A Language-Agnostic Model for Semantic Source Code Labeling

Ben Gelman
Two Six Labs, LLC.
Arlington, Virginia, USA
ben.gelman@twosixlabs.com

Bryan Hoyle
Two Six Labs, LLC.
Arlington, Virginia, USA
bryan.hoyle@twosixlabs.com

Jessica Moore
Two Six Labs, LLC.
Arlington, Virginia, USA
jessica.moore@twosixlabs.com

Joshua Saxe
Sophos
Fairfax, Virginia, USA
joshua.saxe@sophos.com

David Slater
Two Six Labs, LLC.
Tacoma, Washington, USA
david.slater@twosixlabs.com

ABSTRACT

Code search and comprehension have become more difficult in recent years due to the rapid expansion of available source code. Current tools lack a way to label arbitrary code at scale while maintaining up-to-date representations of new programming languages, libraries, and functionalities. Comprehensive labeling of source code enables users to search for documents of interest and obtain a high-level understanding of their contents. We use Stack Overflow code snippets and their tags to train a language-agnostic, deep convolutional neural network to automatically predict semantic labels for source code documents. On Stack Overflow code snippets, we demonstrate a mean area under ROC of 0.957 over a long-tailed list of 4,508 tags. We also manually validate the model outputs on a diverse set of unlabeled source code documents retrieved from Github, and obtain a top-1 accuracy of 86.6%. This strongly indicates that the model successfully transfers its knowledge from Stack Overflow snippets to arbitrary source code documents.

CCS CONCEPTS

• **Computing methodologies** → **Artificial intelligence**; **Machine learning**; *Natural language processing*; *Machine learning approaches*; Neural networks;

KEYWORDS

deep learning, source code, natural language processing, multilabel classification, semantic labeling, crowdsourcing

ACM Reference Format:
Ben Gelman, Bryan Hoyle, Jessica Moore, Joshua Saxe, and David Slater. 2018. A Language-Agnostic Model for Semantic Source Code Labeling. In *Proceedings of the 1st International Workshop on Machine Learning and Software Engineering in Symbiosis (MASES '18), September 3, 2018, Montpellier, France.* ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/3243127.3243132

Publication rights licensed to ACM. ACM acknowledges that this contribution was authored or co-authored by an employee, contractor or affiliate of the United States government. As such, the Government retains a nonexclusive, royalty-free right to publish or reproduce this article, or to allow others to do so, for Government purposes only.

MASES '18, September 3, 2018, Montpellier, France

© 2018 Copyright held by the owner/author(s). Publication rights licensed to ACM.

ACM ISBN 978-1-4503-5972-6/18/09…$15.00
https://doi.org/10.1145/3243127.3243132

1 INTRODUCTION

In recent years, the quantity of available source code has been growing exponentially [10]. Code reuse at this scale is predicated on understanding and searching through a massive number of projects and source code documents. The ability to generate meaningful, semantic labels is key to comprehending and searching for relevant source code, especially as the diversity of programming languages, libraries, and code content continues to expand.

The search functionality for current large code repositories, such as GitHub [12] and SourceForge [31], will match queried terms in the source code, comments, or documentation of a project. More sophisticated search approaches have shown better performance in retrieving relevant results, but they often insufficiently handle scale, breadth, or ease of use. Santanu and Prakash [26] develop pattern languages for C and PL/AS that allow users to write generic code-like schematics. Although the schematics locate specific source code constructs, they do not capture the general functionality of a program and scale poorly to large code corpora. Bajracharya et al. [3] develop a search engine called Sourcerer that enhances keyword search by extracting features from its code corpus. Sourcerer scales well to large corpora, but it is still hindered by custom language-specific parsers. Suffering from a similar problem, Exemplar [24] is a system that tracks the flow of data through various API calls in a project. Exemplar also uses the documentation for projects/API calls in order to match a user's keywords. Recent applied works have similar shortcomings [5] [17] [29] [32]. Creating a solution that operates across programming languages, libraries, and projects is difficult due to the complexity of modeling such a huge variety of code.

As a step in that direction, we present a novel framework for generating labels for source code of arbitrary language, length, and domain. Using a machine learning approach, we capitalize on a wealth of crowdsourced data from Stack Overflow (SO) [25], a forum that provides a constantly growing source of code snippets that are user-labeled with programming languages, tool sets, and functionalities. Prior works have attempted to predict a single label for an SO post [19] [33] using both the post's text and source code as input. To our knowledge, our work is the first to use Stack Overflow to predict exclusively on source code. Additionally, prior methods do not attempt multilabel classification, which becomes a significant issue when labeling realistic source code documents

Code Snippet

```
detectors = [
    ('noise',
     segment.NoiseDetector()),
    ('logistic regression',
     segment.MLDetector()),
    ('logistic - auto',
     segment.MLDetector(
         transform='autocorrelation'))]
for name, d in detectors:
    print name
    foo(d)
    print
print 'Mean Fixed EER =', np.mean(fixed_eers)
print 'Mean Free EER =', np.mean(free_eers)
```

Model Output

label	certainty
python	0.809
scikit-learn	0.517
machine-learning	0.128

Figure 1: An example prediction of our model. The input code snippet is on the left, while the predicted labels and their raw certainties are on the right. Keyword matching on the predicted labels would not have been able to locate this code.

instead of brief SO snippets. Our approach utilizes SO's code snippets to simultaneously model thousands of concepts and predict on previously unseen source code, as demonstrated in Fig. 1.

We construct a deep convolutional neural network that directly processes source code documents of arbitrary length and predicts their functionality using pre-existing Stack Overflow tags. As users ask questions about new programming languages and tools, the model can be retrained to maintain up-to-date representations.

