Integrative Task II

Andrés Arango

Jhonatan Castaño

Icesi University

Computation and Discrete Structures III

Cali, Colombia

November 24th 2023

Introduction

As in all integrative tasks, the idea is to apply the concepts seen in class, in this one the concepts based on artificial intelligence and recurrent neural networks will be applied.

Aims

Main

1. Build a sentiment analysis model using supervised learning with vanilla Recurrent Neural Networks and LSTM

Secondaries

1. Create a database with sentences and the type of sentiment of itself.
2. Tokenize the sentences to find a way to build a supervised learning model.
3. Implement a DummyClassifier for the model.
4. Implement a vanilla RNN sentiment analysis model.
5. Implement a LSTM sentiment analysis model.

Theoretical framework

Machine Learning

Machine learning is a subfield of artificial intelligence that focuses on the development of algorithms and models that enable computer systems to learn from data and improve their performance over time without being explicitly programmed. The core idea behind machine learning is to enable computers to automatically identify patterns, make predictions, or optimize decisions based on experience or historical data. This is achieved through the utilization of mathematical and statistical techniques that allow algorithms to generalize from examples and adapt to new information. Machine learning finds applications in various domains, including image and speech recognition, natural language processing, recommendation systems, and autonomous vehicles, among others. The overarching goal is to empower machines to acquire knowledge and skills autonomously, enhancing their ability to solve complex tasks and contribute to intelligent decision-making processes.

Supervised Learning

Supervised learning is a type of machine learning paradigm where the algorithm is trained on a labeled dataset, meaning that the input data is paired with corresponding output labels. The goal of supervised learning is for the algorithm to learn a mapping from input data to the correct output by generalizing patterns from the labeled examples provided during training.

Pandas

Pandas is a popular open-source data manipulation and analysis library for Python. It provides data structures for efficiently storing and manipulating large datasets and tools for working with structured data seamlessly.

Pandas simplifies common data tasks, such as:

- Loading data from various file formats (CSV, Excel, SQL databases, etc.).
- Cleaning and preprocessing data by handling missing values, removing duplicates, and transforming data.
- Indexing, slicing, and sub setting data for easy extraction of relevant information.
- Performing statistical and mathematical operations on the data.
- Merging and joining datasets.
- Time-series analysis and handling.

Pandas is widely used in data science, machine learning, and other domains where data manipulation and analysis are essential. Its intuitive and powerful functionality makes it a go-to library for working with structured data in Python.

Sklearn

Scikit-learn, often abbreviated as sklearn, is a popular open-source machine learning library for Python. It provides a simple and efficient set of tools for data analysis and modeling, with a focus on classical machine learning algorithms. Scikit-learn is built on NumPy, SciPy, and Matplotlib, making it compatible with other scientific computing libraries in the Python ecosystem.

NLTK

NLTK, or Natural Language Toolkit, is a powerful library for working with human language data (text) in Python. It provides easy-to-use interfaces to perform various natural language processing (NLP) tasks. NLTK is widely used in research and education and serves as a valuable resource for developers and researchers working on projects involving textual data.

Key features and functionalities of NLTK include:

- Text Processing: NLTK offers tools for tokenization, stemming, lemmatization, and other text processing tasks. These tools allow you to break down text into meaningful components and standardize word forms.
- Part-of-Speech Tagging: NLTK includes modules for part-of-speech tagging, which involves labeling words in a text with their respective parts of speech (e.g., noun, verb, adjective).
- Named Entity Recognition: NLTK provides functions for identifying and classifying named entities (e.g., names of people, organizations, locations) in text.
- Sentiment Analysis: NLTK supports sentiment analysis, allowing you to determine the sentiment (positive, negative, neutral) expressed in a piece of text.
- Corpus and Resources: NLTK includes a variety of corpora and lexical resources for different languages. These resources are useful for training and testing NLP models.

- Machine Learning with Text Data: NLTK integrates with machine learning libraries and provides functionalities for feature extraction and classification of text data.

Dummy Classifier

Dummy Classifier is a simple baseline classifier that is often used for comparison purposes. It serves as a basic reference point for evaluating the performance of more sophisticated machine learning models. The Dummy Classifier makes predictions using simple rules and is particularly useful for assessing whether a more complex model provides significant improvements over a naive or random strategy.

RNN Vanilla

A vanilla Recurrent Neural Network (RNN) is a type of neural network architecture designed to work with sequential data, such as time series or natural language. Unlike traditional feedforward neural networks, which process input data in isolation, RNNs have connections that form a directed cycle, allowing them to maintain a memory of previous inputs in their internal state.

In a vanilla RNN, the key idea is to capture dependencies and patterns in sequential data by maintaining a hidden state that evolves as the network processes each element in the sequence. The hidden state serves as a form of memory, allowing the network to consider context from previous time steps when making predictions or generating output.

However, vanilla RNNs have limitations. They struggle with capturing long-range dependencies in sequences, a problem known as the "vanishing gradient" problem. As the network processes input over time, gradients can diminish, making it challenging to learn from earlier parts of the sequence.

LSTM

A Long Short-Term Memory (LSTM) is a type of recurrent neural network (RNN) architecture designed to address the vanishing gradient problem and capture long-range dependencies in sequential data. LSTMs are particularly effective in tasks involving time series prediction, natural language processing, and other applications with sequential patterns.

The key innovation of LSTMs lies in their ability to maintain a cell state, which acts as a memory unit that can selectively remember or forget information over long sequences. This enables LSTMs to capture and retain relevant information from earlier time steps, making them well-suited for tasks that involve understanding context over extended periods.

The architecture of an LSTM cell includes three interacting gates:

1. Forget Gate: Determines what information from the cell state should be discarded or kept. It regulates the flow of information from the previous cell state.

2. Input Gate: Decides what new information should be stored in the cell state. It involves updating the cell state with new candidate values.
3. Output Gate: Controls what information from the cell state should be used to generate the output. It influences the prediction or the hidden state of the LSTM.

The gating mechanisms in LSTMs enable them to mitigate the vanishing gradient problem by selectively updating and passing information through the network. This makes LSTMs more effective at capturing dependencies in sequential data over longer distances than traditional RNNs.

In summary, LSTMs are a type of RNN designed to capture long-term dependencies in sequential data through the use of memory cells and gating mechanisms, making them well-suited for a wide range of applications involving sequences and temporal patterns.

GridSearchCV

GridSearchCV is a function in the scikit-learn library for Python that performs an exhaustive search over a specified parameter grid for a machine learning algorithm. It is used for hyperparameter tuning, where the goal is to find the best combination of hyperparameter values that maximizes the performance of a model.

How GridSearchCV works:

- Parameter Grid: You define a grid of hyperparameter values that you want to explore. Each point in this grid represents a combination of hyperparameters that you want to test.
- Cross-Validation: GridSearchCV uses cross-validation to evaluate the performance of each combination of hyperparameters. It splits the training data into multiple subsets (folds), trains the model on some folds, and evaluates it on the remaining folds. This process is repeated for each combination of hyperparameters.
- Model Training and Evaluation: For each combination of hyperparameters, the model is trained and evaluated using cross-validation. The performance metric (such as accuracy, precision, or others) is computed for each combination.
- Best Hyperparameters: After the exhaustive search, GridSearchCV identifies the combination of hyperparameters that yielded the best performance according to the specified metric.

Methodology

The methodology for this project is the same of a machine learning project, this one follows these steps:

1. Define the Problem:
    - Clearly define the problem you want to solve. Understand the goals and objectives of the project.
    - Define the scope and constraints of the problem.

2. Gather Data:

    - Collect relevant data for your problem. This may involve acquiring datasets from various sources or generating your own data.
    - Ensure that the data is representative of the problem and is of sufficient quality.

3. Explore and Prepare the Data:

    - Perform exploratory data analysis (EDA) to understand the characteristics of the data.
    - Handle missing values, outliers, and other data preprocessing tasks.
    - Encode categorical variables, scale numerical features, and perform other necessary transformations.

4. Feature Engineering:

    - Create new features that might improve the performance of your model.
    - Select relevant features and discard irrelevant ones.

5. Split the Data:

    - Split the dataset into training and testing sets. The training set is used to train the model, while the testing set is used to evaluate its performance.

6. Choose a Model:

    - Select a machine learning algorithm that is suitable for your problem. Consider factors like the type of problem (classification, regression, etc.) and the characteristics of your data.

7. Train the Model:

    - Train the chosen model using the training dataset.
    - Fine-tune hyperparameters to optimize model performance. Use techniques like cross-validation.

8. Evaluate the Model:

    - Assess the model's performance on the testing set using appropriate evaluation metrics (accuracy, precision, recall, F1 score, etc.).

- Analyze the results and iterate on the model or data preprocessing if necessary.

9. Hyperparameter Tuning:

   - Conduct further hyperparameter tuning to improve model performance. This may involve using techniques like grid search or random search.

10. Document the Project:

    - Document all aspects of the project, including the problem definition, data sources, preprocessing steps, model selection, training process, and evaluation results. This documentation is valuable for future reference and collaboration.

Work development

Define the problem and gather data.

The problem to solve is to find a model to find a way on how to know some sentence can predict sentiments.

To begin the development of this work, we started with the search of the database on which the models to be worked on would be implemented, the selected one has the name of "Sentiment Labelled Sentences Dataset", which is part of the UC Irvine Machine Learning repository. This in turn was created from the paper "From Group to Individual Labels using Deep Features".

