RickMortyTwo

April 23, 2023

1 Text Classification 2: RNN and CNN

Previously, I tried to classify text using a simple feed forward neural network. Now I will be trying more complex methods. To speed up the process, because I've been sick for a couple days now, I'll be reusing the Rick and Morty dataset before. This means we will be able to compare to my previous attempts at detecting if a line is *indeed* Rick from Rick and Morty.

Another differentiation is that we will be using Keras, as these are Deep Learning tools.

1.1 The Data

I will be recreating the data identical to last time, balancing out the number of Rick lines to every other character. I want to keep note of the unbalanced data set and compare it's performance to the balanced dataset. So we will have a df, which has all of the lines leaving Rick in a minority, and the even_df, with the sampling balanced between classes.

[89]:
```
import pandas as pd

# Random seed for reproducability:
seed = 1123

df = pd.read_csv('RickAndMortyScripts.csv',
                 header=0,
                 usecols=[4,5],
                 dtype={'name':'category', 'line': 'str'})
print('rows and columns:', df.shape)
df
```

```
rows and columns: (1905, 2)
```

[89]:

	name	line
0	Rick	Morty! You gotta come on. Jus'… you gotta co…
1	Morty	What, Rick? What's going on?
2	Rick	I got a surprise for you, Morty.
3	Morty	It's the middle of the night. What are you tal…
4	Rick	Come on, I got a surprise for you. Come on, h…
…	…	…
1900	Morty	That was amazing!
1901	Rick	Got some of that mermaid puss!

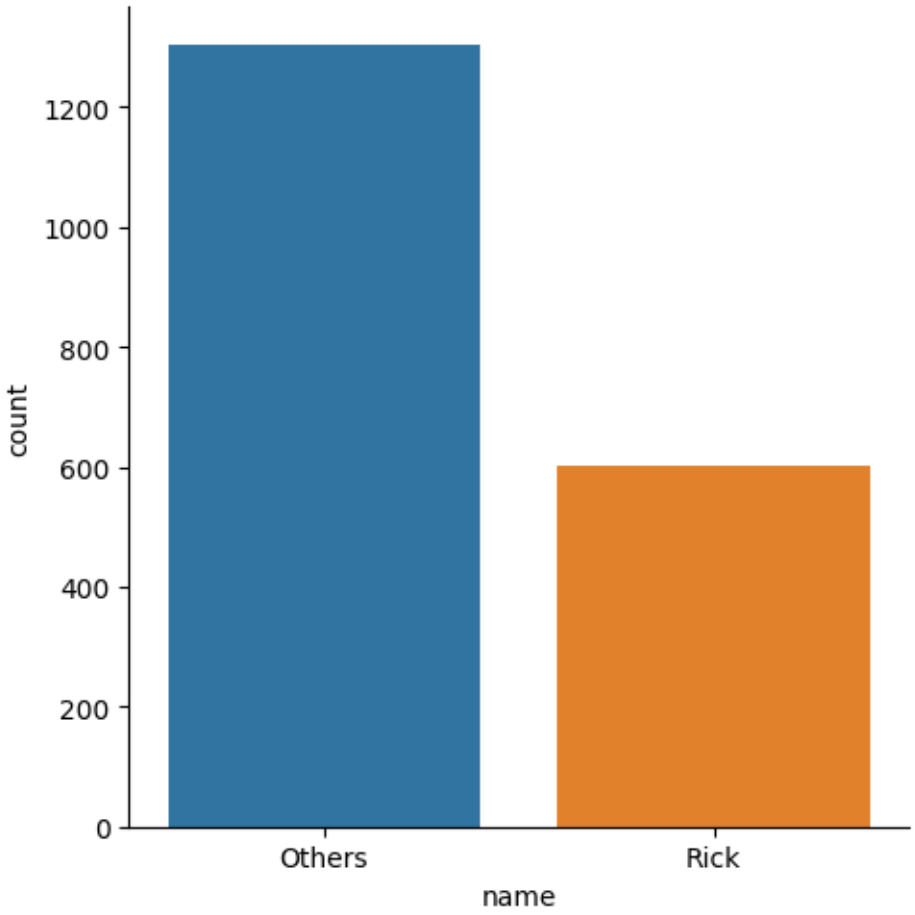
```
1902  Morty  I'm really hoping it wasn't a one-off thing an…
1903   Rick  Pssh! Not at all, Morty. That place will never…
1904  Morty                      Whoo! Yeah! Yeaah! Ohhh, shit!

[1905 rows x 2 columns]
```

[90]:
```
def change_name(row):
  if 'rick' in row['name'].lower():
    return 'Rick'
  else:
    return 'Others'

# Note that the output is still a category!
df['name'] = df.apply(change_name, axis=1).astype('category')
df.head()
```

[90]:
```
     name                                               line
0    Rick  Morty! You gotta come on. Jus'… you gotta co…
1  Others                          What, Rick? What's going on?
2    Rick                    I got a surprise for you, Morty.
3  Others  It's the middle of the night. What are you tal…
4    Rick  Come on, I got a surprise for you.  Come on, h…
```

[91]:
```
import seaborn as sb

sb.catplot(data=df, x='name', kind='count')
```

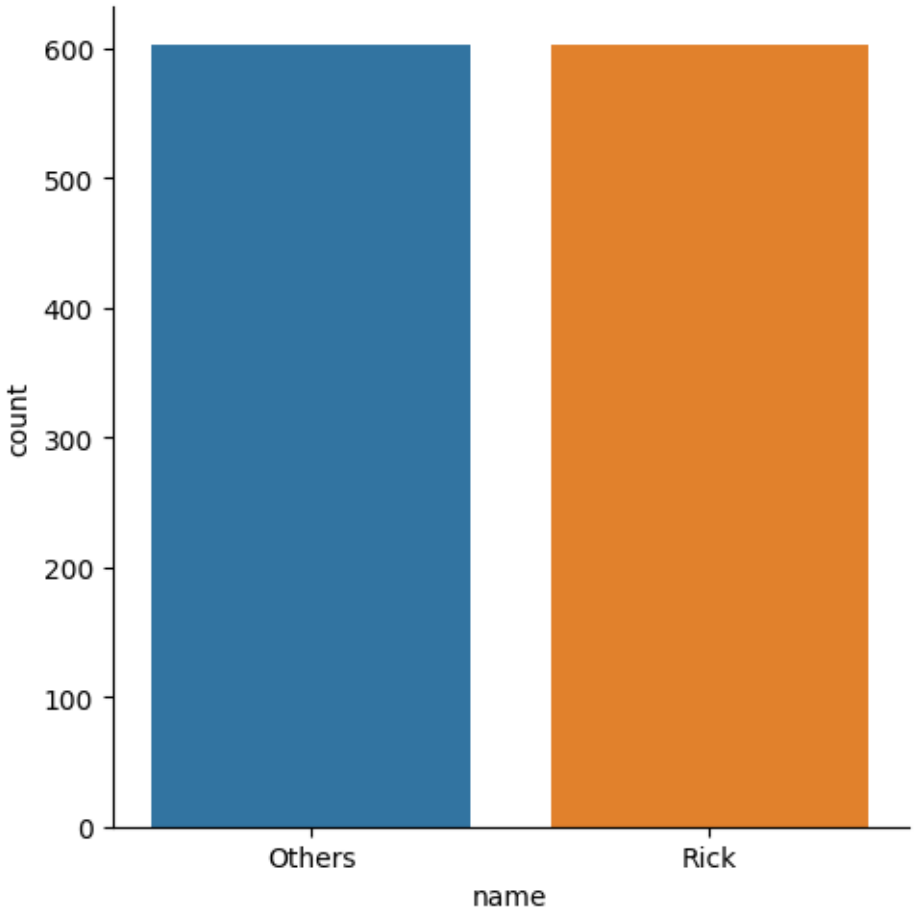
[91]:
```
<seaborn.axisgrid.FacetGrid at 0x7f4a96a30fd0>
```



