# Text Classfication Two

## Part One: Data Exploration

### 1.1 - Describing Data

Hello! This notebook aims to complete text classification with keras and other deep learning techniques with Keras. For more tradititional machine learning models, be sure to look for the precursor to this notebook, aptly titled 'text classification'.

We'll be performing text classification on an e-commerce based dataset found on Kaggle.com. Please, take a look at it here.

https://www.kaggle.com/datasets/saurabhshahane/ecommerce-text-classification

This dataset is comprised of a classification of an e-commerce item based on the following four classes - stated by the author of the dataset that these classes "cover(s) 80% of any E-commerce website."

- Electronics
- Household
- Books
- Clothing

Additionally, the only other feature the .csv file offers is a description of said item.

```
import pandas as pd
import numpy as np
from google.colab import drive
drive.mount('/content/drive')

ecom = '/content/drive/My Drive/ecommerceDataset.csv'
#csv file doesn't list header, want to change for vanity reasons
#edited on round 1 of tests - now we've changed the csv file so this block is irrelevant.
#df = pd.read_csv(ecom)
#df.columns = ['Item_type', 'Description']
#df.to_csv(ecom)

df2 = pd.read_csv(ecom)
#get rid of NAs they actively try to ruin my life why are these datasets never clean
df2 = df2.dropna(subset=['Description'])
print(df2.head())
print("\n", df2.shape)

#attribute counts
item_class_count = df2['Item_type'].value_counts()
print("\n", item_class_count)
```

```
Mounted at /content/drive
   Unnamed: 0  Item_type                                        Description
0           0  Household  SAF 'Floral' Framed Painting (Wood, 30 inch x ...
1           1  Household  SAF 'UV Textured Modern Art Print Framed' Pain...
2           2  Household  SAF Flower Print Framed Painting (Synthetic, 1...
3           3  Household  Incredible Gifts India Wooden Happy Birthday U...
4           4  Household  Pitaara Box Romantic Venice Canvas Painting 6m...

 (50423, 3)

 Household                 19312
Books                     11820
Electronics               10621
Clothing & Accessories     8670
Name: Item_type, dtype: int64
```

I have some concerns with the dataset, mostly that it's imbalanced. We have nearly double the amount of household items than we do electronic items. Luckily the Books, Electronics and to a lesser extent the Clothing sections are relatively similar in their counts.

Additonally, due to the complexity of deep learning algorithms and the poor performance and specfications of my five year old dusty hp notebook, we'll cull our data a bit to make sure we can at least run something.

```
grouped = df2.groupby('Item_type', group_keys=False).apply(lambda x: x.sample(min(len(x), 2500)))

output_sampled = '/content/drive/My Drive/ecommerceDatasetSampled.csv'
grouped.to_csv(output_sampled, index=False)
```

```
#attribute counts
item_class_count_sample = grouped['Item_type'].value_counts()
print("\n", item_class_count_sample)
```

