Software Bug Triage using Machine Learning and Natural Language Processing

Frank Russo, Ramya Raju, Carlo Clarke, Ning Yang, Anthony Escalona, Charles C. Tappert and Avery Leider
Seidenberg School of Computer Science and Information Systems, Pace University
Pleasantville, NY 10570, USA
Email: {fr50558n, rr35976p, cc86686n, ny00685p, ae50483p, ctappert, aleider}@pace.edu

***Abstract*—All software written even today has bugs or defects. It is critical to fix bugs as quickly and efficiently as possible. Triaging bugs can be time-consuming, especially on larger teams or with larger projects. The process of assigning bugs to the proper developer is a task that can easily be automated and is a good exercise for using Machine Learning concepts. We will be studying sample data from the Google Chromium project using Machine Learning to analyze bugs by title and description to determine the best owner of a defect. We will use Natural Language Processing to analyze the bug title and description, looking for patterns to determine the correct developers to assign to bugs.**

***Index Terms*—Machine Learning, Bug Triage, Natural Language Processing, word2vec, Long short-term memory, Recurrent Neural Network, Conditional Random Fields, Name Entity Recognizer, Master-Worker Algorithm, Python.#.**

I. Introduction

A. Bug Triage

Bug triage is an effort to locate bugs needing support, escalation or follow-up via a variety of bugs. A lot of different bugs exist and a slightly different method is needed for each form. Triage is a medical word meaning simply checking or monitoring [1] [2]. When you enter the emergency room, a nurse will carry out the first medical assessment. It is the job of the triage nurse to determine what the patient's issue is, what the priority of the issue is compared to other patients, in what order the patient's issue needs to be addressed, and who is ultimately the best doctor to handle the patient's issue [3]. This is basically what bug triage is about. Determining the type of bug, what is the priority and severity of the bug, and which developer is best suited to work on this type of bug. The role of triage nurse has typically fallen on a manager or team lead of a project, but Machine Learning (ML) is a good candidate to take this role. As long as the bug has good information, ML can decide precisely what will next take place. It can analyze the data to find out about symptoms or problems that have arisen and the review the situation compared to past data in the hopes of determining the right person to whom to assign the defect.

Several other attempts have been made at using ML concepts, specifically Natural Language Processing (NLP), to solve bug triage. The majority of the methods we will compare against have used the Naive Bayes method of NLP to triage bugs. We will compare that to using word2vec and Long Short-Term Memory (LSTM), which are also concepts of NLP.

The goal of this paper is to compare using word2vec with LSTM and Naive Bayes and show that using word2vec with LSTM will result in more accurately triaged defects compared to Naive Bayes.

II. Literature Review

A. Machine Learning & Natural Language Processing

Machine Learning (ML) is a set of statistical techniques for problem-solving. It involves applying a set of training data against models. The models are then used to predict future results against new data coming into a system.It is the awareness that computers are trained to learn and to do as humans do and that over time, they are trained autonomously through observations and real-world experience data and information. There are many types of ML, including unsupervised, semi-supervised, supervised, and reinforcement learning.

Machine learning algorithms may use named examples to predict future events by looking at past data and learning what to apply to the new data. These algorithms produce an inferred function that can predict output values based on the analysis of a known dataset that was used for training. [4] Once the data has been trained adequately, the program will supply goals for any new data. The learning algorithm may also compare its output against what is deemed as the correct, intended output and detect errors allowing for adjustments to the model.

Supervised algorithms for computer training are intended for example learning. The term 'supervised' learning comes from the notion that a instructor controls the whole process by training this kind of algorithm. The data used for training consists of inputs coupled with the right outputs when training a supervised learning algorithm. The algorithm tries to find patterns in the data that lead to the output we want during the workout. After training, a supervised learning algorithm takes on new inputs that are not seen and decides what mark the new inputs are labeled according to prior training results.

