Department of Computer Science and Engineering (Data Science)
Advanced Computational Linguistics
Experiment No 8

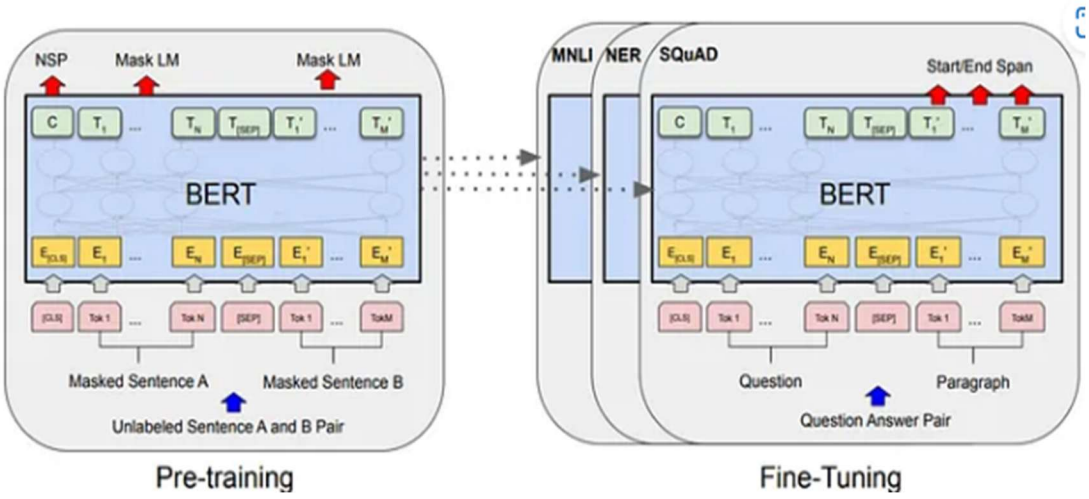
Name: Umang Kirit Lodaya **SAP ID: 60009200032**
Batch: D11

Aim: Fine tuning BERT model to perform Natural Language Processing task.

Theory:

BERT

BERT stands for **B**idirectional **E**ncoder **R**epresentations from **T**ransformers and is a language representation model by Google. It uses two steps, pre-training and fine-tuning, to create state-of-the-art models for a wide range of tasks. Its distinctive feature is the unified architecture across different downstream tasks — what these are, we will discuss soon. That means that the same pre-trained model can be fine-tuned for a variety of final tasks that might not be similar to the task model was trained on and give close to state-of-the-art results.



BERT Architecture

BERT has to differ Architecture BERT Base and BERT Large
BERT Base: L=12, H=768, A=12.
Total Parameters=110M!
BERT Large: L=24, H=1024, A=16.
Total Parameters=340M!!

L = Number of layers (i.e., #Transformer encoder blocks in the stack).
H = Hidden size (i.e. the size of *q, k* and *v* vectors).
A = Number of attention heads.

Pre-training BERT

The BERT model is trained on the following two unsupervised tasks.

1. Masked Language Model (MLM)

This task enables the deep bidirectional learning aspect of the model. In this task, some percentage of the input tokens are masked (Replaced with [*MASK*] token) at random and the model tries to predict these masked tokens — not the entire input sequence. The predicted tokens from the model are then fed into an output softmax over the vocabulary to get the final output words.

This, however creates a mismatch between the pre-training and fine-tuning tasks because the latter does not involve predicting masked words in most of the downstream tasks. This is mitigated by a subtle twist in how we mask the input tokens. Approximately 15% of the words are masked while training, but all of the masked words are not replaced by the [*MASK*] token.

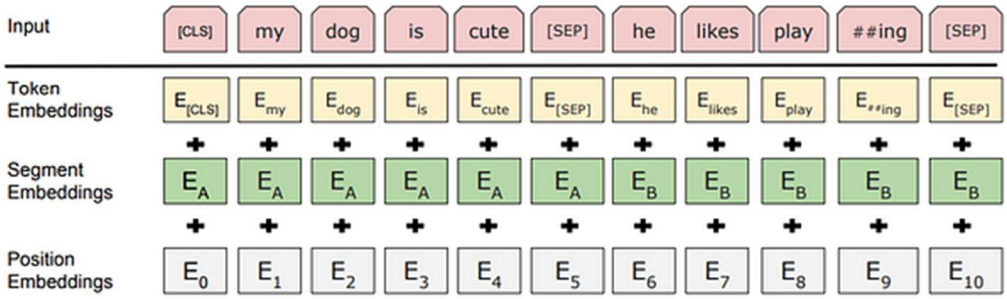
80% of the time with [*MASK*] tokens.
10% of the time with a random tokens.
10% of the time with the unchanged input tokens that were being masked.

2. Next Sentence Prediction (NSP)

The LM doesn't directly capture the relationship between two sentences which is relevant in many downstream tasks such as Question Answering (QA) and Natural Language Inference (NLI). The model is taught sentence relationships by training on binarized NSP task.

In this task, two sentences — A and B — are chosen for pre-training.
50% of the time B is the actual next sentence that follows A.
50% of the time B is a random sentence from the corpus.
Training — Inputs and Outputs.

The model is trained on both above mentioned tasks simultaneously. This is made possible by clever usage of inputs and outputs.
Inputs



The input representation for BERT

The model needs to take input for both a single sentence or two sentences packed together unambiguously in one token sequence. Authors note that a "sentence" can be an arbitrary span of contiguous text, rather than an actual linguistic sentence. A [SEP] token is used to separate two sentences as well as a using a learnt segment embedding indicating a token as a part of segment A or B.

