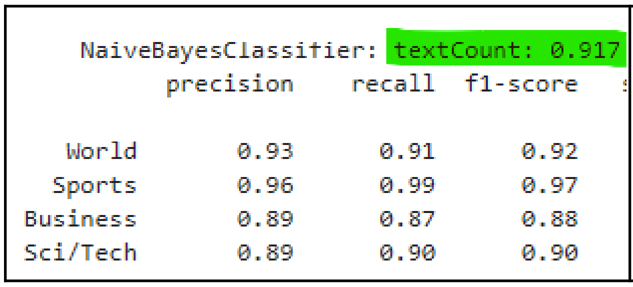
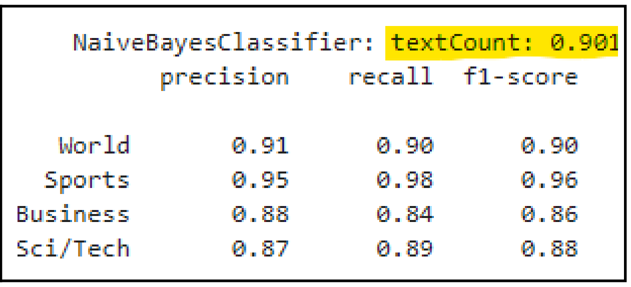
Assignment 1: Text Representation and Classification **Deveremma 300602434**

Step 1 (15%) Build a text classifier using Term Frequency (TF) and a classic learning algorithm. (Decision Trees, Naïve Bayes, KNearestNeighbours). Report the accuracy for both the training and testing sets. You must use both the Title and Description as the text input for each instance.

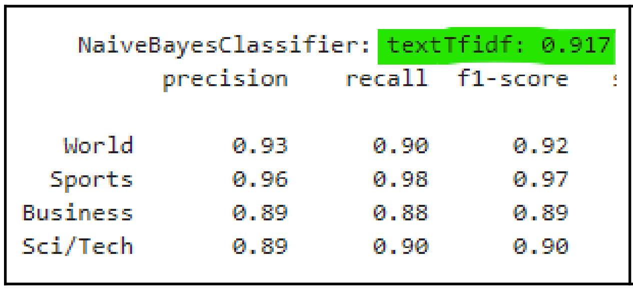


```
    NaiveBayesClassifier: textCount: 0.917
              precision    recall  f1-score

       World       0.93      0.91      0.92
      Sports       0.96      0.99      0.97
    Business       0.89      0.87      0.88
    Sci/Tech       0.89      0.90      0.90
```

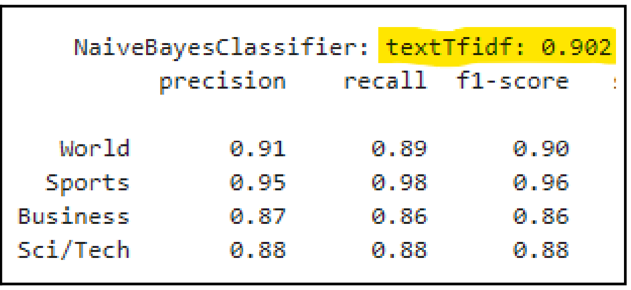


```
    NaiveBayesClassifier: textCount: 0.901
              precision    recall  f1-score

       World       0.91      0.90      0.90
      Sports       0.95      0.98      0.96
    Business       0.88      0.84      0.86
    Sci/Tech       0.87      0.89      0.88
```

Step 2 (10%) Build another text classifier by changing the text representation to TF-IDF while keeping the same classifier used in Step 1. Report the classification performance on the testing data, including the overall accuracy and the Precision, Recall, and F1 score for each class.

