CS 510 Software Engineering
Project 3, Version 1

Instructor: Lin Tan
TA: Yi Sun
Special Thanks: Thibaud Lutellier
Release Date: October 25, 2019

Due: 11:59 PM, December 4, 2019
Group Sign-up Due: October 31, 2019 (only if you want to change your group from Project 2)
Submit: An electronic copy on BLACKBOARD

The whole project 3 is meant to be 22% of your final grade We plan to have the following breakdown for project 3: part (I): 30%, part (II): 40%, part (III): 30%, and part (IV): Bonus

You will work in a group of 1-2 members for this project. We expect that you will be in the same group as Project 2. But if you need to make changes, please notify Yi Sun by the **Group Sign-Up Due Date** above.

Please download `proj-skeleton.tar.gz` from the course repo, which you will need for this project. The skeleton contains the necessary source code and test cases for the project.

Training deep learning models may take hours or longer. Please start early; otherwise, there may not be enough machine time to finish the experiments. Also, the servers get busier/slower when many groups use them at the end when the project is due.

You are expected to use mc18.cs.purdue.edu or cuda.cs.purdue.edu machines to work on your project. Your home directory may not have enough space for the project. **Use /scratch instead, which has enough space for the project**. Remember to remove your data if you no longer need it. Several of the resources required for this project are already installed on these servers, but can also be downloaded independently.

I expect each group to work on the project independently (discussion and collaboration among group)

Submission Instructions:

Go to Blackboard → Project 3 to submit your answer. Submit only one file in .tar.gz format. Please name your file

<FirstName>-<LastName>-<Username>.tar.gz

For example, use John-Smith-jsmith.tar.gz if you are John Smith with username jsmith. The .tar.gz file should contain the following items:

- a single pdf file "proj3_sub.pdf". The first page must include your full name, and your Purdue email address. Your PDF file should contains your results for question I and a report of the improvements you tried for questions II, III, and IV as well as your results.
- a directory "q2" that contains your code (source code only; no binaries, datasets or trained models) for Question II
- a directory "q3" that contains your code (source code only; no binaries, datasets or trained models) for Question III

- (optional) a directory "q4" that contains your code (source code only; no binaries, datasets or trained models) for the competition for Question IV

  If you use new libraries for questions II, III, and IV, also include a requirements.txt that contains the list of libraries used. You can use $pip freeze > requirements.txt$ and include it with your source code.

You can submit multiple times. After submission, **please view your submissions to make sure you have uploaded the right files/versions**.

Building Line-level Defect Detection Models

In this project, you are expected to learn how to build a defect prediction model for software source code from scratch. You are required to apply deep-learning techniques, e.g., classification, tokenization, embedding, etc., to build more accurate prediction models with the dataset provided.

Background

Line-level Defect classifiers predict which lines in a file are likely to be buggy.

A typical line-level defect prediction using deep-learning consists of the following steps:

- Data extraction and labeling: Mining buggy and clean lines from a large dataset of software changes (usually GitHub).
- Tokenization and pre-processing: Deep learning algorithms take a vector as input. Since source code is text, it needs to be tokenized and transformed into a vector before being fed to the model.
- Model Building: Using the tokenized data and labels to train a deep learning classifier. Many different classifiers have been shown to work for text input (RNNs and CNNs). Most of these models can be built using TensorFlow.
- Defect Detection: Unlabelled instances (i.e., line of codes or files) are fed to the trained model that will classify them as buggy or clean.

Evaluation Metrics

Metrics, i.e., *Precision*, *Recall*, and $F1$, are widely used to measure the performance of defect prediction models. Here is a brief introduction:

$$Precision = \frac{true\ positive}{true\ positive + false\ positive} \tag{1}$$

$$Recall = \frac{true\ positive}{true\ positive + false\ negative} \tag{2}$$

$$F1 = \frac{2 * Precision * Recall}{Precision + Recall} \tag{3}$$

These metrics rely on four main numbers: *true positive*, *false positive*, *true negative*, and *false negative*. True positive is the number of predicted defective instances that are truly defective, while false positive is the number of predicted defective ones that are actually not defective. True positive records the number of predicted non-defective instances that are actually defective, while false negative is the number of predicted non-defective instances that are actually defective. F1 is the weighted average of precision and recall.

These methods are threshold-dependent and are not the best to evaluate binary classifiers. In this project, we will also use the Receiver operating characteristic curve (ROC curve) and its associated metric, Area under the ROC curve (AUC) to evaluate our trained models independently from any thresholds. The ROC curve is created by plotting the true positive rate (or recall, see definition above) against the false positive rate at various threshold settings.

$$False\ positive\ rate = \frac{false\ positive}{false\ negative + true\ negative} \tag{4}$$

(I)- Using TensorFlow to build a simple classification model

Part I will guide you through building a simple bidirectional LSTM model, while part II and III will let you explore different ways to improve it.