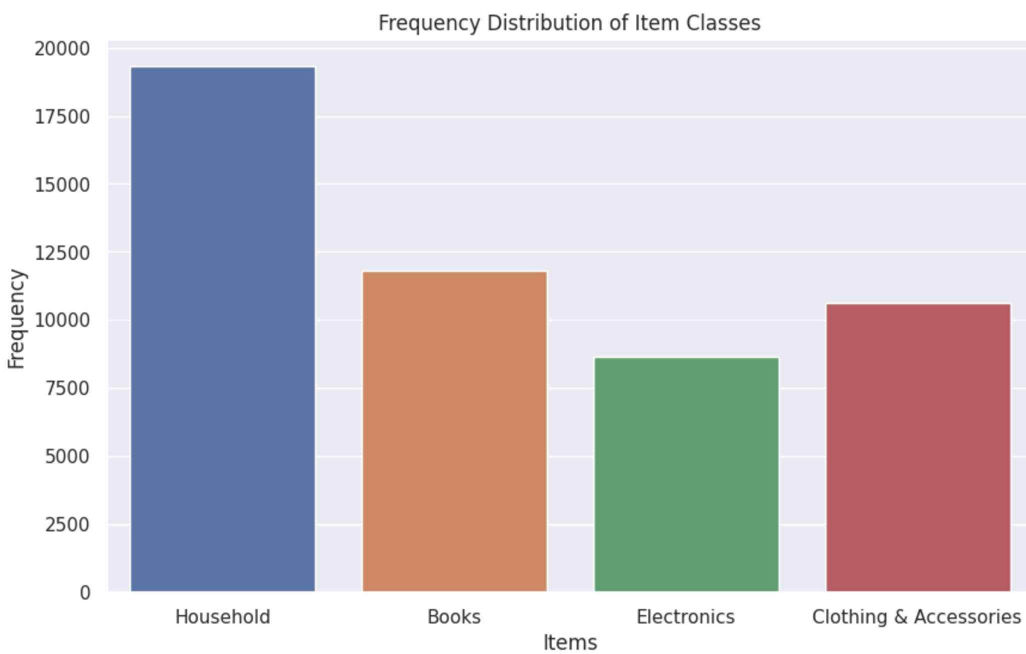
```
 Books                     2500
Clothing & Accessories    2500
Electronics               2500
Household                 2500
Name: Item_type, dtype: int64
```

What we've done above is use the .sample() method from the pandas library to randomly select or sample 2500 entries within each class to trim down our dataset. From here we'll preprocess and divide into train/test for future predictions and evaluations on our model. However before that, let's map out some graphs of our target distributions.

```
import seaborn as sb
import matplotlib.pyplot as plt

sb.set(style = "darkgrid")
plt.figure(figsize=(10, 6))
ax =sb.countplot(x='Item_type', data = df2)
ax.set_xticklabels(item_class_count.keys())
ax.set_title('Frequency Distribution of Item Classes')
ax.set_xlabel('Items')
ax.set_ylabel('Frequency')

plt.show()
```




```
#Preprcess sampled dataset
import nltk
from nltk.corpus import stopwords
from nltk.tokenize import word_tokenize
nltk.download('stopwords')
nltk.download('punkt')
def preprocess(text):
    stop_words = set(stopwords.words('english'))
    words = word_tokenize(text)
    processed = [word for word in words if word not in stop_words or word.isupper()]
    return ' '.join(processed)

grouped['Description'] = grouped['Description'].apply(preprocess)
grouped.to_csv(output_sampled, index = False)
```

```
[nltk_data] Downloading package stopwords to /root/nltk_data...
[nltk_data]   Unzipping corpora/stopwords.zip.
[nltk_data] Downloading package punkt to /root/nltk_data...
[nltk_data]   Unzipping tokenizers/punkt.zip.
```

Rounding out the first section, we'll create a train/test split and go on to building some models.

```
from sklearn.model_selection import train_test_split

#y = target = class, or what we want to predict.
x = grouped.Description
y = grouped.Item_type
x_train, x_test, y_train, y_test = train_test_split(x, y, test_size= 0.2, random_state= 1234)

#peeking at our x and y
print(x.head())
y[:10]
```

```
28109    The Atlas Beauty About Author Mihaela Noroc li...
29103    Essentials Nursing Research - Appraising Evide...
29344    Manipal Prep Manual Medicine It thoroughly rew...
29120    Dermoscopy : Text Atlas About Author Subrata M...
21797                                    Physics JEE Main
Name: Description, dtype: object
28109    Books
29103    Books
29344    Books
29120    Books
21797    Books
27746    Books
21932    Books
19806    Books
24717    Books
23646    Books
Name: Item_type, dtype: object
```