Problem #1*:* All the inputs are fed in one step — as opposed to RNNs in which inputs are fed sequentially, the model is ***not able to preserve the ordering*** of the input tokens. The order of words in every language is significant, both semantically and syntactically.

Problem #2*:* In order to perform Next Sentence Prediction task properly we need to be able to ***distinguish between sentences A and B***. Fixing the lengths of sentences can be too restrictive and a potential bottleneck for various downstream tasks.
Both of these problems are solved by adding embeddings containing the required information to our original tokens and using the result as the input to our BERT model. The following embeddings are added to token embeddings:

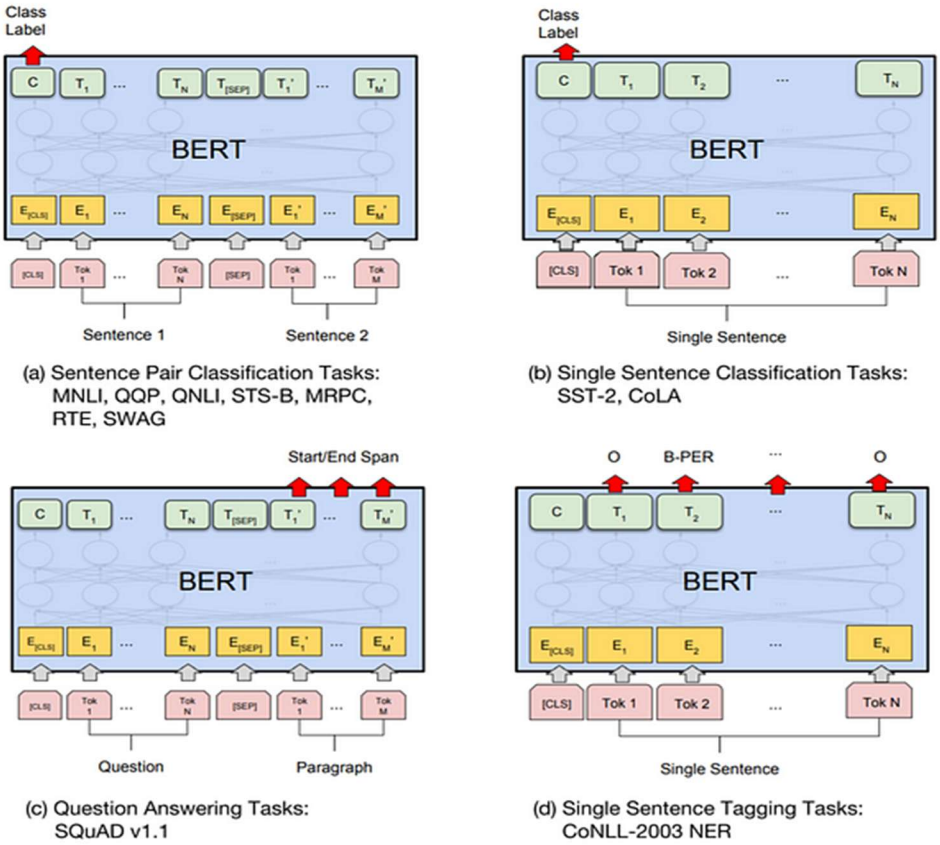
Segment Embedding: They provide information about the sentence a particular token is a part of.

Position Embedding: They provide information about the order of words in the input.
Outputs

Fine-tuning BERT

Fine-tuning on various downstream tasks is done by swapping out the appropriate inputs or outputs. In the general run of things, to train task-specific models, we add an extra output layer to existing BERT and fine-tune the resultant model — all parameters, end to end. A positive consequence of adding layers — input/output and not changing the BERT model is that only a minimal number of parameters need to be learned from scratch making the procedure fast, cost and resource efficient.

Just to give you an idea of how fast and efficient it is, the authors claim that all the results in the paper can be replicated in *at most 1 hour* on a *single Cloud TPU*, or *a few hours on a GPU*, starting from the exact same pre-trained model.



Fine-tuning BERT on various downstream tasks.

In Sentence Pair Classification and Single Sentence Classification, the final state corresponding to [*CLS*] token is used as input for the additional layers that makes the prediction.

In QA tasks, a start (S) and an end (E) vector are introduced during fine tuning. The question is fed as sentence A and the answer as sentence B. The probability of word *i* being the start of the answer span is computed as a dot product between T*i* (final state corresponding to *i*th input token) and S (start vector) followed by a softmax over all of the words in the paragraph. A similar method is used for end span.

Advantages of Fine-Tuning

Quicker Development

First, the pre-trained BERT model weights already encode a lot of information about our language. As a result, it takes much less time to train our fine-tuned model - it is as if we have already trained the bottom layers of our network extensively and only need to gently tune them while using their output as features for our classification task. In fact, the authors recommend only 2-4 epochs of training for fine-tuning BERT on a specific NLP task (compared to the hundreds of GPU hours needed to train the original BERT model or a LSTM from scratch!).

Less Data

In addition, and perhaps just as important, because of the pre-trained weights this method allows us to fine-tune our task on a much smaller dataset than would be required in a model that is built from scratch. A major drawback of NLP models built from scratch is that we often need a prohibitively large dataset in order to train our network to reasonable accuracy, meaning a lot of time and energy had to be put into dataset creation. By fine-tuning BERT, we are now able to get away with training a model to good performance on a much smaller amount of training data.

