SkimLit NLP Project

Project adapted from Udemy Course: TensorFlow Developer Certificate in 2023: Zero to Mastery by Daniel Bourke

Replicating the deep learning model behind the 2017 paper *PubMed 200k RCT: a Dataset for Sequenctial Sentence Classification in Medical Abstracts*.

The PubMed paper presented a new dataset called PubMed 200k RCT which consists of ~200,000 labelled Randomized Controlled Trial (RCT) abstracts.

Goal of the dataset: To explore the ability for NLP models to classify sentences which appear in sequential order. In other words, given the abstract of a RCT, what role does each sentence serve in the abstract?

Problem in a sentence

The number of RCT papers released is continuing to increase, those without structured abstracts can be hard to read and in turn slow down researchers moving through the literature.

Solution in a sentence

Create an NLP model to classify abstract sentences into the role they play (e.g. objective, methods, results, etc) to enable researchers to skim through the literature (hence SkimLit 🤓🔥) and dive deeper when necessary.

Project Details

- Downloading a text dataset (PubMed RCT200k from GitHub)
- Preprocessing Function for Data Prep
- Setting up a series of modelling experiments
  - Making a baseline (TF-IDF classifier)
  - Deep models with different combinations of:
    - token embeddings,
    - character embeddings,
    - pretrained embeddings,
    - positional embeddings
- Building our first multimodal model (taking multiple types of data inputs)

    - Replicating the model architecture from https://arxiv.org/pdf/1612.05251.pdf
- Find the most wrong predictions
- Making predictions on PubMed abstracts from the wild

```
# Check for GPU
!nvidia-smi -L
```

```
GPU 0: Tesla T4 (UUID: GPU-4a3de6aa-24f6-1b10-2192-373451dd0fe6)
```

Import Packages

```
import os
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import tensorflow as tf
from tensorflow.keras import layers
from sklearn.preprocessing import OneHotEncoder
from sklearn.preprocessing import LabelEncoder
```

Getting the data

```
!git clone https://github.com/Franck-Dernoncourt/pubmed-rct.git
!ls pubmed-rct
```

```
fatal: destination path 'pubmed-rct' already exists and is not an empty directory.
PubMed_200k_RCT
PubMed_200k_RCT_numbers_replaced_with_at_sign
PubMed_20k_RCT
PubMed_20k_RCT_numbers_replaced_with_at_sign
README.md
```

```
# Check what files are in the dataset
!ls pubmed-rct/PubMed_20k_RCT_numbers_replaced_with_at_sign
```

```
dev.txt  test.txt  train.txt
```

- train.text - training samples
- dev.txt - dev is short for development set
- test.txt - test samples

```
# Start by using the 20k dataset
```

```
data_dir = "pubmed-rct/PubMed_20k_RCT_numbers_replaced_with_at_sign/"
```

```
# Check all of the filenames in the target directory
filenames = [data_dir + filename for filename in os.listdir(data_dir)]
filenames
```

```
['pubmed-rct/PubMed_20k_RCT_numbers_replaced_with_at_sign/test.txt',
 'pubmed-rct/PubMed_20k_RCT_numbers_replaced_with_at_sign/dev.txt',
 'pubmed-rct/PubMed_20k_RCT_numbers_replaced_with_at_sign/train.txt']
```

```
# Create function to read the lines of a document
def get_lines(filename):
  """
  Reads filename (a text file) and returns the lines of text as a list.

  Args:
      filename: a string containing the target filepath to read.

  Returns:
      A list of strings with one string per line from the target filename.
      For example:
      ["this is the first line of filename",
       "this is the second line of filename",
       "..."]
  """
  with open(filename, "r") as f:
    return f.readlines()
```

```
#Try out the Get_lines functions
train_lines = get_lines(data_dir+"train.txt")
train_lines[:20] # the whole first example of an abstract + a little more of the next one
```

```
['###24293578\n',
 'OBJECTIVE\tTo investigate the efficacy of @ weeks of daily low-dose oral
prednisolone in improving pain , mobility , and systemic low-grade inflammation in the
short term and whether the effect would be sustained at @ weeks in older adults with
moderate to severe knee osteoarthritis ( OA ) .\n',
 'METHODS\tA total of @ patients with primary knee OA were randomized @:@ ; @ received
@ mg/day of prednisolone and @ received placebo for @ weeks .\n',
 'METHODS\tOutcome measures included pain reduction and improvement in function scores
and systemic inflammation markers .\n',
 'METHODS\tPain was assessed using the visual analog pain scale ( @-@ mm ) .\n',
 'METHODS\tSecondary outcome measures included the Western Ontario and McMaster
Universities Osteoarthritis Index scores , patient global assessment ( PGA ) of the
severity of knee OA , and @-min walk distance ( @MWD ) .\n',
 'METHODS\tSerum levels of interleukin @ ( IL-@ ) , IL-@ , tumor necrosis factor ( TNF
) - , and high-sensitivity C-reactive protein ( hsCRP ) were measured .\n',
 'RESULTS\tThere was a clinically relevant reduction in the intervention group
compared to the placebo group for knee pain , physical function , PGA , and @MWD at @
weeks .\n',
 'RESULTS\tThe mean difference between treatment arms ( @ % CI ) was @ ( @-@ @ ) , p <
```

```
@ ; @ ( @-@ @ ) , p < @ ; @ ( @-@ @ ) , p < @ ; and @ ( @-@ @ ) , p < @ , respectively
.\n',
 'RESULTS\tFurther , there was a clinically relevant reduction in the serum levels of
IL-@ , IL-@ , TNF - , and hsCRP at @ weeks in the intervention group when compared to
the placebo group .\n',
 'RESULTS\tThese differences remained significant at @ weeks .\n',
 'RESULTS\tThe Outcome Measures in Rheumatology Clinical Trials-Osteoarthritis
Research Society International responder rate was @ % in the intervention group and @
% in the placebo group ( p < @ ) .\n',
 'CONCLUSIONS\tLow-dose oral prednisolone had both a short-term and a longer sustained
effect resulting in less knee pain , better physical function , and attenuation of
systemic inflammation in older patients with knee OA ( ClinicalTrials.gov identifier
NCT@ ) .\n',
 '\n',
 '###24854809\n',
 'BACKGROUND\tEmotional eating is associated with overeating and the development of
obesity .\n',
 'BACKGROUND\tYet , empirical evidence for individual ( trait ) differences in
emotional eating and cognitive mechanisms that contribute to eating during sad mood
remain equivocal .\n',
 'OBJECTIVE\tThe aim of this study was to test if attention bias for food moderates
the effect of self-reported emotional eating during sad mood ( vs neutral mood ) on
actual food intake .\n',
 'OBJECTIVE\tIt was expected that emotional eating is predictive of elevated attention
for food and higher food intake after an experimentally induced sad mood and that
attentional maintenance on food predicts food intake during a sad versus a neutral
mood .\n',
 'METHODS\tParticipants ( N = @ ) were randomly assigned to one of the two
experimental mood induction conditions ( sad/neutral ) .\n']
```

```
def preprocess_text_with_line_numbers(filename):
  """Returns a list of dictionaries of abstract line data.

  Takes in filename, reads its contents and sorts through each line,
  extracting things like the target label, the text of the sentence,
  how many sentences are in the current abstract and what sentence number
  the target line is.

  Args:
      filename: a string of the target text file to read and extract line data
      from.

  Returns:
      A list of dictionaries each containing a line from an abstract,
      the lines label, the lines position in the abstract and the total number
      of lines in the abstract where the line is from. For example:

      [{"target": 'CONCLUSION',
        "text": The study couldn't have gone better, turns out people are kinder than you thi
        "line_number": 8,
        "total_lines": 8}]
  """
  input_lines = get_lines(filename) # get all lines from filename
```

```
  abstract_lines = "" # create an empty abstract
  abstract_samples = [] # create an empty list of abstracts

  # Loop through each line in target file
  for line in input_lines:
    if line.startswith("###"): # check to see if line is an ID line
      abstract_id = line
      abstract_lines = "" # reset abstract string
    elif line.isspace(): # check to see if line is a new line
      abstract_line_split = abstract_lines.splitlines() # split abstract into separate lines

      # Iterate through each line in abstract and count them at the same time
      for abstract_line_number, abstract_line in enumerate(abstract_line_split):
        line_data = {} # create empty dict to store data from line
        target_text_split = abstract_line.split("\t") # split target label from text
        line_data["target"] = target_text_split[0] # get target label
        line_data["text"] = target_text_split[1].lower() # get target text and lower it
        line_data["line_number"] = abstract_line_number # what number line does the line appe
        line_data["total_lines"] = len(abstract_line_split) - 1 # how many total lines are ir
        abstract_samples.append(line_data) # add line data to abstract samples list

    else: # if the above conditions aren't fulfilled, the line contains a labelled sentence
      abstract_lines += line

  return abstract_samples
```

```
# Get data from file and preprocess it
%%time
train_samples = preprocess_text_with_line_numbers(data_dir + "train.txt")
val_samples = preprocess_text_with_line_numbers(data_dir + "dev.txt") # dev is another name f
test_samples = preprocess_text_with_line_numbers(data_dir + "test.txt")
len(train_samples), len(val_samples), len(test_samples)
```

```
CPU times: user 430 ms, sys: 65.9 ms, total: 496 ms
Wall time: 498 ms
(180040, 30212, 30135)
```

```
# Check the first abstract of our training data
train_samples[:14]
```

tnf) - , and high-sensitivity c-reactive protein (hscrp) were measured .',
  'line_number': 5,
  'total_lines': 11},
 {'target': 'RESULTS',
  'text': 'there was a clinically relevant reduction in the intervention group compared to the placebo group for knee pain , physical function , pga , and @mwd at @ weeks .',
  'line_number': 6,
  'total_lines': 11},
 {'target': 'RESULTS',
  'text': 'the mean difference between treatment arms (@ % ci) was @ (@-@ @) , p < @ ; @ (@-@ @) , p < @ ; @ (@-@ @) , p < @ ; and @ (@-@ @) , p < @ , respectively .',
  'line_number': 7,
  'total_lines': 11},
 {'target': 'RESULTS',
  'text': 'further , there was a clinically relevant reduction in the serum levels of il-@ , il-@ , tnf - , and hscrp at @ weeks in the intervention group when compared to the placebo group .',
  'line_number': 8,
  'total_lines': 11},
 {'target': 'RESULTS',
  'text': 'these differences remained significant at @ weeks .',
  'line_number': 9,
  'total_lines': 11},
 {'target': 'RESULTS',
  'text': 'the outcome measures in rheumatology clinical trials-osteoarthritis research society international responder rate was @ % in the intervention group and @ % in the placebo group (p < @) .',
  'line_number': 10,
  'total_lines': 11},
 {'target': 'CONCLUSIONS',
  'text': 'low-dose oral prednisolone had both a short-term and a longer sustained effect resulting in less knee pain , better physical function , and attenuation of systemic inflammation in older patients with knee oa (clinicaltrials.gov identifier nct@) .',
  'line_number': 11,
  'total_lines': 11},
 {'target': 'BACKGROUND',
  'text': 'emotional eating is associated with overeating and the development of obesity .',
  'line_number': 0,
  'total_lines': 10},
 {'target': 'BACKGROUND',
  'text': 'yet , empirical evidence for individual (trait) differences in emotional eating and cognitive mechanisms that contribute to eating during sad mood remain equivocal .',

