Universidad Icesi

Facultad de Ingeniería, Diseño y Ciencias Aplicadas

Departamento de Computación y Sistemas Inteligentes

Computing and Discrete Structures III

Project report:

Sentiment analysis model using supervised learning with vanilla Recurrent Neural Networks and LSTM.

Team members:

Diana Lorena Balanta Solano

Carlos Javier Bolaños Riascos

Danna Alexandra Espinosa Arenas

Professor:

Ph.D Andrés Alberto Aristizabal Pinzón

Santiago de Cali, 24 november 2023

Abstract

This project focuses on the application of supervised learning coupled with neural networks to analyze and understand the emotions expressed in product, movie and restaurant reviews. Inspired by the structure of the human brain, neural networks process data through adjustable functions, allowing the assimilation of complex patterns and the generation of accurate outputs. With the development of the project, we seek the identification and analysis of relevant data sets, the use of Python libraries to process and train models, and the implementation of sentiment analyzers with various neural network models.

Introduction

Supervised learning, in conjunction with neural networks, is a powerful approach to model formulation aimed at generating predictions, performing pattern recognition and facilitating decision making based on contextual information from input data sets. In this process, neural networks, inspired by the structure of the human brain, play a crucial role by processing these inputs through tunable functions, optimizing their performance through learning. This pairing allows models to assimilate complex patterns and develop outputs that accurately reflect expected decisions or predictions. In this way, supervised learning in combination with neural networks emerges as a fundamental tool for solving tasks that require deep analysis and understanding of complex data.

For this reason, this project seeks the application, analysis and approach of the models seen in the course "Computing and Discrete Structures III" to study the input data and database outputs, with relevant information about feelings and emotions related to the perspective of users in reviews of products, movies, and restaurants.

Objectives

With the development of this project we seek to:

- Identify and analyze relevant dataset for neural networks and supervised learning approach.
- Apply and use Python libraries for data processing and training and testing of the models generated in neural networks.

- Implement sentiment analyzers with neural network models such as: DummyClassifier, vanilla RNN and LSTM.
- Apply the knowledge learned in the "Computing and Discrete Structures III" course on supervised learning and neural network models.

Process

In the development of the project, we found key stages, which allowed us to analyze the data and its application in the required neural network models. The stages included in the development were:

1. Analysis of the databases. We started with the review of the databases to be processed. The data to be processed are about interviews and reviews of different users who consume products, movies and restaurants, so, it is expected to find negative and positive keywords, and thus give decisions and predictions.
2. Data reading. Using the "pandas" library, the databases were read in "csv" format. Subsequently, the blanks were eliminated.
3. Tokenization. Each line of the database was read. Then, the keywords were separated in order to be able to work on them. Blank spaces were removed, strings were separated and English prepositions were eliminated, in order to leave only the key words (feelings or emotions) of the interviews.

4. Application of the models. With the refined data and their tokens, the training process begins with each of the neural network models proposed in the task. Several problems were encountered in this step. The versions of the libraries used had eliminated important elements to use, as it was seen from reference in tutorials on learning models and neural networks.

   The libraries and versions used were:

```
!pip install pandas==2.1.3
!pip install nltk==3.6.5
!pip install scikit-learn==1.3.2
!pip install scipy==1.11.4
!pip install threadpoolctl==3.2.0
!pip install tensorflow==2.8.0
```

With this, the models were applied in this way:

- We started with the DummyClasifier, the most basic supervised learning model. We applied training and testing of the model. The decision was made to use the stratified strategy.
- We continued with the use of the vanilla RNN. In this model we applied training, testing and search for the best hyperparameters with "GridSearchCV".
- Finally, we used the LSTMs in order to be able to compare the results of the three models. In this case, we also applied the training, testing and search of the hyperparameters with the same tool.

5. Comparison and evaluation. With the results generated by each of the models, the results were compared. For example: training time and statistical metrics.
6. Conclusions. With the comparisons made, conclusions were drawn with the considerations and objectives set out in the project.

