Folder: api

File: training.py

Documented code for training.py:

Code Explanation:

1. Import Statements:

```
import os
import numpy as np
import tensorflow as tf
from tensorflow import keras
from keras.models import Sequential, load_model
from keras.layers import Dense, Embedding, GlobalAveragePooling1D
from keras.preprocessing.sequence import pad_sequences
import pickle
import pandas as pd
```

- We import necessary libraries and modules for data manipulation, model building, and training.

2. Variable Declarations:

```
VOCAB_SIZE = 20000
MAX_LEN = 250
EMBEDDING_DIM = 16
MODEL_PATH = 'sentiment_analysis_model.h5'
```

- `VOCAB_SIZE`: Maximum number of words in the vocabulary.
- `MAX_LEN`: Maximum length of a padded sequence.
- `EMBEDDING_DIM`: Dimensionality of word embeddings.

- `MODEL_PATH`: Path to save the trained model.

3. Data Loading and Preprocessing:

```
file_path = 'data.csv'
data = pd.read_csv(file_path, encoding='ISO-8859-1')
df_shuffled = data.sample(frac=1).reset_index(drop=True)

texts = []
labels = []

for _, row in df_shuffled.iterrows():
    texts.append(row[-1])
    label = row[0]
    labels.append(0 if label == 0 else 1 if label == 2 else 2)

texts = np.array(texts)
labels = np.array(labels)
```

- We load the CSV data, shuffle it, and extract the text and labels.
- Labels are converted to 0 (negative), 1 (neutral), and 2 (positive).

4. Tokenization and Padding:

```
tokenizer = keras.preprocessing.text.Tokenizer(num_words=VOCAB_SIZE-1, oov_token='<OOV>')
tokenizer.fit_on_texts(texts)

tokenizer.word_index = {word: idx for word, idx in tokenizer.word_index.items() if idx < VOCAB_SIZE - 1}
tokenizer.word_counts = {word: count for word, count in tokenizer.word_counts.items() if
```

```
tokenizer.word_index.get(word)}

sequences = tokenizer.texts_to_sequences(texts)
padded_sequences = pad_sequences(sequences, maxlen=MAX_LEN, value=VOCAB_SIZE-1, padding='post')
```

- We tokenize the text data, limiting the vocabulary size to `VOCAB_SIZE-1` and using `<OOV>` for out-of-vocabulary words.
- We pad the sequences to a maximum length of `MAX_LEN`.

5. Splitting Data into Training and Test Sets:

```
train_data = padded_sequences[:-5000]
test_data = padded_sequences[-5000:]
train_labels = labels[:-5000]
test_labels = labels[-5000:]
```

- We split the data into training and test sets, reserving the last 5000 samples for testing.

6. Model Building and Training:

```
if os.path.exists(MODEL_PATH):
    print("Loading saved model...")
    model = load_model(MODEL_PATH)
else:
    print("Training a new model...")
    model = Sequential([
        Embedding(VOCAB_SIZE, EMBEDDING_DIM, input_length=MAX_LEN),
        GlobalAveragePooling1D(),
```

```
        Dense(16, activation='relu'),
        Dense(3, activation='softmax')  # 3 classes: negative, neutral, positive
    ])

    model.compile(optimizer='adam', loss='sparse_categorical_crossentropy', metrics=['accuracy'])

    model.fit(train_data, train_labels, epochs=10, batch_size=32, validation_split=0.2)

    model.save(MODEL_PATH)
```

- We check if a saved model exists. If it does, we load it; otherwise, we train a new model.
- We define a sequential model with an embedding layer, global average pooling, and dense layers for classification.
- We compile the model with the Adam optimizer, sparse categorical cross-entropy loss, and accuracy metric.
- We train the model for 10 epochs with a batch size of 32 and a 20% validation split.
- We save the trained model to a file.

7. Model Evaluation:

```
loss, accuracy = model.evaluate(test_data, test_labels)
print(f"Test accuracy: {accuracy * 100:.2f}%")
```

- We evaluate the trained model on the test data and print the accuracy.

8. Interactive Loop for Predictions:

```
def encode_text(text):
    tokens = tf.keras.preprocessing.text.text_to_word_sequence(text)
```

```
    tokens = [tokenizer.word_index[word] if word in tokenizer.word_index else 0 for word in tokens]
    return pad_sequences([tokens], maxlen=MAX_LEN, padding='post', value=VOCAB_SIZE-1)

while True:
    user_input = input("Enter a sentence for sentiment analysis (or 'exit' to quit): ")
    if user_input.lower() == 'exit':
        break

    encoded_input = encode_text(user_input)
    prediction = np.argmax(model.predict(encoded_input))

    if prediction == 0:
        print("Sentiment: Negative")
    elif prediction == 1:
        print("Sentiment: Neutral")
    else:
        print("Sentiment: Positive")
```

- We define a function to encode user input text into a padded sequence.
- We enter an interactive loop where the user can input sentences for sentiment analysis.
- We encode the user input, make a prediction using the trained model, and print the sentiment.

