Natural Language Processing

HomeWork 3

Ilia Hashemi Rad

99102456

AmirMohammad Fakhimi

99170531

AmirMahdi Namjoo

402211467

Introduction

Explanations:

In this project, we have implemented a classification problem on sentiment analysis both on a document and text level on the Taaghche comments dataset. Taaghche is an Iranian online ebook marketplace featuring user comments on each book. Based on a dataset of scores and comments from users and other users' likes in the comments, we implemented a classifier for sentiment analysis of the text. We use both base models like LSTM or SVM and also Transfomer-based models.

In addition, we also wrote a crawler Taaghche website to crawl all the book pages and get the book information like name and author names so that we can have a list of NER for them.

Crawlers

Explanations:

In the first part, we implemented a crawler on the Taaghche website to get all the book info. The crawler downloads the pages of all books on the website by iterating on the "id" in the URL and saving the result as HTML. We then feed the results into an extractor module written using BeautifulSoup to extract the main parts of book info, including name, publication, author, translator, etc., and save them into CSV files for further use.

`crawler.py`

```python
base_url = "https://taaghche.com/book/"
def save_page(book_id, thread_exceptions):
    url = f"{base_url}{book_id}/"
    try:
        response = requests.get(url)

        if response.status_code == 404:
            print(book_id," : 404")
            return

        with open(os.path.join(output_dir, f"{book_id}.html"), 'w', encoding='utf-8') as f:
            print(f'Saving book with id: {book_id}')
            f.write(response.text)
    except Exception as e:
        print(e)
        thread_exceptions.append(book_id)
```

Explanations:

This is the main part of the crawler that sends requests to Taagche to get the book pages and save them into files. We also used Python threading features to make the whole process faster.

`extactor.py`

```python
    def extract_data_from_html(file_path):
    with open(file_path, 'r', encoding='utf-8') as file:
        soup = BeautifulSoup(file, 'html.parser')
        script_tag = soup.find('script', type='application/ld+json')
        if script_tag:
            try:
                json_data = json.loads(script_tag.string)
                book_name = json_data.get('name', '')
                authors = ' $ '.join([author['name'] for author in json_data.get('author', [])])
```

```
                translators = ' $ '.join(
                    [translator['name'] for translator in json_data.get('workExample', {}).get('translator', [])])
                publisher = json_data.get('workExample', {}).get('publisher', {}).get('name', '')
                data.append({
                    'name': book_name,
                    'author': authors,
                    'translator': translators,
                    'publisher': publisher
                })
            except json.JSONDecodeError:
                pass

    for x in os.listdir(input_dir):
    file_name = x
    file_path = os.path.join(input_dir, file_name)
    if os.path.isfile(file_path) and file_path.endswith('.html'):
        extract_data_from_html(file_path)
    if len(data) >= 10000:
        df = pd.DataFrame(data)
        output_file = os.path.join(output_dir, f'books_data_part_{part_number}.csv')
        df.to_csv(output_file, index=False, encoding='utf-8')
        data = []
        part_number += 1
        print("index put into files: ", file_name)

if data:
    df = pd.DataFrame(data)
    output_file = os.path.join(output_dir, f'books_data_part_{part_number}.csv')
    df.to_csv(output_file, index=False, encoding='utf-8')
    data = []
    part_number += 1
    print("index put into files: ", file_name)
    data = []
```

Explanations:

The main part of extracor.py is `extract_data_from_html` function. This function uses BS4 to find the JSON section that includes book data in the HTML and save data json data into a python dictionary. This data is then fed into a Pandas Dataframe and we save it into a csv file.

Document Classifier

The base model is in `DocClassifier_Base.ipynb`. We investigate each segment in the following sections.

