Recurrent Neural Network for Text Classification

Olivia Schultheis

The objective of this project is to classify comments as related to ‘physics’, ‘chemistry’, or ‘biology.’ Each comment contains words that convey subject-related matter to these 3 fields, but there is some overlap as they are all sciences. It will be interesting to see whether the model can distinguish between comments related to these subjects. The dataset comes from Kaggle, and two CSV files were provided, already split into ‘training’ and ‘testing’ data. The training data consists of 8,695 comments (or rows), and the testing data consists of 1,586 comments (or rows). The data of course requires cleaning, as there are extraneous symbols, punctuation, numbers, and extra spaces that will only slow down the training process. Stop words will be removed and words will be reduced to their stems to help with training. The link to the dataset is:

<https://www.kaggle.com/datasets/vivmankar/physics-vs-chemistry-vs-biology/data>

Besides an ‘id’ column, the only other column except for the textual comments themselves is a column called ‘Topic’, which is either biology, chemistry, or physics. This makes the RNN model development a classification task with 3 categories. The ‘softmax’ activation function will be used with the categorical cross-entropy loss function.

Set working Directory:

import os  
os.getcwd()  
os.chdir('/users/oliviaschultheis/Desktop/Machine Learning with Python')

Import Necessary Packages

import pandas as pd  
  
import numpy as np  
import pandas as pd  
import tensorflow as tf  
  
import re  
from nltk.corpus import stopwords  
from nltk.stem import SnowballStemmer  
from keras.utils import np\_utils  
  
import matplotlib.pyplot as plt

import nltk  
nltk.download('stopwords')

[nltk\_data] Downloading package stopwords to  
[nltk\_data] /Users/oliviaschultheis/nltk\_data...  
[nltk\_data] Package stopwords is already up-to-date!

Now, we import the two CSV files to create the training data and testing data. We specify the columns that we would like to import in the ‘usecols’ argument.

pd.set\_option('display.max\_colwidth', None)

train = pd.read\_csv(  
 'science\_train.csv',   
 usecols=['Comment', 'Topic'],   
 dtype={'Comment': str, 'Topic': str}  
)

Looking at the shape of the training data, we can confirm that there are 8,695 comments in the training data.

train.shape # 8695 sentences

(8695, 2)

train.info()

<class 'pandas.core.frame.DataFrame'>  
RangeIndex: 8695 entries, 0 to 8694  
Data columns (total 2 columns):  
 # Column Non-Null Count Dtype   
--- ------ -------------- -----   
 0 Comment 8695 non-null object  
 1 Topic 8695 non-null object  
dtypes: object(2)  
memory usage: 136.0+ KB

It is immediately noticeable upon looking at the data that there are plenty of new-line characters that are extraneous for training. We can replace these with blank spaces with the regular expression below. As a result, there are now blank spaces where the new-line characters were before.

train['Comment'] = train['Comment'].replace(r'\s+|\\n', ' ', regex=True) # replaces new line characters in training data

Let’s look at the first 5 comments in the training data.

train.head() # See first 5 training sentences

**OUTPUT:**

|  |  |  |
| --- | --- | --- |
|  | Comment | Topic |
| 0 | A few things. You might have negative- frequency dependent selection going on where the least common phenotype, reflected by genotype, is going to have an advantage in the environment. For instance, if a prey animal such as a vole were to have a light and a dark phenotype, a predator might recognize the more common phenotype as food. So if the light voles are more common, foxes may be keeping a closer eye out for light phenotypic voles, recognising them as good prey. This would reduce the light causing alleles due to increased predation and the dark genotypes would increase their proportion of the population until this scenario is reversed. This cycle continues perpetually. However, this is unlikely to be strictly yearly as it usually takes more time than a year for an entire populations allele frequencies to change enough to make a large enough difference to alter fitness. More likely on a \*year to year\* basis, the population is experiencing fluctuating selection where alternating conditions in the environment favor one genotype over another. Perhaps a plant species is living in an area that is flooded every other year and the two phenotypes in the population are plants that do much better in the dryer year and one that does better in the wet year. If there is no flooding, the dry-type genotype will have more fitness leading to more offspring and therefore more dry alleles in the population, however, in flooded years the wet-liking phenotype will do better and propagate the wet genes. | Biology |
| 1 | Is it so hard to believe that there exist particulars out that that we can't detect with anything we've invented so far. I mean look how long it took humans to find out a way to detect radiation. | Physics |
| 2 | There are bees | Biology |
| 3 | I'm a medication technician. And that's alot of drugs on your liver. You probably won't die immediately you'll be fine. Take care of your self tho that's definitely not good for your body | Biology |
| 4 | Cesium is such a pretty metal. | Chemistry |