Analysis

- **DummyClasifier:** It is a very simple and basic base classifier. It is used as a benchmark to compare the performance of more complex models. The "stratified" strategy was applied to build the model. Subsequently, model training is performed.

```
# Assume 'X' are the preprocessed features and 'y' are the labels
X = combined_df['Tokenized_Phrase']
y = combined_df['tag']

# Split the data into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

# Strategy: random classification, but maintaining the original class ratio of the training set
dummy_clf = DummyClassifier(strategy="stratified")

#Train the model
dummy_clf.fit(X_train, y_train)
```

Afterwards, the model is tested.

```
# Make predictions on the test set
y_pred_test = dummy_clf.predict(X_test)

# Calculate evaluation metrics on the test set
accuracy_test_dummy = accuracy_score(y_test, y_pred_test)
precision_test_dummy = precision_score(y_test, y_pred_test)
recall_test_dummy = recall_score(y_test, y_pred_test)
f1_test_dummy = f1_score(y_test, y_pred_test)
kappa_test_dummy = cohen_kappa_score(y_test, y_pred_test)
```

Implementation for the generation of the metrics generated by the test.

```
# Create a dictionary with the test metrics
metrics_test = {
    'Accuracy': accuracy_test_dummy,
    'Precision': precision_test_dummy,
    'Recall': recall_test_dummy,
    'F1 Score': f1_test_dummy,
    'Kappa': kappa_test_dummy

}

# Create a DataFrame from the test metrics dictionary
metrics_df_test = pd.DataFrame(list(metrics_test.items()), columns=['Metric', 'Value'])

# Print the DataFrame
print('\n' + 'Metrics for test set' + '\n')
display(metrics_df_test)
```

The results generated by the model are:

Metrics for test set

	Metric	Value
0	Accuracy	0.490000
1	Precision	0.491857
2	Recall	0.501661
3	F1 Score	0.496711
4	Kappa	-0.020079

Implementation of the metrics generated by the training:

```
##Calculate evaluation metrics (Train)
y_pred_train = dummy_clf.predict(X_train)

#Calculate evaluation metrics (Train)
accuracy_train_dummy = accuracy_score(y_train, y_pred_train)
precision_train_dummy = precision_score(y_train, y_pred_train)
recall_train_dummy = recall_score(y_train, y_pred_train)
f1_train_dummy = f1_score(y_train, y_pred_train)
kappa_train_dummy = cohen_kappa_score(y_train, y_pred_train)

metrics_train = {
    'Accuracy': accuracy_train_dummy,
    'Precision': precision_train_dummy,
    'Recall': recall_train_dummy,
    'F1 Score': f1_train_dummy,
    'Kappa': kappa_train_dummy
}

metrics_df_train = pd.DataFrame(list(metrics_train.items()), columns=['Metric', 'Value'])

# Show DataFrame
print('\n' + 'Metrics for train set' + '\n')
display(metrics_df_train)
```

Metrics for train set

	Metric	Value
0	Accuracy	0.488750
1	Precision	0.487993
2	Recall	0.474562
3	F1 Score	0.481184
4	Kappa	-0.022524

By studying this model, we can see that the accuracy and prediction are approximately 50%. This indicates that in the training and test set, this percentage of

correct predictions is obtained with the data set. By reviewing the Kappa, it can be said that the model tends more to chance than to prediction.

- **RNN:** It is a type of neural network model designed to process sequences of data. The structure of a Vanilla RNN is based on the repetition of network units at each time step, and these units have the ability to maintain an internal "memory". To start the implementation, a tokenization process was initiated.

```
# Maximum number of words to consider
max_words = 10000

# Create a Tokenizer with a specified maximum number of words
tokenizer = Tokenizer(num_words=max_words)

# Fit the Tokenizer on the training text data
tokenizer.fit_on_texts(X_train)

# Convert the training and test text data into sequences of indices
X_train_seq = tokenizer.texts_to_sequences(X_train)
X_test_seq = tokenizer.texts_to_sequences(X_test)

# Adjust the length of the sequences to a fixed size
maxlen = 100
X_train_pad = pad_sequences(X_train_seq, maxlen=maxlen)
X_test_pad = pad_sequences(X_test_seq, maxlen=maxlen)
```

