

# Natural Language Processing

Integrative Project

Br. Luis David Martínez Gutiérrez/ Sonia  
Estefanía Mendiá Martínez/ Hageo Juda Balam  
Méndez/ Angel David Sansores Cruz/ Yahir  
Benjamin Sulu Chi  
Prof. Mario Campos Soberanis

Universidad Politécnica de Yucatán



Data Engineering

9th quarter

December 6, 2024

# Contents

	Page
0.1 Purpose . . . . .	2
0.2 Project Objective . . . . .	2
0.3 Library Installation . . . . .	2
0.4 Library Imports . . . . .	3
0.5 Environment Setup and Data Download . . . . .	4
0.5.1 Setting up the Kaggle API . . . . .	4
0.5.2 Unzipping the Dataset . . . . .	4
0.6 Loading Training and Test Data . . . . .	4
0.6.1 Alternative Approach (with CSV Paths) . . . . .	5
0.6.2 Displaying the First Rows of the Training DataFrame . . . . .	5
0.7 Displaying Class Distribution . . . . .	6
0.7.1 Displaying Example Reviews . . . . .	6
0.8 Printing Preprocessed and Original Reviews . . . . .	7
0.8.1 Importing Libraries . . . . .	7
0.8.2 Defining the <code>preprocess_text</code> Function . . . . .	8
0.8.3 Applying Preprocessing to the Data . . . . .	8
0.9 Model Definition and Data Preparation . . . . .	9
0.10 Evaluation on the Test Set . . . . .	12
0.10.1 Confusion Matrix . . . . .	14
0.11 Deep Neural Network (DNN) . . . . .	15
0.11.1 DNNClassifier Class: Architecture and Functionalities . . . . .	15
0.11.2 . . . . .	18
0.12 Analysis of DNN with Cross-Validation . . . . .	19
0.13 Analyzing the Reversed Polarity Generator . . . . .	21
0.13.1 Using Only the 'text' Column for Analysis . . . . .	22
0.14 Sequence-to-Sequence Model: Positive to Negative Review: Model 3: Inverted Polarity 1 . . . . .	23
0.14.1 pandas DataFrame into Hugging Face Dataset Conversion . . . . .	25
0.15 Sequence-to-Sequence Model Evaluation . . . . .	32
0.16 Evaluation with Device Check . . . . .	33
0.17 Model 4: Encoder - Decoder . . . . .	36
0.18 API using <i>ngrok</i> . . . . .	38

## 0.1 Purpose

This project aims to create a system that classifies Amazon reviews as positive or negative using two different approaches: one based on TF-IDF and logistic regression, and another using deep neural networks (DNN). In addition, we will develop a generative model capable of reversing the polarity of a review, that is, transforming a positive review into a negative one and vice versa. Finally, we will make these models available through two API endpoints.

## 0.2 Project Objective

The purpose of this project is to develop a system capable of:

- Classifying product reviews into sentiment categories (positive, negative, or neutral).
- Generating automatic reviews based on a generative model, such as Recurrent Neural Networks (RNNs).

## 0.3 Library Installation

```
1 !pip install numpy pandas scikit-learn nltk keras tensorflow flask
2 !pip install --upgrade tensorflow
3 !pip install --upgrade keras
```

**Purpose:** Installs necessary libraries for the project.

Breakdown:

- **numpy**: For numerical operations and array manipulation.
- **pandas**: For data analysis and manipulation (e.g., reading and cleaning CSV files).
- **scikit-learn**: For machine learning tasks like classification, regression, and feature extraction.
- **nltk**: For natural language processing tasks like tokenization and stemming.
- **keras**: For building and training deep learning models.
- **tensorflow**: The backend framework for Keras, providing tensor operations and machine learning algorithms.
- **flask**: For creating web applications and APIs to serve the models.

## 0.4 Library Imports

```
4
5 import os
6 import pandas as pd
7 import numpy as np
8 import matplotlib.pyplot as plt
9 import seaborn as sns
10 import re
11 import nltk
12 from sklearn.model_selection import train_test_split, StratifiedKFold,
    cross_val_score
13 from sklearn.metrics import classification_report, confusion_matrix,
    accuracy_score
14 from sklearn.feature_extraction.text import TfidfVectorizer
15 from sklearn.linear_model import LogisticRegression
16 from sklearn.base import BaseEstimator, ClassifierMixin
17 from tensorflow.keras.preprocessing.text import Tokenizer
18 from tensorflow.keras.preprocessing.sequence import pad_sequences
19 from tensorflow.keras.models import Sequential, Model
20 from tensorflow.keras.layers import Embedding, Dense, LSTM, Input
```

**Purpose:** Imports specific modules from the installed libraries for various tasks.  
Breakdown:

- **os:** For interacting with the operating system.
- **pandas:** For data manipulation and analysis.
- **numpy:** For numerical operations and array manipulation.
- **matplotlib.pyplot:** For plotting data visualizations.
- **seaborn:** For statistical data visualization.
- **re:** For regular expressions (text pattern matching).
- **nltk:** For natural language processing tasks.
- **sklearn:** For machine learning tasks.
- **tensorflow.keras:** For building and training deep learning models.

Overall, this part sets up the environment for the Amazon review classification and generation project. It installs and imports the necessary libraries for data manipulation, visualization, text preprocessing, feature extraction, model building, and evaluation.

## 0.5 Environment Setup and Data Download

Data preprocessing is essential before feeding the data into any Machine Learning model or neural network. In this stage, the text is cleaned and transformed to facilitate its processing.

### 0.5.1 Setting up the Kaggle API

To begin, the environment was set up using Google Colab, and the Amazon data was downloaded from Kaggle.

```
22 !pip install kaggle
23 import os
24 os.environ['KAGGLE_CONFIG_DIR'] = '/content'
25 !kaggle datasets download -d kritanjalijain/amazon-reviews
26 !unzip amazon-reviews.zip -d amazondata
```

**Purpose:** Download and extract the Amazon review data.

- `\!pwd`: Prints the current working directory.
- `os.environ['KAGGLE_CONFIG_DIR'] = '/content'`: Sets the environment variable `KAGGLE_CONFIG_DIR` to the `/content` directory. This is where the Kaggle API credentials are stored.
- `\!kaggle datasets download -d kritanjalijain/amazon-reviews`: Downloads the Amazon Reviews dataset from Kaggle.

### 0.5.2 Unzipping the Dataset

```
28 !unzip amazon-reviews.zip -d amazondata
```

**Purpose:** It downloads the dataset from Kaggle and unzips it for further processing.

- `\!unzip amazon-reviews.zip -d amazondata`: Unzips the downloaded dataset into the `amazondata` directory.

## 0.6 Loading Training and Test Data

```
29 # Load training and test data
30 train_df = pd.read_csv('amazondata/train.csv', header=None, names=['
    polarity', 'title', 'text'])
31 test_df = pd.read_csv('amazondata/test.csv', header=None, names=['
    polarity', 'title', 'text'])
32
33 # Sample 10% of the data
34 train_df = train_df.sample(frac=0.1, random_state=42)
```

```
35 test_df = test_df.sample(frac=0.1, random_state=42)
36
37 print(f"Size of training set: {train_df.shape}")
38 print(f"Size of test set: {test_df.shape}")
```

Breakdown:

- `pd.read_csv()`: Reads the CSV files `train.csv` and `test.csv` into pandas DataFrames.
- `header=None, names=['polarity', 'title', 'text']`: Specifies that the CSV files don't have headers and assigns the column names `polarity`, `title`, and `text` to the DataFrames.
- `train_df.sample(frac=0.1, random_state=42)`: Samples 10% of the rows from the `train_df` DataFrame with a random state of 42 for reproducibility.
- `print(f"Size of training set: {train_df.shape}")`: Prints the shape (number of rows and columns) of the training set.
- `print(f"Size of test set: {test_df.shape}")`: Prints the shape of the test set.

### 0.6.1 Alternative Approach (with CSV Paths)

```
39 # Path to your CSV files
40 train_csv_path = 'amazondata/train.csv'
41 test_csv_path = 'amazondata/test.csv'
42
43 # Load training data with column names
44 train_df = pd.read_csv(train_csv_path, header=None, names=['polarity', 'title', 'text'])
45
46 # Load test data with column names
47 test_df = pd.read_csv(test_csv_path, header=None, names=['polarity', 'title', 'text'])
```

**Purpose:** This alternative approach is similar to the first one, but it explicitly defines the paths to the CSV files.