Link to the dataset is: Sentiment Labelled Sentences Data Set

The database has the following characteristics:

It contains sentences from three different websites:

- Amazon.com
- Imbd.com
- Yelp.com

Each sentence is rated 1 for positive, 0 for negative.

Each of the sites has the same number of positive and negative sentences.

Having already implemented the .txt files in a pandas dataframe to use them in models, we found the following information:

For the Amazon.com database we found the number of 1000 sentences each has a rating both to be positive or negative.

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 1000 entries, 0 to 999
Data columns (total 2 columns):
 #   Column    Non-Null Count  Dtype
---  ------    --------------  -----
 0   sentence  1000 non-null   object
 1   target    1000 non-null   int64
dtypes: int64(1), object(1)
memory usage: 15.8+ KB
```

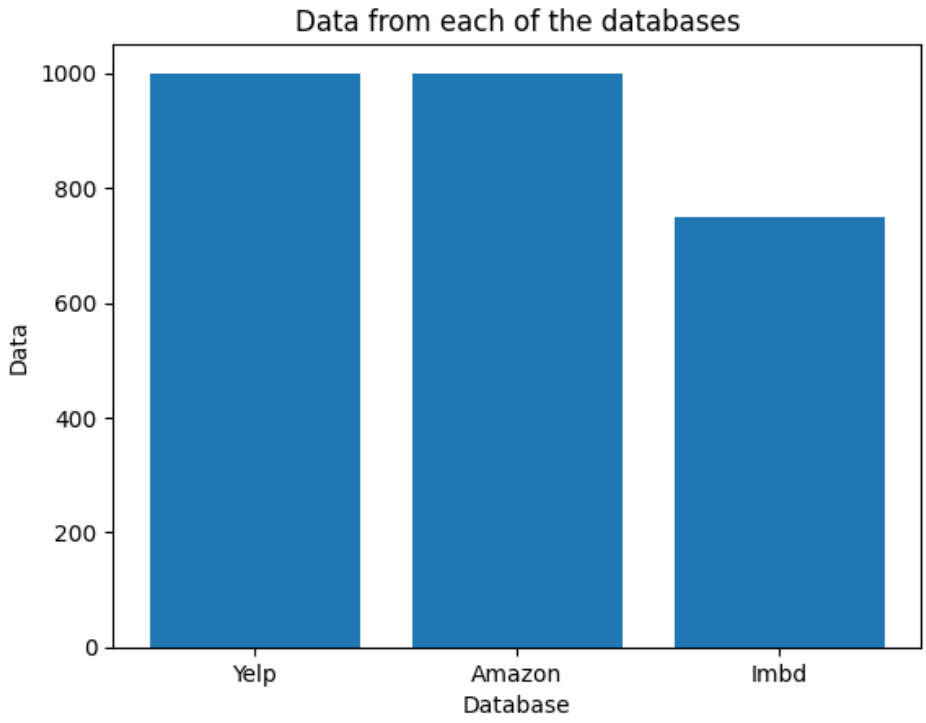
For the Imbd.com database we found the amount of 748 sentences each with a rating either positive or negative.

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 748 entries, 0 to 747
Data columns (total 2 columns):
 #   Column    Non-Null Count  Dtype
---  ------    --------------  -----
 0   sentence  748 non-null    object
 1   target    748 non-null    int64
dtypes: int64(1), object(1)
memory usage: 11.8+ KB
```

For the Yelp.com database we found 1000 phrases each with a rating of either positive or negative.

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 1000 entries, 0 to 999
Data columns (total 2 columns):
 #   Column    Non-Null Count  Dtype
---  ------    --------------  -----
 0   sentence  1000 non-null   object
 1   target    1000 non-null   int64
dtypes: int64(1), object(1)
memory usage: 15.8+ KB
```

The data can be viewed as follows:



Having joined all these dataframes using the pandas append function, we create the following database.

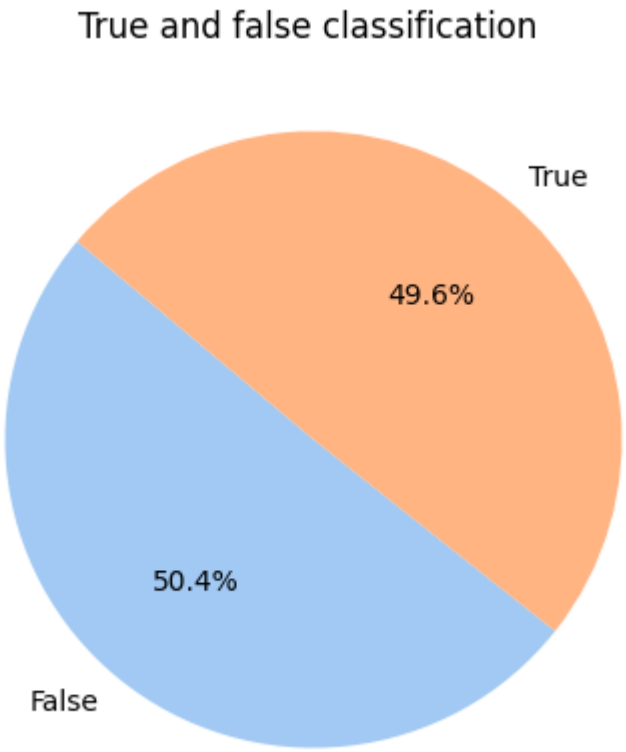
```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 2748 entries, 0 to 2747
Data columns (total 2 columns):
 #   Column    Non-Null Count  Dtype
---  ------    --------------  -----
 0   sentence  2748 non-null   object
 1   target    2748 non-null   int64
dtypes: int64(1), object(1)
memory usage: 43.1+ KB
```

This database presents a total of 2748 data which are divided into two columns, one called sentence where the sentences are found and the other called target where the classification of whether it is a positive (1) or negative (0) sentiment is found.

Explore the data.

Analyzing the database, it was observed that it has a column with object type data, to analyze this data a tokenization process of the data has to be done, this will be done in the features engineering step that comes later.

On the other hand, the other column of numeric type presents a binary classification from which the following graph can be obtained.



50,4% of the sentences are related with bad emotions and 49,6% are related with good emotions.

The common statistics of the function describing pandas were also extracted, but since they were from a classification column, they were not very helpful or it would not be necessary to adjust them for the models.

	target
count	2748.000000
mean	0.504367
std	0.500072
min	0.000000
25%	0.000000
50%	1.000000
75%	1.000000
max	1.000000

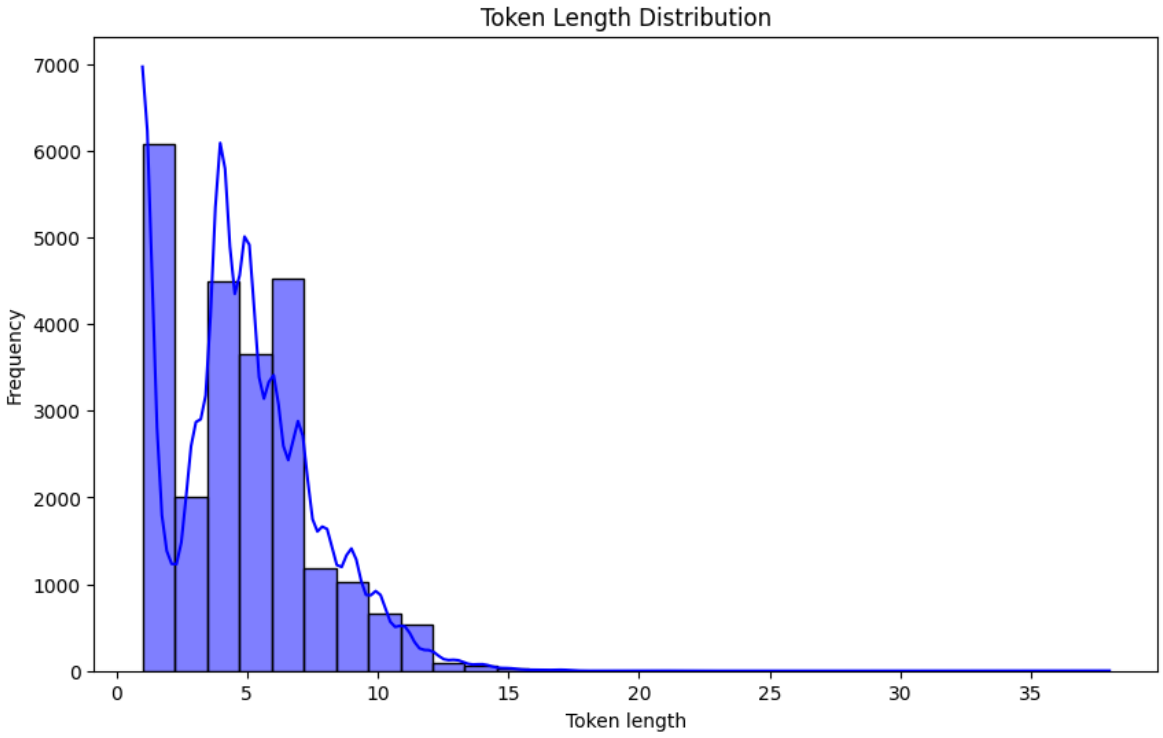
Feature Engineering

For this part of the project, the sentences were tokenized by creating a function called preprocess_text, which was in charge of making a token in each of the sentences, convert all of them to lowercase, eliminate the spaces and give lemmas to the words of the sentences and join them together.

This function was applied to each of the sentences in the database, resulting in more simplified sentences, here is a glimpse of the first data in the database.

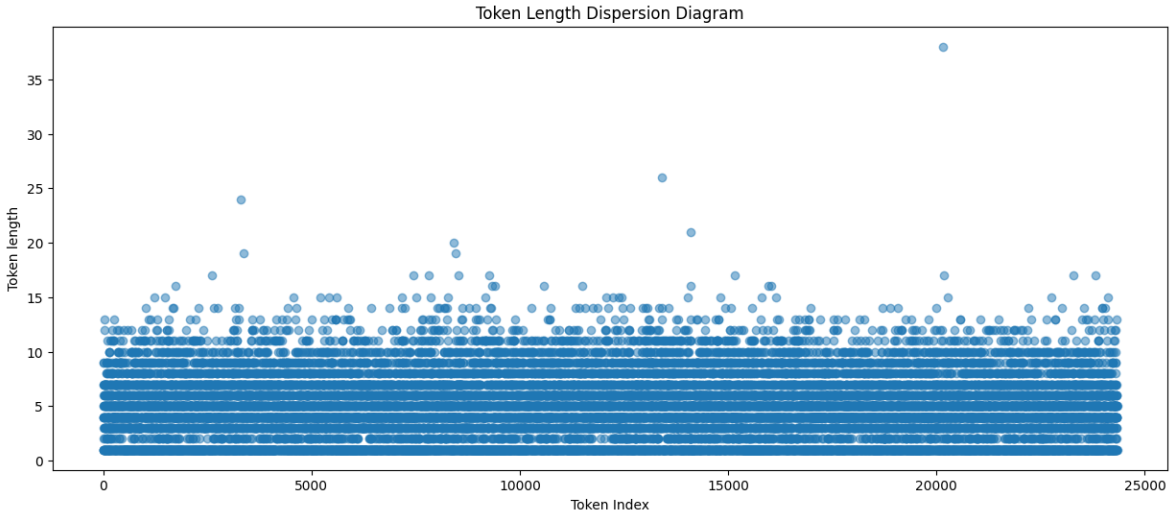
	sentence	target
0	way plug u unless go converter .	0
1	good case , excellent value .	1
2	great jawbone .	1
3	tied charger conversation lasting 45 minutes.m...	0
4	mic great .	1

The following graphs can be drawn from the tokenized words:



The graph shows the token length distribution, i.e., the number of times each token length is observed. The X-axis represents the token length in bytes and the Y-axis represents the frequency, i.e. the number of tokens having that length.

We can see that the token length is concentrated around 10 bytes. There are a large number of tokens that have a length of 10 bytes, and then the frequency decreases rapidly as the token length increases. There are very few tokens that are longer than 30 bytes.



The graph you sent me shows the dispersion of token length in a data set. The X-axis represents the token index, and the Y-axis represents the token length in bytes.

We can see that the length of tokens is relatively evenly distributed. There are a small number of tokens with a length of 1 or 2 bytes, but most tokens are 4 to 10 bytes long. There are a small number of tokens with a length of more than 10 bytes, but these are relatively rare.

This distribution suggests that the tokens in this data set represent a variety of data types. Small-sized data, such as integers or strings, are well represented in this distribution. Larger data, such as images or audio files, are relatively rare.

In this image you can see which are the most common words that could be extracted from the tokenization of the data, the more visible the word is, the more frequent it is.

Split the Data

The division of the data was done using the function train_test_split, where X was selected as the sentences, and Y as the feelings, these were separated into sets of training and test, that is an array X for training and test and an array Y for training and test, the proportion of this division was 80% for training and 20% for tests.