Prepping the Data

```
train_df = pd.DataFrame(train_samples)
val_df = pd.DataFrame(val_samples)
test_df = pd.DataFrame(test_samples)
train_df.head(14)
```

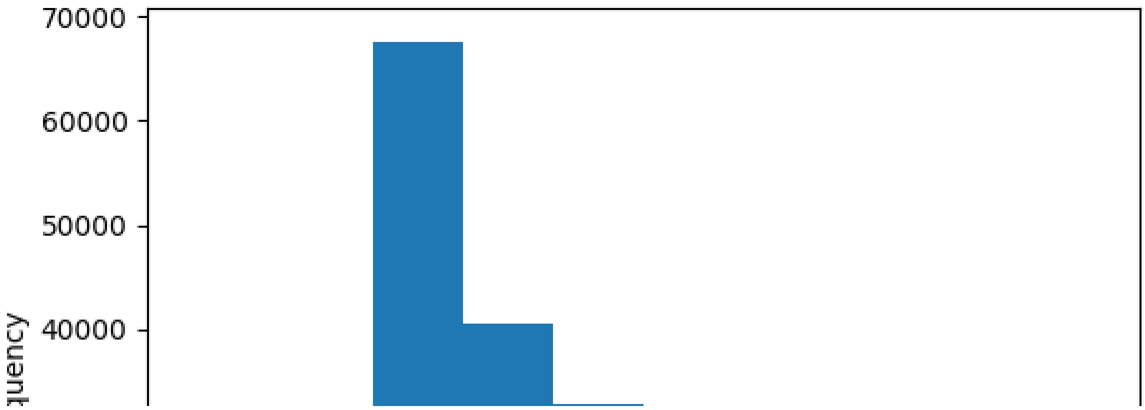
```
train_df.head(14)
```

	target	text	line_number	total_lines
0	OBJECTIVE	to investigate the efficacy of @ weeks of dail...	0	11
1	METHODS	a total of @ patients with primary knee oa wer...	1	11
2	METHODS	outcome measures included pain reduction and i...	2	11
3	METHODS	pain was assessed using the visual analog pain...	3	11
4	METHODS	secondary outcome measures included the wester...	4	11
5	METHODS	serum levels of interleukin @ (il-@) , il-@ ...	5	11
6	RESULTS	there was a clinically relevant reduction in t...	6	11
7	RESULTS	the mean difference between treatment arms (@...	7	11
8	RESULTS	further , there was a clinically relevant redu...	8	11
9	RESULTS	these differences remained significant at @ we...	9	11
10	RESULTS	the outcome measures in rheumatology clinical ...	10	11
11	CONCLUSIONS	low-dose oral prednisolone had both a short-te...	11	11
12	BACKGROUND	emotional eating is associated with overeating...	0	10
13	BACKGROUND	yet , empirical evidence for individual (trai...	1	10

```
# Distribution of labels in training data
train_df.target.value_counts()
```

```
METHODS        59353
RESULTS        57953
CONCLUSIONS    27168
BACKGROUND     21727
OBJECTIVE      13839
Name: target, dtype: int64
```

```
train_df.total_lines.plot.hist();
```



Abstracts are around 7 to 15 sentences in length.

Get List of Sentences

using `tolist()` method on `text` columns.

```
# Convert abstract text lines into lists
train_sentences = train_df["text"].tolist()
val_sentences = val_df["text"].tolist()
test_sentences = test_df["text"].tolist()
len(train_sentences), len(val_sentences), len(test_sentences)
```

```
(180040, 30212, 30135)
```

```
# View first 10 lines of training sentences
train_sentences[:10]
```

```
['to investigate the efficacy of @ weeks of daily low-dose oral prednisolone in
improving pain , mobility , and systemic low-grade inflammation in the short term and
whether the effect would be sustained at @ weeks in older adults with moderate to
severe knee osteoarthritis ( oa ) .',
 'a total of @ patients with primary knee oa were randomized @:@ ; @ received @ mg/day
of prednisolone and @ received placebo for @ weeks .',
 'outcome measures included pain reduction and improvement in function scores and
systemic inflammation markers .',
 'pain was assessed using the visual analog pain scale ( @-@ mm ) .',
 'secondary outcome measures included the western ontario and mcmaster universities
osteoarthritis index scores , patient global assessment ( pga ) of the severity of
knee oa , and @-min walk distance ( @mwd ) .',
 'serum levels of interleukin @ ( il-@ ) , il-@ , tumor necrosis factor ( tnf ) - ,
and high-sensitivity c-reactive protein ( hscrp ) were measured .',
 'there was a clinically relevant reduction in the intervention group compared to the
placebo group for knee pain , physical function , pga , and @mwd at @ weeks .',
 'the mean difference between treatment arms ( @ % ci ) was @ ( @-@ @ ) , p < @ ; @ (
@-@ @ ) , p < @ ; @ ( @-@ @ ) , p < @ ; and @ ( @-@ @ ) , p < @ , respectively .',
 'further , there was a clinically relevant reduction in the serum levels of il-@ ,
il-@ , tnf - , and hscrp at @ weeks in the intervention group when compared to the
```

```
placebo group .',
 'these differences remained significant at @ weeks .']
```

Convert Text Columns to Numeric Labels

```
train_df["target"]
```

```
0           OBJECTIVE
1             METHODS
2             METHODS
3             METHODS
4             METHODS
             ...
180035        RESULTS
180036        RESULTS
180037        RESULTS
180038    CONCLUSIONS
180039    CONCLUSIONS
Name: target, Length: 180040, dtype: object
```

```
# One hot encode labels
trial = train_df["target"].to_numpy().reshape(-1, 1)
trial
```

```
array([['OBJECTIVE'],
       ['METHODS'],
       ['METHODS'],
       ...,
       ['RESULTS'],
       ['CONCLUSIONS'],
       ['CONCLUSIONS']], dtype=object)
```

```
one_hot_encoder = OneHotEncoder(sparse=False)
train_labels_one_hot = one_hot_encoder.fit_transform(train_df["target"].to_numpy().reshape(-1
val_labels_one_hot = one_hot_encoder.transform(val_df["target"].to_numpy().reshape(-1, 1))
test_labels_one_hot = one_hot_encoder.transform(test_df["target"].to_numpy().reshape(-1, 1))

# Check what training labels look like
train_labels_one_hot
```

```
/usr/local/lib/python3.10/dist-packages/sklearn/preprocessing/_encoders.py:868: FutureW
  warnings.warn(
array([[0., 0., 0., 1., 0.],
       [0., 0., 1., 0., 0.],
       [0., 0., 1., 0., 0.],
       ...,
       [0., 0., 0., 0., 1.],
       [0., 1., 0., 0., 0.],
       [0., 1., 0., 0., 0.]])
```

Label Encode Labels

```
train_df["target"]
```

```
0           OBJECTIVE
1             METHODS
2             METHODS
3             METHODS
4             METHODS
             ...
180035        RESULTS
180036        RESULTS
180037        RESULTS
180038    CONCLUSIONS
180039    CONCLUSIONS
Name: target, Length: 180040, dtype: object
```

```
train_df["target"].to_numpy()
```

```
array(['OBJECTIVE', 'METHODS', 'METHODS', ..., 'RESULTS', 'CONCLUSIONS',
       'CONCLUSIONS'], dtype=object)
```

```
# Extract labels ("target" columns) and encode them into integers
label_encoder = LabelEncoder()
train_labels_encoded = label_encoder.fit_transform(train_df["target"].to_numpy())
val_labels_encoded = label_encoder.transform(val_df["target"].to_numpy())
test_labels_encoded = label_encoder.transform(test_df["target"].to_numpy())

# Check what training labels look like
train_labels_encoded
```

```
array([3, 2, 2, ..., 4, 1, 1])
```

```
# Get class names and number of classes from LabelEncoder instance
num_classes = len(label_encoder.classes_)
class_names = label_encoder.classes_
num_classes, class_names
```

```
(5,
 array(['BACKGROUND', 'CONCLUSIONS', 'METHODS', 'OBJECTIVE', 'RESULTS'],
       dtype=object))
```

Creating a Series of Model Experiments

Model 0: Getting a baseline

TF-IDF Multinomial Naive Bayes

```
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.naive_bayes import MultinomialNB
from sklearn.pipeline import Pipeline

#Create a pipeline
model_0 = Pipeline([
    ('tf-idf', TfidfVectorizer()),
    ('clf', MultinomialNB())
])

#Fit the pipeline to the training data
model_0.fit(X = train_sentences,
            y = train_labels_encoded)
```

```
▸ Pipeline
▸ TfidfVectorizer
▸ MultinomialNB
```

```
#Evaluate baseline on the validation dataset
model_0.score(X = val_sentences,
              y = val_labels_encoded)
```

```
0.7218323844829869
```

```
#Make predictions
baseline_preds = model_0.predict(val_sentences)
baseline_preds
```

```
array([4, 1, 3, ..., 4, 4, 1])
```

```
from sklearn.metrics import accuracy_score, precision_recall_fscore_support

def calculate_results(y_true, y_pred):
  """
  Calculates model accuracy, precision, recall and f1 score of a binary classification model.

  Args:
      y_true: true labels in the form of a 1D array
      y_pred: predicted labels in the form of a 1D array
```

```
    Returns a dictionary of accuracy, precision, recall, f1-score.
    """
    # Calculate model accuracy
    model_accuracy = accuracy_score(y_true, y_pred) * 100
    # Calculate model precision, recall and f1 score using "weighted average
    model_precision, model_recall, model_f1, _ = precision_recall_fscore_support(y_true, y_pred
    model_results = {"accuracy": model_accuracy,
                     "precision": model_precision,
                     "recall": model_recall,
                     "f1": model_f1}
    return model_results
```

```
baseline_results = calculate_results(y_true = val_labels_encoded,
                                     y_pred = baseline_preds)
```

Preparing Data for Deep Sequence Models

```
#How long is each sentence on average
sent_lens = [len(sentence.split()) for sentence in train_sentences]
avg_sent_len = np.mean(sent_lens)
avg_sent_len
```

```
26.338269273494777
```

```
plt.hist(sent_lens, bins = 7)
```

```
(array([1.5999e+05, 1.8760e+04, 1.1510e+03, 9.9000e+01, 2.8000e+01,
        1.0000e+01, 2.0000e+00]),
 array([  1.        ,  43.14285714,  85.28571429, 127.42857143,
        169.57142857, 211.71428571, 253.85714286, 296.        ]),
 <BarContainer object of 7 artists>)
```

Vast majority of sentences are between 0 and 50 tokens in length.

```
#How long of a sentence covers 95% of the lengths?
output_seq_len = int(np.percentile(sent_lens, 95))
output_seq_len
```

```
55
```

```
#Maximum sentence length in the training set
max(sent_lens)
```

```
296
```

