In [2]:
```
#https://medium.com/swlh/a-simple-guide-on-using-bert-for-text-classificatio
from __future__ import absolute_import, division, print_function
import pandas as pd

train_df = pd.read_csv('data/train_60.csv', error_bad_lines=False)
train_df.head()
```

Out[2]:

	qid	question_text	target
0	b468dea651573e2c284a	I am fed up while waiting for Ericsson joining...	0
1	81fa64a5da93d64d20e2	Am I the only one who just wants to eat whenev...	0
2	998b3cac31f31d87045b	How can I study economics in IIM?	0
3	ea5fbd69537138afeb2f	What is the best way to prepare for architecture?	0
4	4812ed679e33339581b5	Do you think showing the hypocrisy of others c...	0

In [1]:
```
import tools
```

```
---------------------------------------------------------------------------
ModuleNotFoundError                       Traceback (most recent call last)
<ipython-input-1-c94ac9b044f5> in <module>
----> 1 import tools

ModuleNotFoundError: No module named 'tools'
```

In [ ]:
```
test_df = pd.read_csv('data/val_20.csv',error_bad_lines=False)
test_df.head()
```

In [ ]:
```
train_df.shape
```

In [ ]:
```
train_df['target']
```

In [ ]:
```
train_df_bert = pd.DataFrame({
    'id':train_df['qid'],
    'label':train_df['target'],
    'alpha':['a']*train_df.shape[0],
    'text': train_df['question_text']
})

train_df_bert.head()
```

In [ ]:
```
dev_df_bert = pd.DataFrame({
    'id':test_df['qid'],
    'label':test_df['target'],
    'alpha':['a']*test_df.shape[0],
    'text': test_df['question_text'].replace(r'\n', ' ', regex=True)
})

dev_df_bert.head()
```

In [ ]:
```
train_df_bert.to_csv('data/train.tsv', sep='\t', index=False, header=False)
```

In [ ]:
```
dev_df_bert.to_csv('data/dev.tsv', sep='\t', index=False, header=False)
```

In [ ]:
```

```

In [ ]:

```
import csv
import os
import sys
import logging

logger = logging.getLogger()
csv.field_size_limit(2147483647) # Increase CSV reader's field limit incase


class InputExample(object):
    """A single training/test example for simple sequence classification."""

    def __init__(self, guid, text_a, text_b=None, label=None):
        """Constructs a InputExample.
        Args:
            guid: Unique id for the example.
            text_a: string. The untokenized text of the first sequence. For
            sequence tasks, only this sequence must be specified.
            text_b: (Optional) string. The untokenized text of the second se
            Only must be specified for sequence pair tasks.
            label: (Optional) string. The label of the example. This should
            specified for train and dev examples, but not for test examples.
        """
        self.guid = guid
        self.text_a = text_a
        self.text_b = text_b
        self.label = label


class DataProcessor(object):
    """Base class for data converters for sequence classification data sets.

    def get_train_examples(self, data_dir):
        """Gets a collection of `InputExample`s for the train set."""
        raise NotImplementedError()

    def get_dev_examples(self, data_dir):
        """Gets a collection of `InputExample`s for the dev set."""
        raise NotImplementedError()

    def get_labels(self):
        """Gets the list of labels for this data set."""
        raise NotImplementedError()

    @classmethod
    def _read_tsv(cls, input_file, quotechar=None):
        """Reads a tab separated value file."""
        with open(input_file, "r", encoding="utf-8") as f:
            reader = csv.reader(f, delimiter="\t", quotechar=quotechar)
            lines = []
            for line in reader:
                if sys.version_info[0] == 2:
                    line = list(unicode(cell, 'utf-8') for cell in line)
                lines.append(line)
            return lines
```

```
class BinaryClassificationProcessor(DataProcessor):
    """Processor for binary classification dataset."""

    def get_train_examples(self, data_dir):
        """See base class."""
        #print('rekha--------------------')
        #print(os.path.join(data_dir, "train.tsv"))
        return self._create_examples(
            self._read_tsv(os.path.join(data_dir, "train.tsv")), "train")

    def get_dev_examples(self, data_dir):
        """See base class."""
        return self._create_examples(
            self._read_tsv(os.path.join(data_dir, "dev.tsv")), "dev")

    def get_labels(self):
        """See base class."""
        return ["0", "1"]

    def _create_examples(self, lines, set_type):
        """Creates examples for the training and dev sets."""
        examples = []
        for (i, line) in enumerate(lines):
            if(len(line) == 0):
                print('Found empty line')
            elif(len(line)<4):
                print(line)
#                 print('line[0]', line[0])
#                 print('line[1]', line[1])
#                 print('line[2]', line[2])
#                 print('line[3]', line[3])
            else:
                #guid = "%s-%s" % (set_type, i)
                guid = line[0]
                text_a = line[3]
                label = line[1]
                examples.append(
                    InputExample(guid=guid, text_a=text_a, text_b=None, labe
        return examples
```

