize the predicted score $p(x^*|C, Q)$.

Once trained, the bimodal model can be applied in a variety of scenarios. Figure 3.5 illustrates four applications that are supported by the model, which we review below respectively.

3.5.3.1 Code to Code

If we only supply the code context C to the model (see Figure 3.5(a)), then the model is computing $p(x|C)$, essentially predicting what the developer would write next, given the code she has already written. Table 3.1 gives an example ranked list of suggested code *n*-

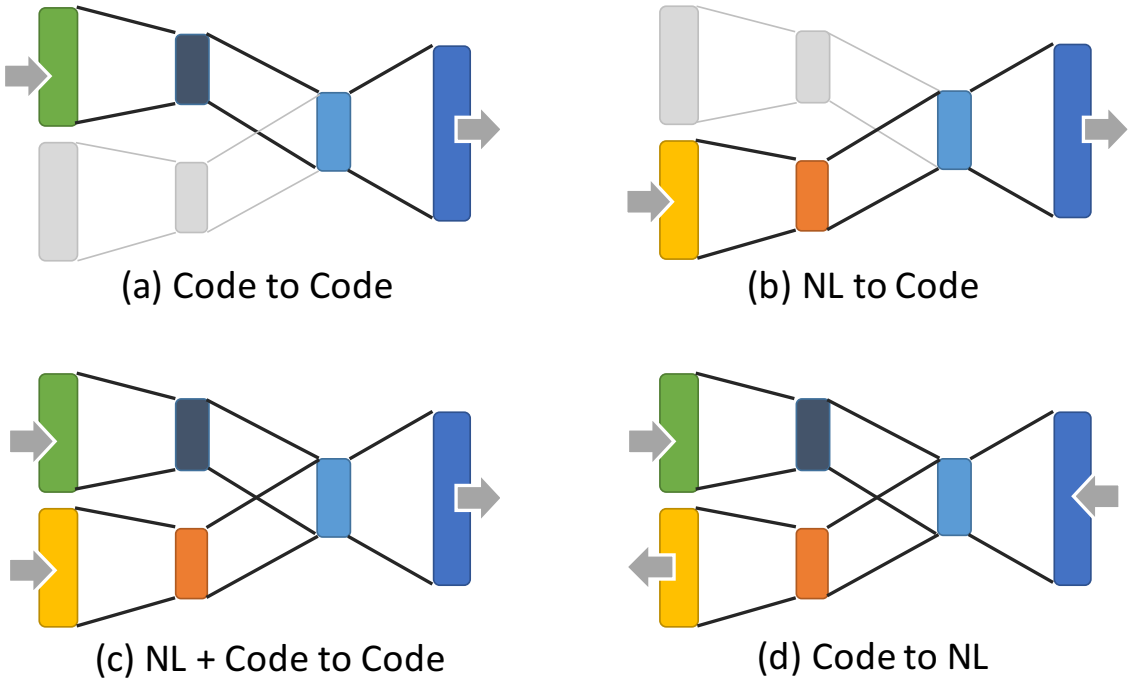


Figure 3.5: Four usage scenarios of the bimodal embedding model

grams. It is interesting to note that the model captures the `plt.bar@label` in the code context, and provides a reasonable recommendation to "create a legend" among the highest ranked result, i.e., `plt.legend`.

Suggested *n*-gram	Score (Unnormalized)
`plt.bar@1`	-3.510
`plt.legend`	-3.715
`plt.bar@0`	-4.478
`plt.bar@hatch`	-4.556
`plt.bar@log`	-5.512

Table 3.1: Top-ranked code *n*-grams based on code context only. The code context contains: `plt.bar`,`plt.title`,`plt.bar@label`.

3.5.3.2 NL to Code

As shown in Figure 3.5(b), if we only give the model NL input Q, then the model is predicting $p(x|Q)$—the equivalent to performing a function or parameter look-up. Table 3.2 shows a ranked list of code *n*-grams based only on an NL query, "*add text label*".

Suggested *n*-gram	Score (Unnormalized)
`plt.plot@label`	6.143
`plt.text`	5.881
`plt.clabel`	5.638
`plt.text@1`	4.832
`plt.clabel@0`	4.215

Table 3.2: Top-ranked code *n*-grams based on NL query only. The query is "*add text label*".

3.5.3.3 NL & Code to Code

Figure 3.5(c) shows the scenario in which the full model is at work. In this scenario, the model is predicting $p(x|Q, C)$—recommending a code *n*-gram based on both code and NL contexts. The NL query is the same as in the last example, but the additional code context promotes the *n*-gram that creates a contour plot (`plt.contourf`) to the top. This example illustrates how code context helps improve the precision of NL-based search.

Suggested *n*-gram	Score (Unnormalized)
`plt.clabel`	5.342
`plt.plot@label`	4.210
`plt.clabel@0`	4.154
`plt.text`	4.023
`plt.xlabel`	3.955

Table 3.3: Top-ranked code *n*-grams based on a combination of NL query and code context. The query is "*add text label*" and the code context is `plt.contourf`, which creates a contour plot.

3.5.3.4 Code to NL

Figure 3.5(d) shows the reverse process, in which the model computes $p(Q|x, C)$—given a code context and one or more code *n*-grams, as well as a collection of short NL utterances, determine which utterance would have best predicted the given code *n*-gram. The last scenario can be used to generate code captions, i.e., short text that summarizes the code. Table 3.4 shows a set of results obtained through this process.

In addition to being able to perform the above prediction tasks, the vectors learned by the neural network also captures the *regularities* of the code elements. One aspect of such regularities is analogous relations between code *n*-grams. Figure 3.6 demonstrates this feature. The vectors V_X's of four "symmetric" pairs of functions are visualized in a 2D plot. Such relations can be potentially used to complete a prediction based on existing

Suggested NL utterance	Score (Unnormalized)
make a log log histogram	54.19
fit to a log scale	38.64
annotate doesn't work on log scale	33.93
create square log-log plots	30.79
use log scale on polar axis	29.55

Table 3.4: Top-ranked NL utterances based on code context. The code context is `plt.hist@log`. The collection of NL utterances are extracted from the titles of the Stack Overflow posts that are tagged *matplotlib*.

context. For example, if the user has called `plt.xlim`, `plt.ylim`, `plt.xlabel`, then the model can complete the analogy and recommend `plt.ylabel`.

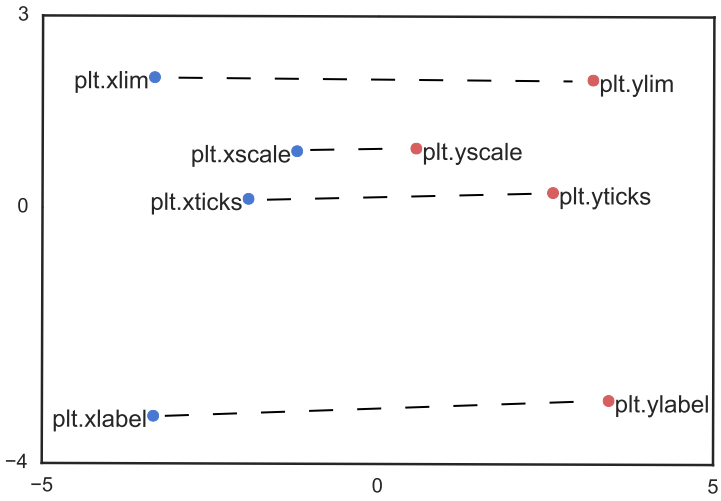


Figure 3.6: Function name analogy in the vector space

3.6 User Interface

We implemented our front-end UI using a client-server model. The interface is loaded in a browser. The code editor is rendered using CodeMirror[3], and the interactive results are rendered using D3.[4] Whenever the user's cursor moves in the code editor or the user types in a query, the code and the query are sent to a back-end Python server to process. The server loads a bimodal neural network model to compute prediction scores. The scores are then sent back to the front-end to render as interactive visualized suggestions.

[3]https://codemirror.net/
[4]https://d3js.org/

Unlike most existing code suggestion approaches that involve complex algorithmic search and inference, our model is simple in that it is stateless and global. At any given point (as illustrated in Figure 3.5), the model takes the combination of code context (can be empty) and NL query (can be empty) as input, and produces an output likelihood distribution among tens of thousands of possible code elements. Since the model essentially operates like a search engine, we could, in theory, directly display the results as a series of search engine results and have the user browse through them. However, showing results this way has two shortcomings: (1) it might be hard for the user to interpret and adapt the result to their code context correctly; (2) it might be frustrating to the user if the model misinterprets the user's intent due to severe vocabulary mismatch or ambiguity that the model is not yet able to handle. So the question becomes: what interface design can not only take full advantage of the neural embedding model's predictive power, but also make its results easily understandable by the user, easily convertible to actual modifications to the code, and will not get in the way of the user's workflow when the model fails?

3.6.1 Nested-layer Spotlight Search

Our solution follows the idea of using multiple cues to guide the user's focus to the correct result. When the user submits a search query, matching lines in the code editor will be highlighted. New functions that can be inserted are suggested in the right-hand panel and are ranked by their likelihood. If the user clicks on a line in the editor, or clicks on a suggested function on the right (e.g., `plt.grid` in Figure 3.7), then the detailed documentation and parameter suggestions will appear. These parameters are, again, ranked by their likelihoods as predicted by the same model. If the user is interested in any of the parameters (e.g., `linestyle` in Figure 3.7), then the possible values mined from the code repositories will appear, which are ranked by their usage frequency. If the user clicks on any of the items, the choice will be backpropagated to the top-layer so that the user can preview the modification made so far. She may later accept or abandon the modifications. In some cases, the value of a parameter can have its own substructure, such as a dictionary, then a deeper layer will be displayed to allow more precise control of the program's behavior.

Using this design of nested-layer spotlight search, we direct the user's attention to the most relevant information on the interface, while providing a mapping between the structure of the search results and the user's mental model of the code structure. Compared to conventional search interfaces, where a fixed number of search results are displayed per query, our interface makes a much larger number of alternative solutions (i.e., functions,

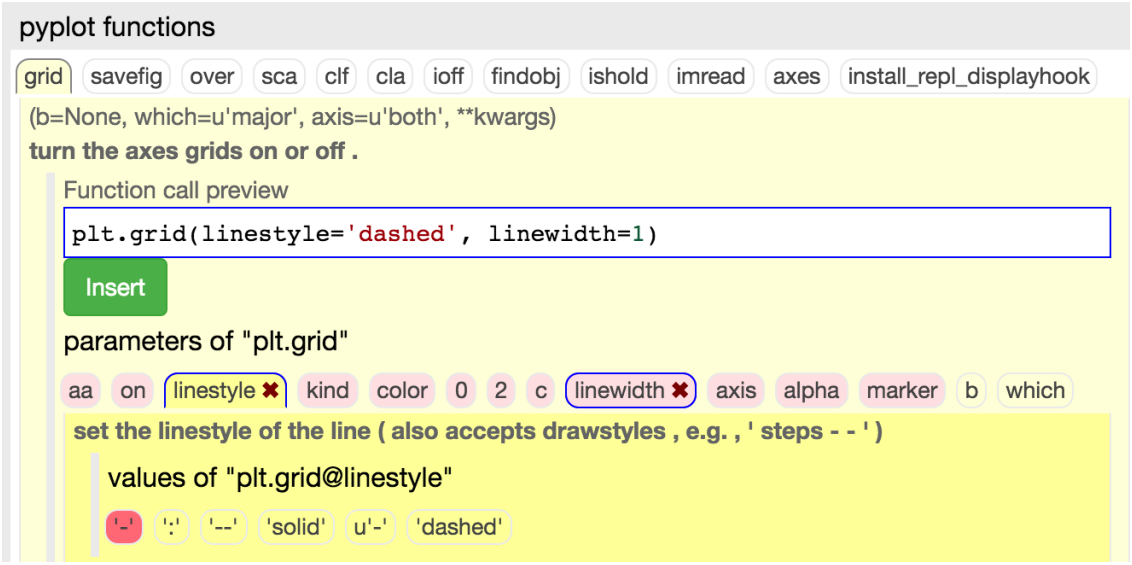


Figure 3.7: Example of nested-layer spotlight search

parameters, values) visible to the user at once. As a result, when the system fails to rank a certain expected function high enough, the user can easily identify and try other options, instead of having to experiment with other queries. This also enables the user to conveniently explore the solution space and even have serendipitous findings.

3.6.2 Automatic Search Scoping

When the user enters a new query, there may still be several layers of suggestions expanded. In such cases, it is not obvious whether the user intends to refine the existing query (i.e., stays in the current nested layer view), or to start a search for a new task (i.e., expands a different set of layers). To resolve this ambiguity, we rely on the output of the model. If the predicted ranking of the items on the current layer is above a cut-off threshold, then we leave the current layer expanded, otherwise we close the current layer, and recursively inspect the upper layers. If none of the expanded layers match the user's intention, then we close all layers and allow the user to pick new lines to focus on.

3.7 Evaluation

We performed two evaluations of CodeMend. The first tested the CodeMend model's performance with respect to the function and parameter search task. This was a more conventional information retrieval (IR)-style analysis which allowed us to quantify how well

CodeMend could retrieve relevant results. The second experiment tested CodeMend's usability through a lab user study.

3.7.1 Search Task Evaluation

One of the main features of CodeMend is that it finds relevant functions and parameters and then highlights lines of code that are targets for modification. We framed the function and parameter search as a standard IR problem and tested where relevant results were ranked.

3.7.1.1 Query Collection

To generate a test set of queries, we leveraged workers on Amazon's Mechanical Turk. We generated five pairs of plots, covering bar charts, pie charts, scatter plots, line plots, and contour plots. Each pair had one "original" plot and one "modified" plot (see Figure 3.8 for an example). Workers saw the pair and were asked to provide NL descriptions of the changes between the two. They were prompted to generate these as if they would issue a Google query to find code to achieve this change.

For the example pair shown in Figure 3.8, workers produced queries like: "change the color of bars", "remove the grid", "move the position of the legend", and "add the shadow into the bars". In the end, 50 workers provided queries. On average, for each pair of images, a worker spent 150.2 seconds, composed 3.74 queries, and was paid $0.13 dollars.

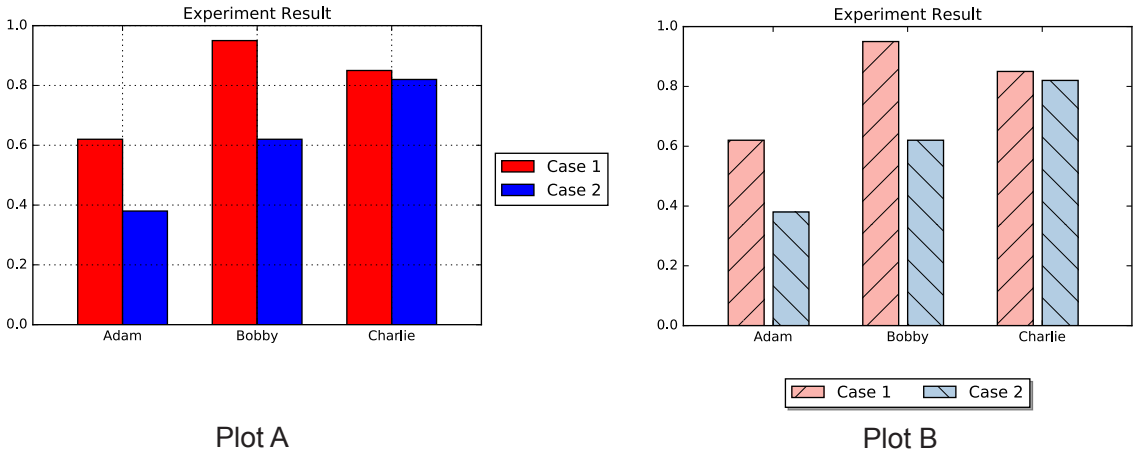


Figure 3.8: Example image pair shown to workers in Amazon Mechanical Turk

The quality of the obtained queries varied greatly. For example, some workers misunderstood the task instructions and, instead of describing how one would specifically change Figure A to obtain Figure B, many workers submitted vague descriptions (e.g., "the figure

styles are different"). Among the 883 queries we initially obtained, we manually selected 361 qualified queries. We filtered out queries if they were (1) duplicate ($\sim$40%); (2) too vague or incorrect ($\sim$30%); or (3) junk (e.g., a single word taken from the figure labels, $\sim$30%).

In the end, we observed that most selected queries were commands to change functions and parameters in the code, such as "change the font size of title." The selected queries cover the different changes and different chart types.

3.7.1.2 Results

We used the queries generated above to test CodeMend in the context of the matplolib API. Because we had both the code that generated the original plot as well as the modified code, we naturally had ground truth on which to test the queries. We used mean reciprocal rank (MRR) as the metric to evaluate the ranking results of CodeMend. We also used $R@1$, $R@5$, and $R@10$, where $R@K$ tests if the correct answer was ranked among the top K results.

We developed three models for the function and parameter search: (1) a *word2vec* model (treated as baseline); (2) a bimodal model that only used code context, as another baseline; and (3) a bimodal model using both code context and NL.

As shown in Table 3.5, the bimodal model using both NL and code context outperformed the baseline *word2vec* model (using NL Only) as well as the baseline that used only code context. Although the bimodal model did not solve parameter and function search tasks perfectly, it demonstrated the ability to return the correct parameters and functions in many scenarios.[5] Note that because CodeMend does not list the search results in linear fashion like Google, this current performance is actually very reasonable, as we can direct the user's attention to the results by using line highlighting and nested-layer suggestion in the interface.

3.7.2 Lab User Study

We recruited 20 subjects in our lab user study to investigate whether CodeMend could help with coding tasks. All but one students were graduate students (the last was an undergraduate). All were in CS/IS or related majors. Based on pre-study reports on skills we had: Python: 6 experts, 9 intermediates, and 5 beginners; Matplotlib: 2 experts, 7 intermediates,

[5]The MRR value of .245 looks much worse than it actually is. As a reminder: the reciprocal rank (RR) of a result is the inverse ($1/K$) of the rank of the correct answer. If the winner is at rank 1, RR is 1/1; at rank 2: 1/2=.5; at rank 3: 1/3=.33; at rank 4: 1/4 = .25. Therefore, a score of .245 means that on average the desired answer is ranked at the 4th place for bimodal but 6th or lower for the other models.

Model	MRR	R@1	R@5	R@10
NL Only	0.153	0.091	0.224	0.249
Code Only	0.090	0.055	0.116	0.141
NL + Code (Bimodal)	**0.245**	**0.163**	**0.335**	**0.429**

Table 3.5: Performance of different models on the search task. NL Only is the *word2vec* baseline; Code Only is the bimodal model with only code as context; NL + Code (Bimodal) is the bimodal model with both NL and code as context.

7 beginners, and 4 never used it before. To ensure a variety of responses, we did not filter participants based on their self-reported experience. This greatly increased the variance of the distribution in both control and treatment groups and likely influenced the significance in the results.

In the study, all subjects undertook a brief training session with CodeMend, and were subsequently given programming tasks, with and without the help of CodeMend. The training was provided through a short demo video and an interactive tutorial. Subjects were asked to complete a set of tasks. In each task, a subject was shown one original plot and images of three modified versions (each building upon the result of the last). Subjects were asked to generate the modified plots, one at a time, starting from the original plot.

For example, one task had the participant change a bar plot. The modified versions included the addition of a grid, rotation of the labels on the x-axis, and addition of shadows to the bars. The pie chart task required changing the size of title box, changing the color of title box, and rotating the pie for 90 degrees. Each participant completed both tasks (bar and pie) using one of two interfaces: a version of CodeMend that replaced the suggestions with a list of Google search results for the query (clicking on these would open the webpage), and a version of CodeMend with all features enabled (and the Google search results listed underneath). We opted to offer Google in both as we did not feel that preventing Web search would be a realistic environment. We logged all interactions with both versions.

Each participant completed one task with the CodeMend+Google version, and one with only Google. Tasks were counterbalanced to account for learning effects.

At the end of the lab user study, the users were asked to fill in another survey to discuss the strengths and weaknesses of CodeMend, and describe whether CodeMend helped them with programming. They were also asked to assign grades to the search results of CodeMend and Google.

3.7.2.1 Results

In the user studies, we found that CodeMend helped the users find parameters and functions to use quickly, and as a result, the users who used CodeMend accomplished more subtasks

than those using just the Google baseline. It is also interesting to note that while the users in the treatment group (with CodeMend) were also given access to Google, they averaged 1.5 Google queries per session, while the number of queries under the Google-only setting averaged at 5.5.

We counted the number of completed subtasks for users, and found that users completed 46 subtasks using CodeMend, whereas they completed 41 using Google search only. Especially, when faced with challenging subtasks, those users with CodeMend were more likely to complete the subtasks compared to users with Google search.

The time spent on completing a subtask with CodeMend was 108.70 seconds on average, whereas the time with Google search only is 134.12 seconds. Furthermore, we divided the users into two groups based on their responses in the pre-study survey: one with high expertise in programming with matplotlib, the other with low expertise. We found the difference of spent time was relatively larger in the group with low expertise than that in the group with high expertise. Thus CodeMend appears to be more helpful for users with low expertise in the programming with matplotlib. However, we note that these were not significantly different statistically. Based on our observations of participants, we believe that the novelty of the CodeMend led subjects to spend more time with the tool than we might expect in ordinary use. In addition, to ensure a variety of responses, we did not filter participants based on their self-reported experience. This also greatly increased the variance of the distribution in both control and treatment groups and likely influenced the significance in the results.

According to the responses of the post-survey, 70% of users agreed or strongly agreed with "CodeMend system efficiently helps me solve my assigned tasks," whereas 55% of users agreed or strong agreed with "Google can efficiently helps me solve my assigned tasks." Users appeared to be more satisfied with the results from CodeMend than Google search. We also found that users uniformly appreciated the function and parameter suggestions provided by CodeMend. Most users chose this as their favorite feature of CodeMend.

3.7.2.2 Limitations

According to the users' responses, the returned results of CodeMend were not always accurate enough. Users sometimes needed to change the query multiple times before eventually finding the correct functions and parameters. We took this concern to heart, and after the conclusion of the study we improved the model with additional training and tuning. Anecdotally, we found that performance improved (i.e., by checking the queries from the lab study, better suggestions were generated).

Subjects also found the number of options in the interface overwhelming at times. In

part, this was due to the unfamiliarity of the tool for the subjects. Additional, long-term use may correct for this concern. However, reducing the amount of information in the UI and making the matches more salient is an area of future work for us.