With this, the implementation of the vanilla RNN is done.

```
##Define the architecture of the model:
def create_rnn_model(units=50):
    model = Sequential()
    model.add(Embedding(input_dim=max_words, output_dim=50, input_length=maxlen))
    model.add(SimpleRNN(units, activation='relu'))
    model.add(Dense(1, activation='sigmoid'))

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

Unlike the DummyClasifier model, in this one we apply a prior configuration of the hyperparameters, in order to find the best one.

Configure GridSearchCV to find the best hyperparameters

```
#Configure GridSearchCV to find the best hyperparameters
# Wrap the Keras model in a scikit-learn estimator
rnn_model = KerasClassifier(build_fn=create_rnn_model, epochs=5, batch_size=32, verbose=1)

# Configure GridSearchCV to find the best hyperparameters
param_grid = {'units': [50, 100, 150], 'batch_size': [32, 64, 128]}
grid_search = GridSearchCV(estimator=rnn_model, param_grid=param_grid, scoring='accuracy', cv=3)

# Fit the model
grid_result = grid_search.fit(X_train_pad, y_train)
```

The model was trained with the best hyperparameters:

Train the model with the best hyperparameters

```
best_units = grid_result.best_params_['units']
best_batch_size = grid_result.best_params_['batch_size']

best_rnn_model = create_rnn_model(units=best_units)
best_rnn_model.fit(X_train_pad, y_train, epochs=5, batch_size=best_batch_size, validation_split=0.2)
```

```
Epoch 1/5
60/60 [==============================] - 7s 98ms/step - loss: 0.6894 - accuracy: 0.5437 - val_loss: 0.6818 - val_accuracy: 0.5
875
Epoch 2/5
60/60 [==============================] - 5s 86ms/step - loss: 0.6443 - accuracy: 0.8000 - val_loss: 0.6481 - val_accuracy: 0.6
375
Epoch 3/5
60/60 [==============================] - 9s 147ms/step - loss: 0.4394 - accuracy: 0.8833 - val_loss: 0.5391 - val_accuracy: 0.
7625
Epoch 4/5
60/60 [==============================] - 6s 98ms/step - loss: 0.2040 - accuracy: 0.9568 - val_loss: 0.5434 - val_accuracy: 0.7
625
Epoch 5/5
60/60 [==============================] - 5s 89ms/step - loss: 0.1105 - accuracy: 0.9771 - val_loss: 0.5197 - val_accuracy: 0.7
854
<keras.callbacks.History at 0x7edac03ccbe0>
```

We see that, as the iterations progress, a better result is reflected in the "accuracy". We test the model.

Evaluate the performance of the model on the test set

```
y_pred_rnn_test = (best_rnn_model.predict(X_test_pad) > 0.5).astype("int32")

# Calcular métricas de evaluación
accuracy_test_rnn = accuracy_score(y_test, y_pred_rnn_test)
precision_test_rnn = precision_score(y_test, y_pred_rnn_test)
recall_test_rnn = recall_score(y_test, y_pred_rnn_test)
f1_test_rnn = f1_score(y_test, y_pred_rnn_test)
kappa_test_rnn = cohen_kappa_score(y_test, y_pred_rnn_test)

metrics_rnn_test = {
    'Accuracy': accuracy_test_rnn,
    'Precision': precision_test_rnn,
    'Recall': recall_test_rnn,
    'F1 Score': f1_test_rnn,
    'Kappa': kappa_test_rnn
}

metrics_rnn = pd.DataFrame(list(metrics_rnn_test.items()), columns=['Metric', 'Value'])

# Show DataFrame
print('\n' + 'Metrics for test set' + '\n')
display(metrics_rnn)
```

```
Metrics for test set
```

	Metric	Value
0	Accuracy	0.771667
1	Precision	0.775168
2	Recall	0.767442
3	F1 Score	0.771285
4	Kappa	0.543343

Subsequently, we do the training

Evaluate the performance of the model on the train set

```
y_pred_rnn = (best_rnn_model.predict(X_train_pad) > 0.5).astype("int32")

