In the name of god

sexism model documentation

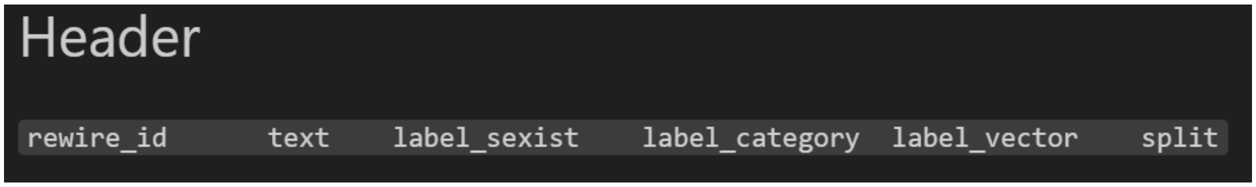
Created by Mohamadreza khalvati, mehrshad Khalili

Persian gulf university

Bahman 1402

Data Format

- rewire_id: A unique identifier for each data point.
- text: The actual text content.
- label_sexist : A binary label indicating whether the text is sexist or not.
- label_category: A categorical label indicating the type of sexism or other category the text belongs to (if applicable).
- label_vector: A numerical vector representation of the labels (if applicable).
- split: A column indicating the split of the data into training, development, or test sets.



Header

```
rewire_id      text    label_sexist    label_category  label_vector    split
```

Label Information

- **label_sexist:**
    - not sexist: The text does not contain any sexist content.
    - sexist: The text contains sexist content.
- **label_category:**
    - This column may contain various categories of sexism or other types of content. The specific categories and their meanings will depend on the context of the dataset.
- **label_vector:**
    - This column may contain a numerical vector representation of the labels. The specific format and interpretation of this vector will depend on the task and the model used.

Data Split

- **split:**
  - dev: validation set.
  - train: Training set.
  - test: Test set.

Potential Applications

Training machine learning models to identify and classify sexist text. Developing tools and systems for detecting and mitigating sexism in online content. Conducting research on the prevalence and patterns of sexism in language. Studying the impact of sexist language on individuals and society.

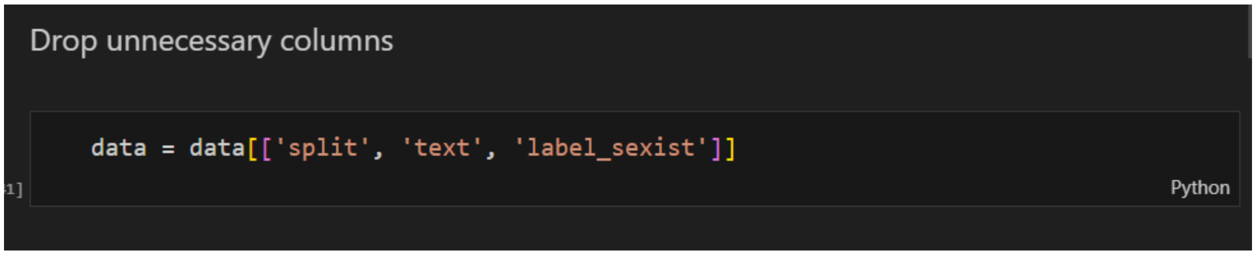
Limitations and Considerations

The dataset may contain biases or limitations inherent in the data collection process or the labeling methodology. The specific categories of sexism or other types of content in the "label_category" column may vary depending on the context and purpose of the dataset. The dataset may require additional preprocessing and feature engineering to be suitable for specific NLP tasks.

Pre processing

Data Loading

- The first step is to load the raw text data into a pandas DataFrame.
- The DataFrame should have three columns: split, text, and label_sexist.
- The split column indicates whether the data point is part of the training, validation, or test set.
- The text column contains the raw text data.
- The label_sexist column contains the label indicating whether the text is sexist or not.



