Divide into train/test. Create a graph showing the distribution of the target classes

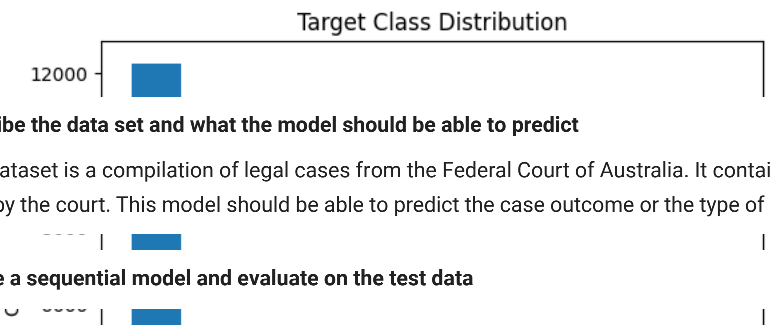
```
# setup
import pandas as pd

# read csv
df = pd.read_csv('legal_text_classification.csv')
df = df.fillna(0)
df = df[pd.notnull(df['case_text'])]

df.head()
```

	case_id	case_outcome	case_title	case_text
0	Case1	cited	Alpine Hardwood (Aust) Pty Ltd v Hardys Pty Lt...	Ordinarily that discretion will be exercised s...
1	Case2	cited	Black v Lipovac [1998] FCA 699 ; (1998) 217 AL...	The general principles governing the exercise ...
2	Case3	cited	Colgate Palmolive Co v Cussons Pty Ltd (1993) ...	Ordinarily that discretion will be exercised s...
			Dais Studio Pty Ltd v Bullett	The general principles

```
# train test split
import numpy as np

i = np.random.rand(len(df)) < 0.8
train = df[i]
test = df[~i]
print("train data size: ", train.shape)
print("test data size: ", test.shape)
```

```
train data size:  (20064, 4)
test data size:  (4921, 4)
```

```
# create graph
import matplotlib.pyplot as plt

classes = df['case_outcome'].value_counts()

plt.bar(classes.index, classes.values)
plt.title('Target Class Distribution')
plt.xlabel('Class')
plt.xticks(rotation=45, ha="right")
plt.ylabel('Count')
plt.show()
```



Describe the data set and what the model should be able to predict

This dataset is a compilation of legal cases from the Federal Court of Australia. It contains the case text, case title, and how the the cases were cited by the court. This model should be able to predict the case outcome or the type of legal citation used for a case based on the text.

Create a sequential model and evaluate on the test data

```
import tensorflow as tf
from tensorflow.keras.preprocessing.text import Tokenizer
from tensorflow.keras import layers, models

from sklearn.preprocessing import LabelEncoder

# set up X and Y
num_labels = 2
vocab_size = 25000
batch_size = 100

# fit the tokenizer on the training data
tokenizer = Tokenizer(num_words=vocab_size)
tokenizer.fit_on_texts(item for item in train.case_text if not isinstance(item, int))

x_train = tokenizer.texts_to_matrix((item for item in train.case_text if not isinstance(item, int)), mode='tfidf')
x_test = tokenizer.texts_to_matrix((item for item in train.case_text if not isinstance(item, int)), mode='tfidf')

encoder = LabelEncoder()
encoder.fit(train.case_outcome)
y_train = encoder.transform(train.case_outcome)
y_test = encoder.transform(test.case_outcome)

# check shape
print("train shapes:", x_train.shape, y_train.shape)
print("test shapes:", x_test.shape, y_test.shape)
print("test first five labels:", y_test[:5])
```

```
    train shapes: (19921, 25000) (20064,)
    test shapes: (19921, 25000) (4921,)
    test first five labels: [3 3 1 7 7]
```

```
# fit model
model = models.Sequential()
model.add(layers.Dense(32, input_dim=vocab_size, kernel_initializer='normal', activation='relu'))
model.add(layers.Dense(1, kernel_initializer='normal', activation='softmax'))

model.compile(loss='categorical_crossentropy',
              optimizer='adam',
              metrics=['accuracy'])

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