In [ ]:

```
class InputFeatures(object):
    """A single set of features of data."""

    def __init__(self, input_ids, input_mask, segment_ids, label_id):
        self.input_ids = input_ids
        self.input_mask = input_mask
        self.segment_ids = segment_ids
        self.label_id = label_id


def _truncate_seq_pair(tokens_a, tokens_b, max_length):
    """Truncates a sequence pair in place to the maximum length."""

    # This is a simple heuristic which will always truncate the longer seque
    # one token at a time. This makes more sense than truncating an equal pe
    # of tokens from each, since if one sequence is very short then each tok
    # that's truncated likely contains more information than a longer sequen
    while True:
        total_length = len(tokens_a) + len(tokens_b)
        if total_length <= max_length:
            break
        if len(tokens_a) > len(tokens_b):
            tokens_a.pop()
        else:
            tokens_b.pop()


def convert_example_to_feature(example_row):
    # return example_row
    example, label_map, max_seq_length, tokenizer, output_mode = example_row

    tokens_a = tokenizer.tokenize(example.text_a)

    tokens_b = None
    if example.text_b:
        tokens_b = tokenizer.tokenize(example.text_b)
        # Modifies `tokens_a` and `tokens_b` in place so that the total
        # length is less than the specified length.
        # Account for [CLS], [SEP], [SEP] with "- 3"
        _truncate_seq_pair(tokens_a, tokens_b, max_seq_length - 3)
    else:
        # Account for [CLS] and [SEP] with "- 2"
        if len(tokens_a) > max_seq_length - 2:
            tokens_a = tokens_a[:(max_seq_length - 2)]

    tokens = ["[CLS]"] + tokens_a + ["[SEP]"]
    segment_ids = [0] * len(tokens)

    if tokens_b:
        tokens += tokens_b + ["[SEP]"]
        segment_ids += [1] * (len(tokens_b) + 1)

    input_ids = tokenizer.convert_tokens_to_ids(tokens)

    # The mask has 1 for real tokens and 0 for padding tokens. Only real
    # tokens are attended to.
```

```
    input_mask = [1] * len(input_ids)

    # Zero-pad up to the sequence length.
    padding = [0] * (max_seq_length - len(input_ids))
    input_ids += padding
    input_mask += padding
    segment_ids += padding

    assert len(input_ids) == max_seq_length
    assert len(input_mask) == max_seq_length
    assert len(segment_ids) == max_seq_length

    if output_mode == "classification":
        label_id = label_map[example.label]
    elif output_mode == "regression":
        label_id = float(example.label)
    else:
        raise KeyError(output_mode)

    return InputFeatures(input_ids=input_ids,
                         input_mask=input_mask,
                         segment_ids=segment_ids,
                         label_id=label_id)
```

