**Source Code Documentation: Library Imports and Initialization**

1. **Importing Necessary Libraries**
2. from ollama import chat
3. from nltk.sentiment import SentimentIntensityAnalyzer
4. from transformers import pipeline, AutoTokenizer, AutoModelForSequenceClassification
5. from collections import defaultdict
6. import re
7. import matplotlib.pyplot as plt
8. import tkinter as tk
9. from tkinter import ttk
10. from nltk.corpus import stopwords
11. import spacy
12. from datasets import Dataset
13. import pandas as pd
14. import os
15. import torch
16. import pymysql
17. import csv
18. nlp = spacy.load("en\_core\_web\_sm")

**1.1 Machine Learning and NLP Libraries**

* from ollama import chat → Imports the chat function from the ollama library (likely related to AI-based chat functionalities).
* from nltk.sentiment import SentimentIntensityAnalyzer → Loads the SentimentIntensityAnalyzer from the Natural Language Toolkit (NLTK) for sentiment analysis.
* from transformers import pipeline, AutoTokenizer, AutoModelForSequenceClassification → Imports key components from the Hugging Face transformers library:
  + pipeline for high-level NLP tasks.
  + AutoTokenizer for automatic text tokenization.
  + AutoModelForSequenceClassification for pre-trained transformer models used in classification tasks.

**1.2 Data Processing and Storage**

* from collections import defaultdict → Imports defaultdict to simplify handling dictionary-based data structures.
* import re → Provides regex-based text processing capabilities.
* import pandas as pd → Enables structured data handling and manipulation with Pandas DataFrames.
* import csv → Supports reading and writing CSV files.
* import os → Provides access to system-related functions, such as file handling.

**1.3 Visualization and GUI Development**

* import matplotlib.pyplot as plt → Enables graphical data visualization using Matplotlib.
* import tkinter as tk → Imports Tkinter for GUI development.
* from tkinter import ttk → Enhances GUI with additional styled widgets from ttk.

**1.4 Natural Language Processing**

* from nltk.corpus import stopwords → Provides stopwords for text preprocessing.
* import spacy → Loads the spaCy NLP toolkit for advanced text analysis.
* from datasets import Dataset → Enables handling of large-scale NLP datasets.

**1.5 Deep Learning and Database Handling**

* import torch → Provides tensor-based operations and deep learning capabilities using PyTorch.
* import pymysql → Enables MySQL database connectivity for storing and retrieving processed data.

**1.6. Initializing NLP Model**

* nlp = spacy.load("en\_core\_web\_sm") → Loads the **spaCy Small English Model**, which provides:
  + Tokenization
  + Named Entity Recognition (NER)
  + Part-of-Speech (POS) tagging
  + Dependency parsing