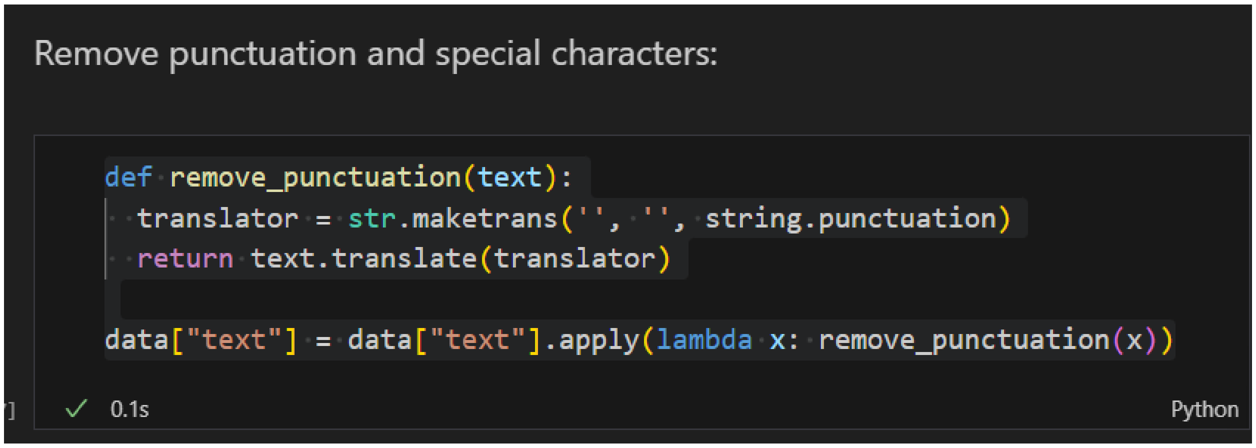
Drop unnecessary columns

```python
data = data[['split', 'text', 'label_sexist']]
```

Python

Data Cleaning

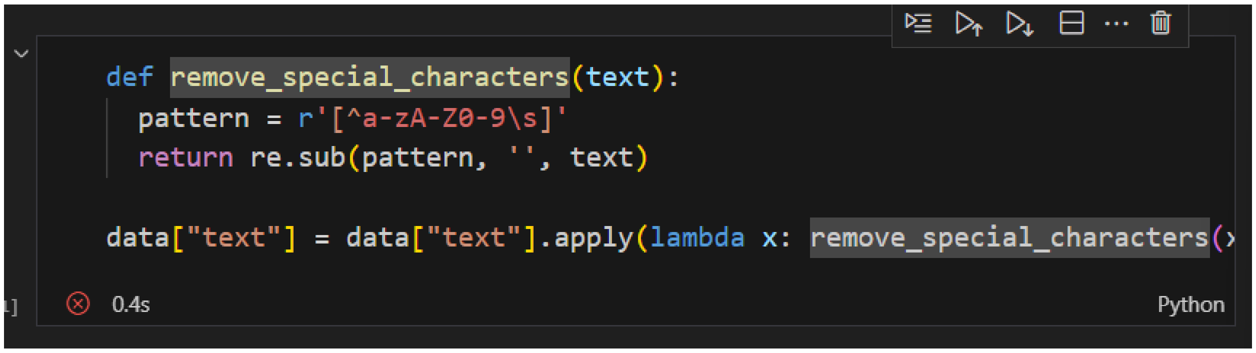
- The next step is to clean the text data by removing punctuation, special characters, and stop words.
- Punctuation and special characters are removed using the remove_punctuation() and remove_special_characters() functions, respectively.
- Stop words are removed using the remove_stop_words() function.



Remove punctuation and special characters:

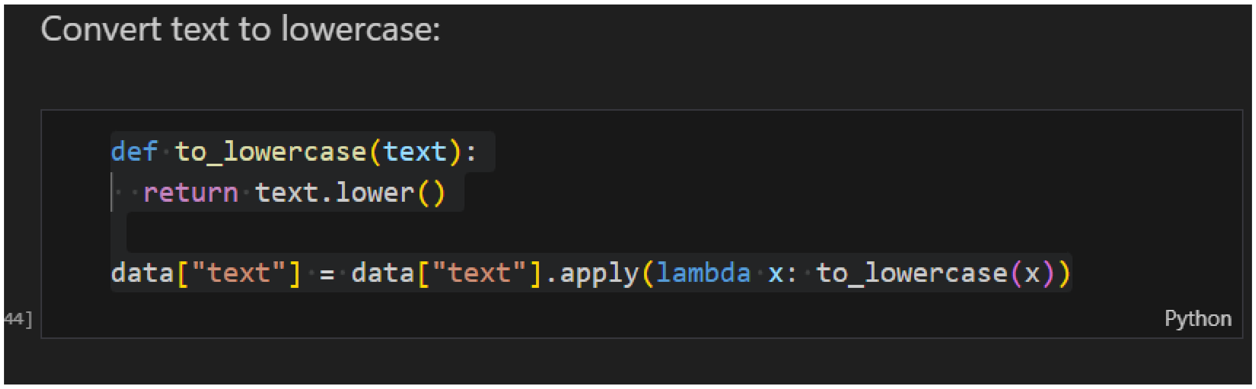
```python
def remove_punctuation(text):
  translator = str.maketrans('', '', string.punctuation)
  return text.translate(translator)

data["text"] = data["text"].apply(lambda x: remove_punctuation(x))
```