Loading and Preparing Data

```
# Import the pandas library for data manipulation
import pandas as pd

# Load the CSV file into a pandas DataFrame
# The file 'taghche.csv' is located in the 'datasets/' directory
data = pd.read_csv('datasets/taghche.csv')

# Remove any duplicate rows in the DataFrame
data = data.drop_duplicates()

# Drop rows where the 'comment' or 'rate' columns have missing values (NaN)
data.dropna(subset=['comment', 'rate'], inplace=True)

# Print the first 5 rows of the cleaned DataFrame
print(data.head())
```

```python
def label_sentiment(rate, positive_threshold, neutral_threshold):
"""
Labels sentiment based on rating thresholds.

Args:
- rate (int or float): The numerical rating to evaluate.
- positive_threshold (int or float): The minimum rating value that qualifies as 'positive'.
- neutral_threshold (int or float): The minimum rating value that qualifies as 'neutral'; ratings below this are considered 'negative'.

Returns:
- str: The sentiment label ('positive', 'neutral', or 'negative') based on the rating.
"""
# Check if the rating is greater than or equal to the positive threshold
if rate >= positive_threshold:
return 'positive'
# If the rating is not 'positive', check if it is greater than or equal to the neutral threshold
elif rate >= neutral_threshold:
return 'neutral'
# If the rating is neither 'positive' nor 'neutral', label the sentiment as 'negative'
else:
return 'negative'
```

Explanations:

At first, we load data using Pandas Library. We then define a fucntion to label the sentiment of each comment based on its rating and threshold. note that we have two thresholds, one for positive, and one for neutral comments. everything below neutral is considered negative.

Balancing Data

```python
    # Function to prepare data and labels based on given thresholds
def prepare_data(positive_threshold, neutral_threshold):
    """
    Prepares the data and labels based on given thresholds for sentiment classification.

    Args:
    - positive_threshold (int or float): The minimum rating value that qualifies as 'positive'.
    - neutral_threshold (int or float): The minimum rating value that qualifies as 'neutral';
        ratings below this are considered 'negative'.

    Returns:
    - tuple: A tuple containing:
        - pandas.Series: The comments from the balanced dataset.
        - pandas.Series: The corresponding sentiment labels from the balanced dataset.
    """
    # Create a copy of the original data to avoid modifying it
    labeled_data = data.copy()

    # Apply the label_sentiment function to the 'rate' column to create a new 'sentiment' column
    labeled_data['sentiment'] = labeled_data['rate'].apply(lambda x: label_sentiment(x,
        positive_threshold, neutral_threshold))

    # Combine the 'comment' and 'sentiment' columns into a single DataFrame
    df = pd.concat([labeled_data['comment'], labeled_data['sentiment']], axis=1)

    # Separate the DataFrame into three classes based on sentiment
    positive = df[df['sentiment'] == 'positive']
    neutral = df[df['sentiment'] == 'neutral']
    negative = df[df['sentiment'] == 'negative']

    # Determine the size of the smallest class to balance the dataset
    min_class_size = min(len(positive), len(neutral), len(negative))

    # Downsample each class to the size of the smallest class to ensure balance
```

```python
    positive_downsampled = resample(positive, replace=False, n_samples=min_class_size, random_state=42)
    neutral_downsampled = resample(neutral, replace=False, n_samples=min_class_size, random_state=42)
    negative_downsampled = resample(negative, replace=False, n_samples=min_class_size, random_state=42)

    # Combine the downsampled classes into a single DataFrame
    df_balanced = pd.concat([positive_downsampled, neutral_downsampled, negative_downsampled])

    # Shuffle the balanced DataFrame to mix the rows
    df_balanced = df_balanced.sample(frac=1, random_state=42).reset_index(drop=True)

    # Return the 'comment' and 'sentiment' columns as separate pandas Series
    return df_balanced['comment'], df_balanced['sentiment']
```

Explanations:

- The function starts by creating a copy of the original dataset to avoid modifying it:

```python
labeled_data = data.copy()
```

- It applies the `label_sentiment` function to the `rate` column to create a new `sentiment` column:

```python
labeled_data['sentiment'] = labeled_data['rate'].apply(lambda x: label_sentiment(x, positive_threshold, neutral_threshold))
```

- The function then combines the `comment` and `sentiment` columns into a single DataFrame:

```python
df = pd.concat([labeled_data['comment'], labeled_data['sentiment']], axis=1)
```

- It separates the DataFrame into three classes based on sentiment:

```python
positive = df[df['sentiment'] == 'positive']
neutral = df[df['sentiment'] == 'neutral']
negative = df[df['sentiment'] == 'negative']
```