In [2]:

```
import torch
import pickle
from torch.utils.data import (DataLoader, RandomSampler, SequentialSampler,
from torch.nn import CrossEntropyLoss, MSELoss

from tqdm import tqdm_notebook, trange
import os
from pytorch_pretrained_bert import BertTokenizer, BertModel, BertForMaskedL
from pytorch_pretrained_bert.optimization import BertAdam

from tools import *
from multiprocessing import Pool, cpu_count

import convert_examples_to_features

# OPTIONAL: if you want to have more information on what's happening, activa
import logging
logging.basicConfig(level=logging.INFO)

device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
```

```
Better speed can be achieved with apex installed from https://www.github.co
m/nvidia/apex. (https://www.github.com/nvidia/apex.)

---------------------------------------------------------------------------
ModuleNotFoundError                       Traceback (most recent call last)
<ipython-input-2-8b7f662647cf> in <module>
     12 from multiprocessing import Pool, cpu_count
     13
---> 14 from bert.run_classifier import convert_examples_to_features
     15
     16 # OPTIONAL: if you want to have more information on what's happenin
g, activate the logger as follows

ModuleNotFoundError: No module named 'bert'
```

In [ ]:

```
# The input data dir. Should contain the .tsv files (or other data files) fo
DATA_DIR = "data/"

# Bert pre-trained model selected in the list: bert-base-uncased,
# bert-large-uncased, bert-base-cased, bert-large-cased, bert-base-multiling
# bert-base-multilingual-cased, bert-base-chinese.
BERT_MODEL = 'bert-base-uncased'

# The name of the task to train.I'm going to name this 'yelp'.
TASK_NAME = 'quora'

# The output directory where the fine-tuned model and checkpoints will be wr
OUTPUT_DIR = f'outputs/{TASK_NAME}/'

# The directory where the evaluation reports will be written to.
REPORTS_DIR = f'reports/{TASK_NAME}_evaluation_report/'

# This is where BERT will look for pre-trained models to load parameters fro
CACHE_DIR = 'cache/'

# The maximum total input sequence length after WordPiece tokenization.
# Sequences longer than this will be truncated, and sequences shorter than t
MAX_SEQ_LENGTH = 128

TRAIN_BATCH_SIZE = 24
EVAL_BATCH_SIZE = 32
LEARNING_RATE = 2e-5
NUM_TRAIN_EPOCHS = 1
RANDOM_SEED = 42
GRADIENT_ACCUMULATION_STEPS = 1
WARMUP_PROPORTION = 0.1
OUTPUT_MODE = 'classification'

CONFIG_NAME = "config.json"
WEIGHTS_NAME = "pytorch_model.bin"
```

In [ ]:

```
output_mode = OUTPUT_MODE

cache_dir = CACHE_DIR

if os.path.exists(REPORTS_DIR) and os.listdir(REPORTS_DIR):
        REPORTS_DIR += f'/report_{len(os.listdir(REPORTS_DIR))}'
        os.makedirs(REPORTS_DIR)
if not os.path.exists(REPORTS_DIR):
    os.makedirs(REPORTS_DIR)
    REPORTS_DIR += f'/report_{len(os.listdir(REPORTS_DIR))}'
    os.makedirs(REPORTS_DIR)

if os.path.exists(OUTPUT_DIR) and os.listdir(OUTPUT_DIR):
        raise ValueError("Output directory ({}) already exists and is not em
if not os.path.exists(OUTPUT_DIR):
    os.makedirs(OUTPUT_DIR)
```

In [ ]:
```
print("REKHAAAAAAAAAAAAAA use BinaryClassificationProcessor to load in the d
processor = BinaryClassificationProcessor()
train_examples = processor.get_train_examples(DATA_DIR)
train_examples_len = len(train_examples)
print("REKHAAAAAAAAAAAAAA get_train_examples completed")
```

In [ ]:
```
train_examples_len
```

In [ ]:
```
label_list = processor.get_labels() # [0, 1] for binary classification
num_labels = len(label_list)
```

In [ ]:
```
num_train_optimization_steps = int(
    train_examples_len / TRAIN_BATCH_SIZE / GRADIENT_ACCUMULATION_STEPS) * N
```

In [ ]:
```
# Load pre-trained model tokenizer (vocabulary)
print('Load pre-trained model tokenizer (vocabulary)')
tokenizer = BertTokenizer.from_pretrained('bert-base-uncased')
```

