###B) **FastText**

```
[ ]: from gensim.models import FastText
     from nltk.tokenize import word_tokenize

     # Tokenize the text
     sentences = [word_tokenize(doc.lower()) for doc in data['clean_text']]

     # Train FastText model
     model = FastText(sentences, vector_size=100, window=5, min_count=1, workers=4)

     # Get word vectors
     word_vectors = model.wv

     # Get the combined matrix of word vectors
     fasttext_matrix = word_vectors.vectors
     print(fasttext_matrix)
```

```
[[-0.92287266  0.14702174 -0.79065204 … -0.13592456  0.3070964
   0.14550059]
 [-1.2960643  -0.11427958 -0.99751246 …  0.23815921  0.97278845
   0.26405624]
 [-1.1532832   0.02814879 -0.8235454  …  0.03156775  0.03897018
   0.0059722 ]
 …
 [-0.16125236 -0.02818967 -0.47869277 … -0.15219589 -0.01365738
   0.31606424]
 [-0.16397035 -0.05725078 -0.47975725 … -0.13754298 -0.01615353
   0.32184893]
 [-0.12186304 -0.10724075 -0.5856966  … -0.19292068 -0.12182381
   0.45474076]]
```

5 Modeling

The model features used in these project are going to be * **LSTM Model** * **Simple RNN Model**

###A) **LSTM Modeling**

```
[ ]: #import libraries for deep learning

     from sklearn.model_selection import train_test_split
     from tensorflow.keras.preprocessing.text import Tokenizer
     from tensorflow.keras.preprocessing.sequence import pad_sequences
     from tensorflow.keras.models import Sequential
     from tensorflow.keras.layers import Embedding, LSTM, Dense, SpatialDropout1D,␣
      ↪Dropout
     from tensorflow.keras.callbacks import EarlyStopping
     from sklearn.metrics import classification_report, confusion_matrix, roc_curve,␣
       ↪roc_auc_score
     import seaborn as sns
     import matplotlib.pyplot as plt
```

[ ]:
```
#Split the data into training and testing data
X = clean_text.values
y = data['labeled']
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,␣
  ↪random_state=42)
```

[ ]:
```
#Words, length and embedding values to be used for tokenization
MAX_NB_WORDS = 50000
MAX_SEQUENCE_LENGTH = 250
EMBEDDING_DIM = 100
```

[ ]:
```
# Tokenization of the splitted data
tokenizer = Tokenizer(num_words=MAX_NB_WORDS)
tokenizer.fit_on_texts(X_train)
X_train_sequences = tokenizer.texts_to_sequences(X_train)
X_test_sequences = tokenizer.texts_to_sequences(X_test)
```

[ ]:
```
#Padding of the splitted data
X_train_padded = pad_sequences(X_train_sequences, maxlen=MAX_SEQUENCE_LENGTH)
X_test_padded = pad_sequences(X_test_sequences, maxlen=MAX_SEQUENCE_LENGTH)
```

[ ]:
```
#defining the lstm model

model_lstm = Sequential()
model_lstm.add(Embedding(MAX_NB_WORDS, EMBEDDING_DIM,␣
  ↪input_length=MAX_SEQUENCE_LENGTH))
model_lstm.add(SpatialDropout1D(0.2))
model_lstm.add(LSTM(100, dropout=0.2, recurrent_dropout=0.2))
model_lstm.add(Dense(3, activation='softmax'))
```

[ ]:
```
# compile the model
model_lstm.compile(loss='sparse_categorical_crossentropy', optimizer='adam',␣
  ↪metrics=['accuracy'])
```

[ ]:
```
#Initiate early stopping
early_stopping = EarlyStopping(monitor='val_loss', patience=3, min_delta=0.
  ↪0001, restore_best_weights=True)
```