```
X = df['sentence']
y = df['target']


label_encoder = LabelEncoder()
y = label_encoder.fit_transform(y)

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42, train_size=0.8, shuffle=True)
✓ 0.0s
```

Before proceeding further, both the test and training sets of sentences had to be converted from words to numbers using the Keras tokenizer function, which serves to convert the texts into sequences of numbers, determines the maximum length of those sequences and performs padding to ensure that all sequences have the same length before feeding them to a machine learning model.

From this transformation emerged the train_data and test_data variables

```
tokenizer = Tokenizer()
tokenizer.fit_on_texts(X_train)
train_sequences = tokenizer.texts_to_sequences(X_train)
test_sequences = tokenizer.texts_to_sequences(X_test)
max_len = max(max(len(seq) for seq in train_sequences), max(len(seq) for seq in test_sequences))
train_data = pad_sequences(train_sequences, maxlen=max_len)
test_data = pad_sequences(test_sequences, maxlen=max_len)
✓ 0.0s
```

Choose a Model

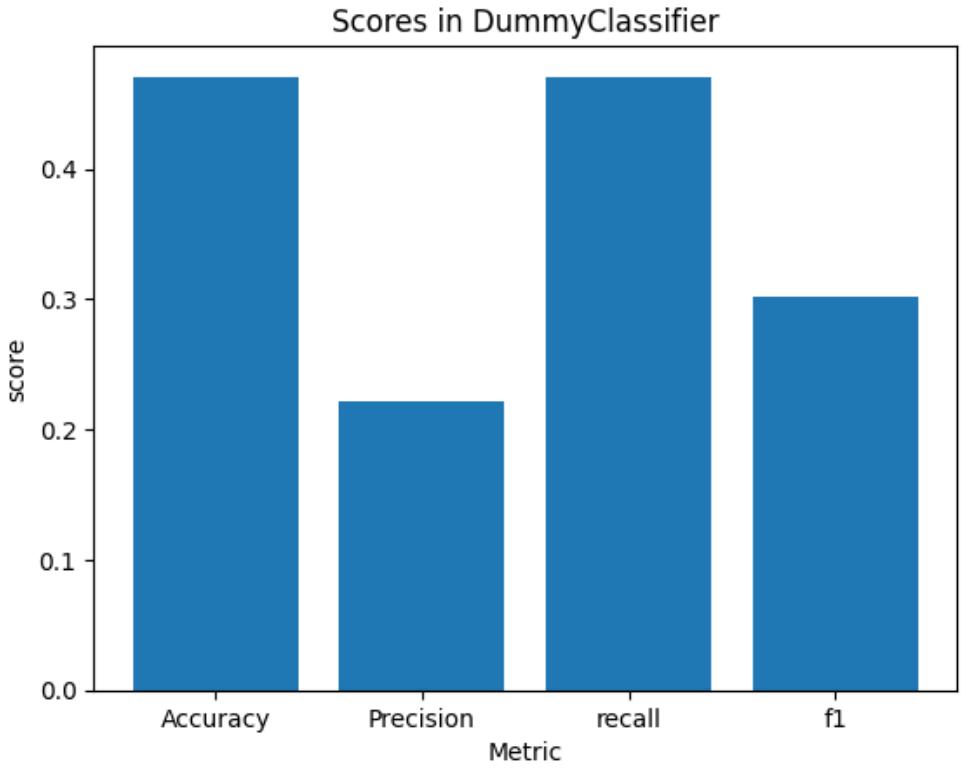
DummyClassifier, RNN and LSMT models were selected for this project.

All of these were presented in the theorical framework.

Train the model and evaluate the model.

DummyClassifier

A dummyClassifier was started with default values, trained with the data set designed for this purpose and these were the results.



RNN

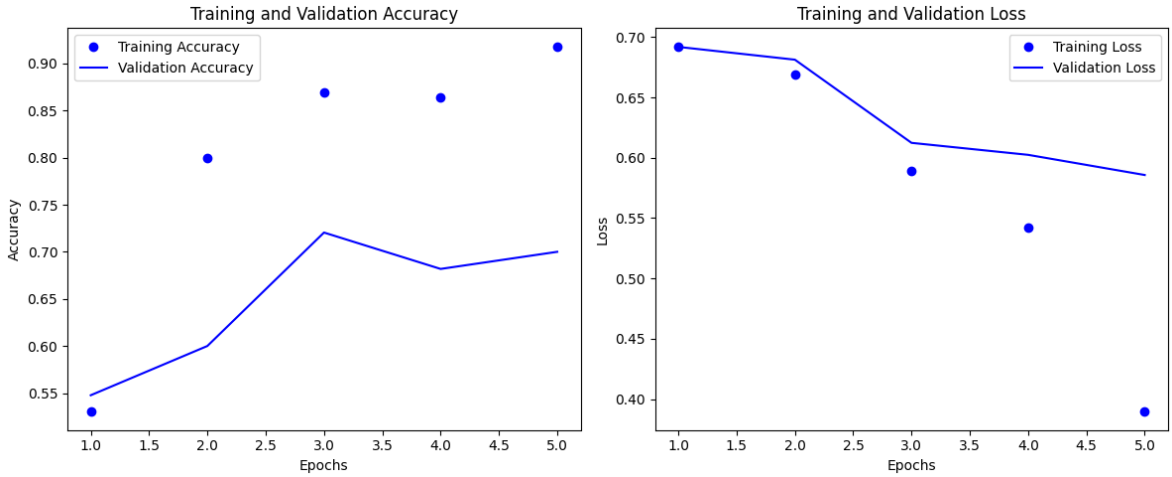
For RNN, a function was created to create the network, which was assigned to a variable on which the first evaluation was made using 5 epochs and the training data.