CS Linux Servers have the environment ready to use. The following instructions assume using one of these machines unless stated otherwise.

(We've tested on mc18.cs.purdue.edu and cuda.cs.purdue.edu. Other mc machines may or may not work)

The environment uses Python 3 and virtualenv. For more information on how to use virtualenv, please look at the virtualenv documentation (`https://virtualenv.pypa.io/en/latest/userguide/`)

```
source /homes/cs510/project-3/venv/bin/activate
```

*If you work on your own machine, after you created your virtualenv session and activated it, you can install the required library using the requirements.txt file we provided:

```
pip install --upgrade pip
pip install -r requirements.txt
```

0.1 Load the Input Data:

Since the dataset is quite large (9GB uncompressed), we put it in /homes/cs510/project-3/data folder on the servers. You can also download it from `https://drive.google.com/file/d/1MTBAQ-Nw2yPr8drU-cQPae17eSHvz-4j` if you want to work on your own machine.

If you prefer to work on your own machine, you will need to download the data and update the path in tokenization.py

The training, validation and test data are made available in pickled Pandas dataframes, respectively in train.pickle, valid.pickle, and test.pickle

The panda dataframes consists of 4 columns:

- **instance**: the line under test
- **context_before**: the context of the line under test right before the line. In this question, the context_before consists of all the lines in the functions before the tested line.
- **context_after**: the context of the line under test right after the line. In this question, the context_after consists of all the lines in the functions after the tested line.
- **is_buggy**: the label of the line tested. 0 means the line is not buggy, 1 means the line is buggy.

The first step is to load the data and tokenize it. To load the data, use the following code (modify the paths if necessary):

```
# Load the data:
with open('data/train.pickle', 'rb') as handle:
    train = pickle.load(handle)
with open('data/valid.pickle', 'rb') as handle:
    valid = pickle.load(handle)
with open('data/test.pickle', 'rb') as handle:
    test = pickle.load(handle)
```

The custom tokenizer implemented in tokenization.py is a basic java tokenizer from the javalang library (`https://github.com/c2nes/javalang`) that is enhanced to also abstract string literals and numbers different from 0 and 1.

```
# Tokenize and shape our input:
def custom_tokenize(string):
    try:
        tokens = list(javalang.tokenizer.tokenize(string))
    except:
        return []
    values = []
    for token in tokens:
        # Abstract strings
        if '"' in token.value or "'" in token.value:
            values.append('$STRING$')
        # Abstract numbers (except 0 and 1)
        elif token.value.isdigit() and int(token.value) > 1:
            values.append('$NUMBER$')
```

```
        #other wise: get the value
        else:
            values.append(token.value)
    return values


def tokenize_df(df):
    df['instance'] = df['instance'].apply(lambda x: custom_tokenize(x))
    df['context_before'] = df['context_before'].apply(lambda x: custom_tokenize(x))
    df['context_after'] = df['context_after'].apply(lambda x: custom_tokenize(x))
    return df

test = tokenize_df(test)
train = tokenize_df(train)
valid = tokenize_df(valid)

with open('data/tokenized_train.pickle', 'wb') as handle:
    pickle.dump(train, handle, protocol=pickle.HIGHEST_PROTOCOL)
with open('data/tokenized_valid.pickle', 'wb') as handle:
    pickle.dump(valid, handle, protocol=pickle.HIGHEST_PROTOCOL)
with open('data/tokenized_test.pickle', 'wb') as handle:
    pickle.dump(test, handle, protocol=pickle.HIGHEST_PROTOCOL)
```

Loading the data and tokenizing it can be done by running the script:

```
python tokenization.py
```

The tokenized dataset will be saved in the data folder under proj-skeleton (not data folder under /homes/cs510/project-3). You can change it if necessary. The tokenization should take about 80 minutes.

0.2 Preprocessing data

Once we have the tokenized data, we need to transform them into vectors before feeding them to the deep learning model.

This part can be done by running the script:

```
python preprocess.py
```

It will do the transformation and save the transformed data (x_train.pickle, etc.) under data folder.

For this question, we represent each instance as one vector of tokens:
tokenized_context_before, $< START >$, tokenized_line_under_test, $< END >$, tokenized_context_after

The tokens $< START >$ and $< END >$ indicates when the line under test starts.