[92]:
```
num_rick = df['name'].value_counts()['Rick']
print("Number of Rick lines: %d" % (num_rick))
num_not_rick = df['name'].value_counts()['Others']
print("Number of Other lines: %d" % (num_not_rick))
diff = num_not_rick-num_rick
print("Number of lines to remove: %d" % (diff))

df_subset = df[df['name']=='Others'].sample(diff, random_state=seed)

even_df = df.drop(df_subset.index)
sb.catplot(data=even_df, x='name', kind='count')
```

```
Number of Rick lines: 602
Number of Other lines: 1303
Number of lines to remove: 701
```

[92]: <seaborn.axisgrid.FacetGrid at 0x7f4a912fd0d0>



[222]:
```
from wordcloud import WordCloud, STOPWORDS, ImageColorGenerator
from matplotlib import pyplot as plt
from PIL import Image
import numpy as np

rick_coloring = np.array(Image.open("Rick_Sanchez-0.png"))

wordcloud = WordCloud(background_color='white', stopwords = STOPWORDS,
                max_words = 2000, max_font_size = 100,
                random_state = seed,
                mask=rick_coloring)

plt.figure(figsize=(16, 12))
wordcloud.generate(''.join(even_df.loc[even_df['name'] == 'Rick', 'line']))

image_colors = ImageColorGenerator(rick_coloring)

# Show Image
```

```
plt.axis("off")
plt.imshow(wordcloud.recolor(color_func=image_colors), interpolation="bilinear")
```

[222]: <matplotlib.image.AxesImage at 0x7f4aff3f0520>


Oh
Yeah
thing
know
gonna
people
think
want
need
let
mean
Rick
Well
Come
man
whoa
go
take
shit
Jerry
good
Listen
guy
right
something
time
really
Look
Morty
even
seeds
now
make
pickle
one
Uh
place
whole
Summer
see
put
Alright
first

1.2 Splitting Data

Now we will start using tensorflow and Keras. Thankfully, keras models have the `validation_split` function, which allows models to evaluate their accuracy each epoch. It takes a decimal value representing a percentage, and takes that percentage of data from the training data for evaluation. It's worth noting that it just takes a continguous chunk from the data, not a random sampling.

To account for this, we will use both a test and validation dataset using the sklearn function to split and shuffle the data. We can evaluate hyperparameters on the validation dataset, as well as the final product on the test dataset.

[178]:
```
import tensorflow as tf
from tensorflow.keras import datasets, layers, models, optimizers
```

[95]:
```
from sklearn.model_selection import train_test_split

X = even_df['line']
y = even_df['name']
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,␣
 ↪random_state=seed)
```

[96]:
```
print("Train data size: ", X_train.shape)
print("Test data size: ", X_test.shape)
```

```
Train data size:  (963,)
Test data size:  (241,)
```

1.3 Preprocessing!

I want to both encode the classes as just 1 and 0, as well as tokenize the data using the Tfidf vectorizer from sklearn.

[97]:
```
from tensorflow.keras.preprocessing.text import Tokenizer
from sklearn.preprocessing import LabelEncoder

# Tokenize using tfidf
tokenizer = Tokenizer()
tokenizer.fit_on_texts(X_train)

X_train_vec = tokenizer.texts_to_matrix(X_train, mode='tfidf')
X_test_vec = tokenizer.texts_to_matrix(X_test, mode='tfidf')

encoder = LabelEncoder()
encoder.fit(y_train)
y_train_enc = encoder.transform(y_train)
```

```
y_test_enc = encoder.transform(y_test)
```

[59]:
```
print("train shapes:", X_train_vec.shape, y_train_enc.shape)
print("test shapes:", X_test_vec.shape, y_test_enc.shape)
print("test first five labels:", y_test_enc[:5])
```

```
train shapes: (963, 2519) (963,)
test shapes: (241, 2519) (241,)
test first five labels: [0 0 1 1 0]
```

Note, it's a very sparse dataset, so the vector encoding isn't very viewable.

[98]:
```
X_train_vec[0]
```

[98]:
```
array([0., 0., 0., …, 0., 0., 0.])
```

We can also now see the vocab is 2519 in size.

[99]:
```
vocab_size = 2519
batch_size = 100
```

1.4 Sequential Model

It's worth noting that we are performing a binary classification. That means the best loss function is binary crossentropy, and the best activation function is a sigmoid (as we are performing 0,1 classification). I set up a sequential model with this in mind, as well as enabling a validation set.

[182]:
```
# fit model
model = models.Sequential()
model.add(layers.Dense(64, input_dim=vocab_size, kernel_initializer='normal',␣
 ↪activation='sigmoid'))
model.add(layers.Dense(1, kernel_initializer='normal', activation='sigmoid'))
```