```
[36]  rnn_model.summary()
      ✓ 0.0s

...   Model: "sequential"
      _________________________________________________________________
       Layer (type)                Output Shape              Param #
      =================================================================
       embedding (Embedding)       (None, 816, 32)           133440

       simple_rnn (SimpleRNN)      (None, 32)                2080

       dense (Dense)               (None, 1)                 33

      =================================================================
      Total params: 135,553
      Trainable params: 135,553
      Non-trainable params: 0
      _________________________________________________________________
```

The model is sequential with an Embedding layer, a SimpleRNN layer and a Dense layer. Detailed information about the number of parameters in each layer is provided to

understand the complexity of the model and the amount of tuning you can do during training.

The accuracy and loss results for each of the epochs of this initial model were as follows.

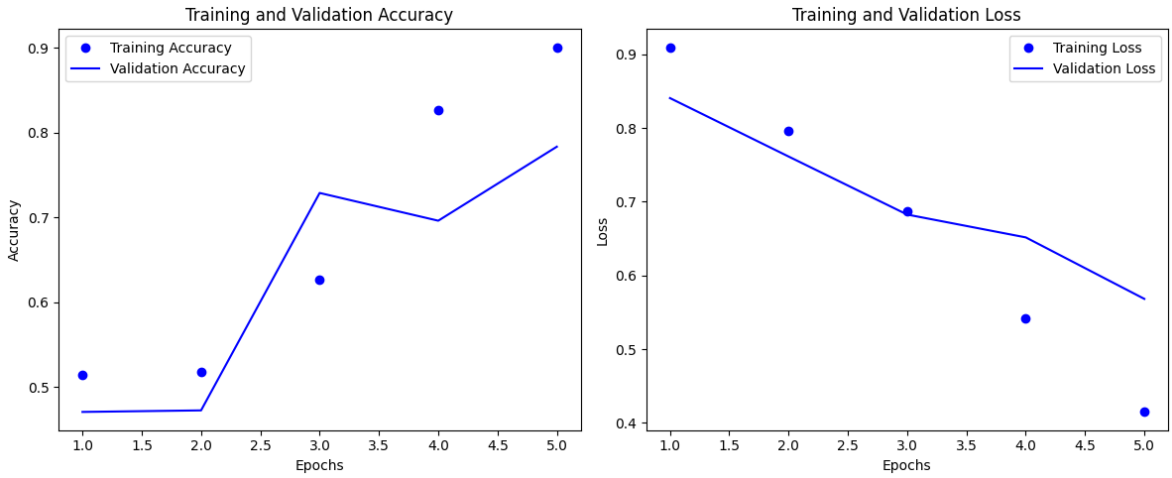


LSTM

For this model a function was created which is responsible for creating the model, this is assigned to a variable, on which a test case was made with 5 epochs and the training data and test as the evaluative, these were the results.

```