For this question, we will only keep 50,000 training instances to save time. You can try to use larger dataset (1 million or more) in part II-IV.

Loading tokenized data and reshaping the input:

```
# Loading tokenized data
with open('data/tokenized_train.pickle', 'rb') as handle:
    train = pickle.load(handle)
with open('data/tokenized_valid.pickle', 'rb') as handle:
   valid = pickle.load(handle)
with open('data/tokenized_test.pickle', 'rb') as handle:
   test = pickle.load(handle)

# Reshape instances:
def reshape_instances(df):
    df["input"] = df["context_before"].apply(lambda x: " ".join(x)) + " <START> " + df["instance"].
        apply(lambda x: " ".join(x)) + " <END> " + df["context_after"].apply(lambda x: " ".join(x))
    X_df = []
    Y_df = []
    for index, rows in df.iterrows():
        X_df.append(rows.input)
        Y_df.append(rows.is_buggy)
    return X_df, Y_df

X_train, Y_train = reshape_instances(train)
X_test, Y_test = reshape_instances(test)
X_valid, Y_valid = reshape_instances(valid)

X_train = X_train[:50000]
Y_train = Y_train[:50000]
X_test = X_test[:25000]
```

```
Y_test = Y_test[:25000]
X_valid = X_valid[:25000]
Y_valid = Y_valid[:25000]
```

Since the deep learning model takes a fixed-length vector of numbers as input, we use the training set to build a vocabulary that maps each token to a number. Then we encode our training, testing and validation instances and created vectors of fixed length representing the encoded instances. We limit the size of an instance to 1,000 tokens. In part II-IV, you might want to experiment with different vector sizes.

```
# Build vocabulary and encoder from the training instances
maxlen = 1000
vocabulary_set = set()
for data in X_train:
   vocabulary_set.update(data.split())

vocab_size = len(vocabulary_set)
print(vocab_size)

# Encode training, valid and test instances
encoder = tfds.features.text.TokenTextEncoder(vocabulary_set)

def encode(text):
  encoded_text = encoder.encode(text)
  return encoded_text

X_train = list(map(lambda x: encode(x), X_train))
X_test = list(map(lambda x: encode(x), X_test))
X_valid = list(map(lambda x: encode(x), X_valid))

X_train = pad_sequences(X_train, maxlen=maxlen)
X_test = pad_sequences(X_test, maxlen=maxlen)
X_valid = pad_sequences(X_valid, maxlen=maxlen)
```

0.3 Training the model

Training and evaluation of the model is done by train_and_test.py

For our first model, we will try to train a two layers bidirectional RNN model using LSTM layers. RNNs have been known to work well with text data. A tutorial showing how to create a basic RNN model with TensorFlow is available on `https://www.tensorflow.org/tutorials/text/text_classification_rnn`

Our model will be defined as followed:

```
# Model Definition
model = tf.keras.Sequential([
    tf.keras.layers.Embedding(encoder.vocab_size, 64),
    tf.keras.layers.Bidirectional(tf.keras.layers.LSTM(64,  return_sequences=True)),
    tf.keras.layers.Bidirectional(tf.keras.layers.LSTM(32)),
    tf.keras.layers.Dense(64, activation='relu'),
    tf.keras.layers.Dropout(0.5),
    tf.keras.layers.Dense(1, activation='sigmoid')
])

model.compile(loss='binary_crossentropy',
              optimizer=tf.keras.optimizers.Adam(1e-4),
              metrics=['accuracy'])

model.summary()
```

Since the data is pretty large, we might not be able to fit an embedding for the entire dataset in memory. Therefore, we need to build a batch generator to generate the embedding for the input data on the fly.

```
# Building generators
class CustomGenerator(Sequence):
    def __init__(self, text, labels, batch_size, num_steps=None):
        self.text, self.labels = text, labels
        self.batch_size = batch_size
        self.len = np.ceil(len(self.text) / float(self.batch_size)).astype(np.int64)
        if num_steps:
            self.len = min(num_steps, self.len)
    def __len__(self):
        return self.len
    def __getitem__(self, idx):
        batch_x = self.text[idx * self.batch_size:(idx + 1) * self.batch_size]
        batch_y = self.labels[idx * self.batch_size:(idx + 1) * self.batch_size]
```

```
        return batch_x, batch_y


train_gen = CustomGenerator(X_train, Y_train, batch_size)
valid_gen = CustomGenerator(X_valid, Y_valid, batch_size)
test_gen = CustomGenerator(X_test, Y_test, batch_size)
```

We feed this data generator and start training the model as shown below:

```
# Training the model
checkpointer = ModelCheckpoint('data/models/model-{epoch:02d}-{val_loss:.5f}.hdf5',
                               monitor='val_loss',
                               verbose=1,
                               save_best_only=True,
                               mode='min')

callback_list = [checkpointer] #, , reduce_lr
his1 = model.fit_generator(
                    generator=train_gen,
                    epochs=1,
                    validation_data=valid_gen,
                    callbacks=callback_list)
```

0.4 Evaluating the model

Once the model is trained, we evaluate it on the test set. predict_generator will generate a probability of a given instance to be buggy or clean.