Better Results

Finally, this simple fine-tuning procedure (typically adding one fully-connected layer on top of BERT and training for a few epochs) was shown to achieve state of the art results with minimal task-specific adjustments for a wide variety of tasks: classification, language inference, semantic similarity, question answering, etc. Rather than implementing custom and sometimes-obscure architectures shown to work well on a specific task, simply fine-tuning BERT is shown to be a better (or at least equal) alternative.

Steps to Fine Tune BERT Model to perform Multi Class Text Classification

1. **Load dataset**
2. **Pre-process data**
3. **Define model**
4. **Train the model**
5. **Evaluate**

Lab Exercise to be Performed in this Session:
Perform Multi Label Text Classification by Fine tuning BERT model.

NAME: UMANG KIRIT LODAYA

SAP ID: 60009200032

BATCH: D11 ¶

In [ ]:
```
import tensorflow as tf
print(tf.version.VERSION)
```

In [ ]:
```
!git clone --depth 1 -b v2.4.0 https://github.com/tensorflow
odels.git
```

In [ ]:
```
# install requirements to use tensorflow/models repository
!pip install -Uqr models/official/requirements.txt
# you may have to restart the runtime afterwards, also ignore
any ERRORS popping up at this step
```

In [1]:
```
!pip install --upgrade -q wandb
```

In [2]:
```
from kaggle_secrets import UserSecretsClient
user_secrets = UserSecretsClient()
wandb_api = user_secrets.get_secret("wandb-api")
```

In [3]:
```
import wandb
from wandb.keras import WandbCallback
wandb.login(key=wandb_api)
```

```
wandb: W&B API key is configured (use `wandb login --relogin`
to force relogin)
wandb: WARNING If you're specifying your api key in code, ensu
re this code is not shared publically.
wandb: WARNING Consider setting the WANDB_API_KEY environment
variable, or running `wandb login` from the command line.
wandb: Appending key for api.wandb.ai to your netrc file: /roo
t/.netrc
```

Out[3]:
```
True
```

In [4]:
```
import numpy as np
import tensorflow as tf
import tensorflow_hub as hub
import sys
sys.path.append('models')
from official.nlp.data import classifier_data_lib
from official.nlp.bert import tokenization
from official.nlp import optimization
import matplotlib.pyplot as plt
%matplotlib inline
import seaborn as sns
sns.set()
```

In [5]:
```
print("TF Version: ", tf.__version__)
print("Eager mode: ", tf.executing_eagerly())
print("Hub version: ", hub.__version__)
print("GPU is", "available" if tf.config.experimental.list_physical_devices("GPU") else "NOT AVAILABLE")
```

```
TF Version:  2.4.1
Eager mode:  True
Hub version:  0.12.0
GPU is available
```

In [6]:
```
# TO LOAD DATA FROM ARCHIVE LINK
# import numpy as np
# import pandas as pd
# from sklearn.model_selection import train_test_split

# df = pd.read_csv('https://archive.org/download/quora_dataset_train.csv/quora_dataset_train.csv.zip',
#                  compression='zip',
#                  low_memory=False)
# print(df.shape)
```

In [7]:
```
# TO LOAD DATA FROM KAGGLE
import numpy as np
import pandas as pd
from sklearn.model_selection import train_test_split

df = pd.read_csv('../input/quora-insincere-questions-classification/train.csv')
print(df.shape)
```

```
(1306122, 3)
```

In [8]:
```
df.head(10)
# label 0 == non toxic
# label 1 == toxic
```

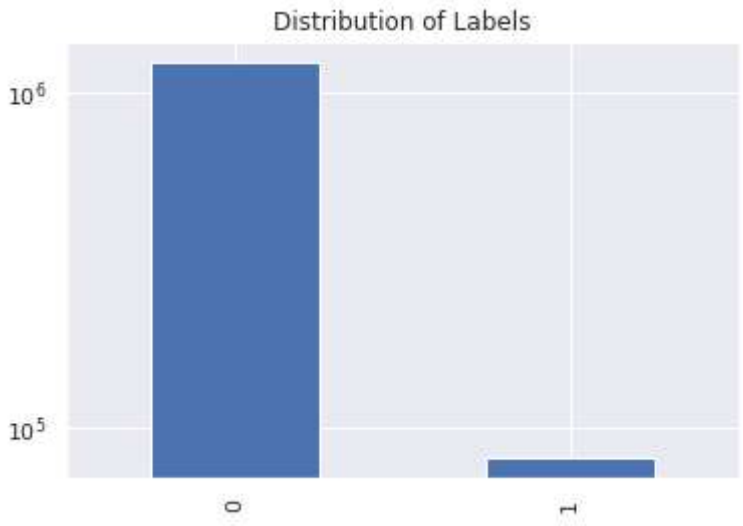
Out[8]:

	qid	question_text	target
0	00002165364db923c7e6	How did Quebec nationalists see their province...	0
1	000032939017120e6e44	Do you have an adopted dog, how would you enco...	0
2	0000412ca6e4628ce2cf	Why does velocity affect time? Does velocity a...	0
3	000042bf85aa498cd78e	How did Otto von Guericke used the Magdeburg h...	0
4	0000455dfa3e01eae3af	Can I convert montra helicon D to a mountain b...	0
5	00004f9a462a357c33be	Is Gaza slowly becoming Auschwitz, Dachau or T...	0
6	00005059a06ee19e11ad	Why does Quora automatically ban conservative ...	0
7	0000559f875832745e2e	Is it crazy if I wash or wipe my groceries off...	0
8	00005bd3426b2d0c8305	Is there such a thing as dressing moderately, ...	0
9	00006e6928c5df60eacb	Is it just me or have you ever been in this ph...	0