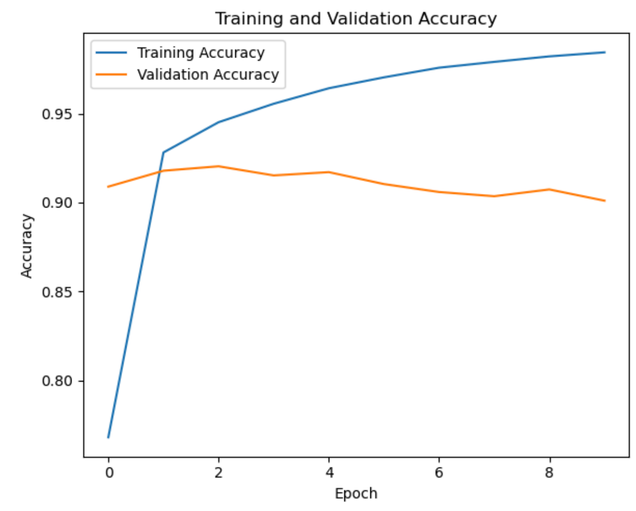


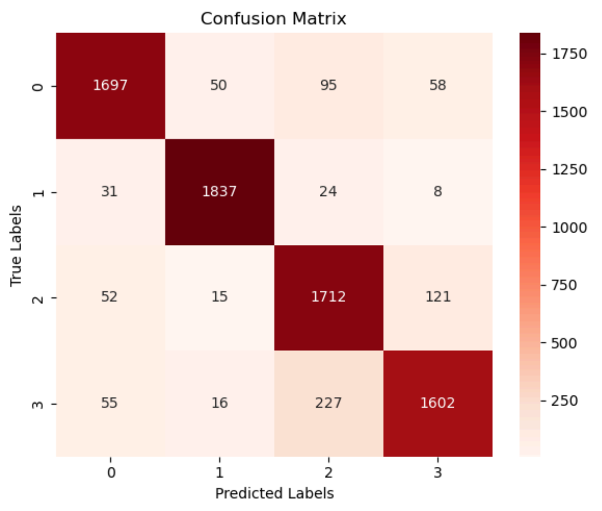
```
    NaiveBayesClassifier: textTfidf: 0.917
              precision    recall  f1-score

       World       0.93      0.90      0.92
      Sports       0.96      0.98      0.97
    Business       0.89      0.88      0.89
    Sci/Tech       0.89      0.90      0.90
```



```
    NaiveBayesClassifier: textTfidf: 0.902
              precision    recall  f1-score

       World       0.91      0.89      0.90
      Sports       0.95      0.98      0.96
    Business       0.87      0.86      0.86
    Sci/Tech       0.88      0.88      0.88
```

Step 3 (20%) Build a text classifier using a Convolutional Neural Network (CNN). Initialize the embedding layer with random numbers and ensure it is trainable. Plot a figure showing how the accuracy changes during the training process for both the training and testing data.