**Class: FileHandler**

* class FileHandler:
* def \_\_init\_\_(self, dataset, stopwords, emotions, dataProcessor):
* self.dataset = dataset
* self.stopwords = stopwords
* self.emotions = emotions
* self.dataProcessor = dataProcessor  # Store the dataProcessor instance
* @staticmethod
* def cleanData(filePath, outputFilePath):
* try:
* # Load the dataset
* data = pd.read\_csv(filePath, quotechar='"')
* print(f"Dataset loaded successfully. Shape: {data.shape}")
* except FileNotFoundError:
* print(f"Error: File not found at {filePath}.")
* return
* except pd.errors.EmptyDataError:
* print(f"Error: Dataset at {filePath} is empty.")
* return
* except Exception as e:
* print(f"Unexpected error loading dataset: {e}")
* return
* # Detect and fill missing values
* print("Checking for missing values...")
* missingCount = data.isnull().sum()
* print("Missing values before cleaning:")
* print(missingCount)
* for column in data.columns:
* if data[column].dtype == "object":  # String or object columns
* data[column] = data[column].fillna("None")
* else:  # Numeric columns
* data[column] = data[column].fillna(0)
* print("Missing values after cleaning:")
* print(data.isnull().sum())
* # Save the cleaned dataset
* try:
* data.to\_csv(outputFilePath, index=False, quoting=csv.QUOTE\_MINIMAL)
* print(f"Cleaned dataset saved to {outputFilePath}.")
* except Exception as e:
* print(f"Error saving cleaned dataset: {e}")
* def createTextsCsv(self, calculateToneImpact, dataProcessor):
* def addToneAndImpact(dataset):
* try:
* return dataset.map(calculateToneImpact, batched=True)
* except Exception as e:
* print(f"Error in 'addToneAndImpact': {e}")
* return dataset
* def addEmotions(dataset):
* def processBatch(batch):
* smallBatchSize = 16  # Reduce if API limitations exist
* emotionScores = defaultdict(list)
* for i in range(0, len(batch['text']), smallBatchSize):
* smallBatch = {"text": batch['text'][i:i + smallBatchSize]}
* try:
* smallBatchScores = dataProcessor.extractEmotions(smallBatch)
* for emotion, scores in smallBatchScores.items():
* emotionScores[emotion].extend(scores)
* except Exception as e:
* print(f"Error processing sub-batch: {e}")
* for emotion, scores in emotionScores.items():
* if len(scores) < len(batch['text']):
* scores.extend([0] \* (len(batch['text']) - len(scores)))
* batch[emotion] = scores
* for emotion in emotionScores.keys():
* impactKey = f"impact\_{emotion}"
* batch[impactKey] = [
* score \* ((int(likes) // 10) + int(comments))
* for score, likes, comments in zip(batch[emotion], batch['likes'], batch['comments'])
* ]
* return batch
* try:
* return dataset.map(processBatch, batched=True)
* except Exception as e:
* print(f"Error in 'addEmotions': {e}")
* return dataset
* print("Creating 'temp1texts.csv'...")
* dataset = self.dataset
* dataset = addToneAndImpact(dataset)
* dataset = addEmotions(dataset)
* dataset.to\_csv('temp1texts.csv', index=False, quoting=csv.QUOTE\_MINIMAL)
* def createWordsCsv(self, inputFilePath, outputFilePath):
* print("Processing texts to create words.csv...")
* try:
* textsDf = pd.read\_csv(inputFilePath, quotechar='"')
* # Ensure all columns except 'text' are numeric
* for col in textsDf.columns:
* if col != "text":
* textsDf[col] = pd.to\_numeric(textsDf[col], errors="coerce")
* except Exception as e:
* print(f"Error loading or processing input file: {e}")
* return
* if "text" not in textsDf.columns:
* print("Input file must contain a 'text' column.")
* return
* wordData = {}
* nlp = spacy.load("en\_core\_web\_sm")
* for \_, row in textsDf.iterrows():
* text = row['text']
* doc = nlp(text.lower())
* lemmatizedWords = set(token.lemma\_ for token in doc if token.is\_alpha)
* for word in lemmatizedWords:
* if word not in self.stopwords:
* if word not in wordData:
* wordData[word] = {col: 0 for col in textsDf.columns if col != "text"}
* for col in textsDf.columns:
* if col != "text":
* wordData[word][col] += row[col]  # Numeric addition
* wordDf = pd.DataFrame.from\_dict(wordData, orient="index").reset\_index()
* wordDf.rename(columns={"index": "word"}, inplace=True)
* try:
* wordDf.to\_csv(outputFilePath, index=False, quoting=csv.QUOTE\_MINIMAL)
* print(f"Words data saved to {outputFilePath}.")
* except Exception as e:
* print(f"Error saving words.csv: {e}")
* @staticmethod
* def cleanTempFiles():
* tempFiles = ['temp1texts.csv', 'temp2texts.csv', 'temp3texts.csv', 'temp4texts.csv', 'temp5texts.csv']
* for file in tempFiles:
* try:
* if os.path.exists(file):
* os.remove(file)
* print(f"Removed temporary file: {file}")
* except Exception as e:
* print(f"Error removing file {file}: {e}")

This class is responsible for handling file operations, including data cleaning, text processing, and emotion extraction.

**1.1 Constructor (\_\_init\_\_ Method)**

* **Parameters:**
  + dataset: Input dataset containing text data.
  + stopwords: Stopword list for filtering words.
  + emotions: Emotion labels used for sentiment analysis.
  + dataProcessor: Object responsible for processing textual data.
* **Purpose:**
  + Initializes the FileHandler object and stores the dataset and processing objects for later use.

**2. Static Method: cleanData(filePath, outputFilePath)**

* **Purpose:** Cleans and preprocesses a dataset.
* **Steps:**
  1. Reads the dataset from a CSV file.
  2. Handles errors related to missing or incorrect files.
  3. Identifies and fills missing values:
     + Fills missing text values with "None".
     + Fills missing numerical values with 0.
  4. Saves the cleaned dataset to a new CSV file.

**3. Method: createTextsCsv(calculateToneImpact, dataProcessor)**

* **Purpose:** Generates a CSV file (temp1texts.csv) containing processed text data with sentiment and emotion analysis.
* **Steps:**
  1. **Helper Function: addToneAndImpact(dataset)**
     + Applies the calculateToneImpact function to compute sentiment scores.
  2. **Helper Function: addEmotions(dataset)**
     + Processes text in small batches.
     + Extracts emotion scores using dataProcessor.extractEmotions().
     + Computes emotion impact scores based on likes and comments.
  3. Saves the processed dataset to temp1texts.csv.

**4. Method: createWordsCsv(inputFilePath, outputFilePath)**

* **Purpose:** Generates a CSV file (words.csv) containing word-level frequency and sentiment analysis.
* **Steps:**
  1. Reads and validates inputFilePath (ensures numeric data and presence of text column).
  2. Processes text using spaCy NLP:
     + Tokenizes text and extracts lemmatized words.
     + Filters out stopwords.
     + Aggregates sentiment and engagement metrics for each word.
  3. Saves processed word data to outputFilePath.