In [9]:
```
print(df['target'].value_counts())
df['target'].value_counts().plot.bar()
plt.yscale('log');
plt.title('Distribution of Labels')
```

```
0    1225312
1      80810
Name: target, dtype: int64
```

Out[9]: Text(0.5, 1.0, 'Distribution of Labels')



In [10]:
```
print('Average word length of questions in dataset is {0:.0
f}.'.format(np.mean(df['question_text'].apply(lambda x: len(x.
split())))))
print('Max word length of questions in dataset is {0:.0f}.'.fo
rmat(np.max(df['question_text'].apply(lambda x: len(x.split
()))))
print('Average character length of questions in dataset is
{0:.0f}.'.format(np.mean(df['question_text'].apply(lambda x: l
en(x)))))
```

```
Average word length of questions in dataset is 13.
Max word length of questions in dataset is 134.
Average character length of questions in dataset is 71.
```

In [11]:
```
# Since the dataset is very imbalanced we will keep the same
istribution in both train and test set by stratifying it based
on the labels
# using small portions of the data as the over all dataset wou
ld take ages to train, feel free to include more data by chang
ing train_size
train_df, remaining = train_test_split(df, random_state=42, tr
ain_size=0.1, stratify=df.target.values)
valid_df, _ = train_test_split(remaining, random_state=42, tra
in_size=0.01, stratify=remaining.target.values)
print(train_df.shape)
print(valid_df.shape)
```

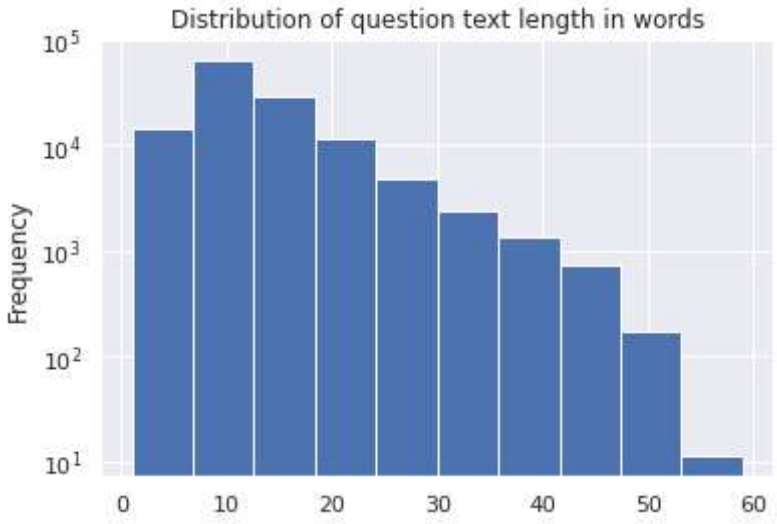
```
(130612, 3)
(11755, 3)
```

In [12]:
```
print("FOR TRAIN SET\n")
print('Average word length of questions in train set is {0:.0
f}.'.format(np.mean(train_df['question_text'].apply(lambda x:
len(x.split())))))
print('Max word length of questions in train set is {0:.0f}.'.
format(np.max(train_df['question_text'].apply(lambda x: len(x.
split())))))
print('Average character length of questions in train set is
{0:.0f}.'.format(np.mean(train_df['question_text'].apply(lambd
a x: len(x)))))
print('Label Distribution in train set is \n{}.'.format(train_
df['target'].value_counts()))
print("\n\nFOR VALIDATION SET\n")
print('Average word length of questions in valid set is {0:.0
f}.'.format(np.mean(valid_df['question_text'].apply(lambda x:
len(x.split())))))
print('Max word length of questions in valid set is {0:.0f}.'.
format(np.max(valid_df['question_text'].apply(lambda x: len(x.
split())))))
print('Average character length of questions in valid set is
{0:.0f}.'.format(np.mean(valid_df['question_text'].apply(lambd
a x: len(x)))))
print('Label Distribution in validation set is \n{}.'.format(v
alid_df['target'].value_counts()))
```

```
FOR TRAIN SET

Average word length of questions in train set is 13.
Max word length of questions in train set is 59.
Average character length of questions in train set is 71.
Label Distribution in train set is
0    122531
1      8081
Name: target, dtype: int64.


FOR VALIDATION SET

Average word length of questions in valid set is 13.
Max word length of questions in valid set is 53.
Average character length of questions in valid set is 70.
Label Distribution in validation set is
0    11028
1      727
Name: target, dtype: int64.
```

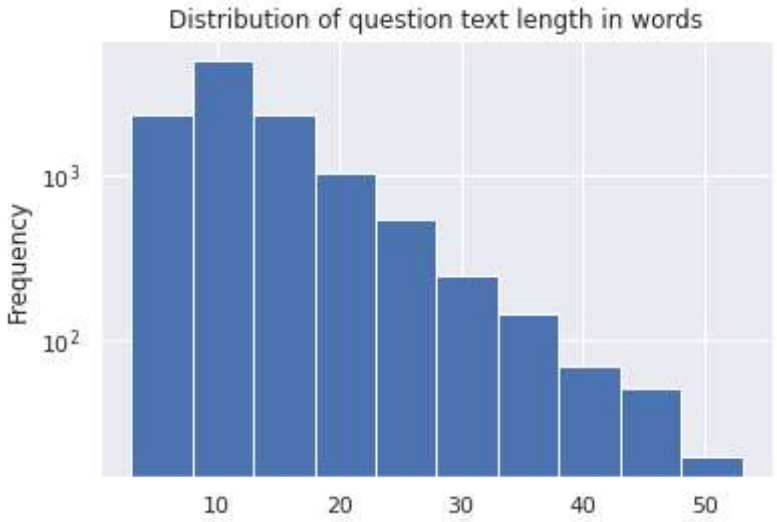
In [13]:
```
# TRAIN SET
train_df['question_text'].apply(lambda x: len(x.split())).plot
(kind='hist');
plt.yscale('log');
plt.title('Distribution of question text length in words')
```