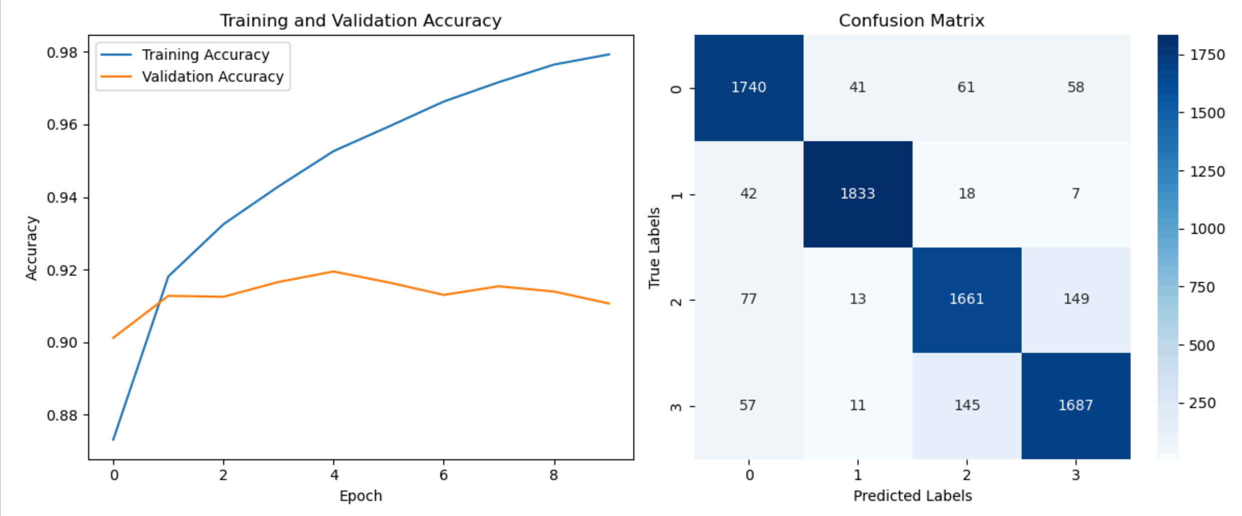




	0	1	2	3
0	1697	50	95	58
1	31	1837	24	8
2	52	15	1712	121
3	55	16	227	1602

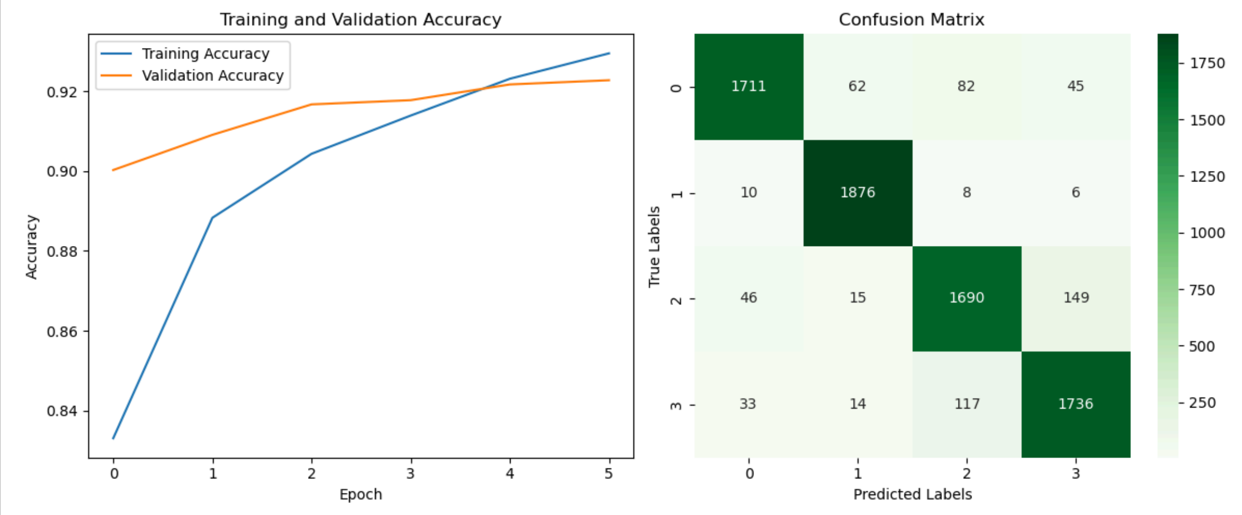
Training Accuracy: 0.9863 Testing Accuracy: 0.9012 Validation Loss: 0.4057

Step 4 (20%) Create the CNN-based text classifier using pre-trained word embeddings such as GloVe. Report the accuracy on the testing data and draw the confusion matrix for this model.



Training Accuracy: 0.9815 Testing Accuracy: 0.9107 Validation Loss: 0.3397

Step 5 (20%) Create a third version of the CNN-based classifier by applying techniques such as data preprocessing, hyperparameter tuning, changing classifiers, or text representations. The testing data must not be used for tuning. List the Details of the improvements made in Step 5.



Training Accuracy: 0.9312 Testing Accuracy: 0.9228 Validation Loss: 0.3133

Improvements: Data Preprocessing

Converted every word to lowercase to prevent “Call” and “call” from being 2 different entities.

Removed stopping words, like the, a, for etc. in order to focus on key discriminating words.

Removed Punctuation as it adds unnecessary complexity to analysis given our simple project.

Removed words that contain digits to keep uniformity and handle noise i.e URL, dates etc.

Improvements: Hyperparameter Tuning:

Dimensions: Tested {50, 100, 300} Selected **50** #100 and esp 300 results in train acc overfitting

Trainable: Tested {False, True} Selected **True** #False reduces train acc but True pushes val acc

Batch Size: Tested {256, 64, 48, 32} Selected **48** #Lower values push val acc but longer waiting

Learning Rate: Tested {1e-4, 5e-4, 1e-3} Selected **5e-4** #Higher pushes val acc but its unstable

Dropout: Tested {0, 0.3, 0.5} Selected **0.5** #Reduces train acc overfitting & keeps val climbing

Dropout location: Tested {Pre-Dense, Post-Dense} Selected **Post-Dense** #Post is anti-overfit

Weight Decay: Tested {1e-4, 1e-3} Selected **1e-3** #Higher appears to improve generalizability