### 0.6.2 Displaying the First Rows of the Training DataFrame

```
48 print(train_df.head())
```

**Purpose:** Prints the first 5 rows of the `train_df` DataFrame, providing a quick overview of the data.

## 0.7 Displaying Class Distribution

```
49 print("Class distribution in the training set:")
50 print(train_df['polarity'].value_counts())
51
52 print("\nClass distribution in the training set:")
53 print(test_df['polarity'].value_counts())
```

Breakdown:

- `print("Distribución de clases en el conjunto de entrenamiento:")`: Prints a descriptive message about the class distribution in the training set.
- `print(train_df['polarity'].value_counts())`: Prints the frequency of each class (positive and negative) in the `polarity` column of the `train_df` DataFrame.
- `print("\nDistribución de clases en el conjunto de prueba:")`: Prints a descriptive message about the class distribution in the test set.
- `print(test_df['polarity'].value_counts())`: Prints the frequency of each class in the `polarity` column of the `test_df` DataFrame.

### 0.7.1 Displaying Example Reviews

```
54 print("\nExample of a positive review")
55 print(train_df[train_df['polarity'] == 1]['review'].iloc[0])
56
57 print("\nExample of a negative review")
58 print(train_df[train_df['polarity'] == 0]['review'].iloc[0])
```

**Purpose:** provides insights into the distribution of positive and negative reviews in the training and test sets. It also displays examples of positive and negative reviews to illustrate the data.

Breakdown:

- `print("\nEjemplo de reseña positiva")`: Prints a descriptive message about an example of a positive review.
- `print(train_df[train_df['polarity'] == 1]['review'].iloc[0])`: Prints the first positive review from the `train_df` DataFrame.
- `print("\nEjemplo de reseña negativa")`: Prints a descriptive message about an example of a negative review.
- `print(train_df[train_df['polarity'] == 0]['review'].iloc[0])`: Prints the first negative review from the `train_df` DataFrame.

## 0.8 Printing Preprocessed and Original Reviews

```
59 import re
60 import nltk
61 nltk.download('stopwords')
62 from nltk.corpus import stopwords
63
64 stop_words = set(stopwords.words('english'))
65
66 def preprocess_text(text):
67     # Convertir a min sculas
68     text = text.lower()
69     # Eliminar etiquetas HTML
70     text = re.sub(r'<.*?>', '', text)
71     # Eliminar caracteres especiales y n meros
72     text = re.sub(r'[~a-zA-Z\s]', '', text)
73     # Eliminar palabras vac as
74     tokens = text.split()
75     tokens = [word for word in tokens if word not in stop_words]
76     # Unir tokens
77     text = ' '.join(tokens)
78     return text
79
80 # Aplicar preprocesamiento
81 train_df['clean_review'] = train_df['review'].apply(preprocess_text)
82 test_df['clean_review'] = test_df['review'].apply(preprocess_text)
```

**Purpose:** It involves a series of steps to clean, normalize, and prepare text data for further analysis or modeling.

Breakdown:

### 0.8.1 Importing Libraries

```
83 import nltk
84 nltk.download('stopwords')
85 from nltk.corpus import stopwords
```

- `import nltk`: Imports the Natural Language Toolkit (NLTK) library.
- `nltk.download('stopwords')`: Downloads the stop words corpus, which contains common words that are often removed in text preprocessing.
- `from nltk.corpus import stopwords`: Imports the stop words corpus from NLTK.



## 0.8.2 Defining the preprocess\_text Function

```
86 def preprocess_text(text):
87     # Convert to lowercase
88     text = text.lower()
89
90     # Remove HTML tags
91     text = re.sub(r'<[^>]+>', '', text)
92
93     # Remove special characters and numbers
94     text = re.sub(r'[^\w\s]', '', text)
95
96     # Remove stop words
97     tokens = text.split()
98     tokens = [word for word in tokens if not word in stopwords.words('
    english')]
99
100    # Join tokens back into text
101    text = ' '.join(tokens)
102
103    return text
```

- `text = text.lower()`: Converts the text to lowercase.
- `text = re.sub(r'<[^>]+>', '', text)`: Removes HTML tags from the text.
- `text = re.sub(r'[^\w\s]', '', text)`: Removes special characters and numbers from the text.
- `tokens = text.split()`: Splits the text into individual words.
- `tokens = [word for word in tokens if not word in stopwords.words('english')]`: Removes stop words from the list of tokens.
- `text = ' '.join(tokens)`: Joins the remaining tokens back into a single text string.

## 0.8.3 Applying Preprocessing to the Data

```
104 train_df['review'] = train_df['review'].apply(preprocess_text)
105 test_df['review'] = test_df['review'].apply(preprocess_text)
106
107 print(train_df['review'].iloc[0])
108 print("\nOriginal review:")
109 print(train_df['review'].iloc[0])
```

- `train_df['review'] = ...`: Applies the `preprocess_text` function to each review in the `train_df` DataFrame.

- `test_df['review'] = ...`: Applies the `preprocess_text` function to each review in the `test_df` DataFrame.
- `print(train_df['review'].iloc[0])`: Prints the preprocessed review at index 0 from the `train_df` DataFrame.
- `print("\nReseña original:")`: Prints a descriptive message about the original review.
- `print(train_df['review'].iloc[0])`: Prints the original review at index 0 from the `train_df` DataFrame.

Overall, this code preprocesses the text data by converting it to lowercase, removing HTML tags, special characters, numbers, and stop words. It then applies the preprocessing function to both the training and test data and displays examples of preprocessed and original reviews.

## 0.9 Model Definition and Data Preparation

```
110 from sklearn.base import BaseEstimator, ClassifierMixin
111 from sklearn.feature_extraction.text import TfidfVectorizer
112 from sklearn.linear_model import LogisticRegression
113
114 class TfidfLogisticClassifier(BaseEstimator, ClassifierMixin):
115     def __init__(self):
116         self.tfidf = TfidfVectorizer(max_features=5000)
117         self.model = LogisticRegression(max_iter=1000)
118
119     def fit(self, X, y):
120         X_tfidf = self.tfidf.fit_transform(X)
121         self.model.fit(X_tfidf, y)
122         self.classes_ = self.model.classes_ # Aadir esta linea
123         return self
124
125     def predict(self, X):
126         X_tfidf = self.tfidf.transform(X)
127         return self.model.predict(X_tfidf)
128
129     def predict_proba(self, X):
130         X_tfidf = self.tfidf.transform(X)
131         return self.model.predict_proba(X_tfidf)
```

**Purpose:** This section sets the stage for the model training and evaluation process. By defining the model architecture and preparing the data, you ensure that the model can be trained effectively and evaluated accurately.

Breakdown:

- `BaseEstimator, ClassifierMixin`: Inherits from `BaseEstimator` and `ClassifierMixin` to ensure compatibility with scikit-learn's pipeline and grid search functionalities.

- `TfidfVectorizer(max_features=5000)`: Initializes a TF-IDF vectorizer with a maximum of 5000 features.
- `LogisticRegression(max_iter=1000)`: Initializes a logistic regression model with a maximum of 1000 iterations.
- `fit(X, y)`: Fits the model to the training data `X` and labels `y`.
  - Transforms the text data `X` into TF-IDF features using `self.tfidf.fit_transform(X)`.
  - Fits the logistic regression model to the transformed features and labels using `self.model.fit(X_tfidf, y)`.
  - Stores the class labels in `self.classes_`.
- `predict(X)`: Predicts the class labels for new data `X`.
  - Transforms the new data into TF-IDF features using `self.tfidf.transform(X)`.
  - Uses the fitted logistic regression model to predict the labels.
- `predict_proba(X)`: Predicts the class probabilities for new data `X`.
  - Transforms the new data into TF-IDF features using `self.tfidf.transform(X)`.
  - Uses the fitted logistic regression model to predict the class probabilities.

## Data Preparation

```
132 X_train = train_df['clean_review']
133 y_train = train_df['polarity']
```

- `X_train = train_df['clean_review']`: Extracts the preprocessed reviews from the `train_df` DataFrame as the features `X_train`.
- `y_train = train_df['polarity']`: Extracts the polarity labels from the `train_df` DataFrame as the target variable `y_train`.