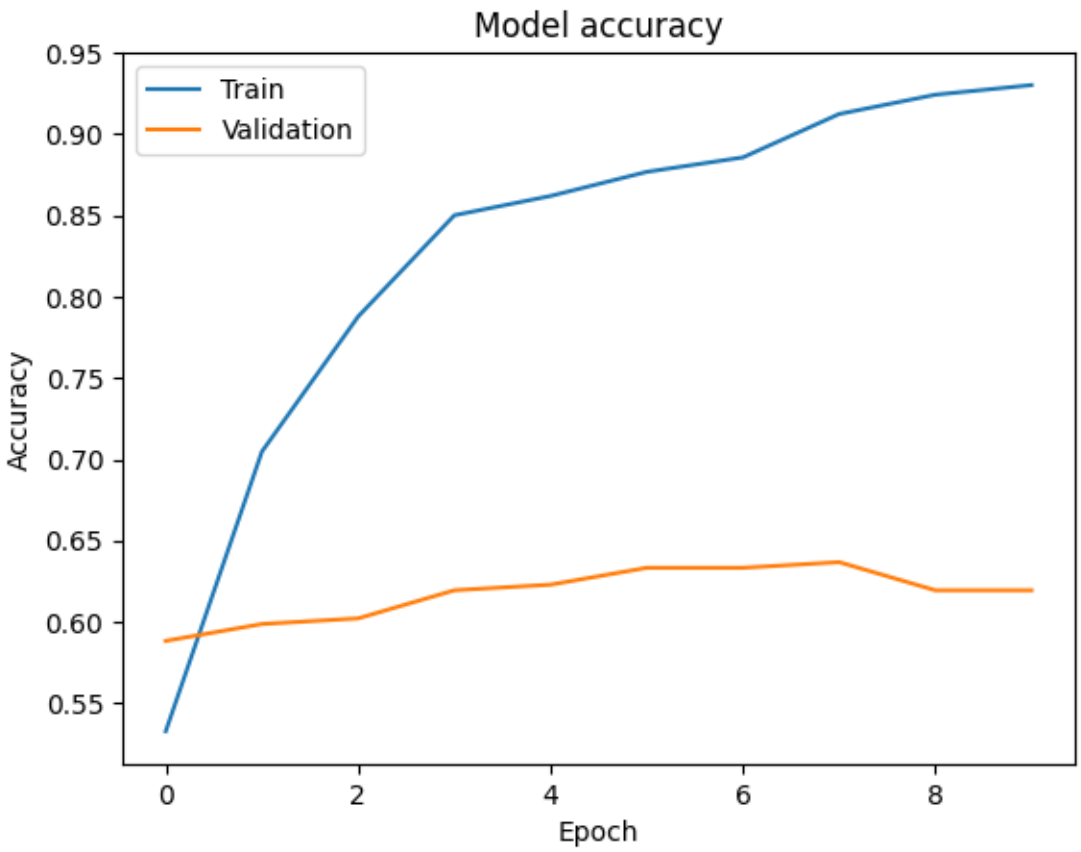
[183]:
```
model.compile(loss='binary_crossentropy',
              optimizer='adam',
              metrics=['accuracy'])
```

[184]:
```
history = model.fit(X_train_vec, y_train_enc, validation_split=0.3, epochs=10,␣
 ↪batch_size=64)
```

```
Epoch 1/10
11/11 [==============================] - 1s 28ms/step - loss: 0.6944 - accuracy:
0.5326 - val_loss: 0.6849 - val_accuracy: 0.5882
Epoch 2/10
11/11 [==============================] - 0s 12ms/step - loss: 0.6579 - accuracy:
0.7047 - val_loss: 0.6820 - val_accuracy: 0.5986
Epoch 3/10
11/11 [==============================] - 0s 12ms/step - loss: 0.6319 - accuracy:
0.7878 - val_loss: 0.6753 - val_accuracy: 0.6021
```

```
Epoch 4/10
11/11 [==============================] - 0s 11ms/step - loss: 0.6027 - accuracy:
0.8501 - val_loss: 0.6687 - val_accuracy: 0.6194
Epoch 5/10
11/11 [==============================] - 0s 11ms/step - loss: 0.5689 - accuracy:
0.8620 - val_loss: 0.6598 - val_accuracy: 0.6228
Epoch 6/10
11/11 [==============================] - 0s 10ms/step - loss: 0.5314 - accuracy:
0.8769 - val_loss: 0.6534 - val_accuracy: 0.6332
Epoch 7/10
11/11 [==============================] - 0s 11ms/step - loss: 0.4901 - accuracy:
0.8858 - val_loss: 0.6483 - val_accuracy: 0.6332
Epoch 8/10
11/11 [==============================] - 0s 11ms/step - loss: 0.4477 - accuracy:
0.9125 - val_loss: 0.6432 - val_accuracy: 0.6367
Epoch 9/10
11/11 [==============================] - 0s 11ms/step - loss: 0.4066 - accuracy:
0.9243 - val_loss: 0.6412 - val_accuracy: 0.6194
Epoch 10/10
11/11 [==============================] - 0s 12ms/step - loss: 0.3679 - accuracy:
0.9303 - val_loss: 0.6389 - val_accuracy: 0.6194
```

[185]:
```
plt.plot(history.history['accuracy'])
plt.plot(history.history['val_accuracy'])
plt.title('Model accuracy')
plt.ylabel('Accuracy')
plt.xlabel('Epoch')
plt.legend(['Train', 'Validation'], loc='upper left')
plt.show()
```



Just in our test run, I can tell we need to tune those hyperparameters based on how our validation set didn't get better accuracy while the model was overfitting.

[186]:
```
score = model.evaluate(X_test_vec, y_test_enc, batch_size=batch_size, verbose=1)
print('Accuracy: ', score[1])
```

```
3/3 [==============================] - 0s 8ms/step - loss: 0.6403 - accuracy:
0.6515
Accuracy:  0.6514523029327393
```

1.4.1 Messing with Parameters

I really would like to use Grid Search. I'm going to go ahead and import the required KerasClassifier and GridSearchCV. KerasClassifier is a wrapper to use Keras with Sklearn, in our case, we want to use GridSearch. The most up to date version is from a library scikeras, which we must install.

[113]:
```
!pip install scikeras
```

```
Looking in indexes: https://pypi.org/simple, https://us-python.pkg.dev/colab-
wheels/public/simple/
Collecting scikeras
```

```
  Downloading scikeras-0.10.0-py3-none-any.whl (27 kB)
Requirement already satisfied: scikit-learn>=1.0.0 in
/usr/local/lib/python3.9/dist-packages (from scikeras) (1.2.2)
Requirement already satisfied: packaging>=0.21 in /usr/local/lib/python3.9/dist-
packages (from scikeras) (23.1)
Requirement already satisfied: numpy>=1.17.3 in /usr/local/lib/python3.9/dist-
packages (from scikit-learn>=1.0.0->scikeras) (1.22.4)
Requirement already satisfied: joblib>=1.1.1 in /usr/local/lib/python3.9/dist-
packages (from scikit-learn>=1.0.0->scikeras) (1.2.0)
Requirement already satisfied: threadpoolctl>=2.0.0 in
/usr/local/lib/python3.9/dist-packages (from scikit-learn>=1.0.0->scikeras)
(3.1.0)
Requirement already satisfied: scipy>=1.3.2 in /usr/local/lib/python3.9/dist-
packages (from scikit-learn>=1.0.0->scikeras) (1.10.1)
Installing collected packages: scikeras
Successfully installed scikeras-0.10.0
```

[118]:
```
from keras.models import Sequential
from keras.layers import Dense, Dropout
from scikeras.wrappers import KerasClassifier
from sklearn.model_selection import GridSearchCV
from keras.optimizers import Adam
```

[115]:
```
param_grid = {
    'batch_size': [16, 32, 64],
    'epochs': [5, 10, 20],
    'learning_rate': [0.001, 0.01, 0.1],
    'activation': ['relu', 'sigmoid'],
    'dropout_rate': [0.0, 0.2, 0.4]
}
```

[130]:
```
# Define a function that creates a Keras model with the desired hyperparameters
def create_model(learning_rate=0.01, activation='relu', dropout_rate=0.0):
    model = Sequential()
    model.add(Dense(32, input_dim=vocab_size, kernel_initializer='normal',␣
 ↪activation=activation))
    model.add(Dropout(dropout_rate))
    model.add(Dense(1, kernel_initializer='normal', activation='sigmoid'))
    optimizer = Adam(learning_rate=learning_rate)
    model.compile(loss='binary_crossentropy', optimizer=optimizer,␣
 ↪metrics=['accuracy'])
    return model
```

[131]:
```
# Now use that function with the scikit wrapper so we can plug the Keras
# model into the SciKit GridSearch function. Note, we have to define a default
# value for each parameter for the function to work properly
```

```
model = KerasClassifier(build_fn=create_model, activation='relu',␣
 ↪learning_rate=0.01, dropout_rate=0.0)
```

Above we set up a function that creates Keras models in a way SciKitLearn can understand, and define a grid of parameters we can test out building different models out of. We can then use the GridSearchCV function to run through different combinations of these parameters on this model.