In [ ]:
```
print('train_examples_for_processing')
label_map = {label: i for i, label in enumerate(label_list)}
train_examples_for_processing = [(example, label_map, MAX_SEQ_LENGTH, tokeni
```

In [ ]:
```
process_count = cpu_count() - 1
if __name__ ==  '__main__':
    print(f'Preparing to convert {train_examples_len} examples..')
    print(f'Spawning {process_count} processes..')
    with Pool(process_count) as p:
        train_features = list(tqdm_notebook(p.imap(convert_examples_to_featu
```

In [ ]:
```
with open(DATA_DIR + "train_features.pkl", "wb") as f:
    pickle.dump(train_features, f)
```

In [ ]:
```
print('Fine tuning bert')
# Load pre-trained model (weights)
model = BertForSequenceClassification.from_pretrained(BERT_MODEL, cache_dir=
# model = BertForSequenceClassification.from_pretrained(CACHE_DIR + 'uncased
```

In [ ]:
```
model.to(device)
```

In [ ]:
```
param_optimizer = list(model.named_parameters())
no_decay = ['bias', 'LayerNorm.bias', 'LayerNorm.weight']
optimizer_grouped_parameters = [
    {'params': [p for n, p in param_optimizer if not any(nd in n for nd in n
    {'params': [p for n, p in param_optimizer if any(nd in n for nd in no_de
    ]
```

In [ ]:
```
optimizer = BertAdam(optimizer_grouped_parameters,
                     lr=LEARNING_RATE,
                     warmup=WARMUP_PROPORTION,
                     t_total=num_train_optimization_steps)
```

In [ ]:
```
global_step = 0
nb_tr_steps = 0
tr_loss = 0
```

In [ ]:
```
logger.info("***** Running training *****")
logger.info("  Num examples = %d", train_examples_len)
logger.info("  Batch size = %d", TRAIN_BATCH_SIZE)
logger.info("  Num steps = %d", num_train_optimization_steps)
all_input_ids = torch.tensor([f.input_ids for f in train_features], dtype=to
all_input_mask = torch.tensor([f.input_mask for f in train_features], dtype=
all_segment_ids = torch.tensor([f.segment_ids for f in train_features], dtyp

if OUTPUT_MODE == "classification":
    all_label_ids = torch.tensor([f.label_id for f in train_features], dtype
elif OUTPUT_MODE == "regression":
    all_label_ids = torch.tensor([f.label_id for f in train_features], dtype
```

In [ ]:
```
print('Setting up our DataLoader for training..')
train_data = TensorDataset(all_input_ids, all_input_mask, all_segment_ids, a
train_sampler = RandomSampler(train_data)
train_dataloader = DataLoader(train_data, sampler=train_sampler, batch_size=
```

In [ ]:
```
print('Model.train!!!!')
model.train()
for _ in trange(int(NUM_TRAIN_EPOCHS), desc="Epoch"):
    tr_loss = 0
    nb_tr_examples, nb_tr_steps = 0, 0
    for step, batch in enumerate(tqdm_notebook(train_dataloader, desc="Itera
        batch = tuple(t.to(device) for t in batch)
        input_ids, input_mask, segment_ids, label_ids = batch

        logits = model(input_ids, segment_ids, input_mask, labels=None)

        if OUTPUT_MODE == "classification":
            loss_fct = CrossEntropyLoss()
            loss = loss_fct(logits.view(-1, num_labels), label_ids.view(-1))
        elif OUTPUT_MODE == "regression":
            loss_fct = MSELoss()
            loss = loss_fct(logits.view(-1), label_ids.view(-1))

        if GRADIENT_ACCUMULATION_STEPS > 1:
            loss = loss / GRADIENT_ACCUMULATION_STEPS

        loss.backward()
        print("\r%f" % loss, end='')

        tr_loss += loss.item()
        nb_tr_examples += input_ids.size(0)
        nb_tr_steps += 1
        if (step + 1) % GRADIENT_ACCUMULATION_STEPS == 0:
            optimizer.step()
            optimizer.zero_grad()
            global_step += 1
```