## Cross Validation with K-folds

```
134 from sklearn.model_selection import StratifiedKFold, cross_val_score
135
136 tfidf_logistic_clf = TfidfLogisticClassifier()
137
138 # Cross Validation with K-folds
139 kfold = StratifiedKFold(n_splits=5, shuffle=True, random_state=42)
140 scores = cross_val_score(tfidf_logistic_clf, X_train, y_train, cv=kfold,
141                           scoring='accuracy')
142
143 print("Average accuracy in cross validation:", scores.mean())
```

**Purpose:** to assess the model's generalization performance and to prevent overfitting.

Overfitting occurs when a model becomes too complex and fits the training data too closely, leading to poor performance on new, unseen data. By using cross-validation, we can get a more accurate estimate of how well the model will perform on new data, as it is evaluated on multiple subsets of the training data.

## Benefits of Cross-Validation

- **Identify the optimal hyperparameters:** By testing different hyperparameter settings on multiple folds, we can choose the best combination that leads to the best performance.
- **Compare different models:** We can compare the performance of different models on the same cross-validation folds to select the best-performing model.
- **Assess the model's robustness:** Cross-validation can help identify potential issues in the model, such as sensitivity to data variations or overfitting.

By using cross-validation, we can build more reliable and robust machine learning models.  
Breakdown:

## Imports

```
143 from sklearn.model_selection import StratifiedKFold, cross_val_score
```

- `from sklearn.model_selection import StratifiedKFold`: Imports the `StratifiedKFold` class for stratified k-fold cross-validation, ensuring that each fold has a similar distribution of class labels.
- `from sklearn.model_selection import cross_val_score`: Imports the `cross_val_score` function for evaluating the model using cross-validation.

## Model Initialization

```
144 tfidf_logistic_clf = TfidfLogisticClassifier()
```

- `model = TfidfLogisticClassifier()`: Initializes an instance of the `TfidfLogisticClassifier` class created earlier.

## Cross-Validation

```
145 kfold = StratifiedKFold(n_splits=5, shuffle=True, random_state=42)
146 scores = cross_val_score(tfidf_logistic_clf, X_train, y_train, cv=kfold,
147                           scoring='accuracy')
147 print("Average accuracy in cross validation:", scores.mean())
```

- `StratifiedKFold(n_splits=5, shuffle=True, random_state=42)`: Creates a stratified k-fold cross-validator with 5 folds, shuffling the data before splitting and setting a random seed for reproducibility.
- `cross_val_score(tfidf_logistic_clf, X_train, y_train, cv=kfold, scoring='accuracy')`: Performs cross-validation on the `tfidf_logistic_clf` model using the `X_train` and `y_train` data, with the specified `kfold` cross-validator and `accuracy` as the scoring metric.
- `print("Average Accuracy in Cross-Validation:", scores.mean())`: Prints the average accuracy across the 5 folds of cross-validation.

## Model Training

```
148
149 tfidf_logistic_clf.fit(X_train, y_train)
```

- `tfidf_logistic_clf.fit(X_train, y_train)`: Trains the `tfidf_logistic_clf` model on the entire training data `X_train` and `y_train`.

## 0.10 Evaluation on the Test Set

TF-IDF Classifier with Logistic Regression Evaluation on the Test Set: We already trained the model and performed cross-validation. Now, we will evaluate the model on the test set and generate the classification report.

### Analysis of Metrics

- **Precision (Accuracy)**: Proportion of correct predictions over the total predictions.
  - **Precision by class**: Ability of the classifier to not label a negative sample as positive and vice versa.
- **Recall (Sensitivity)**: Ability of the classifier to find all positive samples.
- **F1-Score**: Harmonic mean of precision and recall.

**Interpretation:** analyze the values obtained in the classification report to understand the performance of the model in each class. Focus on metrics such as precision, recall, and F1-score to evaluate how well the model distinguishes between classes and handles the test set.

```
150 from sklearn.metrics import classification_report, confusion_matrix
151
152 # Datos de prueba
153 X_test = test_df['clean_review']
154 y_test = test_df['polarity']
```

```
155
156 # Predicciones
157 y_pred_tfidf = tfidf_logistic_clf.predict(X_test)
158
159 # Informe de clasificaci n
160 print("Informe de clasificaci n para el clasificador TF-IDF con
      Regresi n Log stica:")
161 print(classification_report(y_test, y_pred_tfidf))
162
163 # Matriz de confusi n
164 cm_tfidf = confusion_matrix(y_test, y_pred_tfidf)
165 print("Matriz de confusi n:")
166 print(cm_tfidf)
```

**Purpose:** to assess the final performance of the trained machine learning model on a completely unseen dataset. This helps to estimate how well the model will generalize to real-world data.

Breakdown:

## Imports

```
167 from sklearn.metrics import classification_report, confusion_matrix
```

- `from sklearn.metrics import classification_report, confusion_matrix`: Imports the `classification_report` and `confusion_matrix` functions from the `sklearn.metrics` module. These functions are used to evaluate the performance of the classification model.

## Data Preparation

```
168 X_test = test_df['clean_review']
169 y_test = test_df['polarity']
```

- `X_test = test_df['clean_review']`: Extracts the preprocessed reviews from the `test_df` DataFrame and assigns them to `X_test`.
- `y_test = test_df['polarity']`: Extracts the true polarity labels from the `test_df` DataFrame and assigns them to `y_test`.

## Model Predictions

```
170 y_pred_tfidf = tfidf_logistic_clf.predict(X_test)
```

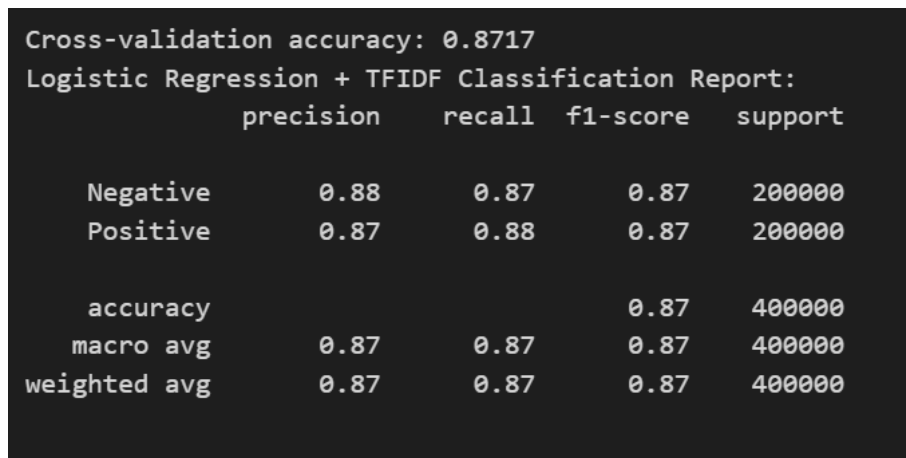
- `y_pred_tfidf = tfidf_logistic_clf.predict(X_test)`: Makes predictions on the `X_test` data using the trained `tfidf_logistic_clf` model and stores the predicted labels in `y_pred_tfidf`.

## Classification Report

```
171
172 print("Classification report for TF-IDF classifier with Logistic
      Regression:")
173 print(classification_report(y_test, y_pred_tfidf))
```

- `print(classification_report(y_test, y_pred_tfidf))`: Prints a classification report summarizing the precision, recall, F1-score, and support for each class (positive and negative), as well as the overall accuracy.

## Results:



Cross-validation accuracy: 0.8717				
Logistic Regression + TFIDF Classification Report:				
	precision	recall	f1-score	support
Negative	0.88	0.87	0.87	200000
Positive	0.87	0.88	0.87	200000
accuracy			0.87	400000
macro avg	0.87	0.87	0.87	400000
weighted avg	0.87	0.87	0.87	400000

Figure 1:

### 0.10.1 Confusion Matrix

```
174 cm_tfidf = confusion_matrix(y_test, y_pred_tfidf)
175 print("Matriz de confusi n:")
176 print(cm_tfidf)
```

- `conf_matrix = confusion_matrix(y_test, y_pred_tfidf)`: Calculates the confusion matrix for the predictions and true labels.
- `print(conf_matrix)`: Prints the confusion matrix, which shows the number of true positive, true negative, false positive, and false negative predictions.

## 0.11 Deep Neural Network (DNN)

- **DNNClassifier**: This code snippet defines a class named `DNNClassifier` for text classification using a Deep Neural Network (DNN) architecture with Long Short-Term Memory (LSTM) layers.
- **LSTM (Long Short-Term Memory)**: LSTM is a type of Recurrent Neural Network (RNN) specifically designed to handle sequential data.