- The function determines the size of the smallest class to balance the dataset:

```python
min_class_size = min(len(positive), len(neutral), len(negative))
```

- It down-samples each class to the size of the smallest class to ensure balance:

```python
positive_downsampled = resample(positive, replace=False, n_samples=min_class_size, random_state=42)
neutral_downsampled = resample(neutral, replace=False, n_samples=min_class_size, random_state=42)
negative_downsampled = resample(negative, replace=False, n_samples=min_class_size, random_state=42)
```

- The function combines the down-sampled classes into a single DataFrame:

```python
df_balanced = pd.concat([positive_downsampled, neutral_downsampled, negative_downsampled])
```

- It shuffles the balanced DataFrame to mix the rows:

```python
df_balanced = df_balanced.sample(frac=1, random_state=42).reset_index(drop=True)
```

- Finally, the function returns the `comment` and `sentiment` columns as separate pandas Series:

```python
return df_balanced['comment'], df_balanced['sentiment']
```

Preprocess

```python
    # Function to preprocess and normalize the text
def preprocess(text):
    """
    Preprocesses and normalizes text data by removing special characters,
    non-Persian characters, digits, and multiple spaces.

    Args:
    - text (str): Input text to be processed.

    Returns:
    - str: Processed text with normalized format.
    """
    # Replace one or more newline characters with a single newline
    pattern = re.compile(r"\n+")
    text = pattern.sub("\n", text)

    # Replace '\n' and '\n' with a single space
    text = re.sub(r'\\n|\n', ' ', text)

    # Remove non-Persian characters and digits
    text = re.sub(r'[^-\s]', ' ', text)

    # Replace one or more spaces with a single space
    pattern = re.compile(r" +")
    text = pattern.sub(" ", text)

    return text

# Apply the preprocess function to the 'comment' column in the DataFrame data
data['comment'] = data['comment'].apply(preprocess)
# Remove any duplicate rows in the DataFrame
data = data.drop_duplicates()

# Drop rows where the 'comment' or 'rate' columns have missing values (NaN)
data.dropna(subset=['comment', 'rate'], inplace=True)
```

Explanations:

- The function starts by replacing one or more newline characters with a single newline:

```python
        pattern = re.compile(r"\n+")
        text = pattern.sub("\n", text)
```

- Next, it replaces occurrences of the newline character (`\n`) and escaped newline (`\\n`) with a single space:

```python
        text = re.sub(r'\\n|\n', ' ', text)
```

- The function then removes all non-Persian characters and digits. This is done using a regular expression that matches any character not in the Persian alphabet (-) or whitespace:

```python
        text = re.sub(r'[^-\s]', ' ', text)
```

- Finally, it replaces one or more spaces with a single space to normalize the spacing in the text:

```python
        pattern = re.compile(r" +")
        text = pattern.sub(" ", text)
```

- The processed text is then returned by the function.

The following lines apply the `preprocess` function to the `comment` column of the DataFrame `data`:

```python
    # Apply the preprocess function to the 'comment' column in the DataFrame data
    data['comment'] = data['comment'].apply(preprocess)

    # Remove any duplicate rows in the DataFrame
    data = data.drop_duplicates()

    # Drop rows where the 'comment' or 'rate' columns have missing values (NaN)
    data.dropna(subset=['comment', 'rate'], inplace=True)
```

- The `preprocess` function is applied to each entry in the `comment` column to clean and normalize the text.
- After preprocessing, any duplicate rows in the DataFrame are removed using:

```python
        data = data.drop_duplicates()
```

- Finally, rows where the `comment` or `rate` columns have missing values (NaN) are dropped:

```python
        data.dropna(subset=['comment', 'rate'], inplace=True)
```