```
    Epoch 1/30
    /usr/local/lib/python3.9/dist-packages/tensorflow/python/util/dispatch.py:1176: SyntaxWarning: In loss categorical_crossentropy, expe
      return dispatch_target(*args, **kwargs)
    180/180 [==============================] - 6s 27ms/step - loss: 0.0000e+00 - accuracy: 0.0951 - val_loss: 0.0000e+00 - val_accuracy: 
    Epoch 2/30
    180/180 [==============================] - 4s 25ms/step - loss: 0.0000e+00 - accuracy: 0.0951 - val_loss: 0.0000e+00 - val_accuracy: 
    Epoch 3/30
    180/180 [==============================] - 5s 30ms/step - loss: 0.0000e+00 - accuracy: 0.0951 - val_loss: 0.0000e+00 - val_accuracy: 
    Epoch 4/30
    180/180 [==============================] - 5s 28ms/step - loss: 0.0000e+00 - accuracy: 0.0951 - val_loss: 0.0000e+00 - val_accuracy: 
    Epoch 5/30
    180/180 [==============================] - 6s 31ms/step - loss: 0.0000e+00 - accuracy: 0.0951 - val_loss: 0.0000e+00 - val_accuracy: 
    Epoch 6/30
    180/180 [==============================] - 5s 28ms/step - loss: 0.0000e+00 - accuracy: 0.0951 - val_loss: 0.0000e+00 - val_accuracy: 
    Epoch 7/30
```

```
180/180 [==============================] - 5s 28ms/step - loss: 0.0000e+00 - accuracy: 0.0951 - val_loss: 0.0000e+00 - val_accuracy: 0
Epoch 8/30
180/180 [==============================] - 6s 32ms/step - loss: 0.0000e+00 - accuracy: 0.0951 - val_loss: 0.0000e+00 - val_accuracy: 0
Epoch 9/30
180/180 [==============================] - 5s 25ms/step - loss: 0.0000e+00 - accuracy: 0.0951 - val_loss: 0.0000e+00 - val_accuracy: 0
Epoch 10/30
180/180 [==============================] - 6s 32ms/step - loss: 0.0000e+00 - accuracy: 0.0951 - val_loss: 0.0000e+00 - val_accuracy: 0
Epoch 11/30
180/180 [==============================] - 5s 27ms/step - loss: 0.0000e+00 - accuracy: 0.0951 - val_loss: 0.0000e+00 - val_accuracy: 0
Epoch 12/30
180/180 [==============================] - 4s 23ms/step - loss: 0.0000e+00 - accuracy: 0.0951 - val_loss: 0.0000e+00 - val_accuracy: 0
Epoch 13/30
180/180 [==============================] - 5s 29ms/step - loss: 0.0000e+00 - accuracy: 0.0951 - val_loss: 0.0000e+00 - val_accuracy: 0
Epoch 14/30
180/180 [==============================] - 4s 23ms/step - loss: 0.0000e+00 - accuracy: 0.0951 - val_loss: 0.0000e+00 - val_accuracy: 0
Epoch 15/30
180/180 [==============================] - 8s 46ms/step - loss: 0.0000e+00 - accuracy: 0.0951 - val_loss: 0.0000e+00 - val_accuracy: 0
Epoch 16/30
180/180 [==============================] - 8s 44ms/step - loss: 0.0000e+00 - accuracy: 0.0951 - val_loss: 0.0000e+00 - val_accuracy: 0
Epoch 17/30
180/180 [==============================] - 6s 31ms/step - loss: 0.0000e+00 - accuracy: 0.0951 - val_loss: 0.0000e+00 - val_accuracy: 0
Epoch 18/30
180/180 [==============================] - 5s 26ms/step - loss: 0.0000e+00 - accuracy: 0.0951 - val_loss: 0.0000e+00 - val_accuracy: 0
Epoch 19/30
180/180 [==============================] - 6s 32ms/step - loss: 0.0000e+00 - accuracy: 0.0951 - val_loss: 0.0000e+00 - val_accuracy: 0
Epoch 20/30
180/180 [==============================] - 7s 39ms/step - loss: 0.0000e+00 - accuracy: 0.0951 - val_loss: 0.0000e+00 - val_accuracy: 0
Epoch 21/30
180/180 [==============================] - 5s 30ms/step - loss: 0.0000e+00 - accuracy: 0.0951 - val_loss: 0.0000e+00 - val_accuracy: 0
Epoch 22/30
180/180 [==============================] - 6s 35ms/step - loss: 0.0000e+00 - accuracy: 0.0951 - val_loss: 0.0000e+00 - val_accuracy: 0
Epoch 23/30
180/180 [==============================] - 4s 24ms/step - loss: 0.0000e+00 - accuracy: 0.0951 - val_loss: 0.0000e+00 - val_accuracy: 0
Epoch 24/30
180/180 [==============================] - 5s 30ms/step - loss: 0.0000e+00 - accuracy: 0.0951 - val_loss: 0.0000e+00 - val_accuracy: 0
Epoch 25/30
180/180 [==============================] - 6s 31ms/step - loss: 0.0000e+00 - accuracy: 0.0951 - val_loss: 0.0000e+00 - val_accuracy: 0
Epoch 26/30
180/180 [==============================] - 5s 28ms/step - loss: 0.0000e+00 - accuracy: 0.0951 - val_loss: 0.0000e+00 - val_accuracy: 0
Epoch 27/30
180/180 [==============================] - 5s 30ms/step - loss: 0.0000e+00 - accuracy: 0.0951 - val_loss: 0.0000e+00 - val_accuracy: 0
```