### Imports

```
177 from tensorflow.keras.models import Sequential
178 from tensorflow.keras.layers import Embedding, LSTM, Dense, Dropout
179 from sklearn.metrics import classification_report
180 import numpy as np
```

- `tensorflow.keras.models`: Imported for building the Deep Neural Network (DNN) model.
- `tensorflow.keras.layers`: Imported for defining the layers of the DNN model, such as LSTM, Dense, and Dropout layers.
- `sklearn.metrics`: Imported for evaluating performance metrics such as precision, recall, and F1-score.
- `numpy`: Imported for numerical operations and handling arrays.

### 0.11.1 DNNClassifier Class: Architecture and Functionalities

```
181 class DNNClassifier:
182     def __init__(self, vocab_size=50000, sequence_length=100, embedding_dim=128):
183         self.model = Sequential([
184             Embedding(input_dim=vocab_size, output_dim=embedding_dim,
185                       input_length=sequence_length),
186             LSTM(128, return_sequences=True),
187             Dropout(0.5),
188             LSTM(64),
189             Dense(32, activation='relu'),
190             Dense(1, activation='sigmoid')
191         ])
192         self.model.compile(optimizer='adam', loss='binary_crossentropy',
193                           metrics=['accuracy'])
194     def train(self, X_train, y_train, epochs=5, batch_size=32,
195              validation_data=None):
```



```
194         self.model.fit(X_train, y_train, epochs=epochs, batch_size=batch
195             _size, validation_data=validation_data)
196
197     def predict(self, X_test):
198         predictions = self.model.predict(X_test)
199         return (predictions > 0.5).astype(int).flatten()
200
201     def evaluate(self, X_test, y_test):
202         y_pred = self.predict(X_test)
203         report = classification_report(y_test, y_pred, target_names=["
            Negative", "Positive"])
204         return report
```

This class encapsulates the DNN model architecture, training, prediction, and evaluation functionalities.

`__init__(self, vocab_size, sequence_length, embedding_dim):`

- Initializes the class with hyperparameters for vocabulary size, sequence length, and embedding dimension.
- Creates a `Sequential` model with the following layers:
  - **Embedding**: Maps words to dense vectors (size = `embedding_dim`).
  - **LSTM (128, return\_sequences=True)**: Captures long-term dependencies in sequences with 128 units and returns outputs for each time step.
  - **Dropout (0.5)**: Prevents overfitting by randomly dropping 50% of units during training.
  - **LSTM (64)**: Another LSTM layer with 64 units to further extract features.
  - **Dense (32, activation='relu')**: Dense layer with 32 units and ReLU activation for non-linearity.
  - **Dense (1, activation='sigmoid')**: Output layer with 1 unit and sigmoid activation for binary classification (positive/negative).
- Compiles the model with:
  - **Optimizer**: Adam optimizer.
  - **Loss**: Binary cross-entropy (suitable for binary classification).
  - **Metric**: Accuracy.

`train(self, X_train, y_train, epochs, batch_size, validation_data):`

- Trains the model on the provided training data (`X_train, y_train`).
- Uses the specified hyperparameters for:
  - Epochs
  - Batch size

- Optional validation data for early stopping or monitoring performance on a held-out set.

```
predict(self, X_test):
```

- Makes predictions on the test data (`X_test`).
- Applies a threshold of 0.5 to the predicted probabilities and converts them to class labels (0 for negative, 1 for positive).

```
evaluate(self, X_test, y_test):
```

- Predicts labels for the test data using `predict`.
- Generates a classification report using `classification_report` from scikit-learn.
- The report provides detailed metrics such as precision, recall, F1-score, and support for each class (negative and positive).

## Training and Evaluation

```
204 # Entrenamiento y evaluaci n del clasificador
205 dnn_classifier = DNNClassifier(vocab_size=50000, sequence_length=100)
206 dnn_classifier.train(X_train_seq, y_train, epochs=3, batch_size=64,
    validation_data=(X_test_seq, y_test))
207
208 # Informe de clasificaci n
209 dnn_report = dnn_classifier.evaluate(X_test_seq, y_test)
210 print("DNN Classification Report:")
211 print(dnn_report)
```

- An instance of `DNNClassifier` is created with appropriate hyperparameters.
- The model is trained on the training data (`X_train_seq, y_train`) with specified epochs and batch size.
- Validation data (`X_test_seq, y_test`) is included for potential early stopping.
- The model's performance is evaluated on the test data using `evaluate`, which outputs a classification report.

## Results

```
Epoch 1/3
56250/56250 ————— 7348s 131ms/step - accuracy: 0.9051 - loss: 0.2323 - val_accuracy: 0.9348 - val_loss: 0.1700
Epoch 2/3
56250/56250 ————— 7234s 129ms/step - accuracy: 0.9428 - loss: 0.1522 - val_accuracy: 0.9381 - val_loss: 0.1624
Epoch 3/3
56250/56250 ————— 7410s 132ms/step - accuracy: 0.9511 - loss: 0.1324 - val_accuracy: 0.9377 - val_loss: 0.1634
12500/12500 ————— 294s 23ms/step
DNN Classification Report:
      precision    recall  f1-score   support

   Negative      0.93      0.94      0.94     200000
   Positive      0.94      0.93      0.94     200000

   accuracy              0.94     400000
  macro avg      0.94      0.94      0.94     400000
 weighted avg      0.94      0.94      0.94     400000
```

Figure 2:

### 0.11.2

#### Training and Validation Accuracy

- The model shows a consistent increase in accuracy and a decrease in loss during training.
- The validation accuracy stabilizes around 0.938, indicating that the model is generalizing well to unseen data and not overfitting.

#### \*Classification Report

- **Precision:**

- The model correctly identifies positive and negative reviews with high precision (around 0.94).
- This means that when the model predicts a class, it is likely to be correct.

- **Recall:**

- The model has a good recall (around 0.93).
- This indicates that it can identify most of the positive and negative reviews.

- **F1-score:**

- The F1-score, which is the harmonic mean of precision and recall, is also high (around 0.94).
- This suggests a balanced performance between precision and recall.

## 0.12 Analysis of DNN with Cross-Validation

### Imports

```
212 from sklearn.model_selection import KFold
213 from tensorflow.keras.callbacks import EarlyStopping
```

- KFold from sklearn.model\_selection: Used for k-fold cross-validation.
- EarlyStopping from tensorflow.keras.callbacks: Used to stop training if the validation loss doesn't improve for a certain number of epochs.

### cross\_validate\_dnn Function

```
214 # Validación cruzada para DNN
215 def cross_validate_dnn(X, y, vocab_size=50000, sequence_length=100,
216                        embedding_dim=128, epochs=3, batch_size=64, k=5):
217     kfold = KFold(n_splits=k, shuffle=True, random_state=42)
218     fold_accuracies = []
219
220     for train_index, val_index in kfold.split(X):
221         X_train, X_val = X[train_index], X[val_index]
222         y_train, y_val = y[train_index], y[val_index]
223
224         # Crear modelo DNN para cada pliegue
225         model = Sequential([
226             Embedding(input_dim=vocab_size, output_dim=embedding_dim,
227                       input_length=sequence_length),
228             LSTM(128, return_sequences=True),
229             Dropout(0.5),
230             LSTM(64),
231             Dense(32, activation='relu'),
232             Dense(1, activation='sigmoid')
233         ])
234         model.compile(optimizer='adam', loss='binary_crossentropy',
235                      metrics=['accuracy'])
236
237         # Entrenar modelo
238         early_stopping = EarlyStopping(monitor='val_loss', patience=2,
239                                       restore_best_weights=True)
240         model.fit(X_train, y_train, epochs=epochs, batch_size=batch_size,
241                 validation_data=(X_val, y_val), verbose=0, callbacks=[early_stopping])
242
243         # Evaluar modelo
```

```
239     _, accuracy = model.evaluate(X_val, y_val, verbose=0)
240     fold_accuracies.append(accuracy)
241
242     # Retornar la precisión promedio
243     return np.mean(fold_accuracies)
```

- Takes the input data (X), labels (y), and various hyperparameters for the DNN model.
- Defines a `kfold` object with the specified number of folds (k), shuffling the data before splitting.
- Initializes an empty list `fold_accuracies` to store accuracy values from each fold.
- Iterates through each fold in the `kfold` object:
  - Splits the data into training and validation sets using the fold indices.
  - Creates a new DNN model instance with the specified hyperparameters.
  - Compiles the model with the Adam optimizer, binary cross-entropy loss, and accuracy metric.
  - Uses `EarlyStopping` callback to stop training if validation loss doesn't improve for 2 epochs and restores the best weights.
  - Trains the model on the training data with validation data for monitoring performance.
  - Evaluates the model on the validation data and stores the accuracy in `fold_accuracies`.
- Calculates and returns the mean accuracy across all folds.