[ ]:
```
#define the epochs and batch_size to be used
epochs = 10
batch_size = 64
```

```
[ ]: #train the model
     history = model_lstm.fit(X_train_padded, y_train, epochs=epochs,␣
      ↪batch_size=batch_size, validation_data=(X_test_padded, y_test),␣
      ↪callbacks=[early_stopping])
```

```
Epoch 1/10
211/211 [==============================] - 150s 696ms/step - loss: 0.4603 -
accuracy: 0.8043 - val_loss: 0.3221 - val_accuracy: 0.8833
Epoch 2/10
211/211 [==============================] - 145s 689ms/step - loss: 0.2135 -
accuracy: 0.9199 - val_loss: 0.2974 - val_accuracy: 0.8925
Epoch 3/10
211/211 [==============================] - 167s 792ms/step - loss: 0.1577 -
accuracy: 0.9440 - val_loss: 0.2821 - val_accuracy: 0.9026
Epoch 4/10
211/211 [==============================] - 170s 804ms/step - loss: 0.1224 -
accuracy: 0.9588 - val_loss: 0.3246 - val_accuracy: 0.8904
Epoch 5/10
211/211 [==============================] - 148s 702ms/step - loss: 0.0977 -
accuracy: 0.9675 - val_loss: 0.2917 - val_accuracy: 0.8981
Epoch 6/10
211/211 [==============================] - 140s 663ms/step - loss: 0.0770 -
accuracy: 0.9748 - val_loss: 0.3333 - val_accuracy: 0.9058
```

```
[ ]: #Evaluate the model
     loss, accuracy = model_lstm.evaluate(X_test_padded, y_test, verbose=2)
     print(f'Test Accuracy: {accuracy}')
```

```
106/106 - 7s - loss: 0.2821 - accuracy: 0.9026 - 7s/epoch - 63ms/step
Test Accuracy: 0.9025762677192688
```

The Lstm Model, test accuracy is 0.902 which indicates a good performance to this model.

###B)**Simple RNN Model**

```
[ ]: #import Libraries
     from tensorflow.keras.layers import Embedding, SimpleRNN, Dense
     from tensorflow.keras.preprocessing.sequence import pad_sequences
     from tensorflow.keras.models import Sequential

     # Define the model
     model_rnn = Sequential()
     model_rnn.add(Embedding(MAX_NB_WORDS, EMBEDDING_DIM,␣
      ↪input_length=MAX_SEQUENCE_LENGTH))
     model_rnn.add(SimpleRNN(100, dropout=0.2, recurrent_dropout=0.2))
     model_rnn.add(Dense(3, activation='softmax'))

     # Compile the model
```

```
model_rnn.compile(loss='sparse_categorical_crossentropy', optimizer='adam',␣
 ↪metrics=['accuracy'])

# Train the model
history = model_rnn.fit(X_train_padded, y_train, epochs=epochs,␣
 ↪batch_size=batch_size, validation_data=(X_test_padded, y_test),␣
 ↪callbacks=[early_stopping])

# Evaluate the model
loss, accuracy = model_rnn.evaluate(X_test_padded, y_test, verbose=2)
print(f'Test Accuracy: {accuracy}')
```

```
Epoch 1/10
211/211 [==============================] - 49s 226ms/step - loss: 0.5991 -
accuracy: 0.7399 - val_loss: 0.5317 - val_accuracy: 0.7569
Epoch 2/10
211/211 [==============================] - 45s 215ms/step - loss: 0.4306 -
accuracy: 0.8029 - val_loss: 0.4181 - val_accuracy: 0.8134
Epoch 3/10
211/211 [==============================] - 45s 211ms/step - loss: 0.2548 -
accuracy: 0.9010 - val_loss: 0.3675 - val_accuracy: 0.8659
Epoch 4/10
211/211 [==============================] - 45s 215ms/step - loss: 0.2020 -
accuracy: 0.9291 - val_loss: 0.5072 - val_accuracy: 0.8282
Epoch 5/10
211/211 [==============================] - 44s 210ms/step - loss: 0.1763 -
accuracy: 0.9385 - val_loss: 0.4282 - val_accuracy: 0.8605
Epoch 6/10
211/211 [==============================] - 47s 224ms/step - loss: 0.1496 -
accuracy: 0.9486 - val_loss: 0.5561 - val_accuracy: 0.8365
106/106 - 3s - loss: 0.3675 - accuracy: 0.8659 - 3s/epoch - 24ms/step
Test Accuracy: 0.8658572435379028
```

The Test accuracy of the simple RNN model is at 0.865 which is lower than the LSTM model performance at 0.90

6 Model Evaluation

###Roc/AUC Curve Comparison for LSTM and Simple RNN models

We will proceed check on ROC and AUC metrics for both models to see the performance.