Next, we read in the testing data.

test = pd.read\_csv(  
 'science\_test.csv',   
 usecols=['Comment', 'Topic'],   
 dtype={'Comment': str, 'Topic': str}  
)

It is confirmed that there are 1,586 comments in the testing data.

test.shape # 1586 sentences

(1586, 2)

test.info() # No null values

<class 'pandas.core.frame.DataFrame'>  
RangeIndex: 1586 entries, 0 to 1585  
Data columns (total 2 columns):  
 # Column Non-Null Count Dtype   
--- ------ -------------- -----   
 0 Comment 1586 non-null object  
 1 Topic 1586 non-null object  
dtypes: object(2)  
memory usage: 24.9+ KB

Next, we replace new-line characters in the testing data with blank spaces using the same regular expression used for the training data.

test['Comment'] = test['Comment'].replace(r'\s+|\\n', ' ', regex=True) # replace new line characters with empty space

Next, we can look at the first 5 sentences in the testing data, along with the topics they correspond to.

test.head() # See first 5 sentences in test data

**OUTPUT:**

|  |  |  |
| --- | --- | --- |
|  | Comment | Topic |
| 0 | Personally I have no idea what my IQ is. I’ve never been tested. However, the test is an outdated, inaccurate, inappropriate measuring tool that has been largely abandoned by actual science. Only Mensa cares and their members tend to be insufferable misogynistic and racist assholes. So. Ya know. Go off I guess? | Biology |
| 1 | I'm skeptical. A heavier lid would be needed to build pressure, while a lighter lid is needed to move a lot with the release of pressure. I feel like I am missing something here. | Physics |
| 2 | I think I have 100 cm of books on the subject. TL;DR: The problem of consciousness is universally acknowledged as one of the most important in science, tens of thousands of scientists have devoted their careers to chipping away at it, numerous Nobel laureates have turned from their original fields to tackle it, and to date, no one has solved it. I'd point you to the works of Gerald Edelman and Thomas Metzinger as the authors who seemed closest to global theories, but that may simply be because I found their writings the most confounding. It may be possible that humans lack the language and hence mental tools to efficiently share concepts, think cogently, and progress in this field. It's possible that we could build supercomputer models of brains that have outputs that appear to have most traits of consciousness, and still not be able to understand it. | Biology |
| 3 | Is chemistry hard in uni. Ive read somewhere that its the hardest degree. But ive really been enjoying chem in high school right now, and want to do something involving science after high school. | Chemistry |
| 4 | In addition to the other comment, you can criticize a theory without checking off lots of "crackpot" indicators, like claiming that there's a vast conspiracy, comparing the current state of affairs to geocentrism or ponzi schemes, etc etc ([here's](https://iopscience.iop.org/article/10.1088/0031-8949/80/03/035702/meta?casa\_token=G851wji16m8AAAAA:dsiYA7-bePp2XHTZbQFTBV9XgHRj3E0GQBscLtU\_9pyfM2M7oa\_c7R2wtdNlFcjHWk2drTAeoH0) another one of Hirsch's extremely unhinged papers). And also, the hydrides are a class of material where BCS applies. | Physics |