TF IDF - Logistic Regression

Next, we implement a TF-IDF vectorizer and use logistic regression for the task. Based on our study and the papers we investigated, logistic regression is the best base model for this task.

```python
    # Create a pipeline with TF-IDF and logistic regression
logReg_PL = Pipeline([
    ("tfidf", TfidfVectorizer()),
    ("logreg", LogisticRegression(max_iter=500, solver='newton-cg'))
])

# Define the parameter grid for GridSearchCV
param_grid = {
    'tfidf__ngram_range': [(1, 1), (1, 2), (1, 3)],
    'tfidf__max_features': [5000, 10000],
    'logreg__C': [0.01, 0.1, 1, 10]
}

# Custom GridSearchCV implementation to iterate over parameter grid
best_score = 0
best_params = None

# Thresholds to evaluate
rate_thresholds = [(1, 2), (1, 3), (1, 4), (2, 3), (2, 4), (3, 4)]

# Iterate over each pair of thresholds and perform GridSearchCV
for neutral_threshold, positive_threshold in tqdm(rate_thresholds):
    # Prepare the data using the specified thresholds
    X_prepared, y_prepared = prepare_data(positive_threshold, neutral_threshold)

    # Split the data into training and testing sets
    X_train, X_test, y_train, y_test = train_test_split(X_prepared, y_prepared, test_size=0.1,
        random_state=42)

    # Initialize GridSearchCV with the pipeline and parameter grid
    grid_search = GridSearchCV(logReg_PL, param_grid, cv=5, scoring='accuracy')

    # Fit GridSearchCV on the training data
    grid_search.fit(X_train, y_train)

    # Get the best score and parameters from GridSearchCV
    score = grid_search.best_score_

    # Update the best score and best parameters if the current score is better
```

```python
    if score > best_score:
        best_score = score
        best_params = grid_search.best_params_
        best_params['positive_threshold'] = positive_threshold
        best_params['neutral_threshold'] = neutral_threshold

# Print the best parameters found by GridSearchCV
print("Best parameters for TF-IDF model are:", best_params)

# Prepare the data using the best parameters found from GridSearchCV
X_prepared, y_prepared = prepare_data(best_params['positive_threshold'], best_params['
    neutral_threshold'])

# Split the prepared data into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(X_prepared, y_prepared, test_size=0.1,
    random_state=42)

# Create a pipeline for the best Logistic Regression model with the best parameters
best_logReg_model = Pipeline([
    ("tfidf", TfidfVectorizer(ngram_range=best_params['tfidf__ngram_range'], max_features=
        best_params['tfidf__max_features'])),
    ("logreg", LogisticRegression(C=best_params['logreg__C'], max_iter=500, solver='newton-cg'))
])

# Fit the best Logistic Regression model on the training data
best_logReg_model.fit(X_train, y_train)

# Predict the labels on the test set using the best model
y_test_pred = best_logReg_model.predict(X_test)

# Calculate the accuracy score of the best model on the test set
test_accuracy = accuracy_score(y_test, y_test_pred)

# Print the test accuracy score of the best Logistic Regression model
print("Test accuracy of Logistic Regression model:", test_accuracy)
```

Explanations:

```python
        logReg_PL = Pipeline([
            ("tfidf", TfidfVectorizer()),
            ("logreg", LogisticRegression(max_iter=500, solver='newton-cg'))
        ])
```

- The pipeline `logReg_PL` is created with two steps:
  1. `TfidfVectorizer()`: Converts text data into TF-IDF features.
  2. `LogisticRegression()`: Applies logistic regression for classification, with a maximum of 500 iterations and the 'newton-cg' solver.

```python
        # Define the parameter grid for GridSearchCV
        param_grid = {
            'tfidf__ngram_range': [(1, 1), (1, 2), (1, 3)],
            'tfidf__max_features': [5000, 10000],
            'logreg__C': [0.01, 0.1, 1, 10]
        }
```

- The `param_grid` defines the hyperparameters for GridSearchCV to search over:
  - `tfidf__ngram_range`: N-gram ranges (unigrams, bigrams, trigrams).
  - `tfidf__max_features`: Maximum number of features (5000 or 10000).
  - `logreg__C`: Inverse of regularization strength (0.01, 0.1, 1, 10).

```python
        # Custom GridSearchCV implementation to iterate over parameter grid
        best_score = 0
        best_params = None

        # Thresholds to evaluate
        rate_thresholds = [(1, 2), (1, 3), (1, 4), (2, 3), (2, 4), (3, 4)]
```

