Natural Language Processing

Lab 1 Text Processing Tools

Objective

This lab is to introduce ***Python*** tools/libraries which can assist developer to implement NLP models or deal with NLP tasks (indexing, similarity calculation, retrieval, ranking and etc). These tools include NLTK, Gensim, and CoreNLP. Then some basic methods to process data, and train models (seq2seq structure and BERT) will be introduced. After this lab, students are expected to know how to implement operations mentioned in Lecture1~3 (mainly about text normalization and classification). The lab will be demonstrated in Jupyter notebook (Lab PC: start → anaconda navigator (anaconda3); download instruction: https://jupyter.org/install).

NLTK

Natural language toolkit (NLTK) is the most popular library for natural language processing (NLP) which was written in Python and has a big community behind it. NLTK is quite easy to learn, and it is also the easiest NLP library that we are going to use. It contains text processing libraries for tokenization, parsing, classification, stemming, tagging and semantic reasoning. Besides NLTK, some other third-party libraries:

- spaCy: a toolkit similar to NLTK; the main competitor of NLTK.
- polyglot: an uncommon toolkit, which is similar to NLTK. It can support massive multilingual applications.
- scikit-learn: a module for machine learning.
- Gensim: a library for topic modelling, document indexing and similarity retrieval (mentioned in the following parts).

Installation

Downloading Libs and Testing That They Are Working
https://github.com/hb20007/hands-on-nltk-tutorial

We need to download NLTK before proceeding:

```
>>>pip install nltk

>>>python
import nltk
nltk.download()
```

Tokenization / Segmentation

To classify and analyze a body of text in a more granular fashion, it is necessary to consider how to chop it up into pieces (individual sentences, words or "tokens"). For instance:

Input: Friends, Romans, Countrymen, lend me your ears;
Output: | Friends | Romans | Countrymen | lend | me | your | ears |

Broadly there are two tasks: **Sentence Tokenization**, and **Word Tokenization**.

Sentence Tokenizer

The default Sentence Tokenizer is the PunktSentenceTokenizer from the nltk.tokenize.punkt module.

In the example below (taken from James Joyce's Ulysses), we load the NLTK library and process a block of text.

```
import nltk

ulysses = "Mrkgnao! the cat said loudly. She blinked up out of her avid shameclosing eyes, mewing \
plaintively and long, showing him her milkwhite teeth. He watched the dark eyeslits narrowing \
with greed till her eyes were green stones. Then he went to the dresser, took the jug Hanlon's\
milkman had just filled for him, poured warmbubbled milk on a saucer and set it slowly on the floor.\
— Gurrhr! she cried, running to lap."

doc = nltk.sent_tokenize(ulysses)
for s in doc:
    print(">",s)
```

Even in this very simple example, we can see that the results are not always perfect depending on the style of the text and variations from the style of the Corpus that was used to develop the tokenization approach.

Word Tokenization

Input: Friends, Romans, Countrymen, lend me your ears;
Output: | Friends | | Romans | | Countrymen | | lend | | me | | your | | ears |

There are many methods for tokenizing text into words. The default Penn Treebank Tokenizer is the tokenizer based on the Penn TreeBank Corpus. A few examples of different tokenizers giving different results are listed below:

1. TreebankWordTokenizer
2. WordPunctTokenizer
3. WhitespaceTokenizer

We can see a simple illustration of the impact of choosing a different tokenization method by looking at the different results we get for a simple sentence:

```
from nltk import word_tokenize

sentence = "Mary had a little lamb it's fleece was white as snow."
# Default Tokenization
tree_tokens = word_tokenize(sentence)   # nltk.download('punkt') for this

# Other Tokenizers
punct_tokenizer = nltk.tokenize.WordPunctTokenizer()
punct_tokens = punct_tokenizer.tokenize(sentence)

print("DEFAULT: ", tree_tokens)
print("PUNCT  : ", punct_tokens)
```

Other Tokenization:

- BERT tokenization

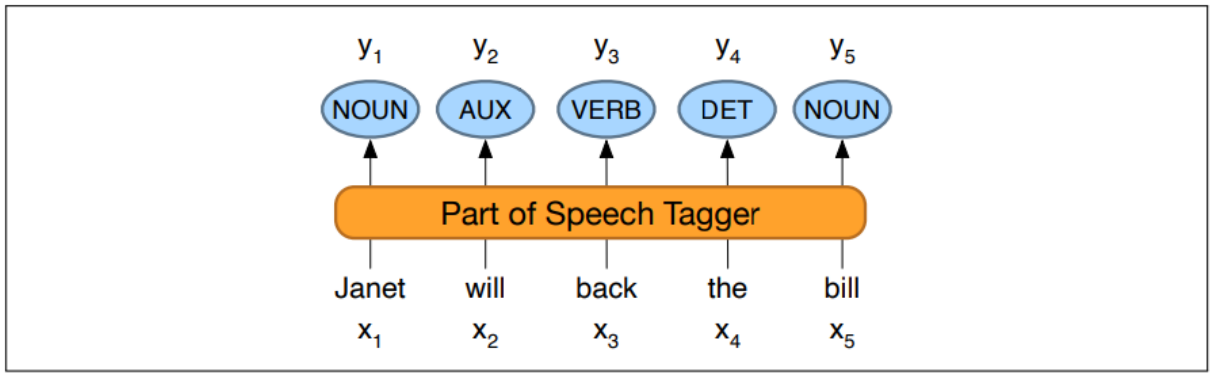
Input	Welcome to LAB1!
Output	Welcome to LA ##B ##1 !

- Character-based tokenization

Input	Welcome to LAB1!
Output	W e l c o m e t o L A B 1 !

Part of Speech Tagging

To go beyond counting the frequency or occurrence of actual words, we need to classify words in general categories that signify their parts in the construct of the sentence - for instance Noun, Verb Adjective etc. This is generally known as: Part of Speech or POS Tagging



For each word-token the nltk pos_tag method can be used to classify its Part of Speech (POS), automating the classification of words into their parts of speech and labeling them accordingly.

The outcome depends on how the sentence has been split up into individual tokens and which Tokenizer and Corpus the POS-tagger has been trained against:

```
pos = nltk.pos_tag(tree_tokens)
print(pos)
pos_punct = nltk.pos_tag(punct_tokens)
print(pos_punct)
```

PoS Tag Descriptions

Tag	**Description**	**Tag**	**Description**
CC	Coordinating conjunction	RB	Adverb
CD	Cardinal number	RBR	Adverb, comparative
DT	Determiner	RBS	Adverb, superlative
EX	Existential there	RP	Particle
FW	Foreign word	SYM	Symbol
IN	Preposition or subordinating conjunction	TO	to
JJ	Adjective	UH	Interjection
JJR	Adjective, comparative	VB	Verb, base form
JJS	Adjective, superlative	VBD	Verb, past tense
LS	List item marker	VBG	Verb, gerund or present participle
MD	Modal	VBN	Verb, past participle
NN	Noun, singular or mass	VBP	Verb, non-3rd person singular present
NNS	Noun, plural	VBZ	Verb, 3rd person singular present
NNP	Proper noun, singular	WDT	Wh-determiner
NNPS	Proper noun, plural	WP	Wh-pronoun
PDT	Predeterminer	WP$	Possessive wh-pronoun
POS	Possessive ending	WRB	Wh-adverb
PRP	Personal pronoun		
PRP\$	Possessive pronoun		

The naming convention of the PoS tags makes it easy to use regular expressions to extract classes of word-type (e.g. all the Nouns):

```
import re
regex = re.compile("^N.*")
nouns = []
for l in pos:
    if regex.match(l[1]):
        nouns.append(l[0])
print("Nouns:", nouns)
```

Stemming and Lemmatization

Striping off the suffixes from words is known as stemming.
Mapping a word to a known dictionary word is known as lemmatization

There are multiple Stemming methods available and the NLTK book references a few methods in particular:

1. The Porter Stemmer - see https://tartarus.org/martin/PorterStemmer/
2. Lancaster Stemmer - (Chris Paice, University of Lancaster) additionally the
3. Snowball Stemmer - "Porter 2" developed by Martin Porter is generally considered the de-facto optimal Stemmer

A list of other stemming methods can be found here: http://www.nltk.org/api/nltk.stem.html. Current Stemming and "Lemming" techniques are an inexact process as things currently stand.