Create Text Vectorizer

```
# How many words are in our vocabulary? (taken from 3.2 in https://arxiv.org/pdf/1710.06071.p
max_tokens = 68000
```

```
# Create text vectorizer
from tensorflow.keras.layers.experimental.preprocessing import TextVectorization

text_vectorizer = TextVectorization(max_tokens=max_tokens, # number of words in vocabulary
                                    output_sequence_length = output_seq_len) # 55 desired out
```

```
# Adapt text vectorizer to training sentences
text_vectorizer.adapt(train_sentences)
```

```
# Test out text vectorizer
import random
target_sentence = random.choice(train_sentences)
print(f"Text:\n{target_sentence}")
print(f"\nLength of text: {len(target_sentence.split())}")
print(f"\nVectorized text:\n{text_vectorizer([target_sentence])}")
```

```
Text:
an alternative strategy could be to increase physical capacity of the worker by physica

Length of text: 16
```

```
Vectorized text:
[[  26  775  606  281   36    6  179  189  713    4    2 6132   22  189
  4912    0    0    0    0    0    0    0    0    0    0    0    0    0
     0    0    0    0    0    0    0    0    0    0    0    0    0    0
     0    0    0    0    0    0    0    0    0    0    0    0    0]]
```

```
# How many words in our training vocabulary?
rct_20k_text_vocab = text_vectorizer.get_vocabulary()
print(f"Number of words in vocabulary: {len(rct_20k_text_vocab)}"),
print(f"Most common words in the vocabulary: {rct_20k_text_vocab[:5]}")
print(f"Least common words in the vocabulary: {rct_20k_text_vocab[-5:]}")
```

```
Number of words in vocabulary: 64841
Most common words in the vocabulary: ['', '[UNK]', 'the', 'and', 'of']
Least common words in the vocabulary: ['aainduced', 'aaigroup', 'aachener', 'aachen', '
```

```
# Get the config of our text vectorizer
text_vectorizer.get_config()
```

```
{'name': 'text_vectorization_1',
 'trainable': True,
 'dtype': 'string',
 'batch_input_shape': (None,),
 'max_tokens': 68000,
 'standardize': 'lower_and_strip_punctuation',
 'split': 'whitespace',
 'ngrams': None,
 'output_mode': 'int',
 'output_sequence_length': 55,
 'pad_to_max_tokens': False,
 'sparse': False,
 'ragged': False,
 'vocabulary': None,
 'idf_weights': None,
 'encoding': 'utf-8',
 'vocabulary_size': 64841}
```

Creat Custom Text Embedding

`token_vectorization` layer maps the words in our text directly to numbers.

To create a richer numerical representation of our text, we can use an **embedding**.

As our model learns (by going through many different examples of abstract sentences and their labels), it'll update its embedding to better represent the relationships between tokens in our corpus.

The `input_dim` parameter defines the size of our vocabulary. And the `output_dim` parameter defines the dimension of the embedding output.

```
# Create token embedding layer
token_embed = layers.Embedding(input_dim=len(rct_20k_text_vocab), # length of vocabulary
                               output_dim = 128, # Note: different embedding sizes result in
                               # Use masking to handle variable sequence lengths (save space)
                               mask_zero=True,
                               name="token_embedding")

# Show example embedding
print(f"Sentence before vectorization:\n{target_sentence}\n")
vectorized_sentence = text_vectorizer([target_sentence])
print(f"Sentence after vectorization (before embedding):\n{vectorized_sentence}\n")
embedded_sentence = token_embed(vectorized_sentence)
print(f"Sentence after embedding:\n{embedded_sentence}\n")
print(f"Embedded sentence shape: {embedded_sentence.shape}")
```

```
Sentence before vectorization:
an alternative strategy could be to increase physical capacity of the worker by physica

Sentence after vectorization (before embedding):
[[  26  775  606  281   36    6  179  189  713    4    2 6132   22  189
  4912    0    0    0    0    0    0    0    0    0    0    0    0    0
     0    0    0    0    0    0    0    0    0    0    0    0    0    0
     0    0    0    0    0    0    0    0    0    0    0    0    0]]

Sentence after embedding:
[[[ 0.00880931 -0.0445334  -0.01594496 ...  0.0023556  -0.04319636
   -0.0186436 ]
  [-0.03816579  0.04303927 -0.02679392 ... -0.01621354 -0.0165054
    0.00041231]
  [-0.00529419 -0.01399207 -0.03380575 ... -0.00660425  0.02043691
    0.01213894]
  ...
  [-0.03553051  0.03729558  0.02503233 ... -0.03742898  0.01919576
    0.03499012]
  [-0.03553051  0.03729558  0.02503233 ... -0.03742898  0.01919576
    0.03499012]
  [-0.03553051  0.03729558  0.02503233 ... -0.03742898  0.01919576
    0.03499012]]]

Embedded sentence shape: (1, 55, 128)
```

```
# Turn our data into TensorFlow Datasets
train_dataset = tf.data.Dataset.from_tensor_slices((train_sentences, train_labels_one_hot))
valid_dataset = tf.data.Dataset.from_tensor_slices((val_sentences, val_labels_one_hot))
test_dataset = tf.data.Dataset.from_tensor_slices((test_sentences, test_labels_one_hot))

train_dataset
```

```
<_TensorSliceDataset element_spec=(TensorSpec(shape=(), dtype=tf.string, name=None),
TensorSpec(shape=(5,), dtype=tf.float64, name=None))>
```

```
# Take the TensorSliceDataset's and turn them into prefetched batches
train_dataset = train_dataset.batch(32).prefetch(tf.data.AUTOTUNE)
valid_dataset = valid_dataset.batch(32).prefetch(tf.data.AUTOTUNE)
test_dataset = test_dataset.batch(32).prefetch(tf.data.AUTOTUNE)

train_dataset
```

```
<_PrefetchDataset element_spec=(TensorSpec(shape=(None,), dtype=tf.string, name=None),
TensorSpec(shape=(None, 5), dtype=tf.float64, name=None))>
```

Model 1: Conv1D with token Embeddings

```
# Create 1D Convolutional Model to Process Sequences
inputs = layers.Input(shape = (1, ), dtype = tf.string)
text_vectors = text_vectorizer(inputs) #vectorize text inputs
token_embeddings = token_embed(text_vectors)
x = layers.Conv1D(64,
                  kernel_size = 5,
                  padding = 'same',
                  activation = 'relu')(token_embeddings)
# Condense the  output of our feature vectors
x = layers.GlobalAveragePooling1D()(x)
outputs = layers.Dense(num_classes, activation = "softmax")(x)
model_1 = tf.keras.Model(inputs, outputs)

# Compile
# if your labels are integer form (not one hot) use sparse_categorical_crossentropy
model_1.compile(loss = "categorical_crossentropy",
                optimizer = tf.keras.optimizers.Adam(),
                metrics = ["accuracy"])
```

```
# Get Summary of Conv1D model
model_1.summary()
```

```
Model: "model"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 input_2 (InputLayer)        [(None, 1)]               0

 text_vectorization_1 (TextV  (None, 55)               0
 ectorization)

 token_embedding (Embedding)  (None, 55, 128)          8299648
```

```
 conv1d (Conv1D)            (None, 55, 64)            41024

 global_average_pooling1d (G  (None, 64)              0
 lobalAveragePooling1D)

 dense (Dense)              (None, 5)                 325

=================================================================
Total params: 8,340,997
Trainable params: 8,340,997
Non-trainable params: 0
_________________________________________________________________
```

```
# Fit the model
model_1_history = model_1.fit(train_dataset,
                              steps_per_epoch=int(0.1 * len(train_dataset)), # only fit on 10
                              epochs=3,
                              validation_data=valid_dataset,
                              validation_steps=int(0.1 * len(valid_dataset))) # only validate
```

```
Epoch 1/3
562/562 [==============================] - 49s 68ms/step - loss: 0.9092 - accuracy: 0.6
Epoch 2/3
562/562 [==============================] - 8s 15ms/step - loss: 0.6538 - accuracy: 0.75
Epoch 3/3
562/562 [==============================] - 6s 10ms/step - loss: 0.6165 - accuracy: 0.77
```

```
# Evaluate on whole validation dataset (we only validated on 10% of batches during training)
model_1.evaluate(valid_dataset)
```

```
945/945 [==============================] - 3s 3ms/step - loss: 0.5970 - accuracy: 0.785
[0.597001314163208, 0.7850853800773621]
```

```
# Make predictions (our model outputs prediction probabilities for each class)
model_1_pred_probs = model_1.predict(valid_dataset)
model_1_pred_probs
```

```
945/945 [==============================] - 4s 4ms/step
array([[4.2378804e-01, 1.7756297e-01, 7.6124303e-02, 2.9941672e-01,
        2.3108045e-02],
       [4.7093645e-01, 2.6000926e-01, 1.4449337e-02, 2.4587958e-01,
        8.7253703e-03],
       [1.3312785e-01, 4.5904196e-03, 1.4005371e-03, 8.6083597e-01,
        4.5218771e-05],
       ...,
       [3.5058601e-06, 7.0858566e-04, 6.4593734e-04, 4.2823972e-06,
        9.9863774e-01],
       [5.8962382e-02, 4.1146785e-01, 9.6570678e-02, 7.3417261e-02,
        3.5958183e-01],
```

```
           [1.6352762e-01, 7.2294986e-01, 4.1887861e-02, 2.5467094e-02,
            4.6167538e-02]], dtype=float32)
```

```
# Convert pred probs to classes
model_1_preds = tf.argmax(model_1_pred_probs, axis=1)
model_1_preds
```

```
<tf.Tensor: shape=(30212,), dtype=int64, numpy=array([0, 0, 3, ..., 4, 1, 1])>
```

```
# Calculate model_1 results
model_1_results = calculate_results(y_true=val_labels_encoded,
                                    y_pred=model_1_preds)
model_1_results
```

```
{'accuracy': 78.50853965311796,
 'precision': 0.7816778768510123,
 'recall': 0.7850853965311797,
 'f1': 0.7825152743433659}
```

Model 2: Feature Extraction with Pretrained Token Embeddings

```
# Download pretrained TensorFlow Hub USE
import tensorflow_hub as hub
tf_hub_embedding_layer = hub.KerasLayer("https://tfhub.dev/google/universal-sentence-encoder/
                                        trainable=False,
                                        name="universal_sentence_encoder")
```

```
# Test out the embedding on a random sentence
random_training_sentence = random.choice(train_sentences)
print(f"Random training sentence:\n{random_training_sentence}\n")
use_embedded_sentence = tf_hub_embedding_layer([random_training_sentence])
print(f"Sentence after embedding:\n{use_embedded_sentence[0][:30]} (truncated output)...\n")
print(f"Length of sentence embedding:\n{len(use_embedded_sentence[0])}")
```

```
Random training sentence:
average satisfaction scores were analyzed in relation to demographics , questionnaires

Sentence after embedding:
[-0.0231625  -0.06202654  0.04716952  0.0176926  -0.00405168 -0.0739309
 -0.03128929 -0.0143909  -0.02396229  0.0321661  -0.02577093  0.01497683
  0.06344731 -0.03138847 -0.03727197 -0.00570509  0.04735863  0.08149817
 -0.02998076 -0.04346459 -0.06780574  0.03653744 -0.05652207 -0.00099776
 -0.00352655  0.05447556 -0.00184657 -0.00246195  0.02418112  0.00249578] (truncated ou

Length of sentence embedding:
512
```

