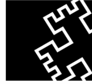


OULUN YLIOPISTO
UNIVERSITY of OULU

Improving Visualization on Code Repository Issues for Tasks Understanding

University of Oulu
Faculty of Information Technology and Electrical Engineering
Master's Thesis
Robroo Somkiadcharoen
16 June 2019

Abstract

Understanding the tasks and bug locating are extremely challenging and time-consuming. Achieving a new state of the art of understanding the tasks or issues and provide a high-level visualization to the users would be an incredible asset to both developers and research communities. Open Github archive are gathered, and the data is programmatically labelled. The Fasttext embedding model was trained to map the words to together based on semantics. Then, both CNN and RNN types of deep learning architectures are trained to classify whether each tokenized instance is a source file attribute or not. The word embedding and LSTM models worked well and did generalize in the real-world usage up to an extent. The models could achieve around 0.80 F1 scores on the test set. Along with the model, the generated usage graphs are presented that are the final output of the thesis work. Some types of issues were suitable for this workflow and did produce reasonable graphs which might be useful for the users to see the big picture of an issue.

Keywords
Task Visualization, Dependency Graphs, Deep Learning

Supervisor
D. Sc. (Tech), Professor Mika Mäntylä

Foreword

My sincere appreciation for the brilliant contributors of this thesis and supported both my mental and physical health during this journey.

I would like to thank my thesis supervisor Dr. Mika Mäntylä. He was always open to discussion whenever I had a question regarding the research even when he was on sick leave. He consistently steered me in the proper direction when things went south. My thanks to Dr. Maëlick Claes as a master’s thesis opponent. His advice were highly analytical and crucial to the research.

I am thankful to every single person who contributed to the European Masters Programme in Software Engineering (EMSE) program for scholarship opportunity of master’s degree. All colleagues at Softagram were awesome and always be supportive.

Finally, I must express my very profound gratitude to my parents and to Poy, my girlfriend for providing me with unfailing support and continuous encouragement throughout my years of study and through the process of researching and writing this thesis.

This accomplishment would not have been possible without them.

Thank you.

Robroo Somkiadcharoen

Oulu, June 16, 2019

Abbreviations

The following list is a collection of abbreviations that are commonly used in the publications and also appears many times in this thesis.

DNN	Neural Networks
RNN	Recurrent Neural Networks
NLP	Natural Language Processing
MSE	Mean squared error
FP	False Positive
TP	True Positive
FN	False Negative
TN	True Negative
F1	F1-score
LSTM	Long-Short Term Memory
Bi-LSTM	Bi-directional Long-Short Term Memory
CNN	Convolutional Neural Networks
ReLU	Rectified Linear Unit
XML	Extensible Markup Language
RQ	Research Question
DSR	Design Science Research
1D	1 Dimension

Contents

Abstract....2
Foreword....3
Abbreviations....4
Contents....5
1. Introduction....6
2. Background and related work....8
  2.1 Neural networks and deep learning....9
    2.1.1 Neural networks....9
    2.1.2 Training deep neural network....10
    2.1.3 Recurrent neural network....11
    2.1.4 Convolutional neural network....12
  2.2 Deep Learning in natural language processing....13
    2.2.1 Representation of raw text....13
    2.2.2 Many-to-Many RNN Applications....14
  2.3 Deep Learning in Tasks understanding....14
  2.4 Dependency Graph Generators....15
3. Research Settings....16
  3.1 Research Objectives....16
  3.2 Research Methods....16
    3.2.1 Design Science Research....17
  3.3 Research Design....17
4. Identifying source code attributes....19
  4.1 Methodology....19
    4.1.1 Data....19
    4.1.2 Models....21
    4.1.3 Training....23
  4.2 Result and Evaluation....23
5. Visualizing source code attributes....28
  5.1 Methodology....28
  5.2 Interpretation....28
6. Threats to validity....31
  6.1 Internal validity....31
  6.2 External validity....31
7. Conclusion and Future work....32
  7.1 Future Work....32
References....33
Appendix A. Training Graphs....41

1. Introduction

Issue in open source software development is used to refer to the issue ticket from issue tracking systems. The issue ticket can be seen as an artifact for reporting and requesting a bug or a feature to the software developer team. In addition, the issues usually contain the only textual descriptions of the environment and situation which seem to be sufficient for software developers to understand the objective of them.

Practically, even if a well-established database exists, understanding issues is not straightforward (Bettenburg et al., 2007). There are several challenges reported by developers who contribute in open-source projects about the complexity of understanding the source code and the issue objective (Guo, Zimmermann, Nagappan, & Murphy, 2011; Stol, Avgeriou, & Ali Babar, 2010). Even highly experienced engineers, who are not familiar with the job or the code base, might need hours to learn only to locate the usage chain of a function. Early research tried analysing the past git commit histories and numerically pointed out where the bug is (Zhou, Zhang, & Lo, 2012).

From an internal meeting in Softagram, the team notices that some issues type which provided the source file attributes should hint the developers about the dependency of the specific source files. In the focus of this research, source file attributes are the items that can be referred within the source code such as method name, class name, and file name. The example can be seen in “fileWriter.write()” as a method name, “FileWriter” as a class name, and “softagram-live/main.py” as a file name. Fortunately, Softagram, as a Finnish’s leading software analysis and visualization solution company, has the source code visualization engine available. The tool can show how the source files attributes are connected based on the usage of them (“Softagram Products – See Your Software Visualized,” n.d.).

Moreover, the integration of the Softagram’s source code visualization tool and the issue description would be beneficial to the software developers. That is, the developers will have the ability to understand the high-level dependency graph of the extracted source file attributes.

However, extracting those source file attributes is not a trivial task. It is possible that using a lot of regular expression patterns can achieve the same goal, but it is time-consuming and difficult to maintain. In addition, it is much more difficult to construct a temporal-based rule. For example, a rule such that after a keyword “This” followed by “file”, there is a 20% chances that the next word is a source file attribute is hard to construct.

Deep Neural Networks (DNN) keep showing futuristic results in many areas. In the area of computer vision, it can easily achieve more than 60 frames per second in the object recognition task and can even generate highly realistic image of a person which remains as a challenge for people to differentiate them from real ones (Horev, 2018; Karras, Laine, & Aila, 2018; Sandler, Howard, Zhu, Zhmoginov, & Chen, 2018). In the natural language processing task, OpenAI has developed a GPT-2, a DNN language model which can generate professional text that they chose to not release the pre-trained model due to safety and security reasons (Radford et al., 2019).

The Softagram team see this opportunity and believe that the deep learning algorithm can help to extract source file attributes as well. The benefit of using this DNN is that one does not need to manually create static rules for extracting the source file attributes. It can learn the features of the data automatically. As a startup company, introducing a killer feature would provide the company with an ability to stand out in a crowd. For example, one company applied artificial intelligence extensively for source code enhancement task like common bug detection (source{d}, n.d.). With experiences and interests in modern machine learning methodologies, researching machine learning ideas and see if it can eventually enhance the existing Softagram services and customers' experiences would be a great exploration.

The focus of the thesis is to answer the research questions (RQ):

RQ1 How to detect source code attributes automatically?

RQ2 How to build a visualization of an issue/bug description?

To find the answers, the execution is to create an automatic issue visualization. It is done by using deep learning to extract the source file attributes. Then, the results are fed to the Softagram's graph generator which creates a dependency graph between the source file attributes. The study is done by the design science research methods which focus on building an artifact to be used in the real-world settings.

This thesis will contribute and set up a practical state-of-the-art by applying natural language processing, and dependencies visualization to give a practical high-level task description to the users in Figure 1. The research community benefits from the research pipeline from the data source to the interpretation of the end result. Moreover, after making the product of the end result, this feature will be free to use for open source projects on Github.

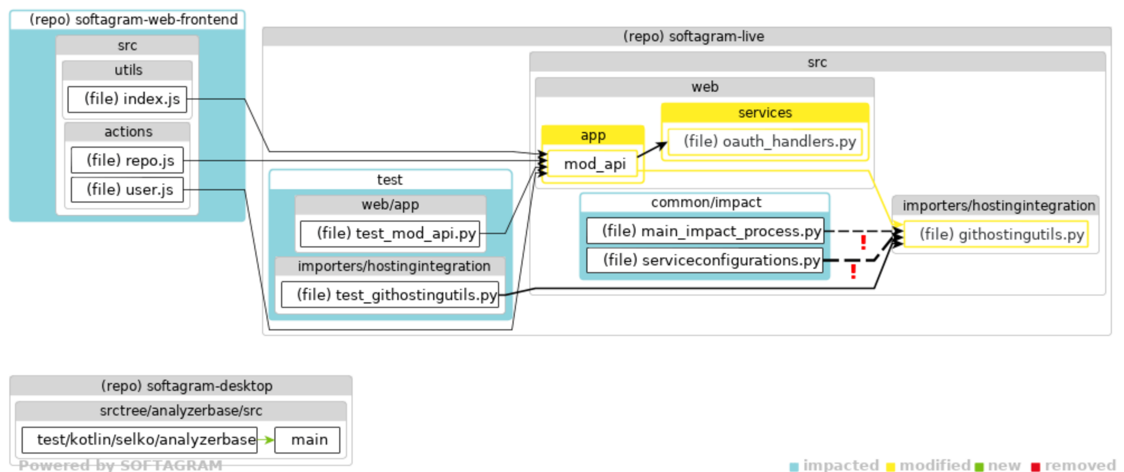


Figure 1. A change graph of Softagram pull requests review

The thesis has 7 chapters. Chapter 2 shows the background and existing work of both issues understanding the task and deep learning in natural language processing area. In Chapter 3, the research settings are presented which are research methods, and the step of evaluations. Chapter 4 shows how deep learning is used in keyword extractions, and Chapter 5 is about how to visualize the source code attributes. Threats to validity can be seen in Chapter 6. In the end, Chapter 7 presents the conclusion and future work.

2. Background and related work

In the last section, the brief background and motivation of the research are presented. This section is, based on previous literature, It presents a background of the issues understanding tasks, and a concise history of neural networks on the fields of natural language processing while providing some applications related to the practices. Finally, the current work on source code visualization and the Softagram's graph generator are reported as existing work.

Issue tracking systems are software that helps software developers store the bug or features reported by stakeholders at one place. It helps in reporting, assignment, progress tracking, and archiving of issues (Bissyandé et al., 2013). Giving the same understanding among stakeholders can be seen in one of the importance of issues reports (Bertram, Voida, Greenberg, & Walker, 2010). The issues usually have a description of an issue, an identifier, a description of the specific issue, and priority. This information leads to a meaningful discussion about the possibilities of solving if done with enough context (Bettenburg et al., 2007).

Traditionally, issues can be seen as a database of feature or bug reports, where it acts as a place to store documentation only. However, in modern development practices, especially in open source software development, one can see that the git hosting services with built-in issue tracking systems have been developed to serve as a communication channel (Bertram et al., 2010).

It is essential for the stakeholders of the software that have the same goal of each issue, since performing a refactoring when there is a mis-implementation could be expensive (Boehm, 2002).

Surprisingly, even if a well-established database exists, understanding issues is a non-trivial task (Bettenburg et al., 2007). Popular issues tracking systems nowadays, for instance, Github, Gitlab, Bitbucket, and Bugzilla, only represent issues with description in text format, tag, and links to others issues ("Issue trackers - Atlassian Documentation," n.d.; "Issues | GitLab," n.d.; "Mastering Issues · GitHub Guides," n.d.)

There are several challenges reported by students when they would like to identify the architectural pattern in open source software development. One example can be seen in "Hierarchy of source code directory organization is counter-intuitive, Manually browsing source code is tricky and time-consuming, and Code comments are not clear" (Stol et al., 2010). In the Mozilla project, researchers discover that in many cases if a bug introduces meaningful problems, there might be a need to assign the task to a person who knows better code structure. This seems to be the problem that the core developers who have a better understanding of the architecture have a better understanding than volunteers who just joined the development (Reis, Fortes, Pontin, & Fortes, 2002). Apart from the Mozilla project, some developers reassign the bug because the bug report quality was terrible. It is normally caused when the developers do not understand the task (Guo et al., 2011).

To solve the problems, there are some researchers come up with the solution which provides a machine learning solution which can tell if an issue is newbie friendly or not (Stanik, Montgomery, Martens, Fucci, & Maalej, 2018). Providing, also, class diagrams

or dependency views or any code visualization tool would help newcomers understand the problem in the bigger picture and be able to work on it and even find further problems (Park & Jensen, 2009).