Stemming Example

```
porter = nltk.PorterStemmer()
lancaster = nltk.LancasterStemmer()
snowball = nltk.stem.snowball.SnowballStemmer("english")

print([porter.stem(t) for t in tree_tokens])
print([lancaster.stem(t) for t in tree_tokens])
print([snowball.stem(t) for t in tree_tokens])

sentence2 = "When I was going into the woods I saw a bear lying asleep on the forest floor"
tokens2 = word_tokenize(sentence2)

print("\n",sentence2)
for stemmer in [porter, lancaster, snowball]:
    print([stemmer.stem(t) for t in tokens2])
```

Lemmatizing Example

Lemmatization aims to achieve a similar base "stem" for a word, but aims to derive the genuine dictionary root word, not just a truncated version of the word.

The default lemmatization method with the Python NLTK is the WordNet lemmatizer.

```
# if nltk.download() is not run before, then we should run the following two lines
# nltk.download("averaged_perceptron_tagger")
# nltk.download("wordnet")
wnl = nltk.WordNetLemmatizer()
tokens2_pos = nltk.pos_tag(tokens2)  #nltk.download("averaged_perceptron_tagger")
```

```
print([wnl.lemmatize(t) for t in tree_tokens])
print([wnl.lemmatize(t) for t in tokens2])
```

Vectorization, and Document Similarity Calculation

Gensim is billed as a Natural Language Processing package that does 'Topic Modeling for Humans'. But it is practically much more than that. It is a leading and a state-of-the-art package for processing texts, working with word vector models (such as Word2Vec, FastText and etc.).

Topic models and word embedding are available in other packages like scikit, R etc. But the width and scope of facilities to build and evaluate topic models are unparalleled in gensim, plus many more convenient facilities for text processing. Another important benefit with gensim is that it allows you to manage big text files without loading the whole file into memory.
Install gensim by the following command:

```
pip install gensim
```

Practice

We have three documents in demofile.txt:

```
Mars is the fourth planet in our solar system.
It is second-smallest planet in the Solar System after Mercury.
Saturn is yellow planet.
```

And we have a query in demofiles.txt.

```
Mars is approximately half the diameter of Earth.
```

Tasks: compute the similarities between the query and each document in the demofile.txt.

Now, we are going to process the documents.

```
import nltk
from nltk.tokenize import word_tokenize, sent_tokenize

gen_docs = []

with open ('demofile.txt') as f:
    docs=f.readlines()
    for doc in docs:
        gen_docs.append([w.lower() for w in word_tokenize(doc)])

print(gen_docs)
```

In order to work on text documents, Gensim requires the words (aka tokens) be converted to unique ids. So, Gensim lets you create a Dictionary object that maps each word to a unique id. Let's convert our sentences to a [list of words] and pass it to the corpora.Dictionary() object.

```
dictionary = gensim.corpora.Dictionary(gen_docs)
print(dictionary.token2id)
```

A dictionary maps every word to a number. Gensim lets you read the text and update the dictionary, one line at a time, without loading the entire text file into system memory.

The next important object you need to familiarize with in order to work in gensim is the Corpus (a Bag of Words). It is a basically object that contains the word id and its frequency in each document (just lists the number of times each word occurs in the sentence).

Note that, a ‘token’ typically means a ‘word’. A ‘document’ can typically refer to a ‘sentence’ or ‘paragraph’ and a ‘corpus’ is typically a ‘collection of documents as a bag of words’.