In order to distinguish the related data points, classification is used to group them in separate parts. Machine Learning is used for discovering the rules about how different data points can be separated. Regression is another style of this learning. The distinction between regression and classification is that regression generates a number rather than a class. Regression is also useful when predicting a variety of issues, such as stock prices, the temperature of the day, or the probability of an occurrence.

Unattended machine learning algorithms are useful when the training knowledge is neither marked nor numbered. Uncontrolled learning investigates how structures can extract a function from unlabeled knowledge to discover a hidden composition. The system is unable to find the right output, but examines the data and draw inferences from data sets to describe unlabeled compositions.

The most part of this type of learning is for clustering. Clustering is the process of forming separate classes. Clustering attempts to identify various subgroups in a dataset. As it is unattended learning, we are free to choose the number of clusters to build and do not restrict ourselves to any number of labels. It's both a curse and a blessing. In the empirical sample selection process, the selection of a sample with the correct number of clusters.

Semi-controlled machine learning algorithms lie somewhere between supervised and unsupervised learning, as they use both labeled and unlabeled training information – generally a much smaller number of data labeled and a larger number of undeclared information [4] The systems using this approach will increase learning precision considerably. In general, semi-monitored learning is preferred when the data obtained needs.

Reinforcing machine algorithms are methods of learning built to interact with the surrounding environment using actions and seeks error or rewards. The most significant characteristics of enhancement learning include testing, search for errors, and delayed reward. In order to optimize its productivity, this approach enables software agents and machines to programmatically identify the desired behavior under a specific context. [4] In order for the agent to better learn the behavior, clear input from the reward is required; this is referred to as the strengthening signal.

NLP is considered a discipline within Artificial Intelligence (AI). It deals with computer processing of large data sets containing natural human language. It is a way to analyze text based on a set of theories and a set of technologies. [5] By this it is meant that there are multiple techniques from which to choose to analyze language. The multiple techniques is similar to how humans use many types of language processing to understand language, though NLP is far from truly understanding language.

Typical NLP modules focus on the lower levels of language processing. There are many reasons for this. First, low levels may suffice to solve the problem at hand. Secondly, more effort has been made to research and implement models at the lower levels. Thirdly, the lower levels focus on smaller units of language analysis, e.g. words and sentences. These levels are typically rule-governed. Higher levels tend to be regularity-governed because they deal with world knowledge and texts. [5]

The nature of a software defect lends itself to NLP because the data is mostly large blocks of text that is ideal to processing through current NLP algorithms.

B. word2vec & Long Short-Term Memory

As mentioned earlier, the two methods we will focus on are word2vec and LSTM. Word2vec is a set of algorithms for unsupervised training of vectors of words on large blocks of text. The vectors that are the result of the algorithm show semantic relationships between their corresponding words. [6]

There are several types of models, for example, Neural Bag-of-Words (NBOW) model, Recurrent Neural Network (RNN), Recursive Neural Network (RecNN), and Convolutional Neural Network (CNN). These models take word embeddings in the text sequence as input and summarizes their meaning with a fixed length vector. [7]

Word2vec focuses on two models, RNN and CNN. These models take tokenized, non-processed text and build a feature vector for every type in the data set. [8] Although word2vec is not a deep neural network, it makes text numerical, and deep neural networks are capable of understanding it. We will be using RNN as it is able to process a text sequence of varying length by recursively running the text through a transition function to a state vector h_t. The activation of h_t at time-step t can be computed as a function f of the input symbol x_t and the state h_{t-1}. [7]

$$h_t = \begin{cases} 1 & t = 0 \\ f(h_{t-1}, x_t) & otherwise \end{cases} \tag{1}$$

Typically, a basic strategy is to take the input sequence and map it onto a fixed-sized vector passing it through one RNN, and then to take the resulting vector and feed it to a softmax function, or squashing function. A problem with RNN's is that these transition functions cause parts of the gradient vector to grow or decay while passing through the training model if it has long sequences. [7] Because of the exploding or disappearing gradients, the model has a difficult time learning long distance associations in a sequence.