[132]:
```
grid = GridSearchCV(estimator=model, param_grid=param_grid, n_jobs=-1, cv=3)
grid_result = grid.fit(X_train_vec, y_train_enc)
```

```
/usr/local/lib/python3.9/dist-
packages/joblib/externals/loky/process_executor.py:700: UserWarning: A worker
stopped while some jobs were given to the executor. This can be caused by a too
short worker timeout or by a memory leak.
  warnings.warn(

Epoch 1/10

/usr/local/lib/python3.9/dist-packages/scikeras/wrappers.py:301: UserWarning:
``build_fn`` will be renamed to ``model`` in a future release, at which point
use of ``build_fn`` will raise an Error instead.
  warnings.warn(

61/61 [==============================] - 1s 4ms/step - loss: 0.6897 - accuracy:
0.5337
Epoch 2/10
61/61 [==============================] - 0s 4ms/step - loss: 0.6634 - accuracy:
0.6771
Epoch 3/10
61/61 [==============================] - 0s 4ms/step - loss: 0.6202 - accuracy:
0.7684
Epoch 4/10
61/61 [==============================] - 0s 4ms/step - loss: 0.5468 - accuracy:
0.8349
Epoch 5/10
61/61 [==============================] - 0s 4ms/step - loss: 0.4705 - accuracy:
0.8744
Epoch 6/10
61/61 [==============================] - 0s 4ms/step - loss: 0.3918 - accuracy:
0.9045
Epoch 7/10
61/61 [==============================] - 0s 4ms/step - loss: 0.3344 - accuracy:
0.9169
Epoch 8/10
61/61 [==============================] - 0s 4ms/step - loss: 0.2847 - accuracy:
0.9325
Epoch 9/10
61/61 [==============================] - 0s 4ms/step - loss: 0.2426 - accuracy:
0.9398
```

```
Epoch 10/10
61/61 [==============================] - 0s 5ms/step - loss: 0.2132 - accuracy:
0.9533
```

[187]:
```
best_model = grid_result.best_estimator_
y_pred = best_model.predict(X_test_vec)
accuracy = np.mean(y_pred == y_test_enc)
print("Accuracy on test set:", accuracy)
```

```
16/16 [==============================] - 0s 2ms/step
Accuracy on test set: 0.6929460580912863
```

[153]:
```
grid_result.best_params_
```

[153]:
```
{'activation': 'sigmoid',
 'batch_size': 16,
 'dropout_rate': 0.2,
 'epochs': 10,
 'learning_rate': 0.001}
```

[199]:
```
best_model
```

[199]:
```
KerasClassifier(
        model=None
        build_fn=<function create_model at 0x7f4aa4e633a0>
        warm_start=False
        random_state=None
        optimizer=rmsprop
        loss=None
        metrics=None
        batch_size=16
        validation_batch_size=None
        verbose=1
        callbacks=None
        validation_split=0.0
        shuffle=True
        run_eagerly=False
        epochs=10
        activation=sigmoid
        learning_rate=0.001
        dropout_rate=0.2
        class_weight=None
)
```

I can tell here that smaller is better in this case on such a small dataset. From the future I'll be trying smaller values overall, and stick to the sigmoid activation.

For whatever reason, the KerasClassifier object doesn't store all of the attributes I would expect,

like optmizer and metrics.

Now I would like to try slightly different architecture with these parameters in mind. Just seeting what adding an extra dropout layer would do to minimize the overfitting

[159]:
```
smaller_model = models.Sequential()
smaller_model.add(layers.Dense(32, input_dim=vocab_size,␣
 ↪kernel_initializer='normal', activation='sigmoid'))
smaller_model.add(Dropout(.2))
smaller_model.add(layers.Dense(32, input_dim=vocab_size,␣
 ↪kernel_initializer='normal', activation='sigmoid'))
smaller_model.add(Dropout(.2))
smaller_model.add(layers.Dense(1, kernel_initializer='normal',␣
 ↪activation='sigmoid'))
```

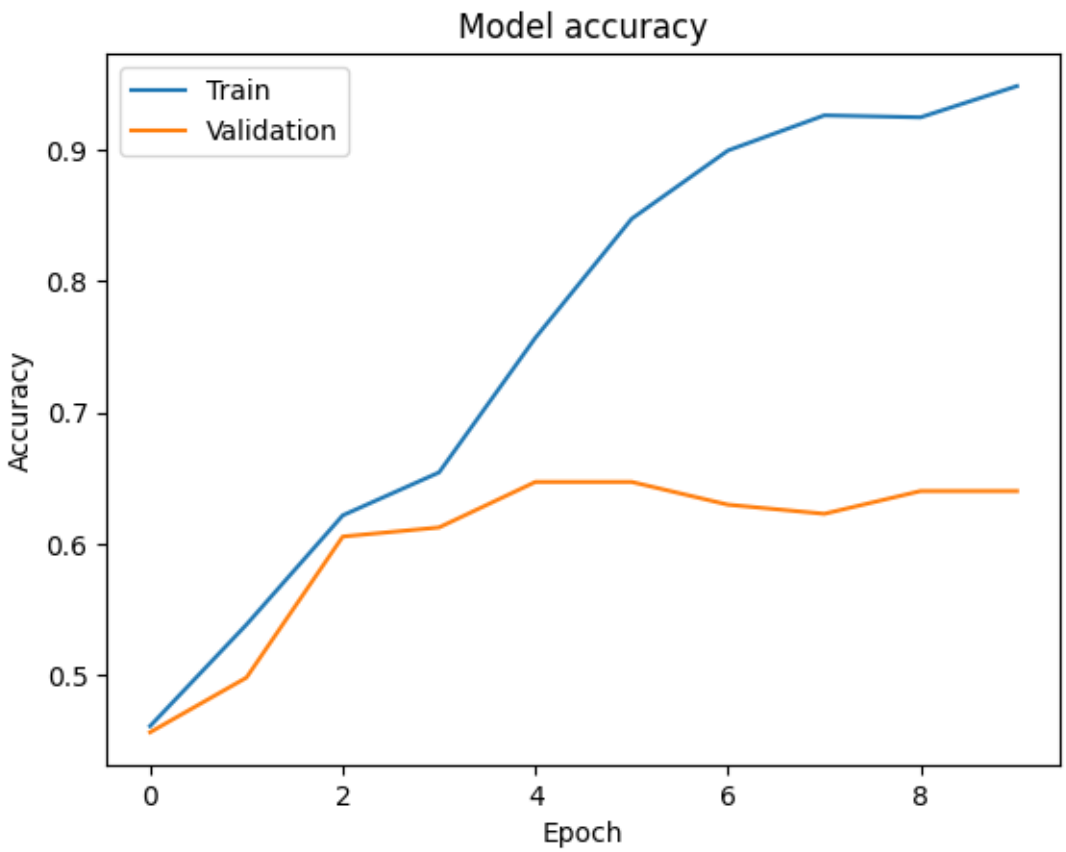
[160]:
```
smaller_model.compile(loss='binary_crossentropy',
              optimizer='adam',
              metrics=['accuracy'])
```