As mentioned before, there is a lot of cleaning that needs done to improve the training process. We can see in the last comment printed in the ‘head’ of the testing data that there is a long URL that does not reveal much to the model to suggest which subject the comment is about. Fortunately, regular expressions can help specify different items that can be removed from the comments to get rid of unnecessary noise. In the following code, URLs, the ’@’ symbol, emojis, symbols, and stop words, like “a”, and “the” are removed. Also, words are reduced to their stems to eliminate extra suffixes using the SnowballStemmer from nltk. Finally, punctuation marks are removed since there are redundant question marks and periods in the comments. Also, numbers are removed. The code joins together the text after each operation is performed to remove what is specified in the regular expression.

# removes urls  
def remove\_url(sentence):  
 url = re.compile(r'https?://\S+|www\.\S+')  
 return url.sub(r'', sentence)

# remove '@' tags  
def remove\_at(sentence):  
 url = re.compile(r'@\S+')  
 return url.sub(r'', sentence)

# remove emojis, symbols, and pictographs  
  
def remove\_emoji(sentence):  
 emoji\_pattern = re.compile("["  
 u"\U0001F600-\U0001F64F"   
 u"\U0001F300-\U0001F5FF"   
 u"\U0001F680-\U0001F6FF"   
 u"\U0001F1E0-\U0001F1FF"   
 u"\U00002702-\U000027B0"  
 u"\U000024C2-\U0001F251"  
 "]+", flags=re.UNICODE)  
   
 return emoji\_pattern.sub(r'', sentence)

# remove common stop words  
def remove\_stopwords(sentence):  
 words = sentence.split()  
 words = [word for word in words if word not in stopwords.words('english')]  
   
 return ' '.join(words)

# revert words to stems  
stemmer = SnowballStemmer('english')  
  
def stem\_words(sentence):  
 words = sentence.split()  
 words = [stemmer.stem(word) for word in words]  
   
 return ' '.join(words)

# remove punctuation  
def punct\_remove(sentence):  
 punct = re.compile('[^\w\s]+')  
   
 return punct.sub(r'', sentence)

# remove numbers   
def num\_remove(sentence):  
 numb = [num for num in sentence if not num.isdigit()]  
   
 return ''.join(numb)

In this chunk, all of the functions are applied at once to create a function called ‘clean.’ In the next step, the ‘clean’ function will be applied to the ‘train’ and ‘test’ objects previously created.

# Apply all the above cleaning functions  
def clean(data):  
 data['Comment'] = data['Comment'].apply(lambda x : remove\_url(x))  
 data['Comment'] = data['Comment'].apply(lambda x : remove\_at(x))  
 data['Comment'] = data['Comment'].apply(lambda x : remove\_emoji(x))  
 data['Comment'] = data['Comment'].apply(lambda x : remove\_stopwords(x))  
 data['Comment'] = data['Comment'].apply(lambda x : stem\_words(x))  
 data['Comment'] = data['Comment'].apply(lambda x : punct\_remove(x))  
 data['Comment'] = data['Comment'].apply(lambda x : num\_remove(x))  
 return(data)

We replace what was previously ‘train’ and ‘test’ with the same names after the ‘clean’ function is applied to the comments in the training and testing data.

# Create cleaned training and test sets  
train = clean(train)  
test = clean(test)

Since multiple elements were replaced with white spaces, the data currently has spaces in spots where URLs once were, for example. We can eliminate this space as well, by replacing larger spaces with just a single space. We can also remove white space at the beginning and end of comments with the str.strip() method. This is done for training and testing data.

train['Comment'].replace('\s+', ' ', regex=True, inplace=True) # replaces spaces with a single space  
train['Comment'].str.strip() # removes leading and trailing spaces  
  
test['Comment'].replace('\s+', ' ', regex=True, inplace=True)  
test['Comment'].str.strip()

Next, some semi-colons and colons could be replaced with blank spaces to further clean the data.   
  