Model: "sequential_9"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 embedding_9 (Embedding)     (None, 816, 16)           160000

 lstm (LSTM)                 (None, 816, 50)           13400

 dropout (Dropout)           (None, 816, 50)           0

 lstm_1 (LSTM)               (None, 50)                20200

 dropout_1 (Dropout)         (None, 50)                0

 dense_9 (Dense)             (None, 8)                 408

 dense_10 (Dense)            (None, 1)                 9

=================================================================
Total params: 194,017
Trainable params: 194,017
Non-trainable params: 0
_________________________________________________________________
```

This architecture shows a neural network model with Embedding layers, two LSTM layers, Dropout layers to avoid overfitting and Dense layers for final classification. The total number of parameters gives an idea of the complexity of the model.



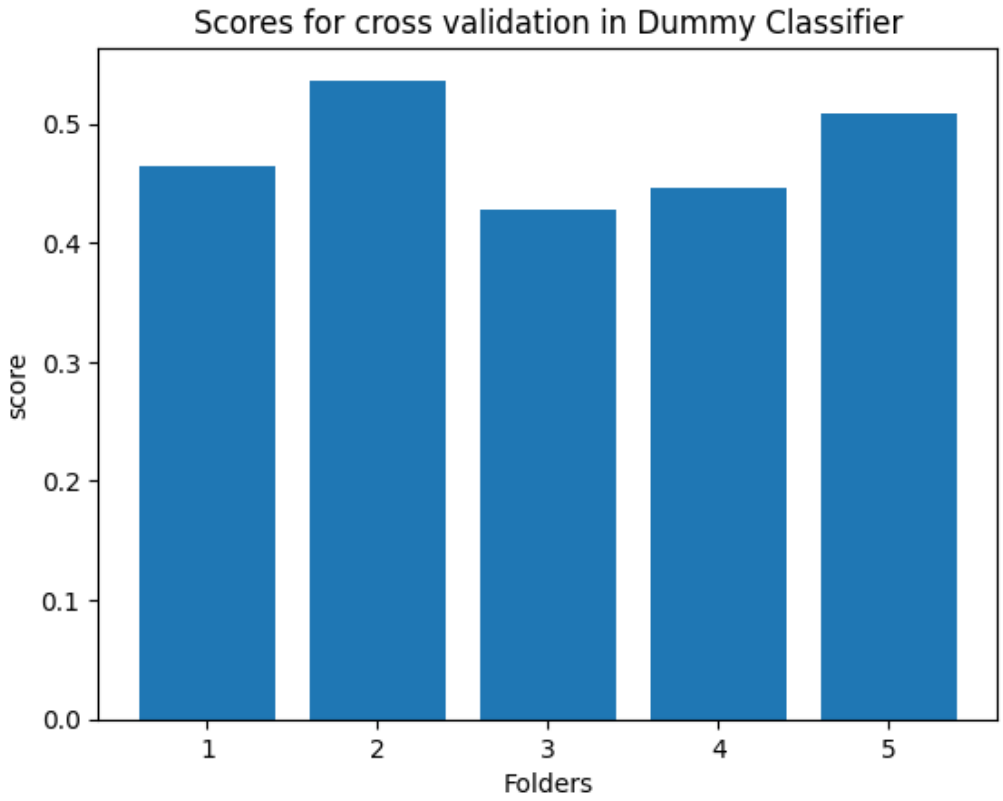
Hyperparameter Tunning

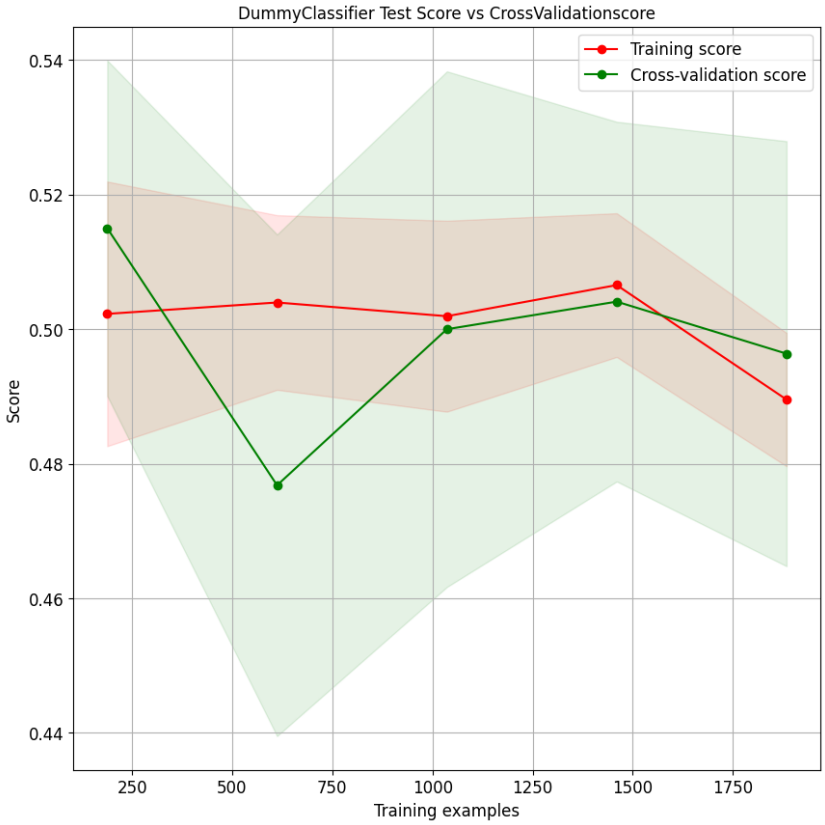
For hyperparameter tuning, the GridSearchCV function was used, in which a grid of parameters and results was placed, as well as the training data on which a cross validation was performed.

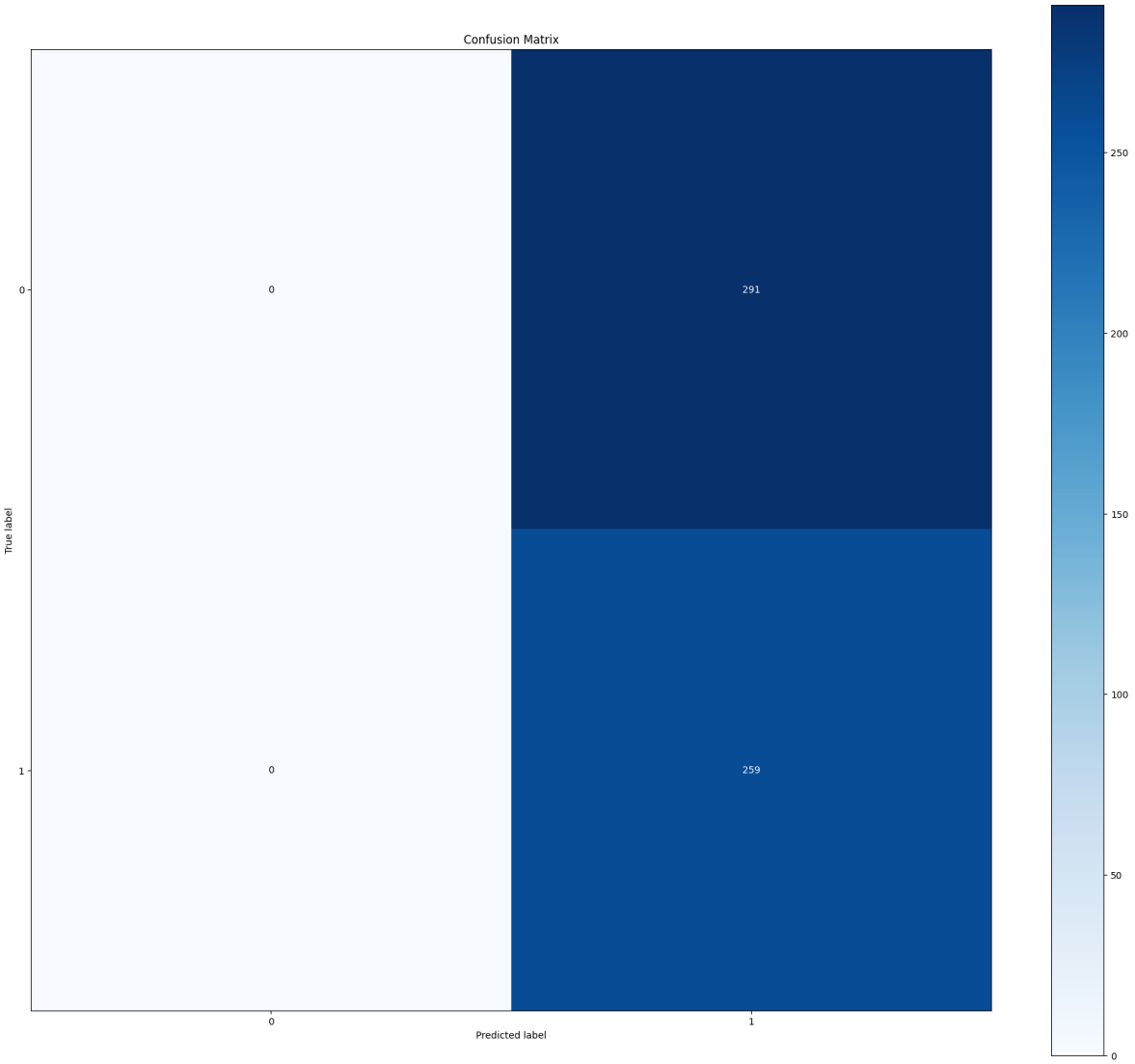
DummyClassifier

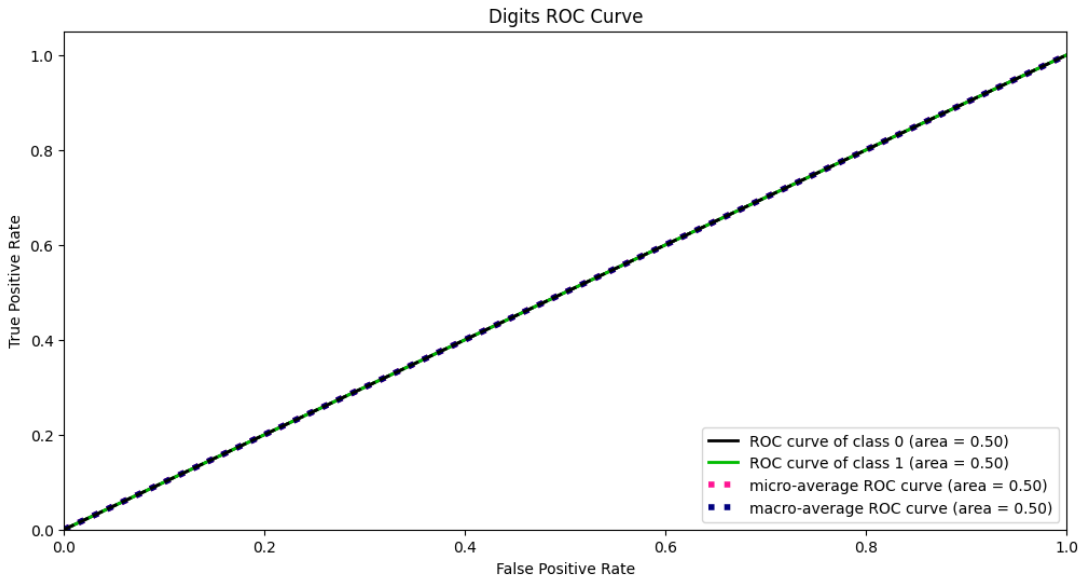
```
param_grid = {
    'strategy': ['uniform', 'most_frequent', 'stratified', 'constant'],
    'constant': [0, 1, 2]
}