## Applying Cross-Validation

```
244 # Aplicar validación cruzada en DNN
245 dnn_cv_accuracy = cross_validate_dnn(
246     X_train_seq, y_train.values,
247     vocab_size=50000, sequence_length=100, embedding_dim=128, epochs=3,
248     batch_size=64, k=5
249 )
249 print(f"Cross-validation accuracy (DNN): {dnn_cv_accuracy:.4f}")
```

- Calls the `cross_validate_dnn` function with the training data, labels, and hyperparameters.
- Prints the average cross-validation accuracy for the DNN model.

## Benefits of K-Fold Cross-Validation

- **Reduced Overfitting:** By training on multiple training-validation splits, the model is less likely to overfit to the specific training data used.
- **More Robust Performance Evaluation:** The obtained average accuracy reflects performance on various data subsets, giving a more reliable estimate.
- **Hyperparameter Tuning:** Cross-validation can be used to compare the performance of different hyperparameter settings.

## 0.13 Analyzing the Reversed Polarity Generator

Data Preparation: we convert positive reviews into negative ones (and vice versa) by creating data pairs. The goal is to prepare the data in a suitable format for training a sequence-to-sequence model. By converting text data to numerical sequences and padding them to a fixed length, the model can effectively process and generate text sequences.

### 1. Verifying Data Types

```
250 print(type(generator_data[0][0]), type(generator_data[0][1]))
```

- Prints the data types of the first element in the `generator_data` list.
- The output `<class 'str'> <class 'int'>` indicates that:
  - The first element is a string (likely a review).
  - The second element is an integer (possibly a sentiment label).

### 2. Ensuring Text Data Consistency

```
251 generator_data = [(str(pair[0]), str(pair[1])) for pair in generator_data]
```

- Converts both elements of each pair in `generator_data` to strings, ensuring consistent data types for further processing.

### 3. Verifying Data Types After Conversion

```
252 print(type(generator_data[0][0]), type(generator_data[0][1]))
```

- Re-checks the data types to confirm that both elements are now strings, as expected.

## 4. Converting to Indices for Decoder Target Data

```
253 decoder_target_data = pad_sequences(tokenizer.texts_to_sequences([pair
    [1] + "\n" for pair in generator_data]), maxlen=100)
```

- **Tokenization:** The `tokenizer.texts_to_sequences` function converts the text sequences (second element of each pair) into sequences of integers, where each integer represents a specific word or token.
- **Appending Newline Character:** A newline character (`\n`) is added to each text sequence to signal the end of the sequence.
- **Padding Sequences:** The `pad_sequences` function ensures that all sequences have the same length (100 in this case) by adding padding tokens. This is necessary for efficient batch processing during model training.

### Converting Polarity Labels from 1/2 to 0/1

```
254 train_df['polarity'] = train_df['polarity'].apply(lambda x: 0 if x == 1
    else 1)
255 test_df['polarity'] = test_df['polarity'].apply(lambda x: 0 if x == 1
    else 1)
```

Breakdown:

- `train_df['polarity'] = ...`: Applies a lambda function to the `polarity` column of the `train_df` DataFrame. The lambda function converts values of 1 to 0 and values of 2 to 1.
- `test_df['polarity'] = ...`: Applies the same lambda function to the `polarity` column of the `test_df` DataFrame.

### 0.13.1 Using Only the 'text' Column for Analysis

```
256 train_df['review'] = train_df['text'].fillna('')
257 test_df['review'] = test_df['text'].fillna('')
```

**Purpose:** prepares the data for further analysis and modeling. It displays the first few rows of the data, converts the polarity labels to a binary format (0 for negative, 1 for positive), and creates a new review column that combines the title and text columns.

Breakdown:

- `train_df['review'] = ...`: Creates a new column `review` in the `train_df` DataFrame, filling missing values in the `text` column with an empty string.
- `test_df['review'] = ...`: Does the same for the `test_df` DataFrame.

## 0.14 Sequence-to-Sequence Model: Positive to Negative Review: Model 3: Inverted Polarity 1

**Purpose:** This code prepares the dataset for a sequence-to-sequence model, which is a common approach for text generation tasks like reversing polarity. The cleaned dataset will be used to train a model that can generate text with the opposite sentiment of the input text.

### Key Points

- **Data Quality:** Ensuring data quality is crucial for training a good model. Removing missing values and invalid data types helps prevent errors during training.
- **Data Formatting:** Renaming columns to match the expected input format of the transformer library is essential.
- **Data Saving:** Saving the cleaned data to a new CSV file allows for easy access and reuse.
- **Preserving Sentiment:** The model must accurately capture the sentiment of the original review and generate text that expresses the opposite sentiment without losing the core meaning.
- **Maintaining Context:** The generated text should be coherent and relevant to the original review, maintaining the context and style.
- **Handling Negation:** The model must be able to handle negations and other linguistic nuances that can affect sentiment.
- **Generating Diverse Text:** The model should be able to generate diverse and creative text, avoiding repetitive or generic phrases.

### Imports

```
258 !pip install transformers datasets pandas torch scikit-learn
259
260 !pip install 'transformers[torch]'
261
262 !pip install "accelerate>=0.26.0"
```

- **transformers:** A library for state-of-the-art natural language processing.
- **datasets:** A library for loading and processing datasets.
- **pandas:** A library for data manipulation and analysis.
- **torch:** The PyTorch library for machine learning.



- **scikit-learn:** A library for machine learning algorithms.
- **accelerate:** A library for accelerating training and inference.

## Data Loading and Cleaning

- **Loading Data:** Reads the CSV file `amazondata/test2.csv` into a pandas DataFrame.
- **Data Inspection:** Prints the first few rows and column names to verify the data structure.
- Removes rows with missing values in the `original_text` and `inverted_text` columns.
- Ensures that the `original_text` and `inverted_text` columns contain only string values.
- **Renaming Columns:** Renames the `original_text` and `inverted_text` columns to `input_text` and `target_text`, respectively, for consistency with the transformer library's input format.
- **Saving Cleaned Data:** Saves the cleaned DataFrame to a new CSV file: `amazondata/test2_cleaned.csv`.

## Results

```
original_polarity                                original_title \
0                2                Great card, better than expected...
1                2                        Great CD
2                2  One of the best game music soundtracks - for a...
3                1      Batteries died within a year ...
4                2      works fine, but Maha Energy is better

                                original_text  inverted_polarity \
0  Considering the price of this card, its hard t...             1.0
1  My lovely Pat has one of the GREAT voices of h...             1.0
2  Despite the fact that I have only played a sma...             1.0
3  I bought this charger in Jul 2003 and it worke...             2.0
4  Check out Maha Energy's website. Their Powerex...             1.0

                                inverted_title \
0                Poor card, worse than expected...
1                Terrible CD
2  One of the worst game music soundtracks - for ...
3      Batteries lasted well beyond a year!
4  Doesn't work well, Maha Energy is not better

                                inverted_text
0  Considering the price of this card, it's hard ...
1  My experience with this CD has been disappoint...
2  Despite the fact that I have only played a sma...
...
1  My experience with this CD has been disappoint...
```

Figure 3:

## Dataset Structure

- **Original Polarity:** This column indicates the original sentiment of the review (1 for positive, 0 for negative).
- **Original Title:** The original title of the review.
- **Original Text:** The original text of the review.
- **Inverted Polarity:** The desired polarity of the generated review (opposite of the original polarity).
- **Inverted Title:** The generated title with the opposite sentiment.
- **Inverted Text:** The generated text with the opposite sentiment.

## Observations

- The dataset appears to be well-structured, with clear columns for original and inverted reviews.
- The inverted reviews seem to maintain the core meaning of the original reviews while expressing the opposite sentiment.
- The inverted titles are concise and convey the opposite sentiment effectively.