Out[13]:
```
Text(0.5, 1.0, 'Distribution of question text length in word
s')
```



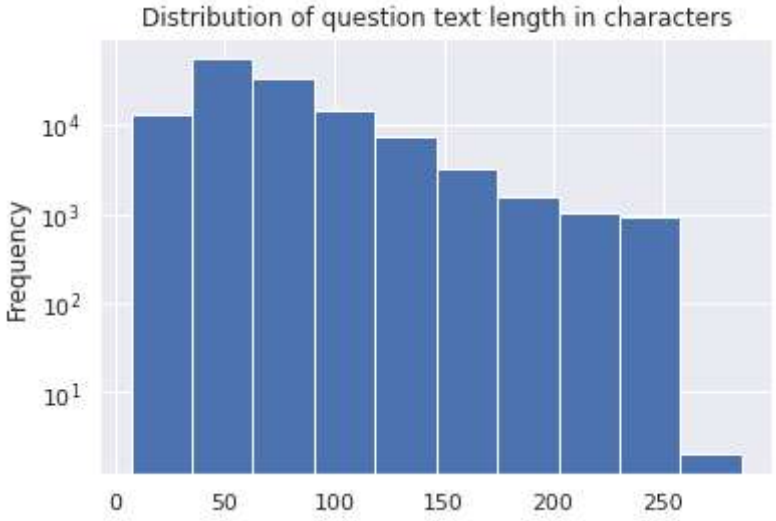
In [14]:
```
# VALIDATION SET
valid_df['question_text'].apply(lambda x: len(x.split())).plot
(kind='hist');
plt.yscale('log');
plt.title('Distribution of question text length in words')
```

Out[14]:
```
Text(0.5, 1.0, 'Distribution of question text length in word
s')
```



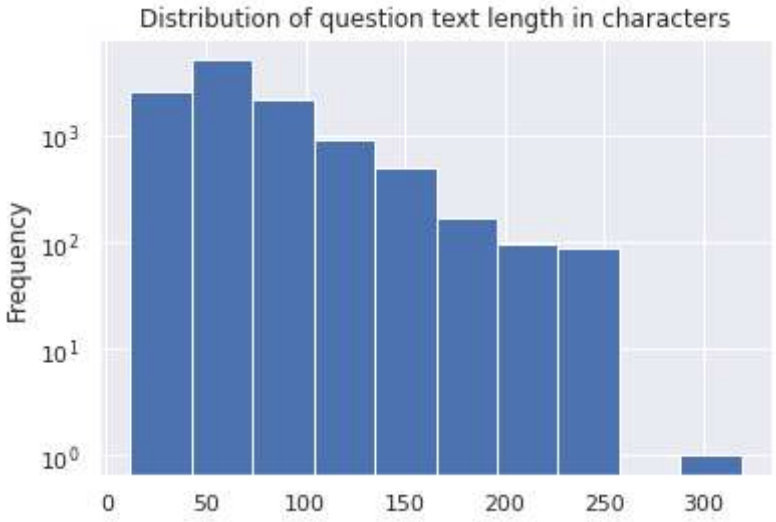
In [15]:
```
# TRAIN SET
train_df['question_text'].apply(lambda x: len(x)).plot(kind='h
ist');
plt.yscale('log');
plt.title('Distribution of question text length in character
s')
```

Out[15]:
```
Text(0.5, 1.0, 'Distribution of question text length in char
ters')
```



In [16]:
```
# VALIDATION SET
valid_df['question_text'].apply(lambda x: len(x)).plot(kind='h
ist');
plt.yscale('log');
plt.title('Distribution of question text length in character
s')
```

Out[16]:
```
Text(0.5, 1.0, 'Distribution of question text length in char
ters')
```



In [17]:
```
# we want the dataset to be created and processed on the cpu
with tf.device('/cpu:0'):
    train_data = tf.data.Dataset.from_tensor_slices((train_df
['question_text'].values, train_df['target'].values))
    valid_data = tf.data.Dataset.from_tensor_slices((valid_df
['question_text'].values, valid_df['target'].values))
    # lets look at 3 samples from train set
    for text,label in train_data.take(3):
        print(text)
        print(label)
```

```
tf.Tensor(b'If we trade in hourly timefram how we can predic
what happen in 15 minutes timefram?', shape=(), dtype=string)
tf.Tensor(0, shape=(), dtype=int64)
tf.Tensor(b'Is there any testing or coaching that helps people
decide their college major to begin their career?', shape=(),
dtype=string)
tf.Tensor(0, shape=(), dtype=int64)
tf.Tensor(b'What is Norton 360 useful for?', shape=(), dtype=s
tring)
tf.Tensor(0, shape=(), dtype=int64)
```

In [18]:
```
print(len(train_data))
print(len(valid_data))
```

```
130612
11755
```

In [19]:
```
# Setting some parameters

config = {'label_list' : [0, 1], # Label categories
          'max_seq_length' : 128, # maximum length of (token)
input sequences
          'train_batch_size' : 32,
          'learning_rate': 2e-5,
          'epochs':5,
          'optimizer': 'adam',
          'dropout': 0.5,
          'train_samples': len(train_data),
          'valid_samples': len(valid_data),
          'train_split':0.1,
          'valid_split': 0.01
         }
```