3.8 Discussion

There are several aspects of the system that can be improved.

Scaling up: The current implementation of LAMVI runs fully in the browser and can only support a small corpus and vocabulary. However, the framework and visual tools are designed to be extensible to support full-sized models with millions of words in the vocabulary using a server-client model. Since most interactive visualizations are focused on a watch-list of just a few words, the overhead of logging additional information per training instance is small. Also note that training efficiency is not usually the primary concern for those who are debugging the model for its quality.

Scaling to other embedding models: The proposed framework can be extended to support a full range of embedding models, including GloVe [135], DeepWalk [136], and LINE [160], because they all share the same underlying neural network architecture. Our framework can also be adapted to embedding models with (slightly) more complex structures, such as Doc2Vec [91] and bimodal embedding models [2]. Adapting to these would require making model-dependent modification to the visualization interface, such as adding a new input channel (e.g. document identity, or input of a different modality). However, the nature of inspecting vector similarity, vector interaction, and tracking the ranks of watched candidate items will remain the same.

Scaling to sequential contexts: It is also possible to extend LAMVI to support neural language models that make predictions using sequential contexts. For example, memory networks can "generate" sentences given a few cue words or a piece of computer source code given a few characters [79]. To debug such models, the user may specify the inputs as a sequence of words or characters, and observe, as the model consumes training data, how different candidate words or characters are reranked among the model's predicted probabilistic distribution. One may also look "further into the future", making the model generate N words or characters in a row, and inspecting how the likelihood of generating a given expected output evolves as the training proceeds. However, it can be challenging to locate specific influential training instances in a meaningful way given the complex nature of sequential contexts.

Explaining model behavior: There are many limitations to our current way of defining most influential training instances or features. An important part of our future work is to

develop meaningful metrics that distinguish which set of training instances or which aspect of the model configurations is most responsible for a given candidate being ranked higher than another.

Supporting exploratory data analysis: In our system, as the model consumes training instances, a wide variety of information is logged. For example: the ranks of watched vectors, their gradients, and learning rates. When using LAMVI to debug a model, the user may have her own information need. Therefore, providing an exploratory data analysis environment provides the end-user with greater flexibility in terms of generating different visualizations and getting insights from the model's training footprint. For example, the user may define customized grouping of the contexts (e.g. by part-of-speech, or rarity of words), and inspect the influences of these training instances category by category.

Linguistic regularity: Our current implementation also supports inspecting the emergence of linguistic regularity captured by the model. The user may enter queries like "king –queen woman" and observe how the desired candidate, "man", evolves. The user may also inspect the activation levels of the hidden units given all three words as context.

Model diff: Our current version does not support direct comparison between two model versions trained with different configurations. Such comparison can be potentially very useful, as the user may directly see the effects of changing one hyperparameter. It would also be interesting to enable the user to adjust the model configurations and see what potential impact that configuration may have on the contributions of specific features on-the-fly, which can, nonetheless, be far more challenging than doing diffs on trained models.

Avoiding overfitting: A potential hazard of the presented debugging pattern is that the user may possibly overfit the specific cases that she selects to focus on, and fail to make the model work well on the overall dataset. Therefore, it is important that the user combine such kind of case-specific debugging routines with benchmark-based testing mechanisms (train/validate/test routines) to avoid overfitting. It would also be interesting to develop a recommended workflow/debugging strategy that combines low-level and high-level debugging routines.

3.9 Summary

In this chapter, we introduced CodeMend, an integrated system that supports natural language queries for code modification suggestions. CodeMend is able to highlight areas of code to change and suggest lines, functions, parameters, or values to use based on the context. We have shown that our model, a bimodal embedding model trained with unlabeled data (text and code), can indeed support programming tasks. We have also proposed a

novel UI to provide a way for developers to interpret suggested results and easily integrate them. Through information retrieval benchmark evaluations as well as in-lab user studies, we demonstrated that the proposed model can indeed accurately suggest the relevant solutions and the proposed interface can help with query disambiguation and support the programmer to efficiently explore multiple different parameters to discover new solutions to their task.

We believe there is significant opportunity in the use of better models of code and natural language. These new techniques also present both opportunity and challenge in developing UIs that can provide effective interfaces between the end-user and underlying models.

CHAPTER 4

Programming by Visual Example

4.1 Overview

Visualizations are an important tool for identifying patterns from data and communicating information. While data tables packed with text and numbers can be impenetrable by human eyes, a well-designed chart or infographic can often serve to present the same data in a more interpretable way. A large number of software applications support the creation of visualizations. In Excel, it often just takes a few clicks to create a compelling data chart out of a table. However, many visualizations are harder to create—the operations can involve multiple steps of data transformation and heavy customization. Some less frequently used statistical charts (e.g., contour plots) may not be supported by GUI-based applications. In such cases, textual programming can be the only or the most efficient way of creating the desired visualization.

When using a programming library to create a visualization, the user may sometimes refer to existing resources for reference or design inspirations. Such resources can be research papers, websites, textbooks, or even whiteboard sketches. Reproducing the charts in these resources and/or adapting them to new data may take considerable amount of effort. This may be due to the complexity of the chart or the user's unfamiliarity with the programming library. In such cases, the user can be greatly benefited by a system that scans the original chart and generates a code snippet that can be used to reproduce the chart. We define this problem as *programming by visual example*, which we study in this chapter.

Driven by recent advancement in deep learning, especially techniques that jointly model images and natural language text [4, 78, 165, 170], we investigate whether computers can help human to write code to generate charts given image examples. Surrounding this new problem are the following new challenges to deep learning models: (1) code has a more rigid grammar and structure than natural language, and it needs to compile in order to work; (2) data visualization charts have unique attributes, such as heavy clutter, deformation, and

heavy presence of markers and text [152], which may create new difficulties for vision—the modeling of natural-scene images are heavily studied and optimized for, but much less effort has been spent for scientific charts, especially the chart types on the long-tail; (3) association between code and image may need stronger connection than normal image captioning because we expect the code to precisely *reproduce* the target image instead of *captioning* it. Overall, solving these problems may provide meaningful advancement to the understanding of neural network technology, and is a meaningful step further towards making AI understand and generate charts automatically.

Given the nature of the problem, one might speculate a retrieval approach might work just fine: if we build a code search engine indexed by images, then each query can be served by finding the most similar images under some measure of image distance. This approach is appealingly simple and may, in fact, works very well for the most common chart types. However, compared to a learning-based generative model, the retrieval approach has a number of drawbacks: (1) chart images may vary by chart type, data, themes, sub-components, and there may not be a good image distance measure that effectively captures all the variations, especially the chart types on the long-tail; (2) the user's query may be a combination of many components found in different images. One may need to break the parts of multiple code examples and recombine them to get the desired output. Deep neural networks can directly learn to do this as long as there are sufficient training data; (3) it has relatively poor potential to generalize to sketch recognition, whereas deep learning approaches have generated promising results in the domain [19, 121]. Therefore, in this chapter, we focus on building a deep learning pipeline, and will include the retrieval approach as a baseline.

To build the deep learning pipeline, we propose to start with an existing architecture, NeuralTalk [78], which connects a convolutional network (ConvNet) and a recurrent network (RNN) for decoding the image and generate source code. We propose to use various new or existing techniques that can be used to optimize the performance, including synthetic training data generation, attention mechanism, and tree-based LSTMs. The study will make the following contributions: (1) the first end-to-end pipeline that performs image-to-code generation; (2) a large set of synthetic dataset for benchmarking; (3) insights into how to make code synthesis work well by conditioning neural language models on external information source.

In the rest of this chapter, we review related work followed by a formal definition of the problem. Then, we discuss our data collection and the proposed methods, and introduce our experiment setup and preliminary results. Finally we have a discussion on the limitations of the proposed approach and list the project timeline.

4.2 Related Work

We review related work on how data visualization charts are automatically analyzed, and existing techniques related to generating sequences in natual language or markup language based on input images.

4.2.1 Reverse Engineering of Charts

Past work has identified chart types and extracted data from charts. For chart type identification, many approaches build specialized algorithms to disintegrate the original charts into smaller components (vectorized elements or graphemes) or bag of *visual words*, and then train supervised models to classify chart types [27, 70, 138, 151]. Other approaches bypass the recognition step (thus not limited by the accuracy of the step) and consider the entire image for classification [152, 177]. Some methods separate text and graphical regions and develop specific feature sets for each region and achieve improved performance [27, 48, 69]. Our work adopts the state-of-the-art deep learning architecture for interpreting chart types and content and does not need the intermediate step to classify images to specific chart types.

For data extraction and chart understanding, the vectorized components of a chart are extracted and regression models are often used to obtain the data represented by the chart elements [71]. Models associating text (especially the axis labels and legend elements) and image components are also essential to chart understanding [27, 69, 152]. For improved association between graphemes and legend marks, assignment algorithms (e.g., Munkres [85]) have been applied [27] and convolutional networks have been used [152]. Some approaches offer an interactive workflow where the user can contribute to the information extraction process under the guidance of the system, thus enabling the extraction of some information that is otherwise difficult to be automatically captured [26, 111]. After the data are inferred from the charts, new charts, often with improved visual representation, can then be generated. For example, ReVision [149] generate new charts representing the same data, by following the guidelines based on perceptual effectiveness rankings [105].

There is also another line of research on recognizing charts hand-drawn by the user [17, 20, 93, 94]. Such charts rarely contain real data and often serve as input to systems that facilitate rapid data exploration and visual story-telling [17]. These systems are often set up in a computational supported whiteboard environment, where the users can draw simple marks and the system provides interactive mechanism to link these marks to data in order to generate complex or compound visualizations. Some example systems are NapkinVis [20], SketchVis [17] and SketchStory [94]. These systems rely on specially designed interactive

patterns for each specific chart type, and are often costly to scale up to a wider range of charts. In comparison, our proposed pipeline can be trained end-to-end that supports a wide variety of chart types.

4.2.2 Image Captioning

Past work has also proposed solutions that automatically generates descriptions for images using retrieval models, combinatorial models, template-based methods, or generative grammars [56, 65, 82, 87, 88, 97, 120, 156, 172]. The state-of-the-art result is achieved by deep learning architectures, in which a deep convolutional network that encodes images is connected with a deep recurrent network to decode the image representation as a sequence of words [78, 107, 165]. Further improvement is shown to have been achieved by using explicit alignment between words and regions of the images or applying attention mechanism to anchor the recurrent generative process with image regions [22, 40, 171]. Despite the impressive performance achieved by the state-of-the-art systems, the generated descriptions are often solely associated with one aspect of the image, and do not serve to *reproduce* the original image content. While the techniques may be adapted for our purpose, a stronger coupling between the source image and output generative process is required for the code generation process to serve our need.

4.2.3 Image-based Code Synthesis

Closest to our work is research done by Deng et al., who propose a deep learning pipeline that can generate markup languages based on input images of equations [37]. Their pipeline achieves near perfect accuracy in reproducing HTML markups given snapshots of webpages with nested layout, and can achieve more than 80% accuracy in generating LaTeX source code given images of equations. Compared to generating captions for natural images, their task of markup language generation needs output to be more strict, so as to reproduce the source code that is not only *grammatical* but also indeed captures *all* the information in the input image. Compared to their task, our task of image-to-code conversion also demands output a language that is stricter than natural image caption (it needs to be able to compile and generate meaningful output), but we have an additional challenge in that the source code that generates a visualization chart often do not have mostly linear local association with the chart, and the source code needs to grasp global information and in a way such that the number of needed edits is minimized.

4.3 Problem Definition

In this section, we formally define the problem of chart-to-code conversion. Given an image I, we want to generate a piece of code P, so that executing P will result in an image I' that satisfies a number of constraints $f(I, I')$. The constraints are defined below.

1. I and I' should belong to the same major chart type (e.g., bar chart, pie chart, line chart, etc.).
2. I and I' should belong to the same sub-chart category (e.g., simple bar chart, stacked bar chart, grouped bar chart).
3. I and I' should have similar graphical elements, including axis label, title, legend, trend line, text annotation, error bar, etc.
4. I and I' should have similar styles, including colors, line types, line widths, markers, and background color, etc.
5. I and I' should have the same hierarchical layout, if I is a figure with multiple charts organized in a hierarchical structure.

Figures 4.2, 4.2, 4.3 describe the image2code architecture.

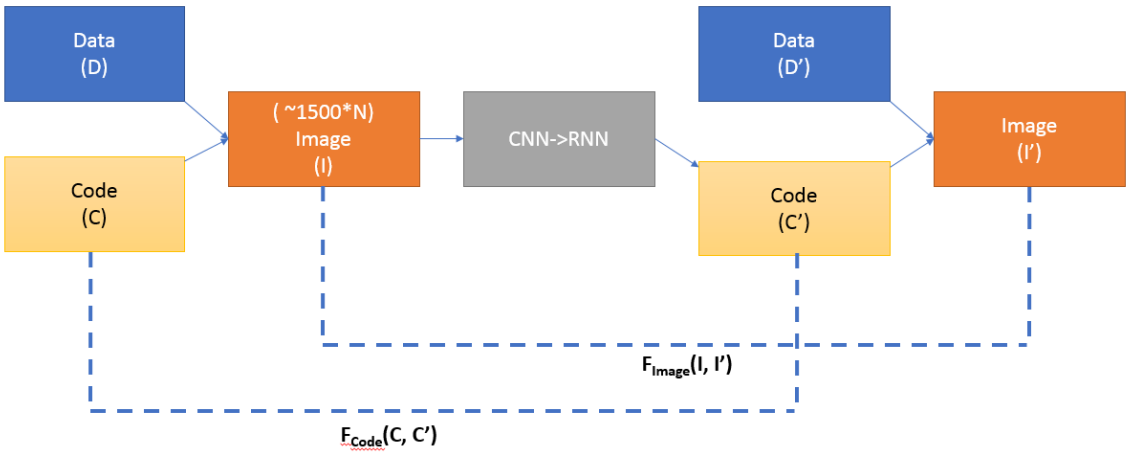


Figure 4.1: image2code encoding flow. N is the factor of data augmentation (ranges from 500 to 1000), F_{code} and F_{image} are evaluation metrics. Test conditions: (i) $D = D'$ (their theme, therir data), experimental scenario (D is given); (ii) $D \neq D'$ (their theme, my data), real-world scenario (D is unknown, and need to generate D' as placeholder)

Having defined the problem, next we introduce our methods.

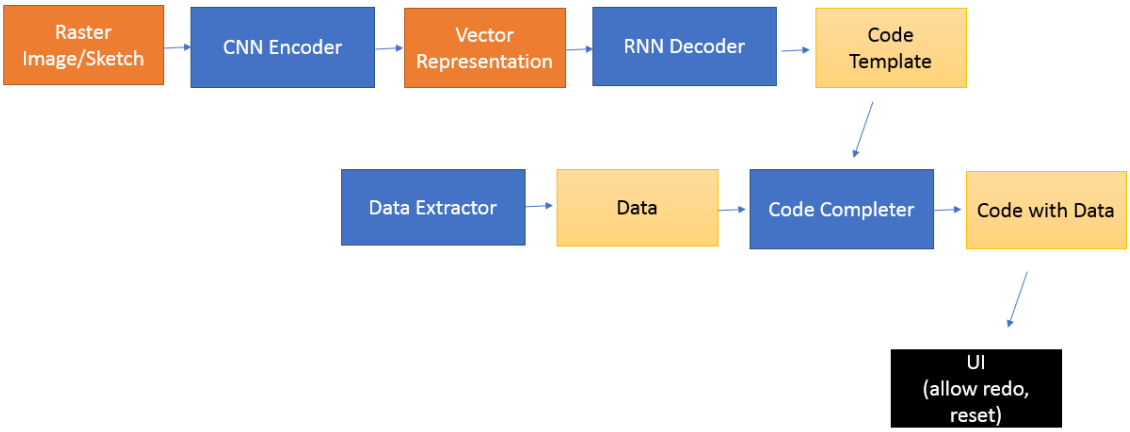


Figure 4.2: image2code system flow

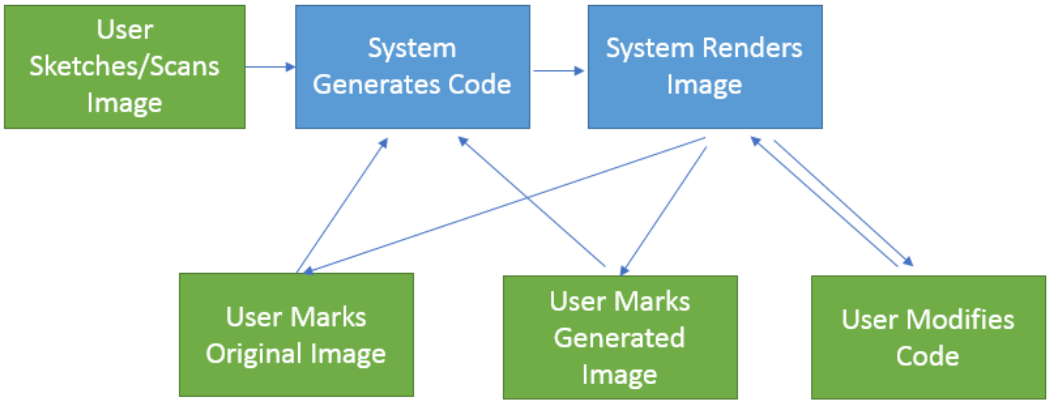


Figure 4.3: image2code System State Machine

4.4 Method

4.4.1 Data Collection

Training our deep learning pipeline requires a large collection of code-image pairs. While there are abundant images on the Internet, few come with source code. In places like online tutorials, social Q&A sites, or GitHub, one can find a number of code examples of graphing library APIs. By filtering those that are self-contained (i.e., those that come with data) and executable, we can accumulate an initial set of training examples.

This initial training set can be expanded by varying the parameters in the source code, or by adding or removing lines. After each modification we attempt to execute the modified code. If it is executable and the generated image is different from the existing images, then we add the new image-code pair to the training set. Within this process, an important step is to identify the part of the code that can be altered. A solution is to first identify the line(s) of code that is actually calling the graphing API, and then execute the original code up until that line. Then we can obtain the attributes (e.g., data dimensions, data types) of the variables inside the function call. These attributes can then be used to guide the altering of code. Some human work is still needed to clean up the code contract of the API library for machine access.

Another cheap way to acquire more training data is to apply global theme changes to the plot, which is supported by libraries like *matplotlib* and *seaborn*). When a theme is changed, the corresponding font, background, and foreground colors are all modified simultaneously, which can result in significant graphical variations.

The goal of expanding the training dataset is not to generate *every possible* chart variation so as to increase the likelihood of hitting the user's input in testing time. The real goal is to reduce data sparsity and prevent overfitting—deep neural networks has great expressive power, and can fit both data and noise in the training set—having a large number of different images that all correspond to similar code serve as regularization of the ConvNet and may help improve the generalizability of the trained model.

Another source of training data we collect is the hand sketching of data charts. We ask crowd workers to draw sketches for selected chart types. Alternatively, the *matplotlib* library can also generate sketch-style charts, although it may be visually very different from what is produced by human.

For evaluation, we leverage the above datasets for which we have the ground truth. To improve our external validity (e.g., to demonstrate that how well our system covers in a real collection of charts), we can also use real-world chart collections, such as those from Wikipedia, arXiv, JSTOR, SSRN, and DBLP, or creating a Web-crawled dataset on our

own (as was done by Chen et al. [23]). Since for these images we do not have their source code as ground truth, we can either hire human annotators or crowd workers to rate image similarity, or apply the aforementioned image distance metrics.

4.4.2 Model

We use the *NeuralTalk* deep learning architecture [78]. A family of similar deep network architectures are the current best performers on standardized image captioning benchmarks [165].

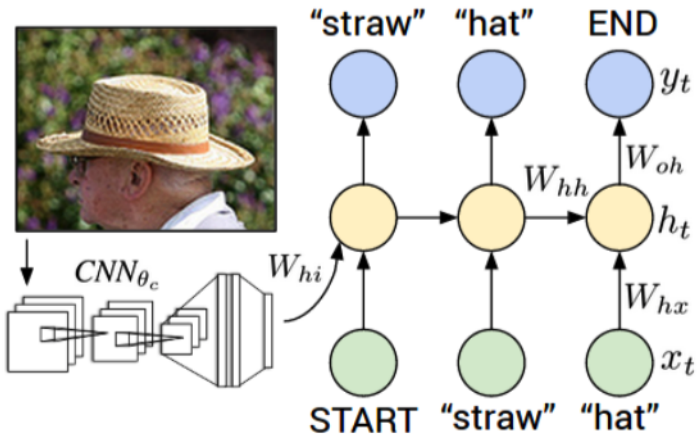


Figure 4.4: *NeuralTalk* architecture: a convolutional neural network (CNN, or, ConvNet) encodes images. The RNN takes a word, the context from all the previous time steps, and calculates a distribution over possible words for the next time step. In the first time step the the context of the RNN is given by the ConvNet's encoded image. (Adapted from [78])

NeuralTalk combines a convolutional neural network (ConvNet) and a recurrent neural network (RNN), which can be instantiated with long short-term memory (LSTM) units. It works by consuming a given image using the CNN, which encodes the image as a low-dimensional vector (i.e., a continuous distributed representation of the image). The vector is then passed as the initial hidden state of the RNN, which takes a beginning-of-sentence token as input, and iteratively spits out the natural language caption, one token at a time. Usually a beam search is executed for improved quality of the generated sentence.

The input image needs to be of a fixed size, and can have multiple color channels (e.g., the standard R, G, B channels). The output is a sequence of tokens drawn from a vocabulary. The entire model is essential replacing the encoder of a standard sequence-to-sequence (machine translation) model with a ConvNet encoder. Training can be done using backpropagation. Normally one starts with a ConvNet pre-trained on a large set of images,

and trains the parametes of the RNN. After the parameters of the RNN stabilize, one can then start fine-tuning the ConvNet by allowing the gradients to flow through RNN to the ConvNet.