#replace colons and semi colons with empty space  
train['Comment'].replace(':', '', inplace = True)  
test['Comment'].replace(';', '', inplace = True)

Now, we can see what the comments appear as without extra symbols, punctuation, numbers, etc. Also, the text will no longer read as smoothly since words were broken down into their stems or bases, but this should help the model learn.

**Training Data:**

train.head() # First 5 rows of cleaned training data

**OUTPUT:**

|  |  |  |
| --- | --- | --- |
|  | Comment | Topic |
| 0 | a things you might negative frequenc depend select go least common phenotype reflect genotype go advantag environment for instance prey anim vole light dark phenotype predat might recogn common phenotyp food so light vole common fox may keep closer eye light phenotyp voles recognis good prey this would reduc light caus allel due increas predat dark genotyp would increas proport popul scenario reversed this cycl continu perpetually however unlik strict year usual take time year entir popul allel frequenc chang enough make larg enough differ alter fitness more like year year basis popul experienc fluctuat select altern condit environ favor one genotyp another perhap plant speci live area flood everi year two phenotyp popul plant much better dryer year one better wet year if flooding drytyp genotyp fit lead offspr therefor dri allel population however flood year wetlik phenotyp better propag wet genes | Biology |
| 1 | is hard believ exist particular cant detect anyth wev invent far i mean look long took human find way detect radiation | Physics |
| 2 | there bee | Biology |
| 3 | im medic technician and that alot drug liver you probabl die immedi fine take care self tho that definit good bodi | Biology |
| 4 | cesium pretti metal | Chemistry |

**Testing Data**

test.head() # First 5 rows of cleaned testing data

**OUTPUT:**

|  |  |  |
| --- | --- | --- |
|  | Comment | Topic |
| 0 | person i idea iq is iv never tested however test outdated inaccurate inappropri measur tool larg abandon actual science onli mensa care member tend insuffer misogynist racist assholes so ya know go i guess | Biology |
| 1 | im skeptical a heavier lid would need build pressure lighter lid need move lot releas pressure i feel like i miss someth here | Physics |
| 2 | i think i cm book subject tldr the problem conscious univers acknowledg one import science ten thousand scientist devot career chip away it numer nobel laureat turn origin field tackl it date one solv it id point work gerald edelman thoma metzing author seem closest global theories may simpli i found write confounding it may possibl human lack languag henc mental tool effici share concepts think cogently progress field it possibl could build supercomput model brain output appear trait consciousness still abl understand it | Biology |
| 3 | is chemistri hard uni ive read somewher hardest degree but ive realli enjoy chem high school right now want someth involv scienc high school | Chemistry |
| 4 | in addit comment critic theori without check lot crackpot indicators like claim there vast conspiracy compar current state affair geocentr ponzi schemes etc etc heres anoth one hirsch extrem unhing papers and also hydrid class materi bcs applies  Until now, the training and testing set have consisted of both comments and the topics that the comments relate to. When we train the model, we want to have the comments assigned to an “X” variable, and the subjects (or classes) assigned to a “y” variable.  This chunk assigns just the comments to be classified as ‘X\_train’ or ‘X\_test’ | Physics |

# Assign just the sentences to training and testing 'X'  
X\_train = train['Comment']  
X\_test = test['Comment']

Now, the subjects are assigned to objects called ‘classify\_train’ and ‘classify\_test.’

# Assign the subject column of 'Biology', 'Chemistry' or 'Physics' to 'classify\_train' and 'classify\_test'  
classify\_train = train['Topic']  
classify\_test = test['Topic']

**INPUT:**

classify\_train.head()

**OUTPUT:**

0 Biology  
1 Physics  
2 Biology  
3 Biology  
4 Chemistry  
Name: Topic, dtype: object

**INPUT:**

classify\_test.head()

**OUTPUT:**

0 Biology  
1 Physics  
2 Biology  
3 Chemistry  
4 Physics

# The length of the testing data is still the same size.

len(test)

**OUTPUT:**

1586

For our classification task, we will want the response to be in the form of one-hot encoding. Each response will have either a 0 or 1 in the spot for ‘biology’, ‘chemistry’, or ‘physics’ within an array. Since each comment only corresponds to one subject, the sum of the row of the one-hot encoded arrays will be one.