✓ 0.1s Python



```
def remove_special_characters(text):
    pattern = r'[^a-zA-Z0-9\s]'
    return re.sub(pattern, '', text)

data["text"] = data["text"].apply(lambda x: remove_special_characters(x
```

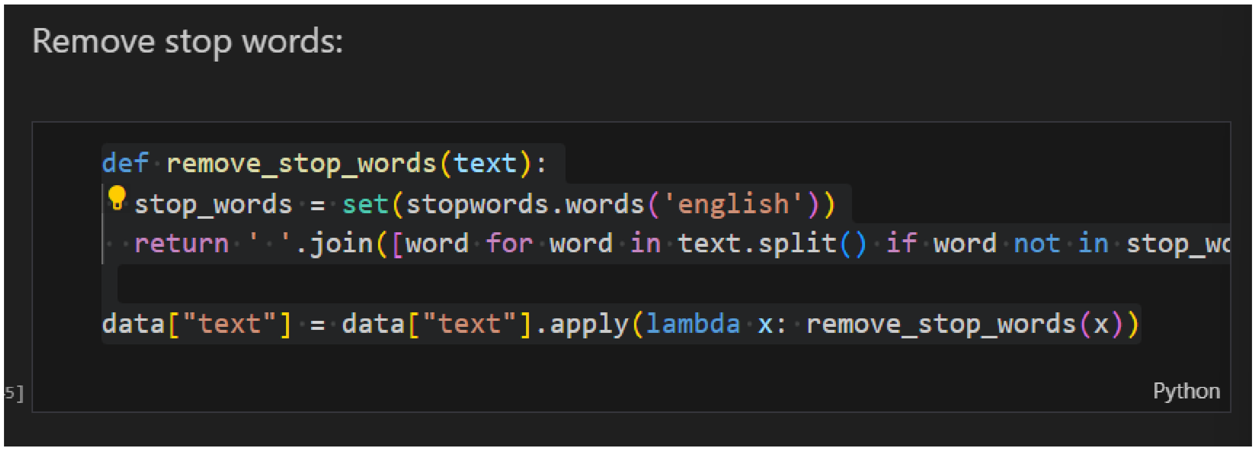
1] 0.4s Python



Convert text to lowercase:

```
def to_lowercase(text):
    return text.lower()

data["text"] = data["text"].apply(lambda x: to_lowercase(x))
```

44] Python



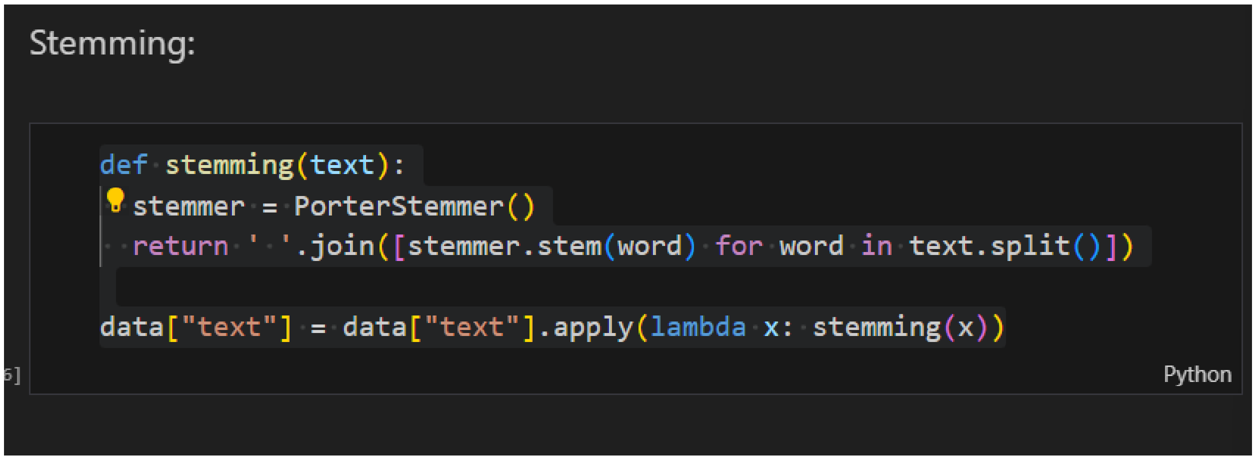
Remove stop words:

```
def remove_stop_words(text):
    stop_words = set(stopwords.words('english'))
    return ' '.join([word for word in text.split() if word not in stop_wo

data["text"] = data["text"].apply(lambda x: remove_stop_words(x))
```