The architecture can be improved in several ways, as outlined below:

- **Tree-based LSTMs and ConvNets:** These models have been shown to be able to capture the dependency structure of natural language sequences and improve performance for tasks like semantic relatedness and sentiment analysis [155, 159]. Similar to natural language, programming language also has innate tree structure—every piece of source code can be converted to and from an abstract syntax tree (AST) without losing information (except for formatting). When generating source code, instead of generating a token at a time as a plain sequence, one can iteratively generate tree nodes instead (in either breadth-first or depth-first fashion). By generating the non-terminal nodes explicitly, one can hypothesize that the internal dependencies of different parts of source code can be enhanced. On the other hand, Mou et al. [125] propose convolutional networks defined on AST representations of source code, and show that the network can be used to classify the functionality of a piece of code and detect whether a given code snippet contains bubble sorting. Mou et al.'s handling of the tree structure can inspire new designs of the decoder in the pipeline.

- **Attention:** Attention is a *soft* alignment model, in which the hidden state of the decoder *controls* which part in the input can affect the current time step of decoding. It has been shown to improve the performance of speech recognition [25], machine translation [6, 104], and image captioning [171]. In the context of sequence-to-sequence learning, it works by leveraging the hidden states of the RNN decoder to compute an attention distribution over all the input tokens, so that the conditioning of the decoding process can be anchored by only selected tokens in the input. In the case of image-based decoding, the attention is applied on different spatial regions of the image features as encoded by the ConvNet [171]. To this end, the feed-forward component that was normally at the bottom of the ConvNet pipeline will then be removed to preserve the spatial relations of different regions of the image.

- **Additional feed-forward network for predicting literals:** While the RNN decoder can be trained to generate the overall structures of the source code, including API call sequences and parameters, a separate model can be specially trained to "fill the placeholders", such as string literals or numeric values in the program. In the case of numeric values a regression model may be involved. By separating the task of code

synthesis into a higher-level sub-task and a lower-level sub-task, we may achieve improvement on the overall quality of the generated code.

- **Supporting hierarchical layout:** Many charts are grouped together in a diagram or an infographic with hierarchical layout. Supporting such structures can be useful to the end user. While this is a well-solved problem by using existing methods [37], training the network to learn the functionality end-to-end can reduce the complexity of the overall pipeline, and it also presents new challenges. For example, it is non-trivial to train a network that can simultaneously generate code snippets for single images and composite images.

4.4.3 User Interface

Finally we build a user interface to support the interactive user environment. The interface supports uploading of the user-provided image, and generates source code, optionally with place-holders for data variables or literals for the user to fill in. If no place-holders are generated, then the system should use randomly generated data or other default data. The source code should be easily edited by the user. A potential option is to integrate this with CodeMend (see Chapter 3).

To support real-time interactive operation, the system may take a video stream as input. As the video signal is consumed, a sequence of static images are sampled, and are fed as input to the ConvNet encoder. Additional processing may be required to, say, extract the boundary of graphics from a full sheet of paper.

4.5 Experiment

In our experiments, we want to understand whether the proposed code parameterizer can effectively generate images that are distinct enough to train the model well, and we would also like to explore and compare different model architectures outlined in the previous section, and reach conclusions on the most effective strategy of statistical code generation.

4.5.1 Tasks and Evaluation Metrics

We want to understand the performance of different components separately; then we test how well they work together in an end-to-end pipeline. The tasks and evaluation metrics are outlined below:

Chart type classification—We take the encoded vectors of the input images, and directly feed them into a standard classification pipeline with support vector machines (SVM) [32]. The primary chart types and secondary chart types are used as labels. We report average accuracy, precision, recall, and F1 scores under 10-fold cross validation. We also report the confusion matrix between categories to understand their associations, and later on, inspect whether fine-tuning the ConvNet model could bring performance gains to this task. Note that many existing works have performed chart type categorization with convolutional networks [27, 152]. We include this task not for the purpose of presenting new results, but using it as a sanity check as to whether our model indeed captures the variety of chart types.

Chart element categorization—For this task, we focus on the details of the charting elements. In particular, we focus on the existence, location, and styles of legends, axis labels, titles, trend lines, and text annotations. For each of the elements, we design classification tasks to test whether such variations are captured by the ConvNet. For example, for legends, we build tasks to categorize whether a legend exists, and furthermore, to detect in which region of the image the legend is placed. The set of metrics being reported are the same as the previous task.

Code generation—Given an image, we directly compare the code generated by the end-to-end pipeline. We compare the edit distances between the generated source code and the ground-truth. The fewer edits one has to make to convert one from the other, the better. Note that different components of the code may have very different weights. Therefore, we apply two different metrics. One metric is unweighted edit distance, and the other is weighted edit distance.

Assigning the weights requires measuring which component is critical to the generated image. The intuition behind this is straightforward: mixing up two different shades of greens and two different chart types may both involve one token being mistakenly chosen, yet the penalties should apparently be very different. Following this intuition, a look-up table can be manually created to map *levels* of code tokens to weights. For example, the API function name should be assigned a high weight, less for parameters. One caveat is that different parameters may also need to be weighted differently. For example, for the Python `pandas` library, the `kind` parameter of the `DataFrame.plot` function is often used to describe the chart type, thus this parameter value can have as big an impact to the output image as a top-level API function. To compensate for this issue, we also leverage image edit distances. We directly use the cosine similarity of the ConvNet vectors of the respective image pairs. Alternative options include using SIFT features, as demonstrated in [27].

Also note that BLEU score [133] is commonly used for evaluating performance in machine translation. BLEU measures a form of precision of n-grams of the translation results against a set of human-provided references. A recently introduced metric, CIDER [164], measures consistency between the occurrences of n-grams while considering the saliency and rarity of n-grams. Other widely used alternatives include Meteor [8] and Rogue [98]. We include these metrics in the evaluation as well.

In addition, we test whether the generated code can be compiled, and how human judges perceive the resemblance of the generated images. For this purpose, we recruit workers on Amazon Mechanical Turk, and ask them to rate the similarity of ground-truth images and the generated images.

4.5.2 Baselines

We compare our method with a baseline approach that involves only a simple image index. Following similar approaches in [27], we extract SIFT features of all the images and build a visual word dictionary. Each image is then indexed as a histogram of these visual words (alternatively each image is represented by its ConvNet vector). During testing, for each input image, the image with the most similar histogram representation is fetched, and the source code used for generating that image is returned as output. We hypothesize that this approach is not as performant as the proposed ConvNet+RNN approach.

For testing the performance of the RNN model, we also compare several alternatives, including vanilla RNN model, gated recurrent unit, LSTM, as well as a n-gram model. In addition, the bimodal embedding model proposed by Allamanis et al. [2] can also be used for code generation, and is thus also included.

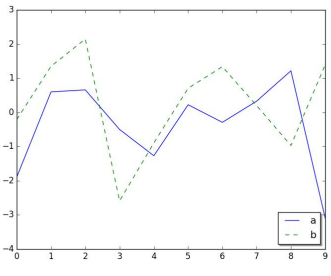
Apart from directly comparing the generated code, we can also compare the intermediate results of code synthesis, such as the API call sequences. If we are able to obtain their data and source code, we will be able to compare against SWIM [139] and DeepAPI [55] as well.

4.5.3 Results

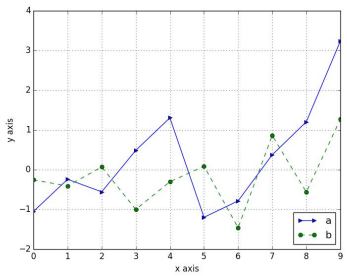
In our preliminary analysis, we used three most common chart types: bar charts, line charts, and pie charts. For each major chart type, we identified 2–3 smaller chart types. For bar charts, the categories include simple bar charts, grouped bar charts, and stacked bar charts; for pie charts, the categories include simple pie charts, and split pie charts; for line charts, we include single-series line charts and multi-series line charts. While they are not

comprehensive, these categories provide a basic proving ground for the ConvNet encoder as well as the RNN decoder.

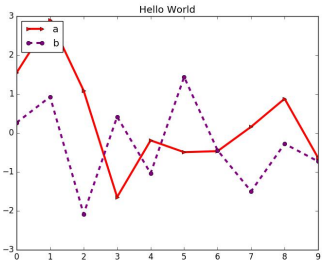
For each sub-category, 200 images were randomly generated. To generate the synthetic data with code and image pairs, we varied the number of data points N ($2 \leq N \leq 14$), the number of categories C ($2 \leq C \leq 4$), only for grouped bar charts, stacked bar charts, multi-series line charts, and pie charts. Both N and C were randomly sampled from discrete uniform distributions specified by their respective ranges. In addition, we also randomly varied the color themes.[1] Since we had seven sub-categories, a total of 1,400 pairs of code snippets and images were generated. Figure 4.5 shows three example pairs of image and




```
plt.plot(a, color="blue", ls="solid")
plt.plot(b, color="green", ls="dashed")
plt.legend(['a', 'b'], loc="lower right",
           shadow=True)
```




```
plt.plot(a, color="blue", marker=">",
         ls="solid")
plt.plot(b, color="green", marker="o",
         ls="dashed")
plt.xlabel("x axis")plt.ylabel("y axis")
plt.legend(['a', 'b'], loc="lower right")
plt.grid()
```




```
plt.plot(a, color="red", marker=">",
         lw=3, ls="solid")
plt.plot(b, color="purple", marker="o",
         lw=3, ls="dashed")
plt.title("Hello World")
plt.legend(['a', 'b'], loc="upper left")
```

Figure 4.5: Seven chart subcategories, example synthetic code snippets, and their generated images. Note that the theme of each image is randomly chosen among 21 built-in options.

For this preliminary experiment, we directly adopted the implementation of Karpathy et al.'s *NeuralTalk* pipeline [78]. We used the recommended publicly available VGG-16 ConvNet model [154][2] pretrained on the MSCOCO dataset [99] as the image encoder, and the default deep LSTM network as the sentence decoder. While the pipeline is built intended for captioning images in the general domain, we want to examine whether such an architecture possesses the potential of being used for image-based code synthesis.

4.5.3.1 Chart Type Classification

We first examined whether these vectors encoded the image features in a way that could be used to distinguish the major chart types. We did a forward-pass on the pre-trained ConvNet for each of the 1,400 images. As a result, we obtained a 768-dim vector representation of

[1]The theme for each image was randomly chosen from 21 built-in themes in `matplotlib.pyplot` module, version 1.5.1.

[2]**`https://github.com/karpathy/neuraltalk2`**.

each image. Using these vectors as features, we trained an SVM classifier to distinguish the major chart types (i.e., one of "bar", "line", and "pie"). Under 10-fold cross validation, the mean accuracy was 99.93% with a standard deviation of 0.14%. These ConvNet features afforded to classify the chart types near perfectly.

Next we examined whether sub-types could also be classified as well. Under 10-fold cross validation, the mean accuracy was 93.71%, with a standard deviation of 1.31%. The drop on performance was expected, as some sub-types had a higher similarity compared to charts of different major types. Table 4.1 shows the normalized confusion matrix of chart sub-type classification. Among different kinds of misclassifications, there is a relatively high chance of misclassifying multi-line charts into single line charts.

Table 4.1: Confusion matrix of chart sub-categories.

Chart Type	simple bar	grouped bar	stacked bar	single line	multi-line	simple pie	split pie
simple bar	**0.905**	0.018	0.075	0	0	0	0
grouped bar	0.043	**0.918**	0.041	0	0	0	0
stacked bar	0.072	0.024	**0.904**	0	0	0	0
single line	0.004	0	0.002	**0.943**	0.051	0	0
multi-line	0	0	0	0.109	**0.890**	0	0
simple pie	0	0	0	0	0	**0.992**	0.008
split pie	0	0	0	0	0	0.028	**0.972**

By placing the images in a 2-D space using t-SNE [163] (Figure 4.6), we further verified that the vectors afforded to cleanly separate the major chart types and most of the sub-categories. The separation of image vectors belonging to the same major chart type (bars and pies) are likely due to the distinction of the randomly chosen themes (there are dark themes and light themes that are visually distinct).

4.5.3.2 Chart Feature Recognition

While chart type and sub-type recognition performances seem reasonably good, we want to explore whether chart features or elements (e.g., markers, titles, legends, etc.) can be captured by the pretrained ConvNet as well. For this analysis we focused on double-series line charts only, and looked at the following 10 features:

- **line1_color_red_or_blue**: the color of the first line, either red or blue;
- **line2_color_purple_or_green**: the color of the second line, either purple or green;
- **linewidth_1_or_3**: if both lines are thin or thick;
- **xlabel_present_or_absent**: whether there is an x-axis label;

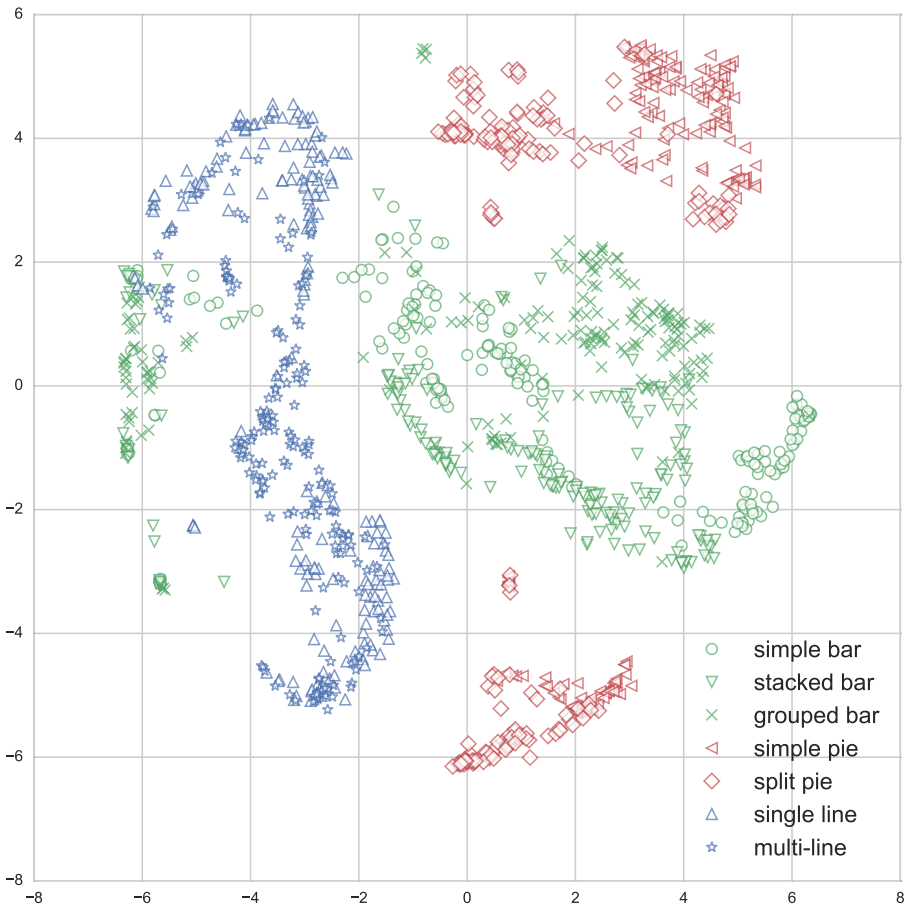
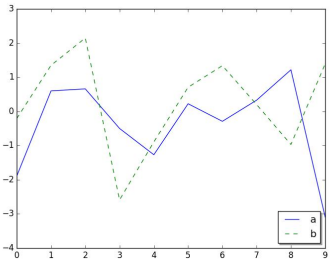


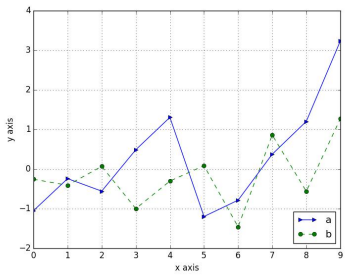
Figure 4.6: Vectors representing charts visualized using t-SNE. A sample of 500 vectors out of 1,400 are shown. Each vector originally has 768 dimensions.

- **ylabel_present_or_absent**: whether there is an y-axis label;
- **title_present_or_absent**: whether there is a title;
- **legend_lowerright_or_topleft**: the position of the legend, either lower-right or top-left;
- **legend_shadow_or_plain**: whether the bounding box of the legend has a shadow;
- **grid_present_or_absent**: whether grid lines are present.

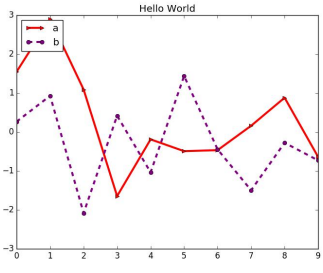
Each of the above features takes a binary value. By enumerating all possible combinations of the 10 features, we generated 1024 code snippets. We then executed the code snippets to obtain their corresponding images. Figure 4.7 shows three example images and




```
plt.plot(a, color="blue", ls="solid")
plt.plot(b, color="green", ls="dashed")
plt.legend(['a', 'b'], loc="lower right",
           shadow=True)
```




```
plt.plot(a, color="blue", marker=">",
         ls="solid")
plt.plot(b, color="green", marker="o",
         ls="dashed")
plt.xlabel("x axis")plt.ylabel("y axis")
plt.legend(['a', 'b'], loc="lower right")
plt.grid()
```




```
plt.plot(a, color="red", marker=">",
         lw=3, ls="solid")
plt.plot(b, color="purple", marker="o",
         lw=3, ls="dashed")
plt.title("Hello World")
plt.legend(['a', 'b'], loc="upper left")
```

Figure 4.7: Three example images for studying chart feature recognition. A total of 1024 such images were created by varying 10 chart features. Data for each image were randomly generated.

For each of the 1,024 images, we did a forward pass on the ConvNet and obtained a 128-dim vector. We then treated each of the 10 chart features[3] as a binary classification task, and did 5-fold cross validation on the entire collection of images.

The classification performance is shown in Table 4.2, ranked by the mean accuracy. As shown in the table, the presence of grids was perfectly classified, and the accuracy for recognizing different line widths was near perfect. Strangely, the accuracies of line1_color and line2_color were very different—this remains an interesting point for further investigation.

The three features listed at the bottom of the Table 4.2 failed to be captured by ConvNet, because the trained classifiers for these features all performed as worse as a 50%-chance

[3]The word "feature" here may be misleading—it is in fact the *label* in the classification problem.

Table 4.2: Chart feature classification accuracy.

Chart Feature	**Accuracy CV=5**	
	mean	**std**
grid_present_or_absent	1.000	0.000
linewidth_1_or_3	0.986	0.006
line1_color_red_or_blue	0.967	0.013
legend_lowerright_or_topleft	0.900	0.023
marker_present_or_absent	0.823	0.016
title_present_or_absent	0.819	0.015
line2_color_purple_or_green	0.759	0.017
ylabel_present_or_absent	0.521	0.025
legend_shadow_or_plain	0.492	0.026
xlabel_present_or_absent	0.470	0.032

random classifier. One explanation is that the image preprocessing we did, which involved rescaling each original image to 256×256-pixel resolution, caused these non-prominent features to become impossible to be captured. Another possible explanation was that the ConvNet—trained for general-domain object detection—was simply not suitable for detecting such features. If this is the case, fine-tuning the ConvNet or re-training one from scratch might alleviate the issue.

4.5.3.3 Code Generation

Finally we examined the performance of the end-to-end code generation pipeline. For this experiment, we used the sub-chart type dataset, with 7 subtypes and a total of 1,400 images. We randomly shuffled the images and used 800 for training, 300 for development, and 300 for testing. For training, we used mostly the default *neuraltalk* hyperparameters. The RNN's hidden state size is 128; the image vector size is also 128; the model was trained for 10,000 iterations. For optimization, we used mini-batch gradient descent with a batch size of 16, and the learning rate was updated using ADAM. For testing, argmax words are sampled with a beam size of 2.

We compare the generated image captions on the test set with ground truth. We report several machine translation metrics (see Figure 4.8).

Seven chart types, all seven types, and a baseline (replacing CNN with randomly sampled vectors) are compared. It can be seen that apart from CIDEr, which has different fluctuation patterns, the rest of the scorers all show that the random baseline performed the worst. Other subtypes generally remain on the same magnitude.

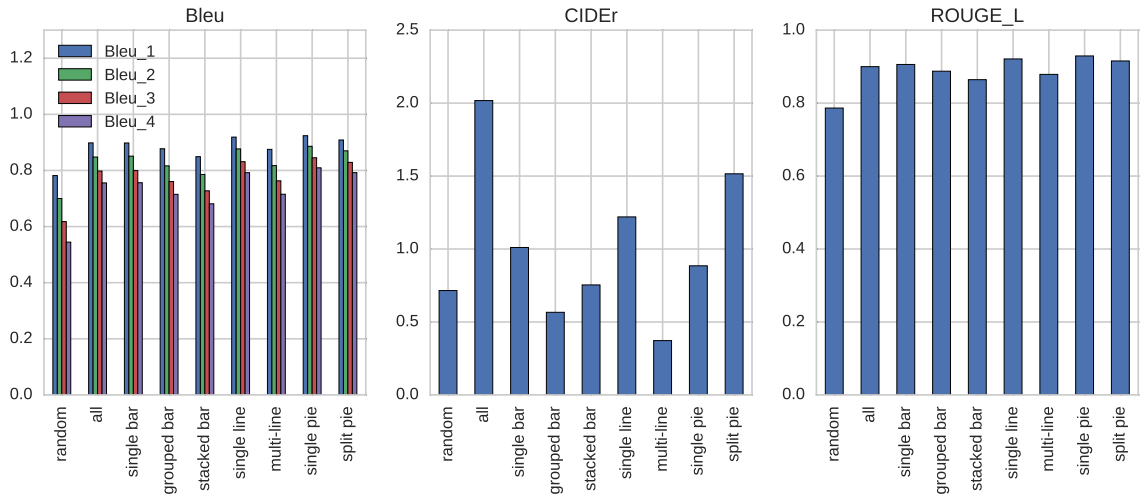


Figure 4.8: Quality of the generated code with machine translation metrics. These are *real* bar charts!

4.6 Discussion

In this section, we discuss the limitations of the proposed approach, including the limitations of statistical language models, as well as our training dataset.

Statistical language models, such as the RNN decoder used in our work, does nothing more than predicting whether a given sentence *looks like* a sentence or phrase that comes from the training corpus [76]. Its capability of generating grammatical English sentences could be misleading in that people may perceive a higher level of intelligence to exist in the trained models. The impressive performance of the recent deep learning models may make people underestimate how much its predictive power is bounded by the domain or coverage of the training corpus.

Out of curiosity, we took a *NeuralTalk* model trained on the general domain MSCOCO image caption dataset and passed in a sample of 50 charts. We inspected the generated captions (see Figure 4.9). Since the dataset is unlikely to contain visualization charts (unverified), it is understandable (and hilarious) to see the generated captions being completely off topic but somehow captures the visual information of the original image.