Conclusion:

This code demonstrates a complete sentiment analysis pipeline, including data loading, preprocessing, model training, evaluation, and interactive predictions. It provides a detailed explanation of each step, making it suitable for beginners and intermediate-level programmers.

Folder: __pycache__
Folder: static
Folder: templates
File: index.html

Documented code for index.html:

Explanation of index.html:

1. HTML Structure and Styling:

- The HTML structure consists of a `<head>` section for metadata and a `<body>` section for the main content.
- Styling is provided using inline CSS within the `<style>` tag. It defines styles for various elements like the body, header, container, form, input fields, table, and headings.

2. Header Section:

- The header contains an image for the logo and a heading for the page title, "YouTube Sentiment Analysis."

3. Main Container:

- The main container is a `<div>` element with a class "container." It serves as the main content area of the page.

4. Form for User Input:

- Inside the container, there's a form with an `id` of "analysisForm" and an `action` attribute set to "/." This form is used to collect user input.
- It includes an input field for the user to enter a YouTube video URL and a submit button labeled "Analyze."

5. Conditional Display of Results:

- The `{% if summary %}` block checks if the `summary` variable exists. If it does, it means the analysis results are available, and the following sections are displayed:

6. Summary Section:

- The summary section displays various statistics about the sentiment analysis:
  - Positive comments count
  - Negative comments count
  - Total number of comments
  - Overall rating percentage

7. Comments Section:

- The comments section displays a table with two columns: "Comment" and "Sentiment."
- Each row in the table represents a comment from the YouTube video along with its sentiment (positive or negative).

8. JavaScript for Dynamic Height Adjustment:

- A `<script>` section is included at the bottom of the page.
- It contains a function called `adjustCommentsContainerHeight()` that dynamically adjusts the height of the comments container based on the available space on the page.
- This function is called on window resize and page load to ensure the comments container fits properly.

Conclusion:

This HTML code sets up the structure and styling for a web page that allows users to analyze the sentiment of comments on a YouTube video. It includes a form for user input, displays analysis results, and dynamically adjusts the height of the comments container to fit the available space.

File: predict.py
Documented code for predict.py:
Code Explanation:

1. Import Statements (Line 1-5):

```
import numpy as np
import tensorflow as tf
from tensorflow import keras
from keras.models import load_model
from keras.preprocessing.sequence import pad_sequences
import pickle
```

- We import necessary libraries and modules:
  - `numpy` for numerical operations.
  - `tensorflow` and `keras` for deep learning.
  - `keras.models` for loading the saved model.
  - `keras.preprocessing.sequence` for text preprocessing.
  - `pickle` for loading the tokenizer.

2. Constants and Variables (Line 6-10):

```
VOCAB_SIZE = 20000
MAX_LEN = 250
MODEL_PATH = "api\\sentiment_analysis_model.h5"
```

- We define constants and variables:
  - `VOCAB_SIZE`: Maximum vocabulary size (number of unique words) considered.
  - `MAX_LEN`: Maximum length of a text sequence (in words).
  - `MODEL_PATH`: Path to the saved sentiment analysis model.

3. Load the Saved Model (Line 12):

```
model = load_model(MODEL_PATH)
```

- We load the pre-trained sentiment analysis model from the specified path.

4. Load the Tokenizer (Line 14-16):

```
with open('api\\tokenizer.pickle', 'rb') as handle:
    tokenizer = pickle.load(handle)
```

- We load the tokenizer used during model training. It maps words to integer indices.

5. Encode Texts (Line 18-26):

```
def encode_texts(text_list):
    encoded_texts = []
    for text in text_list:
        tokens = tf.keras.preprocessing.text.text_to_word_sequence(text)
        tokens = [tokenizer.word_index.get(word, 0) for word in tokens]
        encoded_texts.append(tokens)
    return pad_sequences(encoded_texts, maxlen=MAX_LEN, padding='post', value=VOCAB_SIZE-1)
```

- We define a function `encode_texts` to convert a list of texts into a list of integer sequences.
  - It tokenizes each text into a list of words.
  - Replaces each word with its corresponding integer index using the tokenizer.
  - Pads the sequences to a fixed length (`MAX_LEN`) with a special value (`VOCAB_SIZE-1`).