Long Short Term Memory or LSTM, which is a type of RNN, was proposed [9] to address the disappearing gradients issue. LSTM by default has the capability of learning order dependencies in sequences allowing it to predict problems. They are designed to avoid long term dependency problem. It uses a hidden memory cell to update and expose the content when needed [10]. The LSTM layer is made up of a set of memory blocks that are connected recurrently, having a chain like structure. The memory blocks are similar to the memory in a digital computer. Each block has mulitple connected memory cells with an input gate, an output gate and a forget gate. These gates supply continous read, write, and reset operations for each of the cells.

C. Naive Bayes

Naive Bayes is a set of machine learning classifiers based on Bayes' theorem in probability and statistics. Bayes' theorem is defined as:

$$P(A|B) = \frac{P(B|A)P(A)}{P(B)} \tag{2}$$

Applying this theorem calculates the probability of **A** occurring because **B** has already happened.

Naive Bayes classifiers are typically used as a method for text categorization, which is a way of determining if a block of text belongs to one category or another. Usually multiple classifiers are applied to a dataset with each classifier having equal weight.

There are multiple types of Naive Bayes Classifiers. Multinomial is most widely used for document classification problem. This means checking if a document belongs to the certain category. This is done by checking the frequency the words are presented in the document and using that as a predictor. Bernoulli Naive Bayes is somewhat different than multinomial in that the predictors are boolean based, for example, does a word occur in the text or not. Gaussian Naive Bayes uses predictors that take up a continuous, non-discrete values and are assumed to come from a gaussian distribution.

Naive Bayes is a common solution for solving the bug triage problem [11] [12] [13] [14].

III. Methodology

Here we will talk about the code we have written to solve the bug triage problem. We will also compare it to an implementation of Naive Bayes.

A. Implementing Bug Triage

We first load two sets of data.

1) A triaged bug dataset is used for classifier training and testing by cross validation.
2) An untriaged bug dataset that learns from the target deep learning model in an unsupervised manner.

Next we set up the initialization of global variable, which divides into two parts. The parameters of the first part is for the word2vec function, as follows:

```
min_word_frequency_word2vec = 5
embed_size_word2vec = 200
context_window_word2vec = 5
```

The variable, *min_word_frequency_word2vec*, means a set of unique words that occurred for at least specific times is extracted as the vocabulary. The variable, *embed_size_word2vec*, means dimension of the embedding vector. The variable, *context_window_word2vec*, means how many words to consider left and right.

The parameters of the second part is to define the number of cross validations, the number of iteration times and the batch size of parallel computing, as follows:

```
numCV = 5
max_sentence_length = 50
min_sentence_length = 15
rankK = 10
batch_size = 32
myEpochs = 5
```

The variable, *numCV*, contains the number of cross validations. The variable, *myEpochs*, contains the number of iteration times for all the training examples. The variable, *batch_size*, contains the count of samples that will be passed through the network. A larger the batch size means you'll need more memory space. For example, say you have 1000 samples of training records and you set the batch_size to 100. The model will split the training set into blocks of 100 starting with the first 100 samples (from 1st to 100th) and use that block to train the network. Next, it takes the second block of 100 samples (from 101st to 200th) and uses that to train the network again. We keep looping through all the samples until they have been propagated through of the network. The variable, *rankK*, lets the user configure their own specific times for accuracy. The variable, *min_sentence_length*, is the minimum processing range of time steps in the LSTM model. The variable, *max_sentence_length*, is the maximum processing range of time steps in LSTM model.