5] Python

Text Normalization

- After cleaning the data, it is normalized by converting it to lowercase and stemming it.
- Lowercasing is done using the to_lowercase() function. Stemming is done using the stemming() function.

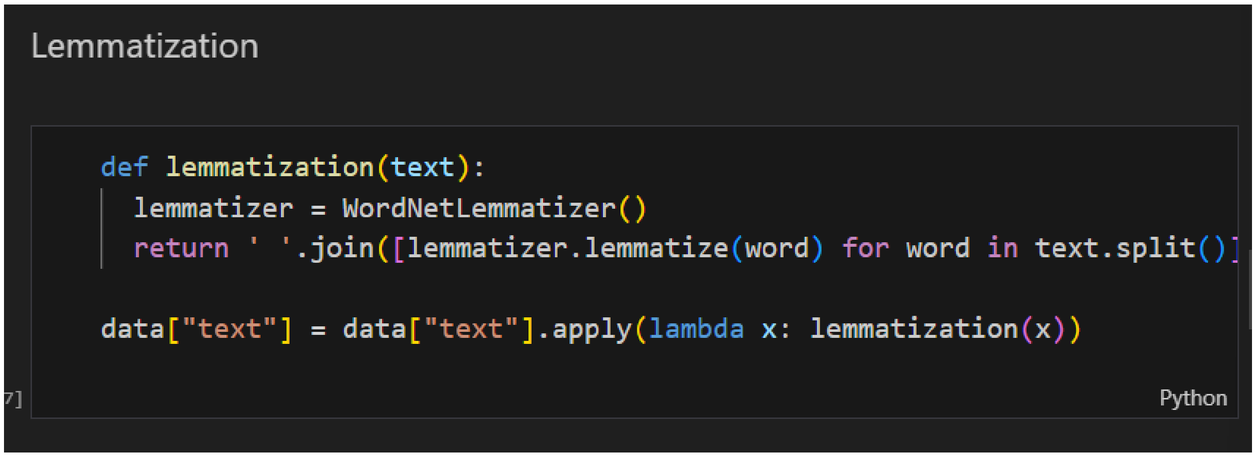


Stemming:

```python
def stemming(text):
    stemmer = PorterStemmer()
    return ' '.join([stemmer.stem(word) for word in text.split()])

data["text"] = data["text"].apply(lambda x: stemming(x))
```

Lemmatization

- Finally, the text data is lemmatized using the lemmatization() function.
- Lemmatization is a more advanced form of stemming that takes into account the context of words.