We can use the ‘get\_dummies’ method from Pandas to do this.

y\_train = pd.get\_dummies(classify\_train).values #Convert to one hot encoding form  
y\_test = pd.get\_dummies(classify\_test).values

Let’s see how this looks.

y\_train # 0 = biology, 1 = chemistry, 2 = physics

**OUTPUT:**

array([[ True, False, False],  
 [False, False, True],  
 [ True, False, False],  
 ...,  
 [False, True, False],  
 [ True, False, False],  
 [ True, False, False]])

**INPUT:**

y\_test

**OUTPUT:**

array([[ True, False, False],  
 [False, False, True],  
 [ True, False, False],  
 ...,  
 [False, True, False],  
 [ True, False, False],  
 [False, False, True]])

Evidently, we now have our response in the form of arrays with Boolean inputs corresponding to whether or not the comment relates to each subject. Matching these with the training and test data output from above, we can see that the first column (index 0) corresponds to biology, the second (index 1) to chemistry, and the third (index 2) to physics.

Whenever we do classification, we will want to see performance on a validation set at the same time as the training data. This helps us be keen on any overfitting that may be occurring. If we see that the validation error is not decreasing, or the validation accuracy is not improving, while the training data approaches very high accuracy or low loss, we can assume that some degree of overfitting is taking place.

The current testing set will be split so that the validation set is 20% of the original testing set and the remaining 80% of the testing set becomes the ‘new’ testing set.

# Split so that 20% of the original testing set becomes validation set; remaining 80% stays as testing set  
  
from sklearn.model\_selection import train\_test\_split  
X\_test, X\_validate, y\_test, y\_validate = train\_test\_split(  
X\_test, y\_test, random\_state=42, test\_size=0.20)

X\_train.shape

**OUTPUT:**

(8695,)

The training set keeps the same shape, which makes sense. It was not changed when creating the validation set.

The validation set is 0.2\*1,586 = 317.2, which rounds up to 318.

X\_validate.shape

**OUTPUT:**

(318,)

The testing data is 0.8\*1,586 = 1,268.8. Since the validation set rounded up, the testing data rounds down to consist of 1,268 comments.

X\_test.shape

**OUTPUT:**

(1268,)

from nltk import word\_tokenize  
  
from keras.utils import pad\_sequences  
from keras.layers import Input, Dense, LSTM, Embedding  
from keras.layers import Dropout, Activation, Bidirectional, GlobalMaxPool1D  
from keras.models import Sequential  
from keras import initializers, regularizers, constraints, optimizers, layers  
from keras.preprocessing import text, sequence

Since the model will not be able to train with sentences in the form of strings, each string needs to be converted to a form of numbers representing the letters/words. Using a tokenizer assigns a numerical value to each word. The first part of this code combines all the comments from the training, test, and validation sets so that each sentence is encoded. Then, the tokenizer is used to encode each string of words in each sentence into an array of numbers that the model interprets as the sentence. To ensure that each sentence is the same size, padding is used to fill in zeros where necessary to ensure that shorter sentences are the same size as longer ones.

# Create function for tokenizer  
def tokenizer\_create(train, val, test):  
 together = pd.concat([train, val, test])  
   
 tokenizer = tf.keras.preprocessing.text.Tokenizer()  
 tokenizer.fit\_on\_texts(together)  
   
 return tokenizer  
  
def encode (together, tokenizer):  
 encoded = tokenizer.texts\_to\_sequences(together)  
 encoded = tf.keras.utils.pad\_sequences(encoded, padding = 'post')  
   
 return encoded

The previous code just defined the function. However, we need to use the ‘tokenizer\_create’ function on each set of our data. So, objects called ‘encoded\_train’, ‘encoded\_test’, and ‘encoded\_validate’ are created with the content being the encoded sentences after using the tokenizer. Now, the sentences are no longer a series of letters to the model that it cannot interpret. Rather, each sentence is now interpreted as an array of numbers.