Building and fitting an NLP feature extraction model from TensorFlow Hub

```
# Define feature extractor model using TF Hub layer
inputs = layers.Input(shape=[], dtype=tf.string)
pretrained_embedding = tf_hub_embedding_layer(inputs) # tokenize text and create embedding
x = layers.Dense(128, activation="relu")(pretrained_embedding) # add a fully connected layer
# Note: you could add more layers here if you wanted to
outputs = layers.Dense(5, activation="softmax")(x) # create the output layer
model_2 = tf.keras.Model(inputs=inputs,
                        outputs=outputs)

# Compile the model
model_2.compile(loss="categorical_crossentropy",
                optimizer=tf.keras.optimizers.Adam(),
                metrics=["accuracy"])
```

```
# Get a summary of the model
model_2.summary()
```

```
Model: "model_1"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 input_3 (InputLayer)        [(None,)]                 0

 universal_sentence_encoder   (None, 512)              256797824
 (KerasLayer)

 dense_1 (Dense)             (None, 128)               65664

 dense_2 (Dense)             (None, 5)                 645

=================================================================
Total params: 256,864,133
Trainable params: 66,309
Non-trainable params: 256,797,824
_________________________________________________________________
```

```
# Fit feature extractor model for 3 epochs
model_2.fit(train_dataset,
            steps_per_epoch=int(0.1 * len(train_dataset)),
            epochs=3,
            validation_data=valid_dataset,
            validation_steps=int(0.1 * len(valid_dataset)))
```

```
Epoch 1/3
562/562 [==============================] - 20s 22ms/step - loss: 0.9160 - accuracy: 0.6
```

```
Epoch 2/3
562/562 [==============================] - 11s 20ms/step - loss: 0.7697 - accuracy: 0.7
Epoch 3/3
562/562 [==============================] - 12s 21ms/step - loss: 0.7541 - accuracy: 0.7
<keras.callbacks.History at 0x7f1bb0fde410>
```

```
# Evaluate on whole validation dataset
model_2.evaluate(valid_dataset)
```

```
945/945 [==============================] - 12s 13ms/step - loss: 0.7437 - accuracy: 0.7
[0.743703305721283, 0.7115053534507751]
```

```
# Make predictions with feature extraction model
model_2_pred_probs = model_2.predict(valid_dataset)
model_2_pred_probs
```

```
945/945 [==============================] - 15s 16ms/step
array([[0.4349652 , 0.36200792, 0.00183972, 0.19422705, 0.00696014],
       [0.3448899 , 0.4849474 , 0.00344235, 0.16320802, 0.00351229],
       [0.23660025, 0.16927485, 0.01857396, 0.5411838 , 0.03436713],
       ...,
       [0.0015691 , 0.00610801, 0.07008409, 0.00100961, 0.9212292 ],
       [0.00358569, 0.04297948, 0.1921192 , 0.00166867, 0.75964695],
       [0.19212972, 0.28074548, 0.46338317, 0.00569623, 0.0580454 ]],
      dtype=float32)
```

```
# Convert the predictions with feature extraction model to classes
model_2_preds = tf.argmax(model_2_pred_probs, axis=1)
model_2_preds
```

```
<tf.Tensor: shape=(30212,), dtype=int64, numpy=array([0, 1, 3, ..., 4, 4, 2])>
```

```
# Calculate results from TF Hub pretrained embeddings results on validation set
model_2_results = calculate_results(y_true=val_labels_encoded,
                                    y_pred=model_2_preds)
model_2_results
```

```
{'accuracy': 71.15053621077718,
 'precision': 0.7124071985799378,
 'recall': 0.7115053621077717,
 'f1': 0.7083211196698715}
```

Model 3: Conv1D with Character Embeddings

Token level embeddings split sequences into tokens (words) and embeddings each of them, character embeddings split sequences into characters and creates a feature vector for each.

```
# Make function to split sentences into characters
def split_chars(text):
  return " ".join(list(text))

# Test splitting non-character-level sequence into characters
split_chars(random_training_sentence)
```

```
'a v e r a g e   s a t i s f a c t i o n   s c o r e s   w e r e   a n a l y z e d   i
n   r e l a t i o n   t o   d e m o g r a p h i c s   ,   q u e s t i o n n a i r e s
,   a n d   i n v o l v e m e n t   i n   r e s e a r c h   .'
```

```
# Split sequence-level data splits into character-level data splits
train_chars = [split_chars(sentence) for sentence in train_sentences]
val_chars = [split_chars(sentence) for sentence in val_sentences]
test_chars = [split_chars(sentence) for sentence in test_sentences]
print(train_chars[0])
```

```
t o   i n v e s t i g a t e   t h e   e f f i c a c y   o f   @   w e e k s   o f   d a
```

```
# What's the average character length?
char_lens = [len(sentence) for sentence in train_sentences]
mean_char_len = np.mean(char_lens)
mean_char_len
```

```
149.3662574983337
```

```
# Check the distribution of our sequences at character-level
import matplotlib.pyplot as plt
plt.hist(char_lens, bins=7);
```


```
# Find what character length covers 95% of sequences
output_seq_char_len = int(np.percentile(char_lens, 95))
output_seq_char_len
```

```
290
```

```
# Get all keyboard characters for char-level embedding
import string
alphabet = string.ascii_lowercase + string.digits + string.punctuation
alphabet
```

```
'abcdefghijklmnopqrstuvwxyz0123456789!"#$%&\'()*+,-./:;<=>?@[\\]^_`{|}~'
```

```
# Create char-level token vectorizer instance
NUM_CHAR_TOKENS = len(alphabet) + 2 # num characters in alphabet + space + OOV token
char_vectorizer = TextVectorization(max_tokens=NUM_CHAR_TOKENS,
                                    output_sequence_length=output_seq_char_len,
                                    standardize="lower_and_strip_punctuation",
                                    name="char_vectorizer")

# Adapt character vectorizer to training characters
char_vectorizer.adapt(train_chars)
```

```
# Check character vocabulary characteristics
char_vocab = char_vectorizer.get_vocabulary()
print(f"Number of different characters in character vocab: {len(char_vocab)}")
print(f"5 most common characters: {char_vocab[:5]}")
print(f"5 least common characters: {char_vocab[-5:]}")
```

```
Number of different characters in character vocab: 28
5 most common characters: ['', '[UNK]', 'e', 't', 'i']
5 least common characters: ['k', 'x', 'z', 'q', 'j']
```

```
# Test out character vectorizer
random_train_chars = random.choice(train_chars)
print(f"Charified text:\n{random_train_chars}")
print(f"\nLength of chars: {len(random_train_chars.split())}")
vectorized_chars = char_vectorizer([random_train_chars])
print(f"\nVectorized chars:\n{vectorized_chars}")
print(f"\nLength of vectorized chars: {len(vectorized_chars[0])}")
```

```
Charified text:
p r o s p e c t i v e   r a n d o m i z e d   s t u d y   (   c a n a d i a n   t a s k

Length of chars: 61

Vectorized chars:
[[14  8  7  9 14  2 11  3  4 21  2  8  5  6 10  7 15  4 25  2 10  9  3 16
  10 19 11  5  6  5 10  4  5  6  3  5  9 23 17  7  8 11  2 11 12  5  9  9
   4 17  4 11  5  3  4  7  6  4  0  0  0  0  0  0  0  0  0  0  0  0  0  0
   0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0
   0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0
   0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0
   0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0
   0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0
   0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0
   0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0
   0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0
   0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0
   0  0]]

Length of vectorized chars: 290
```

Creating a character-level embedding

```
# Create char embedding layer
char_embed = layers.Embedding(input_dim=NUM_CHAR_TOKENS, # number of different characters
                              output_dim=25, # embedding dimension of each character (same as
                              mask_zero=False, # don't use masks (this messes up model_5 if s
                              name="char_embed")

# Test out character embedding layer
print(f"Charified text (before vectorization and embedding):\n{random_train_chars}\n")
char_embed_example = char_embed(char_vectorizer([random_train_chars]))
print(f"Embedded chars (after vectorization and embedding):\n{char_embed_example}\n")
print(f"Character embedding shape: {char_embed_example.shape}")
```

```
Charified text (before vectorization and embedding):
p r o s p e c t i v e   r a n d o m i z e d   s t u d y   (   c a n a d i a n   t a s k

Embedded chars (after vectorization and embedding):
[[[-0.01338626 -0.01549313 -0.04043607 ... -0.01927593  0.01395414
   -0.00104127]
  [ 0.02040675  0.04156807 -0.03940406 ...  0.03686092 -0.03211655
    0.03049949]
  [ 0.01418297 -0.04054802  0.04807081 ...  0.04379657  0.02974819
   -0.00891507]
  ...
  [-0.00655396 -0.00240233 -0.01780012 ...  0.0021453   0.01519904
   -0.01798589]
  [-0.00655396 -0.00240233 -0.01780012 ...  0.0021453   0.01519904
```

```
   -0.01798589]
  [-0.00655396 -0.00240233 -0.01780012 ...  0.0021453   0.01519904
   -0.01798589]]]

Character embedding shape: (1, 290, 25)
```

Building a Conv1D model to fit on character embeddings

```
# Make Conv1D on chars only
inputs = layers.Input(shape=(1,), dtype="string")
char_vectors = char_vectorizer(inputs)
char_embeddings = char_embed(char_vectors)
x = layers.Conv1D(64, kernel_size=5, padding="same", activation="relu")(char_embeddings)
x = layers.GlobalMaxPool1D()(x)
outputs = layers.Dense(num_classes, activation="softmax")(x)
model_3 = tf.keras.Model(inputs=inputs,
                         outputs=outputs,
                         name="model_3_conv1D_char_embedding")

# Compile model
model_3.compile(loss="categorical_crossentropy",
                optimizer=tf.keras.optimizers.Adam(),
                metrics=["accuracy"])


# Check the summary of conv1d_char_model
model_3.summary()
```

```
Model: "model_3_conv1D_char_embedding"
_________________________________________________________________
 Layer (type)                Output Shape              Param #   
=================================================================
 input_4 (InputLayer)        [(None, 1)]               0         
                                                                 
 char_vectorizer (TextVector  (None, 290)              0         
 ization)                                                        
                                                                 
 char_embed (Embedding)      (None, 290, 25)           1750      
                                                                 
 conv1d_1 (Conv1D)           (None, 290, 64)           8064      
                                                                 
 global_max_pooling1d (Globa  (None, 64)               0         
 lMaxPooling1D)                                                  
                                                                 
 dense_3 (Dense)             (None, 5)                 325       
                                                                 
=================================================================
Total params: 10,139
Trainable params: 10,139
```

```
Non-trainable params: 0
_________________________________________________________________
```

```
# Create Char-level batched PrefetchedDataset
# Create char datasets
train_char_dataset = tf.data.Dataset.from_tensor_slices((train_chars, train_labels_one_hot)).
val_char_dataset = tf.data.Dataset.from_tensor_slices((val_chars, val_labels_one_hot)).batch(

train_char_dataset
```

```
<_PrefetchDataset element_spec=(TensorSpec(shape=(None,), dtype=tf.string, name=None),
TensorSpec(shape=(None, 5), dtype=tf.float64, name=None))>
```

```
# Fit the model on chars only
model_3_history = model_3.fit(train_char_dataset,
                              steps_per_epoch=int(0.1 * len(train_char_dataset)),
                              epochs=3,
                              validation_data=val_char_dataset,
                              validation_steps=int(0.1 * len(val_char_dataset)))
```

```
Epoch 1/3
562/562 [==============================] - 7s 8ms/step - loss: 1.2601 - accuracy: 0.484
Epoch 2/3
562/562 [==============================] - 4s 7ms/step - loss: 1.0038 - accuracy: 0.602
Epoch 3/3
562/562 [==============================] - 7s 13ms/step - loss: 0.9216 - accuracy: 0.64
```

```
# Evaluate model_3 on whole validation char dataset
model_3.evaluate(val_char_dataset)
```

```
945/945 [==============================] - 6s 7ms/step - loss: 0.8888 - accuracy: 0.654
[0.8888437151908875, 0.6541440486907959]
```

```
# Make predictions with character model only
model_3_pred_probs = model_3.predict(val_char_dataset)
model_3_pred_probs
```

```
945/945 [==============================] - 4s 4ms/step
array([[0.16518232, 0.49410683, 0.02851628, 0.26578718, 0.04640742],
       [0.12630007, 0.6539366 , 0.01598621, 0.17138754, 0.03238958],
       [0.10227867, 0.19844791, 0.13234648, 0.4381867 , 0.12874018],
       ...,
       [0.02079967, 0.04779807, 0.14578652, 0.02709949, 0.75851625],
       [0.01039874, 0.05465805, 0.3451767 , 0.04382808, 0.54593843],
       [0.4435114 , 0.37000343, 0.05199505, 0.12497956, 0.00951063]],
      dtype=float32)
```

```
# Convert predictions to classes
model_3_preds = tf.argmax(model_3_pred_probs, axis=1)
model_3_preds
```

```
<tf.Tensor: shape=(30212,), dtype=int64, numpy=array([1, 1, 3, ..., 4, 4, 0])>
```

```
# Calculate Conv1D char only model results
model_3_results = calculate_results(y_true=val_labels_encoded,
                                    y_pred=model_3_preds)
model_3_results
```

```
{'accuracy': 65.4144048722362,
 'precision': 0.6466913701886035,
 'recall': 0.654144048722362,
 'f1': 0.6469761220509042}
```