The models are evaluated using accuracy scores, and ROC curves to ensure they accurately classify the sentiment of the reviews.

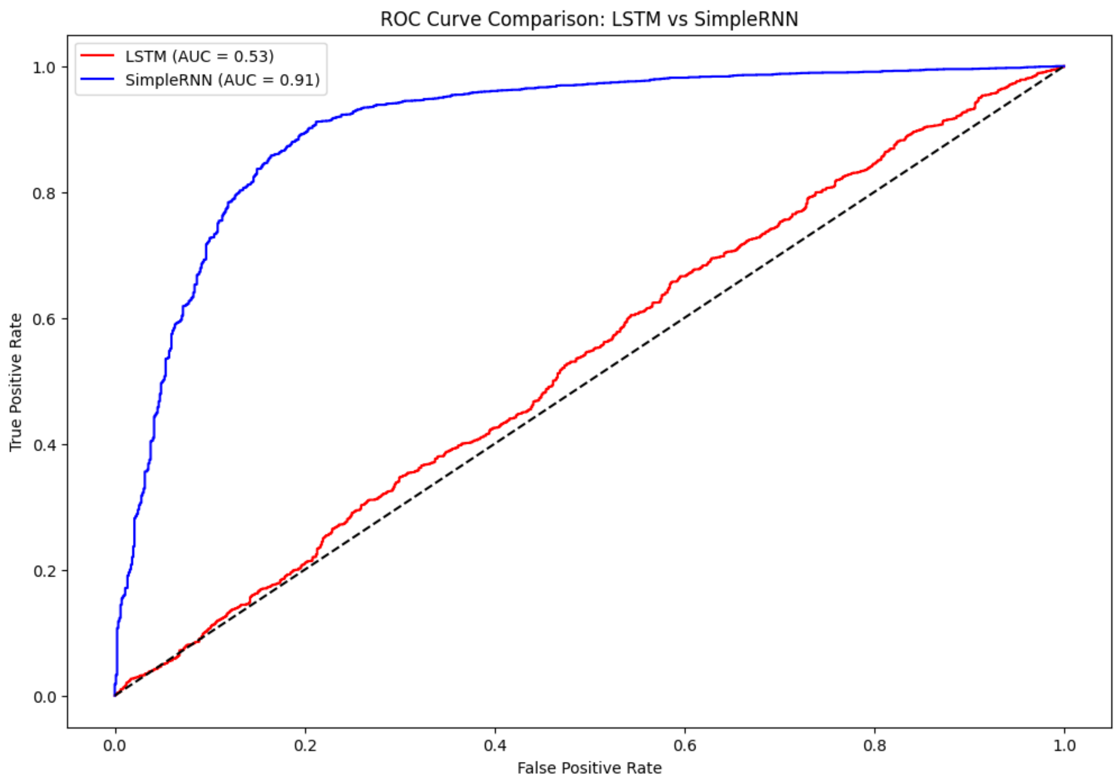
[ ]:
```
#import libraries
from sklearn import metrics
import matplotlib.pyplot as plt
```

```python
# Get predicted probabilities for both models
y_pred_prob_lstm = model_lstm.predict(X_test_padded)
y_pred_prob_rnn = model_rnn.predict(X_test_padded)

# Calculate ROC curves and AUC metrics
fpr_lstm, tpr_lstm, thresholds_lstm = metrics.roc_curve(y_test,␣
 ↪y_pred_prob_lstm[:, 1])
fpr_rnn, tpr_rnn, thresholds_rnn = metrics.roc_curve(y_test, y_pred_prob_rnn[:,␣
 ↪1])
auc_lstm = metrics.auc(fpr_lstm, tpr_lstm)
auc_rnn = metrics.auc(fpr_rnn, tpr_rnn)

# Plot ROC curves
plt.figure(figsize=(12, 8))
plt.plot(fpr_lstm, tpr_lstm, color='red', label='LSTM (AUC = %0.2f)' % auc_lstm)
plt.plot(fpr_rnn, tpr_rnn, color='blue', label='SimpleRNN (AUC = %0.2f)' %␣
 ↪auc_rnn)
plt.plot([0, 1], [0, 1], color='black', linestyle='--')
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')
plt.title('ROC Curve Comparison: LSTM vs SimpleRNN')
plt.legend()
plt.show()
```

```
106/106 [==============================] - 11s 94ms/step
106/106 [==============================] - 2s 19ms/step
```