**5. Static Method: cleanTempFiles()**

* **Purpose:** Removes temporary intermediate CSV files generated during processing.
* **Steps:**
  1. Iterates through a predefined list of temporary files (temp1texts.csv to temp5texts.csv).
  2. Deletes each file if it exists.
  3. Handles file deletion errors gracefully.

**Class: DataProcessor**

* class DataProcessor:
* def \_\_init\_\_(self, dataset, device, apiModels, emotions):
* self.dataset = dataset
* self.device = device
* self.apiPipelines = {
* model: pipeline("text-classification", model=model, top\_k=None, device=DataProcessor.getBestGpu())
* for model in apiModels
* }
* self.emotions = emotions
* @staticmethod
* def getAllStopwords():
* nltkLanguages = stopwords.fileids()
* nltkStopwords = set()
* for lang in nltkLanguages:
* nltkStopwords.update(stopwords.words(lang))
* spacyStopwords = set()
* spacyLanguages = [
* "af", "ar", "bg", "bn", "ca", "cs", "da", "de", "el", "en", "es", "et", "fa", "fi", "fr", "ga",
* "gu", "he", "hi", "hr", "hu", "id", "is", "it", "kn", "lt", "lv", "mr", "nb", "nl",
* "pl", "pt", "ro", "ru", "si", "sk", "sl", "sq", "sr", "sv", "ta", "te", "tl", "tr", "uk",
* "ur", "zh"
* ]
* for lang in spacyLanguages:
* try:
* nlp = spacy.blank(lang)
* spacyStopwords.update(nlp.Defaults.stop\_words)
* except Exception as e:
* print(f"Skipping stopwords for language '{lang}' due to error: {e}")
* return nltkStopwords.union(spacyStopwords)
* @staticmethod
* def getBestGpu():
* bestGpu = -1
* maxFreeMemory = 0
* for i in range(torch.cuda.device\_count()):
* freeMemory = torch.cuda.get\_device\_properties(i).total\_memory - torch.cuda.memory\_allocated(i)
* if freeMemory > maxFreeMemory:
* maxFreeMemory = freeMemory
* bestGpu = i
* return bestGpu
* def calculateToneImpact(self, batch):
* tones = [SentimentIntensityAnalyzer().polarity\_scores(text)['compound'] for text in batch['text']]
* impacts = [
* tone \* ((int(likes) // 10) + int(comments))
* for tone, likes, comments in zip(tones, batch['likes'], batch['comments'])
* ]
* return {"tone": tones, "impact": impacts}
* def extractEmotions(self, batch):
* emotionScores = defaultdict(list)
* for model, apiPipeline in self.apiPipelines.items():
* try:
* outputs = apiPipeline(batch['text'])
* for textIndex, output in enumerate(outputs):
* for emotionData in output:
* label = emotionData['label']
* score = emotionData['score']
* if label not in emotionScores:
* emotionScores[label] = [0] \* len(batch['text'])
* emotionScores[label][textIndex] += score
* except Exception as e:
* print(f"Error processing with model {model}: {e}")
* averagedEmotionScores = {
* emotion: [score / len(self.apiPipelines) for score in scores]
* for emotion, scores in emotionScores.items()
* }
* return averagedEmotionScores
* def calculateImpactEmotions(self, batch):
* emotionImpacts = {}
* for emotion in self.emotions:
* emotionColumn = f"emotion\_{emotion}"
* if emotionColumn not in batch:
* print(f"Missing column: {emotionColumn}. Assigning default values.")
* emotionImpacts[f"impact\_{emotion}"] = [0.0] \* len(batch['text'])
* continue
* emotionImpacts[f"impact\_{emotion}"] = [
* float(emotionScore) \* ((int(likes) // 10) + int(comments))
* for emotionScore, likes, comments in zip(batch[emotionColumn], batch['likes'], batch['comments'])
* ]
* return emotionImpacts

Handles text processing, sentiment analysis, and emotion extraction.

**Constructor (\_\_init\_\_ Method)**

* **Parameters:**
  + dataset: Input dataset for analysis.
  + device: Specifies the computation device (CPU/GPU).
  + apiModels: List of models used for text classification.
  + emotions: Emotion categories for analysis.
* **Functionality:**
  + Initializes API pipelines for sentiment classification using pre-trained models.
  + Determines the best GPU available for execution.

**Static Method: getAllStopwords()**

* **Functionality:**
  + Retrieves stopwords from both NLTK and spaCy across multiple languages.
  + Handles errors for languages unsupported by spaCy.
  + Returns a unified set of stopwords.

**Static Method: getBestGpu()**

* **Functionality:**
  + Identifies the GPU with the highest available memory.
  + Iterates through available CUDA devices and selects the one with the most free memory.
  + Returns the device index of the best GPU.