2.1 Neural networks and deep learning

In supervised machine learning, the normal workflow is that the developers need to manually do feature engineering, which means trying to figure out how to get the important part of the data into the training or inference process of a machine learning model (Chandrashekar & Sahin, 2014). This task is expensive that only domain experts should get involved and seen as the most important part of a machine learning algorithm (Domingos, 2012; Ng, 2011). It simply means that, if the feature engineering is not done right, the statistical model might not learn enough useful data to inference (Parikh, 2014).

Deep learning, which is actually a neural network that has more than 3 layers deep, takes a totally different approach. One of the strongest points of neural networks is that it is a machine learning method that does feature engineering automatically (Nielsen, 2015). The shallower layer is a low-level feature extraction, where the input could be raw data, and it tries to get the low abstract representation of the data. The deeper layers now extract higher-level presentation of the data and eventually might get image objects or get too complex for a human to understand but still useful for the machine learning model (Karpathy, n.d.). Then, these representations can be used to work on various tasks. Since the useful features are already extracted by the deep neural network, there is no need to do the manual work of feature engineering.

2.1.1 Neural networks

The baseline of deep learning models are multilayer perceptrons or deep feed-forward neural network, where the objective of such networks is to make a great function approximator, i.e. how to approximate function F such that $y=f(x)$; where y is the optimization goal (Nielsen, 2015).

When it comes to networks, there can be many function approximators working together, $f(x)=f^{(3)}(f^{(2)}(f^{(1)}(f^{(0)}(x))))$, where each function serves as each own layers. In this example case, the f(0) serves as the first layer, $f(1)$ serves as the second layer, and $f(2)$ serves as the third layer and so on. These now created a depth of the learning model itself, where the word deep learning came from (Goodfellow, Bengio, & Courville, 2016).

In neural networks, the terminology for each function approximator is a neuron. A neuron is a function that receives one or more inputs and produces only one output. A function that computes the output of neurons in a vector form can be seen in the following (Nielsen, 2015).

$y_i=g(W_i x_{i-1}+b_i)$ Where x_{i-1} is an input vector or the output from the last layer, W_i is a weight vector, and b_i is a bias vector, g is an activation function, and i stands for layer i. The weights and biases are the parameters to be optimized during the training.

The activation function is the judge to decide whether this neuron should fire the signal to the deeper layer or not. The characteristics of useful activation functions are differentiable, cheap to compute, and serve non-linearity (Ramachandran, Zoph, & Le, 2017). Nonlinearity is important since combining multiple linear functions would result

in the linear function. That is, the network cannot learn more complex decision shape which is non-ideal in case of training the network on non-linearly separable data. The example of non-linear activation functions are softmax, tanh, and Rectified Linear Unit (ReLU) (Goodfellow et al., 2016; Nair & Hinton, 2010; Ozbulak, Neve, & Messem, 2018).

softmax

$$f(x_i)=\frac{e^{(y_i)}}{\sum_j 1+e^{(y_j)}}$$

tanh

$$f(x)=\frac{2}{1+e^{-2x}}-1$$

ReLU

$$f(x)=max(0,x)$$

2.1.2 Training deep neural network

Training a deep neural network is similar to what is done on a traditional machine learning model. It can also be trained with the gradient-based method (Lecun, Bottou, Bengio, & Haffner, 1998). It is done by adjusting each individual weight according to the error between prediction and ground-truth (Goodfellow et al., 2016). There are 4 main components which are loss functions, optimization, metrics, and transfer learning.

Loss functions get the differences between prediction and ground-truth in number. One example of loss functions can be seen in mean squared error loss (MSE) (Das, Jiang, & Rao, 2004).

$$MSE=\frac{1}{N}\sum(y_i-\hat{y}_i)^2$$

The main goal of training deep learning is to update weights and biases in the network to minimize the loss function which is an optimization problem. Normally, researchers tend to use backpropagation with stochastic gradient descent (Nielsen, 2015).

Gradient descent is an optimization algorithm which tries to find the local minima of a function by moving towards the downhill of a sloping point (Ruder, 2016). In addition, computing gradient of the whole dataset is computationally and memory expensive, so computing gradient only on a batch of a dataset is preferred in practice which is known as stochastic gradient descent (Bottou, 2010).

Backpropagation is a neural network training algorithm which uses the gradient descent to update the weights where it minimizes the loss and uses chain rules to propagate the error from the deep layers to shallow layers (Rojas, 1996). The main functions are to compute the error using loss function, then compute the gradients, i.e. rate of change, of output with respect to weights, then compute the weights with respect to weights in the previous layer. In the end, these gradients are used to update the weights (Goodfellow et al., 2016). During the training, epoch means one pass through the whole dataset. Since the data can grow larger than available memory, sampling a batch of data for training works in practice.

Metrics are how one evaluates the performance of the model which are often dependent on the tasks that are performing. The examples can be seen in F1 score, mean intersect over union, and accuracy (Rezatofighi et al., 2019; Sokolova & Lapalme, 2009). F1 score is based on the Precision (PREC) and Recall (REC). Precision is how well the model can correctly predict the attributes. Recall is how it correctly predicts the attributes on all samples. F1 score is the harmonic mean of precision and recall.

Mathematically, $PREC = \frac{TP}{TP+FP}; REC = \frac{TP}{TP+FN}; F1 = \frac{2 \times PREC \times REC}{PREC + REC}$ where TP is True positives, FP is False positives, and FN is False negatives (Davis & Goadrich, 2006).

TP and TN are the cases where a model predicts a correct class. TP means that the ground-truth of the data is a true class, and the prediction is also true class. TN means the actual class is a false class, and the predicted class is a false class. In contrast, False Positives (FP) and FN are the cases that a model predicts an incorrect class. FP means that the true condition is negative, but the predicted condition is positive. FN is the case where the true class is positive, but the prediction is negative.

The mean intersect over union is a metric that is used in the object detection and image segmentation task. It can be seen as the ratio of the overlapping area over the combined area (Rezatofighi et al., 2019). The accuracy metric is the correct predictions over the total predictions.

Training a deep neural network is time-consuming and resource exhaustive. Luckily, the learned weights of a network might be useful for tasks in the same area. In case of visual recognition, the animal classification network might learn the low-level features which can detect the basic geometry or edges of an object where the high-level features detect the more abstract details of the object (Karpathy, n.d.). In this case, a traffic sign classifier can share the same low-level features from the animal classifier as a pre-trained model. The only parts that need to update are the deeper layers of the network or only the high-level features for the specific domain only (Chunmian, Lin, Wenting, Kelvin, & Guo, 2019). This technique is called transfer-learning, and it helps reduce many numbers of parameters that need to be trained.

2.1.3 Recurrent neural network

Recurrent neural network (RNN) is created to deal with sequences of input which can generate one or many sequences of output (Karpathy, 2015). RNN is very useful to deal with sequences of data. As the design of memorization in mind, RNN neurons in a time step can remember the information from the previous input. Thus, it also captures how each input are related to each other and how they influence the next time step which is called as temporal features (Goodfellow et al., 2016).

The recurrent neural network system can be seen at applying the same weights to each input in every time step as in Figure 2. Where each time step t, the equation looks like the following.

$$a^{(t)} = b + Wh^{(t-1)} + Ux^{(t)}$$
$$h^{(t)} = \tanh(a^{(t)})$$
$$o^{(t)} = c + Vh^{(t)}$$
$$\hat{y} = softmax(o^{(t)})$$

Where b,c are bias vectors and W,U,V are the weight matrices.

H denotes the recurrent state, and o is an unnormalized log probability at time step t (Goodfellow et al., 2016).

One application of this Vanilla RNN can be seen in using a character level neural network to imitate Shakespeare (Karpathy, 2015). The network then learns how each character influences the next character based on past information about how characters arrange. Some successful applications of RNN-based architecture are image captioning and music composition (Choi, Fazekas, & Sandler, 2016; Wang, Yang, Bartz, & Meinel, 2016).

Normally, Vanilla RNN is not used in practice since it suffers from gradient vanishing which causes the network cannot learn long time dependencies among time steps. Therefore, there are modified RNN architectures which improve gradient flow. They are Long-Short Term Memory (LSTM) and Gated Recurrent Unit (Hochreiter, 1998; Kanai, Fujiwara, & Iwamura, 2017).

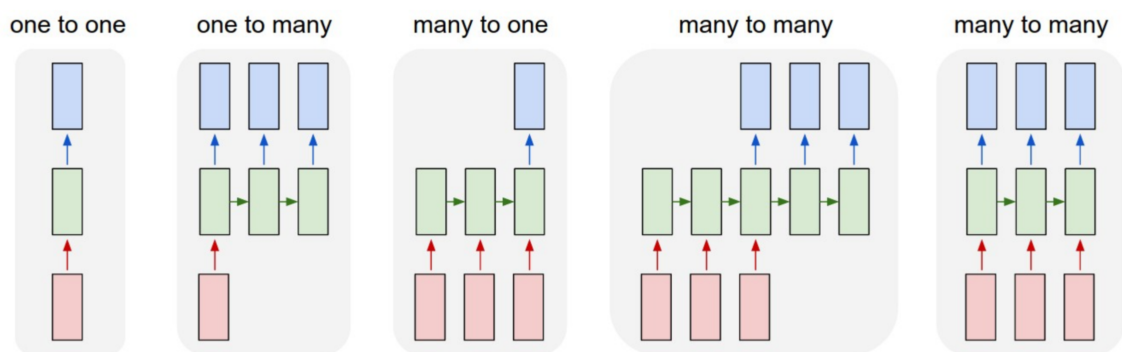


Figure 2. An example of the possibilities of RNN sequences input and output (Karpathy, 2015)

2.1.4 Convolutional neural network

Convolutional neural network (CNN) is a type of deep learning architecture which is originally created in the area of computer vision since it maintains shift, scale, and distortion variants when extracting features from the input (Lecun et al., 1998). However, the inputs can be one dimension (1D) data such as vectors, or 2 dimension data such as images. When introducing a convolutional layer into a deep neural network, this is called a convolutional neural network.

Apart from the images, CNN also works well in Natural Language Processing (NLP) area. Text representations of a sentence can be tokenized into words, where each word can be represented by a d-dimensional vector (Yin, Kann, Yu, & Schütze, 2017). Then, the convolution process in an n-gram fashion, i.e. a word with its closest n-neighbors. The speciality of this convolution is that weights are shared across the network. Then, it can learn to put more or fewer weights to the thing that the network learn that they are important. This intuition is comparable to the attention mechanism in RNN (Vaswani et al., 2017).

The basic computation formula for applying a convolution layer to a 1-dimensional vector is very straightforward as the following.

$y_i = g\left(W_i x_{i-1}[l : l + kernelsize] + b_i\right)$ Where x[i:j] here denotes an array slice of x from index i to index j. The slice represents the scope of input that will be convoluted which will be the same size as the convolutional kernel. l denotes the starting subset location of an input (Lee & Dernoncourt, 2016).

Based on the previous formula, it performs as a sliding window towards a vector. This also introduces the parameters sharing idea that the input will receive the same convolution kernel, W, in every timestep. This helps in reducing the parameters explosion problem in the traditional neural network (Goodfellow et al., 2016).

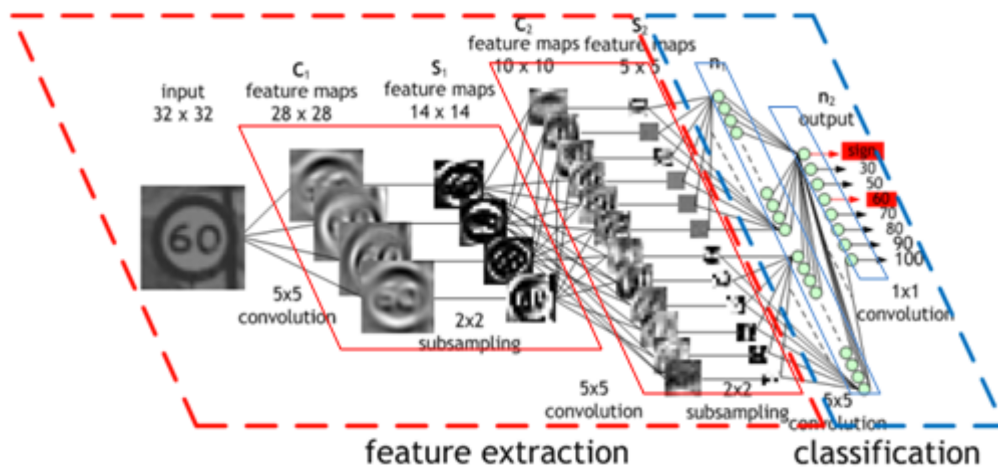


Figure 3. An image of traffic sign classifier using CNN from Maurice Peemen via (“Convolutional Neural Network (CNN),” 2018)

The example of the basic architecture of CNN can be seen in Figure 3. It has a few types of popular layers which are Convolution layer, Pooling layer, Activation layer, and Fully Connected layer. The convolution layer is basically a neural network layer that does the convolution operation on the input. In addition, the pooling layer is a subsampling layer which is meant to reduce the size of the input. The activation layer is the layer that performs the activation function on the input. Also, the fully connected layer is a normal neural network. The orders might be Convolution, Activation, and Pooling layers to extract features. Then, those are normally fed into Fully Connected layers to perform regression or classification tasks.