## Part Two: A Sequential Model

```
#prepare x and y
import tensorflow as tf
from tensorflow.keras.preprocessing.text import Tokenizer
from tensorflow.keras import layers, models
from tensorflow.keras.utils import to_categorical
from sklearn.preprocessing import LabelEncoder

num_labels = 4 #number of labels in dataset - household, clothing, electronics, books
vocab_size = 35000 #size of vocab of dataset. since we're looking at reviews across departments, this needs to be large.
batch_size = 250 #train on 250 samples for each iteration - 0.25% of the whole dataset

#tokenize
tokenizer = Tokenizer(num_words=vocab_size)
tokenizer.fit_on_texts(x_train)

#convert to matrix
x_train_matrix = tokenizer.texts_to_matrix(x_train, mode='tfidf')
x_test_matrix = tokenizer.texts_to_matrix(x_test, mode='tfidf')

encoder = LabelEncoder()
y_train_labels = encoder.fit_transform(y_train)
y_test_labels = encoder.fit_transform(y_test)
#conv to categorical
y_train_cat = to_categorical(y_train_labels)
y_test_cat = to_categorical(y_test_labels)
```

Above I set some variables and ellaborated on why I set them like that in the comments. num_labels = 4: This is because we're doing multiclass classification between four classes.

vocab_size = 35,000: This variable should be bigger than normal because of the amount of different classes and each unique word they have in their respective domains.

