Introduction

This document describes all the important functions in the source code, in the order of preprocessing, building the classifier, training and testing.

Preprocess

The codes for preprocessing are located in the folder 'utility'. 'file_loader.py' is for the data preprocessing, and 'pre_train.py' preprocesses the pre-train weight.

file_loader.py

This class contains functions of reading the file, lowercase the data, removing stopwords, splitting the raw dataset, create the vocabulary file and label file, and encoding sentences and labels. After creating an object of this class and passing the path to the inside function 'read_file', we can do other preprocessing stops for the target data file using the object and its functions.

The followings are the descriptions of the main functions in the class. There are also three helper functions: 'read_stopwords' and 'remove_stopwords' help to remove the stopwords of the data, and 'read_vocab_and_label' helps reading the vocab/label files to encode the sentence and labels.

```
def read_file(self, path, stop_path=''):
    """
    read the original questions from the path file,
    and remove the stopwords from the stopwords file.
    input empty string if the stopwords are not needed
    and split the data into label part and sentence part, store them in the
object.

    :param path: The path question file (raw /train /dev /test...)
    :param stop_path: The path to the stopwords file
    """
```

```
def split_dataset(self, train_file_name, dev_file_name, ratio=0.1):
    """
    This function split the original dataset into train set and development set
    :param dev_file_name:  path to dev.txt
    :param train_file_name: path to train.txt
    :param ratio: The ratio of train data size and development data size
    :return:
    """
```

```
def create_vocab_and_label(self, vocab_path='', label_path=''):
    """
    This function creates the vocabulary/labels for the raw_data.txt
    and store the vocabularies in the vocabulary.txt/ labels.txt
    NOTE THAT THIS FUNCTION IS ONLY FOR THE raw_data.txt
    :param vocab_path: The path to store the vocabulary.txt
    :param label_path: The path to store the labels.txt
    """
```

```
def get_encoded_data(self, padding):
    """
    This function encodes the sentences and labels
    :return: The encoded data in the format of
    [([encoded sentence], encoded label),...] -> example. [([3,1,4,5], 2),...]
    """
```

pre_train.py

This class is used for get the pre-train weight that suitable for our data. More specifically, it reads the pre-trained embeddings from the file and gets the pre-trained embeddings for the words which are in our vocabulary list. If the words are in our vocabulary but not in the pre-trained list, mark the words to #unk#. If the words are in the pre-trained file but not appears in our data, just remove the embeddings.

```
def load_pretrain(self, path, vocab):
    """
    This function creates a dictionary for the words and vectors in
glove.small.txt
    :param vocab: The vocabulary of the raw_data.txt
    :param path: The path of glove.small.txt
    :return: The dictionary
    """
```

Build Classifier

There are two main steps to implement the question classifier. The first one is the sentence representation, and the second one is classification.

We use BoW and BiLSTM to train the sentence representation (get the sentence vector), and use a feed-forward neural network to implement the classification task.

bow.py

```
class BoW(nn.Module):
    def __init__(self,
                 pre_train_weight,
                 vocab_size,
                 pre_train,
                 freeze,
                 embedding_dim):
```

in the above class we built the BoW layer which uses nn.EmbeddingBag to get the sentence vector of the input sentences.

biLSTM.py

```
class BiLSTM(nn.Module):
    def __init__(self,
                 pre_train_weight,
                 vocab_size,
                 pre_train,
                 freeze,
                 embedding_dim,
                 hidden_dim_bilstm):
```

in the BiLSTM class we built the BiLSTM layer which uses recurrent neural network to get the sentence vector from the context. It joints the forward and backward information together to represent the sentence.

classification.py

```
class Classification(nn.Module):
    def __init__(self, n_input, n_hidden, n_output):
```

This is a feed-forward neural network with logSoftmax output to implement the classification task. Using logSoftmax instead of softmax to fit the later NLL Loss function.

model.py

```
def forward(self,input):
    out = self.sentence_rep(input)
    out = self.classifier(out)
    return out
```

This part integrated the sentence representation and classification together. The output of the bow/bilstm is the sentence vector, which is as the input of the later classification Neural Network.

Train & Test

`question_classifier.py`

'question_classifier.py' is the main file which is mainly used to training, evaluation and test the model.

helpers

The followings are the helper functions for training and the descriptions are as follows inside the function.

```
def load_raw_file(config):
    '''
    create a File_loader object,
    in order to read the raw data file, create the vocabulary and label file, and
split the data into train set and validation set.
    :return: the whole vocabulary (from the raw data)
    '''
```

```
def get_vocab(config):
    '''
    read the existing vocabulary file and return the vocabulary
    '''
```

```
def get_encoded_data(path_file, path_vocab,path_label, path_stop, padding):
    '''
    create a File_loader object, in order to read the file
(train/validation/test..)
    and encode the labels and sentences in the file.
    after encoding, add the paddings to make each sentence with the same length,
padding=0.
    :return: the encoded data
    '''
```

```
def compute_acc(outputs, target):
    '''
    compute the accuracy and the f1_score of the model using sklearn.metrics.
    '''
```

train & test

Function 'train' will run when detecting --train in the command line, and 'test' will run when detecting --test.

In the train function, first get the data ready, then create a model object, and start training. In each epoch, the validation data was used to test the performance of the current model.

After all epochs, the model will be saved in 'model_path' (can be set in the config.ini).

Once the model was saved, when in 'test' mode, the test function would load the trained model, and use test data to test the performance of the model (print accuracy and f1 score). Then output a txt file, in which each line is a class for each testing question and the end of which is the performance of the model.