Note also how some of the generated captions are only *weakly conditioned* on the image. For example, the lower-center image in Figure 4.9 contains three foreground colors and one background color, none of which is white, yet the caption reads "a close up of a red and white sign." It *might* be helpful to specially train a separate model, as discussed before, to be *especially sensitive* to certain signals like colors, text labels, or number of groups/categories.

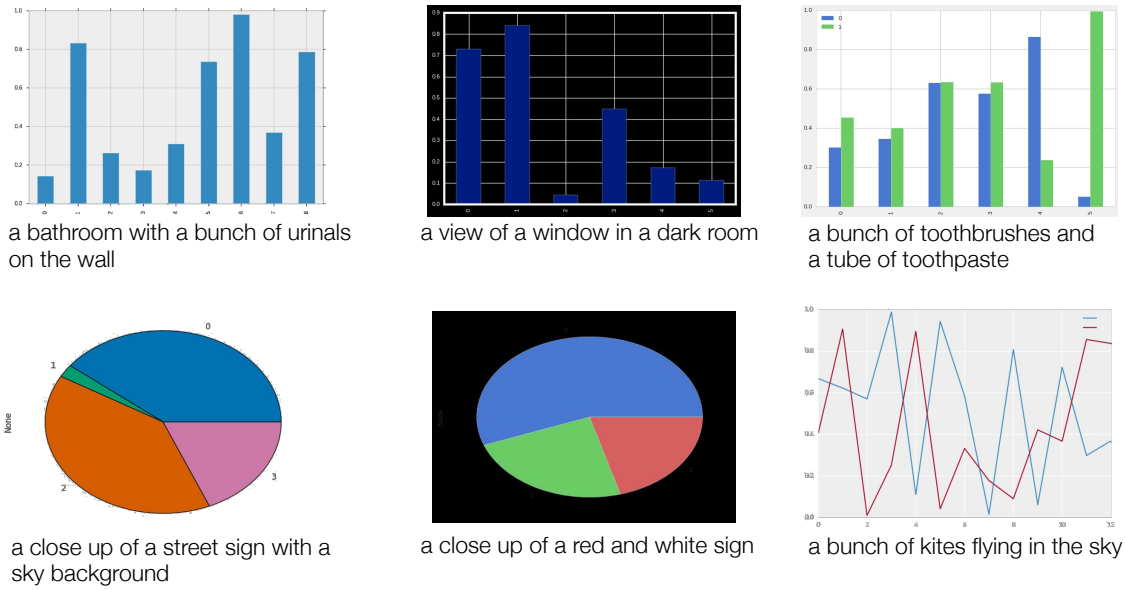


Figure 4.9: Example captions for charts generated by a general-domain model. The model has never been trained on charts, so it can only do its *best* effort to describe what it sees in the image in a funny but revealing way.

If this same model is trained on a collection of charts with source code, we can expect it will generate similar *bizarre* (or totally *expected*?) output when given an image in an unseen domain, or a chart of an unseen type. For example, it is unlikely that the model will generate sensible output given Figure 4.10 if it has never seen contour plots before. What we *can* do is to expand the coverage of the training dataset, and increase the chance of having seen any given chart type. In the future work, we may encode the knowledge of grammar of graphics [169] in the pipeline, so that it truly *understands* the composition of the chart.

4.7 Summary

In summary, we have introduced image2code, a system that can takes the raster image of a data visualization chart as input, and generates source code which can produce the visualization. We adopted state-of-the-art neural network architecture for image captioning as our backend and generated a large synthetic dataset to train the networks. To increase accuracy, we used additional heuristics to extract data from the raw images and used them to modify the code generated by the neural network. To handle cases where the generated source code is inaccurate, we allow the user to scribble on the original/generated image, and we leveraged leveraged soft attention of the neural network and/or beam search of the

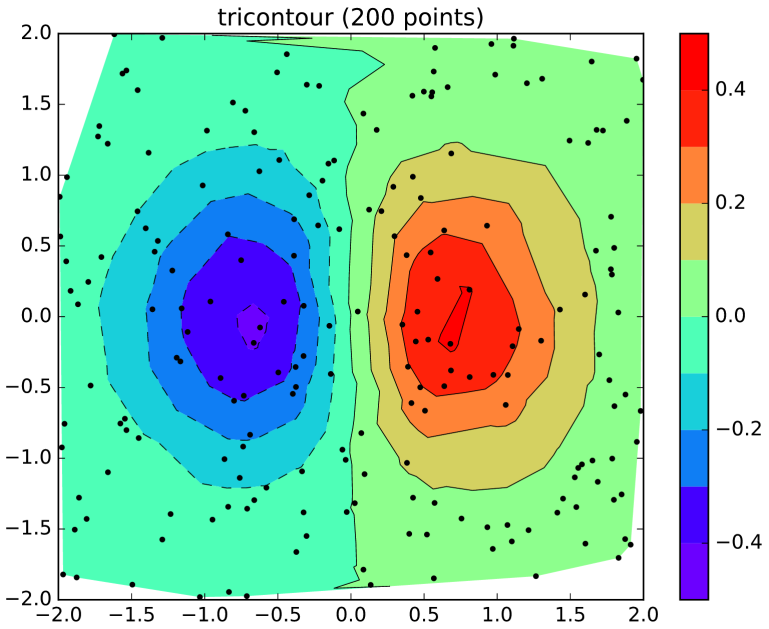


Figure 4.10: Contour plot on an unstructured triangular grid.

generator to make updates to the generated source code conditioned on the changes. We have tested the system for two major chart types, bar charts and pie charts, and the result indicated that our system can handle a large variety of charts and generate source code accurately.

CHAPTER 5

Visual Tools for Debugging Neural Language Models

5.1 Overview

The previous two chapters discussed how neural language models trained on large-scale data can support end-user programming through novel visual interfaces. In this chapter, we discuss how visual interfaces can support the development of neural language models themselves. In a broad sense, developing these machine learning models is also a type of programming, which is being increasingly widely practiced by non-professional programmers. Indeed, the barrier to accessing these advanced models is significantly lowered thanks to the increasing popularity of these models as well as new efforts to publish models (e.g., Gitxiv and many open-source tools). For popular models, such as word2vec, GloVe, or Doc2vec, there are a variety of off-the-shelf implementations and pre-trained models (vectors) to choose from. While in some cases the pre-trained models are sufficient (and highly performant on the provided application/test scenario), developers must often modify the models in some way to work with the specific application, such as adapting a model pre-trained on Wikipedia to analyze the content of a certain sub-reddit, or, embedding users and products in the space for the purpose of building recommender systems. Tuning these models may require the researcher/developer to provide new data, change model parameters, or more significantly change the architecture.

It is often this exercise of modifying existing solutions that creates frustrating challenges to developers. Models can be highly sensitive to numerous factors ranging from pre-processing, to training, to the model configuration itself. These are often non-intuitive to the developer and results in an unguided experimentation with what look like "black-box" models. Often, it is not clear if the progress made in tuning the models is leading to the desired, let alone optimal, outcome. While one can perform grid search or use automatic parameter tuning techniques such as reinforcement learning [11, 54], a reliable

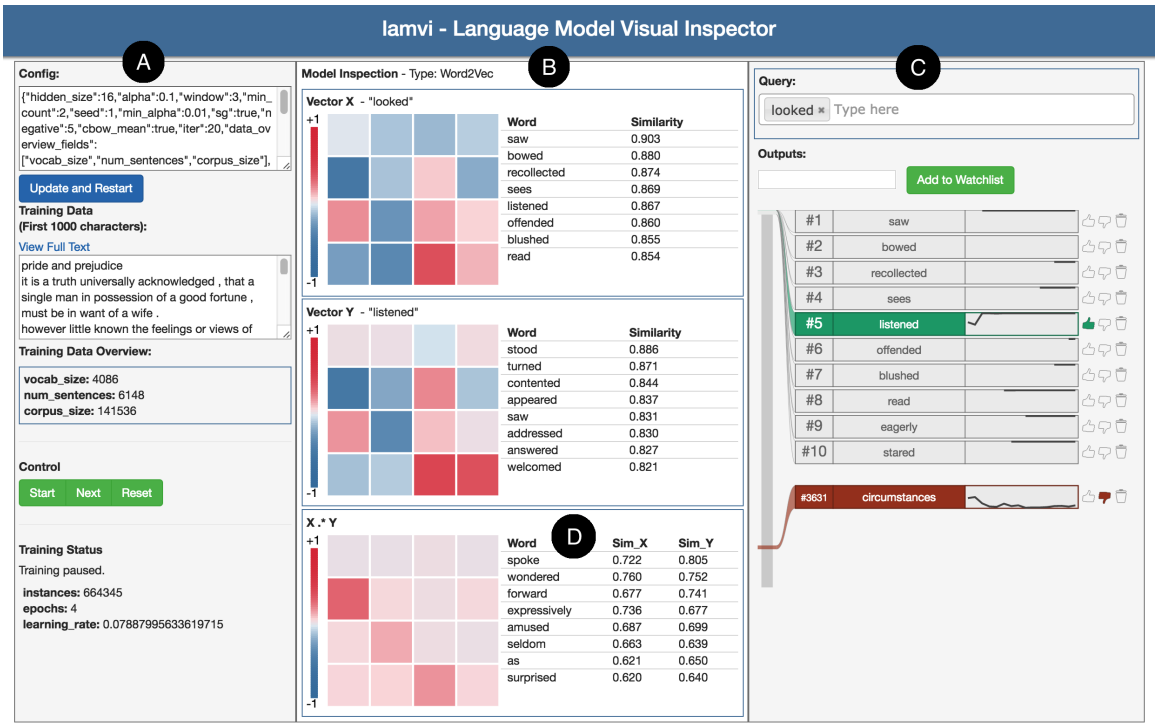


Figure 5.1: Screenshot of LAMVI-1. Panel A shows model and corpus overview; Panel B supports the inspection of activation levels of hidden-layer units, vector interactions, and influential training instances (not shown here; see Figure 5.5); Panel C allows the user to enter queries and select words to *watch*; Panel C also visualizes the change of ranks of the watched words over iterations; Panel D shows the interactions between two vectors, as well as the most influential words contributing to their association.

ground-truth dataset is necessary. However, when the models are applied to a novel domain where little ground-truth exists, the developer needs to rely on her own intuition to judge the quality of a model instantiation. This is often done by inspecting the nearest neighbors of words or performing clustering and checking whether the clusters bear coherent semantics. This process, although intuitive, involves repetitive human effort and can be highly unreliable [44].

Our belief is that model tuning and debugging can better be addressed through novel tools. To that end, we have developed LAMVI (LAnguage Model Visual Inspector), a visualization-driven tool for debugging neural embedding models. We have specifically opted for a visual interface because of the complexity of the task involved. The user needs to keep track of the input parameters, the intermediate states of the models, output performance metrics, and the performance on individual test cases. Debugging such models without "summaries" of the data can be extremely difficult. Visualizations, as cognitive

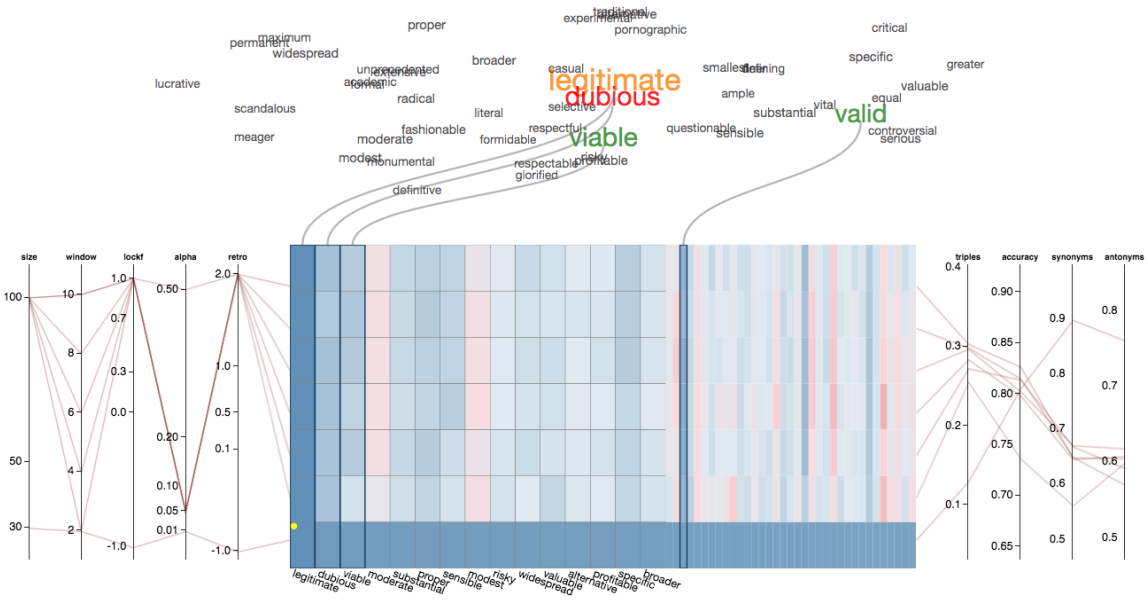


Figure 5.2: Screenshot of LAMVI-2. The heatmap compares the similarity scores between the query word ("legitimate") and a set of words similar to the query assigned by different models. The ranking is based on an underperforming model instantiation (the bottom row). As seen in the heatmap, the rest of the models have large disagreements with the model in focus. Each line in the parallel coordinates visualization represents a model instantiation. The left-hand side shows the model parameters, and the right-hand side shows the model performance metrics. The top panel is a t-SNE visualization of the word vectors for the assistance of navigating through the embedding space.

boosts, provide access to the "black-box" without requiring complete transparency. Rather, those aspects of the data/model that are useful in making decisions are provided through visual channels, leveraging preattentive perception [61] to facilitate the detection of patterns and outliers.

In this chapter, we describe two instantiations of the LAMVI system, namely LAMVI-1 and LAMVI-2. Both instantiations share the same purpose of assisting model developers with debugging. LAMVI-1 is an early research prototype that is focused on one individual model at a time, and LAMVI-2 is a more mature platform that is capable of comparing the performance across multiple different model instantiations.

In LAMVI-1, in addition to focusing on single model instantiations, we also have an assumption that the expected output is largely clear to the developer. For example, given a particular query (e.g., input term), the developer would be able to "grade" the output by assessing the degree to which it matches human expectation. Visually providing access to this output, in a way that supports this grading, is a key design requirement we are target-

ing. Additionally, we would like for the end-user to be able to trace failures of the model in a way that supports tuning decisions. Ideally, the end-user should be able to generate reasonable hypotheses that more training data, different pre-processing, or a different tuning parameter, or even a different architecture, might help. They should then be able to easily test this modification. We have particularly focused on the identification of two types of model failures in our initial implementation: (1) insufficient signal in the input corpus, and (2) the training process was not properly configured. LAMVI-1 allows the developer to see the input corpus, configure the model, and iteratively execute the training. During this training, the developer can inspect the hidden layers of the model, observe how they change, and "probe" them for specific query terms. More critically, the system allows for inspection of *pairs* of words (e.g., query and result), their joint hidden layers, features, and actual training instances (examples pulled from the input corpus). Various visualizations (e.g., heatmaps, and 2D projections) allow the developer to better "understand" the underlying model.

In LAMVI-2, we base our system design on the assumption that the developer has already trained several different instantiations of the model, and wants to gain insights into how the model parameters influence the metrics of the model quality. Using a combination of inter-connected visual tools, including parallel coordinates, a heatmap, and a scatterplot, the developer is able to use a combined interface to test multiple hypotheses about the model parameters. In particular, the parallel coordinates can support the finding of potential causal relations between input parameters and the output metrics; the heatmap can support the comparison of nearest neighborhoods of query words across different models, spotting outliers, and identifying clusters; and the scatterplot serves as a spatial representation of the nearest neighbor words of the query of interest.

Since the target use cases for the LAMVI-2 system is to support the debugging scenarios where the domain is new or the labeled data are scarce (e.g., a few word pairs), it is necessary to introduce measures to prevent the user from tuning the model in a way that overfits the few labeled samples. To this end, LAMVI-2 supports the user to add easily customized ground truth by flagging word pairs as positive or negative, and tracks the average similarity assigned to the positive set and the negative set by each model instantiation respectively. Ideally a "good" model assigns high similarity scores to the positive set and low scores to the negative set. This function provides extra dimensions for the user to assess the model performance, and may reduce the risk for the user to pick a less optimal model instantiation.

To evaluate LAMVI-2, we designed a task of tuning embedding models for capturing the semantics of highly-polarizing adjectives. The task requires the user to tune a skip-

gram embedding model on an IMDB movie review dataset. We recruited 14 test subjects and partitioned them into two groups. One group used the LAMVI-2 system and the other group used a baseline system with only basic support for checking nearest neighbors and evaluating word similarities. We asked the users to use their intuition to find the best-performing model, and used a held-out dataset to examine the actual performance of their final chosen model. The experiment result showed that the group of users using LAMVI-2 consistently outperformed the group of users with the baseline system. This proved that LAMVI-2 can indeed effectively facilitate the user in the debugging of the neural language model.

In the rest of this chapter, we first review related work, then briefly introduce LAMVI-1, followed by a detailed description of the design and the usage patterns of LAMVI-2. Next we describe the user study we conducted for LAMVI-2 to test its usability, and we conclude with discussions on the extension of LAMVI-2 to solving additional real-world debugging problems.

5.2 Related Work

5.2.1 Visual Inspection of Text

There is a large amount of work on use of visualization to inspect text, some with a machine learning focus. For example, AntConc provides concordances and other visual toolkits to support corpus linguistic analysis, such as word frequencies and collocation inspection [3]. Chuang et al. use association matrices and alignment charts to investigate latent topic coverage of a large collection of model variants [28, 29]. LDAvis employs interactive visualizations to facilitate the user to interpret the content and inter-relationships of different latent topics learned by a topic model [153]. Other examples related to topic visualization include [33, 103]. Kulesza et al. use simple bar graphs of feature importance to provide answers to the user's *why*-questions regarding text message classification in an email client application [86]. The visual tools presented in these studies greatly improve the interpretability of their corresponding language models, and many of the techniques, such as concordances, are borrowed in our LAMVI implementation. LAMVI, however, is particularly focused on the training process of neural network models.

5.2.2 Visual Inspection of Neural Networks

Using visualization techniques to improve understandability of neural network (NN) models has also drawn a great deal of attention. Many NN visualization projects are focused on computer vision [175, 148]. There are also several interactive visualization projects developed for educational purposes, such as the recent Tensorflow Playground.[1]

Compared to images, videos, or quantitative multidimensional datasets, words and sentences lack natural visual representation and can be harder to interpret. The required transformation of text to data that can be visually encoded presents unique challenges. To improve understandability of natural language representations learned by neural network models, existing techniques for visualizing high-dimensional data are often borrowed, such as principal component analysis (PCA) [74], multi-dimensional scaling [84], or t-SNE [163]. However, these methods are not neural-network-specific, and do not specifically improve the understandability of the learning process of neural networks. In comparison, a recent work by Karpathy et al. employ multiple views of activation levels to illustrate how recurrent neural networks capture patterns in text sequences, such as long-range dependencies [79]. We have previously released Word Embedding Visual Inspector (WEVI) [145],[2] an interactive educational tool that lets the user play with a toy word2vec model in the browser. In comparison, the presented LAMVI system is not just a tool that supports understanding the underlying neural network model, but provides an interactive debugging environment for model development and deployment as well.

5.2.3 Visual Inspection and Manipulation of Multi-dimensional Data

Various techniques have been proposed for inspecting and manipulating multi-dimensional data. These techniques can be categorized as spreadsheets, point-based, region-based, and line-based [52].

For the purpose of understanding the intrinsic structure in large-scale and high-dimensional data, machine learning community has proposed various dimensionality reduction techniques that can project high-dimensional data down to 2-D or 3-D spaces, so that visual inspection is possible.

[1] `http://playground.tensorflow.org/`
[2] `http://bit.ly/wevi-online`

5.3 LAMVI-1: An Early Prototype

We briefly describe an example interaction to demonstrate LAMVI's expected use. Suppose Alice has trained a word2vec model on Jane Austen's *Pride and Prejudice* using the default model parameters. She inspects the nearest neighbors of "*wife*", expecting "*husband*" to be ranked the highest. Instead, "*engagement*" and "*marriage*" are ranked higher than "*husband*,". She wants to find the reason for this behavior and to fix it. Using LAMVI, she visually inspects the vectors of the words in question, and identifies the most influential contexts that contribute to the false positive outputs. She then uses LAMVI's concordance view to examine the relative positions of these context words to the query word ("*wife*"), and finds that many contexts that contribute to the false positives are farther away from "*wife*" than those contributing to "*husband*". Knowing this, Alice reduces the context window size, and retrains the model. LAMVI automatically tracks the ranks of the words of interest across training iterations. By inspecting the ranking records, Alice confirms that "*husband*" stabilizes at the highest rank after a few iterations. This example illustrates one of the many debugging scenarios enabled by LAMVI.

Figure 5.1 shows a screenshot of LAMVI. The integrated interface supports many common debugging activities, including: configuring model parameters, overview of the training data, pausing training, and stepping in a training instance (Figure 5.1-A); specifying input queries and tracking the ranks of expected candidate outputs (Figure 5.1-C); viewing activation levels of hidden units (Figure 5.1-B) and vector interactions (Figure 5.1-D), as well as inspecting influential training instances (Figure 5.5) and checking 2D projections of vectors of interest (Figure 5.3).

5.3.1 Tracking Ranking of Specific Candidates

As shown in Panel C of Figure 5.1, the user can monitor the change of ranks of specific candidate words given an input query she selects. We find that monitoring the rank trend over iterations can be informative in a number of scenarios. The *ideal* situation is that an expected "good" candidate starts off at a random position and, as the training proceeds, gradually moves to the top among all candidates where it stabilizes (e.g., "listened" in Figure 5.1, Panel C, given the input term "looked"). If the rank of an expected output stabilizes at a low rank position, it can be because of lack of relevant training instances; whereas the rank of an unexpected output stabilizing at a high rank position may indicate too much noise in the training data. If the ranks of multiple watched candidates remain unstable and oscillate, it can be because the learning rate or the learning rate decay strategy may be set inappropriately, or the training data contains too much noise.