[161]:
```
history = smaller_model.fit(X_train_vec, y_train_enc, validation_split=0.3,␣
 ↪epochs=10, batch_size=16)
```

```
Epoch 1/10
43/43 [==============================] - 2s 13ms/step - loss: 0.6970 - accuracy:
0.4614 - val_loss: 0.6937 - val_accuracy: 0.4567
Epoch 2/10
43/43 [==============================] - 0s 8ms/step - loss: 0.6915 - accuracy:
0.5386 - val_loss: 0.6918 - val_accuracy: 0.4983
Epoch 3/10
43/43 [==============================] - 0s 6ms/step - loss: 0.6812 - accuracy:
0.6217 - val_loss: 0.6879 - val_accuracy: 0.6055
Epoch 4/10
43/43 [==============================] - 0s 5ms/step - loss: 0.6705 - accuracy:
0.6543 - val_loss: 0.6814 - val_accuracy: 0.6125
Epoch 5/10
43/43 [==============================] - 0s 5ms/step - loss: 0.6324 - accuracy:
0.7567 - val_loss: 0.6635 - val_accuracy: 0.6471
Epoch 6/10
43/43 [==============================] - 0s 6ms/step - loss: 0.5515 - accuracy:
0.8472 - val_loss: 0.6443 - val_accuracy: 0.6471
Epoch 7/10
43/43 [==============================] - 0s 5ms/step - loss: 0.4402 - accuracy:
0.8991 - val_loss: 0.6298 - val_accuracy: 0.6298
Epoch 8/10
43/43 [==============================] - 0s 5ms/step - loss: 0.3433 - accuracy:
0.9258 - val_loss: 0.6386 - val_accuracy: 0.6228
Epoch 9/10
43/43 [==============================] - 0s 5ms/step - loss: 0.2728 - accuracy:
```

```
0.9243 - val_loss: 0.6639 - val_accuracy: 0.6401
Epoch 10/10
43/43 [==============================] - 0s 5ms/step - loss: 0.2166 - accuracy:
0.9481 - val_loss: 0.7047 - val_accuracy: 0.6401
```

[163]:
```
plt.plot(history.history['accuracy'])
plt.plot(history.history['val_accuracy'])
plt.title('Model accuracy')
plt.ylabel('Accuracy')
plt.xlabel('Epoch')
plt.legend(['Train', 'Validation'], loc='upper left')
plt.show()
```



[164]:
```
score = smaller_model.evaluate(X_test_vec, y_test_enc, batch_size=batch_size,␣
↪verbose=1)
print('Accuracy: ', score[1])
```

```
3/3 [==============================] - 0s 5ms/step - loss: 0.6951 - accuracy:
0.6390
Accuracy:  0.6390041708946228
```

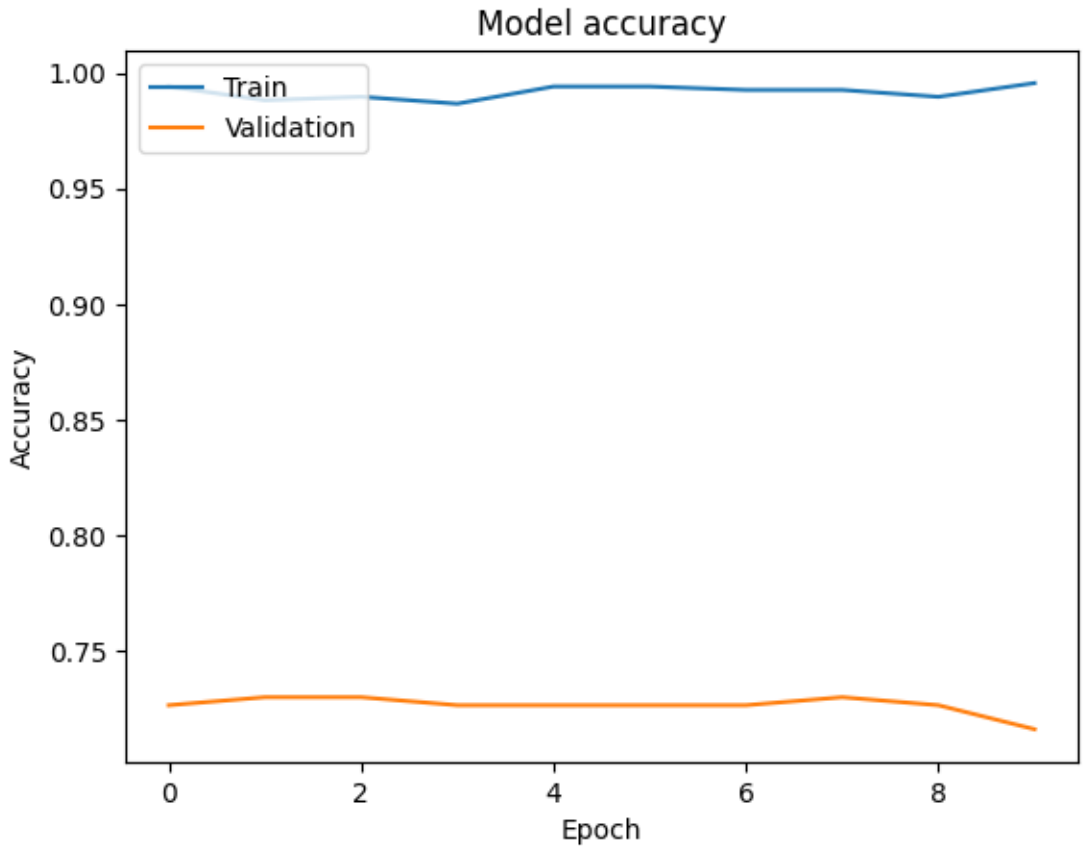
No increase. Now I will try L1/L2 regularization

[191]:
```
from keras import regularizers
reg_model = Sequential()
reg_model.add(Dense(32, input_dim=vocab_size, kernel_initializer='normal',␣
 ↪activation='sigmoid', kernel_regularizer=regularizers.l1(l1=0.0001)))
reg_model.add(Dropout(.2))
reg_model.add(Dense(1, kernel_initializer='normal', activation='sigmoid'))
optimizer = Adam(learning_rate=.001)
reg_model.compile(loss='binary_crossentropy', optimizer=optimizer,␣
 ↪metrics=['accuracy'])
```