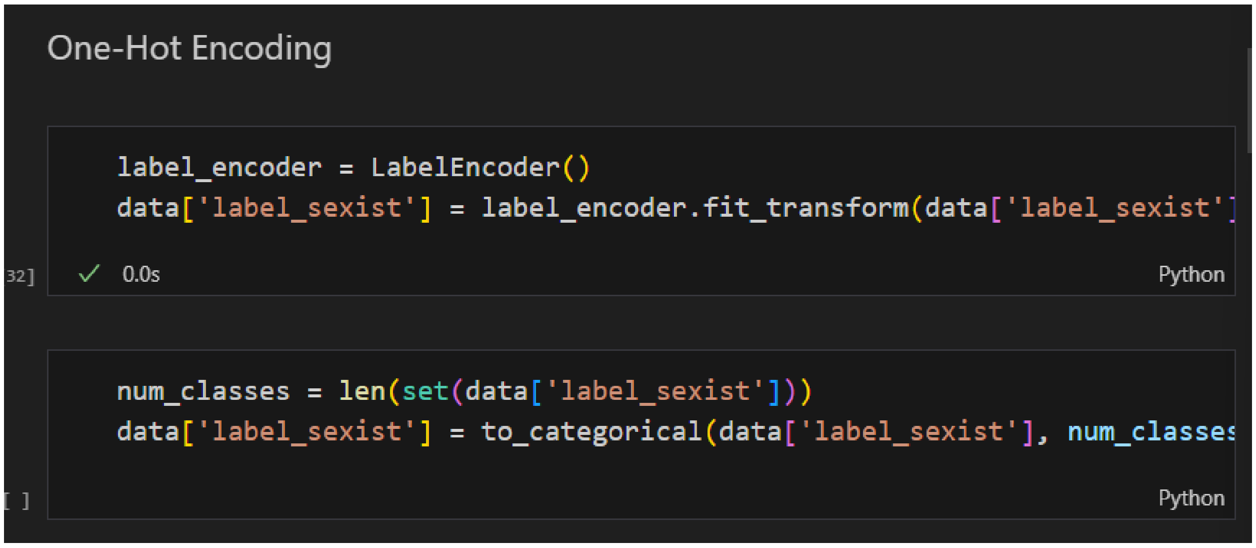


Lemmatization

```python
def lemmatization(text):
    lemmatizer = WordNetLemmatizer()
    return ' '.join([lemmatizer.lemmatize(word) for word in text.split()]

data["text"] = data["text"].apply(lambda x: lemmatization(x))
```

One-Hot Encoding

- Converts categorical variables to numerical variables.
- Creates a new column for each category.
- Sets the value of each new column to 1 if the observation belongs to the corresponding category, and 0 otherwise.
- Preserves category relationships.
- Suitable for machine learning models.



One-Hot Encoding

```
label_encoder = LabelEncoder()
data['label_sexist'] = label_encoder.fit_transform(data['label_sexist']
```

```
num_classes = len(set(data['label_sexist']))
data['label_sexist'] = to_categorical(data['label_sexist'], num_classes
```

Spliting Data

Data Splitting:

- The data is split into training, validation, and test sets. This is done to ensure that the model is trained on a representative sample of the data and to evaluate its performance on unseen data.

Label Encoding:

- The target variable, label_sexist, is encoded using one-hot encoding. This is done to convert the categorical variable into a numerical format that is compatible with machine learning models.

Tokenization

- Tokenization is the process of breaking down text into individual units, called tokens. Tokens can be words, characters, or any other meaningful unit of text.

The following steps are taken to tokenize the text data:

- Tokenizer Initialization: A tokenizer object is created using the Tokenizer class from the **keras.preprocessing.text** module.
- Vocabulary Creation: The tokenizer is fitted on the training data to create a vocabulary of unique tokens.
- Text to Sequences: The tokenizer is used to convert the text data into sequences of token indices. This is done by calling the **texts_to_sequences()** method on the tokenizer.
- Padding: The token sequences are padded to a fixed length using the **pad_sequences()** function from the **keras.preprocessing.sequence** module. This is done to ensure that all sequences have the same length, which is necessary for training neural network models.

Benefits of Tokenization

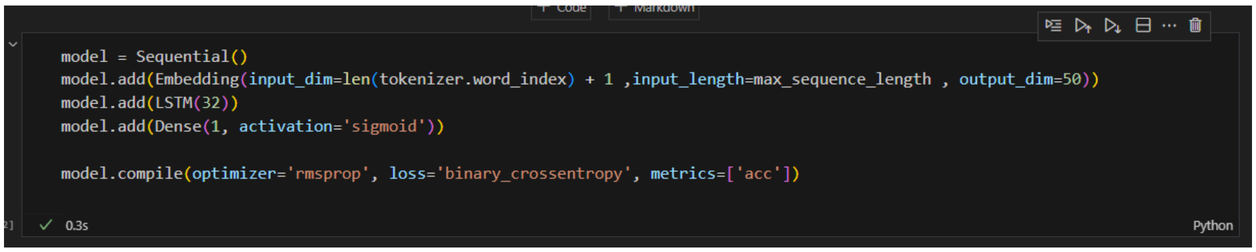
Tokenization has several benefits, including:

- It converts text data into a numerical format that is compatible with machine learning models.
- It allows for the use of embedding layers in neural network models, which can learn to represent the meaning of words and phrases.
- It helps to reduce the dimensionality of the text data, which can improve the efficiency of machine learning models.