Model 4: Combining pretrained token embeddings + character embeddings (hybrid embedding layer)

Create a stacked embedding to represent sequences before passing them to the sequence label prediction layer.

1. Create a token-level model (similar to `model_1`)
2. Create a character-level model (similar to `model_3` with a slight modification to reflect the paper)
3. Combine (using [`layers.Concatenate`]) the outputs of 1 and 2
4. Build a series of output layers on top of 3 similar to Figure 1 and section 4.2 of *[Neural Networks for Joint Sentence Classification in Medical Paper Abstracts]*
5. Construct a model which takes token and character-level sequences as input and produces sequence label probabilities as output

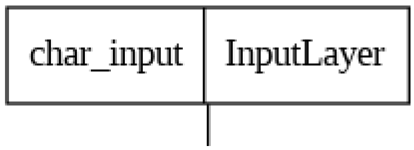
```
# 1. Setup token inputs/model
token_inputs = layers.Input(shape=[], dtype=tf.string, name="token_input")
token_embeddings = tf_hub_embedding_layer(token_inputs)
token_output = layers.Dense(128, activation="relu")(token_embeddings)
token_model = tf.keras.Model(inputs=token_inputs,
                             outputs=token_output)

# 2. Setup char inputs/model
char_inputs = layers.Input(shape=(1,), dtype=tf.string, name="char_input")
char_vectors = char_vectorizer(char_inputs)
char_embeddings = char_embed(char_vectors)
char_bi_lstm = layers.Bidirectional(layers.LSTM(25))(char_embeddings) # bi-LSTM shown in Figu
char_model = tf.keras.Model(inputs=char_inputs,
```

```
                           outputs=char_bi_lstm)

# 3. Concatenate token and char inputs (create hybrid token embedding)
token_char_concat = layers.Concatenate(name="token_char_hybrid")([token_model.output,
                                                                  char_model.output])

# 4. Create output layers - addition of dropout discussed in 4.2 of https://arxiv.org/pdf/161
combined_dropout = layers.Dropout(0.5)(token_char_concat)
combined_dense = layers.Dense(200, activation="relu")(combined_dropout) # slightly different
final_dropout = layers.Dropout(0.5)(combined_dense)
output_layer = layers.Dense(num_classes, activation="softmax")(final_dropout)

# 5. Construct model with char and token inputs
model_4 = tf.keras.Model(inputs=[token_model.input, char_model.input],
                         outputs=output_layer,
                         name="model_4_token_and_char_embeddings")


# Get summary of token and character model
model_4.summary()
```

```
     Model: "model_4_token_and_char_embeddings"
     __________________________________________________________________________________________
      Layer (type)                   Output Shape         Param #     Connected to
     ==========================================================================================
      char_input (InputLayer)        [(None, 1)]          0           []

      token_input (InputLayer)       [(None,)]            0           []

      char_vectorizer (TextVectoriza  (None, 290)         0           ['char_input[0][0]']
      tion)

      universal_sentence_encoder (Ke  (None, 512)         256797824   ['token_input[0][0]']
      rasLayer)

      char_embed (Embedding)         (None, 290, 25)      1750        ['char_vectorizer[1][0

      dense_4 (Dense)                (None, 128)          65664       ['universal_sentence_e
                                                                      ']']

      bidirectional (Bidirectional)  (None, 50)           10200       ['char_embed[1][0]']

      token_char_hybrid (Concatenate  (None, 178)         0           ['dense_4[0][0]',
      )                                                                'bidirectional[0][0]'

      dropout (Dropout)              (None, 178)          0           ['token_char_hybrid[0]

      dense_5 (Dense)                (None, 200)          35800       ['dropout[0][0]']

      dropout_1 (Dropout)            (None, 200)          0           ['dense_5[0][0]']

      dense_6 (Dense)                (None, 5)            1005        ['dropout_1[0][0]']

     ==========================================================================================
```

```
Total params: 256,912,243
Trainable params: 114,419
Non-trainable params: 256,797,824
```

```
# Plot hybrid token and character model
from tensorflow.keras.utils import plot_model
plot_model(model_4)
```




```
# Compile token char model
model_4.compile(loss="categorical_crossentropy",
                optimizer=tf.keras.optimizers.Adam(), # section 4.2 of https://arxiv.org/pdf/
                metrics=["accuracy"])
```

```
# Combine chars and tokens into a dataset
train_char_token_data = tf.data.Dataset.from_tensor_slices((train_sentences, train_chars)) #
train_char_token_labels = tf.data.Dataset.from_tensor_slices(train_labels_one_hot) # make lab
train_char_token_dataset = tf.data.Dataset.zip((train_char_token_data, train_char_token_label

# Prefetch and batch train data
train_char_token_dataset = train_char_token_dataset.batch(32).prefetch(tf.data.AUTOTUNE)

# Repeat same steps validation data
val_char_token_data = tf.data.Dataset.from_tensor_slices((val_sentences, val_chars))
val_char_token_labels = tf.data.Dataset.from_tensor_slices(val_labels_one_hot)
val_char_token_dataset = tf.data.Dataset.zip((val_char_token_data, val_char_token_labels))
val_char_token_dataset = val_char_token_dataset.batch(32).prefetch(tf.data.AUTOTUNE)
```

```
# Check out training char and token embedding dataset
train_char_token_dataset, val_char_token_dataset
```

```
(<_PrefetchDataset element_spec=((TensorSpec(shape=(None,), dtype=tf.string,
name=None), TensorSpec(shape=(None,), dtype=tf.string, name=None)), TensorSpec(shape=
(None, 5), dtype=tf.float64, name=None))>,
 <_PrefetchDataset element_spec=((TensorSpec(shape=(None,), dtype=tf.string,
name=None), TensorSpec(shape=(None,), dtype=tf.string, name=None)), TensorSpec(shape=
(None, 5), dtype=tf.float64, name=None))>)
```

```
# Fit the model on tokens and chars
model_4_history = model_4.fit(train_char_token_dataset, # train on dataset of token and chara
                              steps_per_epoch=int(0.1 * len(train_char_token_dataset)),
                              epochs=3,
                              validation_data=val_char_token_dataset,
                              validation_steps=int(0.1 * len(val_char_token_dataset)))
```

```
Epoch 1/3
562/562 [==============================] - 37s 50ms/step - loss: 0.9748 - accuracy: 0.6
Epoch 2/3
562/562 [==============================] - 21s 37ms/step - loss: 0.7938 - accuracy: 0.6
Epoch 3/3
562/562 [==============================] - 25s 44ms/step - loss: 0.7672 - accuracy: 0.7
```

```
# Evaluate on the whole validation dataset
model_4.evaluate(val_char_token_dataset)
```

```
945/945 [==============================] - 26s 28ms/step - loss: 0.6899 - accuracy: 0.7
[0.6899076104164124, 0.7363630533218384]
```

```
# Make predictions using the token-character model hybrid
model_4_pred_probs = model_4.predict(val_char_token_dataset)
model_4_pred_probs
```

```
945/945 [==============================] - 26s 26ms/step
array([[4.16466296e-01, 3.66388083e-01, 3.04558268e-03, 2.05667108e-01,
        8.43284279e-03],
       [4.05305028e-01, 3.74393225e-01, 4.01754538e-03, 2.13236988e-01,
        3.04718479e-03],
       [2.69065052e-01, 1.18500166e-01, 5.93268387e-02, 5.11718929e-01,
        4.13889922e-02],
       ...,
       [4.23132471e-04, 6.52615540e-03, 4.06211428e-02, 1.90292878e-04,
        9.52239275e-01],
       [6.94347825e-03, 6.24562390e-02, 1.79757550e-01, 3.98890348e-03,
        7.46853828e-01],
       [1.98278725e-01, 4.16518152e-01, 3.10725719e-01, 2.01918855e-02,
        5.42855412e-02]], dtype=float32)
```

```
# Turn prediction probabilities into prediction classes
model_4_preds = tf.argmax(model_4_pred_probs, axis=1)
model_4_preds
```

```
<tf.Tensor: shape=(30212,), dtype=int64, numpy=array([0, 0, 3, ..., 4, 4, 1])>
```

```
# Get results of token-char-hybrid model
model_4_results = calculate_results(y_true=val_labels_encoded,
                                    y_pred=model_4_preds)
model_4_results
```

```
{'accuracy': 73.63630345558056,
 'precision': 0.7351057052888118,
 'recall': 0.7363630345558057,
 'f1': 0.7338996335402785}
```

Model 5: Transfer Learning with pretrained token embeddings + character embeddings + positional embeddings

The process of applying your own knowledge to build features as input to a model is called feature engineering.

```
# Inspect training dataframe
train_df.head()
```

	target	text	line_number	total_lines
0	OBJECTIVE	to investigate the efficacy of @ weeks of dail...	0	11
1	METHODS	a total of @ patients with primary knee oa wer...	1	11
2	METHODS	outcome measures included pain reduction and i...	2	11
3	METHODS	pain was assessed using the visual analog pain...	3	11
4	METHODS	secondary outcome measures included the wester...	4	11

