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 for data manipulation, model building, and text processing.

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 sentiment analysis dataset from a CSV file.
- We shuffle the data to avoid any bias in training.
- We extract the text and label columns from the DataFrame.
- We convert the labels 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 create a tokenizer to convert text to sequences of integers.
- We limit the vocabulary size to `VOCAB_SIZE-1` and use `<OOV>` for out-of-vocabulary words.
- We convert the texts to sequences of integers and pad them 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 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, a global average pooling layer, and two dense layers.
- 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 validation split of 20%.
- 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 model on the test set 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 create an interactive loop that allows users to enter sentences for sentiment analysis.
- We encode the user input, make a prediction, 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 accessible to beginners and intermediate-level programmers.

Folder: __pycache__

Folder: static

Folder: templates

File: predict.py

Documented code for predict.py:

Code Explanation:

1. Import Statements (Line 1-5):

- `import numpy as np`: Imports the NumPy library for numerical operations.
- `import tensorflow as tf`: Imports the TensorFlow library for deep learning.
- `from tensorflow import keras`: Imports the Keras API from TensorFlow for building and training neural networks.
- `from keras.models import load_model`: Imports the `load_model()` function for loading a pre-trained Keras model.
- `from keras.preprocessing.sequence import pad_sequences`: Imports the `pad_sequences()` function for padding sequences to a fixed length.
- `import pickle`: Imports the Pickle library for serializing and deserializing Python objects.

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

- `VOCAB_SIZE = 20000`: Specifies the size of the vocabulary used for tokenization.
- `MAX_LEN = 250`: Specifies the maximum length of the padded sequences.
- `MODEL_PATH = "api\\sentiment_analysis_model.h5"`: Specifies the path to the saved sentiment analysis model.

3. Load the Saved Model (Line 12):

- `model = load_model(MODEL_PATH)`: Loads the pre-trained sentiment analysis model from the specified path.

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

- Opens the file containing the serialized tokenizer using `with open('api\\tokenizer.pickle', 'rb') as handle:`.

- Deserializes the tokenizer using `tokenizer = pickle.load(handle)`.

5. Encode Texts Function (Line 18-27):

- `def encode_texts(text_list):`: Defines a function to encode a list of texts into numerical sequences.
- `encoded_texts = []`: Initializes an empty list to store the encoded texts.
- The function iterates through each text in `text_list`:
  - Tokenizes the text using `tf.keras.preprocessing.text.text_to_word_sequence(text)`.
  - Converts the tokens to integers using the tokenizer's word index.
  - Appends the encoded text to `encoded_texts`.
- Finally, it pads the sequences to a fixed length using `pad_sequences()` and returns the encoded texts.

6. Predict Sentiments Function (Line 29-39):

- `def predict_sentiments(text_list):`: Defines a function to predict the sentiment of a list of texts.
- `encoded_inputs = encode_texts(text_list)`: Encodes the input texts using the `encode_texts()` function.
- `predictions = np.argmax(model.predict(encoded_inputs), axis=-1)`: Makes predictions on the encoded inputs using the loaded model.
- `sentiments = []`: Initializes an empty list to store the predicted sentiments.
- The function iterates through each prediction:
  - If the prediction is 0, it appends "Negative" to `sentiments`.
  - If the prediction is 1, it appends "Neutral" to `sentiments`.
  - If the prediction is 2, it appends "Positive" to `sentiments`.
- Finally, it returns the list of predicted sentiments.

Conclusion:

This code provides a comprehensive solution for sentiment analysis. It loads a pre-trained model, tokenizes and encodes input texts, makes predictions, and returns the predicted sentiments. This code can be used to analyze the sentiment of text data, such as customer reviews or social media posts.

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 Flask for creating a web application, request for handling HTTP requests, render_template for rendering HTML templates, and CORS for enabling cross-origin resource sharing.
- We also import predict_sentiments from the predict module, get_video_comments from the youtube module, and urlparse from the urllib.parse module.

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 the sentiment analysis results for the video's comments.
- It first checks if the video ID is provided. If not, it returns an error message.
- It then fetches the video's comments using the get_video_comments function and predicts the sentiment of each comment using the predict_sentiments function.
- It counts the number of positive and negative comments and calculates the overall rating based on the ratio of positive comments to the total number of comments.
- Finally, it returns a dictionary with the predictions, comments, and summary information.