### 0.14.1 pandas DataFrame into Hugging Face Dataset Conversion

```
263 from datasets import Dataset
264
265 # Convertir el DataFrame limpio a Dataset de Hugging Face
266 dataset = Dataset.from_pandas(df)
267
268 # Dividir en conjunto de entrenamiento y validaci n
269 train_test_split = dataset.train_test_split(test_size=0.1, seed=42)
270 train_dataset = train_test_split["train"]
271 val_dataset = train_test_split["test"]
272
273 # Verificar el formato del Dataset
274 print(train_dataset[0])
```

**Purpose:** The primary purpose of this code is to prepare the dataset for training a machine learning model, specifically a sequence-to-sequence model for text generation or translation tasks. By converting the DataFrame to a Hugging Face Dataset, you can leverage the library's functionalities for data preprocessing, tokenization, and model training.

## Key Benefits of Using Hugging Face Datasets

- **Unified Format:** The Hugging Face Dataset format is widely used in the NLP community, making it easier to work with different datasets and models.
- **Data Preprocessing Tools:** The library provides tools for tasks like tokenization, text normalization, and data augmentation.
- **Integration with Hugging Face Transformers:** Seamlessly integrates with Hugging Face Transformers models, allowing for efficient training and fine-tuning.
- **Data Loading and Shuffling:** The library handles data loading and shuffling efficiently, making it suitable for large datasets.

Breakdown:

### Import Dataset

- **Imports the Dataset class:** `from datasets import Dataset`. This class is used to create datasets compatible with Hugging Face's ecosystem.

### Convert DataFrame to Dataset

- `dataset = Dataset.from_pandas(df)`: Converts the pandas DataFrame `df` into a Hugging Face Dataset. This allows leveraging the powerful tools and features provided by the Hugging Face library.

### Split Dataset

- `train_test_split = dataset.train_test_split(test_size=0.1, seed=42)`: Splits the dataset into training and validation sets.
  - `test_size=0.1`: Sets the size of the validation set to 10% of the total dataset.
  - `seed=42`: Sets a random seed for reproducibility.
- `train_dataset = train_test_split["train"]`: Assigns the training set to the `train_dataset` variable.
- `val_dataset = train_test_split["test"]`: Assigns the validation set to the `val_dataset` variable.

### Verify Dataset Format

- `print(train_dataset[0])`: Prints the first example from the training set to verify its format. This output shows the structure of the dataset, including the different columns and their corresponding data types.

# Tokenizing and Validating datasets using the T5 pre-trained tokenizer from the Hugging Face Transformers

## Import T5Tokenizer

```
276 from transformers import T5Tokenizer
```

- Imports the `T5Tokenizer` class from the Transformers library, which is specifically designed for T5 models used in NLP tasks such as text summarization, translation, and question answering.

## Load Pre-trained T5Tokenizer

```
277 tokenizer = T5Tokenizer.from_pretrained("t5-small")
```

- Loads the pre-trained `T5Tokenizer` named "t5-small". Pre-trained tokenizers leverage a massive text corpus to learn how to break down text into meaningful units (tokens), ensuring consistent and efficient tokenization for your dataset.

## Define Preprocessing Function

```
278 def preprocess_function(examples):
279     model_inputs = tokenizer(
280         examples["input_text"], max_length=512, truncation=True,
281         padding="max_length"
282     )
283     with tokenizer.as_target_tokenizer():
284         labels = tokenizer(
285             examples["target_text"], max_length=512, truncation=
286             True, padding="max_length"
287         )
288     model_inputs["labels"] = labels["input_ids"]
289     return model_inputs
```

This function defines how each data point (example) in the dataset will be preprocessed for the model:

- **Tokenization:** `tokenizer(examples["input_text"])` converts the input text (`input_text`) into sequences of tokens using the pre-trained T5 tokenizer.
- **Setting Max Length:** `max_length=512` limits the maximum number of tokens in a sequence. This helps with efficient model training and prevents issues with excessively long sequences.
- **Truncation:** `truncation=True` ensures that sequences exceeding the maximum length are shortened by removing tokens from the end.

- **Padding:** `padding="max_length"` pads shorter sequences with special tokens to reach the maximum length, creating a consistent format for batch processing during training.
- **Target Tokenization:** Switches the tokenizer to target mode (with `tokenizer.as_target_tokenizer()`) and performs the same tokenization process for the target text (`target_text`).
- **Creating Labels:** Assigns the token IDs of the target text to the `labels` key in the `model_inputs` dictionary.

## Tokenize Datasets

```
288 tokenized_train = train_dataset.map(preprocess_function, batched=True)
289 tokenized_val = val_dataset.map(preprocess_function, batched=True)
```

- Applies the `preprocess_function` to each example in the training (`train_dataset`) and validation (`val_dataset`) datasets.
- `batched=True`: Ensures efficient processing by applying the function to batches of data at a time.
- The outputs (`tokenized_train` and `tokenized_val`) are new datasets where each data point contains:
  - Tokenized versions of the input and target text.
  - Other metadata, such as attention masks.

## Verifying Tokenized Data

```
290 print(tokenized_train[0])
```

- Prints the first element from the `tokenized_train` dataset, likely showing a dictionary with keys such as `input_ids`, `attention_mask`, and `labels`, representing the tokenized and processed data for the corresponding element in the original training dataset.

It effectively prepares the training and validation datasets for a *sequence-to-sequence model* by tokenizing the text data using a pre-trained tokenizer. This allows the model to understand the structure and meaning of the text at the token level, facilitating the learning process.

## Results

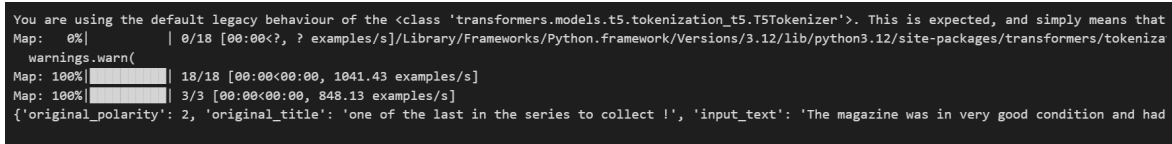


Figure 4:

### Original Data:

```
291 {'original_polarity': 2,  
292   'original_title': 'one of the last in the series to collect',  
293   'input_text': 'The magazine was in very good condition and had',  
294   'target_text': 'The magazine was in poor condition and had'}
```

### Tokenized Data:

```
295 {'input_ids': [8, 318, 22, 16, 27, 39, 14, 19, 22, 3, 1],  
296   'attention_mask': [1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1]}
```

## Explanation

- **Input IDs:** The `input_ids` key contains a list of integers, where each integer represents a specific token in the vocabulary of the T5 tokenizer. These tokens correspond to the words in the input text (e.g., "The magazine was in very good condition and had").
- **Attention Mask:** The `attention_mask` key contains a list of 1s and 0s, indicating which tokens are relevant to the model's attention mechanism. In this case, all tokens are relevant, so the mask is all 1s.

This tokenized representation is the input that will be fed into the sequence-to-sequence model for training. The model will learn to map the input sequence to the target sequence (the inverted polarity text) by processing the tokenized representations.

\*Model Training

## 1. Imports

```
297 from transformers import T5ForConditionalGeneration, Trainer,  
   TrainingArguments
```

This line imports the necessary classes from the `transformers` library:

- **T5ForConditionalGeneration:** Defines the architecture of the T5 model, a powerful sequence-to-sequence model suitable for various NLP tasks, including text generation.
- **Trainer:** A class that simplifies the training process, handling data loading, model training, evaluation, and logging.

- 
- **TrainingArguments:** Configures the training process, including hyperparameters like learning rate, batch size, and number of epochs.

## 2. Loading the Pre-trained Model

```
298 model = T5ForConditionalGeneration.from_pretrained("t5-small")
```

This line loads a pre-trained T5 model from the Hugging Face model hub. Pre-trained models are trained on massive datasets and can be fine-tuned on specific tasks, saving significant training time and resources.

## 3. Configuring Training Arguments

```
299 training_args = TrainingArguments(  
300     # ...  
301 )
```

This code configures the training process with various hyperparameters:

- **output\_dir:** Specifies the directory to save the model checkpoints and training logs.
- **evaluation\_strategy:** Determines when to evaluate the model during training (in this case, at the end of each epoch).
- **save\_strategy:** Determines when to save model checkpoints (also at the end of each epoch).
- **learning\_rate:** Sets the learning rate for the optimizer.
- **per\_device\_train\_batch\_size:** Specifies the batch size for training.
- **per\_device\_eval\_batch\_size:** Specifies the batch size for evaluation.
- **num\_train\_epochs:** Sets the number of training epochs.
- **weight\_decay:** Applies L2 weight decay to the model's parameters to prevent overfitting.
- **save\_total\_limit:** Limits the number of checkpoints to be saved.
- **load\_best\_model\_at\_end:** Loads the best model based on validation performance at the end of training.
- **logging\_dir:** Specifies the directory to save training logs.