Create positional embeddings

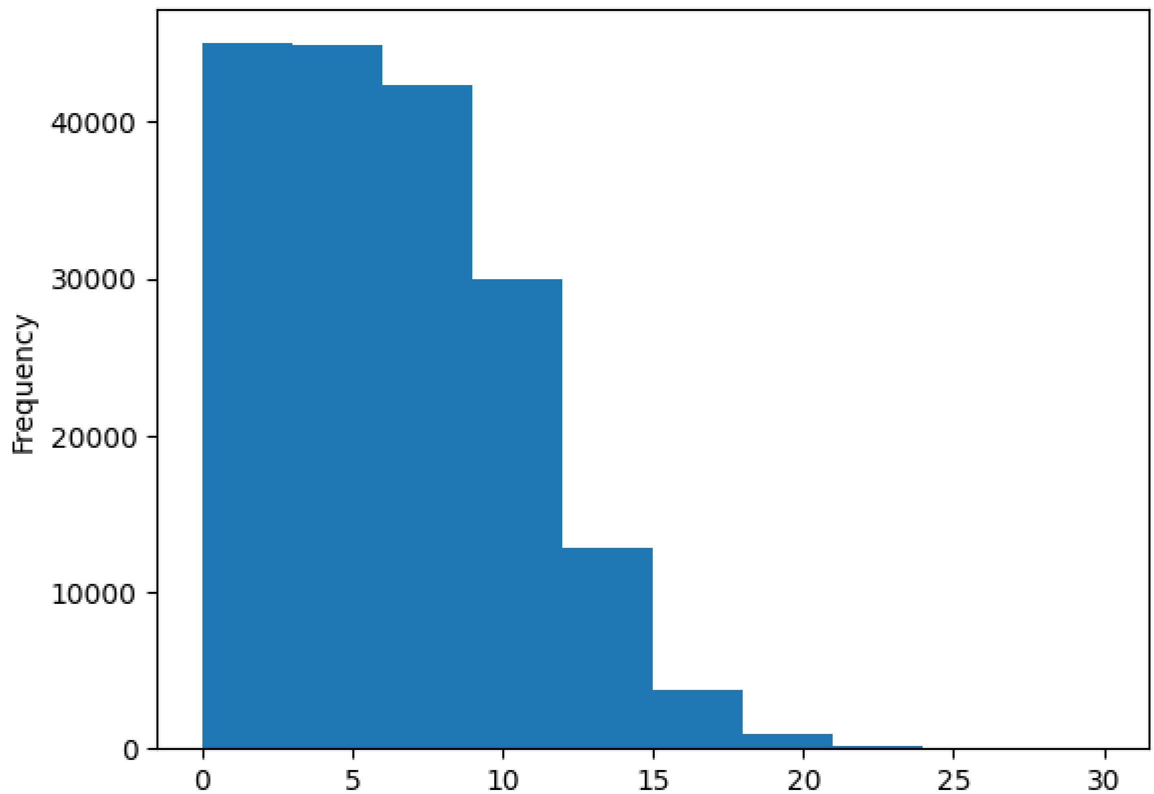
```
# How many different line numbers are there?
train_df["line_number"].value_counts()
```

```
0     15000
1     15000
2     15000
3     15000
4     14992
5     14949
6     14758
7     14279
8     13346
9     11981
10    10041
11     7892
12     5853
13     4152
14     2835
15     1861
16     1188
17      751
18      462
19      286
20      162
21      101
22       66
23       33
24       22
25       14
26        7
27        4
```

```
28      3
29      1
30      1
Name: line_number, dtype: int64
```

```
# Check the distribution of "line_number" column
train_df.line_number.plot.hist()
```

```
<Axes: ylabel='Frequency'>
```




```
# Use TensorFlow to create one-hot-encoded tensors of our "line_number" column
train_line_numbers_one_hot = tf.one_hot(train_df["line_number"].to_numpy(), depth=15)
val_line_numbers_one_hot = tf.one_hot(val_df["line_number"].to_numpy(), depth=15)
test_line_numbers_one_hot = tf.one_hot(test_df["line_number"].to_numpy(), depth=15)
```

```
# Check one-hot encoded "line_number" feature samples
train_line_numbers_one_hot.shape, train_line_numbers_one_hot[:20]
```

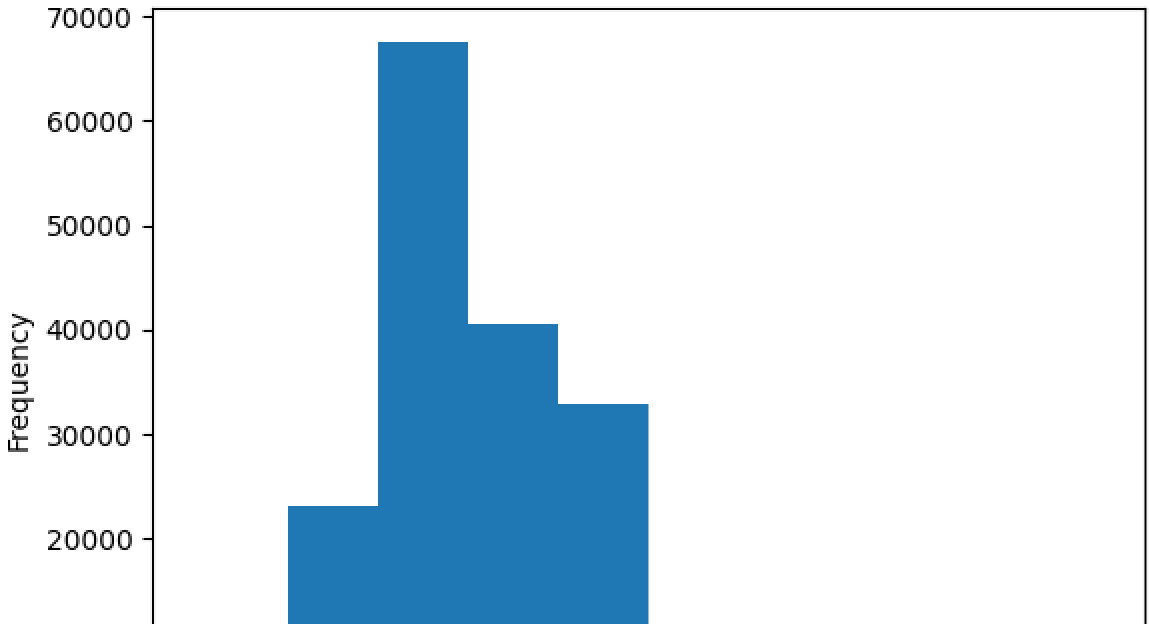
```
(TensorShape([180040, 15]),
 <tf.Tensor: shape=(20, 15), dtype=float32, numpy=
 array([[1., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.],
        [0., 1., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.],
        [0., 0., 1., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.],
        [0., 0., 0., 1., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.],
        [0., 0., 0., 0., 1., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.],
        [0., 0., 0., 0., 0., 1., 0., 0., 0., 0., 0., 0., 0., 0., 0.],
        [0., 0., 0., 0., 0., 0., 1., 0., 0., 0., 0., 0., 0., 0., 0.],
        [0., 0., 0., 0., 0., 0., 0., 1., 0., 0., 0., 0., 0., 0., 0.],
        [0., 0., 0., 0., 0., 0., 0., 0., 1., 0., 0., 0., 0., 0., 0.],
```

```
        [0., 0., 0., 0., 0., 0., 0., 0., 0., 1., 0., 0., 0., 0., 0.],
        [0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 1., 0., 0., 0., 0.],
        [0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 1., 0., 0., 0.],
        [1., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.],
        [0., 1., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.],
        [0., 0., 1., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.],
        [0., 0., 0., 1., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.],
        [0., 0., 0., 0., 1., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.],
        [0., 0., 0., 0., 0., 1., 0., 0., 0., 0., 0., 0., 0., 0., 0.],
        [0., 0., 0., 0., 0., 0., 1., 0., 0., 0., 0., 0., 0., 0., 0.],
        [0., 0., 0., 0., 0., 0., 0., 1., 0., 0., 0., 0., 0., 0., 0.]],
       dtype=float32)>)
```

```
# How many different numbers of lines are there?
train_df["total_lines"].value_counts()
```

```
11    24468
10    23639
12    22113
9     19400
13    18438
14    14610
8     12285
15    10768
7      7464
16     7429
17     5202
6      3353
18     3344
19     2480
20     1281
5      1146
21      770
22      759
23      264
4       215
24      200
25      182
26       81
28       58
3        32
30       31
27       28
Name: total_lines, dtype: int64
```

```
# Check the distribution of total lines
train_df.total_lines.plot.hist();
```




```
# Check the coverage of a "total_lines" value of 20
np.percentile(train_df.total_lines, 98) # a value of 20 covers 98% of samples
```

```
20.0
```

```
# Use TensorFlow to create one-hot-encoded tensors of our "total_lines" column
train_total_lines_one_hot = tf.one_hot(train_df["total_lines"].to_numpy(), depth=20)
val_total_lines_one_hot = tf.one_hot(val_df["total_lines"].to_numpy(), depth=20)
test_total_lines_one_hot = tf.one_hot(test_df["total_lines"].to_numpy(), depth=20)

# Check shape and samples of total lines one-hot tensor
train_total_lines_one_hot.shape, train_total_lines_one_hot[:10]
```

```
(TensorShape([180040, 20]),
 <tf.Tensor: shape=(10, 20), dtype=float32, numpy=
 array([[0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 1., 0., 0., 0., 0.,
         0., 0., 0., 0.],
        [0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 1., 0., 0., 0., 0.,
         0., 0., 0., 0.],
        [0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 1., 0., 0., 0., 0.,
         0., 0., 0., 0.],
        [0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 1., 0., 0., 0., 0.,
         0., 0., 0., 0.],
        [0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 1., 0., 0., 0., 0.,
         0., 0., 0., 0.],
        [0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 1., 0., 0., 0., 0.,
         0., 0., 0., 0.],
        [0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 1., 0., 0., 0., 0.,
         0., 0., 0., 0.],
        [0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 1., 0., 0., 0., 0.,
         0., 0., 0., 0.],
        [0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 1., 0., 0., 0., 0.,
         0., 0., 0., 0.],
        [0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 1., 0., 0., 0., 0.,
         0., 0., 0., 0.]], dtype=float32)>)
```

Building a tribrid embedding model

1. Create a token-level model (similar to `model_1`)
2. Create a character-level model (similar to `model_3` with a slight modification to reflect the paper)
3. Create a `"line_number"` model (takes in one-hot-encoded `"line_number"` tensor and passes it through a non-linear layer)
4. Create a `"total_lines"` model (takes in one-hot-encoded `"total_lines"` tensor and passes it through a non-linear layer)
5. Combine (using `layers.Concatenate`) the outputs of 1 and 2 into a token-character-hybrid embedding and pass it series of output to Figure 1 and section 4.2 of *Neural Networks for Joint Sentence Classification in Medical Paper Abstracts*
6. Combine (using `layers.Concatenate`) the outputs of 3, 4 and 5 into a token-character-positional tribrid embedding
7. Create an output layer to accept the tribrid embedding and output predicted label probabilities
8. Combine the inputs of 1, 2, 3, 4 and outputs of 7 into a `tf.keras.Model`