2.2 Deep Learning in natural language processing

NLP is a domain that aims to understand and process languages that are used by humans. There are already a lot of researches in this area. The NLP tasks that deep neural network provides successful results can be seen in part of speech tagging, sentiment classification, question answering, and text summarization (Huang, Xu, & Yu, 2015; Li, Jiang, Liu, Ren, & Li, 2018; A. W. Yu et al., 2018; L.-C. Yu, Wang, Lai, & Zhang, 2017).

Even though RNN tends to perform better in general NLP tasks, they all have roughly the same performance in sentiment classification, part of speech tagging, and answer selection tasks (Yin et al., 2017). In general, CNN benefits from recognizing the importance of the surroundings, and train much faster than RNN since the matrices and convolution operation can be parallelized (Bradbury, Merity, Xiong, & Socher, 2016).

2.2.1 Representation of raw text

For the data preprocessing of NLP tasks toward deep learning, researchers usually need to map each word of an input to a vector first before feeding them as an input to the model. The most traditional way is to use Bag of Words vector to represent the number of occurrences of each word in the input (Deepu, Pethuru, & S., n.d.). The problem with

this is that the input is not weighted. Therefore, some words that appear too many times in a context might be noise in data instead. This is how TF-iDF originated from. The idea is to weight the terms that appear a lot (Ramos & others, 2003).

In the neural word embeddings, Word2Vec is a model to represent raw text as a vector. It emphasizes on the relationship between words rather than the occurrences of them (Mikolov, Sutskever, Chen, Corrado, & Dean, 2013). Facebook's Fasttext model is also very popular and is an extension of word2vec. The main difference is that each word vector is composed of a character n-gram (Bojanowski, Grave, Joulin, & Mikolov, 2016). Therefore, the out of vocabulary words can be solved by constructing the word vector from its n-gram.

2.2.2 Many-to-Many RNN Applications

In the structure of deep learning model, Seq2Seq is a general-purpose neural network structure composing of encoder and decoder parts. Sequence to sequence network is a way to make recurrent neural network process on many-to-many input to output. It is useful in the sense that the network might want to read a sequence of input and output as a sequence. For example, translating a sentence from English to French is possible by employing the LSTM network (Sutskever, Vinyals, & Le, 2014).

In the keyword extractions tasks, the model is normally in a seq2seq fashion since the input needs to read in a sentence and output a few words afterwards. Also, this can be modelled as a classification task where the training data is flagged whether it is a keyword or not. Multiple variations of RNN show a promising result of extracting the key-phrases from tweets on Twitter (Zhang, Wang, Gong, & Huang, 2016). In addition, an encoder-decoder RNN model is able to generate key-phrases on many datasets, and can even generate the keywords that are not in the text (Meng et al., 2017).

2.3 Deep Learning in Tasks understanding.

There are many studies on adopting deep learning on the tasks understanding. Many of them actually tried to learn the representation of the issues itself. Then they get creative with a lot of applications that they could imagine. However, even they achieved outstanding performance, they are still far from perfect.

Anvik, Hiew, & Murphy (2006) show how they use machine learning methods to make an automatic system of assigning a task to a suitable developer. The study did train the model through repository histories and bug histories. Mani, Sankaran, & Aralikatte (2018) introduce a deep learning method with an attention mechanism to represent a bug report based on title and description. The representation is then fed into the classifier for assigning a bug to developers. Lyubinets, Boiko, & Nicholas (2018) invent hierarchical attention deep learning based model to automatically learn to label the bugs. The authors show that their work can outperform traditional machine learning solutions. Li et al. (2018) train unsupervised deep learning to summarize a bug report in the open source projects, where the architecture could provide a summary of 40% of ground-truth length where it covers more than 50% of the sentences in ground-truth summary. Dam, Tran, Grundy, & Ghose (2016) also created an abstract representation of the whole software life-cycle including issues and releases. They proposed to convert a text representation of issues description and source code into a vector representation using word2vec or paragraph2vec. Then, the representation can be feed into a recurrent deep neural network architecture and created an abstract representation of those. Huo, Li, & Zhou (2016) propose a CNN solution for inputting both source code and bugs description. Then, a representation is used to predict the location of the bugs in the source code. Choetkiertikul et al. (2018) show a case where they employed sequential-

based deep learning architecture with a highway network and word embedding to learning representation of issues. Then, they train a regressor to predict a story point of a specific issue. Stanik et al. (2018) created a classifier from a TF-iDF representation of issue tracker data. They successfully classify the data into three classes which are newbie friendly, experienced newbie friendly, and experts friendly. While the work is done by traditional machine learning, the importance of this work is how they pre-process the text features to create a representation that can be feed into deep learning based model too.

2.4 Dependency Graph Generators

Dependency graphs are the graphs that visualize the usage between source file attributes. In the case of method level, it shows whether the particular methods import which functions from which files. In addition, the outgoing edges of how the methods are imported to the neighbors are displayed too. The idea is to help developers understand what are the impacts that happen if a particular file or function is modified. The example can be seen Softagram's pull requests impact visualization graph in Figure 1.

Softagram has an architecture visualization platform ready to use. It can run on git repositories to extract dependencies among them, then the information on what files are changed, added, or removed can be fed to create a dependency graph. The application programming interfaces are written in both Java and Python which already exist and easy to integrate. The graph generation interface indexes the repositories as an Extensible Markup Language (XML) file. This XML file store relationships of all the classes and functions within the repositories. Moreover, Softagram also integrates with various git hosting services which are Github, Gitlab, Gitlab Enterprise, BitBucket, Bitbucket Enterprise, Visual Studio Team Services, and Gerrit. The dependency graph is computed automatically and serves right in the pull requests and cross-platform desktop application. The graph generator is based on the same graph engine which powered a popular graph visualization tool yEd.

Code Flower is an open-source MIT license source code visualization tool. It provides a fixed file level visualization for any project. However, the process of creating the input data for the Code Flower itself is still manual which requires a lot of work. In addition, there is not much interaction to do with the visualization (Zaninotto, n.d.).

Flow is a free source code visualization tool which supports Java language only. In contrast to Softagram and Code Flower, it does not focus on finding dependencies or architectural meaning of source code. The behavioral-driven flow visualization is the main goal of this program ("Visualize Java code execution," n.d.).

Sourcetrail is an offline cross-platform source code browser. It works by indexing a project, then interactively go through the source code via the diagram or the code. In addition, the dependency graphs can be created at any levels and it supports connection to an integrated development environment or code editors for digging through the source code ("Sourcetrail - The offline cross-platform code browser," n.d.).

3. Research Settings

This section will describe the research methods and research plans. It begins by showing the needs of executing design science research. Then, plans for integrating a deep learning model as a feature are presented.

3.1 Research Objectives

There are four main types of research knowledge contribution framework as shown in Figure 4 (Gregor & Hevner, 2013). They all could be seen in improvement, invention, routine design and exaptation. Firstly, improvement means developing new solutions to existing problems. Secondly, the invention is the invention of a new solution to new problems. Thirdly, using existing solutions to new problems are exaptation. Finally, the Routine design, which is not considered as a design science research (DSR) because there is no contribution done to the science society, is how one applied existing solutions to existing problems.

This thesis aims to contribute as an exaptation since the deep learning approaches to natural language understanding are widely adapted into different domains. However, according to the best of the team's knowledge and literature reviews, there is no application in natural language understanding with visualization on the issues understanding before.

The goal of this research is to answer the research questions:

RQ1 How to detect source code attributes automatically?

RQ2 How to build a visualization of an issue/bug description?

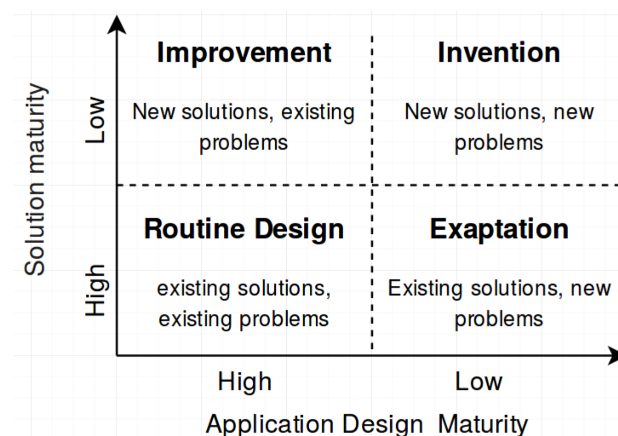


Figure 4. DSR Knowledge Contribution Framework (adapted from Gregor & Hevner, 2013)

3.2 Research Methods

In order to choose the suitable methods for particular research, looking at the goal that the research is trying to achieve is encouraged.

There are many research methods that can answer these **RQs**. The reasons that qualitative research and systematic literature reviews are not used in this study are that both methods can only give the idea, which is useful for research planning or obtaining knowledge.

On the other hand, design science research can produce the output as an information technology artifact and evaluate it under the real environment setting. The ultimate goal of this thesis is to implement the automated issue visualization as a feature to serve customers worldwide which is applicable to design science research.

3.2.1 Design Science Research

Design Science research is a research method that aims to understand and improve human performances (Aken, van, 2005). In addition, one can follow the template of this research to implement novelty methodologies into the real-world environment and evaluate the artifacts.

The design science research framework can be broken down into three cycles which are a relevant cycle, design cycle, and rigor cycle. As seen in Figure 5, the environment means the people, organizational and technical systems, and problems and opportunities. The design science research deals with the building of artifacts and evaluation. In the middle between design science research and environment, there is a relevant cycle. This cycle acts as a bridge between these two components. That is, it is a recurring process of evaluation of the design science artifacts based on environmental testing and requirements. Within the design science research itself, there is a design cycle between the building and evaluating the designed artifacts. On the rightmost component in the picture, there is a knowledge base. The knowledge base is scientific knowledge or expertise in the area. The rigor cycle acts between the design science research and knowledge base. Without a doubt, it means to use the foundations on the area that one already has to enhance the design science building and evaluation process (Hevner, 2007).

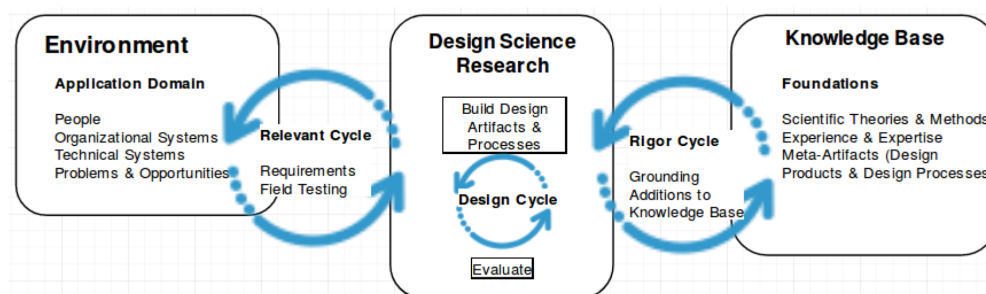


Figure 5. DSR Cycles (adapted from (Hevner, 2007))

3.3 Research Design

In order to answer the **RQs**, the methodology was building an automated issue visualization system which took the text description of issues as an input. The technology behind it was deep learning on natural language processing which would help with extracting source file attributes. The research team saw the opportunities and proposed an alternative to the regular expression solution. With this setting, the model should be able to recognize both the semantic meanings and temporal features of the task.

The rough picture that summarizes the method can be seen in Figure 6. Open source issues data from an existing database were gathered. Then, the data needed to be

explored and analyzed how one could pre-process it before inputting to the models. After that, deep learning architectures were implemented and trained on the preprocessed data. The training objective was to create a word classification model whether it is a source file attribute or not which the positive class belonged to source file attributes and negative class belonged to the non source file attributes.

During training, the models were evaluated based on metrics whether the model learned enough for the next stage. In the final part, the model was inferred on the unseen dataset, clean the predictions that are noise from the input, and fed it to Softagram graph generator. The purpose of Softagram graph generator was to create a dependencies graph on the source code attributes which can provide a high-level representation of the issues.

There were 2 levels of evaluation methods which were the model level and the practicality level. The **RQ1** was answered during the model level as a methodology to construct the artifact. The machine learning model performance evaluation was done by using the testing set where the network did not see those sets during learning. Metrics that used to check the performance were precision, recall, and F1 score. In other words, if the models achieved at least 0.80 F1 score, it meant that the models actually learned the training data and could generalize well in testing data which hinted that it might generalize in real-world data.