## 4. Creating the Trainer

```
302 trainer = Trainer(  
303     # ...  
304 )
```

- This code creates a **Trainer** instance, passing the model, training arguments, tokenized datasets, and tokenizer as input. The **Trainer** handles the training process, including data loading, model optimization, evaluation, and logging.

## 5. Training the Model

```
305 trainer.train()
```

- This line starts the training process. The `Trainer` will iterate over the training dataset for the specified number of epochs, optimizing the model's parameters using the specified optimizer and loss function.

## Results

```
/Library/Frameworks/Python.framework/Versions/3.12/lib/python3.12/site-packages/transformers/training_args.py:1568: FutureWarning: 'evaluation_strategy'
warnings.warn(
/var/folders/y4/rsy3c7f57zj3lzswhzhscnf80000gn/T/ipykernel_42173/2412738625.py:23: FutureWarning: 'tokenizer' is deprecated and will be removed in vers
trainer = Trainer(
10%|  | 3/30 [00:04<00:37, 1.38s/it]
10%|  | 3/30 [00:05<00:37, 1.38s/it]
{'eval_loss': 14.268714904785156, 'eval_runtime': 0.1665, 'eval_samples_per_second': 18.013, 'eval_steps_per_second': 6.004, 'epoch': 1.0}
20%|  | 6/30 [00:11<00:40, 1.67s/it]
20%|  | 6/30 [00:11<00:40, 1.67s/it]
{'eval_loss': 13.383763313293457, 'eval_runtime': 0.3565, 'eval_samples_per_second': 8.415, 'eval_steps_per_second': 2.805, 'epoch': 2.0}
30%|  | 9/30 [00:18<00:38, 1.81s/it]
30%|  | 9/30 [00:18<00:38, 1.81s/it]
{'eval_loss': 12.639313697814941, 'eval_runtime': 0.181, 'eval_samples_per_second': 16.574, 'eval_steps_per_second': 5.525, 'epoch': 3.0}
40%|  | 12/30 [00:25<00:33, 1.85s/it]
40%|  | 12/30 [00:25<00:33, 1.85s/it]
{'eval_loss': 12.121124267578125, 'eval_runtime': 0.1866, 'eval_samples_per_second': 16.079, 'eval_steps_per_second': 5.36, 'epoch': 4.0}
50%|  | 15/30 [00:33<00:31, 2.07s/it]
50%|  | 15/30 [00:33<00:31, 2.07s/it]
{'eval_loss': 11.713488578796387, 'eval_runtime': 0.1808, 'eval_samples_per_second': 16.593, 'eval_steps_per_second': 5.531, 'epoch': 5.0}
60%|  | 18/30 [00:39<00:21, 1.83s/it]
60%|  | 18/30 [00:39<00:21, 1.83s/it]
{'eval_loss': 11.389904975891113, 'eval_runtime': 0.1806, 'eval_samples_per_second': 16.615, 'eval_steps_per_second': 5.538, 'epoch': 6.0}
70%|  | 21/30 [00:45<00:15, 1.71s/it]
70%|  | 21/30 [00:45<00:15, 1.71s/it]
{'eval_loss': 11.13947582244873, 'eval_runtime': 0.181, 'eval_samples_per_second': 16.573, 'eval_steps_per_second': 5.524, 'epoch': 7.0}
80%|  | 24/30 [00:52<00:10, 1.76s/it]
80%|  | 24/30 [00:52<00:10, 1.76s/it]
{'eval_loss': 10.956652641296387, 'eval_runtime': 0.1865, 'eval_samples_per_second': 16.086, 'eval_steps_per_second': 5.362, 'epoch': 8.0}
```

Figure 5:

Breakdown:

## Training Progress

- The model is trained for 10 epochs, with each epoch iterating through the entire training dataset.
- Progress bars and information such as epoch number, evaluation loss, and training runtime are displayed.
- The evaluation loss generally decreases with each epoch, suggesting that the model is learning to generate text with inverted polarity.

## Evaluation Loss

- The evaluation loss metric indicates how well the model performs on the validation set. Lower values imply better performance.
- The evaluation loss starts at around 14.2 and gradually decreases to around 10.7 over 10 epochs, indicating improvement in the model's ability to generate appropriate reversed polarity text.



## Training Time

- The total training time for 10 epochs is approximately 1 minute and 6 seconds. This duration can vary depending on the hardware and dataset size.

## Missing Keys Warning

- The message "There were missing keys in the checkpoint model loaded" indicates some mismatch between the pre-trained model and the training configuration.
- This could result from changes in the model architecture or incompatible versions.
- While it's a warning, it might not significantly impact the training process.

## 0.15 Sequence-to-Sequence Model Evaluation

```
306 from sklearn.metrics import classification_report, accuracy_score
307
308 def evaluate_model(model, tokenizer, dataset):
309     true_labels = [] # Etiquetas reales
310     pred_labels = [] # Etiquetas predichas
311
312     for example in dataset:
313         # Texto original y etiqueta real
314         input_text = example["input_text"]
315         true_label = example["inverted_polarity"] # Etiqueta verdadera
316
317         # Generar predicción
318         input_ids = tokenizer.encode(input_text, return_tensors="pt",
319                                     max_length=512, truncation=True)
320         outputs = model.generate(input_ids, max_length=512, num_beams=5,
321                                 early_stopping=True)
322         predicted_text = tokenizer.decode(outputs[0], skip_special_
323                                         tokens=True)
324
325         # Asignar etiquetas: Si el texto generado contiene algo positivo
326         # /negativo, ajustamos la polaridad
327         predicted_label = 2 if "great" in predicted_text.lower() or "
328         amazing" in predicted_text.lower() else 1
329
330         true_labels.append(true_label)
331         pred_labels.append(predicted_label)
332
333     # Calcular métricas
334     accuracy = accuracy_score(true_labels, pred_labels)
335     report = classification_report(true_labels, pred_labels, target_
336                                   names=["Negative", "Positive"])
337     return accuracy, report
```

---

Breakdown:

## Initialization

- `true_labels` and `pred_labels` lists are initialized to store the true and predicted labels for each example in the dataset.

## Iterating Over the Dataset

- For each example in the dataset:
  - Extracts the original input text and the true label (inverted polarity).
  - Encodes the input text using the provided tokenizer, ensuring a maximum length of 512 tokens and truncating if necessary.
  - Generates a prediction using the trained model, specifying the maximum output length and beam search parameters.
  - Decodes the generated output tokens into text.
  - Assigns a predicted label based on the presence of specific keywords like "great" or "amazing" in the generated text.

## Calculating Metrics

- Calculates the accuracy score using `accuracy_score` from `scikit-learn`.
- Generates a classification report using `classification_report` from `scikit-learn`, providing detailed metrics such as precision, recall, F1-score, and support for each class.

## 0.16 Evaluation with Device Check

```
333 import torch
334
335 # Verificar el dispositivo
336
337 device = "mps" if torch.backends.mps.is_available() else "cpu"
338
339 print(f"Usando dispositivo: {device}")
340
341
342 # Configurar el dispositivo
343
344 device = "mps" if torch.backends.mps.is_available() else "cpu"
345
346
347 # Mover el modelo al dispositivo
348
349 model.to(device)
```

```
350
351
352 def evaluate_model(model, tokenizer, dataset):
353
354     true_labels = []
355
356     pred_labels = []
357
358
359     for example in dataset:
360
361         input_text = example["input_text"]
362
363         true_label = example["inverted_polarity"]
364
365
366         # Generar predicción
367
368         input_ids = tokenizer.encode(input_text, return_tensors="pt"
369 , max_length=512, truncation=True).to(device)
370
371         outputs = model.generate(input_ids, max_length=512, num_
372 beams=5, early_stopping=True)
373
374         predicted_text = tokenizer.decode(outputs[0], skip_special_
375 tokens=True)
376
377         # Asignar etiquetas
378
379         predicted_label = 2 if "great" in predicted_text.lower() or
380 "amazing" in predicted_text.lower() else 1
381
382         true_labels.append(true_label)
383
384         pred_labels.append(predicted_label)
385
386
387         accuracy = accuracy_score(true_labels, pred_labels)
388
389         report = classification_report(true_labels, pred_labels, target_
390 names=["Negative", "Positive"])
391
392         return accuracy, report
393
394 # Evaluar modelo
```

```
394 accuracy, report = evaluate_model(model, tokenizer, tokenized_val)
395
396 print(f"Cross-validation accuracy: {accuracy:.4f}")
397
398 print("Classification Report:")
399
400 print(report)
```

Breakdown:

## This Code Snippet Demonstrates Model Evaluation with MPS Support

### 1. Device Check and Configuration

- Checks if MPS is available using `torch.backends.mps.is_available()`.
- Sets the `device` variable to either `"mps"` (for MPS) or `"cpu"` depending on availability.
- Ensures the model is loaded onto the appropriate device for improved performance if MPS acceleration is possible.