In [ ]:
```
print('model_to_save')
model_to_save = model.module if hasattr(model, 'module') else model  # Only

# If we save using the predefined names, we can load using `from_pretrained`
output_model_file = os.path.join(OUTPUT_DIR, WEIGHTS_NAME)
output_config_file = os.path.join(OUTPUT_DIR, CONFIG_NAME)

torch.save(model_to_save.state_dict(), output_model_file)
model_to_save.config.to_json_file(output_config_file)
tokenizer.save_vocabulary(OUTPUT_DIR)
```

In [ ]:
```
print('EVALUATION----------------------------------')
```

In [ ]:

```
import torch
import numpy as np
import pickle

from sklearn.metrics import matthews_corrcoef, confusion_matrix

from torch.utils.data import (DataLoader, RandomSampler, SequentialSampler,
                              TensorDataset)
from torch.utils.data.distributed import DistributedSampler
from torch.nn import CrossEntropyLoss, MSELoss

from tools import *
from multiprocessing import Pool, cpu_count
import convert_examples_to_features

from tqdm import tqdm_notebook, trange
import os
from pytorch_pretrained_bert import BertTokenizer, BertModel, BertForMaskedL
from pytorch_pretrained_bert.optimization import BertAdam, WarmupLinearSched

# OPTIONAL: if you want to have more information on what's happening, activa
import logging
logging.basicConfig(level=logging.INFO)

device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
```

In [ ]:

```
# The input data dir. Should contain the .tsv files (or other data files) fo
DATA_DIR = "data/"

# Bert pre-trained model selected in the list: bert-base-uncased,
# bert-large-uncased, bert-base-cased, bert-large-cased, bert-base-multiling
# bert-base-multilingual-cased, bert-base-chinese.
BERT_MODEL = 'yelp.tar.gz'

# The name of the task to train.I'm going to name this 'yelp'.
TASK_NAME = 'yelp'

# The output directory where the fine-tuned model and checkpoints will be wr
OUTPUT_DIR = f'outputs/{TASK_NAME}/'

# The directory where the evaluation reports will be written to.
REPORTS_DIR = f'reports/{TASK_NAME}_evaluation_reports/'

# This is where BERT will look for pre-trained models to load parameters fro
CACHE_DIR = 'cache/'

# The maximum total input sequence length after WordPiece tokenization.
# Sequences longer than this will be truncated, and sequences shorter than t
MAX_SEQ_LENGTH = 128

TRAIN_BATCH_SIZE = 24
EVAL_BATCH_SIZE = 8
LEARNING_RATE = 2e-5
NUM_TRAIN_EPOCHS = 1
RANDOM_SEED = 42
GRADIENT_ACCUMULATION_STEPS = 1
WARMUP_PROPORTION = 0.1
OUTPUT_MODE = 'classification'

CONFIG_NAME = "config.json"
WEIGHTS_NAME = "pytorch_model.bin"
```

In [ ]:

```
if os.path.exists(REPORTS_DIR) and os.listdir(REPORTS_DIR):
        REPORTS_DIR += f'/report_{len(os.listdir(REPORTS_DIR))}'
        os.makedirs(REPORTS_DIR)
if not os.path.exists(REPORTS_DIR):
    os.makedirs(REPORTS_DIR)
    REPORTS_DIR += f'/report_{len(os.listdir(REPORTS_DIR))}'
    os.makedirs(REPORTS_DIR)
```

In [ ]:
```
def get_eval_report(task_name, labels, preds):
    mcc = matthews_corrcoef(labels, preds)
    tn, fp, fn, tp = confusion_matrix(labels, preds).ravel()
    return {
        "task": task_name,
        "mcc": mcc,
        "tp": tp,
        "tn": tn,
        "fp": fp,
        "fn": fn
    }

def compute_metrics(task_name, labels, preds):
    assert len(preds) == len(labels)
    return get_eval_report(task_name, labels, preds)
```