scoring_metrics = {
    'accuracy': make_scorer(accuracy_score),
    'precision': make_scorer(precision_score, average='weighted'),
    'recall': make_scorer(recall_score, average='weighted'),
    'f1': make_scorer(f1_score, average='weighted')
}

grid_search = GridSearchCV(dummy_clf, param_grid, cv=5, scoring=scoring_metrics, refit="accuracy")
```

```
Mejores hiperparámetros: {'constant': 0, 'strategy': 'most_frequent'}
```


Scores for cross validation in Dummy Classifier
score
0.0
0.1
0.2
0.3
0.4
0.5
1
2
3
4
5
Folders


DummyClassifier Test Score vs CrossValidationscore
Training score
Cross-validation score
Score
0.44
0.46
0.48
0.50
0.52
0.54
250
500
750
1000
1250
1500
1750
Training examples

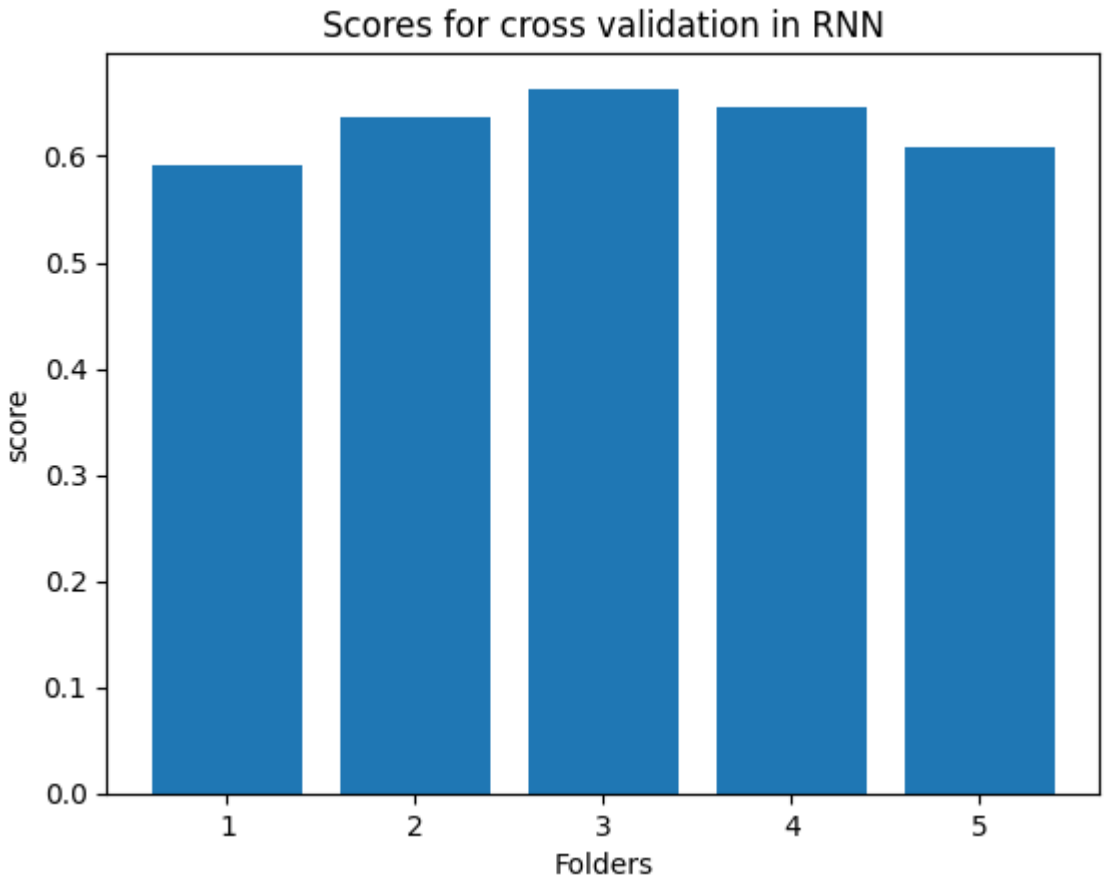

Confusion Matrix
0
291
0
259
True label
Predicted label
0
1
0
1
250
200
150
100
50
0

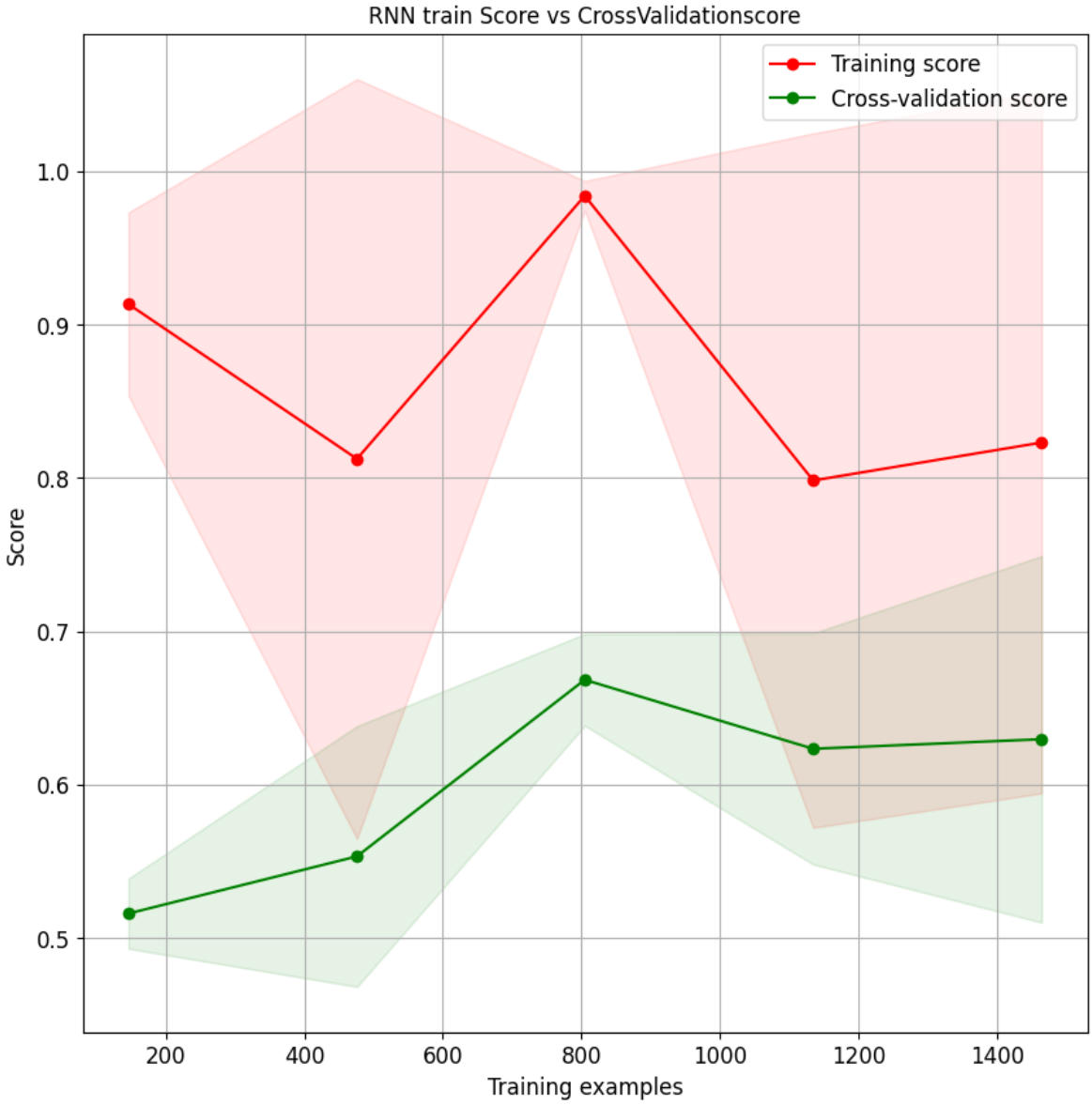

Digits ROC Curve
True Positive Rate
False Positive Rate
0.0
0.2
0.4
0.6
0.8
1.0
ROC curve of class 0 (area = 0.50)
ROC curve of class 1 (area = 0.50)
micro-average ROC curve (area = 0.50)
macro-average ROC curve (area = 0.50)

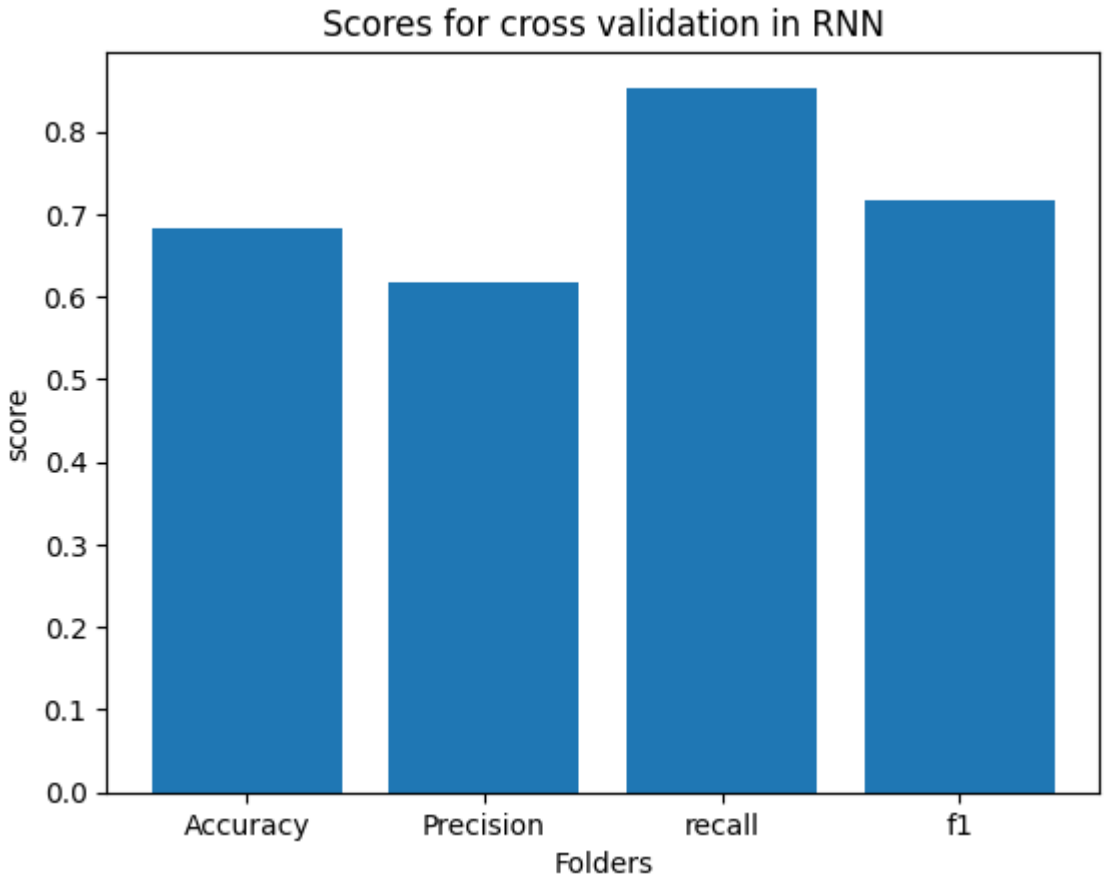
RNN

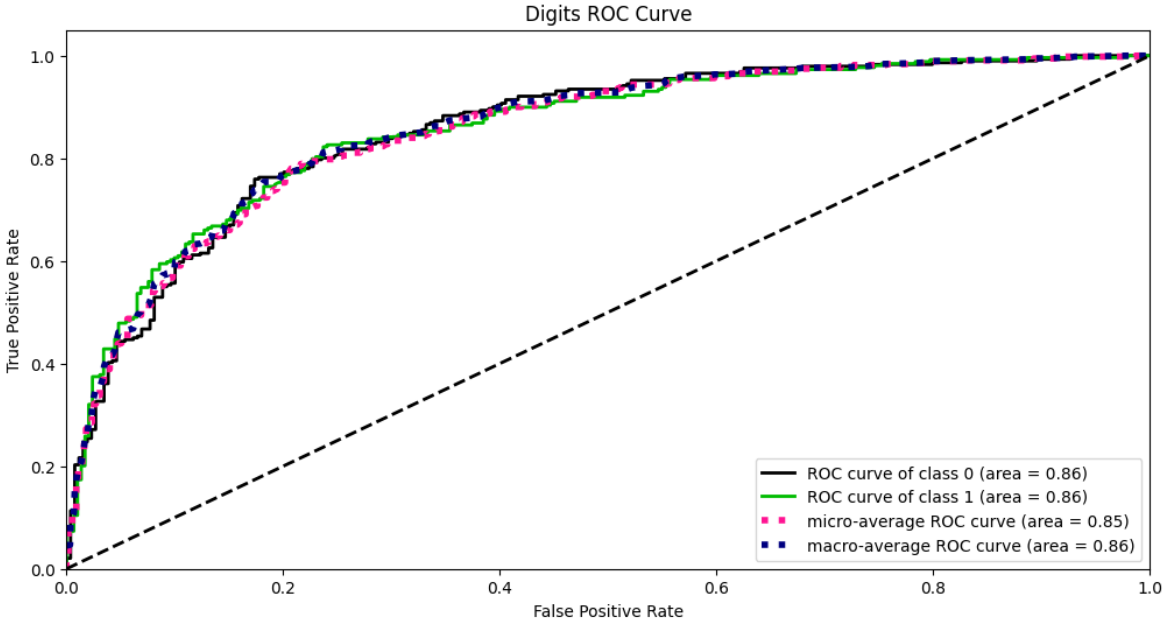
```
param_grid = {
    'units': [50, 100],
    'epochs': [5, 10],
    'batch_size': [32, 64]
}

scoring_metrics = {
    'accuracy': make_scorer(accuracy_score),
    'precision': make_scorer(precision_score, average='weighted'),
    'recall': make_scorer(recall_score, average='weighted'),
    'f1': make_scorer(f1_score, average='weighted')
}