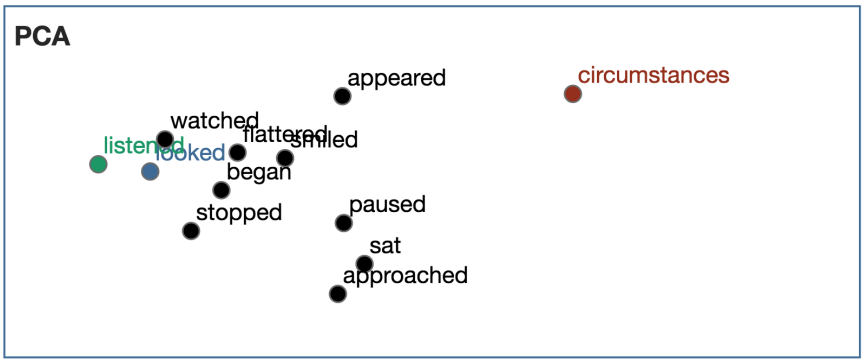


Figure 5.3: Multifunctional center panel of LAMVI: 2D visualization of word embedding vectors of user-specified input query, watched candidate outputs, as well as nearest neighbors of the input query. PCA is used for dimensionality reduction. The user may label certain candidate outputs as *good* or *bad*, and such candidates will be colored differently.

While the visualization may not identify a single cause, it narrows the possibilities down and provides avenues for additional exploration. Rank monitoring provides a high-level sense of the training process. However, one can dig deeper into the low-level details of word representations learned by the model in order to identify the true causes of certain model behavior.

5.3.2 Inspecting Vector Representations

We provide three different ways to let the user explore the vector representations learned by the model.

First, a heat map (Figure 5.1, Panel B) can directly illustrate the values of different components of a word vector. Given a word w, suppose the hidden layer size is K, then its vector can be denoted as $\mathbf{v}_w = \{v_{w,1}, \cdots, v_{w,K}\}$. Each cell is a color-encoded representation of a vector component, $v_{w,j}$, $j \in \{1, \cdots, K\}$. While the positions of the cells do not have actual meanings, rendering them as a matrix instead of an array not only makes the layout "tighter," but also makes it easier for human eyes to spot patterns. However, we do recognize the inherent risk that gestalt heuristics will lead the end-user to spot a pattern that isn't there (see Chapter 6 of [167]). If the training configuration is set properly, the user can typically observe that the cells start off with random colors, and, as the training proceeds, a few cells turn darker colors (meaning $v_{w,j}$ is getting close to either -1 or 1) and stabilize, while a majority of cells stabilize at lighter colors ($v_{w,j}$ close to 0). A deviation from this pattern may indicate improper learning rate selection (e.g., causing vector

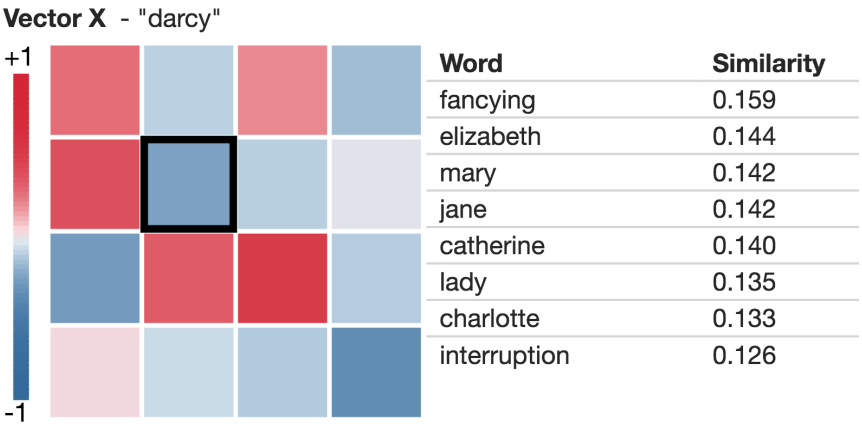


Figure 5.4: Inspection of the topic(s) associated with a single vector component, highlighted by a black rectangle. The query word is "*darcy*".

components explode to infinity), error in model implementation, and improper initial value selection.

Second, we provide a list of nearest neighbors based on a specific dimension j^*. When the user selects a specific cell, e.g. j, the corresponding dimension is then used to find a ranked list of words $\{w'\}$ that are both (1) similar to the current query word w, where similarity is measured by cosine similarity,

$$\cos(\mathbf{v}_w, \mathbf{v}_{w'}) = \frac{\mathbf{v}_w \cdot \mathbf{v}_{w'}}{\|\mathbf{v}_w\| \|\mathbf{v}_{w'}\|}; \tag{5.1}$$

and (2) share similar activity as w on dimension j^*, i.e., $v_{w,j^*} v_{w',j^*}$ is high. Using this view (see Figure 5.4), the end-user can gain an understanding of the "meaning" of a dimension by observing which words are activated.

Third, LAMVI offers a 2D plot showing the nearest neighbors of the query as well as all the watched candidates (see Figure 5.3). We use principal component analysis (PCA) for dimensionality reduction.[3] As the training proceeds, the user can monitor the change of the positions of vectors, and even potentially spot interesting clusters.

While these visual tools allow the user to quickly gain insights into what is learned by the model, they do not directly answer *why* certain candidates are ranked higher than others.

[3]One can also use many other techniques, such as *t*-SNE [163].

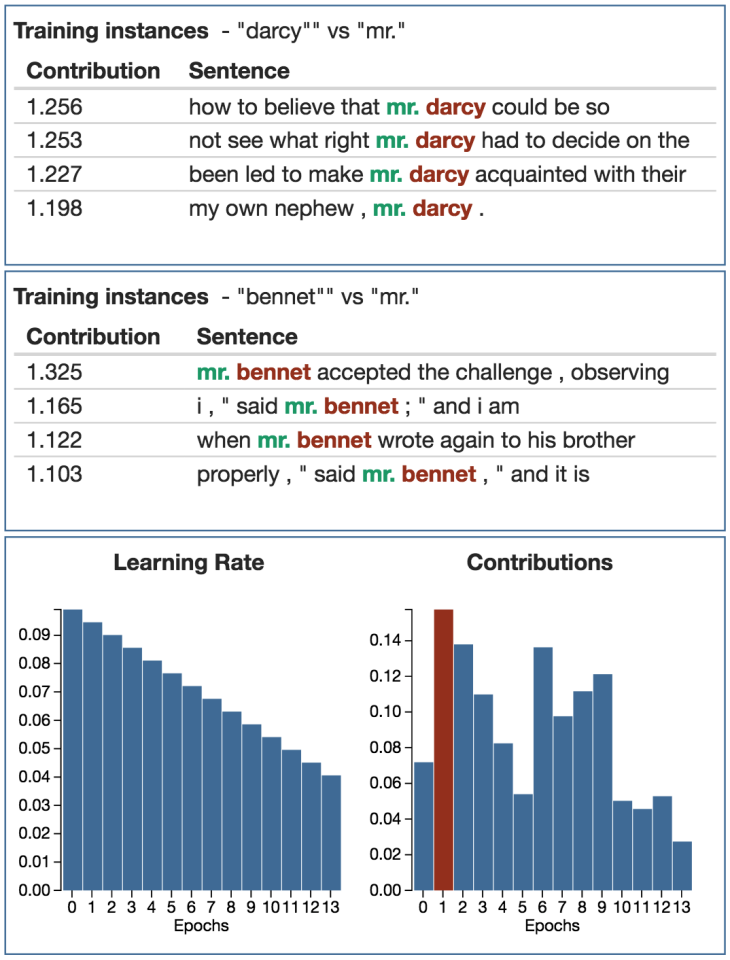


Figure 5.5: Inspecting training instances. The top panels show context words in sentences. The bottom panel compares the learning rates and contributions of a single sentence by training epoch.

5.3.3 Inspecting Interactions of Vectors

To understand how a pair of vectors $(\mathbf{v}_w, \mathbf{v}_{w'})$ become "close neighbors" we would like to inspect the training instances we have encountered. We can calculate, among all the training instances we have encountered while learning ($\mathbf{v}_w$ and $\mathbf{v}_{w'}$), which ones are the most influential. Figure 5.1, Panel D) illustrates two ways of inspecting such results.

First, we can show the element-wise product of the two vectors, i.e., $\{v_{w,1}v_{w',1}, \cdots, v_{w,K}v_{w',K}\}$ as a heatmap. As the user focuses on individual cells, we show which words are most activated by that corresponding vector component. Through this view the user can get an idea of what topics are *shared* by the two words.

Second, we show a list of training instances that have made the most contributions. Each instance is a context word w_c encountered for the word of interest w during training, and the contribution of the instance is the L2 norm of the gradient on $\mathbf{v}_w$ when that instance is encountered, i.e., $\|\frac{\partial E}{\partial \mathbf{v}_w}\|$, where E is the learning objective.

For example, if the vectors of two characters' names are close (e.g., "darcy" and "bennet"), we may observe that they share most influential features related to human beings, such as "mr." and "mrs." When some expected features do not show up, the user may consider actions such as improving the corpus preprocessing routine or adjusting the window size.

5.3.4 Inspecting Training Instances

Since an influential feature may occur in multiple sentences throughout the corpus, we may want to go further and inspect which specific training instances (i.e., sentences) contribute the most to certain associations learned by the model. Figure 5.5 illustrates two of our solutions.

First, we borrow the idea of concordances and show a ranked list of sentence snippets from which the influential features are extracted. These snippets are ranked, again, by their contributions to the position of $\mathbf{v}_w$. By inspecting the actual sentences one may spot errors or potential improvements to be made in the corpus preprocessing routines (e.g., "mr. darcy" may be concatenated as a phrase to distinguish the term from "mrs. darcy"), or make better judgment about the size of the context window.

Second, since a single sentence may be encountered multiple times in different training epochs we need access to the contribution each context has made. In the interface we show what learning rate is applied to the contexts in the sentence (in each epoch), and what contributions (again, in the epoch) that the context has made to $\mathbf{v}_w$. By inspecting such information, the user may compare the effects of different relevant hyperparameters, in-

cluding learning rate, speed of learning rate decay, downsampling, and negative sampling. For example, if the user spots that some "good" features are downsampled too heavily, she may consider adjusting the "sampling" hyperparameter of the model to avoid losing important signals in the corpus.

Note that there are hazards of overfitting the specific instances inspected here. The user should always check multiple instances and have a benchmark to keep track of the model's overall performance. Ideally, future instances of the tool can help guard against this by better supporting this kind of tracking.

5.4 LAMVI-2: Parallel Coordinates with Nearest Neighbor Inspection

In this section, we first analyze the requirements of the user's need for visually debugging neural embedding models. Then we introduce the design of LAMVI-2 by individual components. Lastly we discuss several typical workflows following which LAMVI-2 is intended to be used.

5.4.1 Requirement Analysis

Through interviews and feedbacks from domain experts, we identified the following requirements for the fundamental functionality of the user interface.

First, the interface must be able to provide the user with a mechanism to intuitively and efficiently identify evaluation benchmarks. Since we target scenarios for debugging neural language models that are applied on domains where little ground-truth data is available, the user needs to be able to establish customized ground-truth based on data exploration. The forms of ground-truth can vary from case to case. For example, the user might want a pair of words to be similar, or, want a certain group of words to form a coherent cluster. Alternatively, the user might want to exclude certain words from a cluster. The visualization interface must make it easy for the user to explore the possible candidates for establishing such ground-truth items, and to manage them by creating, updating, and deleting individual items.

Second, the interface must support the user to quickly find associations between control input (e.g., a model's hyperparameters) and the model performance. The performance of the model can be evaluated against pre-established benchmarks or be dynamically udpated as the user creates and maintains customized ground-truth. The types of associations

between parameters and benchmark scores can include both linear and non-linear relationships. There can also be multi-variate associations involved. The user might also want to filter the existing model instantiations by the range of a single benchmark score, or, the combination of multiple benchmark scores. The user might also want to inspect the effects of varying one specific paramter while keeping all the others fixed. The interface should support these interactions intuitively without inducing unnecessary complexity.

Third, the interface should make it convenient and intuitive to compare the embeddings produced by different models. The specific targets of comparison may include the nearest neighbor words of a specific query, the similarity scores assigned by different models to the same pair of words, and, the local and global structures of words in the embedding manifold. For example, given a single word as a query, the user may want to know whether two well-performing models differ in the ranking of nearest neighbors for selected query words, and whether the difference is significant enough to judge the superiority of one model over the other.

Overall, we intended to design a visualization system that could support flexible benchmark settings, lucid representation of parameter influences, and multi-model comparison for parameter selection.

5.4.2 Parallel Coordinates

Parallel coordinate plots [73] map data of multiple dimensions onto a 2-D plot in which the axes are parallel to each other. This visualization technique supports rapid identification of positive/negative relations between data dimensions. A limitation of the technique is that, at any given time, each dimension can have at least two neighboring dimensions, but this is remediable by allowing the user's interactive manipulation. Since one of our requirements above is to facilitate efficient identification of relationships between model parameters and performance scores, we opt to use parallel coordinate plots to depict both kinds of variables.

We note that, in specific, the following features are the highlights of the visualization design and an integral part of the intended user workflow. All of the features below are inherent parts of the design of the original parallel coordinate plots.

- **Filtering:** by brushing along a specific axis, the user is able to narrow down the number of lines being highlighted to a smaller set whose crossing points with the axis are within the brushed range. The brushed region can be rescaled or moved through cursor interactions. This feature is helpful for locating a specific model or set of models through a certain criterion.

- **Flipping:** by double clicking on the axis label on the top of each axis line, the user can have the y-scale of the axis reversed. This feature may help the user reduce the number of line crossings between axes, so as to simplify the pattern recognition process.

- **Coloring:** the lines can use colors to encode an additional dimension to further support the identification of patterns. Alternatively, we can use qualitative color palettes to support the user to easily distinguish between model instantiations.

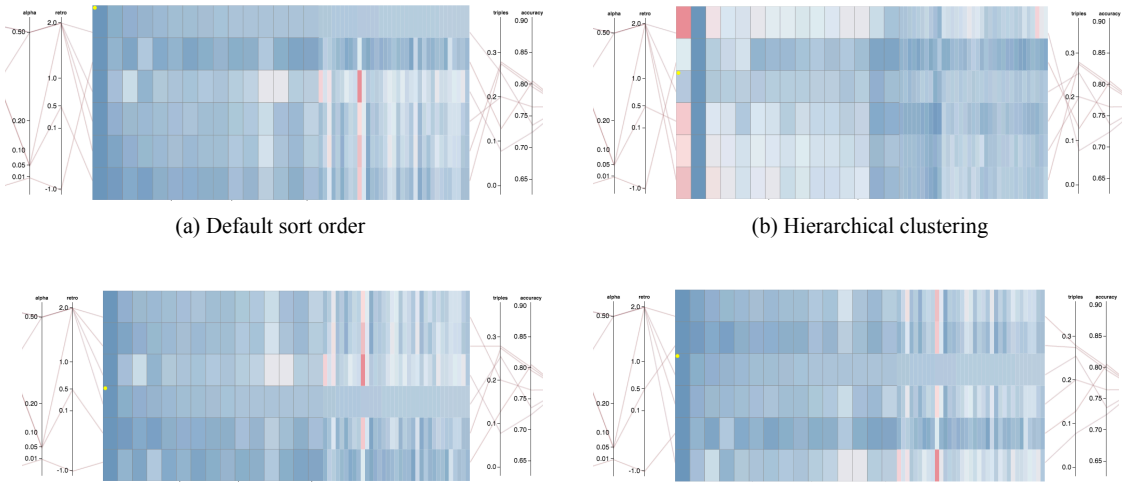
5.4.3 Nearest-neighbor Heatmap

Heatmaps are a great means to use compact space to provide a high information-density summary of data with natural matrix alignment [126]. In our requirement analysis above, we identified the need for comparing the nearest neighbor similarities computed across different models. The quantitative variables to be visually encoded are naturally aligned within a matrix whose axes are model identities and word identities. Therefore, a heatmap is an ideal choice for such requirement.

In our design, the heatmap component not only provides the user with an overview of the similarity results computed by different models, but also allows the user to dive into individual matrix elements for further investigations. As the number of nearest neighbors increases, the difficulty to accurately find and click a specific cell rises as well. Inspired by Tablelens [140], we designed a two-mode display of the matrix columns. Under the "normal" mode, the columns in the matrix are compressed to almost a few pixels, minimizing the space usage, but still allows the user to obtain an overall picture of the similarity score distribution and to spot outliers. Under the "zoomed-in" mode, the columns in the matrix are displayed with a larger width, and the labels are displayed below the bottom row. By default, the top K nearest neighbors of the query word will have their corresponding columns zoomed-in. As the user interacts with the other parts of the interface, pertinent columns will be adjusted to zoom in. This focus switching mechanism optimizes the efficiency of space usage while allowing the user to quickly navigate through the most relevant columns.

Several other design highlights of the heatmap module include:

- **Clustering:** While clustering seems a natural choice for understanding the structure of heatmaps, there are specific meanings of the clusters that are worth pointing out regarding our design. Since the primary focus for our heatmap is for the identification of model disagreement, the clustering of models can simplify the process greatly. In particular, by grouping the model instantiations—rows of the matrix—into clusters using hierarchical clustering, one can further investigate the similarity



(a) Default sort order

(b) Hierarchical clustering

(c) Sorted by model parameters

(d) Sorted by model performance

Figure 5.6: Four sorting modes of the LAMVI-2 heatmap: (a) default mode – models are sorted by loading order; (b) cluster-sort mode – models and words are sorted according to the dendrogram of hierarchical clustering; (c) left-sort mode – models are sorted by the closest control parameter on the LHS parallel coordinate plot; (d) right-sort mode – models are sorted by the performance metric on the RHS parallel coordinate plot.

and dissimialrity of the parameters used to create these model instantiations. Parameters that matter little to the similarity scores can be quickly identified. In addition, the clustering of words—columns of the matrix—can be used in combination with the clustering of model instantiations. This can support a more rapid identification of the points where two model instantiations disagree.

- **Sorting**: By default, the rows of the heatmap are sorted by the order their associated models are loaded into the system. In clustering mode, these rows can be sorted according to the hierarchical clustering dendrogram. In other situations, the user may sort the rows of the matrix by the closest dimension in the parallel-coordinate plot. By sorting, one can eliminate the line crossings between a nearest dimension and the heatmap. Accompanished by brushing on the parallel coordinates, this feature allows the user to inspect the nearest neighbors according to their corresponding model parameter or performance score. See Figure 5.6 for an overview of the four different sorting modes of the heatmap.

- **Filtering**: The behavior of the heatmap rows are synchronized with the axes in the parallel coordinate plot. That means, whenever the user applies brushing to one of

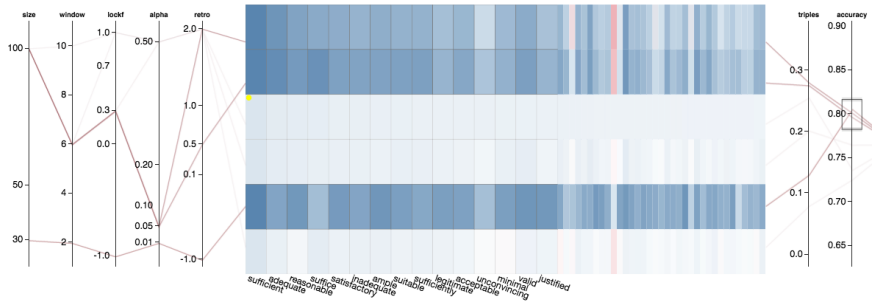


Figure 5.7: Heatmap rows can be filtered and highlighted by brushing axes on the parallel coordinate plot. This allows a focused comparison among models that meet a user-specified filtering criterion.

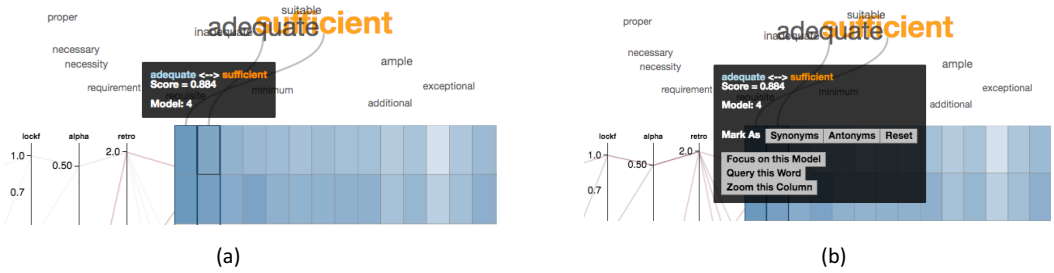


Figure 5.8: Multifunctional tooltip in LAMVI-2 heatmap: (a) compact tooltip shows the similarity of the word pair under focus; (b) expanded tooltip allows the user to assign a label to the word pair, creating user-customized ground truth.

the dimensions in the parallel coordinate plot, the corresponding rows in the heatmap will be highlighted. The non-highlighted rows are still visible but will be applied with a semi-transparent mask. See Figure 5.7 for an example of filtering via brushing.

- **User Labeling**: The heatmap embeds an additional function—supporting the user to create customized ground-truth dataset. This is achieved by selecting a cell in the heatmap and selecting which category the word pair represented by the cell should fall in. For the prototype, the categories include *synonyms*, *antonyms*. However, the labels can be extended to have many different meanings. In the future, user labeling that involves clustering should also be supported. See Figure 5.8 for an illustration on how a tooltip on heatmap can support the creation of user-customized ground truth.

Compared to using parallel coordinate plots for encoding the nearest neighbor similarities, using heatmaps not only provides a fuller overview of the big picture, but also makes

outliers more salient. For example, when all words are more or less scored the same, yet one word is ranked spuriously high by one model, whereas the other models gives it significantly lower ratings, heatmaps expose this different more sharply than parallel coordinate plots and lead to more rapid exploration and decision making.

5.4.4 Embedding Explorer

While the parallel coordinate plot and the heatmap enable rich user interactions, there is a high risk that the user loses track of the spatial distribution of the words of question in the high-dimensional space. This spatial awareness is important because the user needs to rely on spatial exploration to identify more queries, and to understand how words or word clusters shift from one model instantiation to another. For this purpose, in our design, we included an embedding explorer over the top of the parallel coordinate plot and the similarity score heatmap.