In [20]:
```
# Get BERT layer and tokenizer:
# All details here: https://tfhub.dev/tensorflow/bert_en_uncas
ed_L-12_H-768_A-12/2

bert_layer = hub.KerasLayer('https://tfhub.dev/tensorflow/bert
_en_uncased_L-12_H-768_A-12/2',
                            trainable=True)
vocab_file = bert_layer.resolved_object.vocab_file.asset_path.
numpy()
do_lower_case = bert_layer.resolved_object.do_lower_case.numpy
() # checks if the bert layer we are using is uncased or not
tokenizer = tokenization.FullTokenizer(vocab_file, do_lower_ca
se)
```

In [22]:
```
# This provides a function to convert row to input features
d label,
# this uses the classifier_data_lib which is a class defined i
n the tensorflow model garden we installed earlier
def create_feature(text, label, label_list=config['label_lis
t'], max_seq_length=config['max_seq_length'], tokenizer=tokeni
zer):
    """
    converts the datapoint into usable features for BERT using
the classifier_data_lib

    Parameters:
    text: Input text string
    label: label associated with the text
    label_list: (list) all possible labels
    max_seq_length: (int) maximum sequence length set for bert
    tokenizer: the tokenizer object instantiated by the files
in model assets

    Returns:
    feature.input_ids: The token ids for the input text string
    feature.input_masks: The padding mask generated
    feature.segment_ids: essentially here a vector of 0s since
classification
    feature.label_id: the corresponding label id from lable_li
st [0, 1] here

    """
    # since we only have 1 sentence for classification purpos
e, textr_b is None
    example = classifier_data_lib.InputExample(guid = None,
                                               text_a = text.nump
y(),
                                               text_b = None,
                                               label = label.nump
y())
    # since only 1 example, the index=0
    feature = classifier_data_lib.convert_single_example(0, ex
ample, label_list,
                                         max_seq_length, tokenizer)

    return (feature.input_ids, feature.input_mask, feature.seg
ment_ids, feature.label_id)
```

You want to use `Dataset.map` (https://www.tensorflow.org/api_docs/python/tf/data/Dataset#map) to apply this function to each element of the dataset. `Dataset.map` (https://www.tensorflow.org/api_docs/python/tf/data/Dataset#map) runs in graph mode.

- Graph tensors do not have a value.
- In graph mode you can only use TensorFlow Ops and functions.

So you can't `.map` this function directly: You need to wrap it in a `tf.py_function` (https://www.tensorflow.org/api_docs/python/tf/py_function). The `tf.py_function` (https://www.tensorflow.org/api_docs/python/tf/py_function) will pass regular tensors (with a value and a `.numpy()` method to access it), to the wrapped python function.

In [23]:
```
def create_feature_map(text, label):
    """
    A tensorflow function wrapper to apply the transformation
on the dataset.
    Parameters:
    Text: the input text string.
    label: the classification ground truth label associated wi
th the input string

    Returns:
    A tuple of a dictionary and a corresponding label_id with
it. The dictionary
    contains the input_word_ids, input_mask, input_type_ids
    """
    input_ids, input_mask, segment_ids, label_id = tf.py_funct
ion(create_feature, inp=[text, label],
                                    Tout=[tf.int32, tf.int32, tf.i
nt32, tf.int32])
    max_seq_length = config['max_seq_length']

    # py_func doesn't set the shape of the returned tensors.
    input_ids.set_shape([max_seq_length])
    input_mask.set_shape([max_seq_length])
    segment_ids.set_shape([max_seq_length])
    label_id.set_shape([])

    x = {
        'input_word_ids': input_ids,
        'input_mask': input_mask,
        'input_type_ids': segment_ids
    }
    return (x, label_id)

    # the final datapoint passed to the model is of the format
a dictionary as x and labels.
    # the dictionary have keys which should obv match
```

In [24]:
```
# Now we will simply apply the transformation to our train a
test datasets
with tf.device('/cpu:0'):
  # train
  train_data = (train_data.map(create_feature_map,
                               num_parallel_calls=tf.data.exper
imental.AUTOTUNE)

                           .shuffle(1000)
                           .batch(32, drop_remainder=True)
                           .prefetch(tf.data.experimental.AUTOT
UNE))

  # valid
  valid_data = (valid_data.map(create_feature_map,
                               num_parallel_calls=tf.data.expe
rimental.AUTOTUNE)
                           .batch(32, drop_remainder=True)
                           .prefetch(tf.data.experimental.AUTOT
UNE))
```

In [25]:
```
# train data spec, we can finally see the input datapoint is
ow converted to the BERT specific input tensor
train_data.element_spec
```

Out[25]:
```
({'input_word_ids': TensorSpec(shape=(32, 128), dtype=tf.int
2, name=None),
  'input_mask': TensorSpec(shape=(32, 128), dtype=tf.int32, na
me=None),
  'input_type_ids': TensorSpec(shape=(32, 128), dtype=tf.int3
2, name=None)},
 TensorSpec(shape=(32,), dtype=tf.int32, name=None))
```

In [26]:
```
# valid data spec
valid_data.element_spec
```

Out[26]:
```
({'input_word_ids': TensorSpec(shape=(32, 128), dtype=tf.int
2, name=None),
  'input_mask': TensorSpec(shape=(32, 128), dtype=tf.int32, na
me=None),
  'input_type_ids': TensorSpec(shape=(32, 128), dtype=tf.int3
2, name=None)},
 TensorSpec(shape=(32,), dtype=tf.int32, name=None))
```