[192]:
```
history = smaller_model.fit(X_train_vec, y_train_enc, validation_split=0.3,␣
 ↪epochs=10, batch_size=16)
```

```
Epoch 1/10
43/43 [==============================] - 1s 14ms/step - loss: 0.0182 - accuracy:
0.9941 - val_loss: 1.2376 - val_accuracy: 0.7266
Epoch 2/10
43/43 [==============================] - 0s 8ms/step - loss: 0.0266 - accuracy:
0.9881 - val_loss: 1.2415 - val_accuracy: 0.7301
Epoch 3/10
43/43 [==============================] - 0s 9ms/step - loss: 0.0233 - accuracy:
0.9896 - val_loss: 1.2481 - val_accuracy: 0.7301
Epoch 4/10
43/43 [==============================] - 0s 10ms/step - loss: 0.0270 - accuracy:
0.9866 - val_loss: 1.2586 - val_accuracy: 0.7266
Epoch 5/10
43/43 [==============================] - 0s 8ms/step - loss: 0.0155 - accuracy:
0.9941 - val_loss: 1.2717 - val_accuracy: 0.7266
Epoch 6/10
43/43 [==============================] - 0s 4ms/step - loss: 0.0159 - accuracy:
0.9941 - val_loss: 1.2825 - val_accuracy: 0.7266
Epoch 7/10
43/43 [==============================] - 0s 5ms/step - loss: 0.0190 - accuracy:
0.9926 - val_loss: 1.2951 - val_accuracy: 0.7266
Epoch 8/10
43/43 [==============================] - 0s 5ms/step - loss: 0.0235 - accuracy:
0.9926 - val_loss: 1.2981 - val_accuracy: 0.7301
Epoch 9/10
43/43 [==============================] - 0s 5ms/step - loss: 0.0242 - accuracy:
0.9896 - val_loss: 1.3037 - val_accuracy: 0.7266
Epoch 10/10
43/43 [==============================] - 0s 5ms/step - loss: 0.0174 - accuracy:
0.9955 - val_loss: 1.3150 - val_accuracy: 0.7163
```

[193]:
```
plt.plot(history.history['accuracy'])
plt.plot(history.history['val_accuracy'])
plt.title('Model accuracy')
plt.ylabel('Accuracy')
plt.xlabel('Epoch')
plt.legend(['Train', 'Validation'], loc='upper left')
plt.show()
```



Oops! The dataset is simply too small it seems to deal with the regularization. We have a good baseline to compare to the other models now. The last thing to test is using a different optimizer. Adam is best for large datasets, I am going to try different optimizers

1.5 CNN: Convolutional Neural Networks

With the input we gained from the sequential model analysis above, we can use similar settings going into another architecture: CNNs. I imagine that RNN will be better at illustrating the sequential nature of this dataset.

[207]:
```
model = models.Sequential()
model.add(layers.Embedding(2600, 32))
```

```
model.add(layers.Conv1D(32, 7, activation='relu'))
model.add(layers.MaxPooling1D(5))
model.add(layers.Conv1D(32, 7, activation='relu'))
model.add(layers.GlobalMaxPooling1D())
model.add(layers.Dense(1))
```

[208]:
```
model.compile(loss='binary_crossentropy',
              optimizer='adam',
              metrics=['accuracy'])
```

[209]:
```
history = smaller_model.fit(X_train_vec, y_train_enc, validation_split=0.3,␣
↪epochs=10, batch_size=16)
```

```
Epoch 1/10
43/43 [==============================] - 0s 7ms/step - loss: 0.0096 - accuracy:
0.9985 - val_loss: 1.3843 - val_accuracy: 0.7232
Epoch 2/10
43/43 [==============================] - 0s 5ms/step - loss: 0.0254 - accuracy:
0.9896 - val_loss: 1.3849 - val_accuracy: 0.7232
Epoch 3/10
43/43 [==============================] - 0s 5ms/step - loss: 0.0163 - accuracy:
0.9941 - val_loss: 1.3930 - val_accuracy: 0.7232
Epoch 4/10
43/43 [==============================] - 0s 6ms/step - loss: 0.0143 - accuracy:
0.9955 - val_loss: 1.3991 - val_accuracy: 0.7197
Epoch 5/10
43/43 [==============================] - 0s 5ms/step - loss: 0.0189 - accuracy:
0.9955 - val_loss: 1.4075 - val_accuracy: 0.7232
Epoch 6/10
43/43 [==============================] - 0s 6ms/step - loss: 0.0124 - accuracy:
0.9955 - val_loss: 1.4117 - val_accuracy: 0.7197
Epoch 7/10
43/43 [==============================] - 0s 6ms/step - loss: 0.0126 - accuracy:
0.9955 - val_loss: 1.4202 - val_accuracy: 0.7197
Epoch 8/10
43/43 [==============================] - 0s 5ms/step - loss: 0.0130 - accuracy:
0.9941 - val_loss: 1.4263 - val_accuracy: 0.7197
Epoch 9/10
43/43 [==============================] - 0s 5ms/step - loss: 0.0148 - accuracy:
0.9941 - val_loss: 1.4356 - val_accuracy: 0.7232
Epoch 10/10
43/43 [==============================] - 0s 5ms/step - loss: 0.0103 - accuracy:
0.9970 - val_loss: 1.4471 - val_accuracy: 0.7266
```

[210]:
```
from sklearn.metrics import classification_report
```

```
score = smaller_model.evaluate(X_test_vec, y_test_enc, batch_size=batch_size,␣
 ↪verbose=1)
```

```
3/3 [==============================] - 0s 6ms/step - loss: 2.0382 - accuracy:
0.6100
```

Not messing with it that much, I would expect a CNN to overfit this data even more than the basic Sequential Model, so I will be moving on.

1.6 RNN: Recurrent Neural Networks

In order to use an RNN, we have to format the data to be a 3D tensor.The first dimension being the number of sequences or lines in a batch, the second being the max length the number of time steps in an input sequence, and lastly is input_dim, or the numberr of features in the input data.

> "For example, if you have a sequence of 100 words, and each word is represented by a vector of length 50, and you are processing 32 samples in each batch, then the input tensor would have shape (32, 100, 50)."

To do this we will modify our data to fit this format. Our data is currently is just ~900 lines vectorized as a 2519 element long vocab vector. Each element in that vector is a word or term in the input sequence mapped to a numerical value based on the frequency of the term in the sequence and the inverse frequency of the term in the overall dataset.

Our number of samples will be our batch size, our number of features will be the vector for each sample, and the length of time steps is something we get to tune.