Our contributions are as follows:

- First work, to our knowledge, to introduce a baseline for multilabel tag prediction on Stack Overflow posts.
- Convolutional neural network architecture that can handle arbitrary length source code documents and is agnostic to programming language.
- State-of-the-art top-1 accuracy (79% vs 65% [33]) for predicting tags on Stack Overflow posts, using only code snippets as input.
- Approach that enables tagging of source code corpora external to Stack Overflow, which is validated by a human study.

We organize the rest of the paper as follows: section 2 discusses related works, section 3 details data preprocessing and correction, section 4 explains our neural network architecture and validation, section 5 displays our results, section 6 presents challenges and limitations, and section 7 considers future work.

2 RELATED WORK

Due to the parallels between source code and natural language [14] [1], we find that recent work in the natural language processing (NLP) domain is relevant to our problem. Modern NLP approaches have generated state-of-the-art results with long short-term memory neural networks (LSTMs) and convolutional neural networks (CNNs). Sundermeyer, Schlüter, and Ney [34] have shown that LSTMs perform better than n-grams for modeling word sequences, but the vocabulary size for word-level models is often large, requiring a massive parameter space. Kim, Jernite, Sontag, and Rush [16] show that by combining a character-level CNN with an LSTM, they can achieve comparable results while having 60% fewer parameters. Further work shows that CNNs are able to achieve state-of-the-art performance without the training time and data required for LSTMs [8]. In the source code domain, however, prior work has utilized a wide variety of methods.

In 1991, Maarek, Berry, and Kaiser [22] recognized that there was a lack of usable code libraries. Libraries were difficult to find, adapt, and integrate without proper labeling, and locating components functionally close to a given topic posed a challenge. The authors developed an information retrieval approach leveraging the co-occurrence of neighboring terms in code, comments, and documentation.

More recently, Kuhn, Ducasse, and Gírba [18] apply Latent Semantic Indexing (LSI) and hierarchical clustering in order to analyze source code vocabulary without the use of external documentation. LSI-based methods have had success in the code comprehension domain, including document search engines [4] and IDE-integrated topic modeling [11]. Although the method seems to perform well, labeling an unseen source code document requires reclustering the entire dataset. This is a significant setback for maintaining a constantly growing corpus of labeled documents.

In the context of source code labeling, supervised methods are mostly unexplored. A critical issue in this task is the massive amount of labeled data required to create the model. A few efforts have recognized Stack Overflow for its wealth of crowdsourced data. Saxe, Turner, and Blokhin [28] search for Stack Overflow posts containing strings found in malware binaries, and use the retrieved tags to label the binaries. Kuo [19] attempts to predict tags on SO posts by computing the co-occurrence of tags and words in each post. He achieves a 47% top-1 accuracy, which in this context is the task of predicting only one tag per post.

Clayton and Byrne [33] also attempt to predict a tag for SO posts. They invest a great deal of effort in feature extraction inspired by ACT-R's declarative memory retrieval mechanisms [2]. Utilizing logistic regression, they achieve a 65% top-1 accuracy.

In this work, we generate a more complex machine learning model than those present in previous attempts. Because we intend to generalize our model to source code files, we make our tests stricter by only using the actual code inside Stack Overflow posts as inputs to the model. Despite the information loss from not taking advantage of the entire post text, we still further improve on the performance of prior work and obtain a 78.7% top-1 accuracy.

3 DATA

The primary goal of our work is to create a machine learning system that will classify the functionality of source code. We achieve this by leveraging Stack Overflow, a large, public question and answer forum focused on computer programming. Users can ask a question, provide a response, post code snippets, and vote on posts. The SO dataset provides several advantages in particular: a huge quantity of code snippets; a wide set of tags that cover concepts, tools, and functionalities; and a platform that is constantly updated by users asking and answering questions about new technologies. Due to the complexity of the data, we use this section to discuss the data's characteristics and our preprocessing procedures in detail.

Figure 2: A Stack Overflow thread with a question and answer. The thread's tags are boxed in red and the code snippets are boxed in blue. For the purpose of training our model, the tags are the output labels and the code snippets are the input features. We can see from this example that the longer snippets look like valid code, while the shorter snippets are not as useful.

Fig. 2 is an example of a Stack Overflow thread. Users who ask a question are allowed to attach a maximum of five tags. Although there is a long list of previously used tags, users are free to enter any text. The tags are often chosen with programming language, concepts, and functionality in mind. The tags for this example, boxed in red, are "python," "list," and "slice." Additionally, any user is allowed to block off text as a code snippet. In this example, the user providing an answer uses many code snippets, which have been boxed in blue. Although the code snippets may describe a particular functionality, they do not necessarily represent a complete or syntactically correct program.

Our initial intuition is that the code snippets can simply be input into a machine learning model with the user-created tags as labels. This trained model would then be able to accept any source code and provide tags as output. As we further analyze the data, several questions need to first be resolved, including how to associate tags with snippets, what constitutes a single code sample, and which data should be filtered from the dataset.