After the model stage, the outputs from the model were fed to the graph generator. The **RQ2** were answered during this stage. Real-world data from open source issues were fed to the whole pipeline for creating the visualization. The evaluation of the visualization was done by describing the meaning of the outcome of the sample to understand the value that it might have created.

To map the research plan to the DSR framework, raw data gathering and visualization stages of this research plan belong with the relevant cycle. Apart from this, model training and evaluation stages were as the design cycle and rigor cycle where the researcher of this research acted as a person in the environment.

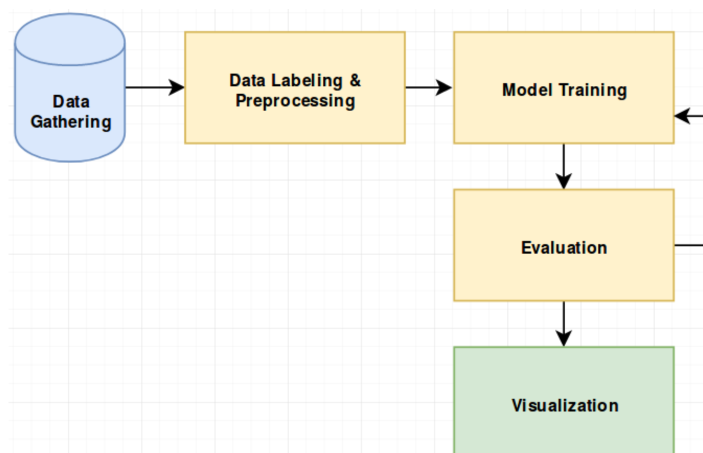


Figure 6. Research Plan

4. Identifying source code attributes

4.1 Methodology

This section is about the methodology of implementing the deep learning model to identify source code attributes. The chapter consists of data labeling and preprocessing, model architectures and training.

4.1.1 Data

The plan was to find open issues or bug trackers that were freely available on the Internet. However, popular datasets that researchers use such as Bugzilla in Eclipse ecosystems were professionally oriented that provided more meaningful description compared to the issues in non-professional and non-industrial Github projects. As the ultimate goal was to use the product as a Github issues extension, it was the best idea to use the data from the same distribution. In addition, acquiring the data from Github is simple.

There was an open dataset by Github. According to (“GH Archive,” n.d.), Github stored its hourly-updated data on Google’s BigQuery system and it was available to the public. Also, the issue events were all kept there as well.

Even if the data was free, the cost of using Google cloud platform was not. Luckily, they provided a coupon for the first user using Google Cloud Platform. The issues are then queried for the ones that opened during the year 2018 which the body must be at least 20 words and the title must be at least ten words. After that, the data was transfer to Google Cloud Storage for the backup purpose, then it was downloaded to a local machine for analysis.

The data consisted of the following columns, which were issue_url, issue_title, issue_body with a total number of around 800k Github Issues.

Labeling

Data labeling was the major problem of this task. Due to the nature of issues understanding, one might need a lot of time to understand the issue or not understand it at all. The scenario here was that the researchers believed that source file attributes in the issue would provide more context in addition to the textual description of an issue. For example, the issue in Figure 7 would make much more sense to come up with a dependency graph of the skimage/morphology/_skeletonize_cy.pyx and method _skeletonize_loop.

In the first setting, a researcher tried to label the data by himself and discovered that he needed more than 2 minutes per issue to only get a rough context. In order to label the whole issues, it would take at least 1,600,000 minutes or roughly three years straight to obtain the result. Even though in this experiment only around 12,000 issues were used, it would take roughly 300 hours to accomplish which was not ideal. Therefore, The core of this research was to try out if the work was doable, then keep enhancing a specific part for the production use. Therefore, using a regular expression as rule-based labeling

was taken in place. The words that were associated with source file attributes were annotated as class 1 and 0 otherwise.

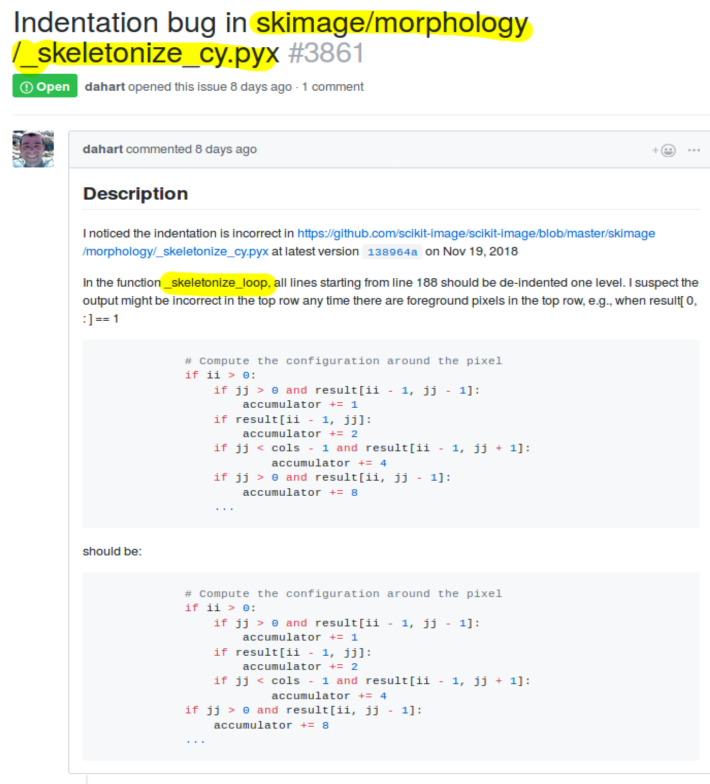


Figure 7. An example issue from scikit-image repository onhttps://github.com/scikit-image/scikit-image/issues/3861. The yellow highlighted has the true class as source file attributes.

Pre-processing

In order to select the issues that were descriptive enough to provide source file attributes, the data were first preprocessed by concatenating the issue title and body. Then, regular expression was employed to detect if there were camel case, small camel case, snake case, file path and methods-like strings in the concatenated title and body. After filtering, the dataset ended up having 360k issues left.

Since there was no access to high processing power available on the task, the dataset was then randomly sampling for only 12k of issues. Initially, stratified random sampling was a reasonable practice since it balanced the data, but the nature of this data was that there might be only some small parts of the issue that are source file attributes, and trying to sample the issues that consist of a lot of true classes might lead to bias in the data for this case eventually. The number of true class and false class in the dataset are 1,396,583 and 66,097 respectively.

Then, the special characters and excess white spaces were cleaned and tokenized base on a white space character. During the manual data labeling for around 60 issues, some heuristics were gathered. Issues could contain many source file attributes in the description, however, only some of them were worth exploring the dependency graph. This source file attributes were called main source file attributes. The meaningful source file attributes of an issue were within the first 100 words more than 50% of the time. In addition, the mean and median of the description lengths were 107.49 and 93.0 words

respectively as in Figure 8. The length of the description was truncated to only 120 words for the purpose of computation efficiency while still preserve the main source file attributes from the heuristic. In addition, this limitation also made it easier to compare the performance between different types of models.

In order to map the word into an n-dimensional space, words embedding were used. However, the pre-trained word embedding based on Wikipedia or some other domains would not work on this case since the source file attributes unlikely to happen in the whole corpus. Fasttext was used for the main reason that it captured the semantic meaning of the word and could handle out of vocab which was lacking in word2vec (Bojanowski et al., 2016). In contrary to the training of source file extraction model, the Fasttext model was trained on the rest 800k cleaned Github issues description.

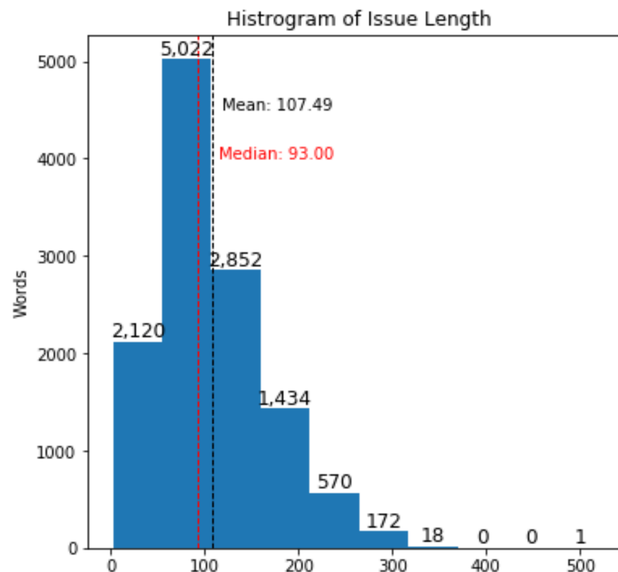


Figure 8. Histogram of Issue Length

4.1.2 Models

The main reason that TensorFlow framework was chosen for this work was that it supports the high-performance production ready serving of the model seamlessly (Olston et al., 2017). This property would help developers deploy their deep learning model to the production easily without being an expert in machine learning model deployment.

Four sequence classification models are constructed. All models here were classic models where the CNN was based on Lenet style and LSTM was from a sequence classification style. However, the 1D convolution model was developed from the intuition of convolution operation itself.

For this task, four types of architecture were chosen. The model 1 was Lenet style CNN model. It was just convolution following by max pooling for three times, then fed the extracted features to the fully connected layer. The input was constructed as a 2D matrix, where each row corresponded to each word in an issue, and the columns were 100-dimensional vectors of a Fasttext embedding of a word. In the first three layers, they were pairs of 32x3x3 convolution layer following by 2x2 max pooling layer. In the end, they were then fed to the two fully connected layers with the neuron size of 300 and 120 respectively. The activations for the intermediate layers were done by ReLU, while the last activation was softmax. For high-level understanding details of the CNN model, Figure 9 shows the visualization and details of it. In the same procedure, the 2nd

model was also a Lenet style CNN model. However, it made use of a 1D convolution along the horizontal axis instead under the intuition that it should extract the features of the word and its neighbors. This model could be seen as applying n-gram word embedding vector to a CNN. The only difference in model 1 and 2 was only the size of convolution filters. The filter sizes in model 2 were 5x100, 5x 50, and 5x25. This should be similar to applying n-grams convolution of an input where n=5. Figure 10 captured the high-level understanding of how the 1D convolution operated.

The 3rd and 4th models were solely based on the sequential model. Both models were two layers stacked LSTM and two layers stacked Bi-Directional Long-Short Term Memory (Bi-LSTM) which directly used the output from Fasttext embedding as an input. The size of the hidden nodes for each LSTM node was 200. The total model workflow can be seen in Figure 11 and Figure 12.

The idea of the comparison of these models was that they both had the ability to extract features from the input automatically. However, CNN tended to extract the feature from the input space, or in n-grams fashion in 1D convolution of this case. In contrast, RNN could recognize temporal features. RNN based model could have answered the hypothesis that the dataset might not need to be perfect since the model should learn semantic features from the Fasttext embedding and temporal features from the construction of a time-series based issues. In other words, it should be able to learn the meaning of the word based on the context. Also, where and when do the source file attributes happened in the context.