Next there are *id*, *issue id*, *issue title*, *reported time*, *owner* and *description* options in each bug sample of the untriaged dataset. One case is shown in this json data:

```
{
  "id" : 1,
  "issue_id" : 2,
  "issue_title" :
    "Testing if chromium id works",
  "reported_time" :
    "2008-08-30 16:00:21",
  "owner" : "",
  "description" :
    "\nwhat steps will reproduce the
   problem\n1\n2\n3\n\r\nwhat is the
   expected output what do you see
   instead\n\r\n\r\nplease use labels and
    text to provide additional
   information\n \n"
}
```

It is necessary to preprocess them so that we can process the valid data easier. For the untriaged dataset, we only focus on the *issue_title* and *description*. We remove the return character "\r", the URLs of the online resource, the stack trace, the hex code and so on, and store these scattered words into an array: *all_data* as following:

```
[
testing, if, chromium, id, works, what,
   steps, will, reproduce, the, problem,
   is, expected, output, do you see
   instead, please, use, labels, and,
   text, to, provide, additional,
   information
]
```

We use the word2vec method to learn a bug representation using a Continuous Bag of Words model and then extract *vocabulary*.

```
wordvec_model =
  Word2Vec(
    all_data,
    min_count=min_word_frequency_word2vec
    ,
    size=embed_size_word2vec,
    window=context_window_word2vec
  )
vocabulary = wordvec_model.wv.vocab
```

We then must process the triaged dataset in a way very similar to the operation of processing the untriaged dataset. We store the title and description of every bug into an array, and the owner into another array as well for the following computing. The use-case of triaged dataset is shown below:

```
{
  "owner" : "amit@chromium.org",
  "issue_title" :
    "Scrolling with some scroll mice (
   touchpad, etc.) scrolls down but not
   up",
  "description" :
    "\nProduct Version      : <see about:
   version>\r\nURLs (if applicable) :0.2.
   149.27\r\nOther browsers tested:
   Firefox / IE\r\nAdd OK or FAIL after
   other browsers where you have tested
   this issue:\nSafari 3:\n    Firefox 3:
    OK\r\n         IE 7:OK\r\n\r\nWhat
   steps will reproduce the problem?\n1.
   Open any webpage on compaq 6715s
   running vista.\r\n2. Try scrolling
   with the touchpad\r\n3. Scrolling down
    will work , but up will not.\r\n\r\
   nWhat is the expected result?\nThe
   page to scroll up.\r\n\r\nWhat happens
    instead?\nThe page doesn't move.\r\n\
   r\nPlease provide any additional
   information below. Attach a screenshot
    if \r\npossible.\r\nOnly a minor bug
   .\n "
}
```

After preprocessing, the *all_data* and *all_owner* will be:

```
all_data = [
  scrolling, with, some, scroll, mice,
   touchpad, etc, scrolls, down, but, not
   , up, product, version, see about, if,
    applicable, other, browsers, tested,
   firefox, ie, add, ok, or, fail, after,
    other, browsers, where, you, have,
   tested, this, issue, safari, 3, 7,
   what, steps, will, reproduce, the,
   problem, open, any, webpage, on, cmpaq
   , 6715s, running, vista, 2, try,
   touchpad, down, but, is, expected,
   result, page, to, happens, instead,
   doesn't, provide, any, additional,
    information, below, attach, a,
    screenshot, possible, only, a,  minor,
     bug
]
all_owner = [amit@chromium.org ]
```

According to the result of triaged dataset preprocessing, we can get the arrays *all_data* and *all_owner*, and the triaged dataset is split into training data and testing data with a specific number of cross validations you provide to remove training bias. Let's say the number of cross validations is 3, the whole triaged bug dataset is divided into three parts, we can call them *sub1*, *sub2* and *sub3*. In the first iteration, we use *sub1* as training data, in the meantime, use *sub1* as testing data as well. In the second iteration, we use *sub1* + *sub2* as training data and use *sub2* as testing data. In the third iteration, we use *sub1* + *sub2* + *sub3* as training data and use *sub3* as testing data. The process above is a classic concept of cross validation.

```
train_data =
  all_data[:i * splitLength - 1]
test_data =
  all_data[(i-1) * splitLength:i *
    splitLength - 1]
train_owner =
  all_owner[:i * splitLength - 1]
test_owner =
  all_owner[(i-1) * splitLength:i *
    splitLength - 1]
```

For each bug of training data, we would remove words outside the vocabulary (generated by untriaged bug dataset) from training data, and also remove words outside the vocabulary from the owner array if the length of current train filter is larger than minimum sentence length. For each bug of testing data, we will make the same operations as each bug of training data.