Now, create a bag of words corpus and pass the tokenized list of words to the Dictionary.doc2bow()

Let's assume that our documents are (demofile2.txt):

```
Mars is a cold desert world. It is half the size of the Earth.
```

```
corpus = [dictionary.doc2bow(gen_doc) for gen_doc in gen_docs]
```

Then we calculate TF-IDF (gensim) as:

```
import numpy as np
tf_idf = gensim.models.TfidfModel(corpus)
for doc in tf_idf[corpus]:
   print([[dictionary[id], np.around(freq, decimals=2)] for id, freq in doc])
```

Now, we are going to create similarity object. The main class is Similarity, which builds an index for a given set of documents. The Similarity class splits the index into several smaller sub-indexes, which are disk-based. Let's just create similarity object then you will understand how we can use it for comparing.

```
# building the index
 sims = gensim.similarities.Similarity('yourworkdir/',tf_idf[corpus],
                                        num_features=len(dictionary))
```

We are storing index matrix in 'yourworkdir' directory but you can name it whatever you want and of course you have to create it with same directory of your program.

Once the index is built, we are going to calculate how similar is this query document to each document in the index. So, create second .txt file which will include query documents or sentences and tokenize them as we did before.

```
with open ('demofile2.txt') as f:
    query=f.readlines()[0]
    query_doc=[w.lower() for w in word_tokenize(query)]

print('query_doc',query_doc)
query_doc_bow = dictionary.doc2bow(query_doc)
```

We get new documents (query documents or sentences) so it is possible to update an existing dictionary to include the new words.

At this stage, you will see similarities between the query and all index documents. To obtain similarities of our query document against the indexed documents:

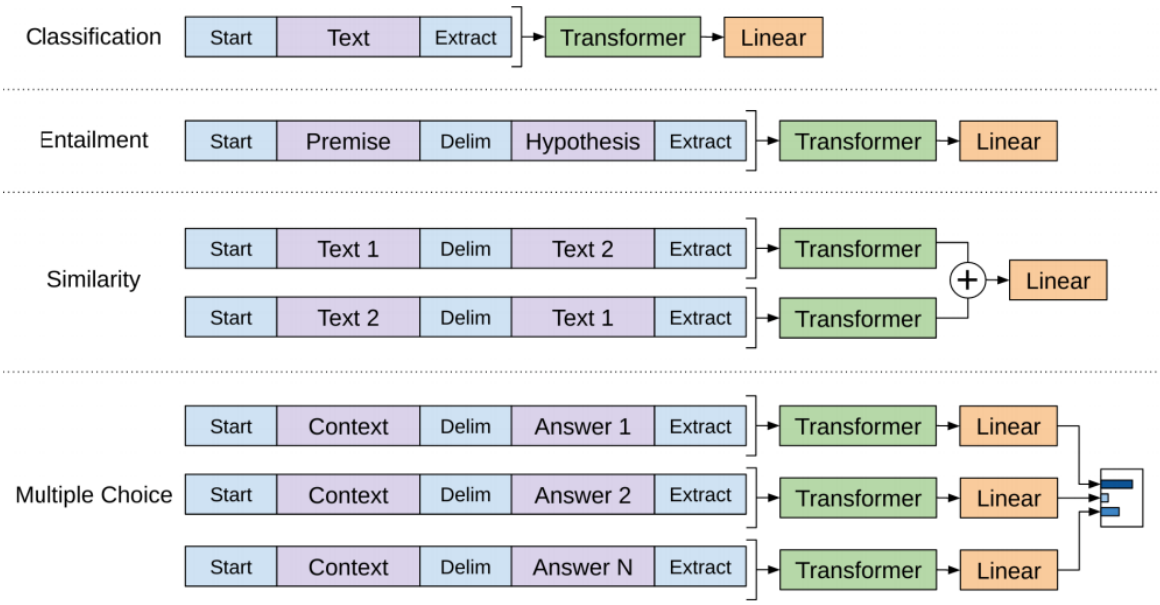
```
# perform a similarity query against the corpus
query_doc_tf_idf = tf_idf[query_doc_bow]
```

```
# print(document_number, document_similarity)
print('Comparing Result:', sims[query_doc_tf_idf])
```

Cosine measure returns similarities in the range (the greater, the more similar).

BERT

BERT is a language representation model released in 2018.
More details related to BERT:
Paper: *BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding*
Quick to learn Blog: https://jalammar.github.io/illustrated-bert/
Model implementation: https://github.com/google-research/bert
Related works and models: transformers, ELMo



Installation

Many third-party packages provide implementation of BERT, and this lab displays how to use BERT with the implementation provided by Hugging Face, i.e., transformers.