tokenizer = tokenizer\_create(X\_train, X\_validate, X\_test)   
  
encoded\_train = encode(X\_train, tokenizer)  
encoded\_test = encode(X\_test, tokenizer)  
encoded\_validate = encode(X\_validate, tokenizer)

Another feature of the tokenizer is that the numbers relating to each word can be looked at in the ‘word\_index’ of the tokenizer. We can also see that, in this case, there are 18,072 words in the vocabulary for this classification task.

len(tokenizer.word\_index)

18072 # Number of words in the dictionary.

An embedding layer will help with establishing deeper meaning in the text, regarding things such as connotation and relationships between words. We can initialize the weights for the embedding layer using a pre-trained method called GloVe Embedding. A text file can be opened in Python to do this.

# Set up initialized for embedding layer  
  
embedding\_dict = {}  
  
with open('glove.6B.100d.txt','r', encoding="utf8") as f:  
 for line in f:  
 values = line.split()  
 word = values[0]  
 vectors = np.asarray(values[1:],'float32')  
 embedding\_dict[word] = vectors  
   
f.close()

The above code breaks down the text file, which consists of numbers corresponding to certain words, and assigns the embedding dictionary to these values. The code following creates a blank array with number of rows equal to number of words in the vocabulary (18072) plus 1 = 18,073. The number of columns equals 100. For each word, a vector of 100 values is assigned to it, coming from the GloVe text file.

In [1298]:

num\_words = len(tokenizer.word\_index) + 1  
embedding\_matrix = np.zeros((num\_words, 100))  
  
for word, i in tokenizer.word\_index.items():  
 emb\_vec = embedding\_dict.get(word)  
 if emb\_vec is not None:  
 embedding\_matrix[i] = emb\_vec

#Convert each set of comment data to np arrays. They were previously Pandas Data Frames.

X\_train = np.array(X\_train)  
X\_validate = np.array(X\_validate)  
X\_test = np.array(X\_test)

Next, we will build the RNN. We will add layers on since we used the Sequential() set-up from Keras. The embedding layer has input dimension equal to the length of the word index plus one (18,073), and an output layer equal to 100. The output dimension must be 100 from the number of columns in the embedding\_matrix. The initializer of the embedding layer is the embedding matrix that was created in the previous code.

After the embedding layer, we add a bidirectional LSTM layer with 64 nodes. This uses two LSTM layers. One processes text in the forward direction and the other processes text in the backward direction. This helps the model understand context further and hopefully understand that the same words can be used in different meanings or environments. We use a dropout rate of 0.2 for both the inputs and the recurrent states. After this, we add additional dropout layers with a rate of 0.2 and two dense layers with 50 nodes and the ‘relu’ activation function. This helps perform additional training and improves the model’s performance. The final dense layer contains 3 units because we have 3 subjects that we are trying to classify text into. The ‘softmax’ activation is used because this is classification of multiple categories using one-hot encoding.

model = Sequential()

model.add(Embedding(input\_dim = (len(tokenizer.word\_index)+1), output\_dim = 100,   
 embeddings\_initializer = tf.keras.initializers.Constant(embedding\_matrix), trainable = True))  
model.add(tf.keras.layers.Bidirectional(LSTM(64, dropout=0.2, recurrent\_dropout=0.2)))  
model.add(Dropout(0.2))  
model.add(Dense(50, activation = 'relu'))  
model.add(Dropout(0.2))  
model.add(Dense(50, activation = 'relu'))  
model.add(Dropout(0.2))  
model.add(Dense(3, activation='softmax'))

An output of the model’s structure follows. We can see the shape change from 100 to 128 in the bidirectional layer. Since 64 was specified as the units in the bidirectional LSTM layer, the last value in the shape for this layer is double this at 128, since input is fed in both the forward and backward direction.