Stack Overflow's threads are the fundamental pieces of our training data. The publicly available SO data dump provides over 70 million source code snippets with labels that would be useful for

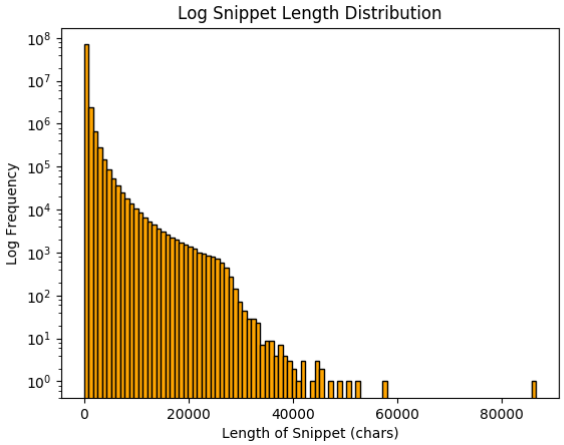

Figure 3: The distribution of snippet lengths in the full dataset, with frequencies logarithmically scaled. Although short code snippets are extremely common, they have limited value.

real world projects. Because the tags are selected at the thread level while snippets occur in individual posts, we assign the thread's tags to each post in that thread. Since a single post can have many code snippets, we choose to concatenate the snippets using newline characters as separators in order to preserve a user's post as a single idea.

Although these transformations ensure that a post will suffice as input to a language-level model, they do not guarantee the usefulness of the snippets themselves. The following section will address several problems with short, uninformative code snippets, user error in tagging posts and generating code with the correct functionality, and the long-tailed distribution of unusable tags.

3.1 Statistics and Data Correction

As of December 15, 2016, the Stack Overflow data dump contains 24,158,127 posts that have at least one code snippet, 73,934,777 individual code snippets, and 46,663 total tags. Despite the large amount of data, there is a severe long-tailed phenomenon that is common in many online communities [13]. The distributions of code-snippet length and number of tags per post are of particular importance to our problem.

Fig. 3 shows the distribution of individual snippet lengths, measured in number of characters, throughout Stack Overflow. As one would expect, the longer snippets are many orders of magnitude less frequent than the shorter snippets. Fig. 4 further demonstrates that, of the many short snippets, there is a huge quantity that are empty strings or are only a few characters long. There are several reasons why these snippets are poor choices for training data. First, a single character is usually not descriptive enough to characterize multiple tags. Saying that 'x' is a good indicator of python, machine learning, and databases does not make sense. Going back to Fig. 2, we can also see that the short snippets are often references to code, but not valid code themselves.

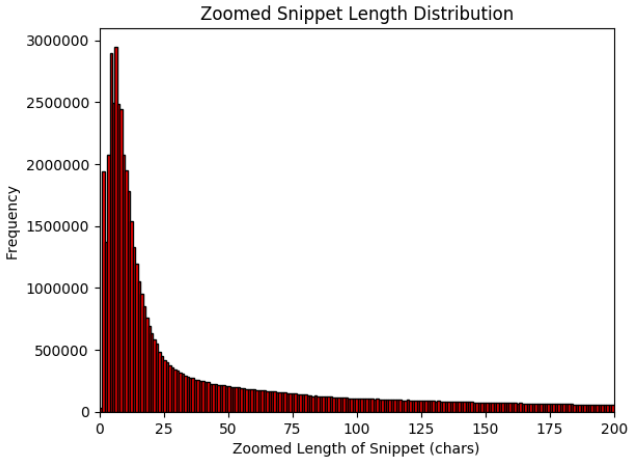

Figure 4: A zoomed view of the snippet length distribution, with 1 bin equal to 1 character. There are many strings that are empty or only a few characters long.

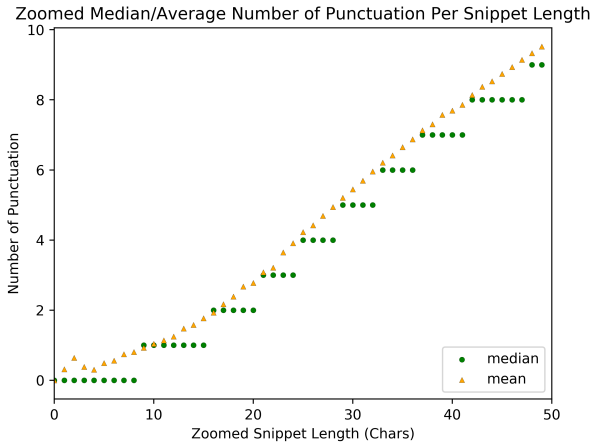

Figure 5: The mean and median number of punctuation marks at different snippet lengths. At a snippet length of 10 characters, the mean and median number of punctuation marks is 1, indicating a reasonable choice for minimum snippet length.

In order to avoid cutting out snippets at an uninformed threshold, we investigate snippets of different lengths in more detail. We found punctuation to be a good indicator of code usefulness in short snippets. The occurrence of punctuation means that we are more likely to see programming language constructs such as "variable = value" or "class.method." However, simply removing all snippets without punctuation is not viable because of valuable alphanumeric keywords and punctuation-free code ("call WriteConsole"), so we instead decide to filter based on a threshold length. Fig. 5 shows the median and mean number of punctuation marks for different snippet lengths. At a snippet length of 10 characters, the mean and median are both greater than one, so we filter out all snippets that are length 9 or below from the data.

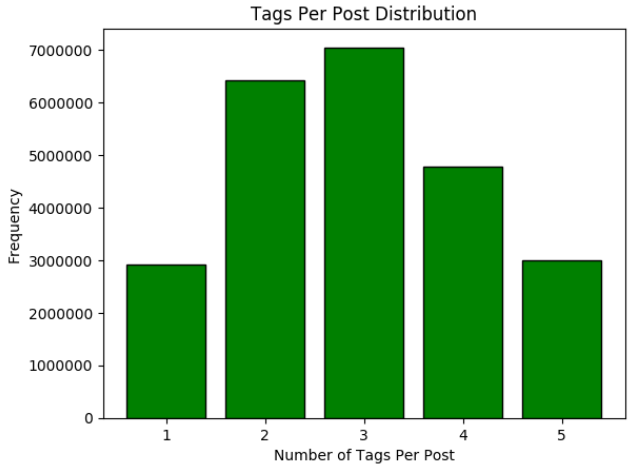

Figure 6: Distribution of tags per post. All posts on Stack Overflow must have at least one tag, but there is a maximum of five tags, resulting in missing labels.