- **SimpleRNN Model**: With an AUC of 0.91, the SimpleRNN model is performing much better than the LSTM model in distinguishing between the classes. It has a high true positive rate and a low false positive rate across various thresholds.
- **LSTM Model**: With an AUC of 0.53, the LSTM model is only marginally better than random guessing. This indicates that the LSTM model is not very effective for this particular task or dataset.**bold text**

###**Model Hyperparameter Tuning**

From the ROC_AUC curve simple RNN has a better AUC score than the LSTM, will proceed to continue with tuning the model to get a better accuracy.

[ ]:
```
!pip install keras-tuner
from tensorflow.keras.layers import Embedding, SimpleRNN, Dense, LSTM
from tensorflow.keras.preprocessing.sequence import pad_sequences
from tensorflow.keras.models import Sequential
from kerastuner.tuners import RandomSearch
import tensorflow as tf


# Define the model-building function
def build_model(hp):
    model = Sequential()
    model.add(Embedding(MAX_NB_WORDS, EMBEDDING_DIM,␣
↪input_length=MAX_SEQUENCE_LENGTH))

    # Tune the number of units in the SimpleRNN layer
    rnn_units = hp.Int('units', min_value=50, max_value=200, step=50)
    model.add(SimpleRNN(rnn_units, dropout=hp.Float('dropout', 0.1, 0.5, step=0.
↪1), recurrent_dropout=hp.Float('recurrent_dropout', 0.1, 0.5, step=0.1)))

    # Tune the learning rate for the optimizer
    learning_rate = hp.Choice('learning_rate', values=[1e-2, 1e-3, 1e-4])

    model.add(Dense(3, activation='softmax'))
    model.compile(loss='sparse_categorical_crossentropy', optimizer=tf.keras.
↪optimizers.Adam(learning_rate=learning_rate), metrics=['accuracy'])

    return model

# Set up the RandomSearch tuner
tuner = RandomSearch(
    build_model,
    objective='val_accuracy',
    max_trials=5,
    executions_per_trial=3,
    directory='hyperparam_tuning',
    project_name='rnn_tuning'
)

# Run the hyperparameter search
tuner.search(X_train_padded, y_train, epochs=epochs, batch_size=batch_size,␣
↪validation_data=(X_test_padded, y_test), callbacks=[early_stopping])

# Get the optimal hyperparameters
best_hps = tuner.get_best_hyperparameters(num_trials=1)[0]
print(f"""
The hyperparameter search is complete. The optimal number of units in the␣
↪SimpleRNN layer is {best_hps.get('units')},
the optimal dropout rate is {best_hps.get('dropout')}, the optimal recurrent␣
↪dropout rate is {best_hps.get('recurrent_dropout')},
and the optimal learning rate for the optimizer is {best_hps.
↪get('learning_rate')}.
""")

# Build the model with the optimal hyperparameters and train it
model_rnn = tuner.hypermodel.build(best_hps)
```

```python
history = model_rnn.fit(X_train_padded, y_train, epochs=epochs,␣
 ↪batch_size=batch_size, validation_data=(X_test_padded, y_test),␣
 ↪callbacks=[early_stopping])

# Evaluate the model
loss, accuracy = model_rnn.evaluate(X_test_padded, y_test, verbose=2)
print(f'Test Accuracy: {accuracy}')
```

```
Requirement already satisfied: keras-tuner in /usr/local/lib/python3.10/dist-
packages (1.4.7)
Requirement already satisfied: keras in /usr/local/lib/python3.10/dist-packages
(from keras-tuner) (2.15.0)
Requirement already satisfied: packaging in /usr/local/lib/python3.10/dist-
packages (from keras-tuner) (24.1)
Requirement already satisfied: requests in /usr/local/lib/python3.10/dist-
packages (from keras-tuner) (2.31.0)
Requirement already satisfied: kt-legacy in /usr/local/lib/python3.10/dist-
packages (from keras-tuner) (1.0.5)
Requirement already satisfied: charset-normalizer<4,>=2 in
/usr/local/lib/python3.10/dist-packages (from requests->keras-tuner) (3.3.2)
Requirement already satisfied: idna<4,>=2.5 in /usr/local/lib/python3.10/dist-
packages (from requests->keras-tuner) (3.7)
Requirement already satisfied: urllib3<3,>=1.21.1 in
/usr/local/lib/python3.10/dist-packages (from requests->keras-tuner) (2.0.7)
Requirement already satisfied: certifi>=2017.4.17 in
/usr/local/lib/python3.10/dist-packages (from requests->keras-tuner) (2024.6.2)
Reloading Tuner from hyperparam_tuning/rnn_tuning/tuner0.json

The hyperparameter search is complete. The optimal number of units in the
SimpleRNN layer is 200,
the optimal dropout rate is 0.4, the optimal recurrent dropout rate is 0.1,
and the optimal learning rate for the optimizer is 0.0001.

Epoch 1/10
211/211 [==============================] - 111s 512ms/step - loss: 0.5913 -
accuracy: 0.7501 - val_loss: 0.5641 - val_accuracy: 0.7569
Epoch 2/10
211/211 [==============================] - 89s 420ms/step - loss: 0.5528 -
accuracy: 0.7664 - val_loss: 0.5502 - val_accuracy: 0.7569
Epoch 3/10
211/211 [==============================] - 91s 431ms/step - loss: 0.5414 -
accuracy: 0.7663 - val_loss: 0.5469 - val_accuracy: 0.7569
Epoch 4/10
211/211 [==============================] - 87s 412ms/step - loss: 0.5276 -
accuracy: 0.7664 - val_loss: 0.5002 - val_accuracy: 0.7569
Epoch 5/10
211/211 [==============================] - 91s 429ms/step - loss: 0.4330 -
```

```
accuracy: 0.8014 - val_loss: 0.4213 - val_accuracy: 0.8084
Epoch 6/10
211/211 [==============================] - 86s 407ms/step - loss: 0.3259 -
accuracy: 0.8606 - val_loss: 0.3257 - val_accuracy: 0.8697
Epoch 7/10
211/211 [==============================] - 92s 434ms/step - loss: 0.2730 -
accuracy: 0.8893 - val_loss: 0.3124 - val_accuracy: 0.8818
Epoch 8/10
211/211 [==============================] - 85s 404ms/step - loss: 0.2444 -
accuracy: 0.9042 - val_loss: 0.3191 - val_accuracy: 0.8765
Epoch 9/10
211/211 [==============================] - 91s 430ms/step - loss: 0.2206 -
accuracy: 0.9180 - val_loss: 0.3093 - val_accuracy: 0.8990
Epoch 10/10
211/211 [==============================] - 86s 408ms/step - loss: 0.2033 -
accuracy: 0.9265 - val_loss: 0.3350 - val_accuracy: 0.8694
106/106 - 4s - loss: 0.3350 - accuracy: 0.8694 - 4s/epoch - 34ms/step
Test Accuracy: 0.8694106936454773
```