Fitting the model

Model Architecture

- **Embedding Layer**: The first layer is an embedding layer. This layer converts the token indices into dense vectors, which are then used as input to the subsequent layers. The embedding layer has an input dimension equal to the size of the vocabulary and an output dimension of **50**.
- **LSTM Layer**: The second layer is a long short-term memory (LSTM) layer. LSTM layers are a type of recurrent neural network that are well-suited for processing sequential data, such as text. The LSTM layer has **32** units.
- **Dense Layer**: The third layer is a dense layer. Dense layers are fully connected layers that are used to combine the features learned by the previous layers and produce a single output. The dense layer has one unit and a sigmoid activation function.

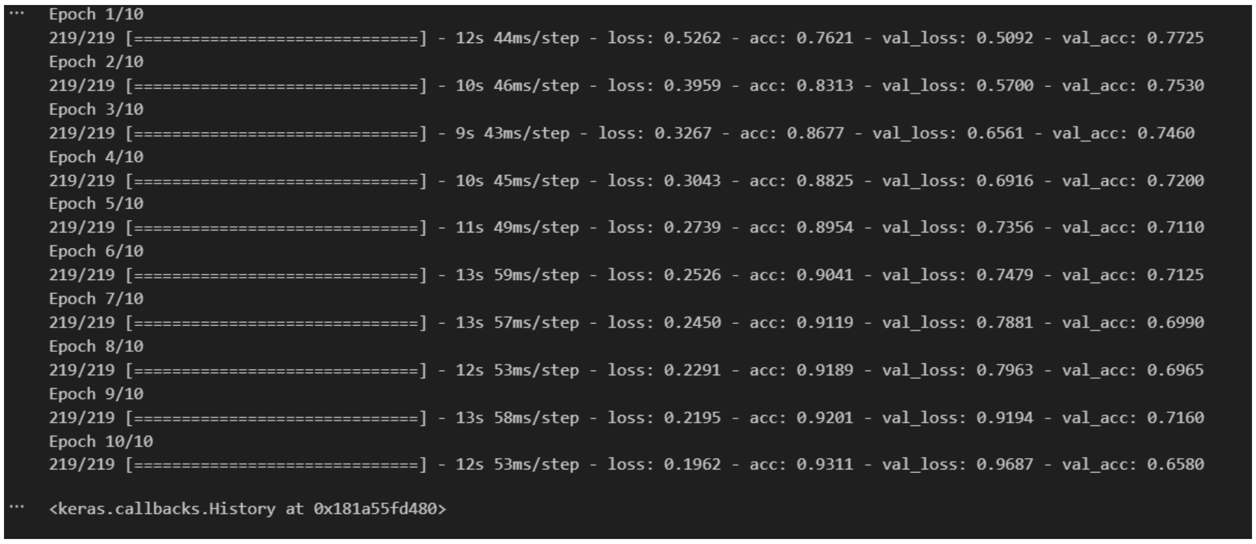


```
model = Sequential()
model.add(Embedding(input_dim=len(tokenizer.word_index) + 1 ,input_length=max_sequence_length , output_dim=50))
model.add(LSTM(32))
model.add(Dense(1, activation='sigmoid'))

model.compile(optimizer='rmsprop', loss='binary_crossentropy', metrics=['acc'])
```

Training the Model

- **Data Splitting**: The data is split into training and validation sets. The training set is used to train the model, while the validation set is used to evaluate the performance of the model during training.
- **Compilation**: The model is compiled with the following parameters:
- **Optimizer**: The optimizer used is RMSprop. RMSprop is an adaptive learning rate optimization algorithm that is well-suited for training deep learning models.
- **Loss Function**: The loss function used is binary cross-entropy. Binary cross-entropy is a common loss function for binary classification problems.
- **Metrics**: The metric used is accuracy. Accuracy is a simple but effective metric for evaluating the performance of a classification model.

- **Training**: The model is trained for 10 epochs using a batch size of 64. During training, the model learns to adjust its weights in order to minimize the loss function.