Additionally, Fig. 6 shows the distribution of tags per post. As stated previously, Stack Overflow allows a maximum of five tags for any given post. Although most posts contain three tags, there is still a significant number of posts with fewer tags. The combined effect from a high quantity of posts that have few tags and an enforced maximum creates a "missing label phenomenon." This is the situation where a given post is not tagged with *all* of the functionalities or concepts actually described in the post. This is a non-trivial challenge for machine learning models because a code snippet is considered a negative example for a given label if that label is missing.

Users can also add errors to the training data by simply being wrong about their tags or posted code on Stack Overflow. Because users can vote based on the quality of a post, we can use scores as an indicator for incorrectly tagged or poorly coded posts. Fig. 7 shows the distribution of scores for posts that have at least one code snippet. We cut all posts with negative scores from the training data. Although we considered cutting posts with zero score because they had not been validated by other users via voting, we ultimately choose to keep them because the score distribution shows that there is a large amount of data with zero score.

After filtering the data for the snippet length and score thresholds, one problem remained with the set of valid labels. Because users are allowed to enter any text as a tag for their posts, there is a long-tailed distribution of tags that are rarely used. Table 1 displays the magnitude of the problem. In the first 4,508 tags, the amount of posts per tag drops from 2.5 million to just 1,000. In order to enable a 99% / 1% training/test split and still have 10 positive labels per tag to estimate performance, we cut off tags with fewer than one thousand positive samples.

In the following section, we explain how we construct our models and perform validation using the snippet, score, and tag-filtered data.

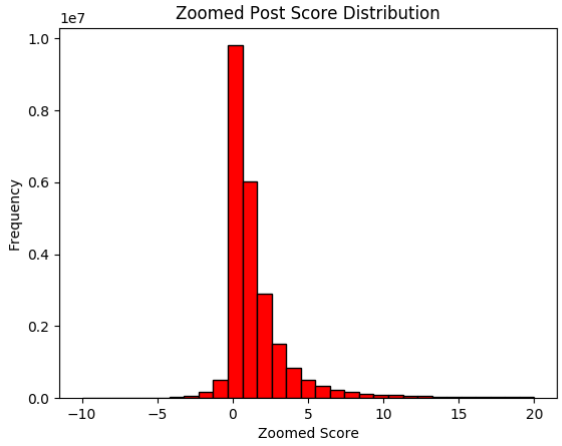

Figure 7: The distribution of scores on Stack Overflow posts. Negative scores are often the result of poorly worded questions, incorrectly tagged posts, or flawed code snippets, so we filter them out of the training set. We keep zero-scored snippets because they may not have been viewed enough to be voted on.

Table 1: Rankings are based on the number of posts that are labeled with a tag, after filtering data for snippet and score thresholds. This shows that the majority of tags have too few samples to train and validate a machine learning model.

Rank	Tag	# of Posts
1	javascript	2,585,182
8	html	1,279,137
73	apache	99,377
751	web-config	10,056
4,508	simplify	1,000
16,986	against	100
46,663	db-charmer	1

4 METHODOLOGY

Our motivations for using neural networks in this work are severalfold. As discussed in the introduction, convolutional neural networks have shown state-of-the-art performance in natural language tasks with less computation than LSTMs [16] [8]. Both natural language and source code tasks must model structure, semantic meaning, and context.

Neural networks also have the ability to efficiently handle multi-label classification problems: rather than training M classifiers for M different output labels, the output layer of a neural network can have M nodes, simultaneously providing predictions for multiple labels. This enables the neural network to learn features that are common across labels, whereas individual classifiers must learn those relationships separately.

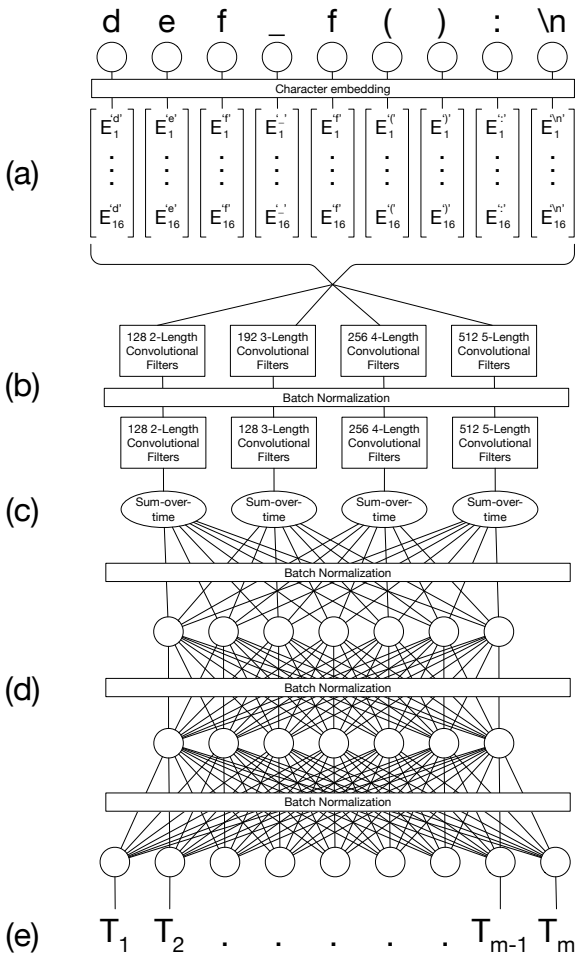

Figure 8: An overview of the neural network architecture. (a) The characters from a given code snippet are converted to real-valued vectors using a *character embedding*. (b) We use stacked convolutional filters of different lengths with ReLU activations over the matrix of embeddings. (c) We perform sum-over-time pooling on the output of each stacked convolution. (d) A flattened vector is fed into two fully-connected, batch-normalized, dense layers with ReLU activations. (e) Each output node uses a logistic activation function to produce a value from 0 to 1, representing the probability of a given label.