# Calcular métricas de evaluación
accuracy_rnn_train = accuracy_score(y_train, y_pred_rnn)
precision_rnn_train = precision_score(y_train, y_pred_rnn)
recall_rnn_train = recall_score(y_train, y_pred_rnn)
f1_rnn_train = f1_score(y_train, y_pred_rnn)
kappa_rnn_train = cohen_kappa_score(y_train, y_pred_rnn)

metrics_rnn_train = {
    'Accuracy': accuracy_rnn_train,
    'Precision': precision_rnn_train,
    'Recall': recall_rnn_train,
    'F1 Score': f1_rnn_train,
    'Kappa': kappa_rnn_train
}

metrics_rnn = pd.DataFrame(list(metrics_rnn_train.items()), columns=['Metric', 'Value'])

# Show DataFrame
print('\n' + 'Metrics for test set' + '\n')
display(metrics_rnn)
```

```
Metrics for test set
```

	Metric	Value
0	Accuracy	0.948750
1	Precision	0.950586
2	Recall	0.946622
3	F1 Score	0.948600
4	Kappa	0.897500

With these data, we can say that both the train and the test have concrete data and a good percentage in the prediction and correctness of the data given the input set. In the test we see that there is a 77% of accuracy, which already indicates a solidity in the accuracy of the model, this can be concretized with the Kappa, which indicates that the model tends more to prediction than to chance. On the other hand, in the train we see a higher result, in the accuracy there is 94%, which indicates a good level of learning. Finally, if we look at the kappa , we observe that there is an agreement beyond randomness.

- **LSTM:** It is a specialized type of recurrent neural network (RNN) designed to address the vanishing gradient problem and to capture long-term dependencies in sequential data. The implementation of the model was:
  Model construction.

```python
def create_lstm_model(units=50):
    model = Sequential()
    model.add(Embedding(input_dim=max_words, output_dim=50, input_length=maxlen))
    model.add(LSTM(units, activation='relu'))
    model.add(Dense(1, activation='sigmoid'))

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

When constructing the model, we search for the hyperparameters

Configuring GridSearchCV to find the best hyperparameters

```python
# Wrap the Keras model in a scikit-learn estimator
lstm_model = KerasClassifier(build_fn=create_lstm_model, epochs=5, batch_size=32, verbose=1)

# Configure GridSearchCV to find the best hyperparameters
param_grid = {'units': [50, 100, 150], 'batch_size': [32, 64, 128]}
grid_search = GridSearchCV(estimator=lstm_model, param_grid=param_grid, scoring='accuracy', cv=3)

# Fit the model
grid_result = grid_search.fit(X_train_pad, y_train)
```

With this, iterations were made in order to find the best hyperparameter according to the behavior.

At the end of the search, the training of the model with the hyperparameters was applied.

Training the model with the best hyperparameters

```python
best_units = grid_result.best_params_['units']
best_batch_size = grid_result.best_params_['batch_size']

best_lstm_model = create_lstm_model(units=best_units)
best_lstm_model.fit(X_train_pad, y_train, epochs=5, batch_size=best_batch_size, validation_split=0.2)
```

The configuration of the model helps us to apply the train and test.

Evaluate the performance of the model on the test set.

```python
y_pred_lstm = (best_lstm_model.predict(X_test_pad) > 0.5).astype("int32")

# Calculate metrics
accuracy_lstm_test = accuracy_score(y_test, y_pred_lstm)
precision_lstm_test = precision_score(y_test, y_pred_lstm)
recall_lstm_test = recall_score(y_test, y_pred_lstm)
f1_lstm_test = f1_score(y_test, y_pred_lstm)
kappa_lstm_test = cohen_kappa_score(y_test, y_pred_lstm)

metrics_lstm_test = {
    'Accuracy': accuracy_lstm_test,
    'Precision': precision_lstm_test,
    'Recall': recall_lstm_test,
    'F1 Score': f1_lstm_test,
    'Kappa': kappa_lstm_test
}

metrics_lstm = pd.DataFrame(list(metrics_lstm_test.items()), columns=['Metric', 'Value'])

# Show DataFrame
print('\n'+'Metrics for LSTM model on test set'+'\n')
display(metrics_lstm)
```

The metrics produced were:

Metrics for LSTM model on test set

	Metric	Value
0	Accuracy	0.803333
1	Precision	0.806020
2	Recall	0.800664
3	F1 Score	0.803333
4	Kappa	0.606671

By applying the train, we see that:

Evaluate the performance of the model on the train set.

```
# Evaluar el rendimiento del modelo en el conjunto de entrenamiento:
y_pred_lstm_train = (best_lstm_model.predict(X_train_pad) > 0.5).astype("int32")

# Calcular métricas de evaluación para el conjunto de entrenamiento
accuracy_lstm_train = accuracy_score(y_train, y_pred_lstm_train)
precision_lstm_train = precision_score(y_train, y_pred_lstm_train)
recall_lstm_train = recall_score(y_train, y_pred_lstm_train)
f1_lstm_train = f1_score(y_train, y_pred_lstm_train)
kappa_lstm_train = cohen_kappa_score(y_train, y_pred_lstm_train)

metrics_lstm_train = {
    'Accuracy': accuracy_lstm_train,
    'Precision': precision_lstm_train,
    'Recall': recall_lstm_train,
    'F1 Score': f1_lstm_train,
    'Kappa': kappa_lstm_train
}

metrics_lstm = pd.DataFrame(list(metrics_lstm_train.items()), columns=['Metric', 'Value'])

# Show DataFrame
print('\n'+'Metrics for LSTM model on training set'+'\n')
display(metrics_lstm)
```

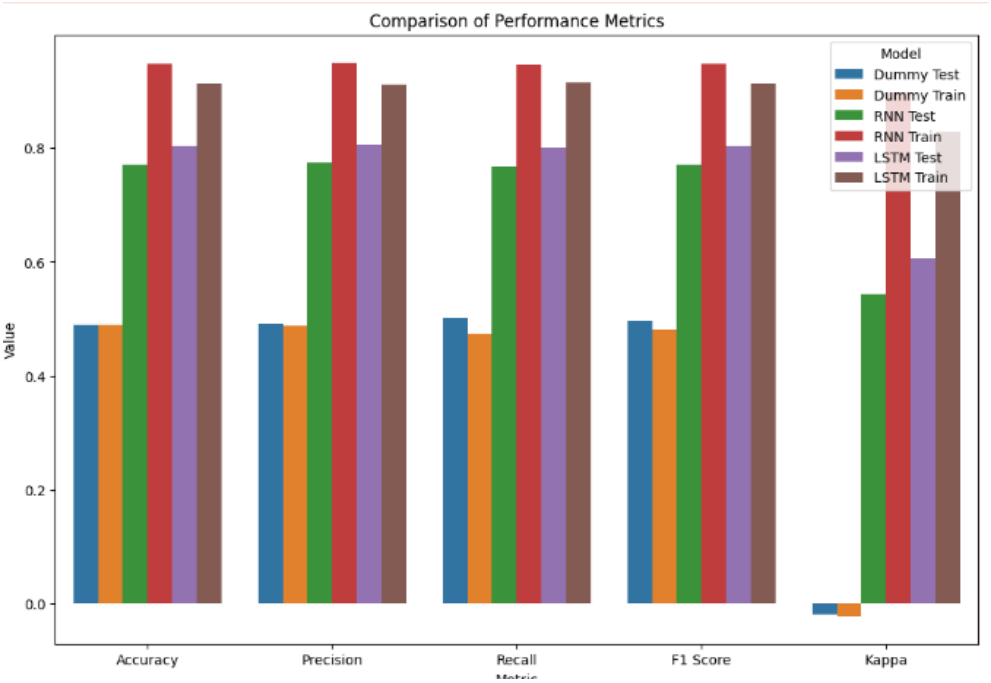
The metrics produced are:

Metrics for LSTM model on training set

	Metric	Value
0	Accuracy	0.914167
1	Precision	0.912718
2	Recall	0.915763
3	F1 Score	0.914238
4	Kappa	0.828334