**Method: calculateToneImpact(batch)**

* **Functionality:**
  + Uses SentimentIntensityAnalyzer to compute sentiment scores for each text in batch.
  + Calculates impact scores based on sentiment tone and engagement metrics (likes/comments).
  + Returns a dictionary with tone and impact values.

**Method: extractEmotions(batch)**

* **Functionality:**
  + Uses pre-trained models to extract emotion scores for given text data.
  + Iterates over API models to obtain classification outputs.
  + Aggregates and normalizes emotion scores across all models.
  + Returns a dictionary containing emotion scores for each text entry.

**Method: calculateImpactEmotions(batch)**

* **Functionality:**
  + Computes emotion-based impact scores by multiplying extracted emotion intensities with engagement metrics (likes/comments).
  + Assigns default values if the required emotion column is missing.
  + Returns a dictionary containing impact scores for each emotion.

**Class: ComplexEmotionProcessor**

* class ComplexEmotionProcessor:
* def \_\_init\_\_(self, tempTextsPath, outputTextsPath, filteredEmotions):
* self.tempTextsPath = tempTextsPath
* self.outputTextsPath = outputTextsPath
* self.filteredEmotions = filteredEmotions
* def calculateComplexEmotions(self, row, filteredEmotions):
* complexEmotionScores = {}
* for complexEmotion, baseEmotions in filteredEmotions.items():
* baseScores = []
* for base in baseEmotions:
* colName = f"emotion\_{base}" if f"emotion\_{base}" in row.index else base
* if colName in row:
* baseScores.append(row[colName])
* else:
* print(f"Missing column: {colName}")
* complexEmotionScores[f"emotion\_{complexEmotion}"] = (
* sum(baseScores) / len(baseScores) if baseScores else 0
* )
* return complexEmotionScores
* def addImpactColumns(self, df, complexEmotions):
* for emotion in complexEmotions.keys():
* colName = f"emotion\_{emotion}" if f"emotion\_{emotion}" in df.columns else emotion
* impactColName = f"impact\_{emotion}"
* if colName in df.columns:
* if impactColName in df.columns:
* print(f"Impact column {impactColName} already exists. Skipping calculation.")
* continue
* df[impactColName] = df[colName] \* (
* (df["likes"] // 10) + df["comments"]
* )
* else:
* print(f"Missing complex emotion column for impact: {colName}")
* return df
* def processTexts(self):
* print("Processing complex emotions for texts...")
* textsDf = pd.read\_csv(self.tempTextsPath, quotechar='"')
* complexEmotionScores = textsDf.apply(
* lambda row: self.calculateComplexEmotions(row, self.filteredEmotions), axis=1
* )
* complexEmotionDf = pd.DataFrame(complexEmotionScores.tolist())
* updatedTextsDf = pd.concat([textsDf, complexEmotionDf], axis=1)
* updatedTextsDf = self.addImpactColumns(updatedTextsDf, self.filteredEmotions)
* updatedTextsDf.to\_csv(self.outputTextsPath, index=False, quoting=csv.QUOTE\_MINIMAL)
* print(f"Updated texts saved to {self.outputTextsPath}")

Handles processing of complex emotions by deriving them from base emotions and calculating their impact scores.

**Constructor (\_\_init\_\_ Method)**

* **Parameters:**
  + tempTextsPath: Path to the temporary dataset containing text and base emotion scores.
  + outputTextsPath: Path to save the processed dataset.
  + filteredEmotions: Dictionary mapping complex emotions to their respective base emotions.
* **Functionality:**
  + Stores file paths and emotion mappings for later processing.

**Method: calculateComplexEmotions(row, filteredEmotions)**

* **Functionality:**
  + Computes complex emotion scores by averaging the scores of base emotions.
  + Iterates through each complex emotion and retrieves scores of associated base emotions.
  + Handles missing emotion columns by printing warnings.
  + Returns a dictionary containing calculated complex emotion scores.

**Method: addImpactColumns(df, complexEmotions)**

* **Functionality:**
  + Calculates impact scores for complex emotions based on engagement metrics (likes/comments).
  + Adds new impact columns if they do not already exist.
  + Prints warnings for missing complex emotion columns.
  + Returns the updated DataFrame with new impact columns.

**Method: processTexts()**

* **Functionality:**
  + Loads the dataset from tempTextsPath.
  + Applies calculateComplexEmotions() to compute complex emotion scores.
  + Combines computed complex emotions with the original dataset.
  + Calls addImpactColumns() to generate impact scores.
  + Saves the updated dataset to outputTextsPath.