Dense Layer Size: Tested {8, 16, 24, 32} Selected **24** #8 is very underfit, 32 is fairly overfitting

No. Conv1D Layers: Tested {1, 2, 3} Selected **1** #3 is similar to 1 but greatly increases waiting

Filter sizes: Tested {64, 128, 256} Selected **256** #Higher values push val acc but longer waiting

Kernel Sizes: Tested {3, 5, 7, 9} Selected **3** #9 greatly overfits train acc, 3 reduces overfitting

Batch Normalization: Tested {none, Post-Conv1D} Selected **none** #Too strong, reduces acc

Epochs: Tested {5, 6, 7, 10} Selected **6** #Acts like early stopping to prevent further overfitting

maxLen: Tested {50, 70, 100} Selected **100** #little difference, anecdotal higher gives better acc

Activation Function: Tested {relu, leaky_relu} Selected **relu** #Leaky_relu greatly overfits data

Pooling: Tested {GlobalMaxPooling1D, GlobalAveragePooling1D} chose **GlobalMaxPooling1D**

Optimizers: Tested {Adam, SGD, RMSprop} Selected **Adam** #SGD & RMSprop make up Adam

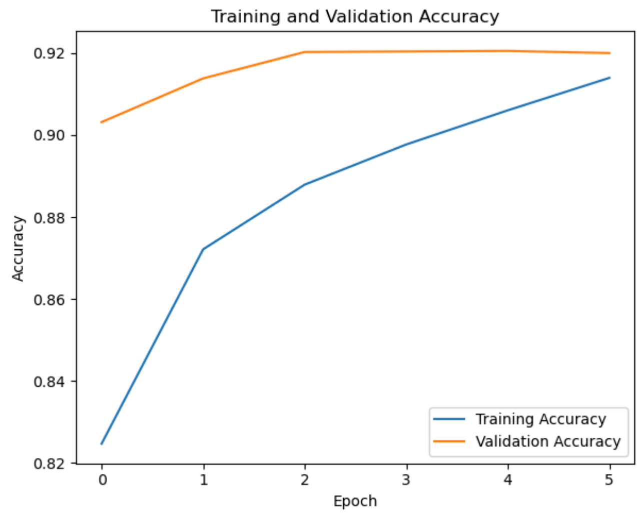
Improvements: Techniques:

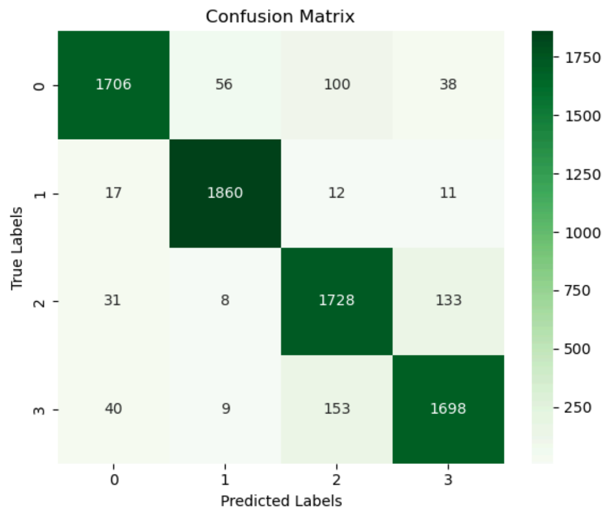
Classifiers: Tested {CNN} Selected **CNN** #Best to keep it for comparison and to stay in scope

Search: Tested {GridSearch, RandomSearch} Selected **None** #Won't recognize KerasClassifier

Embeddings: Tested {GloVe} Selected **GloVe** #Stable implementation although limited vocab.

The goals were to reduce overfitting and push the validation accuracy higher without oscillation.