- `best_score` and `best_params` are initialized to store the best score and corresponding parameters.
- `rate_thresholds` contains pairs of thresholds to evaluate for neutral and positive sentiment classification.

```python
        # Iterate over each pair of thresholds and perform GridSearchCV
        for neutral_threshold, positive_threshold in tqdm(rate_thresholds):
            # Prepare the data using the specified thresholds
            X_prepared, y_prepared = prepare_data(positive_threshold, neutral_threshold)

            # Split the data into training and testing sets
            X_train, X_test, y_train, y_test = train_test_split(X_prepared, y_prepared,
                test_size=0.1, random_state=42)

            # Initialize GridSearchCV with the pipeline and parameter grid
            grid_search = GridSearchCV(logReg_PL, param_grid, cv=5, scoring='accuracy')

            # Fit GridSearchCV on the training data
            grid_search.fit(X_train, y_train)

            # Get the best score and parameters from GridSearchCV
            score = grid_search.best_score_

            # Update the best score and best parameters if the current score is better
            if score > best_score:
                best_score = score
                best_params = grid_search.best_params_
                best_params['positive_threshold'] = positive_threshold
                best_params['neutral_threshold'] = neutral_threshold

        # Print the best parameters found by GridSearchCV
        print("Best parameters for TF-IDF model are:", best_params)
```

- The code iterates over each pair of thresholds in `rate_thresholds`.
- For each pair:
  1. `prepare_data` is called to prepare the dataset with the current thresholds.
  2. The data is split into training and testing sets using `train_test_split`.
  3. `GridSearchCV` is initialized with the pipeline and parameter grid, and fitted to the training data.
  4. The best score and parameters are retrieved from the grid search results.
  5. If the current score is better than the best score, update `best_score` and `best_params`.

```python
        # Prepare the data using the best parameters found from GridSearchCV
        X_prepared, y_prepared = prepare_data(best_params['positive_threshold'], best_params['
            neutral_threshold'])

        # Split the prepared data into training and testing sets
        X_train, X_test, y_train, y_test = train_test_split(X_prepared, y_prepared, test_size
            =0.1, random_state=42)

        # Create a pipeline for the best Logistic Regression model with the best parameters
        best_logReg_model = Pipeline([
            ("tfidf", TfidfVectorizer(ngram_range=best_params['tfidf__ngram_range'],
                max_features=best_params['tfidf__max_features'])),
```

```python
            ("logreg", LogisticRegression(C=best_params['logreg__C'], max_iter=500, solver='
                newton-cg'))
        ])

        # Fit the best Logistic Regression model on the training data
        best_logReg_model.fit(X_train, y_train)

        # Predict the labels on the test set using the best model
        y_test_pred = best_logReg_model.predict(X_test)

        # Calculate the accuracy score of the best model on the test set
        test_accuracy = accuracy_score(y_test, y_test_pred)

        # Print the test accuracy score of the best Logistic Regression model
        print("Test accuracy of Logistic Regression model:", test_accuracy)
```

- The data is prepared using the best parameters found by GridSearchCV.
- The prepared data is split into training and testing sets.
- A pipeline is created for the best logistic regression model with the best parameters.
- The model is fitted to the training data.
- Predictions are made on the test set.
- The accuracy of the model is calculated on the test set.
- The test accuracy is printed.