**Class: ToneAdjuster**

* class ToneAdjuster:
* def \_\_init\_\_(self, positiveEmotions, negativeEmotions):
* self.positiveEmotions = positiveEmotions
* self.negativeEmotions = negativeEmotions
* def adjustToneAndImpact(self, df):
* if "tone" not in df.columns:
* print("Tone column is missing from DataFrame.")
* return df
* positiveSum = df[[col for col in self.positiveEmotions if col in df.columns]].sum(axis=1, skipna=True)
* negativeSum = df[[col for col in self.negativeEmotions if col in df.columns]].sum(axis=1, skipna=True)
* df["adjusted\_tone"] = df["tone"] + positiveSum - negativeSum
* if "likes" in df.columns and "comments" in df.columns:
* df["adjusted\_impact"] = df["adjusted\_tone"] \* ((df["likes"] // 10) + df["comments"])
* else:
* print("Likes or comments column missing. Impact cannot be recalculated.")
* return df

Handles the adjustment of tone and impact scores by incorporating positive and negative emotions.

**Constructor (\_\_init\_\_ Method)**

* **Parameters:**
  + positiveEmotions: List of columns representing positive emotions.
  + negativeEmotions: List of columns representing negative emotions.
* **Functionality:**
  + Stores positive and negative emotion categories for tone adjustment.

**Method: adjustToneAndImpact(df)**

* **Functionality:**
  + Checks if the "tone" column exists in the dataset; prints a warning if missing.
  + Computes the sum of all positive emotions and negative emotions for each row.
  + Adjusts the tone score by adding positive emotion values and subtracting negative emotion values.
  + If "likes" and "comments" columns exist, recalculates the "adjusted\_impact" score based on engagement metrics.
  + Prints a warning if engagement metrics are missing.
  + Returns the updated DataFrame with "adjusted\_tone" and "adjusted\_impact" columns.

**Class: PoliticalScoreProcessor**

1. class PoliticalScoreProcessor:
2. def \_\_init\_\_(self, sentimentDataset, outputTextsPath):
3. self.sentimentDataset = sentimentDataset
4. self.outputTextsPath = outputTextsPath
5. def processTexts(self):
6. try:
7. with open("socio-economic-instructions.txt", 'r') as file:
8. instruction\_text = file.read().replace('\n', ' ')
9. except Exception as e:
10. print(f"Error reading file: {e}")
11. economicScores, socialScores, economicImpacts, socialImpacts = [], [], [], []
12. for index, row in self.sentimentDataset.iterrows():
13. try:
14. text = row['text']
15. likes = int(row.get('likes', 0))
16. comments = int(row.get('comments', 0))
17. formatted\_input = f"{instruction\_text} On the basis of ths evaluate the statement TEXT: {text} Just return the scores"
18. response = chat(
19. model="llama3",
20. messages=[{"role": "user", "content": formatted\_input}]
21. )
22. scores = response['message']['content'].split(", ")
23. economicScore = float(scores[0])
24. socialScore = float(scores[1])
25. economicImpact = economicScore \* ((likes // 10) + comments)
26. socialImpact = socialScore \* ((likes // 10) + comments)
27. economicScores.append(economicScore)
28. socialScores.append(socialScore)
29. economicImpacts.append(economicImpact)
30. socialImpacts.append(socialImpact)
31. except Exception as e:
32. print(f"Error processing row {index}: {e}")
33. economicScores.append(0)
34. socialScores.append(0)
35. economicImpacts.append(0)
36. socialImpacts.append(0)
37. self.sentimentDataset['economic\_score'] = economicScores
38. self.sentimentDataset['social\_score'] = socialScores
39. self.sentimentDataset['economic\_impact'] = economicImpacts
40. self.sentimentDataset['social\_impact'] = socialImpacts
41. self.sentimentDataset.to\_csv(self.outputTextsPath, index=False, quoting=csv.QUOTE\_MINIMAL)
42. print("Political scores processing completed.")

Processes socio-economic and political scores based on sentiment analysis.

**Constructor (\_\_init\_\_ Method)**

* **Parameters:**
  + sentimentDataset: Input dataset containing text data and engagement metrics.
  + outputTextsPath: File path for saving processed results.
* **Functionality:**
  + Stores dataset and output file path for later processing.

**Method: processTexts()**

* **Functionality:**
  + Reads socio-economic instruction text from socio-economic-instructions.txt.
  + Iterates over rows of the dataset to evaluate political sentiment.
  + Constructs a formatted prompt combining instructions and text data.
  + Uses an AI chat model (llama3) to obtain economic and social scores.
  + Extracts and processes response values for economic and social impact calculations.
  + Handles missing engagement metrics (likes, comments) by assigning default values.
  + Appends computed scores and impacts to the dataset.
  + Saves the updated dataset to outputTextsPath.