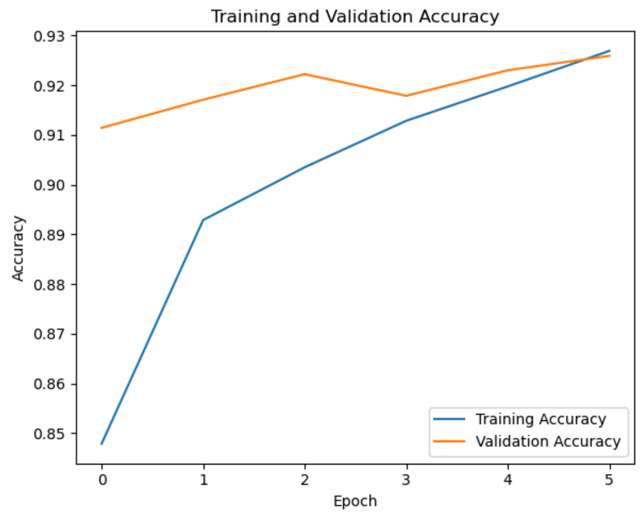


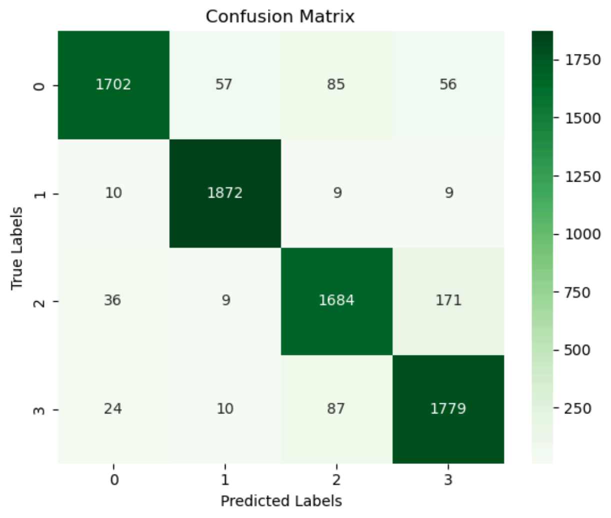


```
train_seq_x = sequence.pad_sequences(token.texts_to_sequences(trainX), maxlen=100)
test_seq_x = sequence.pad_sequences(token.texts_to_sequences(testX), maxlen=100)
train_seq_x[0]

dimensions = 50
embedding_matrix = create_embedding_matrix(f'glove.6B.{dimensions}d.txt', token.word_index, dimensions)
print(numpy.count_nonzero(numpy.count_nonzero(embedding_matrix, axis=1)) / vocab_size)
```

```
Embedding Matrix Complete
0.8062490448603065
```

```
from tensorflow.keras.optimizers import Adam
from tensorflow.keras.regularizers import l2

# Build CNN model with pre-trained embeddings
model3 = Sequential()
model3.add(layers.Embedding(vocab_size, dimensions, weights=[embedding_matrix], trainable=True))
model3.add(layers.Conv1D(filters=512, kernel_size=3, activation='relu', kernel_regularizer=l2(0.001)))
model3.add(layers.GlobalMaxPooling1D())
model3.add(layers.Dense(24, activation='relu', kernel_regularizer=l2(0.001)))
model3.add(layers.Dropout(0.5))
model3.add(layers.Dense(4, activation='softmax'))

model3.compile(optimizer=Adam(learning_rate=0.0005), loss='sparse_categorical_crossentropy', metrics=['acc'])
history = model3.fit(train_seq_x, trainY, epochs=6, batch_size=48, validation_data=(test_seq_x, testY), verbose=1)

results = model3.evaluate(test_seq_x, testY)
print(f'\n[val_loss, val_accuracy]\n{results}')
plotResults(model3, 'Greens')
```





Step 6 (5%) Display a summary that reports the test accuracy of all models from Steps 1 to 5. Provide a summary of the comparison results from Step 6, along with a ***brief analysis*** (why)

	Training Accuracy	**Testing Accuracy**	**Description**
Step 1	0.917 (Naive Bayes)	0.901 (Naive Bayes)	Term Frequency
Step 1	0.792 (KNearestNeighbor)	0.694 (KNearestNeighbor)	Term Frequency
Step 2	0.917 (Naive Bayes)	0.902 (Naive Bayes)	TF-IDF Representation
Step 2	0.931 (KNearestNeighbor)	0.902 (KNearestNeighbor)	TF-IDF Representation
Step 3	0.9863	0.9012	CNN without GloVe
Step 4	0.9815	0.9107	CNN with GloVe
Step 5	0.9312	0.9228	Tuned CNN with GloVe

Why Naive Bayes performed well on both Term Frequency and TF-IDF Representations

Naive Bayes works by assuming independence between words. While this makes it naive as this assumption is not always true, it so happens to work well in high dimensional sparse data, much like our text data. Furthermore, Naive Bayes only requires a good estimate of word occurrences to calculate the likelihood of a class. Since Term frequency provides a direct measure of word importance, Naive Bayes immediately capitalizes on this and scores 90%.