1. Installation with pip

```
# PyTorch has been installed
pip install transformers
# install transformers and PyTorch in one line:
pip install transformers[torch]
# to check whether transformers is properly installed, run the following command in cmd:
python -c "from transformers import pipeline; print(pipeline('sentiment-analysis')('we love you'))"
# if the installation is ready, you will get:
# [{'label': 'POSITIVE', 'score': 0.9998704791069031}]
```

2. Installation with source:

```
pip install git+https://github.com/huggingface/transformers
```

We need to make 3 parts ready when using BERT. They are *tokenization, configuration,* and *model*.

Tokenization

We use BertTokenizer to tokenize sentences.

```
import torch
from transformers import BertTokenizer
tokenizer = BertTokenizer.from_pretrained('bert-base-uncased')
```

Let have a look at the vocabulary.

```
vocab = tokenizer.get_vocab()
print(vocab)
```

Special tokens in BERT play an important role in BERT. There are mainly 3 special tokens: `[CLS]`, `[SEP]`, and `[PAD]`, respectively.

`[CLS]`: class of the sentence.

`[SEP]`: the end of the last sentence, and the beginning of the second sentence.

`[PAD]`: padding.

```
print(vocab["[CLS]"], vocab["[SEP]"], vocab["[PAD]"])
```

Tokenize a sentence and convert tokens into ids:

```
sent1 = "[CLS] He remains characteristically confident and optimistic."
sent2 = "He remains characteristically confident and optimistic. \
        [SEP] Therefore, I believe he will finally achieve success. [PAD]"
encoding1 = tokenizer.tokenize(sent1)
encoding2 = tokenizer.tokenize(sent2)
print(encoding1)
print(encoding2)
sent_ids1 = tokenizer.convert_tokens_to_ids(encoding1)
sent_ids2 = tokenizer.convert_tokens_to_ids(encoding2)
print(sent_ids1)
print(sent_ids2)
```

Tokenize multi sentences by encode_plus:

```
sent_encoding = tokenizer.encode_plus(sent1, sent2)
print(sent_encoding)
```

Configuration & Model

BertConfig can decide the architecture of BERT:

```
from transformers import BertModel, BertConfig
configuration = BertConfig()
model = BertModel(configuration)
configuration = model.config
print(configuration)
```

Use BERT to encode a sentence:

```
model = BertModel.from_pretrained('bert-base-uncased')
inputs = tokenizer("He remains characteristically confident and optimistic.",
                   return_tensors="pt")
print('Inputs:', inputs)
outputs = model(**inputs)
last_hidden_states = outputs[0]
print('Last hidden states:', last_hidden_states)
print('Last hidden states size:',last_hidden_states.size())
```

```
def forward(
    self,
    input_ids=None,
    attention_mask=None,
    token_type_ids=None,
    position_ids=None,
    head_mask=None,
    inputs_embeds=None,
    encoder_hidden_states=None,
    encoder_attention_mask=None,
    past_key_values=None,
    use_cache=None,
    output_attentions=None,
    output_hidden_states=None,
    return_dict=None,
):
```

input_ids or input_embeds is required, others are optional

input_ids: sequence of token indexes

attention_mask (encoder_attention_mask): 0 (masked) or 1 (not masked)

token_type_ids: 0 (tokens belonging to the first sentence) or 1 (… second sentence)

position_ids: the order position of id

head_mask: 0 (invalid attention head) or 1 (valid attention head)

inputs_embeds:
encoder_hidden_states, output_attentions, output_hidden_states: usually used in the decoding stage of the sequence-to-sequence model.