```
score = model.evaluate(x_test[:5000], y_test[:5000], batch_size=batch_size, verbose=1)
print('Accuracy: ', score[1])
```

Try a different architecture like RNN, CNN, etc and evaluate on the test data

RNN

```
# fit model
model = models.Sequential()
model.add(layers.Embedding(10000, 32))
model.add(layers.SimpleRNN(32))
model.add(layers.Dense(1, kernel_initializer='normal', activation='softmax'))
```

```
model.compile(loss='categorical_crossentropy', optimizer='adam', metrics=['accuracy'])
```

```
score = model.evaluate(x_test[:5000], y_test[:5000], batch_size=batch_size, verbose=1)
print('Accuracy: ', score[1])
```

```
history = model.fit(x_train, y_train,
                    batch_size=batch_size,
                    epochs=1,
                    verbose=1,
                    validation_split=0.1)
```

Using Different Embeddings

```
model = models.Sequential()
model.add(layers.Embedding(input_dim=10, output_dim=8, input_length=1))
model.add(layers.Flatten())
model.add(layers.Dense(32, activation='relu'))
model.add(layers.Dense(1, activation='softmax'))
model.compile(loss='categorical_crossentropy', optimizer='adam', metrics=['accuracy'])


history = model.fit(x_train, y_train,
                    batch_size=batch_size,
                    epochs=1,
                    verbose=1,
                    validation_split=0.1)
```

Analysis In this assignment, we used the Sequential model as a base to compare performance. The Sequential model is a feed-forward network. Unlike the text classification 1, we are now working with a dataset that supports categorical classification rather than binary classification since there are multiple ways a legal case can be cited. The Sequential model in this code has one dense layer with 32 nodes and a dense output layer that produces output based on the softmax activation function which I decided would be more appropriate for multi-class classification. Overall, the accuracy was not very good, with an accuracy score of around 10% after evaluating the test data. However, this could be for several reasons such as providing nonoptimal parameters to the layers or picking the wrong model for the dataset which has 10 possible classes for the data because sequential models may not perform as well with text data. I also tried using Recurrent Neural Networks as my other architecture choice to see how it would perform with the RNN memory capabilities suited for processing input sequences like text by maintaining a hidden state with information from previous inputs. However, when I attempted to fit the model, I found that the ETA for completion was around 4.5 hours. Similarly, using a different Embeddings approach on the sequential model also had low accuracy. However, in general, embeddings are very useful in NLP because they are a way of representing text data as numerical vectors. This is good for applications where context and semantics are important. Though I found in this assignment that sequential models were not the best fit.