model.summary()

Model: "sequential\_44"  
\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_  
 Layer (type) Output Shape Param #   
=================================================================  
 embedding\_39 (Embedding) (None, None, 100) 1807300   
   
 bidirectional\_5 (Bidirectio (None, 128) 84480   
 nal)   
   
 dropout\_72 (Dropout) (None, 128) 0   
   
 dense\_91 (Dense) (None, 50) 6450   
   
 dropout\_73 (Dropout) (None, 50) 0   
   
 dense\_92 (Dense) (None, 50) 2550   
   
 dropout\_74 (Dropout) (None, 50) 0   
   
 dense\_93 (Dense) (None, 3) 153   
   
=================================================================  
Total params: 1,900,933  
Trainable params: 1,900,933  
Non-trainable params: 0  
\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Next, we compile the model. The categorical crossentropy loss function is used for this multi-classification problem. I used the ‘Nadam’ optimizer with an initial learning rate of 0.001. The metrics I chose to follow were accuracy, precision, and recall, from which F1 score can be computed.

model.compile(loss = 'categorical\_crossentropy',   
 optimizer=tf.keras.optimizers.legacy.Nadam(0.001),   
 metrics=['accuracy', 'Precision', 'Recall'])

I used a callback to monitor the validation loss, since LSTM can easily overfit. I wanted to monitor the validation loss so that if the validation loss increases over more than 2 epochs, the learning rate will be reduced.

callback = [  
 tf.keras.callbacks.ReduceLROnPlateau(monitor='val\_loss', patience=2, verbose=1)  
]

Now, we fit the model. We must put ‘encoded\_train’ in the position for ‘X’ because this is the numerical array input representing each word that results from the tokenizer. We cannot feed the strings into the model fit. We will fit over 5 epochs for the sake of run-time, using the validation set for comparison to get an idea of how the model would perform on unseen data.

history = model.fit(encoded\_train, y\_train, epochs = 5, validation\_data=(encoded\_validate, y\_validate))

Epoch 1/5  
272/272 [==============================] - 183s 659ms/step - loss: 0.9619 - accuracy: 0.5282 - precision: 0.6890 - recall: 0.2489 - val\_loss: 0.6548 - val\_accuracy: 0.7201 - val\_precision: 0.7742 - val\_recall: 0.6792  
Epoch 2/5  
272/272 [==============================] - 196s 719ms/step - loss: 0.7583 - accuracy: 0.6641 - precision: 0.7588 - recall: 0.5338 - val\_loss: 0.5431 - val\_accuracy: 0.7799 - val\_precision: 0.8194 - val\_recall: 0.7421  
Epoch 3/5  
272/272 [==============================] - 194s 714ms/step - loss: 0.6414 - accuracy: 0.7318 - precision: 0.8027 - recall: 0.6400 - val\_loss: 0.4926 - val\_accuracy: 0.8365 - val\_precision: 0.8426 - val\_recall: 0.8082  
Epoch 4/5  
272/272 [==============================] - 194s 715ms/step - loss: 0.5441 - accuracy: 0.7795 - precision: 0.8347 - recall: 0.7135 - val\_loss: 0.5702 - val\_accuracy: 0.8019 - val\_precision: 0.8103 - val\_recall: 0.7925  
Epoch 5/5  
272/272 [==============================] - 195s 716ms/step - loss: 0.4654 - accuracy: 0.8115 - precision: 0.8590 - recall: 0.7652 - val\_loss: 0.5654 - val\_accuracy: 0.8333 - val\_precision: 0.8424 - val\_recall: 0.8239

**INPUT:**

metrics = model.evaluate(encoded\_test, y\_test)

40/40 [==============================] - 2s 37ms/step - loss: 0.5433 - accuracy: 0.8076 - precision: 0.8193 - recall: 0.7973

From an initial view, it appears that model accuracy is about 0.81.