The embedding explorer uses t-SNE [163] to visualize the words that are the nearest neighbors of the current active query. Besides the support for basic moving and zooming, it has the following design highlights:

- **Connection to Heatmap:** Since the t-SNE embedding explorer and the heatmap are depicting the same set of words but sharing different UI functions, the user needs to find connections between the two components. As the user interacts with either a cell in the heatmap or a word label in the t-SNE embedding, a connector curve line is drawn between the word label and the corresponding column in the heatmap. The user can selectively turn on/off multiple connector curves, useful for keeping track of the positions of multiple words when model switching or word clustering is performed.

- **Object Constancy:** As the user performs filtering, sorting, and model switching, proper animation is instituted to support the user make sense of the change, i.e., which set of words get inserted into the embedding space; which removed; and which have locations shifted. The t-SNE optimization algorithm randomly initializes the embedding vectors and may cause the vectors to all shrink into the center of the space, which may lead to change blindness during animated transitions. To counter this adversarial effect, we halt the updating of embedding vector locations until the first 150 iterations of the t-SNE optimization are silently completed.

5.4.5 Workflows

The following workflows are designed and implemented in the system. These workflows are described for the purpose of making necessary connections between different modules. We start by describing a general workflow that is intended to be followed in all cases, and then describe individual workflows that may become necessary based on selected conditions.

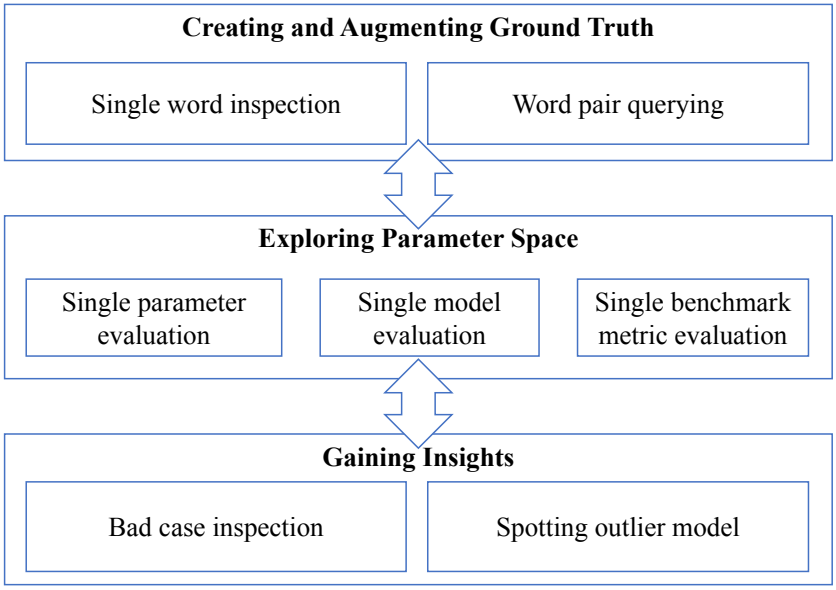


Figure 5.9: User workflow of LAMVI-2.

General Workflow: Figure 5.9 shows the typical user workflow for LAMVI-2, which involves three stages. In the first stage, the user starts by creating and augmenting the ground truth. Without testing a trained model with real queries, the user may have great difficulties coming up with reliable ground truth. If the user only tests with one single model instantiation, many spurious results may show up, still leaving the user with little clue. Therefore, the user creates multiple model instantiations, and once the interface starts cumulatively displaying multiple models, it becomes increasingly obvious to the user what errors are specific to a specific instantiation, and what error is common among all instantiations. Then, by flagging positive and/or negative word pairs, the user is able to aggregate the performance metrics for the marked pairs for each choice of parameter combination, and thus consciously pick through the parameter space and find the right setting. The user-marked word pairs are always highlighted in the subsequent debugging process, drawing the user's attention to these spurious words under different model settings. This, theoreti-

cally, prevents the user from fixating on a selected few pairs and thus reduce over-fitting in the debugging process.

In the second stage, the user explores the parameter space and performs a series of evaluations by reducing the focus on single parameters, models, and benchmark metrics. See below for detailed explanations for each of the sub-workflows.

In the third stage, the user gains insights and makes minor adjustments on the model parameters. This involves performing bad case inspections checking outlier predictions. See below also for detailed explanations for each of the sub-workflows.

Single Parameter Evaluation: When the user wants to see the range of performance one can obtain when one single parameter is varying, she can brush on the corresponding axis on the parallel coordinates plot. This allows the user to filter the set of models down to those whose corresponding parameter falls into the selected range. Another useful pattern is to make the parameter as the only one varying, and making all the other parameters fixed to one specific line. Then the heatmap can be reordered to align with the orders represented by the crossing points on the axis. This step can reveal any disagreement by these models and potentially facilitate the narrowing down of the optimal region of parameter setting. Alternatively, the user may also make one small change to one parameter at a time, and after making the change, the user brushes to filter the model selections down to the only ones before and after the change. This conveniently visualizes the effect of the change.

Single Model Inspection: When the user wants to narrow the focus down to an individual model, she may do so by brushing near the edge of the heatmap, so as to focus on an individual row. This allows the user to inspect the performance of this model.

Single Benchmark Metric Evaluation: The user may be interested in inspecting a single benchmark metric. By visually overviewing the line crossings along the axis, the user may get an intuitive sense of the score distribution under this benchmark. Then the user may inspect how this particular metric correlates with other metrics by checking the line patterns. If the lines are mostly parallel between this and another metric, then it means the two metrics are mostly positively correlated. If the lines have crossings and cross at almost the same point, then the two metrics are mostly negatively correlated. The user may swap the order of the axes in the parallel coordinates plot so as to compare an axis with any other axis. Additionally, the user may reorder the axes so that the axis of a metric of interest is placed closest to the heatmap, and she can then reorder the heatmap rows to align with the magnitudes of scores for that particular metric. This allows the user to conveniently inspects the nearest neighbors of the "good", and "bad" models judged by the current metric.

Single Word Inspection: When the user is interested in particular words, she may

directly search for the word by entering it in the query box and click "Search." The word's nearest neighborhood will be displayed around the word, and their similarity scores will be color-encoded in the central heatmap.

Word Pair Querying: When the user is interested in the similarity and neighborhoods of a pair of words simultaneously, she may directly query the two words together. This allows the user to more easily mark good cases and bad cases as a more diverse set of words are presented. A useful pattern is that the user clicks to highlight two or more words, and switch focus between models. As the highlights preserve themselves during switching, the user can visually track where the words are relocated, and acquire spatial awareness of the change of embedding space across different model instantiations.

Bad Case Inspection: When the user explores the overhead embedding space illustration, she may occasionally discover words that are inappropriately placed in the neighborhood of a query. The user can then click on the word label in the 2-D embedding visualization, which will automatically zoom into the associated column. The user can check the entire column and see if the score is consistently spuriously high, or, filter down to specific rows, and attempt to find out the causes (by parameters) of the undesired score.

Spotting Outlier Model: As the user experiments with different queries or focusing the ranking on different models, the user may suddenly discover certain outliers in the heatmap. While a specific cell has a color that disagrees with the other cells in the same column does not necessarily indicate that the model represented the row it resides in generates a bad output, it is a valid indication of model disagreement and serves as a meaningful signal to draw the user's attention to the query/model. The user can then choose to focus on the outlier model, or, to query the new word to further the investigation.

5.5 Experiment Setup and Evaluation

We evaluate the performance of LAMVI-2 using a controlled lab user study. In the study we provided a simulated model parameter tuning task to the subjects and asked them to try to find the optimal parameter combination. The experiment is aimed to find out how well the LAMVI-2 interface can facilitate the user to identify patterns among model parameters and performance metrics.

5.5.1 Data

We used the Cornell IMDB movie review corpus [132] as the training corpus. There are 100,000 movie reviews in the dataset. The words were converted to lower case and all

punctuations were removed.

As an additional training corpus, we also retrieved the English Wikipedia 2014 full text dump (with 56 million articles and 1.2 billion words). The user has the option to pre-train word embedding vectors on Wikipedia, and continue training on IMDB.

5.5.2 Model

We adopted the word2vec skip-gram model [116] as the word embedding model used for this experiment. To control the scope of the experiment, we have set some of the parameters to be non-adjustable by the user. In specific, we fixed the sample to be 1×10^{-4}, iterations to be 5, and negative samples to be 5. We also allowed the user to select whether to have the vectors pretrained on an external corpus (see *lockf* below) and whether to use an external lexicon to retrofit the vectors (see *retro* below). The retrofitting happens after the model training and is shown to be able to improve the quality of the vectors [43]. The vector update equation for retrofitting is given by:

$$q_i = \frac{\sum_{j:(i,j)\in E} \beta_{ij} q_j + \alpha_i \hat{q}_i}{\sum_{j:(i,j)\in E} \beta_{ij} + \alpha_i} \tag{5.2}$$

where $\hat{q}_i$ is the vector of the word of interest before retrofitting, q_i is the vector after retrofitting, q_j is the vector of a word that has a link to the word of interest in the external lexicon, α_i and β_{ij} are weights, and E is the set of edges between word pairs recorded in the external lexicon. The paraphrase database (PPDB) [47] is used as the semantic lexicon for retrofitting. It contains 8 million English lexical paraphrases, such as *jailed* and *imprisoned*.

- **size:** the number of dimensions in the word vector
- **window:** the length of the sliding window when training the model
- **lockf (lock factor):** whether to lock pre-trained vectors on a different corpus (Wikipedia). Between 0 and 1, the larger this number, the less influence the external corpus has. However, -1 means disabling the external corpus altogether.
- **alpha:** initial learning rate for gradient descent.
- **retro (retrofitting):** whether to use an external dictionary to augment the vectors after training. Between 0 and 2, this number represents α_i in Equation (5.2). The larger this number, the less influence the external lexicon has. However, -1 means disabling the external dictionary altogether.

5.5.3 Procedure

Each lab study session involves a researcher and a participant. The participant is provided with a computer on which the prototype software is installed. Each participant answers a questionnaire about their experience and familiarity with machine learning and word embedding models.

The researcher uses 3 minutes to introduce the system functionality and the task (see "Task" below) to the participant by following a prepared script. Then the participant is allowed 5 minutes to ask clarification questions and experiment with different system features. Once the participant indicates they are ready, the researcher resets the state of the system, and starts automatic logging of the user's interactions.

The participant interacts with the interface with 2 preloaded model instantiations and checks at most 13 additional model instantiations. The participant selects one instantiation as the final submission and ends the interaction section. Finally, Each participant answers a questionnaire with questions taken from the system usability scale (SUS) and answers a few open-ended interview questions from the researcher.

5.5.4 Model Evaluation

We use the average of two metric scores, triples and accuracy, to measure the performance of each model instantiation. The *triples* score, f_T, measures how well a model distinguishes synonyms from antonyms. It is calculated as the average difference between the similarity difference of a set of synonym-antonym triples:

$$f_T = \frac{1}{N}\sum_{i}^{N}(\cos(q_{i,A}, q_{i,B}) - \cos(q_{i,A}, q_{i,C})) \tag{5.3}$$

where $q_{i,A}$ and $q_{i,B}$ are the vectors of a pair of synonyms and $q_{i,A} and q_{i,C}$ are the vectors of a pair of antonyms.

We created the synonym-antonym triples dataset by consulting external lexicons, including SentiWordNet [5], *Oxford American Writer's Thesaurus* [100], and an online thesaurus dictionary.[4] We specifically attempted to gather word pairs that had strong polarity and were neither extremely frequent nor extremely rare in the target corpus. We first identified those words whose absolute labeled polarity score in SentiWordNet is among the top 10%. Then we filtered those words whose occurrence frequencies in the IMDB dataset were between 150 and 250. For these words, we consulted the aforementioned thesauri to

[4] `http://www.thesaurus.com/`.

find their synonyms and antonyms (only one synonym and one antonym were randomly chosen for each word). Therefore, for each word, we obtained a synonym-antonym triple. We randomly sampled 5 triples to be presented to the user as human ground-truth (the "training" set), and we had 35 triples as held-out (the "test" set). See Table 5.1 for an example set of triples.

Table 5.1: Example Synonym-Antonym Triples

A	**B (Synonym to A)**	**C (Antonym to A)**
sufficient	adequate	meager
ghastly	awful	delightful
unconventional	unusual	common
substantial	considerable	insignificant
overbearing	domineering	humble

The *accuracy* score, f_A, is the accuracy of a logistic regression classifier for sentiment polarity classification on the aforementioned IMDB dataset. The dataset has 50,000 documents labeled with sentiment polarity (either positive or negative). These documents are equally partitioned into two groups for training and testing. Within each group there are an equal number of positive samples and negative samples. The features for the documents are the average of the word vectors of that document.

For both f_T and f_A, the user can access the corresponding scores assessed on the training set. The final evaluation of the model was to be assessed on the test set. Figure 5.10 provides a summary of the performance of all candidate models.

5.5.5 System Evaluation

We compare the performance of LAMVI-2 system to that of a baseline system in the user study. The baseline system contains just basic features of loading models, checking performance scores, and inspecting nearest neighborhood by submitting queries. The same set of instructions were given to those users assigned with LAMVI-2 and with the baseline system.

All user interactions were logged. The following metrics are compared:

- training set performance: f_T and f_A evaluated on the training set;
- test set performance: f_T and f_A evaluated on the test set;
- correlation of the user's queries similarities with the triples scores.

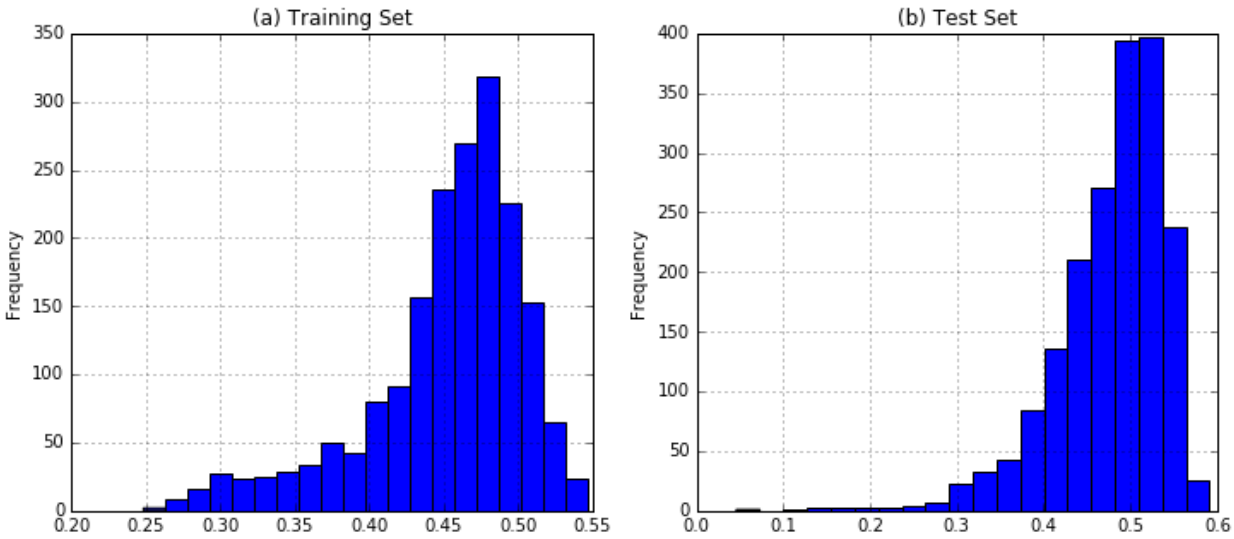


Figure 5.10: Summary of performance of all candidate models.

We want to see if LAMVI-2 is able to outperform the baseline system in the above metrics. We also report how the users actually interact with the LAMVI-2 system. The user interactions of interest include: how a new set of model parameters is selected; how the user interacts with various components of the system; how the user provides additional ground truth; how the final "best-performer" is selected.

5.5.6 Results

12 subjects were recruited. All of them were graduate students in computer science or relevant major. Each subject received a $15 cash payment for participation.

We used a pre-survey to ask the subjects to self-report experience with machine learning and word embedding models. 25% of the subjects reported to be beginner of machine learning, and 75% intermediate to expert level. One third of the subjects have learned about word embedding but have not used it in practice; one third used it as a black box in a project; and one third have done at least a research project on word embedding models.

Table 5.2 summarizes the results for the comparison between LAMVI-2 and the baseline system. On both the training set and the test, the average performance of the models in groups with the baseline and the LAMVI-2 system are reported. LAMVI-2 consistently outperforms the baseline system under the metrics of both f_T and f_A.

Figure 5.11 compares LAMVI-2 with the baseline system in terms of the performance score ($\frac{f_T+f_A}{2}$) for the current best performing model at each step in the debugging process.

Table 5.2: LAMVI-2 Evaluation Results

	Training Set			Test Set		
	triples f_T	accuracy f_A	average	triples f_T	accuracy f_A	average
Random	0.202	0.751	0.477	0.159	0.745	0.452
Baseline	0.277	0.795	0.536	0.203	0.791	0.497
LAMVI-2	**0.295**	**0.814**	**0.554**	**0.223**	**0.810**	**0.517**

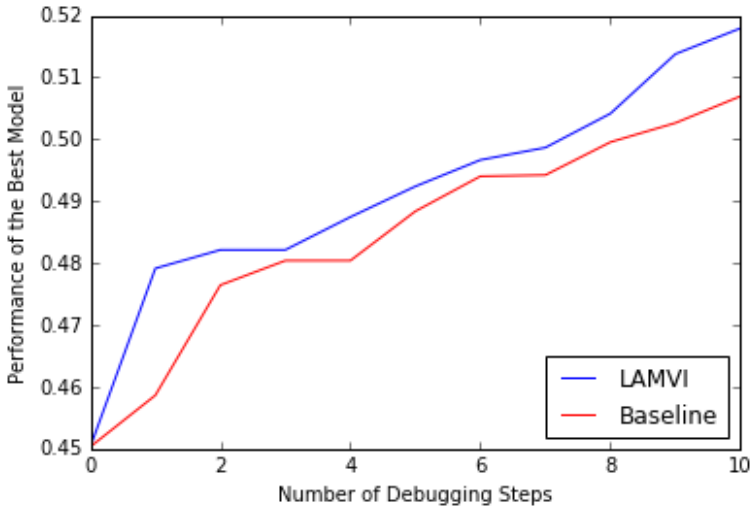


Figure 5.11: Model performnace improvement against the number of models loaded.

The user group with LAMVI-2 manage to improve the model performance faster at the beginning of the debugging process.

With LAMVI-2, the user may query specific words and mark certain word pairs in the neighborhood as synonyms or antonyms. To understand how effective this feature is in supporting the user to select a good model, we examine whether the users' marked synonyms and antonyms correlate well with the test set. Specifically, we look at how well the user-defined triples score correlate well with the actual triples score across various different models attempted by the user. Table 5.3 summarizes the user-flagged word pairs. Only the subjects assigned to use LAMVI are shown in the table. $f_{T,user}$ is the triples score calculated on the user's flagged synonyms and antonyms, based on the user's selected best-performing model; $f_{T,test}$ is the triples score calculated on the held-out triples set; f_A is the accuracy of the sentiment analysis task on the test dataset; and r is the Pearson's correlation between the triples scores calculated using the user's flagged synonym-antonym triples and the held-out triples set over 100 randomly sampled candidate model instantiations. All r values are statistically significant ($p < 0.0001$). This indicates that the users' flagged synonym-antonym triples exhibit similar similarity/dissimilarity patterns as our preselected triples in the held-out dataset. Additionally, a higher correlation (r) generally means that the final user-selected model has a better task performance overall.

Table 5.3: Summary of User-flagged Word Pairs

Subject	**Synonyms**	**Antonyms**	$f_{T,user}$	$f_{T,test}$	f_A	$(f_{T,test} + f_A)/2$	r
P1	14	13	0.175	0.200	0.782	0.491	0.582
P3	6	4	0.254	0.218	0.838	0.528	0.509
P5	23	12	0.252	0.246	0.798	0.522	0.576
P7	23	20	0.236	0.236	0.826	0.531	0.589
P9	13	13	0.326	0.248	0.816	0.532	0.774
P11	17	16	0.257	0.193	0.799	0.496	0.562

During the experiments, we observed how each individual user interacted with LAMVI-2 or the baseline system and took notes. We have made the following significant observations.

First, the user's prior knowledge about the models played an important role. Such prior knowledge originated from the user's prior experience in tuning similar models, and was concentrated on the proper scope of *size*, *window*, and *alpha* (learning rate). For example, users P1, P2, and P7 mentioned that, from prior experience, they knew that *alpha* should not be very big. They opted to start with reducing *alpha* and was able to obtain substantial improvement on both f_T and f_A. Users P2 and P5 mentioned that they knew *window* cannot not be too big or too small. They tuned the parameter towards the middle region (4–8) and

noticed the gradual improvement on the performance, especially with f_A. Several users in the LAMVI group reported that being able to use the parallel coordinates plot helped them confirm such kind of intuitions quickly.

Second, most users developed specific strategies that they used to guide themselves to navigate through the parameter space, and the parallel coordinates plot helped with their parameter searching, especially for the parameters that they were unfamiliar with. For example, during the post-study interview, user P3 mentioned that "I have a search algorithm. I start with a default setting, and vary each parameter at a time, and see if I can get the biggest improvement. Then I fix the one with the biggest improvement, and vary other parameters one by one." User P7 said, "I first applied the prior knowledge that I have. Then I try the medium values for the options that I don't know, and then test the extreme values deviated from the medium values." Similar to these two users, most other users developed a variant of "greedy search" or "gradient descent" algorithm for parameter searching. Note that for the LAMVI users, the parallel coordinate plot was important to these strategies. P5 commented, "It helps tremendously to use the parallel coordinates to tune down the scope of searching. I first look right to the pick the performance I want, and look left to confirm the intuition. I confirmed my intuition that lockf and alpha and retro should be locked to a specific value, and window and size can be fine-tuned."

Third, the nearest-neighbor heatmap was indeed useful for supporting the user to spot outlier models. Whenever the user loads a new model, the system automatically uses the newly loaded model to find the nearest neighbor words to the query and updates the heatmap. When the user loads a model that results in significant disagreement with previously loaded models, there is a dramatic change on the heatmap which is hard to miss by the user. As user P3 mentioned, "When I turn the lockf up, I suddenly see the heatmap change all over, and this helps me know that something bad happens. So I am able to quickly recognize that and revert the change."

In addition to the observations above, we also had the participants comment on how the system could be improved. Participant feedback is summarized below.