batch_size = 250: This one was set almost arbitrarily, and I did it mostly in the interest of saving computation time due to model complexity.

```
#PREPARE THE MODEL.
model = models.Sequential()
model.add(layers.Dense(32, input_dim=vocab_size, kernel_initializer='normal', activation='relu'))
model.add(layers.Dense(4, kernel_initializer='normal', activation='softmax')) #softmax for multi-classfication
#compile THE MODEL
model.compile(
    loss = 'categorical_crossentropy', #again, for multi-class
    optimizer = 'adam', #malkovic
    metrics = ['accuracy']
)

history = model.fit(x_train_matrix, y_train_cat,
  batch_size = batch_size,
  epochs = 30,
  verbose = 1,
  validation_split = 0.1)
```

```
Epoch 1/30
29/29 [==============================] - 6s 47ms/step - loss: 0.9946 - accuracy: 0.6543 - val_loss: 0.5915 - val_accuracy:
Epoch 2/30
29/29 [==============================] - 1s 33ms/step - loss: 0.3880 - accuracy: 0.9407 - val_loss: 0.3686 - val_accuracy:
Epoch 3/30
29/29 [==============================] - 1s 46ms/step - loss: 0.2034 - accuracy: 0.9749 - val_loss: 0.2935 - val_accuracy:
Epoch 4/30
29/29 [==============================] - 2s 69ms/step - loss: 0.1225 - accuracy: 0.9882 - val_loss: 0.2606 - val_accuracy:
Epoch 5/30
29/29 [==============================] - 1s 51ms/step - loss: 0.0793 - accuracy: 0.9944 - val_loss: 0.2470 - val_accuracy:
Epoch 6/30
29/29 [==============================] - 1s 42ms/step - loss: 0.0548 - accuracy: 0.9976 - val_loss: 0.2415 - val_accuracy:
Epoch 7/30
29/29 [==============================] - 1s 29ms/step - loss: 0.0395 - accuracy: 0.9989 - val_loss: 0.2444 - val_accuracy:
Epoch 8/30
29/29 [==============================] - 1s 31ms/step - loss: 0.0308 - accuracy: 0.9992 - val_loss: 0.2410 - val_accuracy:
Epoch 9/30
29/29 [==============================] - 1s 29ms/step - loss: 0.0236 - accuracy: 0.9996 - val_loss: 0.2443 - val_accuracy:
Epoch 10/30
29/29 [==============================] - 1s 30ms/step - loss: 0.0190 - accuracy: 0.9994 - val_loss: 0.2452 - val_accuracy:
Epoch 11/30
29/29 [==============================] - 1s 28ms/step - loss: 0.0154 - accuracy: 0.9994 - val_loss: 0.2493 - val_accuracy:
Epoch 12/30
29/29 [==============================] - 1s 29ms/step - loss: 0.0131 - accuracy: 0.9994 - val_loss: 0.2585 - val_accuracy:
Epoch 13/30
29/29 [==============================] - 1s 31ms/step - loss: 0.0118 - accuracy: 0.9994 - val_loss: 0.2567 - val_accuracy:
Epoch 14/30
29/29 [==============================] - 1s 32ms/step - loss: 0.0096 - accuracy: 0.9996 - val_loss: 0.2594 - val_accuracy:
Epoch 15/30
29/29 [==============================] - 1s 34ms/step - loss: 0.0082 - accuracy: 0.9996 - val_loss: 0.2617 - val_accuracy:
Epoch 16/30
29/29 [==============================] - 1s 48ms/step - loss: 0.0071 - accuracy: 0.9996 - val_loss: 0.2669 - val_accuracy:
Epoch 17/30
29/29 [==============================] - 1s 51ms/step - loss: 0.0064 - accuracy: 0.9997 - val_loss: 0.2677 - val_accuracy:
Epoch 18/30
29/29 [==============================] - 1s 47ms/step - loss: 0.0054 - accuracy: 0.9997 - val_loss: 0.2710 - val_accuracy:
Epoch 19/30
29/29 [==============================] - 1s 37ms/step - loss: 0.0054 - accuracy: 0.9997 - val_loss: 0.2736 - val_accuracy:
Epoch 20/30
29/29 [==============================] - 1s 34ms/step - loss: 0.0045 - accuracy: 0.9997 - val_loss: 0.2789 - val_accuracy:
Epoch 21/30
29/29 [==============================] - 1s 34ms/step - loss: 0.0044 - accuracy: 0.9997 - val_loss: 0.2791 - val_accuracy:
Epoch 22/30
29/29 [==============================] - 1s 30ms/step - loss: 0.0041 - accuracy: 0.9997 - val_loss: 0.2815 - val_accuracy:
Epoch 23/30
29/29 [==============================] - 1s 31ms/step - loss: 0.0038 - accuracy: 0.9997 - val_loss: 0.2842 - val_accuracy:
Epoch 24/30
29/29 [==============================] - 1s 31ms/step - loss: 0.0026 - accuracy: 1.0000 - val_loss: 0.2885 - val_accuracy:
Epoch 25/30
29/29 [==============================] - 1s 43ms/step - loss: 0.0031 - accuracy: 0.9999 - val_loss: 0.2910 - val_accuracy:
Epoch 26/30
29/29 [==============================] - 1s 33ms/step - loss: 0.0022 - accuracy: 0.9999 - val_loss: 0.2943 - val_accuracy:
Epoch 27/30
29/29 [==============================] - 1s 36ms/step - loss: 0.0032 - accuracy: 0.9999 - val_loss: 0.2929 - val_accuracy:
Epoch 28/30
29/29 [==============================] - 1s 33ms/step - loss: 0.0025 - accuracy: 0.9997 - val_loss: 0.2952 - val_accuracy:
Epoch 29/30
```

```
#evaluate
score = model.evaluate(x_train_matrix, y_train_cat, batch_size=batch_size, verbose=1)
print('Accuracy: ', score[1])
print('\n', score)
```

```
32/32 [==============================] - 1s 28ms/step - loss: 0.0322 - accuracy: 0.9950
Accuracy:  0.9950000047683716

 [0.03221438452601433, 0.9950000047683716]
```

```
#predict
y_pred = model.predict(x_test_matrix)
y_pred_labels = [np.argmax(pred) for pred in y_pred]
y_test_arr = np.array(y_test_labels)

from sklearn.metrics import accuracy_score, precision_score, recall_score, f1_score
accuracy = accuracy_score(y_test_arr, y_pred_labels)
precision = precision_score(y_test_arr, y_pred_labels, average='weighted')
recall = recall_score(y_test_arr, y_pred_labels, average='weighted')
f1 = f1_score(y_test_arr, y_pred_labels, average='weighted')

print(f"Accuracy: {accuracy:.4f}")
print(f"Precision: {precision:.4f}")
print(f"Recall: {recall:.4f}")
print(f"F1-Score: {f1:.4f}")

#loss to check overfit
loss, accuracy = model.evaluate(x_test_matrix, y_test_cat, verbose=1)
print('Loss:', loss)

test_loss, test_accuracy = model.evaluate(x_test_matrix, y_test_cat)
print('Test loss:', test_loss)
```

```
63/63 [==============================] - 0s 5ms/step
Accuracy: 0.9550
Precision: 0.9552
Recall: 0.9550
F1-Score: 0.9550
63/63 [==============================] - 0s 6ms/step - loss: 0.3360 - accuracy: 0.9550
Loss: 0.3359505236148834
63/63 [==============================] - 0s 5ms/step - loss: 0.3360 - accuracy: 0.9550
Test loss: 0.3359505236148834
```