Sentence Classification

Read data with pandas as DataFrame

```
import pandas as pd
train_df = pd.read_csv('BBC News Train.csv', header=0)
print('BBC News Training DataFrame:\n', train_df)
```

Prepare the labels:

```
label_text = train_df.Category.tolist()[:50]
label_type = set(label_text)
label2id = {label: id for id, label in enumerate(label_type)}
labels = [label2id[label] for label in label_text]
labels = torch.tensor(labels)
print('Labels:', labels)
```

Convert sentences into token indexes

```
input_ids, attention_masks = [], []
for sent in train_df.Text[:50]:
    encoded_dict = tokenizer.encode_plus(
                        ' '.join(sent.split()[:100]),    # Sentence to encode.
                        add_special_tokens = True, # Add '[CLS]' and '[SEP]'
                        max_length = 102,           # Pad & truncate all sentences.
                        pad_to_max_length = True,
                        return_attention_mask = True,   # Construct attn. masks.
                        return_tensors = 'pt',     # Return pytorch tensors.
                   )
    input_ids.append(encoded_dict['input_ids'])
    attention_masks.append(encoded_dict['attention_mask'])
input_ids = torch.cat(input_ids, dim=0)
attention_masks = torch.cat(attention_masks, dim=0)
print('Original: ', ' '.join(train_df.Text[0].split()[:60]))
print('Token ids:', input_ids[0])
```

Split the dataset:

```
from torch.utils.data import TensorDataset, random_split
dataset = TensorDataset(input_ids, attention_masks, labels)
train_size = int(0.9 * len(dataset))
val_size = len(dataset) - train_size
train_dataset, val_dataset = random_split(dataset, [train_size, val_size])

print('{:>5,} training samples'.format(train_size))
print('{:>5,} validation samples'.format(val_size))
```

Prepare the dataloader:

```
from torch.utils.data import DataLoader, RandomSampler, SequentialSampler
batch_size = 4
train_dataloader = DataLoader(
            train_dataset,  # The training samples.
            sampler = RandomSampler(train_dataset), # Select batches randomly
            batch_size = batch_size # Trains with this batch size.
        )
validation_dataloader = DataLoader(
            val_dataset, # The validation samples.
            sampler = SequentialSampler(val_dataset), # Pull out batches sequentially.
            batch_size = batch_size # Evaluate with this batch size.
        )
```

Prepare the model:

```
from transformers import BertForSequenceClassification
cls_model = BertForSequenceClassification.from_pretrained(
    "bert-base-uncased", # Use the 12-layer BERT model, with an uncased vocab.
    num_labels = len(label_type), # The number of output labels.
    output_attentions = False, # Whether the model returns attentions weights.
    output_hidden_states = False, # Whether the model returns all hidden-states.
)
cls_model.cuda()
params = list(cls_model.named_parameters())
```

```
print('The BERT model has {:} different named parameters.\n'.format(len(params)))
print('==== Embedding Layer ====\n')
for p in params[0:5]:
    print("{:<55} {:>12}".format(p[0], str(tuple(p[1].size()))))
print('\n==== First Transformer ====\n')
for p in params[5:21]:
    print("{:<55} {:>12}".format(p[0], str(tuple(p[1].size()))))
print('\n==== Output Layer ====\n')
for p in params[-4:]:
    print("{:<55} {:>12}".format(p[0], str(tuple(p[1].size()))))
```

Prepare for the optimization:

```
from transformers import AdamW
optimizer = AdamW(cls_model.parameters(),
                  lr = 2e-5, # args.learning_rate - default is 5e-5, our notebook had 2e-5
                  eps = 1e-8 # args.adam_epsilon  - default is 1e-8.
                )
from transformers import get_linear_schedule_with_warmup
epochs = 4
total_steps = len(train_dataloader) * epochs
scheduler = get_linear_schedule_with_warmup(optimizer,
                                            num_warmup_steps = 0, # Default value in run_glue.py
                                            num_training_steps = total_steps)
```

Prepare a help function for accuracy computation:

```
import numpy as np

# Function to calculate the accuracy of our predictions vs labels
def flat_accuracy(preds, labels):
    pred_flat = np.argmax(preds, axis=1).flatten()
    labels_flat = labels.flatten()
    return np.sum(pred_flat == labels_flat) / len(labels_flat)
```

Prepare a help function for elapsed time computation:

```
import time
import datetime

def format_time(elapsed):
    '''
    Takes a time in seconds and returns a string hh:mm:ss
    '''
    # Round to the nearest second.
    elapsed_rounded = int(round((elapsed)))

    # Format as hh:mm:ss
    return str(datetime.timedelta(seconds=elapsed_rounded))
```