6. Predict Sentiments (Line 28-36):

```
def predict_sentiments(text_list):
    encoded_inputs = encode_texts(text_list)
    predictions = np.argmax(model.predict(encoded_inputs), axis=-1)
    sentiments = []
    for prediction in predictions:
        if prediction == 0:
            sentiments.append("Negative")
        elif prediction == 1:
            sentiments.append("Neutral")
        else:
            sentiments.append("Positive")
    return sentiments
```

- We define a function `predict_sentiments` to predict the sentiment of a list of texts.
  - It encodes the texts using the `encode_texts` function.
  - Uses the loaded model to make predictions on the encoded inputs.
  - Converts the predictions (integer labels) into sentiment labels ("Negative", "Neutral", "Positive").

Conclusion:

This code provides a sentiment analysis API that can be used to predict the sentiment of a list of texts. It loads a pre-trained model and tokenizer, encodes the texts into integer sequences, makes predictions using the model, and converts the predictions into sentiment labels.

File: app.py

Documented code for app.py:

1. Import Statements:

```
from flask import Flask, request, render_template
from predict import predict_sentiments
```

```
from youtube import get_video_comments
from flask_cors import CORS
import requests
from urllib.parse import urlparse
```

- We import necessary libraries and modules for the application.
- `Flask` is a lightweight web framework for creating web applications in Python.
- `request` and `render_template` are used for handling HTTP requests and rendering HTML templates.
- `predict_sentiments` and `get_video_comments` are custom modules for sentiment analysis and fetching YouTube comments.
- `CORS` enables Cross-Origin Resource Sharing (CORS) for handling cross-origin requests.
- `requests` is used for making HTTP requests.
- `urlparse` is used for parsing URLs.

2. Flask Application Initialization:

```
app = Flask(__name__)
CORS(app)
```

- We create a Flask application instance named `app`.
- We enable CORS for the application using `CORS(app)`.

3. `get_video` Function:

```
def get_video(video_id):
    if not video_id:
        return {"error": "video_id is required"}
```

```
    comments = get_video_comments(video_id)
    predictions = predict_sentiments(comments)

    positive = predictions.count("Positive")
    negative = predictions.count("Negative")

    summary = {
        "positive": positive,
        "negative": negative,
        "num_comments": len(comments),
        "rating": (positive / len(comments)) * 100
    }

    return {"predictions": predictions, "comments": comments, "summary": summary}
```

- This function takes a `video_id` as input and returns a dictionary containing sentiment analysis results and a summary of the comments.
- It first checks if the `video_id` is provided. If not, it returns an error message.
- It then fetches the comments for the video using the `get_video_comments` function and performs sentiment analysis on the comments using the `predict_sentiments` function.
- It counts the number of positive and negative sentiments and calculates the overall rating.
- Finally, it returns a dictionary containing the predictions, comments, and a summary of the results.

4. `getvideo_id` Function:

```
def getvideo_id(value):
    """
    Examples:
    - http://youtu.be/SA2iWivDJiE
    - http://www.youtube.com/watch?v=_oPAwA_Udwc&feature=feedu
    - http://www.youtube.com/embed/SA2iWivDJiE
```

```
    - http://www.youtube.com/v/SA2iWivDJiE?version=3&amp;hl=en_US
    """
    query = urlparse(value)
    if query.hostname == 'youtu.be':
        return query.path[1:]
    if query.hostname in ('www.youtube.com', 'youtube.com'):
        if query.path == '/watch':
            p = urlparse(query.query)
            return str(p.path[2:]).split('&')[0]
        if query.path[:7] == '/embed/':
            return query.path.split('/')[2]
        if query.path[:3] == '/v/':
            return query.path.split('/')[2]
    # fail?
    return None
```

- This function takes a YouTube URL or video ID as input and extracts the video ID from it.
- It handles various formats of YouTube URLs and returns the video ID as a string.

5. Main Route (`/`) Handler:

```
@app.route('/', methods=['GET', 'POST'])
def index():
    summary = None
    comments = []
    if request.method == 'POST':
        video_url = request.form.get('video_url')
        video_id = getvideo_id(video_url)
        print(video_id)
        data = get_video(video_id)
```

```
        summary = data['summary']
        comments = list(zip(data['comments'], data['predictions']))
    return render_template('index.html', summary=summary, comments=comments)
```

- This function handles requests to the root URL (`/`).
- It initializes variables `summary` and `comments` to store the sentiment analysis results and comments, respectively.
- When a POST request is received, it extracts the video URL from the request form, extracts the video ID using the `getvideo_id` function, and calls the `get_video` function to fetch the sentiment analysis results and comments.
- It then renders the `index.html` template, passing the `summary` and `comments` as variables to be displayed in the template.