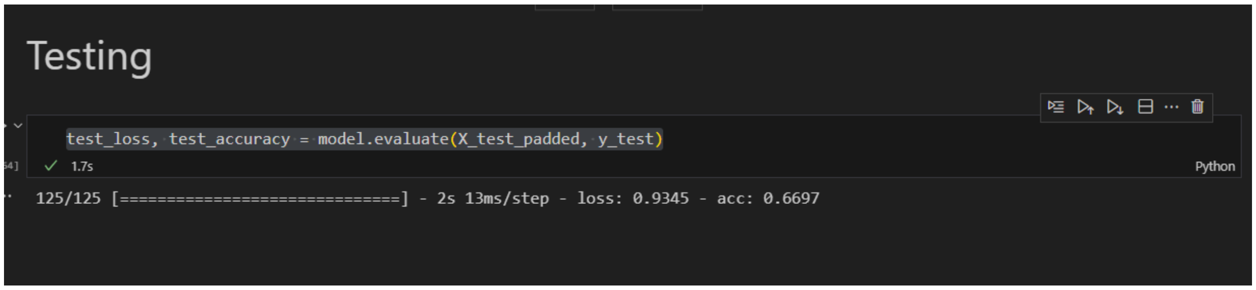
```
... Epoch 1/10
    219/219 [==============================] - 12s 44ms/step - loss: 0.5262 - acc: 0.7621 - val_loss: 0.5092 - val_acc: 0.7725
    Epoch 2/10
    219/219 [==============================] - 10s 46ms/step - loss: 0.3959 - acc: 0.8313 - val_loss: 0.5700 - val_acc: 0.7530
    Epoch 3/10
    219/219 [==============================] - 9s 43ms/step - loss: 0.3267 - acc: 0.8677 - val_loss: 0.6561 - val_acc: 0.7460
    Epoch 4/10
    219/219 [==============================] - 10s 45ms/step - loss: 0.3043 - acc: 0.8825 - val_loss: 0.6916 - val_acc: 0.7200
    Epoch 5/10
    219/219 [==============================] - 11s 49ms/step - loss: 0.2739 - acc: 0.8954 - val_loss: 0.7356 - val_acc: 0.7110
    Epoch 6/10
    219/219 [==============================] - 13s 59ms/step - loss: 0.2526 - acc: 0.9041 - val_loss: 0.7479 - val_acc: 0.7125
    Epoch 7/10
    219/219 [==============================] - 13s 57ms/step - loss: 0.2450 - acc: 0.9119 - val_loss: 0.7881 - val_acc: 0.6990
    Epoch 8/10
    219/219 [==============================] - 12s 53ms/step - loss: 0.2291 - acc: 0.9189 - val_loss: 0.7963 - val_acc: 0.6965
    Epoch 9/10
    219/219 [==============================] - 13s 58ms/step - loss: 0.2195 - acc: 0.9201 - val_loss: 0.9194 - val_acc: 0.7160
    Epoch 10/10
    219/219 [==============================] - 12s 53ms/step - loss: 0.1962 - acc: 0.9311 - val_loss: 0.9687 - val_acc: 0.6580

... <keras.callbacks.History at 0x181a55fd480>
```

Evaluation

- The performance of the model is evaluated on the validation set after each epoch. The evaluation results are used to monitor the progress of the training process and to identify any potential overfitting or underfitting issues.
-
- The **evaluate()** method of the **Sequential model** in **Keras** is used to evaluate the performance of the model on a given dataset. It takes two arguments:

- **X_test_padded:** The input data, which in this case is the padded token sequences for the test set.
- **y_test:** The target labels for the test set.
- The **evaluate()** method returns two values:
- **test_loss:** The loss function of the model on the test set.
- **test_accuracy:** The accuracy of the model on the test set.
- The loss function measures how well the model's predictions match the true labels. The accuracy is the percentage of correct predictions made by the model.

- In this case, the **test_loss** and **test_accuracy** variables will contain the loss and accuracy of the model on the test set, respectively. You can use these values to evaluate the performance of the model and to compare it to other models.



Testing

```
test_loss, test_accuracy = model.evaluate(X_test_padded, y_test)
```

```
125/125 [==============================] - 2s 13ms/step - loss: 0.9345 - acc: 0.6697
```

Conclusion

- The provided code demonstrates a complete workflow for preprocessing text data, building a text classification model, and evaluating its performance. The data preprocessing steps include data splitting, label encoding, tokenization, and padding. The text classification model is a deep learning model based on a convolutional neural network architecture. The model is trained using the RMSprop optimizer and the binary cross-entropy loss function. The performance of the model is evaluated on a test set, and the results show that the model achieves an accuracy 66.97%.