4.1 Neural Network Architecture

Fig. 8 gives an overview of the neural network architecture. In part (a) of Fig. 8, we use a character embedding to transform each printable character into a 16-dimensional real-valued vector. We chose character embeddings over more commonly used word embeddings for multiple reasons. Creating an embedding for every word in the source code domain is problematic because of the massive set of unique identifiers. Forming a dictionary from words only seen in the training set will not generalize, and using all possible identifiers will be infeasible to optimize. The neural network only needs

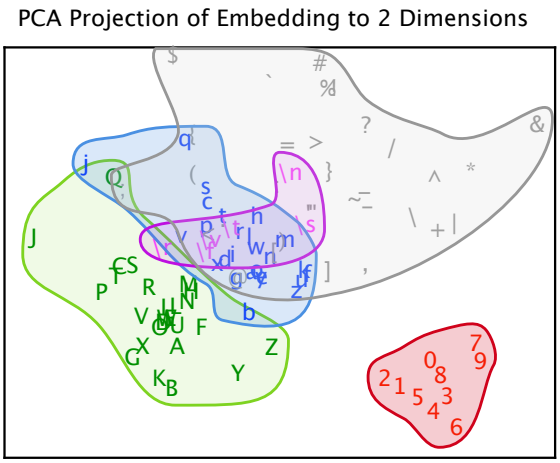

Figure 9: A two-dimensional projection of the character embedding vectors that are optimized during model training. The model generates clear spatial relationships between various sets of characters. The separation between uppercase letters, lowercase letters, symbols, and numbers is of particular interest. In general, meaningful spatial relationships significantly improve the features extracted by the convolutional layers.

to optimize 100 embeddings when using the printable characters. Additionally, the character embeddings are able to function on any text, allowing the model to predict on source code without the use of language-specific parsers or features.

In order to provide an intuition of the character embedding, we use PCA to project the 16-dimensional embedding vectors down to two dimensions, as displayed in Fig. 9. This figure indicates that the model generates salient spatial relationships between the embedded characters during optimization, which is critical to the performance of the convolutional layers. The convolutions are able to preserve information about words and sequences by sliding over the embedding vectors of consecutive characters. We stack two convolutional layers of the same size for various filter lengths, which generates a stacked convolution matrix. Using sum-over-time pooling on the stacked convolution matrix allows us to obtain a fixed-length vector regardless of the initial input size.

After two batch normalized dense layers, the last layer has a logistic activation for each neuron in order to output the probability of a tag occurring. The network is trained on binary vectors containing a 1 for every tag that occurs for a given code snippet and 0 otherwise. We use binary cross-entropy as our loss function.

4.2 Validation Setup

Since we train the model on Stack Overflow and predict on arbitrary source code, we must validate the model in both domains. On the SO data, we use a hold-out test-set strategy so that the model can be evaluated on previously unseen data. In the source code domain, we perform human validation to verify the accuracy of the modelâĂŹs outputs.

4.3 Stack Overflow Validation

To validate the neural network on Stack Overflow, we tested a number of multilabel test set stratification algorithms. Stratification based on k-fold cross-validation, which is a standard technique for binary and multiclass classification tasks, cannot be directly applied to the SO multilabel classification problem due to classes not being disjoint. Furthermore, due to the class imbalance caused by using a long-tailed tag distribution for labels, random stratification produces partitions of the data that do not generate good estimates for multilabel problems [30] [36]. In particular, the label counts for the top tag and the 4,508th tag differ by 3 orders of magnitude, which can result in classes with very few positive labels for the test set.

Since deep CNN models take a long time to train and benefit from large datasets, we want to avoid cross validation and use as much of the dataset as possible to train our model. Our goal is to generate a 98% / 1% / 1% train/validation/test split that still provides a good estimate of performance. With an ideal stratification, this would ensure that even the rarest tags (with 1000 samples each) would have 10 samples in the validation and test sets, which is sufficient for estimating performance. On our dataset, this would result in about 240,000 samples in validation and test sets.

Multilabel stratification begins with the m-by-q label matrix Y, where m is the number of samples in the dataset D, q is the number of labels in the set of labels Λ, and $Y[i,j] = 1$ where sample i has label j, and $Y[i,j] = 0$ otherwise. The goal is to generate a partition $\{D^1, \ldots, D^k\}$ of D that fulfills certain desirable properties. First, the size of each partition should match a designated ratio of samples, in our case, $\frac{|D^{\text{train}}|, |D^{\text{test}}|, |D^{\text{val}}|}{|D|} = (0.98, 0.1, 0.1)$. Additionally, the proportion of positive examples of each label in each partition should be the same; i.e.,

$$\forall s \in \{train, test, val\}, \forall j \in \Lambda : \frac{\sum_{i \in D^s} Y[i,j]}{|D^s|} = c_j$$

where c_j is the proportion of positive examples of label j in D.