Let train the model!

```
import random
import numpy as np
from tqdm import tqdm
import os
os.environ['CUDA_LAUNCH_BLOCKING'] = "1"

# This training code is based on the `run_glue.py` script here:
#
https://github.com/huggingface/transformers/blob/5bfcd0485ece086ebcbed2d008813037968a9e58/examples/run
_glue.py#L128

# Set the seed value all over the place to make this reproducible.
seed_val = 42

random.seed(seed_val)
np.random.seed(seed_val)
torch.manual_seed(seed_val)
torch.cuda.manual_seed_all(seed_val)

# We'll store a number of quantities such as training and validation loss,
# validation accuracy, and timings.
training_stats = []

# Measure the total training time for the whole run.
```

```
total_t0 = time.time()

# For each epoch...
for epoch_i in range(0, epochs):

    # ========================================
    #               Training
    # ========================================

    # Perform one full pass over the training set.

    print("")
    print('======== Epoch {:} / {:} ========'.format(epoch_i + 1, epochs))
    print('Training...')

    # Measure how long the training epoch takes.
    t0 = time.time()

    # Reset the total loss for this epoch.
    total_train_loss = 0

    # Put the model into training mode. Don't be mislead--the call to
    # `train` just changes the *mode*, it doesn't *perform* the training.
    # `dropout` and `batchnorm` layers behave differently during training
    #  vs.  test  (source:  https://stackoverflow.com/questions/51433378/what-does-model-train-do-in-
pytorch)
    cls_model.train()

    # For each batch of training data...
    for step, batch in tqdm(enumerate(train_dataloader), total=len(train_dataloader)):
        # Progress update every 40 batches.
        if step % 4 == 0 and not step == 0:
            # Calculate elapsed time in minutes.
            elapsed = format_time(time.time() - t0)

            # Report progress.
            print('  Batch {:>5,}  of  {:>5,}.    Elapsed: {:}.'.format(step, len(train_dataloader),
elapsed))
        b_input_ids = batch[0].to('cuda')
        b_input_mask = batch[1].to('cuda')
        b_labels = batch[2].to('cuda')

        # Always clear any previously calculated gradients before performing a
        # backward pass. PyTorch doesn't do this automatically because
        # accumulating the gradients is "convenient while training RNNs".
        #  (source:  https://stackoverflow.com/questions/48001598/why-do-we-need-to-call-zero-grad-in-
pytorch)
        cls_model.zero_grad()
        outputs = cls_model(b_input_ids,
                            token_type_ids=None,
                            attention_mask=b_input_mask,
                            labels=b_labels)
        loss = outputs.loss
        logits = outputs.logits

        # Accumulate the training loss over all of the batches so that we can
        # calculate the average loss at the end. `loss` is a Tensor containing a
        # single value; the `.item()` function just returns the Python value
        # from the tensor.
        total_train_loss = total_train_loss + loss.item()

        # Perform a backward pass to calculate the gradients.
        loss.backward()

        # Clip the norm of the gradients to 1.0.
        # This is to help prevent the "exploding gradients" problem.
        torch.nn.utils.clip_grad_norm_(cls_model.parameters(), 1.0)

        # Update parameters and take a step using the computed gradient.
        # The optimizer dictates the "update rule"--how the parameters are
        # modified based on their gradients, the learning rate, etc.
```

```
        optimizer.step()

        # Update the learning rate.
        scheduler.step()
    # Calculate the average loss over all of the batches.
    avg_train_loss = total_train_loss / len(train_dataloader)

    # Measure how long this epoch took.
    training_time = format_time(time.time() - t0)

    print("")
    print("  Average training loss: {0:.2f}".format(avg_train_loss))
    print("  Training epcoh took: {:}".format(training_time))

    # ========================================
    #               Validation
    # ========================================
    # After the completion of each training epoch, measure our performance on
    # our validation set.

    print("")
    print("Running Validation...")


    t0 = time.time()

    # Put the model in evaluation mode--the dropout layers behave differently
    # during evaluation.
    cls_model.eval()

    # Tracking variables
    total_eval_accuracy = 0
    total_eval_loss = 0
    nb_eval_steps = 0

    # Evaluate data for one epoch
    for batch in validation_dataloader:

        # Unpack this training batch from our dataloader.
        #
        # As we unpack the batch, we'll also copy each tensor to the GPU using
        # the `to` method.
        #
        # `batch` contains three pytorch tensors:
        #   [0]: input ids
        #   [1]: attention masks
        #   [2]: labels
        b_input_ids = batch[0].to('cuda')
        b_input_mask = batch[1].to('cuda')
        b_labels = batch[2].to('cuda')

        # Tell pytorch not to bother with constructing the compute graph during
        # the forward pass, since this is only needed for backprop (training).
        with torch.no_grad():

            # Forward pass, calculate logit predictions.
            # token_type_ids is the same as the "segment ids", which
            # differentiates sentence 1 and 2 in 2-sentence tasks.
            # The documentation for this `model` function is here:
            #
https://huggingface.co/transformers/v2.2.0/model_doc/bert.html#transformers.BertForSequenceClassificat
ion
            # Get the "logits" output by the model. The "logits" are the output
            # values prior to applying an activation function like the softmax.

            outputs = cls_model(b_input_ids,
                            token_type_ids=None,
                            attention_mask=b_input_mask,
                            labels=b_labels
                           )
            loss, logits = outputs.loss, outputs.logits
```

```
            print(loss)
        # Accumulate the validation loss.
        total_eval_loss += loss.item()

        # Move logits and labels to CPU
        logits = logits.detach().cpu().numpy()
        label_ids = b_labels.to('cpu').numpy()

        # Calculate the accuracy for this batch of test sentences, and
        # accumulate it over all batches.
        total_eval_accuracy += flat_accuracy(logits, label_ids)


    # Report the final accuracy for this validation run.
    avg_val_accuracy = total_eval_accuracy / len(validation_dataloader)
    print("  Accuracy: {0:.2f}".format(avg_val_accuracy))

    # Calculate the average loss over all of the batches.
    avg_val_loss = total_eval_loss / len(validation_dataloader)

    # Measure how long the validation run took.
    validation_time = format_time(time.time() - t0)

    print("  Validation Loss: {0:.2f}".format(avg_val_loss))
    print("  Validation took: {:}".format(validation_time))

    # Record all statistics from this epoch.
    training_stats.append(
        {
            'epoch': epoch_i + 1,
            'Training Loss': avg_train_loss,
            'Valid. Loss': avg_val_loss,
            'Valid. Accur.': avg_val_accuracy,
            'Training Time': training_time,
            'Validation Time': validation_time
        }
    )

print("")
print("Training complete!")

print("Total training took {:} (h:mm:ss)".format(format_time(time.time()-total_t0)))
```