We have fantastic accuracy! The issue is that I think we're getting such inflated accuracy numbers due to the model overfitting the training data. Looking at our loss values from the training data of 0.0459 for our trained model and comparing to the unseen test loss of 0.2876, we have some pretty huge overfit here.

I wonder if this is due to the vocabulary pool? Since we have a lot of different products covered, I'm wondering if expanding the vocab will have a better impact on our results?

## Part Three: A New Architecture - RNN and variations

RNN networks are best for sequential data, as the model references earlier information when processing later bits of data. Thus it can use previously processed information to inform it's results later down the line. This is done with a feedback loop and creates a memory for the model to refernce. This is different from what we had done earlier in the sequential model, as that was more or less a standard neural network that had a sequntial paradigm, processing data sequentially. Notably sequential models do not maintain an internal state that the model can look up and inform new decisions like the RNN pattern does.

```
from tensorflow.keras import preprocessing
from tensorflow.keras.preprocessing.text import Tokenizer
from tensorflow.keras.preprocessing.sequence import pad_sequences
max_features = 35000 #lots of words in item descriptions
maxlen = 1000 #want to encapsulate as much of the description as possible
batch_size = 48

#load data and pad
#y = target = class, or what we want to predict.
x = grouped.Description
y = grouped.Item_type
x_train_rnn, x_test_rnn, y_train_rnn, y_test_rnn = train_test_split(x, y, test_size= 0.2, random_state= 1234)


#tokenize for preprocessing
tokenizer = Tokenizer(num_words = max_features)
tokenizer.fit_on_texts(x_train_rnn)
```

```
#conv to int seq
x_train_rnn = tokenizer.texts_to_sequences(x_train_rnn)
x_test_rnn = tokenizer.texts_to_sequences(x_test_rnn)

x_train_rnn = preprocessing.sequence.pad_sequences(x_train_rnn, maxlen=maxlen)
x_test_rnn = preprocessing.sequence.pad_sequences(x_test_rnn, maxlen=maxlen)

#tocat the labels
y_train_labels_rnn = to_categorical(y_train_labels)
y_test_labels_rnn = to_categorical(y_test_labels)



#construct THE MODEL
model = models.Sequential()
model.add(layers.Embedding(max_features, 32))
model.add(layers.SimpleRNN(32))
model.add(layers.Dense(4, activation='softmax'))
model.summary()
```

```
Model: "sequential_1"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 embedding (Embedding)       (None, None, 32)          1120000

 simple_rnn (SimpleRNN)      (None, 32)                2080

 dense_2 (Dense)             (None, 4)                 132

=================================================================
Total params: 1,122,212
Trainable params: 1,122,212
Non-trainable params: 0
_________________________________________________________________
```

```
#compile the model
model.compile(optimizer='rmsprop',
              loss = 'categorical_crossentropy',
              metrics = ['accuracy'])
#train
history = model.fit(x_train_rnn,
                    y_train_labels_rnn,
                    epochs = 10, #computationaly complexity, each epoch takes 3 mins to compute
                    batch_size = 128,
                    validation_split = 0.2)
```

```
Epoch 1/10
50/50 [==============================] - 60s 1s/step - loss: 1.2452 - accuracy: 0.4489 - val_loss: 1.0544 - val_accuracy: 0.
Epoch 2/10
50/50 [==============================] - 59s 1s/step - loss: 0.8431 - accuracy: 0.7437 - val_loss: 0.7860 - val_accuracy: 0.
Epoch 3/10
50/50 [==============================] - 59s 1s/step - loss: 0.5469 - accuracy: 0.8761 - val_loss: 0.6611 - val_accuracy: 0.
Epoch 4/10
50/50 [==============================] - 52s 1s/step - loss: 0.3144 - accuracy: 0.9550 - val_loss: 0.6031 - val_accuracy: 0.
Epoch 5/10
50/50 [==============================] - 51s 1s/step - loss: 0.1679 - accuracy: 0.9781 - val_loss: 0.5947 - val_accuracy: 0.
Epoch 6/10
50/50 [==============================] - 49s 994ms/step - loss: 0.0816 - accuracy: 0.9919 - val_loss: 0.6248 - val_accuracy:
Epoch 7/10
50/50 [==============================] - 52s 1s/step - loss: 0.0449 - accuracy: 0.9950 - val_loss: 0.7256 - val_accuracy: 0.
Epoch 8/10
50/50 [==============================] - 61s 1s/step - loss: 0.0806 - accuracy: 0.9803 - val_loss: 0.7827 - val_accuracy: 0.
Epoch 9/10
50/50 [==============================] - 67s 1s/step - loss: 0.0297 - accuracy: 0.9962 - val_loss: 0.7243 - val_accuracy: 0.
Epoch 10/10
50/50 [==============================] - 71s 1s/step - loss: 0.0167 - accuracy: 0.9970 - val_loss: 0.7119 - val_accuracy: 0.
```

```
from sklearn.metrics import classification_report
from sklearn.preprocessing import MultiLabelBinarizer, label_binarize

#metric report
pred_rnn = model.predict(x_test_rnn)

binarizer = MultiLabelBinarizer()
binarizer.fit(y_test_labels_rnn)
y_test_labels_binary = binarizer.transform(y_test_labels_rnn)
pred_binary = binarizer.transform(label_binarize(pred_rnn.argmax(axis=1), classes=binarizer.classes_))
```

```
# generate classification report
print(classification_report(y_test_labels_binary, pred_binary))
```

```
63/63 [==============================] - 8s 119ms/step
              precision    recall  f1-score   support

           0       1.00      1.00      1.00      2000
           1       1.00      0.52      0.69      2000

   micro avg       1.00      0.76      0.86      4000
   macro avg       1.00      0.76      0.84      4000
weighted avg       1.00      0.76      0.84      4000
 samples avg       1.00      0.76      0.84      4000
```