- **Removal of models:** Multiple participants have reported that they would benefit from having an additional feature of the system—removing loaded models. In the system instantiation used for the user study, once a model is loaded, it will always stay in the user interface. This may cause a waste of screen space because the user may have determined that a certain set of models are too under-performing and have little reference value. Adding this feature may significantly improve the usability of the system and make the interface less cluttered during the later stage of debugging.

- **Intuitiveness of brushing:** Another common feedback from the participants was that the interaction patterns of brushing on the parallel coordinates and the heatmap was not immediately clear to them. P11 found it difficult to initiate brushing to focus on one or two specific model instantiations. It took multiple participants multiple failed attempts before they understood that the logic behind multi-axes filtering on the parallel coordinates—the model instances were the intersections of the filters, which needed to be initialized or canceled individually. Adding visual hints to the interface and allowing click-and-drag over multiple axes may solve this problem.

- **Intuitiveness of flagging:** P3 and P11 found the interactions of flagging word pairs as synonyms and/or antonyms to be cumbersome. Currently, to flag a word pair, the user has to find the target word in the overhead embedding explorer, and follows the connector to the corresponding column in the heatmap and click, and then in a popped-up window select the associated option. This interaction pattern can be simplified by allowing the user directly click on words in the embedding explorer to flag them.

- **Support on identifying non-linear patterns:** In the experiment, multiple parameters for the model in question had non-linear effects on the performance. For example, the two extremes of *lockf* and *retro* had similar effects while the middle-range values of the two parameters cause a linear change on the performance metric. P5 commented that this non-linear pattern was hard to identify with the parallel coordinates without any external knowledge. We hypothesize that additional histograms showing the distribution of values given a certain filtered output may alleviate this issue.

5.6 Discussions

There are several aspects of the system that we are working to improve.

Scaling up: The current implementation of LAMVI runs fully in the browser and can only support a small corpus and vocabulary. However, the framework and visual tools are designed to be extensible to support full-sized models with millions of words in the vocabulary using a server-client model. Since most interactive visualizations are focused on a watch-list of just a few words, the overhead of logging additional information per training instance is small. Also note that training efficiency is not usually the primary concern for those who are debugging the model for its quality.

Scaling to other embedding models: The proposed framework can be extended to support a full range of embedding models, including GloVe [135], DeepWalk [136], and LINE [160], because they all share the same underlying neural network architecture. Our framework can also be adapted to embedding models with (slightly) more complex structures, such as Doc2Vec [91] and bimodal embedding models [2]. Adapting to these would require making model-dependent modification to the visualization interface, such as adding a new input channel (e.g. document identity, or input of a different modality). However, the nature of inspecting vector similarity, vector interaction, and tracking the ranks of watched candidate items will remain the same.

Scaling to sequential contexts: It is also possible to extend LAMVI to support neural language models that make predictions using sequential contexts. For example, memory networks can "generate" sentences given a few cue words or a piece of computer source code given a few characters [79]. To debug such models, The user may specify the inputs as a sequence of words or characters, and observe, as the model consumes training data, how different candidate words or characters are reranked among the model's predicted probabilistic distribution. One may also look "further into the future", making the model generate N words or characters in a row, and inspecting how the likelihood of generating a given expected output evolves as the training proceeds. However, it can be challenging to locate specific influential training instances in a meaningful way given the complex nature of sequential contexts.

Explaining model behavior: There are many limitations to our current way of defining most influential training instances or features. An important part of our work next is to develop meaningful metrics that distinguish which set of training instances or which aspect of the model configurations is most responsible for a given candidate being ranked higher than another.

Supporting exploratory data analysis: In our system, as the model consumes training instances, a wide variety of information is logged. For example: the ranks of watched vectors, their gradients, and learning rates. When using LAMVI to debug a model, the user may have her own information need. Therefore, providing an exploratory data analysis environment provides the end-user with greater flexibility in terms of generating different visualizations and getting insights from the model's training footprint. For example, the user may define customized grouping of the contexts (e.g. by part-of-speech, or rarity of words), and inspect the influences of these training instances category by category.

Linguistic regularity: Our current implementation also supports inspecting the emergence of linguistic regularity captured by the model. The user may enter queries like "king –queen woman" and observe how the desired candidate, "man", evolves. The user may

also inspect the activation levels of the hidden units given all three words as context.

Model diff: Our current version does not support direct comparison between two model versions trained with different configurations. Such comparison can be potentially very useful, as the user may directly see the effects of changing one hyperparameter. It would also be interesting to enable the user to adjust the model configurations and see what potential impact that configuration may have on the contributions of specific features on-the-fly, which can, nonetheless, be far more challenging than doing diffs on trained models.

Avoiding overfitting: A potential hazard of the presented debugging pattern is that the user may possibly overfit the specific cases that she selects to focus on, and fail to make the model work well on the overall dataset. Therefore, it is important that the user combine such kind of case-specific debugging routines with benchmark-based testing mechanisms (train/validate/test routines) to avoid overfitting. It would also be interesting to develop a recommended workflow/debugging strategy that combines low-level and high-level debugging routines.

5.7 Summary

In this chapter we have described two instantiations of the LAMVI system that supports interactive debugging of neural embedding models. While LAMVI-1 supports the deep inspection of the internal states of an individual model, LAMVI-2 supports intuitive comparison across multiple model instantiations. In the design of LAMVI-2, we combined parallel coordinate plots, heatmaps, and t-SNE embedding visualizations to provide the user with multiple perspectives to compare the model instantiations. To prevent overfitting to small ground-truth datasets, we introduced a mechanism to allow the user to create customized ground-truth and used them to track model performance. We used a lab user study to demonstrate that the system was indeed capable of facilitating the user to obtain insights into the influence of control parameters on the model performance and tune model parameters without overfitting to specific ground truth. The user-selected models with LAMVI-2 are demonstrated to be superior to those selected by users using a baseline system. We also demonstrated additional uses of LAMVI-2 in debugging complex neural language models.

CHAPTER 6

Concluding Remarks

Using artificial intelligence to reduce the complexity of programming and engaging a broader population of potential programmers have become the interest of many research communities. The rapid growth of deep learning techniques, combined with innovations in user interfaces, creates opportunities for improving the user experience of navigating through complex APIs and solving diverse issues. To this end, this thesis has proposed two systems that leverage recent advancement of neural language models to facilitate interactive programming and programming by visual examples respectively.

In addition, during the development of these systems, we have accumulated knowledge and experiences with debugging neural language models. These experiences are then combined into a visual debugging interface that not only grays the so-called "black box" models, but also provide intuitive guidance to model developers.

BIBLIOGRAPHY

[1] Eytan Adar, Mira Dontcheva, and Gierad Laput. Commandspace: Modeling the relationships between tasks, descriptions and features. In *Proceedings of the* 27^{th} *Annual ACM Symposium on User Interface Software and Technology*, UIST '14, pages 167–176. ACM, 2014. ISBN 978-1-4503-3069-5. doi: 10.1145/2642918.2647395.

[2] Miltiadis Allamanis, Daniel Tarlow, Andrew Gordon, and Yi Wei. Bimodal modelling of source code and natural language. In *Proceedings of The* 32^{nd} *International Conference on Machine Learning*, pages 2123–2132, 2015.

[3] Laurence Anthony. Antconc (version 3.2. 2)[computer software]. *Tokyo, Japan: Waseda University*, 2011.

[4] Stanislaw Antol, Aishwarya Agrawal, Jiasen Lu, Margaret Mitchell, Dhruv Batra, C Lawrence Zitnick, and Devi Parikh. Vqa: Visual question answering. In *Proceedings of the IEEE International Conference on Computer Vision*, pages 2425–2433, 2015.

[5] Stefano Baccianella, Andrea Esuli, and Fabrizio Sebastiani. Sentiwordnet 3.0: An enhanced lexical resource for sentiment analysis and opinion mining. In *LREC*, volume 10, pages 2200–2204, 2010.

[6] Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. Neural machine translation by jointly learning to align and translate. *arXiv preprint arXiv:1409.0473*, 2014.

[7] Sushil K. Bajracharya, Joel Ossher, and Cristina V. Lopes. Leveraging usage similarity for effective retrieval of examples in code repositories. In *Proceedings of the* 18^{th} *ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE '10, pages 157–166. ACM, 2010. ISBN 978-1-60558-791-2. doi: 10.1145/1882291.1882316. URL `http://doi.acm.org/10.1145/1882291.1882316`.

[8] Satanjeev Banerjee and Alon Lavie. Meteor: An automatic metric for mt evaluation with improved correlation with human judgments. In *Proceedings of the acl workshop on intrinsic and extrinsic evaluation measures for machine translation and/or summarization*, volume 29, pages 65–72, 2005.

[9] Marco Baroni, Georgiana Dinu, and Germán Kruszewski. Don't count, predict! a systematic comparison of context-counting vs. context-predicting semantic vectors. In *ACL (1)*, pages 238–247, 2014.

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

[11] James Bergstra and Yoshua Bengio. Random search for hyper-parameter optimization. *Journal of Machine Learning Research*, 13(Feb):281–305, 2012.

[12] Pavol Bielik, Veselin Raychev, and Martin Vechev. Programming with "Big Code": Lessons, Techniques and Applications. 1^{st} *Summit on Advances in Programming Languages*, page 41, 2015.

[13] David M Blei, Andrew Y Ng, and Michael I Jordan. Latent dirichlet allocation. *Journal of machine Learning research*, 3(Jan):993–1022, 2003.

[14] Piotr Bojanowski, Edouard Grave, Armand Joulin, and Tomas Mikolov. Enriching word vectors with subword information. *arXiv preprint arXiv:1607.04606*, 2016.

[15] Joel Brandt, Mira Dontcheva, Marcos Weskamp, and Scott R Klemmer. Example-centric programming: Integrating web search into the development environment. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, pages 513–522. ACM, 2010.

[16] Peter F Brown, Peter V Desouza, Robert L Mercer, Vincent J Della Pietra, and Jenifer C Lai. Class-based n-gram models of natural language. *Computational linguistics*, 18(4):467–479, 1992.

[17] Jeffrey Browne, Bongshin Lee, Sheelagh Carpendale, Nathalie Riche, and Timothy Sherwood. Data analysis on interactive whiteboards through sketch-based interaction. In *Proceedings of the ACM International Conference on Interactive Tabletops and Surfaces*, pages 154–157. ACM, 2011.

[18] Ozan Caglayan, Loïc Barrault, and Fethi Bougares. Multimodal attention for neural machine translation. *arXiv preprint arXiv:1609.03976*, 2016.

[19] Alex J Champandard. Semantic style transfer and turning two-bit doodles into fine artworks. *arXiv preprint arXiv:1603.01768*, 2016.

[20] William O Chao, Tamara Munzner, and Michiel van de Panne. Poster: Rapid pen-centric authoring of improvisational visualizations with napkinvis. *Posters Compendium InfoVis*, 2(1):2, 2010.

[21] Shaunak Chatterjee, Sudeep Juvekar, and Koushik Sen. Sniff: A search engine for java using free-form queries. In *Fundamental Approaches to Software Engineering*, pages 385–400. Springer, 2009.

[22] Xinlei Chen and C Lawrence Zitnick. Learning a recurrent visual representation for image caption generation. *arXiv preprint arXiv:1411.5654*, 2014.

[23] Zhe Chen, Michael Cafarella, and Eytan Adar. Diagramflyer: A search engine for data-driven diagrams. In *Proceedings of the 24th International Conference on World Wide Web*, pages 183–186. ACM, 2015.

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

[25] Jan K Chorowski, Dzmitry Bahdanau, Dmitriy Serdyuk, Kyunghyun Cho, and Yoshua Bengio. Attention-based models for speech recognition. In *Advances in Neural Information Processing Systems*, pages 577–585, 2015.

[26] Sagnik Ray Choudhury and Clyde Lee Giles. An architecture for information extraction from figures in digital libraries. In *WWW (Companion Volume)*, pages 667–672, 2015.

[27] Sagnik Ray Choudhury, Shuting Wang, and C Lee Giles. Scalable algorithms for scholarly figure mining and semantics. In *Proceedings of the International Workshop on Semantic Big Data*, page 1. ACM, 2016.

[28] Jason Chuang, Christopher D Manning, and Jeffrey Heer. Termite: Visualization techniques for assessing textual topic models. In *Proceedings of the International Working Conference on Advanced Visual Interfaces*, pages 74–77. ACM, 2012.

[29] Jason Chuang, Sonal Gupta, Christopher Manning, and Jeffrey Heer. Topic model diagnostics: Assessing domain relevance via topical alignment. In *Proceedings of the 30th International Conference on Machine Learning (ICML-13)*, pages 612–620, 2013.

[30] Ronan Collobert and Jason Weston. A unified architecture for natural language processing: Deep neural networks with multitask learning. In *Proceedings of the 25th international conference on Machine learning*, pages 160–167. ACM, 2008.

[31] Ronan Collobert, Jason Weston, Léon Bottou, Michael Karlen, Koray Kavukcuoglu, and Pavel Kuksa. Natural language processing (almost) from scratch. *Journal of Machine Learning Research*, 12(Aug):2493–2537, 2011.

[32] Corinna Cortes and Vladimir Vapnik. Support-vector networks. *Machine learning*, 20(3):273–297, 1995.

[33] Weiwei Cui, Shixia Liu, Zhuofeng Wu, and Hao Wei. How hierarchical topics evolve in large text corpora. *Visualization and Computer Graphics, IEEE Transactions on*, 20(12):2281–2290, 2014.

[34] Bhavana Dalvi, William W Cohen, and Jamie Callan. Collectively representing semi-structured data from the web. In *Proceedings of the Joint Workshop on Automatic Knowledge Base Construction and Web-Scale Knowledge Extraction*, pages 7–12. Association for Computational Linguistics, 2012.

[35] Yann Dauphin, Harm de Vries, and Yoshua Bengio. Equilibrated adaptive learning rates for non-convex optimization. In *Advances in Neural Information Processing Systems*, pages 1504–1512, 2015.

[36] Scott Deerwester, Susan T. Dumais, George W. Furnas, Thomas K. Landauer, and Richard Harshman. Indexing by latent semantic analysis. *Journal of the American Society for Information Science*, 41(6):391–407, 1990. ISSN 1097-4571. doi: 10.1002/(SICI)1097-4571(199009)41:6⟨391::AID-ASI1⟩3.0.CO;2-9. URL **http://dx.doi.org/10.1002/(SICI)1097-4571(199009)41:6<391::AID-ASI1>3.0.CO;2-9**.

[37] Yuntian Deng, Anssi Kanervisto, and Alexander M Rush. What you get is what you see: A visual markup decompiler. *arXiv preprint arXiv:1609.04938*, 2016.

[38] Aditya Desai, Sumit Gulwani, Vineet Hingorani, Nidhi Jain, Amey Karkare, Mark Marron, Sailesh R, and Subhajit Roy. Program synthesis using natural language. In *Proceedings of the* 38^{th} *International Conference on Software Engineering*, ICSE '16, pages 345–356. ACM, 2016. ISBN 978-1-4503-3900-1. doi: 10.1145/2884781.2884786. URL **http://doi.acm.org/10.1145/2884781.2884786**.

[39] Alexandre Devert. *matplotlib Plotting Cookbook*. Packt Publishing Ltd, 2014.

[40] Jacob Devlin, Hao Cheng, Hao Fang, Saurabh Gupta, Li Deng, Xiaodong He, Geoffrey Zweig, and Margaret Mitchell. Language models for image captioning: The quirks and what works. *arXiv preprint arXiv:1505.01809*, 2015.

[41] Hubert L Dreyfus, Stuart E Drey-fus, and Lotfi A Zadeh. Mind over machine: The power of human intuition and expertise in the era of the computer. *IEEE Expert*, 2 (2):110–111, 1987.

[42] John Duchi, Elad Hazan, and Yoram Singer. Adaptive subgradient methods for online learning and stochastic optimization. *Journal of Machine Learning Research*, 12(Jul):2121–2159, 2011.

[43] Manaal Faruqui, Jesse Dodge, Sujay K Jauhar, Chris Dyer, Eduard Hovy, and Noah A Smith. Retrofitting word vectors to semantic lexicons. *arXiv preprint arXiv:1411.4166*, 2014.

[44] Manaal Faruqui, Yulia Tsvetkov, Pushpendre Rastogi, and Chris Dyer. Problems with evaluation of word embeddings using word similarity tasks. *arXiv preprint arXiv:1605.02276*, 2016.

[45] Danyel Fisher, Badrish Chandramouli, Robert DeLine, Jonathan Goldstein, Andrei Aron, Mike Barnett, John C Platt, James F Terwilliger, and John Wernsing. Tempe: an interactive data science environment for exploration of temporal and streaming data. Technical report, MSR-TR-2014–148, 2014.

[46] Joel Galenson, Philip Reames, Rastislav Bodik, Björn Hartmann, and Koushik Sen. Codehint: Dynamic and interactive synthesis of code snippets. In *Proceedings of the* 36^{th} *International Conference on Software Engineering*, pages 653–663. ACM, 2014.

[47] Juri Ganitkevitch, Benjamin Van Durme, and Chris Callison-Burch. Ppdb: The paraphrase database. In *HLT-NAACL*, pages 758–764, 2013.

[48] Jinglun Gao, Yin Zhou, and Kenneth E Barner. Classifying chart images with sparse coding. In *SPIE Defense, Security, and Sensing*, pages 83650G–83650G. International Society for Optics and Photonics, 2012.

[49] Yoav Goldberg. A primer on neural network models for natural language processing. *arXiv preprint arXiv:1510.00726*, 2015.

[50] Yoav Goldberg and Omer Levy. word2vec explained: deriving mikolov et al.'s negative-sampling word-embedding method. *arXiv:1402.3722 [cs, stat]*, February 2014. URL **http://arxiv.org/abs/1402.3722**. arXiv: 1402.3722.

[51] Brian Granger, Steven Silvester, Json Grout, Fernando Perez, Sylvain Corlay, Oelsen Cameron Colbert, Chris, David Willmer, and Afshin Darian. Jupyterlab: Building blocks for interactive computing. SciPy 2016, 2016. URL **http://archive.ipython.org/media/SciPy2016JupyterLab.pdf**.

[52] Samuel Gratzl, Alexander Lex, Nils Gehlenborg, Hanspeter Pfister, and Marc Streit. Lineup: Visual analysis of multi-attribute rankings. *IEEE transactions on visualization and computer graphics*, 19(12):2277–2286, 2013.

[53] Alex Graves, Abdel-rahman Mohamed, and Geoffrey Hinton. Speech recognition with deep recurrent neural networks. In *Acoustics, speech and signal processing (icassp), 2013 ieee international conference on*, pages 6645–6649. IEEE, 2013.

[54] Klaus Greff, Rupesh K Srivastava, Jan Koutník, Bas R Steunebrink, and Jürgen Schmidhuber. Lstm: A search space odyssey. *IEEE transactions on neural networks and learning systems*, 2016.

[55] Xiaodong Gu, Hongyu Zhang, Dongmei Zhang, and Sunghun Kim. Deep api learning. *arXiv preprint arXiv:1605.08535*, 2016.

[56] Ankush Gupta and Prashanth Mannem. From image annotation to image description. In *International Conference on Neural Information Processing*, pages 196–204. Springer, 2012.

[57] T. Gvero and V. Kuncak. Interactive synthesis using free-form queries. In *2015 IEEE/ACM* 37^{th} *IEEE International Conference on Software Engineering*, volume 2, pages 689–692, May 2015. doi: 10.1109/ICSE.2015.224.

[58] Tihomir Gvero and Viktor Kuncak. Synthesizing Java expressions from free-form queries. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 416–432. ACM, 2015.

[59] Benjamin V Hanrahan, Gregorio Convertino, and Les Nelson. Modeling problem difficulty and expertise in stackoverflow. In *Proceedings of the ACM 2012 conference on Computer Supported Cooperative Work Companion*, pages 91–94. ACM, 2012.

[60] Björn Hartmann, Daniel MacDougall, Joel Brandt, and Scott R Klemmer. What would other programmers do: suggesting solutions to error messages. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, pages 1019–1028. ACM, 2010.

[61] Christopher G Healey et al. Perception in visualization. *Retrieved February*, 10: 2008, 2007.

[62] Tony Hey, Anthony JG Hey, and Gyuri Pápay. *The computing universe: a journey through a revolution*. Cambridge University Press, 2014.

[63] Abram Hindle, Earl T. Barr, Zhendong Su, Mark Gabel, and Premkumar Devanbu. On the naturalness of software. In *Proceedings of the* 34^{th} *International Conference on Software Engineering*, ICSE '12, pages 837–847. IEEE Press, 2012. ISBN 978-1-4673-1067-3. URL **http://dl.acm.org/citation.cfm?id=2337223.2337322**.

[64] Geoffrey E Hinton, James L Mcclelland, and David E Rumelhart. Distributed representations, parallel distributed processing: explorations in the microstructure of cognition, vol. 1: foundations, 1986.

[65] Micah Hodosh, Peter Young, and Julia Hockenmaier. Framing image description as a ranking task: Data, models and evaluation metrics. *Journal of Artificial Intelligence Research*, 47:853–899, 2013.

[66] Thomas Hofmann. Probabilistic latent semantic indexing. In *Proceedings of the 22nd annual international ACM SIGIR conference on Research and development in information retrieval*, pages 50–57. ACM, 1999.

[67] Reid Holmes and Gail C Murphy. Using structural context to recommend source code examples. In *Proceedings of the* 27^{th} *international conference on Software engineering*, pages 117–125. ACM, 2005.

[68] Chun-Hung Hsiao, Michael Cafarella, and Satish Narayanasamy. Using web corpus statistics for program analysis. In *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications*, OOPSLA '14, pages 49–65, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-2585-1. doi: 10.1145/2660193.2660226. URL **http://doi.acm.org/10.1145/2660193.2660226**.

[69] Weihua Huang and Chew Lim Tan. A system for understanding imaged infographics and its applications. In *Proceedings of the 2007 ACM symposium on Document engineering*, pages 9–18. ACM, 2007.

[70] Weihua Huang, Chew Lim Tan, and Wee Kheng Leow. Model-based chart image recognition. In *International Workshop on Graphics Recognition*, pages 87–99. Springer, 2003.

[71] Weihua Huang, Ruizhe Liu, and C-L Tan. Extraction of vectorized graphical information from scientific chart images. In *Ninth International Conference on Document Analysis and Recognition (ICDAR 2007)*, volume 1, pages 521–525. IEEE, 2007.