In [27]:
```
# Building the model, input --> BERT Layer --> Classificat
n Head
def create_model():

    input_word_ids = tf.keras.layers.Input(shape=(config['max_
seq_length'],), dtype=tf.int32,
                                           name="input_word_ids")
    input_mask = tf.keras.layers.Input(shape=(config['max_seq_
length'],), dtype=tf.int32,
                                       name="input_mask")
    input_type_ids = tf.keras.layers.Input(shape=(config['max_
seq_length'],), dtype=tf.int32,
                                        name="input_type_ids")


    pooled_output, sequence_output = bert_layer([input_word_id
s, input_mask, input_type_ids])
    # for classification we only care about the pooled-output
    # at this point we can play around with the classification
head based on the downstream tasks and its complexity

    drop = tf.keras.layers.Dropout(config['dropout'])(pooled_o
utput)
    output = tf.keras.layers.Dense(1, activation='sigmoid', na
me='output')(drop)

    # inputs coming from the function
    model = tf.keras.Model(
      inputs={
        'input_word_ids': input_word_ids,
        'input_mask': input_mask,
        'input_type_ids': input_type_ids},
      outputs=output)

    return model
```

In [28]:
```
# Calling the create model function to get the keras based f
ctional model
model = create_model()
```

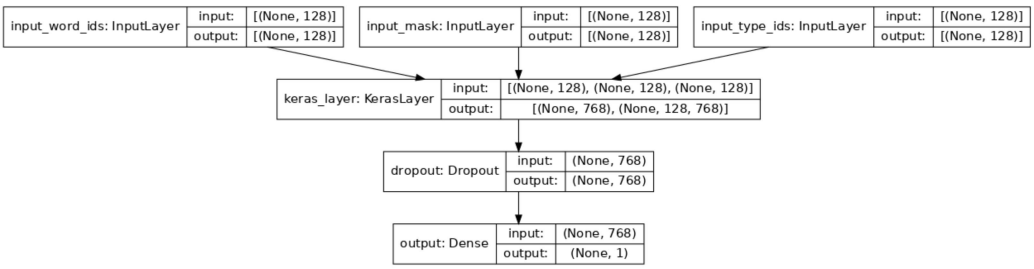
In [29]:
```
# using adam with a lr of 2*(10^-5), loss as binary cross en
opy as only 2 classes and similarly binary accuracy
model.compile(optimizer=tf.keras.optimizers.Adam(learning_rate
=config['learning_rate']),
              loss=tf.keras.losses.BinaryCrossentropy(),
              metrics=[tf.keras.metrics.BinaryAccuracy(),
                       tf.keras.metrics.PrecisionAtRecall(0.
5),
                       tf.keras.metrics.Precision(),
                       tf.keras.metrics.Recall()])
model.summary()
```

```
Model: "model"
__________________________________________________________________
_____________________________________
Layer (type)                 Output Shape              Param #
Connected to
==================================================================
=====================================
input_word_ids (InputLayer)  [(None, 128)]             0
__________________________________________________________________
_____________________________________
input_mask (InputLayer)      [(None, 128)]             0
__________________________________________________________________
_____________________________________
input_type_ids (InputLayer)  [(None, 128)]             0
__________________________________________________________________
_____________________________________
keras_layer (KerasLayer)     [(None, 768), (None, 109482241
input_word_ids[0][0]

input_mask[0][0]

input_type_ids[0][0]
__________________________________________________________________
_____________________________________
dropout (Dropout)            (None, 768)               0
keras_layer[0][0]
__________________________________________________________________
_____________________________________
output (Dense)               (None, 1)                 769
dropout[0][0]
==================================================================
=====================================
Total params: 109,483,010
Trainable params: 109,483,009
Non-trainable params: 1
__________________________________________________________________
_____________________________________
```

One drawback of the tf hub is that we import the entire module as a layer in keras as a result of which we dont see the parameters and layers in the model summary.

In [30]:
```
tf.keras.utils.plot_model(model=model, show_shapes=True, dpi
6, )
```

Out[30]:



In [31]:
```
# Update CONFIG dict with the name of the model.
config['model_name'] = 'BERT_EN_UNCASED'
print('Training configuration: ', config)

# Initialize W&B run
run = wandb.init(project='Finetune-BERT-Text-Classification',
                 config=config,
                 group='BERT_EN_UNCASED',
                 job_type='train')
```

```
wandb: Currently logged in as: akshayuppal12 (use `wandb log
--relogin` to force relogin)

Training configuration:  {'label_list': [0, 1], 'max_seq_len
h': 128, 'train_batch_size': 32, 'learning_rate': 2e-05, 'epoc
hs': 5, 'optimizer': 'adam', 'dropout': 0.5, 'train_samples':
130612, 'valid_samples': 11755, 'train_split': 0.1, 'valid_spl
it': 0.01, 'model_name': 'BERT_EN_UNCASED'}
```

Tracking run with wandb version 0.10.33
Syncing run **unique-pine-14** to Weights & Biases (https://wandb.ai) (Documentation) (https://docs.wandb.com/integrations/jupyter.html).
Project page: https://wandb.ai/akshayuppal12/Finetune-BERT-Text-Classification (https://wandb.ai/akshayuppal12/Finetune-BERT-Text-Classification)
Run page: https://wandb.ai/akshayuppal12/Finetune-BERT-Text-Classification/runs/2niqnqcy (https://wandb.ai/akshayuppal12/Finetune-BERT-Text-Classification/runs/2niqnqcy)
Run data is saved locally in `/kaggle/working/wandb/run-20210630_151759-2niqnqcy`