Now we'll try a new variation on the model, as we've gotten much worse accuracy here. So now we'll try to make a model better suited to large sequential data (text) w/ LSTM instead of SimpleRNN. I've also found that LSTM runs much quicker than SimpleRNN.

```
#new model w/ LSTM
model = models.Sequential()
model.add(layers.Embedding(max_features, 32))
model.add(layers.LSTM(32))
model.add(layers.Dense(4, activation='softmax'))
model.summary()
```

```
Model: "sequential_2"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 embedding_1 (Embedding)     (None, None, 32)          1120000

 lstm (LSTM)                 (None, 32)                8320

 dense_3 (Dense)             (None, 4)                 132

=================================================================
Total params: 1,128,452
Trainable params: 1,128,452
Non-trainable params: 0
_________________________________________________________________
```

```
# comple
#compile the model
model.compile(optimizer='rmsprop',
              loss = 'categorical_crossentropy',
              metrics = ['accuracy'])
#train
history_lstm = model.fit(x_train_rnn,
                    y_train_labels_rnn,
                    epochs = 10, #computationaly complexity, each epoch takes 3 mins to compute
                    batch_size = 128,
                    validation_split = 0.2)
```

```
Epoch 1/10
50/50 [==============================] - 13s 190ms/step - loss: 1.2618 - accuracy: 0.4127 - val_loss: 1.0276 - val_accuracy:
Epoch 2/10
50/50 [==============================] - 9s 187ms/step - loss: 0.8647 - accuracy: 0.6359 - val_loss: 0.7336 - val_accuracy:
Epoch 3/10
50/50 [==============================] - 10s 195ms/step - loss: 0.6265 - accuracy: 0.7841 - val_loss: 0.5876 - val_accuracy:
Epoch 4/10
50/50 [==============================] - 9s 172ms/step - loss: 0.4963 - accuracy: 0.8619 - val_loss: 0.4260 - val_accuracy:
Epoch 5/10
50/50 [==============================] - 9s 182ms/step - loss: 0.3467 - accuracy: 0.9094 - val_loss: 0.3329 - val_accuracy:
Epoch 6/10
50/50 [==============================] - 9s 184ms/step - loss: 0.2118 - accuracy: 0.9472 - val_loss: 0.2854 - val_accuracy:
Epoch 7/10
50/50 [==============================] - 7s 149ms/step - loss: 0.1691 - accuracy: 0.9597 - val_loss: 0.2683 - val_accuracy:
Epoch 8/10
50/50 [==============================] - 8s 169ms/step - loss: 0.1436 - accuracy: 0.9647 - val_loss: 0.2712 - val_accuracy:
Epoch 9/10
50/50 [==============================] - 5s 110ms/step - loss: 0.1057 - accuracy: 0.9748 - val_loss: 0.2395 - val_accuracy:
Epoch 10/10
50/50 [==============================] - 8s 162ms/step - loss: 0.0882 - accuracy: 0.9797 - val_loss: 0.2476 - val_accuracy:
```

```
#predict and eval
#metric report
```

```
pred_rnn = model.predict(x_test_rnn)
pred_rnn = np.argmax(pred_rnn, axis=1)
pred_rnn = to_categorical(pred_rnn, num_classes=len(y_test_labels_rnn[0]))
print(classification_report(y_test_labels_rnn, pred_rnn))
```

```
    63/63 [==============================] - 1s 21ms/step
                  precision    recall  f1-score   support

               0       0.95      0.92      0.94       544
               1       0.94      0.97      0.96       460
               2       0.93      0.93      0.93       518
               3       0.92      0.92      0.92       478

       micro avg       0.94      0.94      0.94      2000
       macro avg       0.94      0.94      0.94      2000
    weighted avg       0.94      0.94      0.94      2000
     samples avg       0.94      0.94      0.94      2000
```