In [ ]:
```
# Load pre-trained model tokenizer (vocabulary)
tokenizer = BertTokenizer.from_pretrained(OUTPUT_DIR + 'vocab.txt', do_lower
```

In [ ]:
```
processor = BinaryClassificationProcessor()
eval_examples = processor.get_dev_examples(DATA_DIR)
label_list = processor.get_labels() # [0, 1] for binary classification
num_labels = len(label_list)
eval_examples_len = len(eval_examples)
```

In [ ]:
```
label_map = {label: i for i, label in enumerate(label_list)}
eval_examples_for_processing = [(example, label_map, MAX_SEQ_LENGTH, tokeniz
```

In [ ]:
```
process_count = cpu_count() - 1
if __name__ ==  '__main__':
    print(f'Preparing to convert {eval_examples_len} examples..')
    print(f'Spawning {process_count} processes..')
    with Pool(process_count) as p:
        eval_features = list(tqdm_notebook(p.imap(convert_examples_to_featur
```

In [ ]:
```
all_input_ids = torch.tensor([f.input_ids for f in eval_features], dtype=tor
all_input_mask = torch.tensor([f.input_mask for f in eval_features], dtype=t
all_segment_ids = torch.tensor([f.segment_ids for f in eval_features], dtype
```

In [ ]:
```
if OUTPUT_MODE == "classification":
    all_label_ids = torch.tensor([f.label_id for f in eval_features], dtype=
elif OUTPUT_MODE == "regression":
    all_label_ids = torch.tensor([f.label_id for f in eval_features], dtype=
```

In [ ]:
```
eval_data = TensorDataset(all_input_ids, all_input_mask, all_segment_ids, al

# Run prediction for full data
eval_sampler = SequentialSampler(eval_data)
eval_dataloader = DataLoader(eval_data, sampler=eval_sampler, batch_size=EVA
```

In [ ]:
```
# Load pre-trained model (weights)
print('Load pre-trained model (weights)')
model = BertForSequenceClassificationion.from_pretrained(CACHE_DIR + BERT_MO
```

In [ ]:
```
model.to(device)
```

In [ ]:

```
model.eval()
eval_loss = 0
nb_eval_steps = 0
preds = []

for input_ids, input_mask, segment_ids, label_ids in tqdm_notebook(eval_data
    input_ids = input_ids.to(device)
    input_mask = input_mask.to(device)
    segment_ids = segment_ids.to(device)
    label_ids = label_ids.to(device)

    with torch.no_grad():
        logits = model(input_ids, segment_ids, input_mask, labels=None)

    # create eval loss and other metric required by the task
    if OUTPUT_MODE == "classification":
        loss_fct = CrossEntropyLoss()
        tmp_eval_loss = loss_fct(logits.view(-1, num_labels), label_ids.view
    elif OUTPUT_MODE == "regression":
        loss_fct = MSELoss()
        tmp_eval_loss = loss_fct(logits.view(-1), label_ids.view(-1))

    eval_loss += tmp_eval_loss.mean().item()
    nb_eval_steps += 1
    if len(preds) == 0:
        preds.append(logits.detach().cpu().numpy())
    else:
        preds[0] = np.append(
            preds[0], logits.detach().cpu().numpy(), axis=0)

eval_loss = eval_loss / nb_eval_steps
preds = preds[0]
if OUTPUT_MODE == "classification":
    preds = np.argmax(preds, axis=1)
elif OUTPUT_MODE == "regression":
    preds = np.squeeze(preds)
result = compute_metrics(TASK_NAME, all_label_ids.numpy(), preds)

result['eval_loss'] = eval_loss

output_eval_file = os.path.join(REPORTS_DIR, "eval_results.txt")
with open(output_eval_file, "w") as writer:
    logger.info("***** Eval results *****")
    for key in (result.keys()):
        logger.info("  %s = %s", key, str(result[key]))
        writer.write("%s = %s\n" % (key, str(result[key])))
```