[242]:
```
sample_size = X_train_vec.shape[0]
num_features = X_train_vec.shape[1]
time_step_size = 1
print("Our dimension is: ", X_train_vec.shape)
print("Our dimensions should be (", sample_size, ",", time_step_size, ",",␣
 ↪num_features, ")")
```

```
Our dimension is:  (963, 2519)
Our dimensions should be ( 963 , 1 , 2519 )
```

[243]:
```
X_reshape = X_train_vec.reshape((sample_size, time_step_size, num_features))
```

Note we sa

[252]:
```
model = models.Sequential()
model.add(layers.SimpleRNN(32, input_shape=(time_step_size, num_features)))
model.add(Dropout(.2))
model.add(layers.Dense(1, activation='sigmoid'))
```

[253]:
```
model.summary()
```

```
Model: "sequential_44"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
```

```
=================================================================
 simple_rnn_6 (SimpleRNN)    (None, 32)                81664

 dropout_23 (Dropout)        (None, 32)                0

 dense_71 (Dense)            (None, 1)                 33

=================================================================
Total params: 81,697
Trainable params: 81,697
Non-trainable params: 0
_________________________________________________________________
```

[254]:
```
# compile
model.compile(optimizer='adam',
              loss='binary_crossentropy',
              metrics=['accuracy'])
```

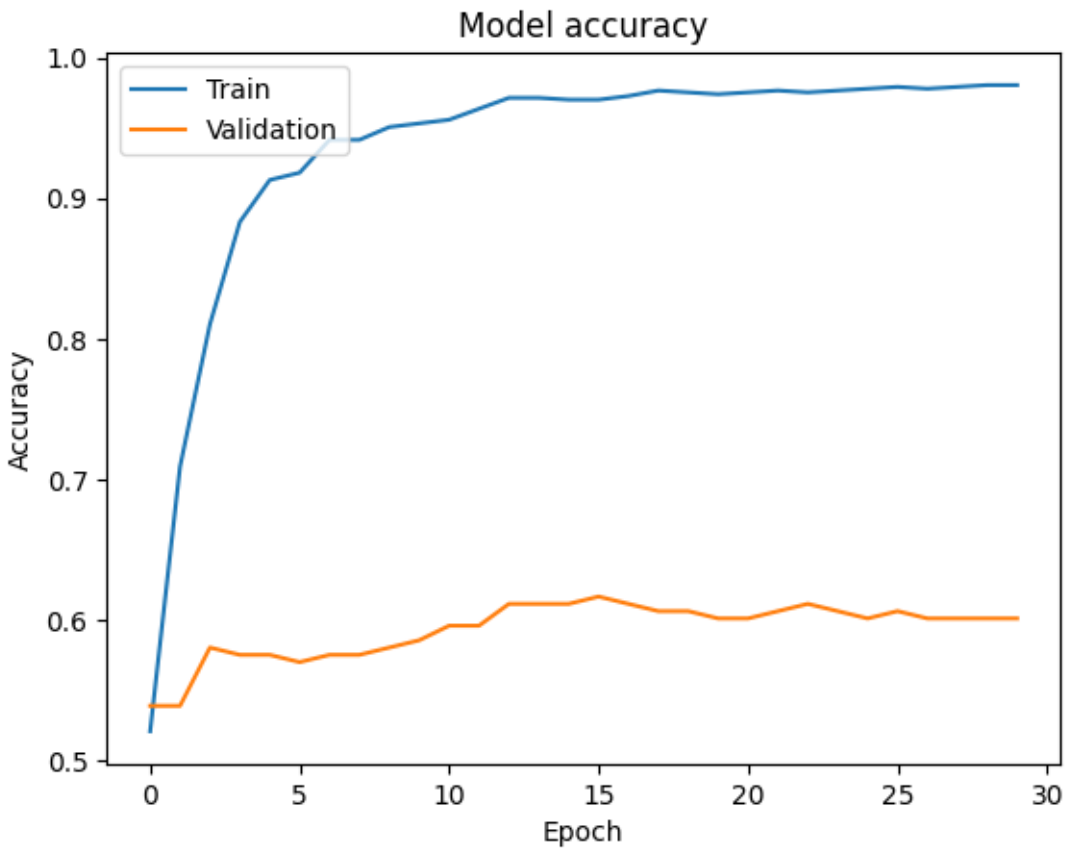
[255]:
```
# train
history = model.fit(X_reshape,
                    y_train_enc,
                    epochs=30,
                    batch_size=128,
                    validation_split=0.2)
```

```
Epoch 1/30
7/7 [==============================] - 2s 58ms/step - loss: 0.7059 - accuracy:
0.5208 - val_loss: 0.7229 - val_accuracy: 0.5389
Epoch 2/30
7/7 [==============================] - 0s 13ms/step - loss: 0.5638 - accuracy:
0.7091 - val_loss: 0.7155 - val_accuracy: 0.5389
Epoch 3/30
7/7 [==============================] - 0s 13ms/step - loss: 0.4787 - accuracy:
0.8104 - val_loss: 0.7089 - val_accuracy: 0.5803
Epoch 4/30
7/7 [==============================] - 0s 14ms/step - loss: 0.4087 - accuracy:
0.8831 - val_loss: 0.7089 - val_accuracy: 0.5751
Epoch 5/30
7/7 [==============================] - 0s 13ms/step - loss: 0.3640 - accuracy:
0.9130 - val_loss: 0.7146 - val_accuracy: 0.5751
Epoch 6/30
7/7 [==============================] - 0s 12ms/step - loss: 0.3247 - accuracy:
0.9182 - val_loss: 0.7226 - val_accuracy: 0.5699
Epoch 7/30
7/7 [==============================] - 0s 13ms/step - loss: 0.2869 - accuracy:
0.9416 - val_loss: 0.7314 - val_accuracy: 0.5751
Epoch 8/30
7/7 [==============================] - 0s 14ms/step - loss: 0.2646 - accuracy:
```

```
0.9416 - val_loss: 0.7429 - val_accuracy: 0.5751
Epoch 9/30
7/7 [==============================] - 0s 15ms/step - loss: 0.2403 - accuracy:
0.9506 - val_loss: 0.7544 - val_accuracy: 0.5803
Epoch 10/30
7/7 [==============================] - 0s 12ms/step - loss: 0.2203 - accuracy:
0.9532 - val_loss: 0.7637 - val_accuracy: 0.5855
Epoch 11/30
7/7 [==============================] - 0s 12ms/step - loss: 0.2045 - accuracy:
0.9558 - val_loss: 0.7732 - val_accuracy: 0.5959
Epoch 12/30
7/7 [==============================] - 0s 12ms/step - loss: 0.1879 - accuracy:
0.9636 - val_loss: 0.7840 - val_accuracy: 0.5959
Epoch 13/30
7/7 [==============================] - 0s 12ms/step - loss: 0.1763 - accuracy:
0.9714 - val_loss: 0.7953 - val_accuracy: 0.6114
Epoch 14/30
7/7 [==============================] - 0s 11ms/step - loss: 0.1649 - accuracy:
0.9714 - val_loss: 0.8078 - val_accuracy: 0.6114
Epoch 15/30
7/7 [==============================] - 0s 14ms/step - loss: 0.1544 - accuracy:
0.9701 - val_loss: 0.8191 - val_accuracy: 0.6114
Epoch 16/30
7/7 [==============================] - 0s 14ms/step - loss: 0.1449 - accuracy:
0.9701 - val_loss: 0.8301 - val_accuracy: 0.6166
Epoch 17/30
7/7 [==============================] - 0s 15ms/step - loss: 0.1381 - accuracy:
0.9727 - val_loss: 0.8418 - val_accuracy: 0.6114
Epoch 18/30
7/7 [==============================] - 0s 14ms/step - loss: 0.1291 - accuracy:
0.9766 - val_loss: 0.8531 - val_accuracy: 0.6062
Epoch 19/30
7/7 [==============================] - 0s 11ms/step - loss: 0.1227 - accuracy:
0.9753 - val_loss: 0.8644 - val_accuracy: 0.6062
Epoch 20/30
7/7 [==============================] - 0s 12ms/step - loss: 0.1169 - accuracy:
0.9740 - val_loss: 0.8775 - val_accuracy: 0.6010
Epoch 21/30
7/7 [==============================] - 0s 14ms/step - loss: 0.1125 - accuracy:
0.9753 - val_loss: 0.8896 - val_accuracy: 0.6010
Epoch 22/30
7/7 [==============================] - 0s 11ms/step - loss: 0.1069 - accuracy:
0.9766 - val_loss: 0.9010 - val_accuracy: 0.6062
Epoch 23/30
7/7 [==============================] - 0s 16ms/step - loss: 0.1035 - accuracy:
0.9753 - val_loss: 0.9125 - val_accuracy: 0.6114
Epoch 24/30
7/7 [==============================] - 0s 13ms/step - loss: 0.0977 - accuracy:
```

```
0.9766 - val_loss: 0.9246 - val_accuracy: 0.6062
Epoch 25/30
7/7 [==============================] - 0s 12ms/step - loss: 0.0930 - accuracy:
0.9779 - val_loss: 0.9363 - val_accuracy: 0.6010
Epoch 26/30
7/7 [==============================] - 0s 11ms/step - loss: 0.0925 - accuracy:
0.9792 - val_loss: 0.9476 - val_accuracy: 0.6062
Epoch 27/30
7/7 [==============================] - 0s 15ms/step - loss: 0.0875 - accuracy:
0.9779 - val_loss: 0.9588 - val_accuracy: 0.6010
Epoch 28/30
7/7 [==============================] - 0s 16ms/step - loss: 0.0843 - accuracy:
0.9792 - val_loss: 0.9696 - val_accuracy: 0.6010
Epoch 29/30
7/7 [==============================] - 0s 14ms/step - loss: 0.0817 - accuracy:
0.9805 - val_loss: 0.9836 - val_accuracy: 0.6010
Epoch 30/30
7/7 [==============================] - 0s 13ms/step - loss: 0.0799 - accuracy:
0.9805 - val_loss: 0.9972 - val_accuracy: 0.6010
```