Our models here seem to yield similar results to the sequential model, and the loss values being so low seem to indicate a similar case of overfitting? I'm not sure, I can't really evaluate this half the time because this whole page takes 20 minutes to complete.

## Part Four: Different Embeddings

Up until here, we've used the simplest way of embedding, one-hot encoding to transform our data into vectors of integers to perform deep learning on. Here, we plan to change how the text of our data is encoded and represented as numerical vectors.

Below is the vectorizer setup for our embeddings. We want to take the top 35k works and have each sample truncate at 350 words. (Since our descriptions can get really wordy.)

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

x = grouped.Description
y = grouped.Item_type
x_train_em, x_test_em, y_train_em, y_test_em = train_test_split(x, y, test_size= 0.2, random_state= 1234)

vectorizer = TextVectorization(max_tokens=35000, output_sequence_length=350)
text_ds = tf.data.Dataset.from_tensor_slices(x_train_em).batch(128)
vectorizer.adapt(text_ds)

voc = vectorizer.get_vocabulary()
word_index = dict(zip(voc, range(len(voc))))


#load gloVe embeddings
embeddings = {}
f = open('/content/drive/MyDrive/glove.6B.300d.txt')
for line in f:
  values = line.split()
  word = values[0]
  coefs =np.asarray(values[1:], dtype = 'float32')
  embeddings[word] = coefs
f.close()


#create embeddings matrix
embedding_dimens = 300
max_words = 35000
embedding_matrix = np.zeros((max_words, embedding_dimens))
for word, i in tokenizer.word_index.items():
  if i< max_words:
    embedding_vector = embeddings.get(word)
    if embedding_vector is not None:
      embedding_matrix[i] = embedding_vector