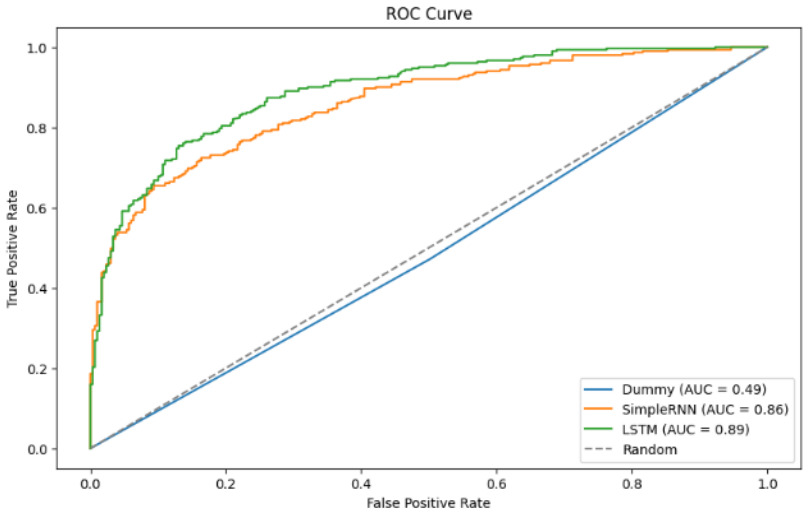
With this, we can see that both in the test and in the training the model has a 1 in its indicators, so the model has correctness and good predictions in the test model. In the training the high percentage in accuracy and kappa shows us that the model learns well, in high level given in input data set, besides having a good agreement.

Performance metrics



In this case we see that in the metrics the Dummy model set tends to have the lowest values indicating lower performance. While the RNN and LSTM model behave better in the train sets but move away from the values of the test set, this could be indicating an overfitting.

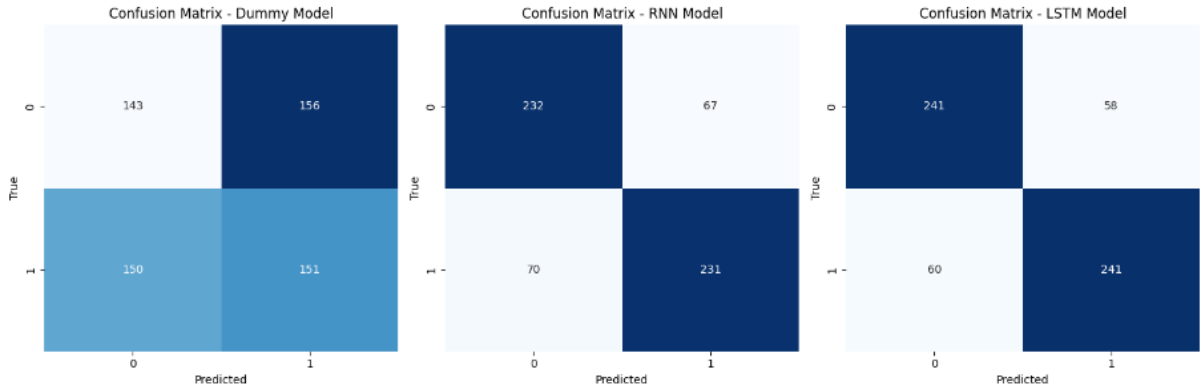
ROC Curve



The ROC curve shows how the true positive rate and false positive rate change simultaneously as you vary the decision threshold of the model. A curve approaching the upper left corner indicates better performance, as it signifies high true positive rates and low

false positive rates compared to other thresholds. In this case, we can say that the RNN model and the LSTM perform better than the Dummy. Being, the LSTM superior to the RNN.
An AUC close to 1 indicates a model with good performance, while an AUC close to 0.5 suggests that the model has difficulty discriminating between classes. Thus we see that the Dummy model has problems, it could be because of the chosen strategy. While RNN and LSTM are closer to 1.
The diagonal from (0,0) to (1,1) represents the performance of a random classifier. A model following this diagonal has no discriminative ability, hence Dummy with the chosen strategy does not discriminate between classes.

Confusion Matrix



This graph shows confusion matrices for each of your models (Dummy, RNN and LSTM). Each confusion matrix is divided into four quadrants, and each quadrant has a specific meaning:

- Upper Left Quadrant (True Negatives - TN): Shows the number of instances that actually belong to the negative class and were correctly classified as negative by the model.
- Upper Right Quadrant (False Positives - FP): Shows the number of instances that actually belong to the negative class, but were incorrectly classified as positive by the model.
- Lower Left Lower Quadrant (False Negatives - FN): Shows the number of instances that actually belong to the positive class, but were incorrectly classified as negative by the model.