[256]:
```
plt.plot(history.history['accuracy'])
plt.plot(history.history['val_accuracy'])
plt.title('Model accuracy')
plt.ylabel('Accuracy')
plt.xlabel('Epoch')
plt.legend(['Train', 'Validation'], loc='upper left')
plt.show()
```



Well, I can tell I really need a better dataset because I am getting no improvements in the validation sets. The issue is, there aren't enough Rick Lines out there to really boost this. *Just not enough data*

To be frank, I have only have a little time left. Embeddings are so useful in reshaping the data in a meaningful way, I must move on to them.

1.7 Embedding

In the context of neural networks, an embedding is typically a dense vector of fixed size, which is learned during training of the model. The embedding captures the meaning of the word or phrase, such that words with similar meanings have embeddings that are close together in the embedding space. Each word now has a meaning that can be referenced for use in the model

This could produce better results for our models, but is also quite useful in quickly setting up input for RNNs.

I've realized I need to make sure I use a max sequence length

[263]:
```
embedding_layer = layers.Embedding(X_train_vec.shape[1]+1, 64)
```

The above layer is an embedding we can then feed into our previous neural network. It basically

says we want to encode our dat

[264]:
```
model = models.Sequential()
model.add(embedding_layer)
model.add(layers.SimpleRNN(32))
model.add(Dropout(.2))
model.add(layers.Dense(1, activation='sigmoid'))
```

[265]:
```
# compile
model.compile(optimizer='adam',
              loss='binary_crossentropy',
              metrics=['accuracy'])
```

[267]:
```
# train
history = model.fit(X_train_vec,
                    y_train_enc,
                    epochs=10,
                    batch_size=128,
                    validation_split=0.2)
```

```
Epoch 1/10
7/7 [==============================] - 18s 3s/step - loss: 0.7000 - accuracy:
0.5013 - val_loss: 0.6953 - val_accuracy: 0.5440
Epoch 2/10
7/7 [==============================] - 13s 2s/step - loss: 0.6995 - accuracy:
0.5026 - val_loss: 0.6990 - val_accuracy: 0.5233
Epoch 3/10
7/7 [==============================] - 13s 2s/step - loss: 0.7016 - accuracy:
0.5000 - val_loss: 0.7033 - val_accuracy: 0.4715
Epoch 4/10
7/7 [==============================] - 13s 2s/step - loss: 0.7038 - accuracy:
0.4870 - val_loss: 0.7155 - val_accuracy: 0.4352
Epoch 5/10
7/7 [==============================] - 13s 2s/step - loss: 0.7107 - accuracy:
0.4753 - val_loss: 0.7258 - val_accuracy: 0.3938
Epoch 6/10
7/7 [==============================] - 13s 2s/step - loss: 0.7006 - accuracy:
0.5039 - val_loss: 0.6828 - val_accuracy: 0.5803
Epoch 7/10
7/7 [==============================] - 13s 2s/step - loss: 0.6977 - accuracy:
0.5143 - val_loss: 0.6919 - val_accuracy: 0.5440
Epoch 8/10
7/7 [==============================] - 13s 2s/step - loss: 0.6941 - accuracy:
0.5208 - val_loss: 0.6952 - val_accuracy: 0.4767
Epoch 9/10
7/7 [==============================] - 14s 2s/step - loss: 0.7036 - accuracy:
0.5104 - val_loss: 0.6994 - val_accuracy: 0.4767
Epoch 10/10
```

```
7/7 [==============================] - 17s 2s/step - loss: 0.7068 - accuracy:
0.5039 - val_loss: 0.6986 - val_accuracy: 0.4715
```

Well I'm having trouble figuring out what I have done wrong. I'd like to preload *Glove* weights to see if that improves the embeddings as intended. By all intents and purposes, this should have improved accuracy. So at least I know I've failed lol.

1.8 Analysis

Well this was a bit rushed and I was a bit frustrated. I have a dental abcess that has knocked me out until I can get dental care and am honestly happy with the work I got done here. I guess the reason this frustrated me was that my data ended up just being too small to yield good results. The overall size of the database was a bit too small. And each individual observation had lines that were quite short. I'm sure if I could put more time into this I could provide better results.

If anything, I would expect a SimpleRNN with embedding to be the best preformance here. There is a very strong difference between the way Rick uses language vs the rest of the cast that I would hypothesize would be reflected in the text embedding. Then, with a simple couple timesteps we could reflect the common phrasing he uses.

Yet I struggle to understand how RNN works well on sequential data when it is using a Vectorization. I want to know more. But at the moment it's just not working for me.

I'm actually happy with the preformance of the sequential model. It did better than I expected with what I gave it. If I could create a more varied dataset with word usage that differs a lot more from how the Rick and Morty script is written, I could imagine this working a lot better.

I… I've just lost my motivation this week.