The following block is for owner operations. We would remove invalid bugs from testing set that is not there in the training set, and also remove owners of testing set that is not there in the training set if the owner of testing bug doesn't exist in (*test_owner* – *train_owner*).

Now we need to create the training and testing data:

```
## Trainning Data
X_train =
  np.empty(shape=[len(updated_train_data)
    , max_sentence_length,
    embed_size_word2vec], dtype='float32')
Y_train =
  np.empty(shape=[len(updated_train_owner
    ), 1], dtype='int32')

## Testing Data
X_test =
  np.empty(shape=[len(updated_test_data),
     max_sentence_length,
    embed_size_word2vec], dtype='float32')
```

```
Y_test =
  np.empty(shape=[len(updated_test_owner)
   , 1], dtype='int32')
```

In this process, we create the empty storage space according to the specific parameters individually, for example, *embed_size_word2vec*, *max_sentence_length*, for training and testing data. The next step is to store trained and tested sentence or content of training and testing dataset.

Now that the data is ready, we start to construct the deep learning model. This model considers the word sequence both in forward direction and in back-ward direction so that a context of a particular word includes both the previous few words and following few words making the representation more robust. LSTM cells are used in the hidden layer which have a memory unit that can remember longer word sequences and can solve the vanishing gradient problem.

```
sequence =
  Input(shape=(max_sentence_length,
   embed_size_word2vec), dtype='float32')
forwards_1 =
  LSTM(1024)(sequence)
after_dp_forward_4 =
  Dropout(0.20)(forwards_1)
backwards_1 =
  LSTM(1024, go_backwards=True)(sequence)
after_dp_backward_4 =
  Dropout(0.20)(backwards_1)
merged =
  concatenate([after_dp_forward_4,
   after_dp_backward_4], axis=-1)
after_dp =
  Dropout(0.5)(merged)
output =
  Dense(len(unique_train_label),
   activation='softmax')(after_dp)
model =
  Model(inputs=sequence, outputs=output)
rms =
  RMSprop(lr=0.001, rho=0.9, epsilon=1e
   -08)
model.compile(
  loss='categorical_crossentropy',
  optimizer=rms, metrics=['accuracy']
)
```

B. Challenges

ML models are data intensive. They consume large sets of data making them perform slowly. The TensorFlow framework has been optimized using the Intel® Math Kernel Library for Deep Neural Networks (Intel® MKL-DNN) primitives, which allows it to get maximum performance. Because of this, the standard TensorFlow build does not run efficiently on all CPUs. For example, on one machine running an Intel® Core™ i7-6700HQ CPU @ 2.6GHz, the process takes a couple of hours at most. When trying to execute on another machine with an AMD A1-7300 Radeon R6, 10 Compute Cores 4C+6G @ 1.9GHz, the process ran for 18 hours and was just under half way done. This proved problematic as only one person was able to reliably execute the code.

One workaround was to try and run on Google Colab. We were able to run on Google Colaboratory with a smaller set of data within a couple of hours. When we tried to run on Colab with our full data set, there were errors reported that were only listed as warnings with the small data set.

IV. Results

First, we performed analysis on small training and test data sets as the time taken to run the data sets took hours and we were unable to verify and test repeatedly. The results seen with the smaller data sets though not as accurate gave a good picture on the learning model. Two methods have been used to perform the comparative analysis were LSTM and Naive Bayes. LSTM is a recurrent neural network proposed to control the gradient problem. Its ability to catch mundane correlations over long periods of time was exploited and trained in the series of vectors as explained. As seen in the output below, accuracy of LSTM is slightly better than the accuracy obtained by the Naive Bayes algorithm. To get a better understanding of the output we define epochs for the dataset. Epoch is a hyper parameter that is defined before training a model. An epoch is a single pass through the entire dataset with iterations. When the data set is large, it passed in batches. Having a huge data set, we divide it into six batches. One epoch is when an entire dataset is passed forward and backward through a network only once. For this dataset we observe six epochs. In the Naive Bayes method we observe the accuracy over the six iterations. The LSTM method gives a better accuracy over the trails when compared with Naive Bayes.