Labelset stratification [36] considers each combination of labels, denoted labelsets, as a unique label and then performs standard k-fold stratification on those labelsets. This works well for multilabel problems where each labelset appears sufficiently often. However, this does not optimize for individual label counts, which is a problem for datasets like SO that include rare labels and rare label combinations. We found that iterative stratification [30], a greedy method that specifically focuses on balancing the label counts for rare labels, produced the best validation and test sets. To produce our partition, we ran iterative stratification twice with a 99%/1% split, which resulted in a 98.01%/0.99%/ 0.1% train/validation/test split.

4.4 Source Code Validation

Validating the model's performance on source code poses a different challenge because of the lack of labeled documents. In order to obtain results, we performed human validation on source code that is randomly sampled from GitHub [12]. Specifically, we ran a script

```
i=1
initial=`date +%s`
actual=`date +%s`
while [ $(($actual - $initial)) -le 60 ]
do
        ./ct.out fec0::3 "abcdefghijklmnopqrstuvwxyz"
        echo "Caso : $i - $(($actual - $initial))"
        i=`expr $i + 1`
        actual=`date +%s`
done
```

Do the following tags make sense?

Tag	Yes	No	Unsure
bash	◉	○	○
shell	◉	○	○
linux	○	○	◉
unix	◉	○	○
scripting	◉	○	○
date	◉	○	○

Figure 10: The GUI for human validation of model outputs on source code documents.

Table 2: Mean, median, and standard deviation of tag AUCs for each model.

Model	Mean	Median	Stdev
Embedding CNN	0.957	0.974	0.048
Embedding Logistic Regression	0.751	0.759	0.099
N-gram Logistic Regression	0.514	0.502	0.093

to download the master branches of random GitHub projects via the site's API until we had 146,408 individual files. We sampled 20 files for each of the following extensions, resulting in a total of 200 source code documents: [py, xml, java, html, c, js, sql, asm, sh, cpp]. Note that the extensions were not presented to the users and that they do not inform the predictions of the model. We created a GUI, displayed in Fig. 10, that presents the top labels and asks users if they agree with, disagree with, or are unsure about each label. There were a total of 3 reviewers, each of whom answered the questions on the GUI for all 200 source code documents. We remove the unsure answers and use simple voting among the remaining ratings to produce ground truth and compute an ROC curve.

5 RESULTS

On the Stack Overflow data, we first calculated the top-1 accuracy previously used by Kuo [19] and Clayton and Byrne [33]. We obtain a 78.7% top-1 accuracy, which is a significant improvement over the previous best of 65%.

However, we found that metric to be lacking: it only checks if the model's top prediction is in the SO post's tag set. Our goal is to predict many tags pertinent to a source code document, not just its primary tag. Because our work is introducing the multilabel tag prediction problem on Stack Overflow code snippets, we train multiple baseline models to demonstrate the significance of our convolutional neural network architecture. In order to evaluate the results, we computed the area under ROC (AUC) for each individual tag. This is a reasonable evaluation because it demonstrates the performance of each model across the entire set of tags.

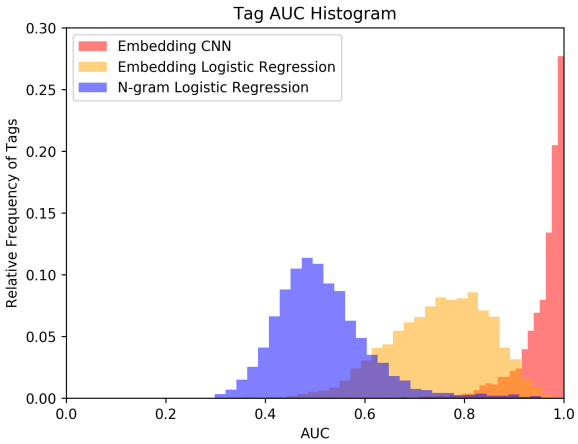

Figure 11: The distribution of tag AUCs for each model. Because our dataset uses 4,508 labels, there are 4,508 AUCs binned and plotted for each model. This graph demonstrates how well each model performs across all the labels.

We used two additional models as baselines for this problem. The first model performs logistic regression on a bag of n-grams. This model obtains the 75,000 most common n-grams (using n=1,2,3) from the training set to use as features. The second model performs logistic regression on a character embedding of the input code using an embedding dimension of 8. We choose these two models as baselines because they test two different types of featurizations and they are able to efficiently train and predict on multilabel problems.

Fig. 11 shows the distributions of tag AUCs for the CNN model and the logistic regression baseline models. Because our dataset uses 4,508 tags, there are 4,508 AUC values that are binned and plotted for each model. The shape of the logistic regression distributions are similar - most of the tags fall within the central range of the models' distributions and there are few tags that perform relatively well or relatively poorly. Our convolutional architecture performs well on most of the tags, and instead has a long-tailed distribution of decreasing performance.

Table 2 displays a summarized, quantitative view of the tag AUC distributions. The logistic regression models have similar standard deviations, but the n-gram model has a considerably lower mean and median, indicating that the n-gram features are not as effective as the character embeddings. The convolutional network has a significantly higher mean and median, and a lower standard deviation. Although all of the models perform worse as the rarity of the tags increases, the lower standard deviation of the convolutional network implies that the model is more robust to the rarity of a given tag.

For source code validation, we use human feedback on the convolutional network to generate Fig. 12. The model obtained a 0.769 AUC. For the sake of comparison, we compute top-1 accuracy with the human validation on source code and obtain an 86.6% accuracy. We note that this is better than the analogous performance on Stack Overflow, which indicates that, on source code, the model performs better for the first tag, but worse for the rest.