The Model tuning for simple RNN Accuracy score is 0.8694 which has no huge difference from the model before tuning which was 0.8658.

The score is slightly higher than our objective of achieving 0.85 accurracy score. The model is therefore is satisfactory.

7 Reccommendations and Conclusion

- **Customer feedback**: The overall feedback is positive. It's therefore, essential to continue monitoring and encouraging positive customer experiences. This can be achieved through maintaining product quality, enhancing customer service, and soliciting feedback from satisfied customers to bolster positive reviews.
- **Product preference**: We recommend investing in the most preferred product categories by expanding product lines, improving features based on customer feedback, and maintaining competitive pricing to sustain positive customer sentiment.
- **Trend Analysis**: Periodic spikes or dips may indicate specific product launches, updates, or marketing campaigns. Consider correlating these fluctuations with internal events to identify factors influencing customer sentiment and adjust strategies accordingly.
- **User Experience**: We recommend encouraging customers to leave detailed and informative reviews by incentivizing feedback or providing clear guidelines on what constitutes a helpful review. This will help stakeholders to highlight these reviews prominently to enhance trust and credibility among prospective buyers.
- **This sentiment analysis system** provides a scalable and automated way to interpret vast amounts of customer feedback. By leveraging NLP and deep learning, businesses can gain valuable insights to improve their products and services, ultimately enhancing customer satisfact