grid_search = GridSearchCV(rnn_model, param_grid, cv=3, scoring=scoring_metrics,refit='accuracy',n_jobs=-1)
```

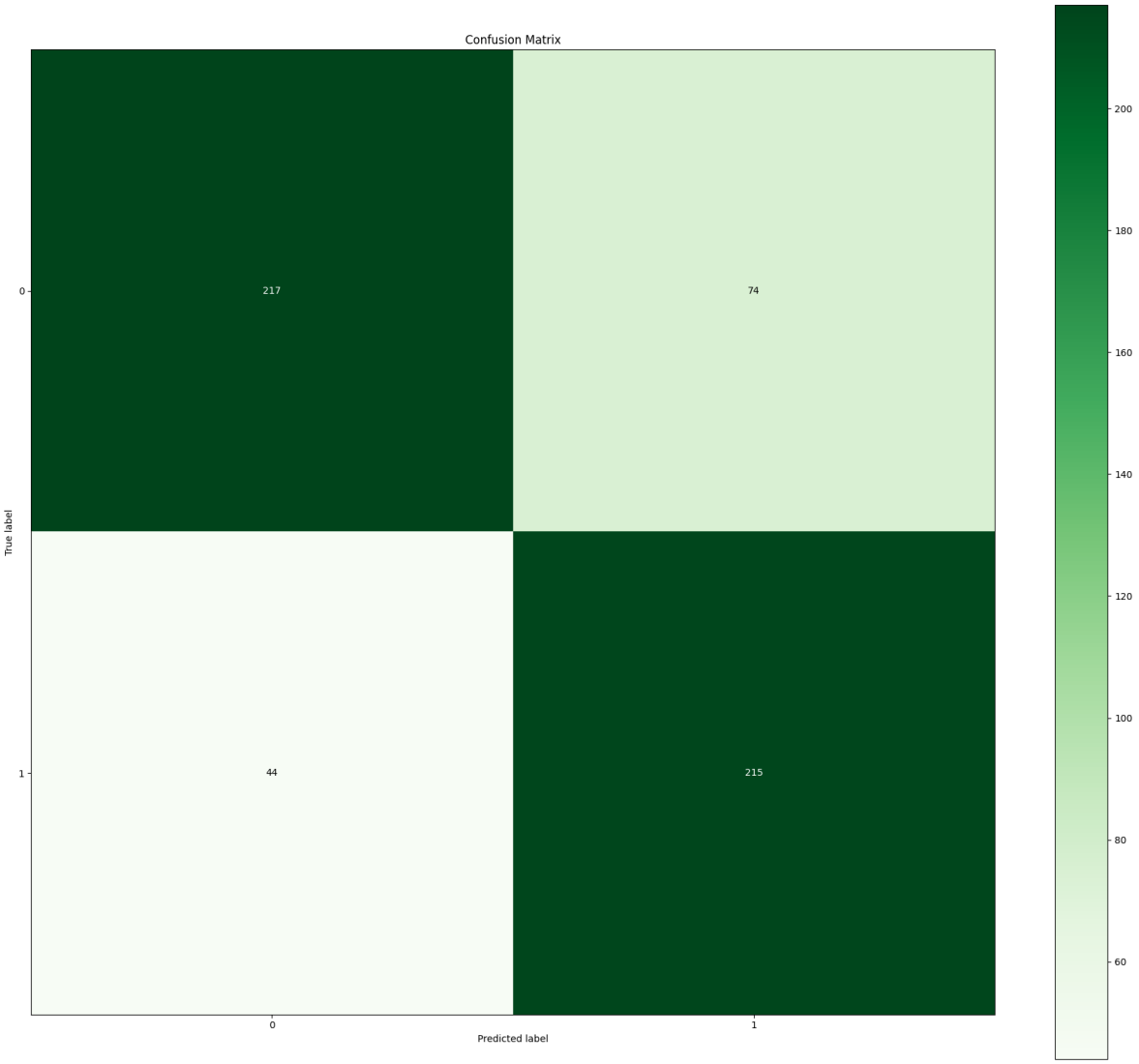
```
Mejores hiperparámetros: {'batch_size': 32, 'epochs': 10, 'units': 50}
```




RNN train Score vs CrossValidationscore
Training score
Cross-validation score
Score
1.0
0.9
0.8
0.7
0.6
0.5
200
400
600
800
1000
1200
1400
Training examples


Scores for cross validation in RNN
score
0.0
0.1
0.2
0.3
0.4
0.5
0.6
0.7
0.8
Accuracy
Precision
recall
f1
Folders

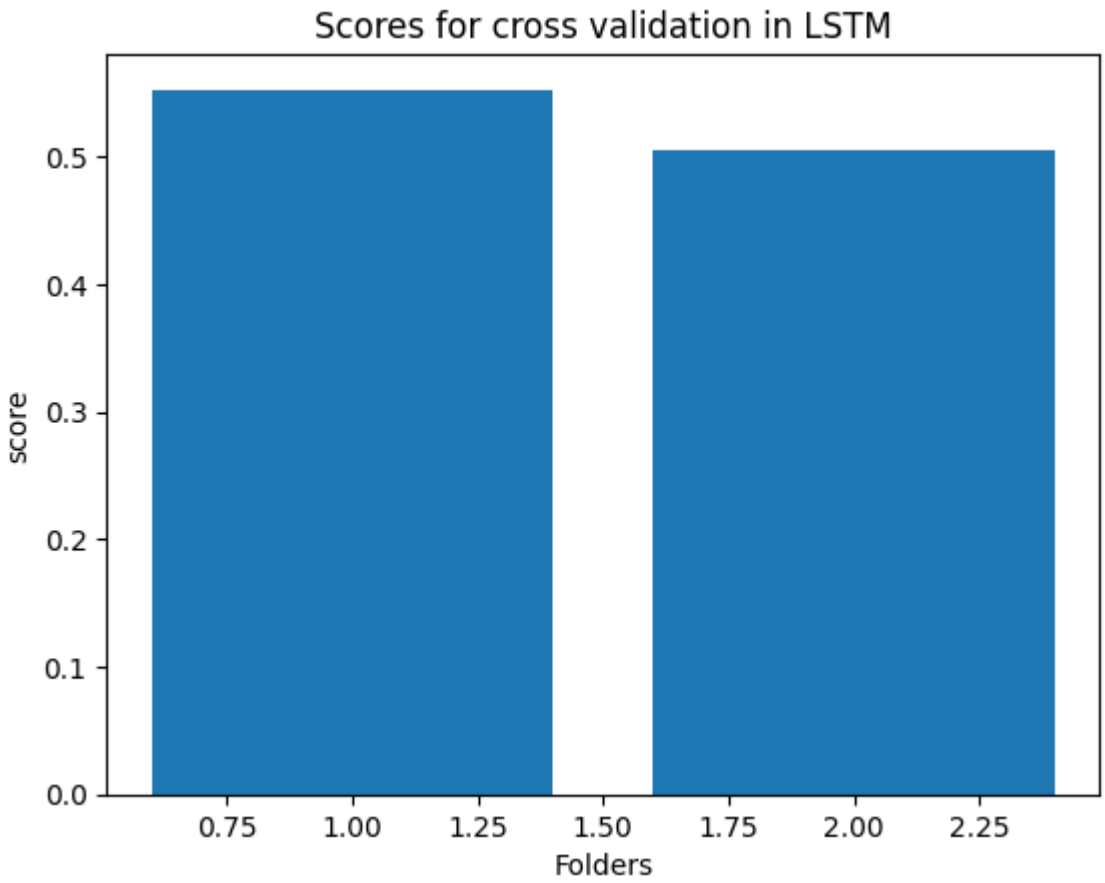

Digits ROC Curve
True Positive Rate
False Positive Rate
0.0
0.2
0.4
0.6
0.8
1.0
ROC curve of class 0 (area = 0.86)
ROC curve of class 1 (area = 0.86)
micro-average ROC curve (area = 0.85)
macro-average ROC curve (area = 0.86)

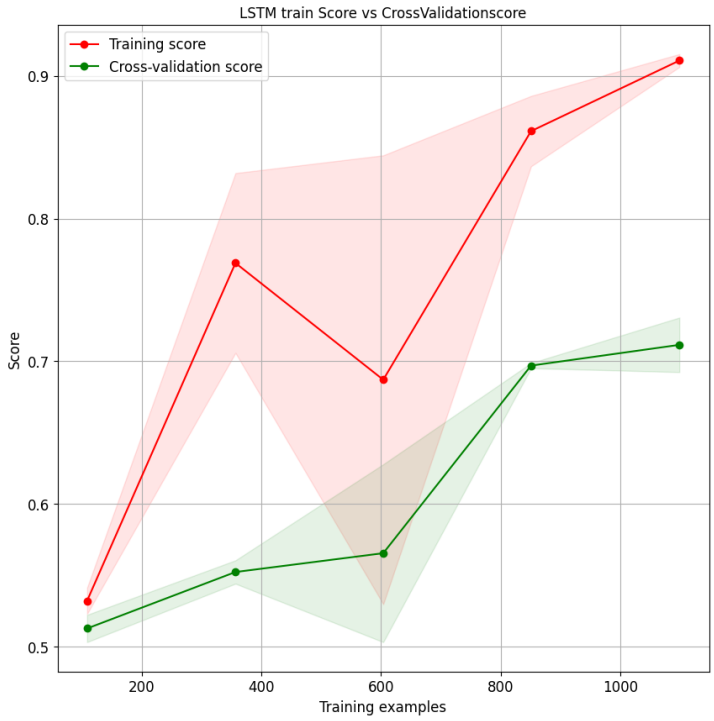


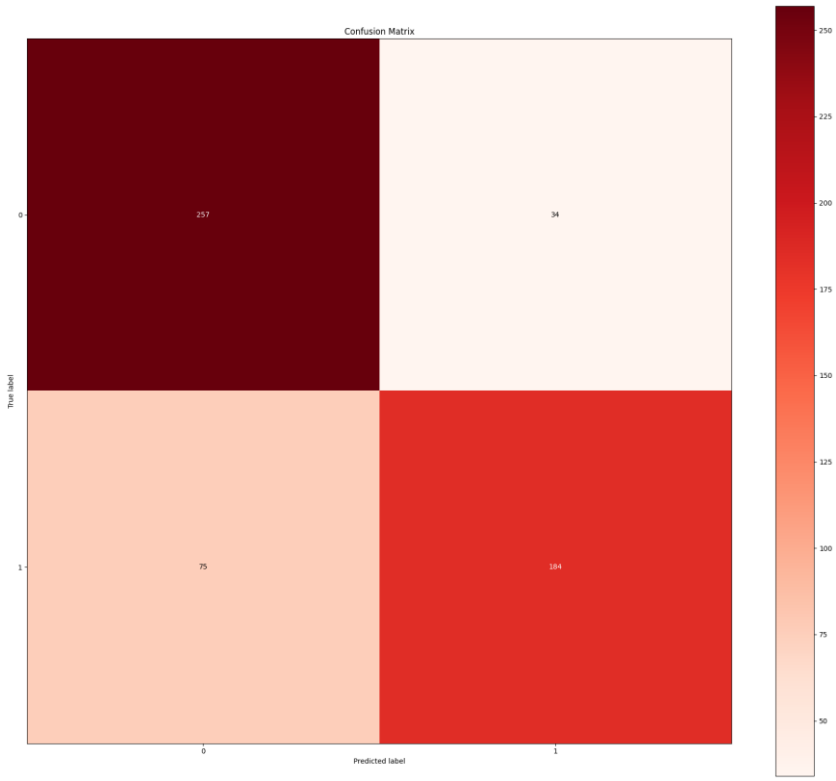
LSTM

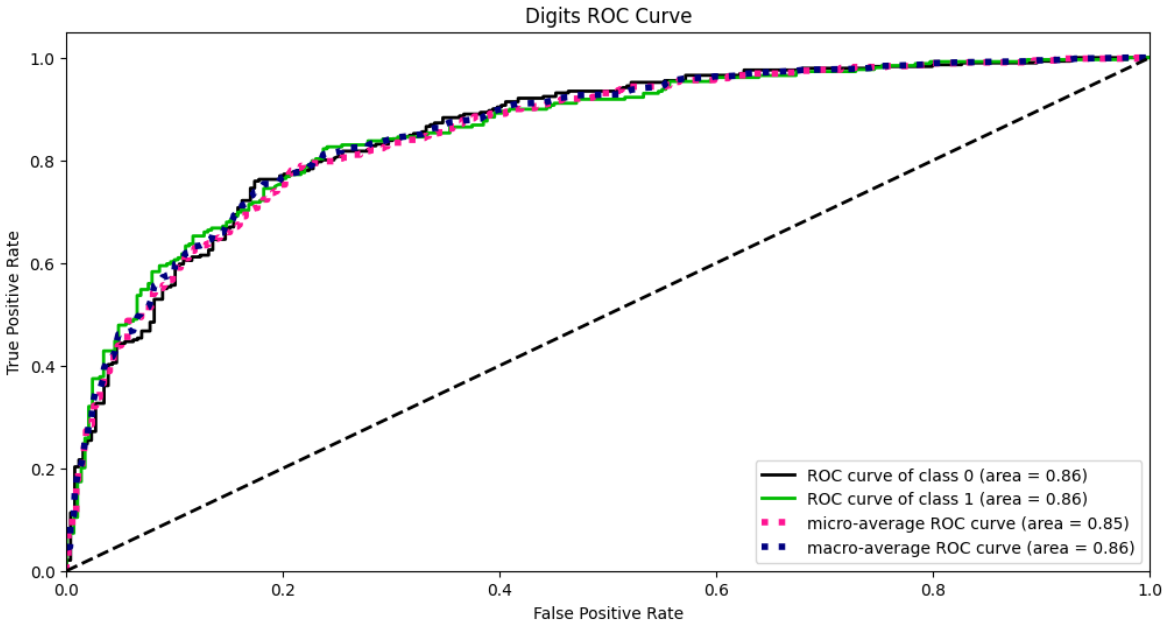
```
param_grid = {
    'epochs': [2, 4],
    'batch_size': [32, 64]
}