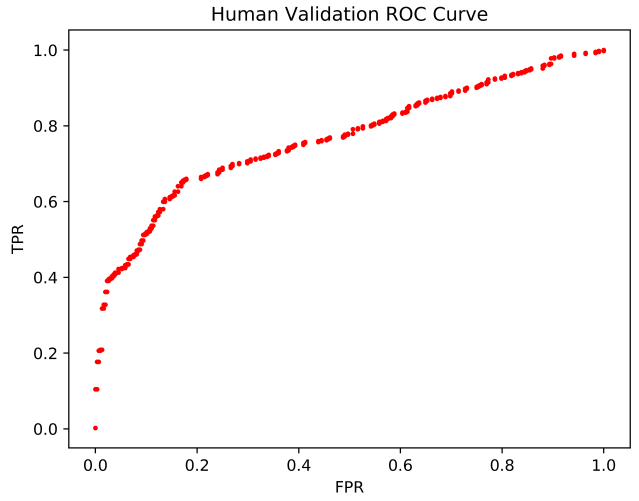

Figure 12: Human validation ROC curve with a 0.769 AUC. This differs from the Stack Overflow AUC values because it operates on the results of human validation, which is limited to only a few tags per document.

As a final note on performance, we trained and tested our model using an NVIDIA 1080 GPU. Our model obtains speeds of about 317,000 characters per second. Assuming an average of 38 characters per line of code (calculated based on a random sample of source files from GitHub[12]), the model is able to achieve prediction speeds of 8,342 source lines of code per second. To put this in context, it would take the model less than an hour to predict on the 20+ million lines of code in the Linux kernel. It is also readily parallelized to quickly predict across much larger source code corpora.

6 CHALLENGES/LIMITATIONS

In the course of our research, we encountered a few limitations that require further study. First is the transfer learning problem between Stack Overflow code snippets and source code. The lack of labeled source code prevents us from training directly on the desired domain.

The size of SO code snippets and the maximum number of tags per post are detrimental to the model's ability to predict on arbitrarily long source code. Due to the five tags per post limit, predicting more tags will increase the model loss, resulting in predictions with few tags. The original hypothesis was that the model would associate few predictions with short snippets and many tags for longer snippets, but the source code evaluation did not strongly support this. Exploring approaches that utilize loss functions other than binary cross-entropy may address these tag limit problems.

Another issue is that Stack Overflow users do not tag their code snippets directly, but rather their questions. For example, a user could post a code snippet of an XML document, ask how to parse it in Java, and tag the thread with "XML," "Java," and "parse." These tags are all extremely relevant to the user's question, but they do not describe the code snippet independently. During training, our model is only able to see that the XML document is an example of XML, Java, and parsing. This creates noise in the Java and parse labels.

Finally, the human verification process is a noisy evaluation of the model's performance on source code. Verifying the predictions is an arduous process because the model is familiar with thousands of functionalities. It is infeasible for individuals to be masters of such a wide range of ideas and tools, which results in a significant amount of labeler disagreement.

7 CONCLUSIONS/FUTURE WORK

We leverage the crowdsourced data from Stack Overflow to train a deep convolutional neural network that can attach meaningful, semantic labels to source code documents of arbitrary language. While most current code search approaches locate documents by matching strings from user queries, our approach enables us to identify documents based on functional content instead of the literal characters used in source code or documentation. A logical next step is to apply this model to large source code corpora and build a search interface to find source code of interest.

Unlike previous supervised SO tag-prediction models, we train and test strictly on code snippets, yet we still advance the top-1 prediction accuracy from 65% to 79% on Stack Overflow. We also achieve 87% on human-validated source code. Using the area under ROC to measure performance, we obtain a mean AUC of 0.957 on the Stack Overflow dataset and an AUC of 0.769 on the human source code validation. Refining the methodology and data preprocessing by training the model with entire threads instead of posts could alleviate the performance drop caused by transfer learning. An alternative direction for future research is to investigate better metrics and loss functions for training and evaluating model performance on long-tailed multilabel datasets. This could prevent the model from being punished for predicting more than five tags.

Finally, extensions of the architecture that broaden the contextual aperture of the convolutional layers may grant the model a deeper understanding of abstract code concepts and semantics. This would enable more sophisticated code search and comprehension.

ACKNOWLEDGMENTS

This project was sponsored by the Air Force Research Laboratory (AFRL) as part of the DARPA MUSE program.

REFERENCES

[1] Miltiadis Allamanis, Earl T Barr, Premkumar Devanbu, and Charles Sutton. 2017. A survey of machine learning for big code and naturalness. *arXiv preprint arXiv:1709.06182* (2017).
[2] John R Anderson, Daniel Bothell, Michael D Byrne, Scott Douglass, Christian Lebiere, and Yulin Qin. 2004. An integrated theory of the mind. *Psychological review* 111, 4 (2004), 1036.
[3] Sushil Bajracharya, Trung Ngo, Erik Linstead, Yimeng Dou, Paul Rigor, Pierre Baldi, and Cristina Lopes. 2006. Sourcerer: a search engine for open source code supporting structure-based search. In *Companion to the 21st ACM SIGPLAN symposium on Object-oriented programming systems, languages, and applications*. ACM, 681–682.
[4] Michael W Berry, Susan T Dumais, and Gavin W O'Brien. 1995. Using linear algebra for intelligent information retrieval. *SIAM review* 37, 4 (1995), 573–595.
[5] Black Duck. 2017. Open Hub. https://www.openhub.net
[6] Francisco Charte, Antonio J Rivera, María J del Jesus, and Francisco Herrera. 2015. Addressing imbalance in multilabel classification: Measures and random resampling algorithms. *Neurocomputing* 163 (2015), 3–16.