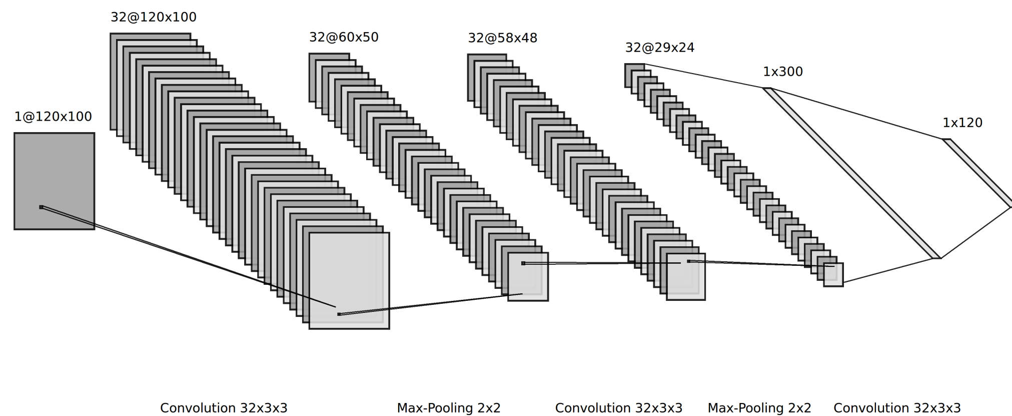


Figure 9. CNN architecture of Model 1

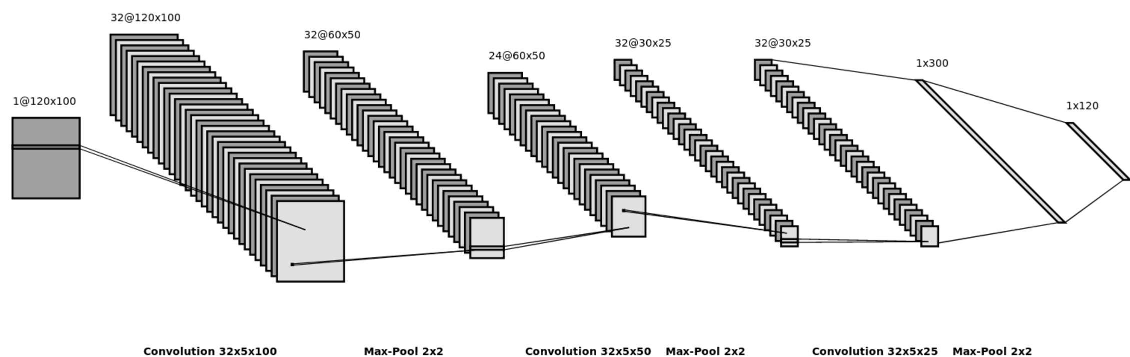


Figure 10. CNN architecture with 1D convolution of Model 2

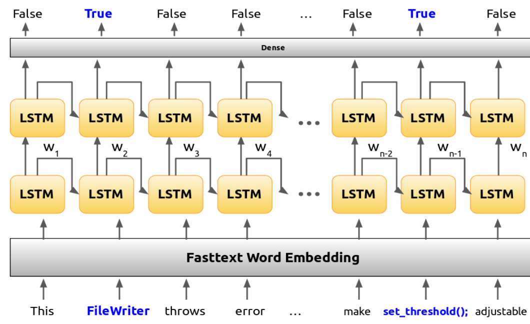


Figure 11. LSTM architecture of Model 3

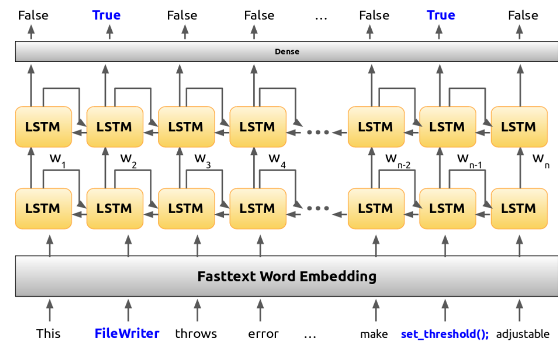


Figure 12. Bi-LSTM architecture of Model 4

4.1.3 Training

A training set and a test set were split based on 65%/35% from the total issue.

They were trained with Adaptive Moment Estimation optimizer since it supported adaptive learning rate with the starting value of 0.001. There were two settings for the batch size. A batch size of 128 was used for LSTM, Bi-LSTM, and CNN models. In addition, the batch size of 32 was chosen for Bi-LSTM. The reason that the upper limit was 128 was because of a memory constraint. In addition, using a larger batch size allowed the model to train much faster, but it also consumed more memory too.

There was no over-fitting prevention done on the models, since it made more sense to see the training & validation loss first whether there was a chance of over-fitting. The loss function was binary cross entropy. Since the hyper-parameter searching was quite timing expensive, the researchers could not explore the search. The only settings that were done in this research were from the gut's feeling.

The training was solely done on Google's colab because it provided free-of-charge access to a GPU ("Colaboratory – Google," n.d.). It was configured to train 60 epochs with early stopping in 10 epochs threshold which monitoring on test set f1-score. The total time per each epoch in LSTM was around 3 minutes per epoch, and Bi-LSTM was around 10 minutes per epoch. For CNN, an architecture which could parallelize the training, the timing was also around 3 minutes per epoch for the current architectures and filter sizes. The monitoring of the training was done seamlessly via Tensorboard via ngrok directly from Google's colab notebook. In order to store the trained weights and model, PyDrive library was used to upload & download models from private Google Drive account (*Google Drive API Python wrapper library.*, 2013/2019).

4.2 Result and Evaluation

	CNN	1D CNN	LSTM	Bi-LSTM	Bi-LSTM (batch_size=32)
Test F1	0.3085	0.3551	**0.8156**	0.8019	0.8049
Test Precision	0.4205	0.5339	**0.8275**	0.8229	0.8091
Test Recall	0.2470	0.2676	**0.8046**	0.7827	0.8032

Table 1. Scores on the test set.

The detailed results of the experiment were reported in both Table 1 and Appendix A. Training Graphs. The LSTM model converged around the 30 epochs with the f1 score on the validation at 0.81. Meanwhile, the Bi-LSTM model converged in only around ten epochs with the validation f1 score as ~0.80. It can be seen that the Bi-LSTM converged much faster than the uni-directional LSTM. In contrast, the time spending to train Bi-LSTM was two times longer than the time needed to spend on the LSTM. In the same ways, there was no significant improvement from adding another direction to the model. Even using the smaller batch size which was supposed to give a better convergence, the smaller batch size of 32 does not make a noticeable significant in F1, Precision, and Recall (Keskar, Mudigere, Nocedal, Smelyanskiy, & Tang, 2016).

In contrast, the CNN models could not learn useful features for prediction. The validation loss stopped reducing since the 5th epoch in model 1. It could only reach around 0.30 for F1 score. Similarly, the 1D CNN model could only learn to calculate F1 score on a test set as 0.3551 in epoch 26. The reason that CNN cannot learn useful features was that the features of the neighborhood might not be useful enough for the network learning style. In addition, temporal features might be crucial for this task. In order to try out a CNN again, maybe just using a CNN on a single word vector might work well. For more information about test loss, PREC, REC, F1 on each model, please see Appendix A. Training Graphs.

The following evaluation was done on the best model which was unidirectional LSTM. The result tables here showed that the RNN based models did not suffer from over-fitting, so the usage of regularization was not needed in this case. The possibilities were that it was because the data was automatically labeled, thus there was much noise which might prevent the model from starting to memorize the data.

In order to describe the performance of the model, it could be done easily through the confusion matrix. It was a matrix which showed the true class and predicted class on a data set which could be translated to the performance of the model. In this case, the predicted results from the test set were shown.

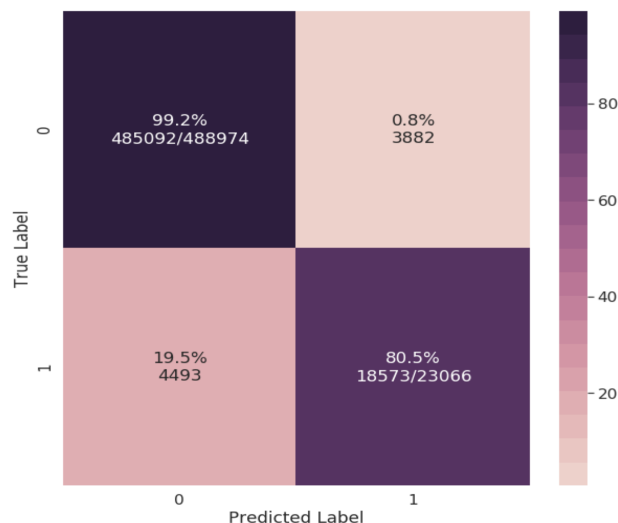


Figure 13. a confusion matrix for LSTM model on the test dataset.

According to the confusion matrix in Figure 13, the majority of the classes were predicted correctly, in other words, the 1st and 4th quadrants of it produced quite high values as 99.2% and 80.5% for TP and TN respectively. The numbers 488,974 and 23,066 denoted the classes of each word instance from the test set. In contrast, the 2nd quadrant was very low as only 0.8% was false positive (FP), and the 3rd quadrant was

19.5% for FN, which was not low enough and produced quite a lot of noises in the predictions. In these cases, TP and TN were quite high and produce reasonable in most cases below.

The example of the 1st, 2nd, and 4th quadrants could be seen in Figure 14. The texts that were highlighted in yellow are TP, while the blue-highlighted indicated that the specific texts were the false positives. The non-highlighted denoted the TN class. Even the confusion matrix showed FP rate was very low, there were some examples of FP in this case. Using the heuristics, the reason that “ii”, “jj”, and “result[:” were predicted as source file attributes was because of the limits of using regular expressions to generate ground-truth. In order to reduce the FP, having a better data labelling method would help.

In the case of the third quadrant, a real-world example was in Figure 16. It followed the color code like the above, except that the ones highlighted in green are FN. This case happened as a failure of a dataset itself due to the constraints from using regular expressions. The regular expression rules must have computed noises to the class labels. In addition, not reporting the “ProgramData” and “dtype_range” could lead to better predictions of the model.

There was also one interesting prediction that can be found in Figure 17. Even though the name of the method in the issue was not labeled in the dataset because of a bug in the patterns used for matching, the network learned to predict this eventually. It was a hint that the model could learn from temporal features and did not over-fitted the dataset. That is, the model was able to learn both the semantic and temporal features. However, there were arguably many parts which could have predicted as source file attributes but were reported as true negatives due to the limit of regular expressions.

There were various types of issues, most of them did not provide any results as they might not be applicable to this task. In other words, there was no clean source file attributes defined in the issue, it usually did not provide useful features for the models and thus lead to no result from the predictions. For example, the feature request or bugs that were caused by the environment usually do not have source file attributes to extract. Therefore, the graph generator would have no input to visualize. In Figure 15, it showed the example of the issue that do not contain source file attributes and no FP and FN presented in the predictions.

Indentation bug in skimage/morphology/_skeletonize_cy.pyx #3861

Open dahart opened this issue 19 days ago · 1 comment

dahart commented 19 days ago

Description

I noticed the indentation is incorrect in https://github.com/scikit-image/scikit-image/blob/master/skimage/morphology/_skeletonize_cy.pyx at latest version 138964a on Nov 19, 2018

In the function _skeletonize_loop, all lines starting from line 188 should be de-indented one level. I suspect the output might be incorrect in the top row any time there are foreground pixels in the top row, e.g., when result[0, :] == 1

```
            # Compute the configuration around the pixel
            if ii > 0:
                if jj > 0 and result[ii - 1, jj - 1]:
                    accumulator += 1
                if result[ii - 1, jj]:
                    accumulator += 2
                if jj < cols - 1 and result[ii - 1, jj + 1]:
                        accumulator += 4
                if jj > 0 and result[ii, jj - 1]:
                    accumulator += 8
                ...
```

should be:

```
            # Compute the configuration around the pixel
            if ii > 0:
```

Figure 14. Example of predictions 1 From https://github.com/scikit-image/scikit-image/issues/3861

Builds broken with QT 5.12 #3739

Closed hmaarrfk opened this issue on Feb 10 · 22 comments

hmaarrfk commented on Feb 10 Member

Description

Qt 5.12 was released today, and likely broke matplotlib.
https://travis-ci.org/scikit-image/scikit-image/jobs/491337465#L1931

https://travis-ci.org/scikit-image/scikit-image/jobs/491337465#L2058

Windows seems ok???

Way to reproduce

See travis, linux 5.12

Figure 15. Example of an issue From https://github.com/scikit-image/scikit-image/issues/3739

AttributeError: 'Image' object has no attribute 'dtype' while using canny edge detector #3700

Closed talhaanwarch opened this issue on Feb 1 · 2 comments

talhaanwarch commented on Feb 1

I am trying to apply canny edge detector on an image, but i am getting an error AttributeError: 'Image' object has no attribute 'dtype' while using canny edge detector.

Though prewitt and sobel filter works well with no error

```
from PIL import Image
from matplotlib import pyplot as plt
from skimage import feature

img=Image.open('ribs.jpg').convert('L')
plt.imshow(img)
edge=feature.canny(img, sigma=1)
```

Here is error i am getting
edge=feature.canny(img,0.001)
Traceback (most recent call last):

File "", line 1, in
edge=feature.canny(img,0.001)

File "C:\ProgramData\Anaconda3\lib\site-packages\skimage\feature_canny.py", line 159, in canny
low_threshold = 0.1 * dtype_limits(image, clip_negative=False)[1]

File "C:\ProgramData\Anaconda3\lib\site-packages\skimage\util\dtype.py", line 49, in dtype_limits
imin, imax = dtype_range[image.dtype.type]

AttributeError: 'Image' object has no attribute 'dtype'

Can any one help me please

Assign No one · Labels None y · Project None y · Milesto No mile · Notific You're from thi · 3 parti

Figure 16. Example of predictions 2 From https://github.com/scikit-image/scikit-image/issues/3700

skimage.transform.resize: anti_aliasing_sigma default value #3890

Open etiennelndr opened this issue 6 days ago · 1 comment

etiennelndr commented 6 days ago

Description

I was reading the documentation of the skimage.transform.resize function when I found that, when `anti_aliasing_sigma` is set to `None`, this parameter should have the default value "(1 - s) / 2". But, line 107 this parameter is set to `np.maximum(0, (factors - 1) / 2)`. Shouldn't it be `np.maximum(0, (1 - factors) / 2)` ?