Evaluation Metrics

```python
    def evaluate_model(y_true, y_pred, class_names):
    """
    Evaluates the performance of a classification model using various metrics and visualizations.

    Args:
    - y_true (array-like): True labels of the data.
    - y_pred (array-like): Predicted labels of the data.
    - class_names (list): List of class names in the same order as the confusion matrix.

    Returns:
    - pd.DataFrame: DataFrame containing the classification report.
    """
    # Generate and print the classification report
    report = classification_report(y_true, y_pred, target_names=class_names, output_dict=True)
    report_df = pd.DataFrame(report).transpose()
    print("Classification Report:\n", report_df)

    # Generate and display the confusion matrix as a heatmap
    cm = confusion_matrix(y_true, y_pred)
    plt.figure(figsize=(10, 7))
    sns.heatmap(cm, annot=True, fmt='d', cmap='Blues', xticklabels=class_names, yticklabels=
        class_names)
    plt.xlabel('Predicted')
    plt.ylabel('True')
    plt.title('Confusion Matrix')
    plt.show()

    # Calculate and print overall metrics: accuracy, precision, recall, and F1 score
    accuracy = accuracy_score(y_true, y_pred)
    precision = precision_score(y_true, y_pred, average='weighted')
    recall = recall_score(y_true, y_pred, average='weighted')
    f1 = f1_score(y_true, y_pred, average='weighted')
```

```python
    metrics = {
        "Accuracy": accuracy,
        "Precision": precision,
        "Recall": recall,
        "F1 Score": f1
    }

    print("\nOverall Metrics:")
    for metric, value in metrics.items():
        print(f"{metric}: {value:.4f}")

    return report_df

# Evaluate the model
report_df = evaluate_model(y_test, y_test_pred, ["Negative", "Neutral", "Positive"])
```

Explanations:

The `evaluate_model` function evaluates the performance of a classification model using various metrics and visualizations.

- The function takes three arguments:
  - `y_true`: The true labels of the data.
  - `y_pred`: The predicted labels of the data.
  - `class_names`: A list of class names in the same order as the confusion matrix.
- The function returns a pandas DataFrame containing the classification report.

Classification Report

- The classification report is generated using `classification_report` from scikit-learn and printed:

```python
    report = classification_report(y_true, y_pred, target_names=class_names, output_dict=True)
    report_df = pd.DataFrame(report).transpose()
    print("Classification Report:\n", report_df)
```

Confusion Matrix

- The confusion matrix is generated and displayed as a heatmap using seaborn:

```python
    cm = confusion_matrix(y_true, y_pred)
    plt.figure(figsize=(10, 7))
    sns.heatmap(cm, annot=True, fmt='d', cmap='Blues', xticklabels=class_names, yticklabels=class_names)
    plt.xlabel('Predicted')
    plt.ylabel('True')
    plt.title('Confusion Matrix')
    plt.show()
```

Overall Metrics

- The function calculates and prints overall metrics including accuracy, precision, recall, and F1 score:

```python
    accuracy = accuracy_score(y_true, y_pred)
    precision = precision_score(y_true, y_pred, average='weighted')
    recall = recall_score(y_true, y_pred, average='weighted')
    f1 = f1_score(y_true, y_pred, average='weighted')

    metrics = {
        "Accuracy": accuracy,
```

```
        "Precision": precision,
        "Recall": recall,
        "F1 Score": f1
    }

    print("\nOverall Metrics:")
    for metric, value in metrics.items():
        print(f"{metric}: {value:.4f}")
```

- These metrics are printed in a readable format.

Function Return

- The function returns the classification report DataFrame:

```
    return report_df
```

Model Evaluation

- The `evaluate_model` function is called to evaluate the model:

```
    report_df = evaluate_model(y_test, y_test_pred, ["Negative", "Neutral", "Positive"])
```

0.0.1 Results and Interesting Notes

For the results section, we analyzed different approaches to the data to check whether we could get a better result. At first, we tested it without any special technique and got an accuracy of about 0.61 on the whole data. Another approach we tested is removing stopwords, but it seems stopwords are important for our task, and after removing them, our accuracy dropped.

Another approach was to use a formalizer to change the text into a formal Persian text. We used a formalizer based on T5 using models from previous semesters, but we got worse results. It is predictable, though, because some informal Persian slang can significantly change the sentiment of a sentence, and they lose their meaning even from a human perspective when we formalize them.

We also used another approach for formalization: giving the Persian text to Google Translate to translate it into English and then back to Persian. It didn't work fine either.

One of the things we find in papers that seem good for this task is a lemmatizer. We tried to use it on our data, but unfortunately, Hazm was very slow on our 70K database, so we could not use this technique for our task.

It seems our result is good enough, though. As it is a multiclass classification and the best results are for ParsBERT on a significantly larger database than ours that is near 70%, it seems our accuracy of 61% is good enough.