#try on new model
#new model w/ LSTM
model = models.Sequential()
model.add(layers.Embedding(max_words, embedding_dimens, input_length=maxlen))
model.add(layers.LSTM(32))
model.add(layers.Dense(4, activation='softmax'))
model.summary()
```

```
model.layers[0].set_weights([embedding_matrix])
model.layers[0].trainable = False
```

```
Model: "sequential_3"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 embedding_2 (Embedding)     (None, 1000, 300)         10500000

 lstm_1 (LSTM)               (None, 32)                42624

 dense_4 (Dense)             (None, 4)                 132

=================================================================
Total params: 10,542,756
Trainable params: 10,542,756
Non-trainable params: 0
_________________________________________________________________
```

```
#model creation and trianing
# comple
#compile the model
model.compile(optimizer='rmsprop',
              loss = 'categorical_crossentropy',
              metrics = ['accuracy'])
#train
history_lstm = model.fit(x_train_em,
                    y_train_em,
                    epochs = 10, #computationaly complexity, each epoch takes 3 mins to compute
                    batch_size = 128,
                    validation_split = 0.2)

#predict and eval
#metric report
pred_em = model.predict(x_test_em)
pred_em = np.argmax(pred_em, axis=1)
pred_em = to_categorical(pred_em, num_classes=len(y_test_em[0]))
print(classification_report(y_test_em, pred_em))
```

```
Epoch 1/10
50/50 [==============================] - 4s 46ms/step - loss: 0.1549 - accuracy: 0.9594 - val_loss: 0.2443 - val_accuracy: 0
Epoch 2/10
50/50 [==============================] - 2s 35ms/step - loss: 0.1385 - accuracy: 0.9634 - val_loss: 0.2358 - val_accuracy: 0
Epoch 3/10
50/50 [==============================] - 2s 37ms/step - loss: 0.1295 - accuracy: 0.9656 - val_loss: 0.2413 - val_accuracy: 0
Epoch 4/10
50/50 [==============================] - 2s 43ms/step - loss: 0.1231 - accuracy: 0.9663 - val_loss: 0.2466 - val_accuracy: 0
Epoch 5/10
50/50 [==============================] - 2s 43ms/step - loss: 0.1147 - accuracy: 0.9680 - val_loss: 0.2378 - val_accuracy: 0
Epoch 6/10
50/50 [==============================] - 2s 39ms/step - loss: 0.1016 - accuracy: 0.9736 - val_loss: 0.2580 - val_accuracy: 0
Epoch 7/10
50/50 [==============================] - 2s 35ms/step - loss: 0.1016 - accuracy: 0.9728 - val_loss: 0.2572 - val_accuracy: 0
Epoch 8/10
50/50 [==============================] - 2s 35ms/step - loss: 0.0939 - accuracy: 0.9737 - val_loss: 0.2450 - val_accuracy: 0
Epoch 9/10
50/50 [==============================] - 2s 35ms/step - loss: 0.0843 - accuracy: 0.9767 - val_loss: 0.2461 - val_accuracy: 0
Epoch 10/10
50/50 [==============================] - 2s 35ms/step - loss: 0.0774 - accuracy: 0.9780 - val_loss: 0.2533 - val_accuracy: 0
63/63 [==============================] - 1s 11ms/step
              precision    recall  f1-score   support

           0       0.96      0.95      0.96       544
           1       0.98      0.97      0.98       460
           2       0.95      0.93      0.94       518
           3       0.91      0.94      0.92       478

   micro avg       0.95      0.95      0.95      2000
   macro avg       0.95      0.95      0.95      2000
weighted avg       0.95      0.95      0.95      2000
 samples avg       0.95      0.95      0.95      2000
```

## Part 5: Findings

With the embeddings we have a slightly larger loss and slightly lower accuracy, and I think that's becasue the embedding structure, freezing the learning, was able to combat the overfitting our model had shown in earlier versions, thus making this the most optimal.

24s completed at 5:40 PM