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 extracts the video ID from a given YouTube URL.
- It handles various formats of YouTube URLs and returns the video ID as a string.

5. Index Route:

```
@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 is the main route of the application, accessible at the root URL '/'.
- It handles both GET and POST requests.
- When a POST request is made, 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.
- It then renders the 'index.html' template with the summary and comments data.

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) using the __name__ == '__main__' condition.
- If so, it starts the Flask application in debug mode, allowing for live reloading of code changes.

File: youtube.py

Documented code for youtube.py:

Code Explanation:

1. Import Statements (Line 1-4):

```
import os
import googleapiclient.discovery
import googleapiclient.errors
import os
from dotenv import load_dotenv
```

- We import necessary libraries and modules:
  - `os`: For interacting with the operating system.
  - `googleapiclient.discovery`: For interacting with YouTube Data API.
  - `googleapiclient.errors`: For handling errors related to YouTube Data API.
  - `dotenv`: For loading environment variables from a `.env` file.

2. Environment Variable Loading (Line 5-6):

```
load_dotenv()
api_key = os.getenv("API_KEY")
```

- We load environment variables from a `.env` file using `load_dotenv()`.
- We retrieve the API key from the environment variable `API_KEY` and store it in the `api_key` variable.

3. Function: `get_comments` (Line 8-32):

```
def get_comments(youtube, **kwargs):
    comments = []
    results = youtube.commentThreads().list(**kwargs).execute()
```

```
    while results:
        for item in results['items']:
            comment = item['snippet']['topLevelComment']['snippet']['textDisplay']
            comments.append(comment)

        # check if there are more comments
        if 'nextPageToken' in results:
            kwargs['pageToken'] = results['nextPageToken']
            results = youtube.commentThreads().list(**kwargs).execute()
        else:
            break

    return comments
```

- This function retrieves comments for a given video.
- It takes two parameters: `youtube` (an authorized YouTube Data API client) and `**kwargs` (keyword arguments).
- It initializes an empty list `comments` to store the retrieved comments.
- It makes an initial call to the YouTube Data API's `commentThreads().list()` method to fetch the first page of comments.
- It enters a `while` loop that continues as long as there are more comments to fetch.
- Inside the loop, it iterates through the `items` in the `results` dictionary and extracts the comment text from each `item`.
- It appends the extracted comment to the `comments` list.
- It checks if there's a `nextPageToken` in the `results` dictionary, indicating more comments to fetch.
- If there's a `nextPageToken`, it updates the `kwargs` dictionary with the `pageToken` and makes another call to the YouTube Data API to fetch the next page of comments.
- Once all comments are fetched, it returns the `comments` list.

4. Function: `main` (Line 34-42):

```
def main(video_id, api_key):
    # Disable OAuthlib's HTTPs verification when running locally.
    os.environ["OAUTHLIB_INSECURE_TRANSPORT"] = "1"

    youtube = googleapiclient.discovery.build(
        "youtube", "v3", developerKey=api_key)

    comments = get_comments(youtube, part="snippet", videoId=video_id, textFormat="plainText")
    return comments
```

- This function is the entry point of the script.
- It takes two parameters: `video_id` (the ID of the video for which comments are to be fetched) and `api_key` (the YouTube Data API key).
- It disables OAuthlib's HTTPS verification when running locally, which is necessary for making API calls without a browser.
- It creates an authorized YouTube Data API client using `googleapiclient.discovery.build()`.
- It calls the `get_comments()` function to fetch comments for the specified `video_id`.
- It returns the list of comments retrieved from the YouTube Data API.

5. Function: `get_video_comments` (Line 44-45):

```
def get_video_comments(video_id):
    return main(video_id, api_key)
```

- This function is a wrapper around the `main()` function.
- It takes a single parameter: `video_id` (the ID of the video for which comments are to be fetched).
- It calls the `main()` function with the provided `video_id` and the `api_key` loaded from the environment variable.

- It returns the list of comments retrieved from the YouTube Data API.

Conclusion:

This script provides a way to fetch comments for a given YouTube video using the YouTube Data API. It uses the `get_comments()` function to retrieve comments in a paginated manner and stores them in a list. The `main()` function serves as the entry point and calls the `get_comments()` function to fetch comments for a specified video. The `get_video_comments()` function is a wrapper around the `main()` function and provides a simplified interface for fetching comments.