Version information

```
from __future__ import print_function
import sys; print(sys.version)
import platform; print(platform.platform())
import skimage; print("scikit-image version: {}".format(skimage.__version__))
import numpy; print("numpy version: {}".format(numpy.__version__))
```

```
3.6.6 (v3.6.6:4cf1f54eb7, Jun 27 2018, 03:37:03) [MSC v.1900 64 bit (AMD64)]
Windows-7-6.1.7601-SP1
scikit-image version: 0.14.2
numpy version: 1.14.5
```

This error is also in the current version (see here).

Figure 17. Example of predictions 3 https://github.com/scikit-image/scikit-image/issues/3890

5. Visualizing source code attributes

5.1 Methodology

The graph generation was done by an executable file from Softagram. The graph generator engine was written in a way that the users needed to input the query as a set format. In addition, there are rules for constructing the query for the engine. The pathnames that had at least one slash('/') character would be treated as a full path. All the special characters must be neglected, except underscore and dot, since these two provide useful information in the query. The underscore could denote the variable name or function name in programming languages that use snake case practice. The dot also acted as an attribution-ship in some programming languages. However, the dot was not supported as a query syntax yet, so there were some special treatments that needed to be done for this type of attributes as shown in the example in the following example. In addition, the source file attributes that looked like outliers and false alarm such as 'print' and 'result[' needed to be neglected.

To make it more clear for the reader, the example can be seen in how to construct the query of Figure 17, the predictions from the models were ["skimage.transform.resize', "anti_aliasing_sigma", "print_function", "sys.version", "platform.platform()", "format(skimage.__version__)", "format(numpy.__version__)"]. The output was then filtered by neglecting the format and print which are the built-in Python 3 functions. As a result, the potential query elements were ["skimage.transform.resize', "anti_aliasing_sigma", "sys.version", "platform.platform()"]. The elements with dot needed to be constructed as an AND of sets, so that the resulted queries were filtered accordingly. In addition, the normal terms needed to be constructed as an OR of sets, so that these terms were included in the final results. The final query of this case was "(skimage AND transform AND resize) OR aliasing OR (sys AND version) OR (platform AND platform)". The final image can be seen in Figure 19.

5.2 Interpretation

Figure 19, Figure 20, and Figure 21 are the outbound dependency graphs of the issues in Figure 17, Figure 14, and Figure 18 respectively. Due to the nature of scientific packages, some implementations are done in Cython and Softagram's indexing analyzer understood those Cython files as an external.

In these cases, the files that were the core files of each issue appeared on the rightmost column of the graph which were skimage.transform.resize, _skeletonize_cy._skeletonize_loop, and points_in_poly.

By showing the usage graph for the aforementioned functions, the graphs captured the total usage of the core files in the repositories. Those usages could be grouped as test files and submodules. It could help developers to locate the other modules that will be affected by implementing a function or fixing a bug in an issue. This type of usage definition could be done in the same manner by browsing source code through some Python integrated development environment like PyCharm. However, it was difficult to understand the whole pictures when developers needed to dig deep into some submodules. Moreover, the source code relationship showed in a textual format, which

might need a longer time to understand the big picture compared to the high-level source code visualizations.

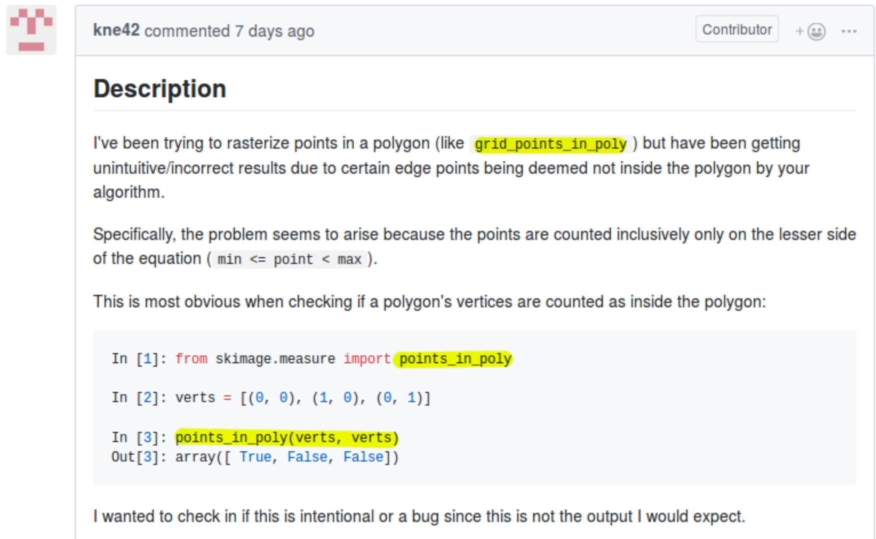


Figure 18. Example of predictions 4 https://github.com/scikit-image/scikit-image/issues/3892

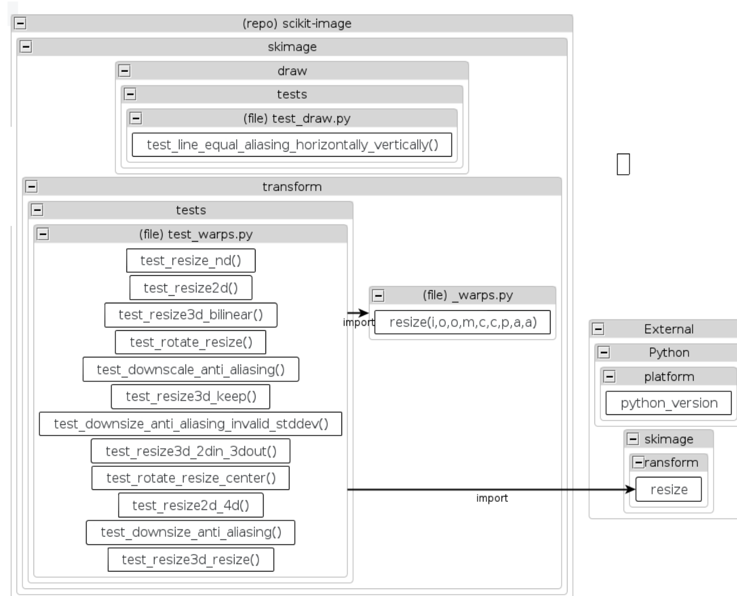


Figure 19. Example of graph generation from Figure 17

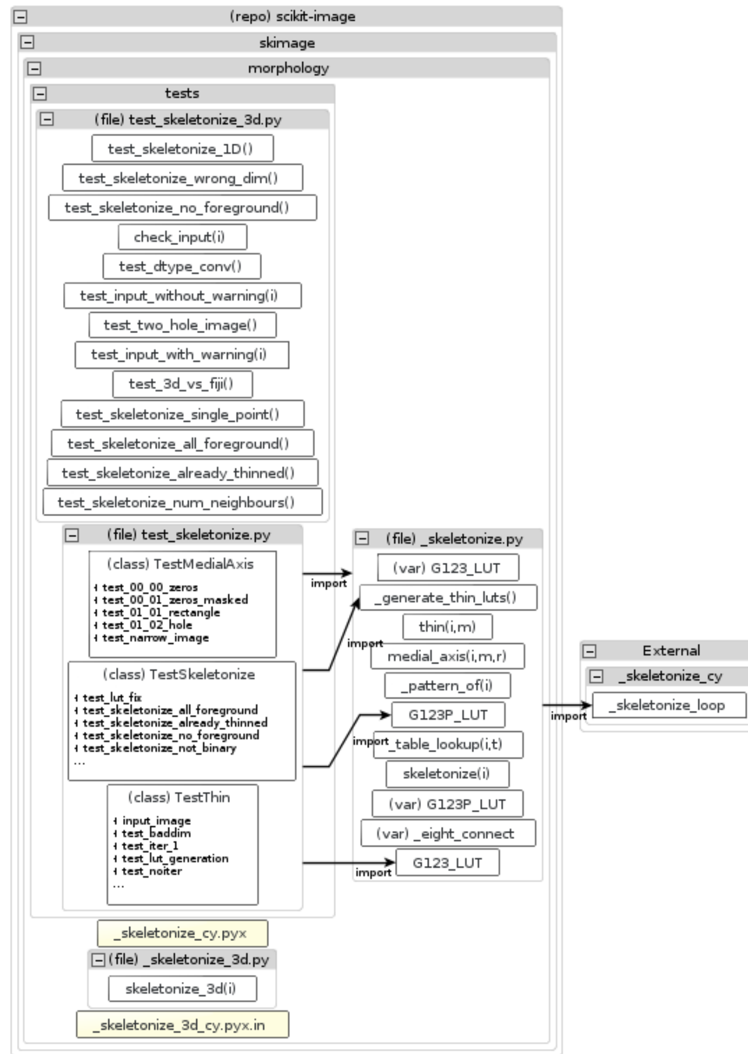


Figure 20. Example of graph generation from Figure 14

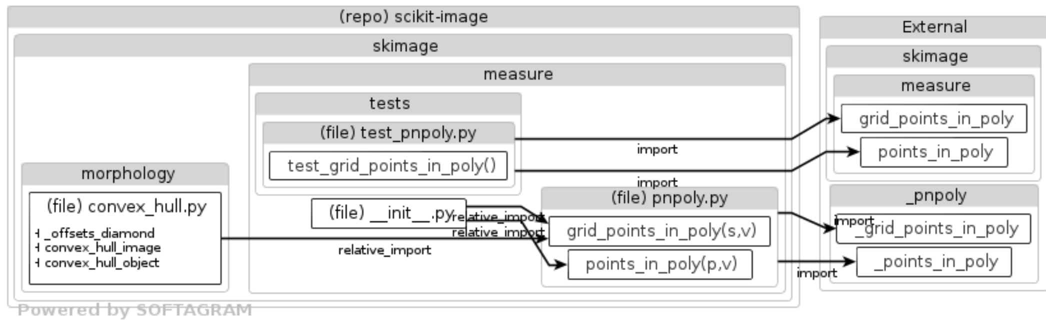


Figure 21: Example of graph generation Figure 18.

6. Threats to validity

This chapter explained how the research could have gone wrong. The chapter consists of internal validity and external validity sub chapters.

6.1 Internal validity

Due to the lack of computational power and time constraints, hyper-parameters of the Fasttext model and source file attributes extraction model can not be explored. In addition, the mapping of each word to an n-dimensional vector using Fasttext is done in an unsupervised way. That is, there is no guarantee that the same result can be achieved when the model is trained from scratch. This might cause a problem when others are trying to replicate the same work since the result could be different.

In addition, the Fasttext training used all available data to train, and the deep learning model used a subset of the same data. That was, the source file attributes in the test dataset were included in the word embedding model which does not reflect the real-world environment.

The dataset is quite poor in the sense that it does not grasp the main context of the issues. In addition, there is a lot of noise in the dataset which could introduce the bias to the models. It could be improved by using manual labeling for the task. However, it would be costly to label the data.

The description cleaning was also done in a fashion that it could fit the time limit. Since the training pipeline was quite long and time-consuming, where there are steps to train Fasttext model, vectorize all inputs, and train the source file extraction models, it was difficult to experiment on different types of input sources.

6.2 External validity

The dataset was programmatically labeled by rules which was not ideal as it was hard to construct rules that could imitate developers' understanding. Even the model could achieve acceptable f1 score from this particular dataset, it might not translate to the distribution of the issues in the real-world setting. In other words, there are rooms of improvements in the dataset to achieve real-world human understanding representation of it.

In addition, the Softagram graph generators have rooms to improve. There are too many parameters for the graph generation. For example, the recursion of dependencies, the level of details, and the dependencies views are crucial parameters to generate an easy to understand graphs. However, some values such as the recursion of dependencies is the project specific value, since it means how deep the dependencies is mined and each project has its own way of organizing the source code pattern. These types of parameter might cause a problem in the production environment.

7. Conclusion and Future work

The research questions that drives this thesis are:

RQ1 How to detect source code attributes automatically?

RQ2 How to build a visualization of an issue/bug description?

Word embedding and deep learning models were trained to solve the source file attribute findings. It showed that both methods worked well and did generalize in real-world usage with around 20% of FN and 0.8% for FP. The processes were querying data from open Github archive in Google BigQuery and training a Fasttext model to map the words together based on semantics. Then, sequential deep learning models were trained to classify whether each tokenized string was a source file attribute or not. The models could achieve around 0.80 F1 scores on the test set.