**Class: ImpactProcessor**

1. class ImpactProcessor:
2. def \_\_init\_\_(self, inputFilePath, outputFilePath, modelContextFile, parameters, metricName):
3. self.inputFilePath = inputFilePath
4. self.outputFilePath = outputFilePath
5. self.modelContextFile = modelContextFile
6. self.parameters = parameters
7. self.metricName = metricName
8. def processParameters(self):
9. with open(self.modelContextFile, "r") as file:
10. context = file.read().replace('\n', ' ')
11. try:
12. existingData = pd.read\_csv(self.inputFilePath, quotechar='"')
13. except Exception as e:
14. print(f"Error loading input file: {e}")
15. return
16. processedData = []
17. for index, row in existingData.iterrows():
18. userInput = row['text']
19. likes = int(row.get("likes", 0))
20. comments = int(row.get("comments", 0))
21. if not userInput:
22. print(f"Skipping empty row at index {index}.")
23. continue
24. try:
25. prompt = f"{context} Text: {userInput}"
26. response = chat(
27. model="llama3",
28. messages=[{"role": "user", "content": prompt}]
29. )
30. response\_lines = response["message"]["content"].split("\n")
31. paramScores = {param: 0 for param in self.parameters}
32. for line in response\_lines:
33. if ":" in line:
34. try:
35. param, value = line.split(":", 1)
36. param = param.strip()
37. value = float(value.strip())
38. param = re.sub(r'[^a-zA-Z\s]', '', param)
39. if param in paramScores or param.lower() in paramScores or param.upper() in paramScores or param.title() in paramScores:
40. paramScores[param] = value
41. except ValueError:
42. print(f"Skipping malformed line: {line}")
43. for param in self.parameters:
44. impactCol = f"impact\_{param}"
45. paramScores[impactCol] = paramScores[param] \* ((likes // 10) + comments)
46. paramScores.update({"text": userInput, "likes": likes, "comments": comments})
47. processedData.append(paramScores)
48. except Exception as e:
49. print(f"Error processing row {index}: {e}")
50. processedData.append({\*\*{param: 0 for param in self.parameters},
51. \*\*{f"impact\_{param}": 0 for param in self.parameters},
52. "text": userInput, "likes": likes, "comments": comments})
53. try:
54. newData = pd.DataFrame(processedData)
55. metricColumns = self.parameters
56. impactColumns = [f"impact\_{param}" for param in self.parameters]
57. # Dynamically calculate metric indices
58. newData[f"{self.metricName}\_index"] = newData[metricColumns].sum(axis=1)
59. newData[f"{self.metricName}\_impact"] = newData[impactColumns].sum(axis=1)
60. newData = newData.drop(columns=["likes", "comments"], errors="ignore")
61. mergedData = pd.merge(existingData, newData, on="text", how="left")
62. mergedData.to\_csv(self.outputFilePath, index=False, quoting=csv.QUOTE\_MINIMAL)
63. print(f"Processed data saved to {self.outputFilePath}.")
64. except Exception as e:
65. print(f"Error saving output file: {e}")

Processes text data to compute impact scores based on predefined parameters.

**Constructor (\_\_init\_\_ Method)**

* **Parameters:**
  + inputFilePath: Path to input CSV file.
  + outputFilePath: Path to save processed results.
  + modelContextFile: File containing context for AI model evaluation.
  + parameters: List of parameters for impact calculation.
  + metricName: Name of the metric to be computed.
* **Functionality:**
  + Stores file paths, parameters, and metric names for processing.

**Method: processParameters()**

* **Functionality:**
  + Reads model context from modelContextFile.
  + Loads text data from inputFilePath.
  + Iterates over text entries and constructs prompts for AI evaluation.
  + Uses AI model (llama3) to compute parameter scores based on text content.
  + Parses model response and extracts parameter scores.
  + Calculates impact scores based on engagement metrics (likes, comments).
  + Saves processed data to outputFilePath with computed impact values.
  + Merges new data with original dataset to retain additional columns.
  + Handles errors in file reading, AI response parsing, and data saving.

**Class: DatabaseHandler**

* class DatabaseHandler:
* def \_\_init\_\_(self, host, user, password, database):
* self.host = host
* self.user = user
* self.password = password
* self.database = database
* self.conn = None
* def connect(self):
* try:
* self.conn = pymysql.connect(
* host=self.host,
* user=self.user,
* password=self.password,
* )
* cursor = self.conn.cursor()
* cursor.execute(f"DROP DATABASE IF EXISTS {self.database};")
* cursor.execute(f"CREATE DATABASE {self.database};")
* self.conn.select\_db(f"{self.database}")
* print(f"Connected to MySQL database: {self.database}")
* except pymysql.MySQLError as e:
* print(f"Error connecting to MySQL database: {e}")
* self.conn = None
* def create\_tables(self, texts\_columns, words\_columns):
* if not self.conn:
* print("No connection. Call connect() first.")
* return
* cursor = self.conn.cursor()
* # Create texts table
* create\_texts\_table = texts\_columns
* cursor.execute(create\_texts\_table)
* # Create words table
* create\_words\_table = words\_columns
* cursor.execute(create\_words\_table)
* self.conn.commit()
* print("Tables created successfully!")
* def insert\_data(self, csv\_file, table\_name):
* if not self.conn:
* print("No connection. Call connect() first.")
* return
* try:
* df = pd.read\_csv(csv\_file, quotechar='"')
* df.fillna(0, inplace=True)
* cursor = self.conn.cursor()
* for \_, row in df.iterrows():
* placeholders = ', '.join(['%s'] \* len(row))
* columns = ', '.join(row.index)
* query = f"INSERT INTO {table\_name} ({columns}) VALUES ({placeholders})"
* cursor.execute(query, tuple(row))
* self.conn.commit()
* print(f"Data from '{csv\_file}' inserted into '{table\_name}'.")
* except Exception as e:
* print(f"Error inserting data into {table\_name}: {e}")
* def close(self):
* if self.conn:
* self.conn.close()
* print("MySQL connection closed.")