A. Output

On running the code we observe the results obtained by the LSTM method as seen in Table 1.

TABLE I: Values obtained on running LSTM

Iteration	Epoch Accuracy	Test Accuracy
1	Epoch 1/6: 0.0588 Epoch 2/6: 0.1143 Epoch 3/6: 0.1571 Epoch 4/6: 0.2047 Epoch 5/6: 0.2518 Epoch 6/6: 0.3044	0.784385
2	Epoch 1/6: 0.0609 Epoch 2/6: 0.1031 Epoch 3/6: 0.1279 Epoch 4/6: 0.1514 Epoch 5/6: 0.1750 Epoch 6/6: 0.1960	0.580757
3	Epoch 1/6: 0.0566 Epoch 2/6: 0.0883 Epoch 3/6: 0.1047 Epoch 4/6: 0.1198 Epoch 5/6: 0.1299 Epoch 6/6: 0.1370	0.662245
4	Epoch 1/6: 0.0547 Epoch 2/6: 0.0827 Epoch 3/6: 0.0934 Epoch 4/6: 0.1026 Epoch 5/6: 0.1060 Epoch 6/6: 0.1073	0.706721
5	Epoch 1/6: 0.0514 Epoch 2/6: 0.0766 Epoch 3/6: 0.0848 Epoch 4/6: 0.0896 Epoch 5/6: 0.0926 Epoch 6/6: 0.0920	0.756831
6	Epoch 1/6: 0.0525 Epoch 2/6: 0.0759 Epoch 3/6: 0.0833 Epoch 4/6: 0.0878 Epoch 5/6: 0.0890 Epoch 6/6: 0.0920	0.779603

B. *Competitor algorithms*

A common approach to solving bug triage is to use a Naive Bayes algorithm. We added an implemention of this using the scikit package of python. It is implemented as following:

```
classifierModel =
  MultinomialNB(alpha=0.01)
classifierModel =
  OneVsRestClassifier(classifierModel).
   fit(train_feats, updated_train_owner)
predict =
  classifierModel.predict_proba(
   test_feats)
classes =
  classifierModel.classes_

accuracy = []
sortedIndices = []
pred_classes = []


for ll in predict:
  sortedIndices.append(
    sorted(
      range(len(ll)),
      key=lambda ii: ll[ii],
      reverse=True
    )
  )

 for k in range(1, rankK + 1):
  id = 0
  trueNum = 0
  for sortedInd in sortedIndices:
    pred_classes.append(classes[sortedInd
   [:k]])
    if(id < len(Y_test)):
      if Y_test[id] in sortedInd[:k]:
        trueNum += 1
      id += 1
  accuracy.append(
    (float(trueNum) / len(predict))
    * 100
  )
print ('Naive Bayes accuracy:', accuracy)
```

Naive Bayes models are suitable for classification with discrete features (e.g., word counts for text classification). Integer feature counts are normally required by the multinomial distribution. Firstly, we create a classifier model for multinomial Naive Bayes. Secondly, we use OneVsRestClassifier strategy to fit underlying estimators. This strategy involves having one classifier in a class. For each classifier, the class is placed against the rest of the other classes. This approach is computational efficient because of only needing one class per classifier. It also has the advantage of being highly interpretable. It is possible to understand the class more easily by by inspecting its corresponding classifier since each class contains one classifier only. Third, we use bag-of-words model features, to make the probability estimates. Finally, we use the real data to get test accuracy.