[72] J. D. Hunter. Matplotlib: A 2D graphics environment. *Computing In Science & Engineering*, 9(3):90–95, 2007.

[73] Alfred Inselberg. The plane with parallel coordinates. *The visual computer*, 1(2): 69–91, 1985.

[74] Ian Jolliffe. *Principal component analysis*. Wiley Online Library, 2002.

[75] Armand Joulin, Edouard Grave, Piotr Bojanowski, and Tomas Mikolov. Bag of tricks for efficient text classification. *arXiv preprint arXiv:1607.01759*, 2016.

[76] Rafal Jozefowicz, Oriol Vinyals, Mike Schuster, Noam Shazeer, and Yonghui Wu. Exploring the limits of language modeling. *arXiv preprint arXiv:1602.02410*, 2016.

[77] Andrej Karpathy. The unreasonable effectiveness of recurrent neural networks, 2015. Available at: **http://karpathy.github.io/2015/05/21/rnn-effectiveness/**.

[78] Andrej Karpathy and Li Fei-Fei. Deep visual-semantic alignments for generating image descriptions. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 3128–3137, 2015.

[79] Andrej Karpathy, Justin Johnson, and Fei-Fei Li. Visualizing and understanding recurrent networks. *arXiv preprint arXiv:1506.02078*, 2015.

[80] Iman Keivanloo, Juergen Rilling, and Ying Zou. Spotting working code examples. In *Proceedings of the* 36^{th} *International Conference on Software Engineering*, pages 664–675. ACM, 2014.

[81] Diederik Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

[82] Ryan Kiros, Ruslan Salakhutdinov, and Richard S Zemel. Multimodal neural language models. In *ICML*, volume 14, pages 595–603, 2014.

[83] Krugle. Krugle Code Search. `http://www.krugle.com/`.

[84] Joseph B Kruskal. Multidimensional scaling by optimizing goodness of fit to a nonmetric hypothesis. *Psychometrika*, 29(1):1–27, 1964.

[85] Harold W Kuhn. The hungarian method for the assignment problem. *Naval research logistics quarterly*, 2(1-2):83–97, 1955.

[86] Todd Kulesza, Simone Stumpf, Weng-Keen Wong, Margaret M Burnett, Stephen Perona, Andrew Ko, and Ian Oberst. Why-oriented end-user debugging of naive bayes text classification. *ACM Transactions on Interactive Intelligent Systems (TiiS)*, 1(1):2, 2011.

[87] Girish Kulkarni, Visruth Premraj, Vicente Ordonez, Sagnik Dhar, Siming Li, Yejin Choi, Alexander C Berg, and Tamara L Berg. Babytalk: Understanding and generating simple image descriptions. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 35(12):2891–2903, 2013.

[88] Polina Kuznetsova, Vicente Ordonez, Alexander C Berg, Tamara L Berg, and Yejin Choi. Collective generation of natural image descriptions. In *Proceedings of the 50th Annual Meeting of the Association for Computational Linguistics: Long Papers-Volume 1*, pages 359–368. Association for Computational Linguistics, 2012.

[89] Cody Kwok, Oren Etzioni, and Daniel S Weld. Scaling question answering to the web. *ACM Transactions on Information Systems (TOIS)*, 19(3):242–262, 2001.

[90] Essi Lahtinen, Kirsti Ala-Mutka, and Hannu-Matti Järvinen. A study of the difficulties of novice programmers. In *Acm Sigcse Bulletin*, volume 37, pages 14–18. ACM, 2005.

[91] Quoc V Le and Tomas Mikolov. Distributed representations of sentences and documents. *arXiv preprint arXiv:1405.4053*, 2014.

[92] Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. Deep learning. *Nature*, 521 (7553):436–444, 2015.

[93] Bongshin Lee, Petra Isenberg, Nathalie Henry Riche, and Sheelagh Carpendale. Beyond mouse and keyboard: Expanding design considerations for information visualization interactions. *IEEE Transactions on Visualization and Computer Graphics*, 18(12):2689–2698, 2012.

[94] Bongshin Lee, Rubaiat Habib Kazi, and Greg Smith. Sketchstory: Telling more engaging stories with data through freeform sketching. *IEEE Transactions on Visualization and Computer Graphics*, 19(12):2416–2425, 2013.

[95] Omer Levy and Yoav Goldberg. Neural word embedding as implicit matrix factorization. In *Advances in neural information processing systems*, pages 2177–2185, 2014.

[96] Omer Levy, Yoav Goldberg, and Ido Dagan. Improving distributional similarity with lessons learned from word embeddings. *Transactions of the Association for Computational Linguistics*, 3:211–225, 2015.

[97] Siming Li, Girish Kulkarni, Tamara L Berg, Alexander C Berg, and Yejin Choi. Composing simple image descriptions using web-scale n-grams. In *Proceedings of the Fifteenth Conference on Computational Natural Language Learning*, pages 220–228. Association for Computational Linguistics, 2011.

[98] Chin-Yew Lin. Rouge: A package for automatic evaluation of summaries. In *Text summarization branches out: Proceedings of the ACL-04 workshop*, volume 8. Barcelona, Spain, 2004.

[99] Tsung-Yi Lin, Michael Maire, Serge Belongie, James Hays, Pietro Perona, Deva Ramanan, Piotr Dollár, and C Lawrence Zitnick. Microsoft coco: Common objects in context. In *European Conference on Computer Vision*, pages 740–755. Springer, 2014.

[100] Christine A Lindberg. *Oxford American writer's thesaurus*. Oxford University Press, USA, 2012.

[101] Erik Linstead, Sushil Bajracharya, Trung Ngo, Paul Rigor, Cristina Lopes, and Pierre Baldi. Sourcerer: Mining and searching internet-scale software repositories. *Data Mining and Knowledge Discovery*, 18(2):300–336, 2009.

[102] Greg Little and Robert C Miller. Keyword programming in Java. *Automated Software Engineering*, 16(1):37–71, 2009.

[103] Shixia Liu, Michelle X Zhou, Shimei Pan, Weihong Qian, Weijia Cai, and Xiaoxiao Lian. Interactive, topic-based visual text summarization and analysis. In *Proceedings of the 18th ACM conference on Information and knowledge management*, pages 543–552. ACM, 2009.

[104] Minh-Thang Luong, Hieu Pham, and Christopher D Manning. Effective approaches to attention-based neural machine translation. *arXiv preprint arXiv:1508.04025*, 2015.

[105] Jock Mackinlay. Automating the design of graphical presentations of relational information. *Acm Transactions On Graphics (Tog)*, 5(2):110–141, 1986.

[106] David Mandelin, Lin Xu, Rastislav Bodík, and Doug Kimelman. Jungloid mining: Helping to navigate the API jungle. *ACM SIGPLAN Notices*, 40(6):48–61, 2005.

[107] Junhua Mao, Wei Xu, Yi Yang, Jiang Wang, Zhiheng Huang, and Alan Yuille. Deep captioning with multimodal recurrent neural networks (m-rnn). *arXiv preprint arXiv:1412.6632*, 2014.

[108] Mikaël Mayer, Gustavo Soares, Maxim Grechkin, Vu Le, Mark Marron, Alex Polozov, Rishabh Singh, Ben Zorn, and Sumit Gulwani. User interaction models for disambiguation in programming by example. In *28th ACM User Interface Software and Technology Symposium*, 2015.

[109] Collin McMillan, Mark Grechanik, Denys Poshyvanyk, Chen Fu, and Qing Xie. Exemplar: A source code search engine for finding highly relevant applications. *Software Engineering, IEEE Transactions on*, 38(5):1069–1087, 2012.

[110] Collin Mcmillan, Denys Poshyvanyk, Mark Grechanik, Qing Xie, and Chen Fu. Portfolio: Searching for relevant functions and their usages in millions of lines of code. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 22 (4):37, 2013.

[111] Gonzalo Gabriel Méndez, Miguel A Nacenta, and Sebastien Vandenheste. ivolver: Interactive visual language for visualization extraction and reconstruction. In *Proceedings of the 2016 CHI Conference on Human Factors in Computing Systems*, pages 4073–4085. ACM, 2016.

[112] Rada Mihalcea and Dragomir Radev. *Graph-based natural language processing and information retrieval*. Cambridge University Press, 2011.

[113] Tomáš Mikolov. *Language Modeling for Speech Recognition in Czech*. PhD thesis, Masters thesis, Brno University of Technology, 2007.

[114] Tomas Mikolov, Jiri Kopecky, Lukas Burget, Ondrej Glembek, et al. Neural network based language models for highly inflective languages. In *2009 IEEE International Conference on Acoustics, Speech and Signal Processing*, pages 4725–4728. IEEE, 2009.

[115] Tomas Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. Efficient estimation of word representations in vector space. *arXiv preprint arXiv:1301.3781*, 2013.

[116] Tomas Mikolov, Ilya Sutskever, Kai Chen, Greg S Corrado, and Jeff Dean. Distributed Representations of Words and Phrases and their Compositionality. In *Advances in Neural Information Processing Systems 26*, pages 3111–3119. Curran Associates, Inc., 2013.

[117] Tomas Mikolov, Wen-tau Yih, and Geoffrey Zweig. Linguistic regularities in continuous space word representations. In *HLT-NAACL*, volume 13, pages 746–751, 2013.

[118] George A Miller. WordNet: a lexical database for English. *Communications of the ACM*, 38(11):39–41, 1995.

[119] Alon Mishne, Sharon Shoham, and Eran Yahav. Typestate-based semantic code search over partial programs. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '12, pages 997–1016, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1561-6. doi: 10.1145/2384616.2384689. URL **http://doi.acm.org/10.1145/2384616.2384689**.

[120] Margaret Mitchell, Xufeng Han, Jesse Dodge, Alyssa Mensch, Amit Goyal, Alex Berg, Kota Yamaguchi, Tamara Berg, Karl Stratos, and Hal Daumé III. Midge: Generating image descriptions from computer vision detections. In *Proceedings of the 13th Conference of the European Chapter of the Association for Computational Linguistics*, pages 747–756. Association for Computational Linguistics, 2012.

[121] Paritosh Mittal, Mayank Vatsa, and Richa Singh. Composite sketch recognition via deep network-a transfer learning approach. In *2015 International Conference on Biometrics (ICB)*, pages 251–256. IEEE, 2015.

[122] Andriy Mnih and Geoffrey E. Hinton. A scalable hierarchical distributed language model. In D. Koller, D. Schuurmans, Y. Bengio, and L. Bottou, editors, *Advances in Neural Information Processing Systems 21*, pages 1081–1088. Curran Associates, Inc., 2009.

[123] Frederic Morin and Yoshua Bengio. Hierarchical probabilistic neural network language model. In *AISTATS*, volume 5, pages 246–252. Citeseer, 2005.

[124] Lili Mou, Rui Men, Ge Li, Lu Zhang, and Zhi Jin. On End-to-End Program Generation from User Intention by Deep Neural Networks. *arXiv preprint arXiv:1510.07211*, 2015.

[125] Lili Mou, Ge Li, Lu Zhang, Tao Wang, and Zhi Jin. Convolutional neural networks over tree structures for programming language processing. In *Proceedings of the Thirtieth AAAI Conference on Artificial Intelligence*, 2016.

[126] Tamara Munzner. *Visualization analysis and design*. CRC Press, 2014.

[127] Daniel G Murray. *Tableau your data!: fast and easy visual analysis with tableau software*. John Wiley & Sons, 2013.

[128] Seyed Mehdi Nasehi, Jonathan Sillito, Frank Maurer, and Chris Burns. What makes a good code example?: A study of programming q&a in stackoverflow. In *Software Maintenance (ICSM), 2012* 28^{th} *IEEE International Conference on*, pages 25–34. IEEE, 2012.

[129] Richard R Nelson and Sidney G Winter. *An evolutionary theory of economic change*. Harvard University Press, 2009.

[130] Tung Thanh Nguyen, Anh Tuan Nguyen, Hoan Anh Nguyen, and Tien N Nguyen. A statistical semantic language model for source code. In *Proceedings of the 2013*

9^{th} *Joint Meeting on Foundations of Software Engineering*, pages 532–542. ACM, 2013.

[131] Stephen Oney and Joel Brandt. Codelets: Linking interactive documentation and example code in the editor. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, pages 2697–2706. ACM, 2012.

[132] Bo Pang, Lillian Lee, and Shivakumar Vaithyanathan. Thumbs up?: sentiment classification using machine learning techniques. In *Proceedings of the ACL-02 conference on Empirical methods in natural language processing-Volume 10*, pages 79–86. Association for Computational Linguistics, 2002.

[133] Kishore Papineni, Salim Roukos, Todd Ward, and Wei-Jing Zhu. Bleu: a method for automatic evaluation of machine translation. In *Proceedings of the 40th annual meeting on association for computational linguistics*, pages 311–318. Association for Computational Linguistics, 2002.

[134] Hao Peng, Lili Mou, Ge Li, Yuxuan Liu, Lu Zhang, and Zhi Jin. Building program vector representations for deep learning. In *Knowledge Science, Engineering and Management*, pages 547–553. Springer, 2015.

[135] Jeffrey Pennington, Richard Socher, and Christopher D Manning. Glove: Global vectors for word representation. In *EMNLP*, volume 14, pages 1532–1543, 2014.

[136] Bryan Perozzi, Rami Al-Rfou, and Steven Skiena. Deepwalk: Online learning of social representations. In *Proceedings of the 20th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 701–710. ACM, 2014.

[137] Chris Piech, Jonathan Huang, Andy Nguyen, Mike Phulsuksombati, Mehran Sahami, and Leonidas Guibas. Learning program embeddings to propagate feedback on student code. In *Proceedings of the* 32^{nd} *International Conference on Machine Learning (ICML)*, pages 1093–1102, 2015.

[138] V Shiv Naga Prasad, Behjat Siddiquie, Jennifer Golbeck, and Larry S Davis. Classifying computer generated charts. In *2007 International Workshop on Content-Based Multimedia Indexing*, pages 85–92. IEEE, 2007.

[139] Mukund Raghothaman, Yi Wei, and Youssef Hamadi. Swim: Synthesizing what i mean. *arXiv preprint arXiv:1511.08497*, 2015.

[140] Ramana Rao and Stuart K Card. The table lens: merging graphical and symbolic representations in an interactive focus+ context visualization for tabular information. In *Proceedings of the SIGCHI conference on Human factors in computing systems*, pages 318–322. ACM, 1994.

[141] Veselin Raychev, Martin Vechev, and Eran Yahav. Code completion with statistical language models. In *Proceedings of the* 35^{th} *ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '14, pages 419–428, New

York, NY, USA, 2014. ACM. ISBN 978-1-4503-2784-8. doi: 10.1145/2594291.2594321. URL **http://doi.acm.org/10.1145/2594291.2594321**.

[142] Veselin Raychev, Martin Vechev, and Andreas Krause. Predicting Program Properties from "Big Code". In *Proceedings of the* 42^{nd} *Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '15, pages 111–124, New York, NY, USA, 2015. ACM. ISBN 978-1-4503-3300-9. doi: 10.1145/2676726.2677009. URL **http://doi.acm.org/10.1145/2676726.2677009**.

[143] Mohammad Raza, Sumit Gulwani, and Natasa Milic-Frayling. Compositional program synthesis from natural language and examples. In *Proceedings of the* 24^{th} *International Conference on Artificial Intelligence*, pages 792–800. AAAI Press, 2015.

[144] Steven P Reiss. Semantics-based code search. In *Software Engineering, 2009. ICSE 2009. IEEE* 31^{st} *International Conference on Software Engineering*, pages 243–253. IEEE, 2009.

[145] Xin Rong. word2vec parameter learning explained. *arXiv preprint arXiv:1411.2738*, 2014.

[146] Xin Rong, Zhe Chen, Qiaozhu Mei, and Eytan Adar. Egoset: Exploiting word ego-networks and user-generated ontology for multifaceted set expansion, 2016.

[147] Naiyana Sahavechaphan and Kajal Claypool. XSnippet: Mining for sample code. *ACM Sigplan Notices*, 41(10):413–430, 2006.

[148] Wojciech Samek, Alexander Binder, Grégoire Montavon, Sebastian Bach, and Klaus-Robert Müller. Evaluating the visualization of what a deep neural network has learned. *arXiv preprint arXiv:1509.06321*, 2015.

[149] Manolis Savva, Nicholas Kong, Arti Chhajta, Li Fei-Fei, Maneesh Agrawala, and Jeffrey Heer. Revision: Automated classification, analysis and redesign of chart images. In *Proceedings of the 24th annual ACM symposium on User interface software and technology*, pages 393–402. ACM, 2011.

[150] Mike Schuster and Kuldip K Paliwal. Bidirectional recurrent neural networks. *IEEE Transactions on Signal Processing*, 45(11):2673–2681, 1997.

[151] Mingyan Shao and Robert P Futrelle. Recognition and classification of figures in pdf documents. In *International Workshop on Graphics Recognition*, pages 231–242. Springer, 2005.

[152] Noah Siegel, Zachary Horvitz, Roie Levin, Santosh Divvala, and Ali Farhadi. Figureseer: Parsing result-figures in research papers. In *European Conference on Computer Vision*, pages 664–680. Springer, 2016.

[153] Carson Sievert and Kenneth E Shirley. Ldavis: A method for visualizing and interpreting topics. In *Proceedings of the workshop on interactive language learning, visualization, and interfaces*, pages 63–70, 2014.

[154] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556*, 2014.

[155] Richard Socher, Cliff C Lin, Chris Manning, and Andrew Y Ng. Parsing natural scenes and natural language with recursive neural networks. In *Proceedings of the 28th international conference on machine learning (ICML-11)*, pages 129–136, 2011.

[156] Richard Socher, Andrej Karpathy, Quoc V Le, Christopher D Manning, and Andrew Y Ng. Grounded compositional semantics for finding and describing images with sentences. *Transactions of the Association for Computational Linguistics*, 2: 207–218, 2014.

[157] Elliot Soloway and James C Spohrer. *Studying the novice programmer*. Psychology Press, 2013.

[158] Fabian M Suchanek, Gjergji Kasneci, and Gerhard Weikum. Yago: A large ontology from wikipedia and wordnet. *Web Semantics: Science, Services and Agents on the World Wide Web*, 6(3):203–217, 2008.

[159] Kai Sheng Tai, Richard Socher, and Christopher D Manning. Improved semantic representations from tree-structured long short-term memory networks. *arXiv preprint arXiv:1503.00075*, 2015.

[160] Jian Tang, Meng Qu, Mingzhe Wang, Ming Zhang, Jun Yan, and Qiaozhu Mei. Line: Large-scale information network embedding. In *Proceedings of the 24th International Conference on World Wide Web*, pages 1067–1077. International World Wide Web Conferences Steering Committee, 2015.

[161] Suresh Thummalapenta and Tao Xie. Parseweb: A programmer assistant for reusing open source code on the web. In *Proceedings of the* 22^{nd} *IEEE/ACM international conference on Automated software engineering*, pages 204–213. ACM, 2007.

[162] Joseph Turian, Lev Ratinov, and Yoshua Bengio. Word representations: a simple and general method for semi-supervised learning. In *Proceedings of the 48th annual meeting of the association for computational linguistics*, pages 384–394. Association for Computational Linguistics, 2010.

[163] Laurens Van der Maaten and Geoffrey Hinton. Visualizing data using t-sne. *Journal of Machine Learning Research*, 9(2579-2605):85, 2008.

[164] Ramakrishna Vedantam, C Lawrence Zitnick, and Devi Parikh. Cider: Consensus-based image description evaluation. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 4566–4575, 2015.

[165] Oriol Vinyals, Alexander Toshev, Samy Bengio, and Dumitru Erhan. Show and tell: Lessons learned from the 2015 mscoco image captioning challenge. 2016.

[166] John Walkenbach. *Excel 2010 power programming with VBA*, volume 6. John Wiley & Sons, 2010.

[167] Colin Ware. *Information visualization: perception for design*. Elsevier, 2012.

[168] Doug Wightman, Zi Ye, Joel Brandt, and Roel Vertegaal. Snipmatch: Using source code context to enhance snippet retrieval and parameterization. In *Proceedings of the 25th Annual ACM Symposium on User Interface Software and Technology*, UIST '12, pages 219–228. ACM, 2012. ISBN 978-1-4503-1580-7. doi: 10.1145/2380116.2380145. URL **http://doi.acm.org/10.1145/2380116.2380145**.

[169] Leland Wilkinson. *The grammar of graphics*. Springer Science & Business Media, 2006.

[170] Huijuan Xu and Kate Saenko. Ask, attend and answer: Exploring question-guided spatial attention for visual question answering. *arXiv preprint arXiv:1511.05234*, 2015.

[171] Kelvin Xu, Jimmy Ba, Ryan Kiros, Kyunghyun Cho, Aaron Courville, Ruslan Salakhutdinov, Richard S Zemel, and Yoshua Bengio. Show, attend and tell: Neural image caption generation with visual attention. *arXiv preprint arXiv:1502.03044*, 2 (3):5, 2015.

[172] Mark Yatskar, Lucy Vanderwende, and Luke Zettlemoyer. See no evil, say no evil: Description generation from densely labeled images. *Lexical and Computational Semantics (* SEM 2014)*, page 110, 2014.

[173] Yunwen Ye and Gerhard Fischer. Reuse-conducive development environments. *Automated Software Engineering*, 12(2):199–235, 2005.

[174] Kuat Yessenov, Shubham Tulsiani, Aditya Menon, Robert C. Miller, Sumit Gulwani, Butler Lampson, and Adam Kalai. A colorful approach to text processing by example. In *Proceedings of the* 26th *Annual ACM Symposium on User Interface Software and Technology*, UIST '13, pages 495–504. ACM, 2013. ISBN 978-1-4503-2268-3. doi: 10.1145/2501988.2502040. URL **http://doi.acm.org/10.1145/2501988.2502040**.

[175] Jason Yosinski, Jeff Clune, Anh Nguyen, Thomas Fuchs, and Hod Lipson. Understanding neural networks through deep visualization. *arXiv preprint arXiv:1506.06579*, 2015.

[176] Lei Yu, Karl Moritz Hermann, Phil Blunsom, and Stephen Pulman. Deep learning for answer sentence selection. *arXiv preprint arXiv:1412.1632*, 2014.

[177] YP Zhou and Chew Lim Tan. Learning-based scientific chart recognition. In *4th IAPR International Workshop on Graphics Recognition, GREC*, pages 482–492. Citeseer, 2001.