**INPUT:**

accuracy = metrics[1]  
loss = metrics[0]  
precision = metrics[2]  
recall = metrics[3]  
f1 = 2 \* (precision \* recall) / (precision + recall)  
  
print('accuracy: ' + str(accuracy))  
print('loss: ' + str(loss))  
print('precision: ' + str(precision))  
print('recall: ' + str(recall))  
print('F1 score: ' + str(f1))

accuracy: 0.8075709939002991  
loss: 0.5432597994804382  
precision: 0.8192868828773499  
recall: 0.7973186373710632  
F1 score: 0.8081534955681414

Model accuracy is about 0.81, and loss is at about 0.58. Precision is close to accuracy at about 0.83. Recall is about 0.80, and the F1 Score is about 0.82. The model is slightly better at having its classifications be correct than at identifying the comments that relate to each category. This is because precision is very slightly higher than recall.

We need to look at the validation metrics as well, so we can see how the model performs on data that it is not trained with, as good performance in that aspect is the ultimate goal of any model.

We can create 4 plots in one figure. One plot will be for accuracy, one for loss, one for precision, and one for recall.

fig, axs = plt.subplots(1, 4, figsize=(20, 5))  
  
axs[0].set\_title('Accuracy')  
axs[0].plot(history.history['accuracy'], label='train')  
axs[0].plot(history.history['val\_accuracy'], label='val')  
axs[0].legend()  
  
axs[1].set\_title('Loss')  
axs[1].plot(history.history['loss'], label='train')  
axs[1].plot(history.history['val\_loss'], label='val')  
axs[1].legend()  
  
axs[2].set\_title('Precision')  
axs[2].plot(history.history['precision'], label='train')  
axs[2].plot(history.history['val\_precision'], label='val')  
axs[2].legend()  
  
axs[3].set\_title('Recall')  
axs[3].plot(history.history['recall'], label='train')  
axs[3].plot(history.history['val\_recall'], label='val')  
axs[3].legend()

**OUTPUT:**

![](data:image/png;base64;base64,)

These plots show that overfitting is not too terrible of a concern here. The training accuracy increases without a simultaneous dramatic decrease in validation accuracy. Overall, the loss for both the training and validation data decreases over time, although the last 2 epochs do see the validation accuracy, although there is a slight decrease in validation accuracy between epochs 3 and 4. The training loss steadily decreases, however, the validation loss decreases at first, but starts to increase at the 3rd epoch. However, we cannot observe the validation loss increasing for more than two epochs because the model was only fit to 5 epochs. As such, the decreasing learning rate was never called, which would decrease the learning rate if validation loss increased for more than two epochs. If the model was fit to more than 5 epochs, it is possible that the learning rate was adjusted from the callback. The precision for the training data increased over all the epochs, looking like the training loss but in the opposite direction. The validation precision started out higher than training precision, but eventually decreased below training precision, before increasing to end close to the training precision. The recall of the validation data also started higher than the training data and did not decrease drastically. The training recall increased for the most part over all the epochs. Of course, more epochs could be conducted with learning rate adjustments to hopefully smooth out the model and improve accuracy, but this model shows overall sufficient accuracy with no extremely concerning patterns.

To conclude, a recurrent neural network was created to classify comments as relating to biology, chemistry, or physics. Because pre-trained embedding was used, the model may have learned words that it was not trained with but that it could be required to classify with testing, unseen data. This helped with preventing a large degree of overfitting, as well as the addition of dropout layers. A tokenizer was used to encode sentences as arrays of numbers, which were first passed through an embedding layer, and then through a bidirectional long-short term memory layer. The embedding layer helped the model pick up on word meanings. The LSTM layer helped the model learn from inputs of the sentences from left to right and from right to left. Following these layers, an alternation of dropout and standard dense layers strives to help the model generalize well to unseen data. This procedure resulted in an accuracy of around 0.81, which is sufficient, but additional tunings could likely increase this value.