Traditionally, instances will be then classified in the class 0 (i.e., clean) if the probability is lower than 50%, and in the class 1 (i.e., buggy) if the probability is higher. However, using the 50% threshold might not be the best choice and using a different threshold might provide better results. Therefore, to take into consideration the impact of the threshold, we draw the ROC curve and use the AUC (area under the curve metrics) to measure the correctness of our classifier.

```
predIdxs = model.predict_generator(test_gen, verbose=1)

fpr, tpr, _ = roc_curve(Y_test, predIdxs)
roc_auc = auc(fpr, tpr)

plt.figure()
lw = 2
plt.plot(fpr, tpr, color='darkorange', lw=lw, label='ROC curve (area = %0.2f)' % roc_auc)
plt.plot([0, 1], [0, 1], color='navy', lw=lw, linestyle='--')
plt.xlim([0.0, 1.0])
plt.ylim([0.0, 1.05])
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')
plt.title('Receiver operating characteristic example')
plt.legend(loc="lower right")

plt.savefig('auc_model.png')
```

For Part (I), please include auc_model.png in your report, and measure the buggy rate (i.e. % of instances labeled 1) in the training, validation and test instances.

(II)- Improving the results by using a better deep-learning algorithm

The model trained in part (I) is simple and does not perform very well. In the past few years, many different models to classify text inputs for diverse tasks (content tagging, sentiment analysis, translation, etc.) have been proposed in the literature. In part (II), you will look at the literature and apply a different deep-learning algorithm to do defect prediction. You can, and are encouraged to use or adapt models that have been proposed by other people for other tasks. Please cite your source and provide a link to a paper or/and GitHub repository showing that this algorithm has been applied successfully for text classification, modeling or generation tasks.

Examples of models to try:

Hierarchical Attention Networks for Document Classification:
`https://www.cs.cmu.edu/~./hovy/papers/16HLT-hierarchical-attention-networks.pdf` and `https://github.com/richliao/textClassifier`,

Independently Recurrent Neural Network (IndRNN): Building A Longer andDeeper RNN: `https://arxiv.org/pdf/1803.04831.pdf` and `https://github.com/titu1994/Keras-IndRNN`

Feed-Forward Networks with Attention Can Solve Some Long-Term Memory Problems: `https://arxiv.org/pdf/1512.08756.pdf` and `https://github.com/ShawnyXiao/TextClassification-Keras`

You can also look at more complex models like BERT, Elmo or XLNet.

You can search GitHub for text-classification models and pick the one you like!

We strongly recommend that you do not implement the CNN-only models from this paper: `https://arxiv.org/abs/1408.5882`. We have extensively tested this model for our specific task and we already know it does not work well.

Report: For this question, please put in your report the model you chose, a link to the paper and/or GitHub repository where you got the model, a small discussion why you chose to try this model, your source code, and an evaluation of your trained model on the test set (AUC and ROC curve). If you get any improvement compared to the model used in part I, please report it too.

If the model you pick is too complex and takes too long to train on the entire training set, please provide an explanation indicating how much time the model would take to train on the entire dataset and only train your model on a sample of the dataset.

(III)- Other ways to improve the results

In this question, you will try to improve the model you worked with in Part II using different methods. **Chose at least two of the methods below** to try to improve the results you got in part II: Report which methods you use and its impact on the results and training time.

Use more training data: In part I), we only used 50,000 instances to train our model. You can try to train your model with the entire training set instead. Based on our experience, using 1 million instances produce much higher AUC than using 50,000 instances. Generally, the more training instances, the higher AUC until it saturates. The constraint is machine time.

Data cleaning: The input data we provided is automatically extracted from GitHub and likely contains a lot of noise. To improve the results, one possibility is to clean the datasets. You can investigate a bit more the raw data and try to clean the input data. Examples (non-exhaustive) of challenges to investigate and solve are:

- Duplicate instances: Are there any instances that are labeled both buggy and clean?
- Length of the input: What is the average length of an instance, are there any outliers? Does removing outliers improve the results?
- Quality of the input: Comments have not been removed from the inputs? Does removing comments help to improve the results?