[7] Hoa Khanh Dam, Truyen Tran, and Trang Pham. 2016. A deep language model for software code. *arXiv preprint arXiv:1608.02715* (2016).

[8] Yann N Dauphin, Angela Fan, Michael Auli, and David Grangier. 2016. Language Modeling with Gated Convolutional Networks. *arXiv preprint arXiv:1612.08083* (2016).

[9] Andrea De Lucia, Massimiliano Di Penta, Rocco Oliveto, Annibale Panichella, and Sebastiano Panichella. 2014. Labeling source code with information retrieval methods: an empirical study. *Empirical Software Engineering* 19, 5 (2014), 1383–1420.

[10] Amit Deshpande and Dirk Riehle. 2008. The total growth of open source. In *IFIP International Conference on Open Source Systems*. Springer, 197–209.

[11] Malcom Gethers, Trevor Savage, Massimiliano Di Penta, Rocco Oliveto, Denys Poshyvanyk, and Andrea De Lucia. 2011. CodeTopics: which topic am I coding now?. In *Proceedings of the 33rd International Conference on Software Engineering*. ACM, 1034–1036.

[12] GitHub. 2017. GitHub. https://github.com

[13] A Grabowski, N Kruszewska, and RA Kosiński. 2008. Properties of on-line social systems. *The European Physical Journal B-Condensed Matter and Complex Systems* 66, 1 (2008), 107–113.

[14] Abram Hindle, Earl T Barr, Zhendong Su, Mark Gabel, and Premkumar Devanbu. 2012. On the naturalness of software. In *Software Engineering (ICSE), 2012 34th International Conference on*. IEEE, 837–847.

[15] Sepp Hochreiter and Jürgen Schmidhuber. 1997. Long short-term memory. *Neural computation* 9, 8 (1997), 1735–1780.

[16] Yoon Kim, Yacine Jernite, David Sontag, and Alexander M Rush. 2015. Character-aware neural language models. *arXiv preprint arXiv:1508.06615* (2015).

[17] Krugle. 2017. krugle. http://opensearch.krugle.org

[18] Adrian Kuhn, Stéphane Ducasse, and Tudor Gírba. 2007. Semantic clustering: Identifying topics in source code. *Information and Software Technology* 49, 3 (2007), 230–243.

[19] Darren Kuo. 2011. *On word prediction methods*. Technical Report. Technical report, EECS Department, University of California, Berkeley.

[20] Otávio Augusto Lazzarini Lemos, Sushil Krishna Bajracharya, and Joel Ossher. 2007. CodeGenie:: a tool for test-driven source code search. In *Companion to the 22nd ACM SIGPLAN conference on Object-oriented programming systems and applications companion*. ACM, 917–918.

[21] Bennet P Lientz, E. Burton Swanson, and Gail E Tompkins. 1978. Characteristics of application software maintenance. *Commun. ACM* 21, 6 (1978), 466–471.

[22] Yoelle S Maarek, Daniel M Berry, and Gail E Kaiser. 1991. An information retrieval approach for automatically constructing software libraries. *IEEE Transactions on software Engineering* 17, 8 (1991), 800–813.

[23] Jon D Mcauliffe and David M Blei. 2008. Supervised topic models. In *Advances in neural information processing systems*. 121–128.

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

[25] Stack Overflow. 2017. Stack Overflow. http://stackoverflow.com

[26] Santanu Paul and Atul Prakash. 1994. A framework for source code search using program patterns. *IEEE Transactions on Software Engineering* 20, 6 (1994), 463–475.

[27] Steven P Reiss. 2009. Semantics-based code search. In *Proceedings of the 31st International Conference on Software Engineering*. IEEE Computer Society, 243–253.

[28] Joshua Saxe, Rafael Turner, and Kristina Blokhin. 2014. CrowdSource: Automated inference of high level malware functionality from low-level symbols using a crowd trained machine learning model. In *Malicious and Unwanted Software: The Americas (MALWARE), 2014 9th International Conference on*. IEEE, 68–75.

[29] searchcode. 2017. searchcode. https://searchcode.com

[30] Konstantinos Sechidis, Grigorios Tsoumakas, and Ioannis Vlahavas. 2011. On the stratification of multi-label data. In *Joint European Conference on Machine Learning and Knowledge Discovery in Databases*. Springer, 145–158.

[31] SourceForge. 2017. SourceForge. https://sourceforge.net

[32] Sourcegraph. 2017. Sourcegraph. https://sourcegraph.com

[33] Clayton Stanley and Michael D Byrne. 2013. Predicting tags for stackoverflow posts. In *Proceedings of ICCM*, Vol. 2013.

[34] Martin Sundermeyer, Ralf Schlüter, and Hermann Ney. 2012. LSTM Neural Networks for Language Modeling.. In *Interspeech*. 194–197.

[35] Stephen W Thomas, Bram Adams, Ahmed E Hassan, and Dorothea Blostein. 2010. Validating the use of topic models for software evolution. In *Source Code Analysis and Manipulation (SCAM), 2010 10th IEEE Working Conference on*. IEEE, 55–64.

[36] Grigorios Tsoumakas, Ioannis Katakis, and Ioannis Vlahavas. 2009. Mining multi-label data. In *Data mining and knowledge discovery handbook*. Springer, 667–685.

[37] Xi-Zhu Wu and Zhi-Hua Zhou. 2016. A Unified View of Multi-Label Performance Measures. *arXiv preprint arXiv:1609.00288* (2016).