Handles MySQL database operations, including connection, table creation, data insertion, and closure.

**Constructor (\_\_init\_\_ Method)**

* **Parameters:**
  + host: MySQL server host.
  + user: MySQL username.
  + password: Password for authentication.
  + database: Name of the database to be managed.
* **Functionality:**
  + Stores credentials and database name.
  + Initializes conn as None.

**Method: connect()**

* **Functionality:**
  + Establishes a connection to the MySQL server.
  + Drops and recreates the specified database.
  + Selects the newly created database for operations.
  + Prints a success message if connected.
  + Handles MySQL connection errors and sets conn to None on failure.

**Method: create\_tables(texts\_columns, words\_columns)**

* **Functionality:**
  + Ensures a database connection exists.
  + Creates the texts and words tables based on provided SQL schema definitions.
  + Commits the table creation to the database.
  + Prints a success message.

**Method: insert\_data(csv\_file, table\_name)**

* **Functionality:**
  + Ensures a database connection exists.
  + Reads data from a CSV file into a Pandas DataFrame.
  + Fills missing values with 0.
  + Constructs and executes SQL INSERT queries for each row in the DataFrame.
  + Commits the inserted data to the specified table.
  + Prints a success message.
  + Handles errors during data insertion.

**Method: close()**

* **Functionality:**
  + Closes the MySQL database connection if it exists.
  + Prints a confirmation message.

**Class: Visualizer**

* class Visualizer:
* def \_\_init\_\_(self, textsDf, wordsDf):
* self.textsDf = textsDf
* self.wordsDf = wordsDf
* def groupAndSummarizeData(self, selection, sortBy):
* if selection in ['words', 'texts']:
* grouped = self.wordsDf.groupby('word', as\_index=False).sum() if selection == 'words' else self.textsDf.groupby('text', as\_index=False).sum()
* grouped = grouped[[selection[:-1], sortBy]].sort\_values(by=sortBy, ascending=False)
* elif selection in ['agegroup', 'country', 'time', 'userid']:
* if selection in self.textsDf.columns:
* grouped = self.textsDf.groupby(selection, as\_index=False).sum()
* grouped = grouped[[selection, sortBy]].sort\_values(by=sortBy, ascending=False)
* else:
* print(f"Column '{selection}' not found in the dataset.")
* return pd.DataFrame()
* else:
* print(f"Selection '{selection}' not found in data columns.")
* return pd.DataFrame()
* return grouped
* def sliceData(self, df, threshold, countVal):
* if threshold == 'Highest':
* return df.head(countVal)
* elif threshold == 'Lowest':
* return df.tail(countVal)
* elif threshold == 'Extremes':
* halfN = countVal // 2
* topN = halfN + (countVal % 2)
* dfTop = df.head(topN)
* dfBottom = df.tail(halfN)
* return pd.concat([dfTop, dfBottom])
* else:
* print(f"Unknown threshold: {threshold}")
* return pd.DataFrame()
* def plotData(self, df, column, graphType, selection, sortBy, actualCount):
* plt.figure(figsize=(10, 6))
* try:
* if graphType == 'Bar':
* plt.bar(df.iloc[:, 0].astype(str), df[column])
* elif graphType == 'Line':
* plt.plot(df.iloc[:, 0].astype(str), df[column], marker='o')
* elif graphType == 'Pie':
* plt.pie(df[column], labels=df.iloc[:, 0].astype(str), autopct='%1.1f%%')
* else:
* print(f"Graph type '{graphType}' not recognized. Defaulting to Bar chart.")
* plt.bar(df.iloc[:, 0].astype(str), df[column])
* plt.xticks(rotation=45, ha='right')
* plotTitle = f"Top {actualCount} {selection} by {sortBy} ({column})"
* plt.title(plotTitle)
* if graphType != 'Pie':
* plt.ylabel(column.capitalize())
* plt.xlabel(selection.capitalize() if selection not in ['words', 'texts'] else selection.capitalize())
* plt.tight\_layout()
* plt.show()
* except Exception as e:
* print(f"Error while plotting data: {e}")

Handles data grouping, filtering, and visualization for text and word-based datasets.