The code used for formalization based on T5 is as follows:

```
from transformers import (T5ForConditionalGeneration, AutoTokenizer, pipeline)
import torch

model = T5ForConditionalGeneration.from_pretrained('parsi-ai-nlpclass/PersianTextFormalizer')
tokenizer = AutoTokenizer.from_pretrained('parsi-ai-nlpclass/PersianTextFormalizer')

pipe = pipeline(task='text2text-generation', model=model, tokenizer=tokenizer)
def formalizer(text):
  device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
  model.to(device)
  formalized = ""
  for line in text.splitlines():

      inputs = tokenizer.encode("informal: " + line, return_tensors='pt', max_length=128,
          truncation=True, padding='max_length')
      inputs = inputs.to(device)

      outputs = model.generate(inputs, max_length=128, num_beams=4)
```

```python
            formalized = formalized + tokenizer.decode(outputs[0], skip_special_tokens=True) + " "
        return formalized

    data['comment'] = data['comment'].apply(formalizer)
```

The code used for formalization based on Google Translate is as follows:

```python
import googletrans as gt
translator = gt.Translator()

def translation(text):
    en_translated = translator.translate(text, 'en', 'fa')
    fa_translated = translator.translate(en_translated.text, 'fa', 'en')
    return fa_translated.text

data['comment'] = data['comment'].apply(translation)
```

Figure 1: Results of running Logistic Regression on 10000 rows of dataset - With Formalization

```
Best parameters for TF-IDF model are: {'logreg__C': 1, 'tfidf__max_features': 10000, 'tfidf__ngram_range': (1, 2), 'positive_threshold': 4,
'neutral_threshold': 2}
Test accuracy of Logistic Regression model:  0.5208333333333334
              precision    recall  f1-score   support

    negative       0.48      0.49      0.49       117
     neutral       0.56      0.55      0.55       137
    positive       0.52      0.52      0.52       130

    accuracy                           0.52       384
   macro avg       0.52      0.52      0.52       384
weighted avg       0.52      0.52      0.52       384
```

Figure 2: Results of running Logistic Regression on 10000 rows of dataset - Without Formalization

```
Best parameters for TF-IDF model are: {'logreg__C': 0.1, 'tfidf__max_features': 5000, 'tfidf__ngram_range': (1, 3), 'positive_threshold': 4,
'neutral_threshold': 2}
Test accuracy of Logistic Regression model:  0.5416666666666666
              precision    recall  f1-score   support

    negative       0.47      0.56      0.52       117
     neutral       0.54      0.50      0.52       137
    positive       0.62      0.57      0.59       130

    accuracy                           0.54       384
   macro avg       0.55      0.54      0.54       384
weighted avg       0.55      0.54      0.54       384
```

Figure 3: Results of running Logistic Regression on 10000 rows of dataset - without stopwords

```
Best parameters for TF-IDF model are: {'logreg__C': 0.1, 'tfidf__max_features': 10000, 'tfidf__ngram_range': (1, 1), 'positive_threshold':
4, 'neutral_threshold': 2}
Test accuracy of Logistic Regression model:  0.4713541666666667
              precision    recall  f1-score   support

    negative       0.40      0.60      0.48       117
     neutral       0.47      0.42      0.44       137
    positive       0.63      0.41      0.50       130

    accuracy                           0.47       384
   macro avg       0.50      0.48      0.47       384
weighted avg       0.50      0.47      0.47       384

+ Code   + Markdown
```

Figure 4: Results of running Logistic Regression on 70000 rows of dataset - without stopwords

```
Best parameters for TF-IDF model are: {'logreg__C': 0.1, 'tfidf__max_features': 10000, 'tfidf__ngram_range': (1, 2), 'positive_threshold': 4, 'neutral_threshold': 2}
Test accuracy of Logistic Regression model:  0.5606361829025845
              precision    recall  f1-score   support

    negative       0.53      0.59      0.56      1020
     neutral       0.52      0.52      0.52       987
    positive       0.65      0.58      0.61      1011

    accuracy                           0.56      3018
   macro avg       0.56      0.56      0.56      3018
weighted avg       0.57      0.56      0.56      3018
```

Note: Some parts of explanations are written with the help of ChatGPT to help us write clear and unambiguous descriptions.