In [32]:

```
# Train model
# setting low epochs as It starts to overfit with this limited
data, please feel free to change
epochs = config['epochs']
history = model.fit(train_data,
                    validation_data=valid_data,
                    epochs=epochs,
                    verbose=1,
                    callbacks = [WandbCallback()])
run.finish()
```

```
Epoch 1/5
4081/4081 [==============================] - 2015s 490ms/step
- loss: 0.1327 - binary_accuracy: 0.9501 - precision_at_recal
l: 0.6104 - precision: 0.6411 - recall: 0.4350 - val_loss: 0.0
947 - val_binary_accuracy: 0.9611 - val_precision_at_recall:
0.7541 - val_precision: 0.7265 - val_recall: 0.5956
Epoch 2/5
4081/4081 [==============================] - 1998s 489ms/step
- loss: 0.0816 - binary_accuracy: 0.9680 - precision_at_recal
l: 0.8358 - precision: 0.7635 - recall: 0.7102 - val_loss: 0.1
048 - val_binary_accuracy: 0.9605 - val_precision_at_recall:
0.7525 - val_precision: 0.6871 - val_recall: 0.6644
Epoch 3/5
4081/4081 [==============================] - 1997s 489ms/step
- loss: 0.0472 - binary_accuracy: 0.9826 - precision_at_recal
l: 0.9422 - precision: 0.8580 - recall: 0.8678 - val_loss: 0.1
640 - val_binary_accuracy: 0.9578 - val_precision_at_recall:
0.7349 - val_precision: 0.7333 - val_recall: 0.4993
Epoch 4/5
4081/4081 [==============================] - 1997s 489ms/step
- loss: 0.0256 - binary_accuracy: 0.9918 - precision_at_recal
l: 0.9819 - precision: 0.9305 - recall: 0.9407 - val_loss: 0.2
049 - val_binary_accuracy: 0.9600 - val_precision_at_recall:
0.7480 - val_precision: 0.7227 - val_recall: 0.5736
Epoch 5/5
4081/4081 [==============================] - 1998s 489ms/step
- loss: 0.0163 - binary_accuracy: 0.9945 - precision_at_recal
l: 0.9938 - precision: 0.9550 - recall: 0.9582 - val_loss: 0.1
876 - val_binary_accuracy: 0.9561 - val_precision_at_recall:
0.7368 - val_precision: 0.6611 - val_recall: 0.5983
```

Waiting for W&B process to finish, PID 284
Program ended successfully.

Find user logs for this run at: `/kaggle/working/wandb/run-20210630_151759-2niqnqcy/logs/debug.log`

Find internal logs for this run at: `/kaggle/working/wandb/run-20210630_151759-2niqnqcy/logs/debug-internal.log`

Run summary:

epoch	4
loss	0.01472
binary_accuracy	0.99501
precision_at_recall	0.99547
precision	0.95863
recall	0.96076
val_loss	0.1876
val_binary_accuracy	0.95615
val_precision_at_recall	0.73684
val_precision	0.66109
val_recall	0.59835
_runtime	10013
_timestamp	1625076293
_step	4
best_val_loss	0.09467
best_epoch	0

Run history:

epoch	
loss	
binary_accuracy	
precision_at_recall	
precision	
recall	
val_loss	
val_binary_accuracy	
val_precision_at_recall	
val_precision	
val_recall	
_runtime	
_timestamp	
_step	

Synced 5 W&B file(s), 1 media file(s), 0 artifact file(s) and 1 other file(s)

In [33]:

```
# Initialize a new run for the evaluation-job
run = wandb.init(project='Finetune-BERT-Text-Classification',
                 config=config,
                 group='BERT_EN_UNCASED',
                 job_type='evaluate')



# Model Evaluation on validation set
evaluation_results = model.evaluate(valid_data,return_dict=Tru
e)

# Log scores using wandb.log()
wandb.log(evaluation_results)

# Finish the run
run.finish()
```

Tracking run with wandb version 0.10.33
Syncing run **pretty-cloud-15** to Weights & Biases (https://wandb.ai) (Documentation) (https://docs.wandb.com/integrations/jupyter.html).
Project page: https://wandb.ai/akshayuppal12/Finetune-BERT-Text-Classification (https://wandb.ai/akshayuppal12/Finetune-BERT-Text-Classification)
Run page: https://wandb.ai/akshayuppal12/Finetune-BERT-Text-Classification/runs/jzwf9er0 (https://wandb.ai/akshayuppal12/Finetune-BERT-Text-Classification/runs/jzwf9er0)
Run data is saved locally in `/kaggle/working/wandb/run-20210630_180530-jzwf9er0`

```
367/367 [==============================] - 59s 161ms/step - l
ss: 0.1876 - binary_accuracy: 0.9561 - precision_at_recall: 0.
7368 - precision: 0.6611 - recall: 0.5983
```

Waiting for W&B process to finish, PID 6917
Program ended successfully.

Find user logs for this run at: `/kaggle/working/wandb/run-20210630_180530-jzwf9er0/logs/debug.log`

Find internal logs for this run at: `/kaggle/working/wandb/run-20210630_180530-jzwf9er0/logs/debug-internal.log`

Run summary:

loss	0.1876
binary_accuracy	0.95615
precision_at_recall	0.73684
precision	0.66109
recall	0.59835
_runtime	66
_timestamp	1625076396
_step	0