Along with the model, the generated usage graphs that are the final output of the thesis work were presented for selected real-world issues from scikit-image repository. In the selected examples, they successfully represent the issue as dependency graphs of source file attributes where the graphs show the usage relationship for those source file attributes.

7.1 Future Work

In the future, there are possibilities to generate high-quality training data for the task of issue understanding. Having realistic data would totally enhance the model to learn the useful pattern which imitates the human understanding. Therefore, the noises in the dataset should be eliminated and might even capture some keywords that are not source file attributes, and make the model moves closer to the human level understanding. In addition to getting better data, using an unsupervised method for training this task would be beneficial since there is no need to spend labor in data labeling.

Moreover, given that once the machine learning models are good enough to predict the source file attributes, it would be very interesting to visualize the output into a dynamic graph environment. That is, the users can modify the dependency graphs upon their own intentions.

References

Aken, van, J. E. (2005). Management research as a design science: articulating the research products of mode 2 knowledge production in management. *British Journal of Management*, *16*(1), 19–36. https://doi.org/10.1111/j.1467-8551.2005.00437.x

Anvik, J., Hiew, L., & Murphy, G. C. (2006). Who Should Fix This Bug? *Proceedings of the 28th International Conference on Software Engineering*, 361–370. https://doi.org/10.1145/1134285.1134336

Bertram, D., Voida, A., Greenberg, S., & Walker, R. (2010). Communication, Collaboration, and Bugs: The Social Nature of Issue Tracking in Small, Collocated Teams. *Proceedings of the 2010 ACM Conference on Computer Supported Cooperative Work*, 291–300. https://doi.org/10.1145/1718918.1718972

Bettenburg, N., Just, S., Schröter, A., Weiß, C., Premraj, R., & Zimmermann, T. (2007). Quality of Bug Reports in Eclipse. *Proceedings of the 2007 OOPSLA Workshop on Eclipse Technology EXchange*, 21–25. https://doi.org/10.1145/1328279.1328284

Bissyandé, T. F., Lo, D., Jiang, L., Réveillère, L., Klein, J., & Traon, Y. L. (2013). Got issues? Who cares about it? A large scale investigation of issue trackers from GitHub. *2013 IEEE 24th International Symposium on Software Reliability Engineering (ISSRE)*, 188–197. https://doi.org/10.1109/ISSRE.2013.6698918

Boehm, B. (2002). Get ready for agile methods, with care. *Computer*, *35*(1), 64–69. https://doi.org/10.1109/2.976920

Bojanowski, P., Grave, E., Joulin, A., & Mikolov, T. (2016). Enriching Word Vectors with Subword Information. *ArXiv Preprint ArXiv:1607.04606*.

Bottou, L. (2010). Large-Scale Machine Learning with Stochastic Gradient Descent. In Y. Lechevallier & G. Saporta (Eds.), *Proceedings of COMPSTAT'2010* (pp. 177–186). Physica-Verlag HD.

Bradbury, J., Merity, S., Xiong, C., & Socher, R. (2016). Quasi-Recurrent Neural Networks. *CoRR*, *abs/1611.01576*. Retrieved from http://arxiv.org/abs/1611.01576

Chandrashekar, G., & Sahin, F. (2014). A survey on feature selection methods. *Computers & Electrical Engineering*, *40*(1), 16–28. https://doi.org/10.1016/j.compeleceng.2013.11.024

Choetkiertikul, M., Dam, H. K., Tran, T., Pham, T. T. M., Ghose, A., & Menzies, T. (2018). A deep learning model for estimating story points. *IEEE Transactions on Software Engineering*, 1–1. https://doi.org/10.1109/TSE.2018.2792473

Choi, K., Fazekas, G., & Sandler, M. B. (2016). Text-based LSTM networks for Automatic Music Composition. *CoRR*, *abs/1604.05358*. Retrieved from http://arxiv.org/abs/1604.05358

Chunmian, L., Lin, L., Wenting, L., Kelvin, W., & Guo, J. (2019). Transfer Learning Based Traffic Sign Recognition Using Inception-v3 Model. *Periodica Polytechnica Transportation Engineering*, *47*(3). https://doi.org/10.3311/PPtr.11480

Colaboratory – Google. (n.d.). Retrieved June 2, 2019, from https://research.google.com/colaboratory/faq.html

Convolutional Neural Network (CNN). (2018, April 23). Retrieved March 28, 2019, from NVIDIA Developer website: https://developer.nvidia.com/discover/convolutional-neural-network

Dam, H. K., Tran, T., Grundy, J., & Ghose, A. (2016). DeepSoft: A Vision for a Deep Model of Software. *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, 944–947. https://doi.org/10.1145/2950290.2983985

Das, K., Jiang, J., & Rao, J. N. K. (2004). Mean squared error of empirical predictor. *The Annals of Statistics*, *32*(2), 818–840. https://doi.org/10.1214/009053604000000201

Davis, J., & Goadrich, M. (2006). The Relationship Between Precision-Recall and ROC Curves. *Proceedings of the 23rd International Conference on Machine Learning*, 233–240. https://doi.org/10.1145/1143844.1143874

Deepu, S., Pethuru, R., & S., R. (n.d.). A Framework for Text Analytics using the Bag of Words (BoW) Model for Prediction. *International Journal of Advanced Networking & Applications*.

Domingos, P. M. (2012). A few useful things to know about machine learning. *Commun. Acm*, *55*(10), 78–87.

GH Archive. (n.d.). Retrieved May 9, 2019, from https://www.gharchive.org/

Goodfellow, I., Bengio, Y., & Courville, A. (2016). *Deep Learning*. MIT Press.

Google Drive API Python wrapper library. [Python]. (2019). Retrieved from https://github.com/gsuitedevs/PyDrive (Original work published 2013)

Gregor, S., & Hevner, A. R. (2013). Positioning and Presenting Design Science Research for Maximum Impact. *MIS Q.*, *37*(2), 337–356. https://doi.org/10.25300/MISQ/2013/37.2.01

Guo, P. J., Zimmermann, T., Nagappan, N., & Murphy, B. (2011). “Not My Bug!” and Other Reasons for Software Bug Report Reassignments. *Proceedings of the ACM 2011 Conference on Computer Supported Cooperative Work*, 395–404. https://doi.org/10.1145/1958824.1958887

Hevner, A. R. (2007). A three cycle view of design science research. *Scandinavian Journal of Information Systems*, *19*(2), 4.

Hochreiter, S. (1998). The Vanishing Gradient Problem During Learning Recurrent Neural Nets and Problem Solutions. *Int. J. Uncertain. Fuzziness Knowl.-Based Syst.*, *6*(2), 107–116. https://doi.org/10.1142/S0218488598000094

Horev, R. (2018, December 26). Style-based GANs – Generating and Tuning Realistic Artificial Faces. Retrieved March 24, 2019, from Lyrn.AI website: https://www.lyrn.ai/2018/12/26/a-style-based-generator-architecture-for-generative-adversarial-networks/

Huang, Z., Xu, W., & Yu, K. (2015). Bidirectional LSTM-CRF Models for Sequence Tagging. *CoRR*, *abs/1508.01991*. Retrieved from http://arxiv.org/abs/1508.01991

Huo, X., Li, M., & Zhou, Z.-H. (2016). Learning Unified Features from Natural and Programming Languages for Locating Buggy Source Code. *Proceedings of the Twenty-Fifth International Joint Conference on Artificial Intelligence*, 1606–1612. Retrieved from http://dl.acm.org/citation.cfm?id=3060832.3060845

Issue trackers - Atlassian Documentation. (n.d.). Retrieved March 24, 2019, from https://confluence.atlassian.com/bitbucket/issue-trackers-221449750.html

Issues | GitLab. (n.d.). Retrieved March 24, 2019, from https://docs.gitlab.com/ee/user/project/issues/

Kanai, S., Fujiwara, Y., & Iwamura, S. (2017). Preventing Gradient Explosions in Gated Recurrent Units. In I. Guyon, U. V. Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, & R. Garnett (Eds.), *Advances in Neural Information Processing Systems 30* (pp. 435–444). Retrieved from http://papers.nips.cc/paper/6647-preventing-gradient-explosions-in-gated-recurrent-units.pdf

Karpathy, A. (2015, May 21). The Unreasonable Effectiveness of Recurrent Neural Networks. Retrieved March 28, 2019, from http://karpathy.github.io/2015/05/21/rnn-effectiveness/

Karpathy, A. (n.d.). CS231n Convolutional Neural Networks for Visual Recognition. Retrieved June 1, 2019, from Visualizing what ConvNets learn website: http://cs231n.github.io/understanding-cnn/

Karras, T., Laine, S., & Aila, T. (2018). A Style-Based Generator Architecture for Generative Adversarial Networks. *CoRR*, *abs/1812.04948*. Retrieved from http://arxiv.org/abs/1812.04948

Keskar, N. S., Mudigere, D., Nocedal, J., Smelyanskiy, M., & Tang, P. T. P. (2016). On Large-Batch Training for Deep Learning: Generalization Gap and Sharp Minima. *CoRR*, *abs/1609.04836*. Retrieved from http://arxiv.org/abs/1609.04836

Lecun, Y., Bottou, L., Bengio, Y., & Haffner, P. (1998). Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, *86*(11), 2278–2324. https://doi.org/10.1109/5.726791

Lee, J. Y., & Dernoncourt, F. (2016). Sequential Short-Text Classification with Recurrent and Convolutional Neural Networks. *ArXiv:1603.03827 [Cs, Stat]*. Retrieved from http://arxiv.org/abs/1603.03827

Li, X., Jiang, H., Liu, D., Ren, Z., & Li, G. (2018). Unsupervised Deep Bug Report Summarization. *Proceedings of the 26th Conference on Program Comprehension*, 144–155. https://doi.org/10.1145/3196321.3196326

Lyubinets, V., Boiko, T., & Nicholas, D. (2018). Automated labeling of bugs and tickets using attention-based mechanisms in recurrent neural networks. *CoRR*, *abs/1807.02892*. Retrieved from http://arxiv.org/abs/1807.02892

Mani, S., Sankaran, A., & Aralikatte, R. (2018). DeepTriage: Exploring the Effectiveness of Deep Learning for Bug Triaging. *CoRR*, *abs/1801.01275*. Retrieved from http://arxiv.org/abs/1801.01275

Mastering Issues · GitHub Guides. (n.d.). Retrieved March 24, 2019, from https://guides.github.com/features/issues/

Meng, R., Zhao, S., Han, S., He, D., Brusilovsky, P., & Chi, Y. (2017). Deep Keyphrase Generation. *CoRR*, *abs/1704.06879*. Retrieved from http://arxiv.org/abs/1704.06879

Mikolov, T., Sutskever, I., Chen, K., Corrado, G., & Dean, J. (2013). Distributed Representations of Words and Phrases and their Compositionality. *CoRR*, *abs/1310.4546*. Retrieved from http://arxiv.org/abs/1310.4546

Nair, V., & Hinton, G. E. (2010). Rectified Linear Units Improve Restricted Boltzmann Machines. *Proceedings of the 27th International Conference on International Conference on Machine Learning*, 807–814. Retrieved from http://dl.acm.org/citation.cfm?id=3104322.3104425

Ng, A. (2011). Machine Learning and AI via Brain simulations. *Speech Recognition*, 43.

Nielsen, M. A. (2015). Neural Networks and Deep Learning. *Determination Press*. Retrieved from http://neuralnetworksanddeeplearning.com

Olston, C., Fiedel, N., Gorovoy, K., Harmsen, J., Lao, L., Li, F., … Soyke, J. (2017). TensorFlow-Serving: Flexible, High-Performance ML Serving. *ArXiv:1712.06139 [Cs]*. Retrieved from http://arxiv.org/abs/1712.06139

Ozbulak, U., Neve, W. D., & Messem, A. V. (2018). How the Softmax Output is Misleading for Evaluating the Strength of Adversarial Examples. *CoRR*, *abs/1811.08577*. Retrieved from http://arxiv.org/abs/1811.08577

Parikh, R. (2014, May 7). How Anomalies Can Wreck Your Data (Garbage In, Garbage Out). Retrieved June 1, 2019, from Garbage In, Garbage Out: How Anomalies Can Wreck Your Data website: https://heap.io/blog/data-stories/garbage-in-garbage-out-how-anomalies-can-wreck-your-data

Park, Y., & Jensen, C. (2009). Beyond pretty pictures: Examining the benefits of code visualization for Open Source newcomers. *2009 5th IEEE International Workshop on Visualizing Software for Understanding and Analysis*, 3–10. https://doi.org/10.1109/VISSOF.2009.5336433

Radford, A., Wu, J., Child, R., Luan, D., Amodei, D., & Sutskever, I. (2019). *Language Models are Unsupervised Multitask Learners*.