6. Main Program:

```
if __name__ == '__main__':
    app.run(debug=True)
```

- This is the entry point of the program.
- It checks if the script is being run directly (not imported as a module).
- If so, it starts the Flask development server in debug mode, which allows for live reloading of code changes.

File: youtube.py

Documented code for youtube.py:

Code Explanation:

1. Import Statements:

```
from flask import Flask, request, render_template
from youtube_transcript_api import YouTubeTranscriptApi
from vaderSentiment.vaderSentiment import SentimentIntensityAnalyzer
```

- We import necessary libraries:
  - `Flask`: For creating a web application.
  - `request`: For handling HTTP requests.
  - `render_template`: For rendering HTML templates.
  - `YouTubeTranscriptApi`: For fetching YouTube video transcripts.
  - `SentimentIntensityAnalyzer`: For sentiment analysis.

2. Variable Declarations:

```

app = Flask(__name__)
analyzer = SentimentIntensityAnalyzer()
```

- We create a Flask application instance (`app`) and an instance of the `SentimentIntensityAnalyzer` for sentiment analysis (`analyzer`).

3. User Input Handling:

```

@app.route('/', methods=['GET', 'POST'])
def index():
    if request.method == 'POST':
        video_url = request.form['video_url']
```

- We define a route handler for the root URL (`/`) that handles both GET and POST requests.
- If the request method is `POST`, we extract the YouTube video URL from the submitted form.

4. Control Flow - Part 1:

```
try:
    transcript = YouTubeTranscriptApi.get_transcript(video_url)
except:
    return render_template('index.html', error="Invalid YouTube URL")
```

- We use a `try-except` block to handle potential errors when fetching the video transcript.
- If the URL is invalid or the transcript is unavailable, we return an error message.

5. Function Calls - Part 1:

```
comments = []
for item in transcript:
    sentiment = analyzer.polarity_scores(item['text'])
    comments.append((item['text'], sentiment['compound']))
```

- We iterate through the transcript items and perform sentiment analysis on each comment.
- We store the comment and its sentiment score in a list called `comments`.

6. Looping Structures:

```
summary = {
    'positive': 0,
    'negative': 0,
    'num_comments': len(comments),
    'rating': 0
```

```
    }

    for comment, sentiment in comments:
        if sentiment > 0:
            summary['positive'] += 1
        elif sentiment < 0:
            summary['negative'] += 1
```

- We initialize a dictionary called `summary` to store the sentiment analysis summary.
- We iterate through the `comments` list and count the number of positive and negative comments.
- We also calculate the overall rating based on the sentiment scores.

7. Data Manipulation:

```

    summary['rating'] = (summary['positive'] / summary['num_comments']) * 100
```

- We calculate the rating percentage by dividing the number of positive comments by the total number of comments and multiplying by 100.

8. Control Flow - Part 2:

```

    if summary['rating'] >= 70:
        summary['rating'] = 'Positive'
    elif summary['rating'] <= 30:
        summary['rating'] = 'Negative'
    else:
        summary['rating'] = 'Neutral'
```

- We classify the rating into three categories: Positive, Negative, and Neutral based on the calculated rating percentage.

9. Function Calls - Part 2:

```
    return render_template('index.html', summary=summary, comments=comments)
```

- We render the `index.html` template, passing the `summary` and `comments` as variables.

10. Output Generation:

```html
{% if summary %}
<div id="summarySection">
    <h2>Summary</h2>
    <p>Positive: {{ summary['positive'] }}</p>
    <p>Negative: {{ summary['negative'] }}</p>
    <p>Number of Comments: {{ summary['num_comments'] }}</p>
    <p>Rating: {{ summary['rating'] }}%</p>
</div>
<div class="comments-container" id="commentsContainer">
    <h3>Comments</h3>
    <table>
        <tr>
            <th>Comment</th>
            <th>Sentiment</th>
        </tr>
        {% for comment, sentiment in comments %}
        <tr>
            <td>{{ comment }}</td>
            <td>{{ sentiment }}</td>
```

```
        </tr>
        {% endfor %}
    </table>
</div>
{% endif %}
```

- In the HTML template, we check if the `summary` variable exists (indicating that the analysis was successful).
- If so, we display the summary and the comments in the appropriate sections.

11. Error Handling:

```
@app.errorhandler(404)
def page_not_found(e):
    return render_template('404.html'), 404
```

- We define an error handler for 404 errors (page not found).
- When a 404 error occurs, it renders the `404.html` template with a 404 status code.

Conclusion:

This Flask application allows users to analyze the sentiment of comments on a YouTube video by providing its URL. It fetches the video transcript, performs sentiment analysis on each comment, and generates a summary of the overall sentiment. The results are displayed on a web page, including a rating based on the sentiment scores. The application also handles errors gracefully, such as invalid URLs or missing transcripts.