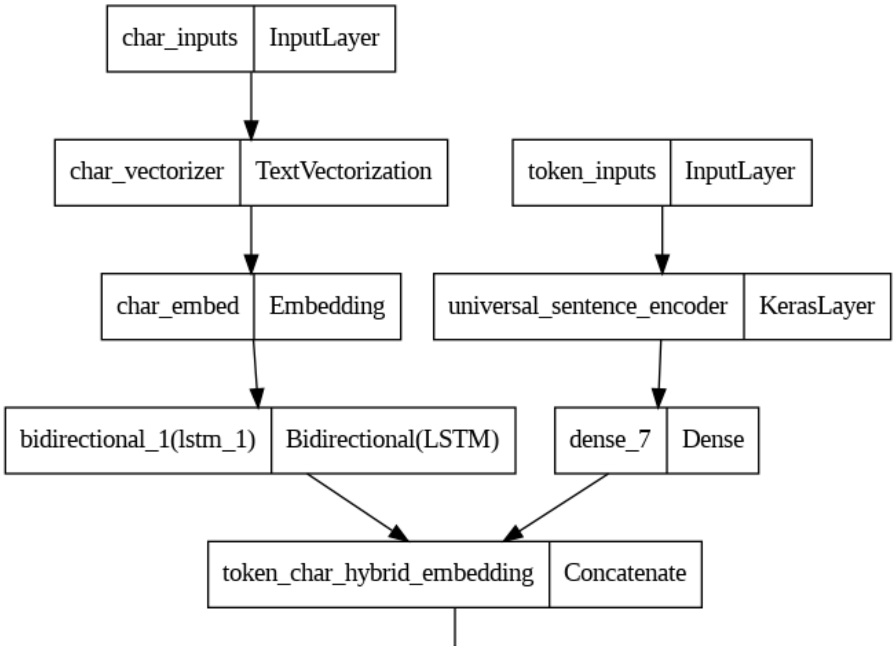
```
# 1. Token inputs
token_inputs = layers.Input(shape=[], dtype="string", name="token_inputs")
token_embeddings = tf_hub_embedding_layer(token_inputs)
token_outputs = layers.Dense(128, activation="relu")(token_embeddings)
token_model = tf.keras.Model(inputs=token_inputs,
                             outputs=token_outputs)

# 2. Char inputs
char_inputs = layers.Input(shape=(1,), dtype="string", name="char_inputs")
char_vectors = char_vectorizer(char_inputs)
char_embeddings = char_embed(char_vectors)
char_bi_lstm = layers.Bidirectional(layers.LSTM(32))(char_embeddings)
char_model = tf.keras.Model(inputs=char_inputs,
                            outputs=char_bi_lstm)

# 3. Line numbers inputs
line_number_inputs = layers.Input(shape=(15,), dtype=tf.int32, name="line_number_input")
x = layers.Dense(32, activation="relu")(line_number_inputs)
line_number_model = tf.keras.Model(inputs=line_number_inputs,
                                   outputs=x)

# 4. Total lines inputs
total_lines_inputs = layers.Input(shape=(20,), dtype=tf.int32, name="total_lines_input")
y = layers.Dense(32, activation="relu")(total_lines_inputs)
```

```
total_line_model = tf.keras.Model(inputs=total_lines_inputs,
                                  outputs=y)

# 5. Combine token and char embeddings into a hybrid embedding
combined_embeddings = layers.Concatenate(name="token_char_hybrid_embedding")([token_model.out
                                                                              char_model.outp
z = layers.Dense(256, activation="relu")(combined_embeddings)
z = layers.Dropout(0.5)(z)

# 6. Combine positional embeddings with combined token and char embeddings into a tribrid emb
z = layers.Concatenate(name="token_char_positional_embedding")([line_number_model.output,
                                                                total_line_model.output,
                                                                z])

# 7. Create output layer
output_layer = layers.Dense(5, activation="softmax", name="output_layer")(z)

# 8. Put together model
model_5 = tf.keras.Model(inputs=[line_number_model.input,
                                 total_line_model.input,
                                 token_model.input,
                                 char_model.input],
                         outputs=output_layer)


# Get a summary of our token, char and positional embedding model
model_5.summary()
```

```
Model: "model_8"
________________________________________________________________________________________
 Layer (type)                   Output Shape         Param #     Connected to
========================================================================================
 char_inputs (InputLayer)       [(None, 1)]          0           []

 token_inputs (InputLayer)      [(None,)]            0           []

 char_vectorizer (TextVectoriza  (None, 290)         0           ['char_inputs[0][0]']
 tion)

 universal_sentence_encoder (Ke  (None, 512)         256797824   ['token_inputs[0][0]']
 rasLayer)

 char_embed (Embedding)         (None, 290, 25)      1750        ['char_vectorizer[2][0

 dense_7 (Dense)                (None, 128)          65664       ['universal_sentence_e
                                                                 ]']

 bidirectional_1 (Bidirectional  (None, 64)          14848       ['char_embed[2][0]']
 )

 token_char_hybrid_embedding (C  (None, 192)         0           ['dense_7[0][0]',
 oncatenate)                                                      'bidirectional_1[0][0

 line_number_input (InputLayer)  [(None, 15)]        0           []
```

```
 total_lines_input (InputLayer)  [(None, 20)]        0           []

 dense_10 (Dense)               (None, 256)          49408       ['token_char_hybrid_em
                                                                 0]']

 dense_8 (Dense)                (None, 32)           512         ['line_number_input[0]

 dense_9 (Dense)                (None, 32)           672         ['total_lines_input[0]

 dropout_2 (Dropout)            (None, 256)          0           ['dense_10[0][0]']

 token_char_positional_embeddin  (None, 320)         0           ['dense_8[0][0]',
 g (Concatenate)                                                  'dense_9[0][0]',
                                                                  'dropout_2[0][0]']

 output_layer (Dense)           (None, 5)            1605        ['token_char_positiona
                                                                 [0][0]']

==========================================================================================
Total params: 256,932,283
Trainable params: 134,459
Non-trainable params: 256,797,824
__________________________________________________________________________________________
```

```
# Plot the token, char, positional embedding model
from tensorflow.keras.utils import plot_model
plot_model(model_5)
```




```
# Check which layers of our model are trainable or not
for layer in model_5.layers:
  print(layer, layer.trainable)
```

```
<keras.engine.input_layer.InputLayer object at 0x7f1bc9eef520> True
<keras.engine.input_layer.InputLayer object at 0x7f1bc9ed2170> True
<keras.layers.preprocessing.text_vectorization.TextVectorization object at 0x7f1bac333d
<tensorflow_hub.keras_layer.KerasLayer object at 0x7f1bc4c8ebf0> False
<keras.layers.core.embedding.Embedding object at 0x7f1b69745d20> True
<keras.layers.core.dense.Dense object at 0x7f1bc9eed300> True
<keras.layers.rnn.bidirectional.Bidirectional object at 0x7f1bc9574c40> True
<keras.layers.merging.concatenate.Concatenate object at 0x7f1bc95ce290> True
<keras.engine.input_layer.InputLayer object at 0x7f1bc9eaaf80> True
<keras.engine.input_layer.InputLayer object at 0x7f1bc95777c0> True
<keras.layers.core.dense.Dense object at 0x7f1bc9575420> True
<keras.layers.core.dense.Dense object at 0x7f1bc9eef700> True
<keras.layers.core.dense.Dense object at 0x7f1bc9574eb0> True
<keras.layers.regularization.dropout.Dropout object at 0x7f1bc9577820> True
<keras.layers.merging.concatenate.Concatenate object at 0x7f1bc9576e90> True
<keras.layers.core.dense.Dense object at 0x7f1bc95ce440> True
```

```
# Compile token, char, positional embedding model
model_5.compile(loss=tf.keras.losses.CategoricalCrossentropy(label_smoothing=0.2), # add labe
                optimizer=tf.keras.optimizers.Adam(),
                metrics=["accuracy"])
```

Create tribrid embedding datasets and fit tribrid model

1. Train line numbers one-hot tensor (`train_line_numbers_one_hot`)
2. Train total lines one-hot tensor (`train_total_lines_one_hot`)

3. Token-level sequences tensor (`train_sentences`)
4. Char-level sequences tensor (`train_chars`)

```
# Create training and validation datasets (all four kinds of inputs)
train_pos_char_token_data = tf.data.Dataset.from_tensor_slices((train_line_numbers_one_hot, #
                                                                train_total_lines_one_hot, #
                                                                train_sentences, # train toke
                                                                train_chars)) # train chars
train_pos_char_token_labels = tf.data.Dataset.from_tensor_slices(train_labels_one_hot) # trai
train_pos_char_token_dataset = tf.data.Dataset.zip((train_pos_char_token_data, train_pos_char
train_pos_char_token_dataset = train_pos_char_token_dataset.batch(32).prefetch(tf.data.AUTOTU

# Validation dataset
val_pos_char_token_data = tf.data.Dataset.from_tensor_slices((val_line_numbers_one_hot,
                                                              val_total_lines_one_hot,
                                                              val_sentences,
                                                              val_chars))
val_pos_char_token_labels = tf.data.Dataset.from_tensor_slices(val_labels_one_hot)
val_pos_char_token_dataset = tf.data.Dataset.zip((val_pos_char_token_data, val_pos_char_toker
val_pos_char_token_dataset = val_pos_char_token_dataset.batch(32).prefetch(tf.data.AUTOTUNE)

# Check input shapes
train_pos_char_token_dataset, val_pos_char_token_dataset
```

```
(<_PrefetchDataset element_spec=((TensorSpec(shape=(None, 15), dtype=tf.float32,
name=None), TensorSpec(shape=(None, 20), dtype=tf.float32, name=None),
TensorSpec(shape=(None,), dtype=tf.string, name=None), TensorSpec(shape=(None,),
dtype=tf.string, name=None)), TensorSpec(shape=(None, 5), dtype=tf.float64,
name=None))>,
 <_PrefetchDataset element_spec=((TensorSpec(shape=(None, 15), dtype=tf.float32,
name=None), TensorSpec(shape=(None, 20), dtype=tf.float32, name=None),
TensorSpec(shape=(None,), dtype=tf.string, name=None), TensorSpec(shape=(None,),
dtype=tf.string, name=None)), TensorSpec(shape=(None, 5), dtype=tf.float64,
name=None))>)
```

```
# Fit the token, char and positional embedding model
history_model_5 = model_5.fit(train_pos_char_token_dataset,
                              steps_per_epoch=int(0.1 * len(train_pos_char_token_dataset)),
                              epochs=3,
                              validation_data=val_pos_char_token_dataset,
                              validation_steps=int(0.1 * len(val_pos_char_token_dataset)))
```

```
Epoch 1/3
562/562 [==============================] - 38s 56ms/step - loss: 1.1003 - accuracy: 0.7
Epoch 2/3
562/562 [==============================] - 25s 45ms/step - loss: 0.9670 - accuracy: 0.8
Epoch 3/3
562/562 [==============================] - 25s 44ms/step - loss: 0.9507 - accuracy: 0.8
```

```
# Make predictions with token-char-positional hybrid model
model_5_pred_probs = model_5.predict(val_pos_char_token_dataset, verbose=1)
model_5_pred_probs
```

```
945/945 [==============================] - 20s 20ms/step
array([[0.47737786, 0.1209317 , 0.0121259 , 0.3697727 , 0.01979188],
       [0.5054566 , 0.12267946, 0.03957064, 0.32215586, 0.01013746],
       [0.27847195, 0.12190523, 0.08974613, 0.4535277 , 0.05634904],
       ...,
       [0.04319233, 0.12466518, 0.04647172, 0.03668554, 0.7489853 ],
       [0.03369179, 0.2986827 , 0.07830434, 0.02900375, 0.5603174 ],
       [0.16861515, 0.6089562 , 0.10881202, 0.03827903, 0.0753376 ]],
      dtype=float32)
```

```
# Turn prediction probabilities into prediction classes
model_5_preds = tf.argmax(model_5_pred_probs, axis=1)
model_5_preds
```

```
<tf.Tensor: shape=(30212,), dtype=int64, numpy=array([0, 0, 3, ..., 4, 4, 1])>
```

```
# Calculate results of token-char-positional hybrid model
model_5_results = calculate_results(y_true=val_labels_encoded,
                                    y_pred=model_5_preds)
model_5_results
```

```
{'accuracy': 83.18217926651663,
 'precision': 0.8310243527374848,
 'recall': 0.8318217926651662,
 'f1': 0.8307479182602164}
```