Multiple Choice

On stage, a woman takes a seat at the piano. She
a) sits on a bench as her sister plays with the doll.
b) smiles with someone as the music plays.
c) is in the crowd, watching the dancers.
d) nervously sets her fingers on the keys.

A girl is going across a set of monkey bars. She
a) jumps up across the monkey bars.
b) struggles onto the monkey bars to grab her head.
c) gets to the end and stands on a wooden plank.
d) jumps up and does a back flip.

The woman is now blow drying the dog. The dog
a) is placed in the kennel next to a woman's feet.
b) washes her face with the shampoo.
c) walks into frame and walks towards the dog.
d) tried to cut her face, so she is trying to do something very close to her face.

```
from transformers import BertForMultipleChoice
MC_model = BertForMultipleChoice.from_pretrained("bert-base-uncased")
```

```
prompt = "In Italy, pizza served in formal settings, such as at a restaurant, is presented unsliced."
choice0 = "It is eaten with a fork and a knife."
```

```
choice1 = "It is eaten while held in the hand."
labels = torch.tensor(0).unsqueeze(0)
encoding = tokenizer([prompt, prompt], [choice0, choice1], return_tensors="pt", padding=True)
outputs = MC_model(**{k: v.unsqueeze(0) for k, v in encoding.items()}, labels=labels)
loss = outputs.loss
logits = outputs.logits
print('Loss:', loss)
print('Logits:', logits)
```

Have a try use *SWAG MC Train.csv*.