scoring_metrics = {
    'accuracy': make_scorer(accuracy_score),
    'precision': make_scorer(precision_score, average='weighted'),
    'recall': make_scorer(recall_score, average='weighted'),
    'f1': make_scorer(f1_score, average='weighted')
}

grid_search_lstm = GridSearchCV(lstm_model, param_grid, cv=2, scoring=scoring_metrics, refit='accuracy',n_jobs=-1)
```

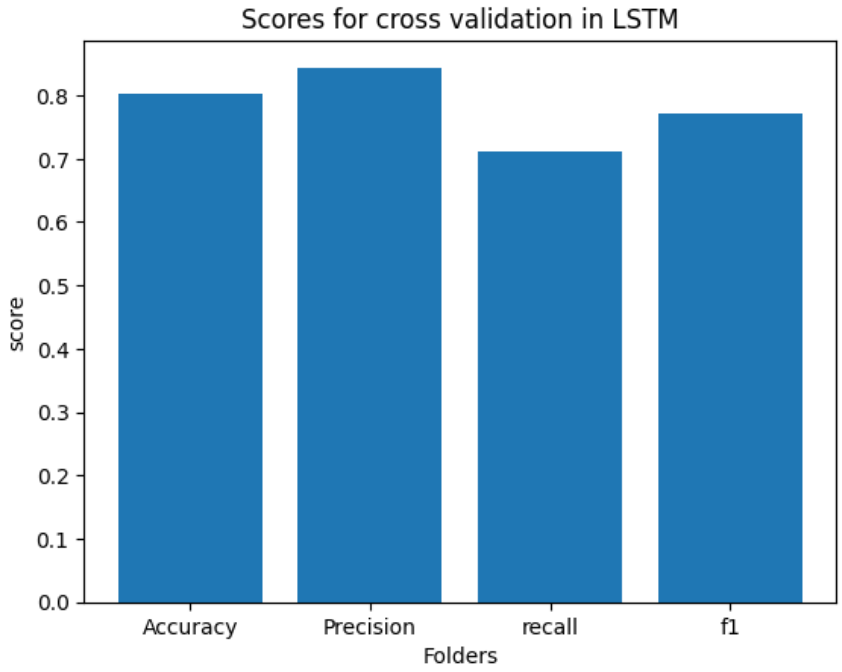
```
Mejores hiperparámetros: {'batch_size': 32, 'epochs': 4}
```


Scores for cross validation in LSTM
score
Folders
0.0
0.1
0.2
0.3
0.4
0.5
0.75
1.00
1.25
1.50
1.75
2.00
2.25


LSTM train Score vs CrossValidationscore
Training score
Cross-validation score
Score
Training examples
0.5
0.6
0.7
0.8
0.9
200
400
600
800
1000


Confusion Matrix
257
34
75
184
0
1
True label
Predicted label
250
225
200
175
150
125
100
75
50


Digits ROC Curve
True Positive Rate
False Positive Rate
0.0
0.2
0.4
0.6
0.8
1.0
ROC curve of class 0 (area = 0.86)
ROC curve of class 1 (area = 0.86)
micro-average ROC curve (area = 0.85)
macro-average ROC curve (area = 0.86)



Results Analysis:

Dummy Classifier:

Accuracy: 0.47

Precision: 0.22

Recall: 0.47

F1 Score: 0.30

This model has a low performance and acts as a basic reference. Its accuracy is low, indicating that it is not able to effectively distinguish between classes.

Recurrent Neural Network (RNN):

Accuracy: 0.79

Accuracy: 0.74

Recall: 0.83

F1 score: 0.78

The RNN shows significantly better performance than the Dummy Classifier. It has high accuracy and recall, indicating a robust ability to correctly classify both classes.

LSTM (Long Term Memory Neural Network):

Accuracy: 0.80

Accuracy: 0.84

Recall: 0.71

F1 score: 0.77

The LSTM model further improves performance compared to the RNN. It has high accuracy and is especially good for the positive class (lower recall but high precision).

Conclusions:

The Dummy Classifier provides a baseline, and any model that cannot outperform its metrics is ineffective.

The RNN shows significant improvement over the Dummy Classifier, but the LSTM outperforms both in terms of accuracy and overall performance.

Accuracy and recall are especially high for the LSTM, indicating that it can correctly classify the two classes effectively.

Opportunities for Improvement:

Explore Other Models:

Try other supervised learning models such as logistic regression, KNN, Random Forest, etc.

Each model has its strengths and weaknesses, and there may be one that fits the data better.

Hyperparameter Optimization:

Further tuning the hyperparameters of existing models could improve performance.

Feature Engineering:

Exploring and creating new features that could help the model learn patterns more effectively.

Handling Unbalanced Data:

If there is imbalance in classes, consider techniques such as oversampling or under sampling to handle this imbalance.

Overall Conclusion:

The LSTM model is the most promising among the three, with a trade-off between accuracy and recall. However, exploring other models and techniques can provide more insight into which one is best suited to your specific dataset. Supervised learning is an iterative process, and it is always valuable to experiment with different approaches to achieve the best possible performance.