Ramachandran, P., Zoph, B., & Le, Q. V. (2017). Searching for Activation Functions. *ArXiv:1710.05941 [Cs]*. Retrieved from http://arxiv.org/abs/1710.05941

Ramos, J., & others. (2003). Using tf-idf to determine word relevance in document queries. *Proceedings of the First Instructional Conference on Machine Learning*, *242*, 133–142. Piscataway, NJ.

Reis, C. R., Fortes, R. P. de M., Pontin, R., & Fortes, M. (2002). *An Overview of the Software Engineering Process and Tools in the Mozilla Project*.

Rezatofighi, H., Tsoi, N., Gwak, J., Sadeghian, A., Reid, I., & Savarese, S. (2019). Generalized Intersection over Union: A Metric and A Loss for Bounding Box Regression. *ArXiv:1902.09630 [Cs]*. Retrieved from http://arxiv.org/abs/1902.09630

Rojas, R. (1996). *Neural Networks: A Systematic Introduction*. Retrieved from https://www.springer.com/gp/book/9783540605058

Ruder, S. (2016). An overview of gradient descent optimization algorithms. *ArXiv:1609.04747 [Cs]*. Retrieved from http://arxiv.org/abs/1609.04747

Sandler, M., Howard, A., Zhu, M., Zhmoginov, A., & Chen, L.-C. (2018). MobileNetV2: Inverted Residuals and Linear Bottlenecks. *ArXiv:1801.04381 [Cs]*. Retrieved from http://arxiv.org/abs/1801.04381

Softagram Products – See Your Software Visualized. (n.d.). Retrieved March 11, 2019, from Softagram website: https://softagram.com/products/

Sokolova, M., & Lapalme, G. (2009). A systematic analysis of performance measures for classification tasks. *Information Processing & Management*, *45*(4), 427–437. https://doi.org/10.1016/j.ipm.2009.03.002

source{d}. (n.d.). Machine Learning for Large Scale Code Analysis. Retrieved March 11, 2019, from source{d} website: https://sourced.tech/

Sourcetrail - The offline cross-platform code browser. (n.d.). Retrieved June 2, 2019, from https://www.sourcetrail.com/

Stanik, C., Montgomery, L., Martens, D., Fucci, D., & Maalej, W. (2018). A Simple NLP-Based Approach to Support Onboarding and Retention in Open Source Communities. *2018 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, 172–182. https://doi.org/10.1109/ICSME.2018.00027

Stol, K.-J., Avgeriou, P., & Ali Babar, M. (2010). Identifying Architectural Patterns Used in Open Source Software: Approaches and Challenges. *Proceedings of the 14th International Conference on Evaluation and Assessment in Software Engineering*, 91–100. Retrieved from http://dl.acm.org/citation.cfm?id=2227057.2227069

Sutskever, I., Vinyals, O., & Le, Q. V. (2014). Sequence to Sequence Learning with Neural Networks. In Z. Ghahramani, M. Welling, C. Cortes, N. D. Lawrence, & K. Q. Weinberger (Eds.), *Advances in Neural Information Processing Systems 27* (pp. 3104–3112). Retrieved from http://papers.nips.cc/paper/5346-sequence-to-sequence-learning-with-neural-networks.pdf

Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., … Polosukhin, I. (2017). Attention Is All You Need. *CoRR*, *abs/1706.03762*. Retrieved from http://arxiv.org/abs/1706.03762

Visualize Java code execution. (n.d.). Retrieved June 2, 2019, from http://findtheflow.io/

Wang, C., Yang, H., Bartz, C., & Meinel, C. (2016). Image Captioning with Deep Bidirectional LSTMs. *CoRR*, *abs/1604.00790*. Retrieved from http://arxiv.org/abs/1604.00790

Yin, W., Kann, K., Yu, M., & Schütze, H. (2017). Comparative Study of CNN and RNN for Natural Language Processing. *CoRR*, *abs/1702.01923*. Retrieved from http://arxiv.org/abs/1702.01923

Yu, A. W., Dohan, D., Luong, M.-T., Zhao, R., Chen, K., Norouzi, M., & Le, Q. V. (2018). QANet: Combining Local Convolution with Global Self-Attention for Reading Comprehension. *CoRR*, *abs/1804.09541*. Retrieved from http://arxiv.org/abs/1804.09541

Yu, L.-C., Wang, J., Lai, K. R., & Zhang, X. (2017). Refining Word Embeddings for Sentiment Analysis. *Proceedings of the 2017 Conference on Empirical Methods*

in Natural Language Processing, 534–539. https://doi.org/10.18653/v1/D17-1056

Zaninotto, F. (n.d.). CodeFlower Source code visualization. Retrieved June 2, 2019, from http://www.redotheweb.com/CodeFlower/

Zhang, Q., Wang, Y., Gong, Y., & Huang, X. (2016). Keyphrase Extraction Using Deep Recurrent Neural Networks on Twitter. *Proceedings of the 2016 Conference on Empirical Methods in Natural Language Processing*, 836–845. https://doi.org/10.18653/v1/D16-1080

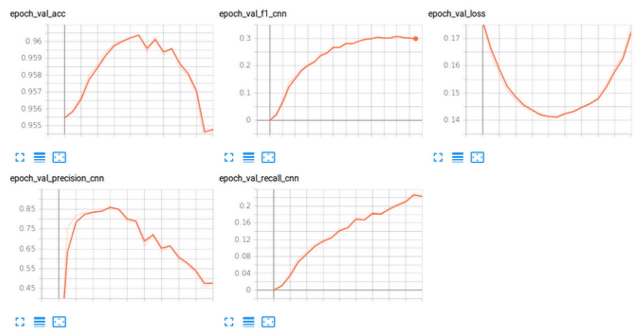
Zhou, J., Zhang, H., & Lo, D. (2012). Where should the bugs be fixed? More accurate information retrieval-based bug localization based on bug reports. *2012 34th International Conference on Software Engineering (ICSE)*, 14–24. https://doi.org/10.1109/ICSE.2012.6227210

Appendix A. Training Graphs

Training results - Model 1

Best Model Epoch# 21

loss: 0.0696	val_loss: 0.1937
acc: 0.9752	val_acc: 0.9511
f1_cnn: 0.6620	val_f1_cnn: 0.3085
precision_cnn: 0.8593	val_precision_cnn: 0.4205
recall_cnn: 0.5401	val_recall_cnn: 0.2470



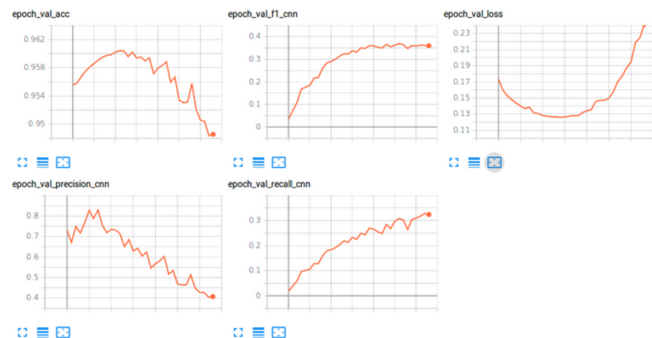
Dataset: 12k sample
Train/test split: 0.65/0.35
Optimizer: Adam
Loss: Binary Cross-Entropy
Total Train: 60 Epoch, Early Stopping with 10 threshold on validation f1-score
Batch Size: 32
Architecture: 2 layers CNN + 2 layers FCN

Figure 22. Training result for CNN model

Training results - Model 2

Best Model Epoch# 26

loss: 0.0760	val_loss: 0.1472
acc: 0.9715	val_acc: 0.9567
f1_cnn: 0.6037	val_f1_cnn: 0.3551
precision_cnn: 0.8076	val_precision_cnn: 0.5339
recall_cnn: 0.4842	val_recall_cnn: 0.2676



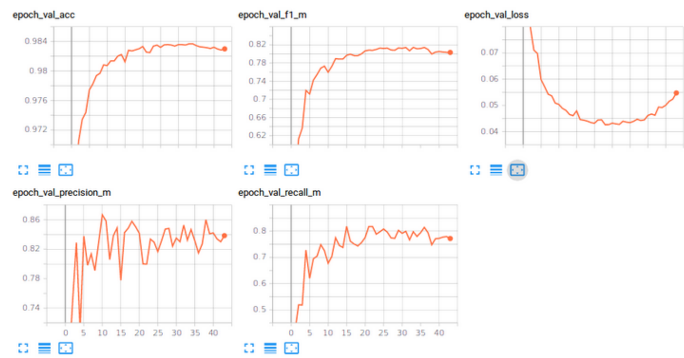
Dataset: 12k sample
Train/test split: 0.65/0.35
Optimizer: Adam
Loss: Binary Cross-Entropy
Total Train: 60 Epoch, Early Stopping with 10 threshold on validation f1-score
Batch Size: 32
Architecture: 2 layers CNN + 2 layers FCN Horizontal convolution window

Figure 23. Training result for 1D CNN model

Training results - Model 3

Best Model Epoch# 31

loss: 0.0272	val_loss: 0.0437
acc: 0.9896	val_acc: 0.9836
f1_m: 0.8818	val_f1_m: 0.8156
precision_m: 0.9047	val_precision_m: 0.8275
recall_m: 0.8621	val_recall_m: 0.8046



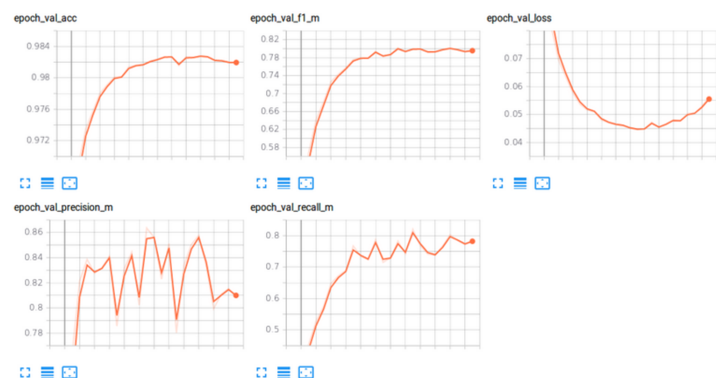
Dataset: 12k sample
Train/test split: 0.65/0.35
Optimizer: Adam
Loss: Binary Cross-Entropy
Total Train: 60 Epoch, Early Stopping with 10 threshold on validation f1-score
Batch Size: 128
Architecture: 2 layers LSTM

Figure 24. Training result for LSTM model

Training results - Model 4

Best Model Epoch# 14

loss: 0.0356	val_loss: 0.0446
acc: 0.9863	val_acc: 0.9827
f1_m: 0.8416	val_f1_m: 0.8019
precision_m: 0.8801	val_precision_m: 0.8229
recall_m: 0.8082	val_recall_m: 0.7827



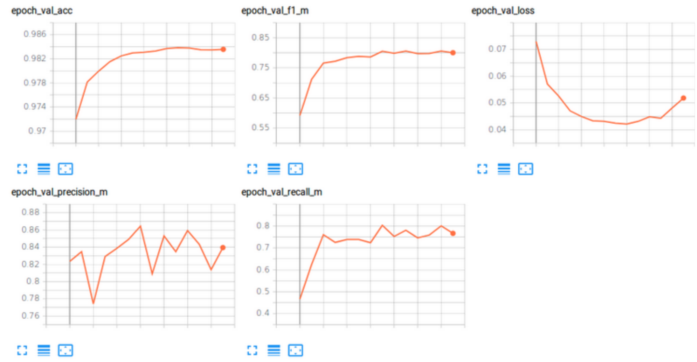
Dataset: 12k sample
Train/test split: 0.65/0.35
Optimizer: Adam
Loss: Binary Cross-Entropy
Total Train: 60 Epoch, Early Stopping with 10 threshold on validation f1-score
Batch Size: 128
Architecture: 2 layers Bi-LSTM

Figure 25. Training result for Bi-LSTM model

Training results - Model 4

Best Model Epoch# 8

loss: 0.0337	val_loss: 0.0424
acc: 0.9867	val_acc: 0.9833
f1_m: 0.8467	val_f1_m: 0.8049
precision_m: 0.8763	val_precision_m: 0.8091
recall_m: 0.8220	val_recall_m: 0.8032



Dataset: 12k sample
Train/test split: 0.65/0.35
Optimizer: Adam
Loss: Binary Cross-Entropy
Total Train: 60 Epoch, Early Stopping with 10 threshold on validation f1-score
Batch Size: 32
Architecture: 2 layers Bi-LSTM

Figure 26. Training result for Bi-LSTM model with batchsize=32