Naive Bayes' simplicity, in that it really only needs frequency for its probability estimation means that applying TF-IDF does not significantly alter its effectiveness. TF-IDF downweighs common words, but in Naïve Bayes, even common words can contribute valuable probability information.

Since NB worked inherently well for both Term Frequency and TF-IDF, almost boringly, I decided to test KNearestNeighbor. Here it responds better under TF-IDF because its weighting (reducing common words) helps KNN's distance-based model focus on crucial words vs jarring distances.

Key points and observations from the Precision, Recall, and F1 scores of each class

- The scores of the sports category are consistently the highest, measuring above 0.95
- Oppositely, Business is consistently the lowest, around 0.88, with Sci/Tech barely above

- Interestingly, World and Business' precision is consistently higher than its recall scores. This means when World or Business is predicted, it is usually right, but less frequently. This could be because of distinct vocabulary: geopolitical, countries, finance, stocks

- Oppositely, Sport and Sci/Tech's recall is consistently higher than its precision scores. This means most predictions are for Sport and Sci/Tech, despite being World/Business. This could be because of generalized vocabulary: Olympics, win, gains, Tesla AI Stock

- As expected, f1-score approximates the mean of precision and recall (as per formula)

Why CNN without pre-trained word embeddings performed similarly to NB TF and TF-IDF

CNNs use convolutional filters to detect patterns of word sequences, similar to how they detect patterns in images. Word sequences are arguably a step up from TF and TF-IDF which capture word importance which is why CNN is able to perform highly at the 90% level. Unfortunately, so does TF and TF-IDF. If the dataset lacks sufficient diversity or complexity, CNN might capture less patterns, whereas TF and TF-IDF can still obtain strong representations in their simplicity.

Regardless, CNN still performs on par with TF and TF-IDF, even without pre-trained word embeddings. This is because initial embeddings can still capture basic information about the words (frequency and co-occurrence) despite not being semantically rich like GloVe. Likewise, through numerous epochs of training, these initial embeddings can be refined, potentially converging to similar values as those in pre-trained, in which case, there's small difference.

Why CNN with GloVe performed slightly better than CNN with random embedding

CNNs are often combined with pre-trained word embeddings (GloVe), which provide a dense representation where similar words are represented by similar vectors. We see the effects of GloVe's embeddings in the plot of Accuracy vs Epoch where training accuracy is much higher in the first epoch 87% (Step 4) compared to random word embeddings at 75% (Step 3). In a way, GloVe helps to kickstart the CNN performance and within the same 10 epochs, is able to push validation accuracy just a fair bit higher compared to the CNN without pre-trained embeddings.

As discussed previously, initial random embeddings will eventually be refined, which is why the performance after 10 epochs is not substantially greater. Moreover, If the training data has a different vocabulary compared to GloVe's embeddings, the advantage of GloVe diminishes. From my code, 80% of the data vocabulary is covered by GloVe, 20% no different to random.

Key points and observations from the Confusion Matrices in Steps 3, 4, and 5

- All three confusion matrices actually follow a similar structure e.g. 2 is often predicted as 3, or 3 may often be predicted as 2. It's just that step 5 matrix has less misclassifications. This is identical to my discussion on precision/recall about dynamics between classes.

Why the changes listed in step 5 resulted in higher testing accuracy than previous steps.

- **Better regularization**: Through dropout and L2 regularization weight decay we reduce overfitting on train data. There's a fine line –higher training reduces testing accuracy
- **Hyperparameter optimization**: Adjustments such as learning rate, batches, dense layer, and filters were finely tuned to push the testing accuracy higher but remain stable.
- **Data Preprocessing**: Removing common stopping words, punctuation, and words with digits helps to reduce noise and ensure the model focuses on distinguishing words.

Unfortunately, despite my attempts, I could not surpass 93% accuracy. This could be because of a genuine upper limit where the fine line that separates the classes e.g. Business and Sci/Tech is in ambiguous/conflicting territory. This is more to do with the data itself. We can see glimpses of this in my discussion on precision/recall/confusion matrices.Regardless, 92.5% is satisfactory.