Table 2 shows the accuracy generated using this method:

TABLE II: Naive Bayes Accuracy rates

Iteration	Accuracy
1	0.3100775
2	0.47913532
3	0.391366
4	0.398827244
5	0.42155314
6	0.4654118

V. Conclusion and Future Work

Both word2vec with LSTM and Naive Bayes are valid approaches to solving the problem of bug triage and will lead to improved times in triaging and ultimately in earlier bug fixes for important defects. However, the accuracy of the word2vec with LSTM method was better than the Naive Bayes method, as can be seen in Figure 1.

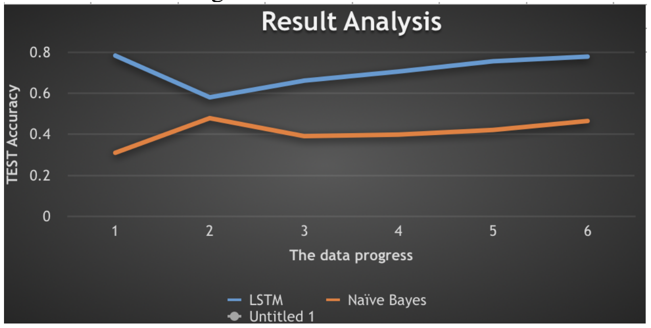

Fig. 1: Graph representing accuracy of word2vec with Long Short-Term Memory vs. Naive Bayes

We originally intended on seeing if the Master-Worker Algorithm would improve the models to successfully improve Triage times, but hardware limitations slowed us down. This would be a good topic for future work.

References

[1] S. Mani, A. Sankaran, and R. Aralikatte, "Deeptriage: Exploring the effectiveness of deep learning for bug triaging," in *Proceedings of the ACM India Joint International Conference on Data Science and Management of Data*, 2019, pp. 171–179.

[2] S.-R. Lee, M.-J. Heo, C.-G. Lee, M. Kim, and G. Jeong, "Applying deep learning based automatic bug triager to industrial projects," in *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, 2017, pp. 926–931.

[3] V. Akila, G. Zayaraz, and V. Govindasamy, "Effective bug triage–a framework," in *International Conference on Intelligent Computing, Communication & Convergence*, 2015, pp. 114–120.

[4] E. Alpaydin, *Introduction to machine learning*. MIT press, 2020.

[5] E. D. Liddy, "Natural language processing," 2001.

[6] E. Ordentlich, L. Yang, A. Feng, P. Cnudde, M. Grbovic, N. Djuric, V. Radosavljevic, and G. Owens, "Network-efficient distributed word2vec training system for large vocabularies," in *Proceedings of the 25th ACM International on Conference on Information and Knowledge Management*, 2016, pp. 1139–1148.

[7] P. Liu, X. Qiu, and X. Huang, "Recurrent neural network for text classification with multi-task learning," *arXiv preprint arXiv:1605.05101*, 2016.

[8] S. K. Sienčnik, "Adapting word2vec to named entity recognition," in *Proceedings of the 20th nordic conference of computational linguistics, nodalida 2015, may 11-13, 2015, vilnius, lithuania*, no. 109. Linköping University Electronic Press, 2015, pp. 239–243.

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

[10] H. Sak, A. W. Senior, and F. Beaufays, "Long short-term memory recurrent neural network architectures for large scale acoustic modeling," 2014.

[11] H. Hu, H. Zhang, J. Xuan, and W. Sun, "Effective bug triage based on historical bug-fix information," in *2014 IEEE 25th International Symposium on Software Reliability Engineering*. IEEE, 2014, pp. 122–132.

[12] G. Jeong, S. Kim, and T. Zimmermann, "Improving bug triage with bug tossing graphs," in *Proceedings of the 7th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, 2009, pp. 111–120.

[13] G. Murphy and D. Cubranic, "Automatic bug triage using text categorization," in *Proceedings of the Sixteenth International Conference on Software Engineering & Knowledge Engineering*. Citeseer, 2004.

[14] J. Xuan, H. Jiang, Z. Ren, J. Yan, and Z. Luo, "Automatic bug triage using semi-supervised text classification," *arXiv preprint arXiv:1704.04769*, 2017.