- Lower Right Quadrant (True Positives - TP): Shows the number of instances that actually belong to the positive class and were correctly classified as positive by the model.

Each cell of the matrix contains the count of instances that fall into the corresponding category. The color of the cell indicates the magnitude of that count, where darker colors usually indicate higher values.

Interpreting confusion matrices:

- Dummy Model:The dummy model, being stratified, should have values relatively evenly distributed throughout the matrix. Therefore, similar numbers are seen in all four quadrants.
- RNN Model and LSTM Model:

In these models, higher numbers are seen in the main diagonal (TN and TP) and low numbers in the other quadrants, indicating that the model is correctly classifying both negative and positive instances.

Training time

```
import time

# Medir el tiempo de entrenamiento para el modelo Dummy
start_time = time.time()
dummy_clf.fit(X_train, y_train)
dummy_training_time = time.time() - start_time

# Medir el tiempo de entrenamiento para el modelo RNN
start_time = time.time()
best_rnn_model.fit(X_train_pad, y_train, epochs=5, batch_size=best_batch_size, validation_split=0.2)
rnn_training_time = time.time() - start_time

# Medir el tiempo de entrenamiento para el modelo LSTM
start_time = time.time()
best_lstm_model.fit(X_train_pad, y_train, epochs=5, batch_size=best_batch_size, validation_split=0.2)
lstm_training_time = time.time() - start_time

# Mostrar los tiempos de entrenamiento
print(f"Training Time - Dummy Model: {dummy_training_time:.2f} seconds")
print(f"Training Time - RNN Model: {rnn_training_time:.2f} seconds")
print(f"Training Time - LSTM Model: {lstm_training_time:.2f} seconds")
```

```
Training Time - Dummy Model: 0.00 seconds
Training Time - RNN Model: 41.00 seconds
Training Time - LSTM Model: 44.44 seconds
```

- Dummy Model Training Time:

This time represents how long it takes to train the Dummy model, which is a very simple model that uses the "most frequent" strategy.This time is low because the Dummy model has no parameters to fit and does not perform any meaningful calculations during training.

- RNN Model Training Time:

This time indicates how long it takes to train the RNN model, which is a more complex model with a recurrent neural network layer.The time pes higher compared to the Dummy model because the RNN has more parameters and performs more intensive computations during training.

- Training Time of the LSTM Model:

This time represents the time required to train the LSTM model, which is a type of recurrent neural network with long-term memory cells. The training time is higher than that of the RNN model due to the higher complexity of the LSTM architecture.

General Interpretation:

Training time is an important factor to consider, especially in large data sets or in cases where many computational resources are required. More complex and deeper models, such as RNN and LSTM models, tend to require more training time.

Conclusion

Based on the metrics and visualizations provided, we can draw the following conclusions:

1. **Overall Performance:**
    - The Dummy model, as expected, underperforms on all metrics compared to the more complex models, as it simply follows a simple strategy.
    - Both RNN and LSTM models outperform the Dummy model on all metrics in the test and training sets.
2. **Overfitting:**
    - Both RNN and LSTM models show slight overfitting in the training set, as their metrics are better in the training set compared to the test set. This could indicate that the models are capturing too much of the specific details of the training set and do not generalize as well on new data.
3. **Comparison between RNN and LSTM:**

- In general, the LSTM model tends to perform better compared to the RNN model in terms of metrics on both sets (training and test).
- However, the training time of the LSTM model is slightly longer than that of the RNN model.

4. **Training Time:**
    - The Dummy model has the lowest training time as it is a simple model with no significant adjustable parameters.- The RNN model has a lower training time compared to the LSTM model.
5. **Considerations:**
    - The choice of the best model will depend on the specific needs of the problem and the available resources.- If training time is prioritized and slightly lower performance is acceptable, the RNN model could be an option.
    - If better performance is sought at the expense of slightly longer training time, the LSTM model might be preferable.

In summary, the LSTM model could be considered the best in this context, as it achieves a better balance between performance and training time compared to the RNN model.However, this choice depends on project-specific priorities, such as the importance of performance versus training time.

Bibliography and resources

The resources used in the development of the project were:

- Sentimental labelled sentences
- NLTK doc
- Pandas documentation
- Report structure
- RNN IBM