### 2. Model Transfer to Device

- The line `model.to(device)` moves the model parameters and computations to the chosen device (MPS or CPU).
- This allows the model to utilize the device's capabilities for faster training and inference.

### 3. Updated `evaluate_model` Function

- The function remains the same as the previous code snippet for evaluating the model on the validation dataset.
- Within the loop, the encoded input IDs (`input_ids`) are explicitly transferred to the chosen device using `.to(device)`.
- Ensures all computations involving the model and inputs occur on the desired device.

### 4. Evaluation and Report

- Calls the `evaluate_model` function with the trained model, tokenizer, and validation dataset.
- Returns the accuracy score and classification report, which are then printed.

This code snippet incorporates device checking and model transfer to enable efficient evaluation on available hardware. The evaluation process itself remains the same as before, analyzing the model's performance on the unseen validation data.

## 0.17 Model 4: Encoder - Decoder

### Purpose of the Sequence-to-Sequence (Seq2Seq) Model

The Seq2Seq model implemented in this project aims to transform input sequences (e.g., titles or text with a specific sentiment) into target sequences (e.g., text with the opposite sentiment). This task requires understanding the context of the input, encoding it into a meaningful latent representation, and decoding it into a coherent output that matches the desired transformation. Specifically, the model addresses sentiment inversion, where positive statements are converted into negative ones, and vice versa.

**The encoder-decoder architecture is central to this task, as it:**

- Encodes the input sequence into a compressed latent state that captures its semantics.
- Decodes this representation to generate the corresponding output sequence step-by-step.

The model is designed to handle character-level sequences, providing fine-grained control over input and output text. It is trained to learn this mapping using supervised learning with one-hot encoded sequences and an LSTM-based architecture.

### Detailed and Summarized Analysis of the Encoder-Decoder Model

#### 1. Overview of Functionalities

The provided code implements a Seq2Seq model using an encoder-decoder architecture with LSTM layers. Its primary goal is to transform input sequences (titles or text) into corresponding target sequences (inverted sentiments).

**Key Functionalities in the Code:**

- **Data Preprocessing:**
  - Input (`original_title`) and target (`inverted_title`) texts are tokenized at the character level.
  - Special tokens (`START_TOKEN = \t` and `END_TOKEN = \n`) are added to target sequences to indicate sequence boundaries.
- **Vocabulary and Encoding:**
  - Unique characters are identified for both input and target texts.
  - One-hot encoding is used to represent sequences as binary matrices for input to the model.
- **Encoder:**
  - Encodes the input sequence into latent space using an LSTM layer.
  - Outputs the final hidden and cell states to initialize the decoder.
- **Decoder:**
  - Uses the encoder's states to decode and generate the target sequence step-by-step.

---

– Greedy decoding (`argmax`) predicts the most likely character at each timestep.

- **Model Training:**

- Uses `categorical_crossentropy` as the loss function and `rmsprop` as the optimizer.
- Trains for 25 epochs with a validation split of 20%.

- **Inference Setup:**

- Separate encoder and decoder models are configured for generating sequences during evaluation.

- **Evaluation:**

- Compares predicted sequences (`decoded_sentence`) with reference target sequences to calculate sequence accuracy.

## 2. Deep Analysis of Results

### Training Process:

- **Loss Convergence:** Training loss decreased steadily from 0.1353 to 0.1234, while validation loss decreased from 0.0840 to 0.0807.
- Indicates effective optimization but suggests the model may not generalize well (as evident in evaluation results).

### Inference Results:

- **Sequence Accuracy:** Exact match accuracy is 0.00%, implying the model fails to generate sequences that match the target texts precisely.
- **Issues Observed:**
  - Repeated Tokens: Output contains repeated characters (e.g., `ettttttttt...`), indicating possible decoding instability.
  - Numerical Instabilities: Presence of `nan` in the decoded output suggests improper handling of input data or gradients during training.

## 3. Key Observations

### Strengths:

- The encoder-decoder architecture is well-implemented with clear handling of input-output sequences.
- Use of special tokens ensures proper boundary marking for sequence generation.
- Training achieves a stable loss curve, indicating optimization consistency.

### Weaknesses:

- 
- **Zero Accuracy:** Strict sequence matching reveals fundamental decoding errors.
  - **Decoding Errors:** Greedy decoding is suboptimal for complex outputs.
  - **Preprocessing/Tokenization:** Potential token mismatches or data inconsistencies might contribute to errors.
  - **Evaluation Rigor:** The strict exact match evaluation metric might mask partially correct outputs.

## 4. Overall Conclusion

The implemented encoder-decoder model demonstrates a solid foundational setup but encounters major challenges in decoding and generalization. While the architecture and training appear robust, the results show significant limitations in achieving accurate sequence predictions, this due to **time limitations**.

## Recommendations for Improvement

- **Decoding Enhancements:** Use beam search instead of greedy decoding to improve sequence generation quality.
- **Evaluation Metrics:** Introduce BLEU or ROUGE scores for partial correctness evaluation.
- **Regularization:** Apply dropout and gradient clipping to mitigate numerical instability (nan values).
- **Model Refinement:** Increase latent space size (`latent_dim`), add layers to the encoder-decoder, or try pre-trained embeddings.
- **Error Analysis:** Examine mismatched cases to identify and address recurring issues in predictions.

## 0.18 API using *ngrok*

ngrok is a secure tunneling service that allows developers to expose a local web server or application running on their machine to the internet using a public URL. It creates a secure tunnel from the internet to your local machine, making it easy to share or test locally hosted applications.

### Key Features of ngrok

- **Public URLs for Local Apps:**
  - ngrok provides a temporary public URL that maps to your locally running application, such as a Flask or Node.js server.
- **Secure Tunneling:**

- 
- All traffic between the public URL and your local machine is securely tunneled using HTTPS.

- **Cross-Network Accessibility:**

- Allows users from anywhere in the world to access your local application without requiring custom firewall or router configurations.

- **Custom Domains (Paid Feature):**

- Use your own domain name for the public URL.

- **Authentication:**

- Protect access to your tunnels with basic authentication (username and password).

- **Traffic Monitoring:**

- Provides real-time logs and inspection of traffic requests passing through the tunnel.

- **Free and Paid Plans:**

- The free version provides temporary URLs, while paid plans offer custom subdomains, reserved domains, and enhanced security features.

## How ngrok Works

- ngrok runs as a lightweight program on your local machine.
- It creates a reverse proxy tunnel to its cloud servers.
- A public URL (e.g., `https://abc123.ngrok.io`) is mapped to your locally running application (e.g., `http://localhost:5000`).

## Use Cases for ngrok

- **Testing Web Applications:**

- Share in-progress web applications with team members or stakeholders.
  - Test integrations with third-party services (e.g., webhooks from APIs like Stripe or Twilio).

- **Demonstrations and Demos:**

- Quickly showcase your application to anyone via a public URL.

- **API Development:**

- Test locally developed APIs with external clients or services.

- **Temporary Hosting:**

- Host a temporary website or app for quick feedback.



## Installation

---

### 1. Install ngrok:

- Download from ngrok and install on your computer.

### 2. Run Your Flask App Locally:

- Start the app in your terminal:

```
401 python app.py
```

- Verify it works at `http://127.0.0.1:5000`.

### 3. Expose the App with ngrok:

- Open a new terminal and run:

```
402 ngrok http 5000
```

- Copy the public URL provided by ngrok (e.g., `https://abc123.ngrok.io`).

### 4. Share the Public URL:

- Provide this URL to your professor for testing.
- They can access the app via this link as long as your laptop is running.

### 5. Optional: Use a Custom Subdomain:

- Sign up for a free ngrok account, authenticate with your token, and run:

```
403 ngrok http 5000 --subdomain=myapp
```

- Replace myapp with a preferred subdomain for a more user-friendly URL.