Tokenization and input abstraction:

In this project, we use a simple tokenization using a java tokenizer and basic abstraction of strings and numbers. This has the inconvenience of creating a gigantic vocabulary that might be difficult to learn for a deep learning network. Many different tokenizers or abstractions can be tried:

- Source code contains structured information that could help abstract data to reduce the vocabulary size. For example, all variables could be abstracted to the same token *variable*, all method calls to the token $method_call$, types to *type*, etc. You can also distinguish between different variables in the same instance by abstracting different variables with slightly different tokens (e.g., var_1, var_2, etc. Such information can be extracted from an AST or a java Parser (the javalang library contains a basic AST parser that could be used). Using such an abstract will significantly reduce the vocabulary and might help the algorithm to learn.
- Subword tokenizers have been used in NLP. You can try tokenizers like SentencePiece (`https://github.com/google/sentencepiece`, or word pieces (`https://www.tensorflow.org/datasets/api_docs/python/tfds/features/text/SubwordTextEncoder`)
- You can also build your own tokenizer.

Context representation: In part I), the context is represented as a sequence of tokens from the entire function. In addition, both the context and the line under test are represented similarly and fed as one input. This might not be the best way to represent the context of a bug. You can propose a different approach to represent the context of a bug:

- You can try to represent the context differently (e.g., use a higher-level abstraction, only use a set of tokens instead of a sequence)
- In this project, we use the entire function as context. This provides a lot of information, but it likely also contains a lot of noise (e.g. irrelevant statements). You can try to use a different context (e.g., reduce the context to only consider the basic block surrounding the line under test).
- You can try to feed the context and the instance under test as different inputs.

Tuning and Building Deeper models: Deep learning models contain a lot of hyper-parameters that can be tuned (e.g., number of epoch trained, number and size of layers, dropout rate, learning rate, etc.). Using different hyper-parameters can lead to very different results. One way to improve the results of a classifier is to pick the "best" hyper-parameters by tuning the model.

Using different Learning methods: Sometimes, learning from one model and one dataset is not enough to achieve good results. There are several possibilities to improve the models:

- Use pre-trained embedding to have a better source code representation. Much work has been done to represent source code from a very large corpus. Instead of training our embedding layers from our limited training data, you could use a pre-trained embedding (e.g. such as the ones proposed in code2seq `https://github.com/tech-srl/code2seq` or train your own embedding (e.g., GloVe or Word2Vec) before training the classifier.
- It is easier to learn from simple instances first. Curriculum Learning has been proposed to help to learn easier instances first. (`https://arxiv.org/abs/1904.03626`)
- Use ensemble learning. One model might not be enough to learn all buggy lines. Instead of building one single model, a combination of several smaller models (trained with different training data or using different hyper-parameters) might provide better performances.

(IV)- Further improvements (competition) - for Bonus

You are also highly encouraged to improve the defect prediction models by using other techniques beyond the ones we recommended or to try to combine all of them to further improve your model.

(Optional) Use GPU for your training

GPU can drastically accelerate the speed of training. In this part, we will guide you to use tensorflow-gpu to train the model.

Server cuda.cs.purdue.edu is equipped with 6 GPUs capable of deep learning. You should have access to this server. However, most of the time, its GPUs are occupied by others, which is out of our control. We highly recommend that you consider using GPUs if you have access to one.

Use GPU of cuda.cs.purdue.edu

We have environment ready to use on this server. Run nvidia-smi to check the avaibility of GPUs before you start. To run training on this server using GPU, you should follow the steps:

```
module load cuda/10.0
source /homes/cs510/project-3/venv-gpu/bin/activate
python train_and_test.py
```

You may get a OUT_OF_MEMORY error if no GPU is available at that time. Since we don't have control over the server, we cannot guarantee your access to the GPU. You may try at different time.

Use your own GPU

If you have control over a machine with Nvidia GPU. You may use tensorflow-gpu to accelerate your training (The performance is varied based on the model).

Prerequisites

- Python3
- CUDA Toolkit 10.0 (`https://developer.nvidia.com/cuda-10.0-download-archive`)
- cuDNN (Any version that is compatible with cuda10.0 `https://developer.nvidia.com/cudnn`)

Once you have meet the prerequisites, you can create a virtualenv and use the provide requirements-gpu.txt to setup your environment.

```
python3 -m venv path/to/venv
source path/to/venv/bin/activate
pip install --upgrade pip
pip install -r requirements-gpu.txt
```

Then, you should be ready to train your model on a large dataset faster.