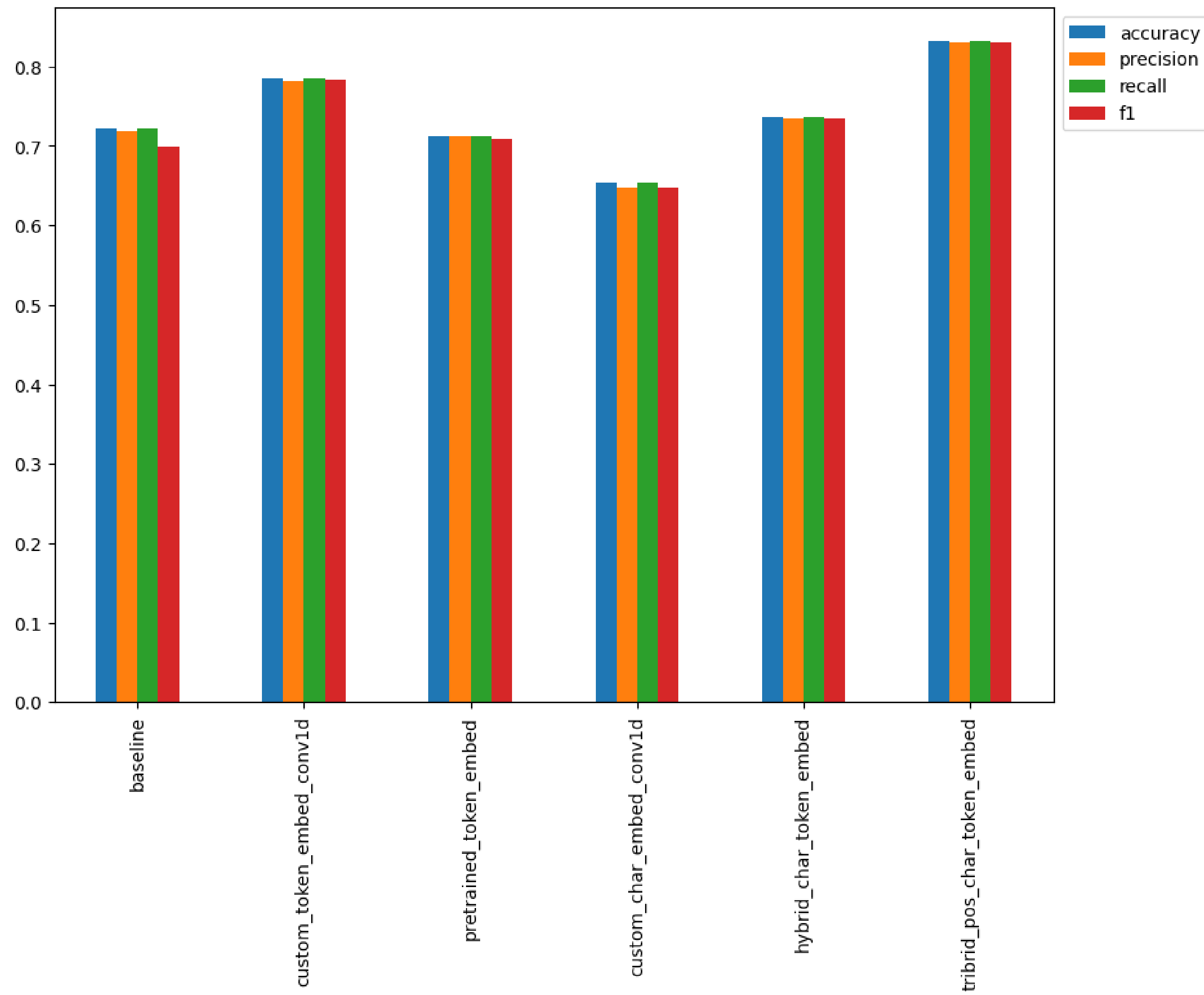
Compare model results

```
# Combine model results into a DataFrame
all_model_results = pd.DataFrame({"baseline": baseline_results,
                                  "custom_token_embed_conv1d": model_1_results,
                                  "pretrained_token_embed": model_2_results,
                                  "custom_char_embed_conv1d": model_3_results,
                                  "hybrid_char_token_embed": model_4_results,
                                  "tribrid_pos_char_token_embed": model_5_results})
all_model_results = all_model_results.transpose()
all_model_results
```

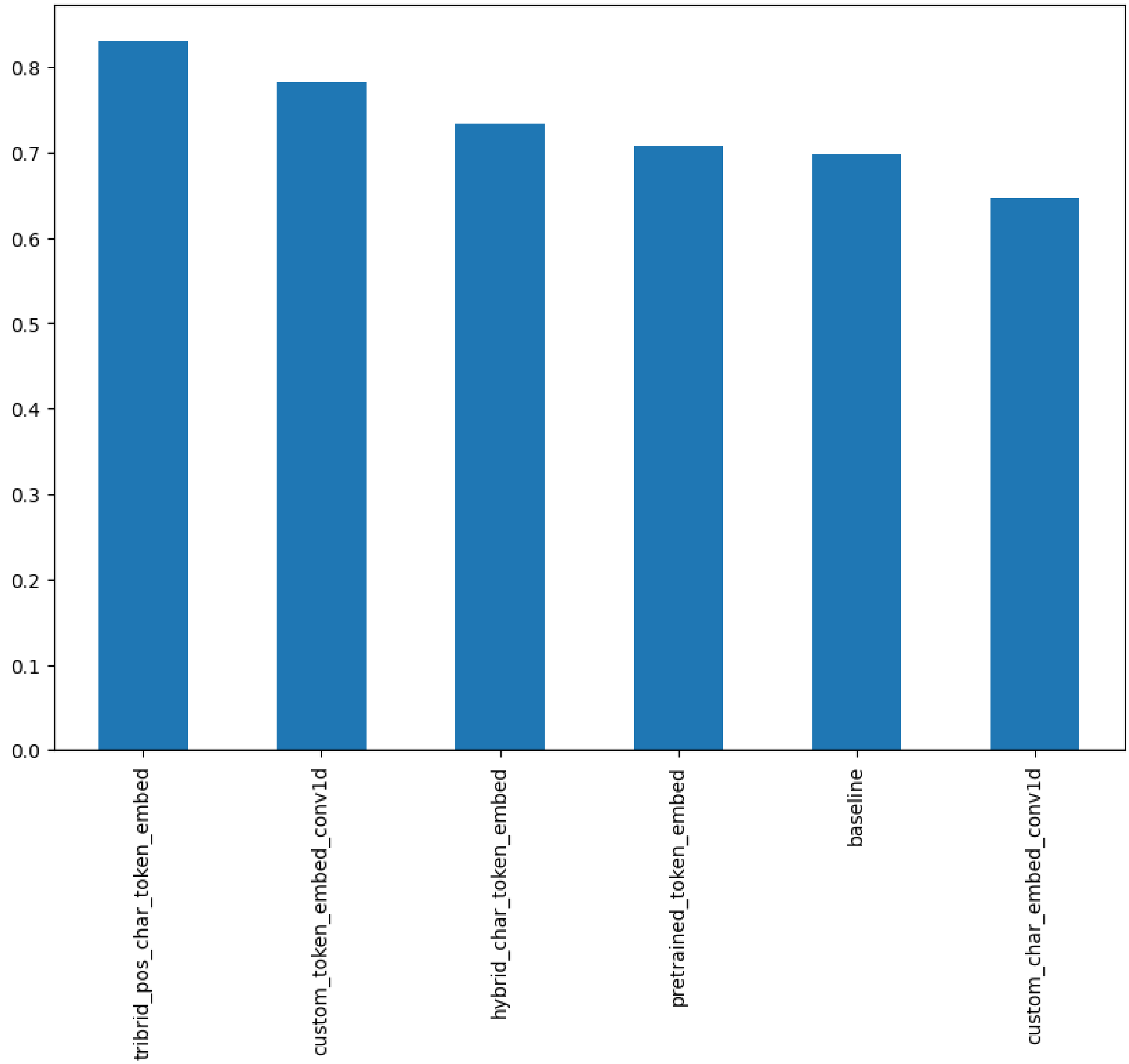
	accuracy	precision	recall	f1
baseline	72.183238	0.718647	0.721832	0.698925
custom_token_embed_conv1d	78.508540	0.781678	0.785085	0.782515
pretrained_token_embed	71.150536	0.712407	0.711505	0.708321
hybrid_char_token_embed	73.636303	0.735106	0.736363	0.733900

```
# Reduce the accuracy to same scale as other metrics
all_model_results["accuracy"] = all_model_results["accuracy"]/100
```

```
# Plot and compare all of the model results
all_model_results.plot(kind="bar", figsize=(10, 7)).legend(bbox_to_anchor=(1.0, 1.0));
```



```
# Sort model results by f1-score
all_model_results.sort_values("f1", ascending=False)["f1"].plot(kind="bar", figsize=(10, 7));
```



Based on F1-scores, it looks like our tribrid embedding model performs the best by a fair margin.

There are some things to note about this difference:

- Our models (with an exception for the baseline) have been trained on ~18,000 (10% of batches) samples of sequences and labels rather than the full ~180,000 in the 20k RCT dataset.
    - This is often the case in machine learning experiments though, make sure training works on a smaller number of samples, then upscale when needed (an extension to this project will be training a model on the full dataset).
- Our model's prediction performance levels have been evaluated on the validation dataset not the test dataset (we'll evaluate our best model on the test dataset shortly).

Save and load best performing model

```
# Save best performing model to SavedModel format (default)
model_5.save("skimlit_tribrid_model") # model will be saved to path specified by string
```

```
WARNING:absl:Found untraced functions such as lstm_cell_4_layer_call_fn, lstm_cell_4_la
```

```
from google.colab import drive

drive.mount('/content/gdrive/', force_remount=True)
```

```
Mounted at /content/gdrive/
```