**Constructor (\_\_init\_\_ Method)**

* **Parameters:**
  + textsDf: DataFrame containing processed text-level data.
  + wordsDf: DataFrame containing processed word-level data.
* **Functionality:**
  + Stores datasets for use in visualization and analysis.

**Method: groupAndSummarizeData(selection, sortBy)**

* **Functionality:**
  + Groups data based on selection criteria (words, texts, agegroup, country, time, userid).
  + Sorts the data by the specified column (sortBy).
  + Handles missing column errors gracefully.
  + Returns a summarized and sorted DataFrame.

**Method: sliceData(df, threshold, countVal)**

* **Functionality:**
  + Filters data based on a threshold (Highest, Lowest, Extremes).
  + Returns a subset of data with the required count (countVal).
  + Handles edge cases for unknown thresholds.

**Method: plotData(df, column, graphType, selection, sortBy, actualCount)**

* **Functionality:**
  + Generates a visualization of the grouped data.
  + Supports Bar, Line, and Pie chart types.
  + Handles label formatting and rotation for better readability.
  + Sets appropriate titles and axis labels.
  + Displays the plot using Matplotlib.
  + Catches and reports plotting errors.

**Class: GUIHandler**

* class GUIHandler:
* def \_\_init\_\_(self, visualizer):
* self.visualizer = visualizer
* self.selection = None
* self.sortBy = None
* self.threshold = None
* self.countVal = None
* self.graphType = None
* def loadDynamicColumns(self, selection):
* if selection in ['texts', 'words']:
* filePath = 'texts.csv' if selection == 'texts' else 'words.csv'
* elif selection in ['agegroup', 'country', 'time', 'userid']:
* filePath = 'texts.csv'
* else:
* print(f"Dynamic columns not applicable for selection: {selection}")
* return []
* try:
* data = pd.read\_csv(filePath, quotechar='"')
* return list(data.columns)
* except Exception as e:
* print(f"Error loading columns for {selection}: {e}")
* return []
* def updateSortByOptions(self, event, sortByMenu, selectionVar):
* selection = selectionVar.get()
* columns = self.loadDynamicColumns(selection)
* if columns:
* sortByMenu["values"] = columns
* else:
* sortByMenu["values"] = []
* def onSubmit(self, selectionVar, sortByVar, thresholdVar, countVar, graphTypeVar):
* self.selection = selectionVar.get()
* self.sortBy = sortByVar.get()
* self.threshold = thresholdVar.get()
* self.graphType = graphTypeVar.get()
* try:
* self.countVal = int(countVar.get())
* except ValueError:
* print("Invalid count value. Please enter a valid number.")
* return
* df = self.visualizer.groupAndSummarizeData(self.selection, self.sortBy)
* if df is None or df.empty:
* print("No data available for the given selection and sortBy.")
* return
* dfSliced = self.visualizer.sliceData(df, self.threshold, self.countVal)
* actualCount = len(dfSliced)
* if actualCount > 0:
* self.visualizer.plotData(dfSliced, self.sortBy, self.graphType, self.selection, self.sortBy, actualCount)
* def launchGUI(self):
* root = tk.Tk()
* root.title("Data Selection")
* root.resizable(False, False)
* selectionVar = tk.StringVar(value='texts')
* sortByVar = tk.StringVar(value='impact')
* thresholdVar = tk.StringVar(value='Highest')
* countVar = tk.StringVar(value='10')
* graphTypeVar = tk.StringVar(value='Bar')
* tk.Label(root, text="Select Data Type:").pack()
* dataTypeMenu = ttk.Combobox(root, textvariable=selectionVar, values=['agegroup', 'country', 'texts', 'time', 'userid', 'words'], state="readonly")
* dataTypeMenu.pack()
* tk.Label(root, text="Sort By:").pack()
* sortByMenu = ttk.Combobox(root, textvariable=sortByVar, values=['impact', 'tone', 'likes', 'comments', 'frequency'], state="readonly")
* sortByMenu.pack()
* dataTypeMenu.bind("<<ComboboxSelected>>", lambda event: self.updateSortByOptions(event, sortByMenu, selectionVar))
* tk.Label(root, text="Threshold:").pack()
* thresholdMenu = ttk.Combobox(root, textvariable=thresholdVar, values=['Highest', 'Lowest', 'Extremes'], state="readonly")
* thresholdMenu.pack()
* tk.Label(root, text="Number of Items to Display:").pack()
* countEntry = ttk.Entry(root, textvariable=countVar)
* countEntry.pack()
* tk.Label(root, text="Select Graph Type:").pack()
* graphTypeMenu = ttk.Combobox(
* root,
* textvariable=graphTypeVar,
* values=['Bar', 'Line', 'Pie'],
* state="readonly"
* )
* graphTypeMenu.pack()
* tk.Button(root, text="Submit", command=lambda: self.onSubmit(selectionVar, sortByVar, thresholdVar, countVar, graphTypeVar)).pack()
* FileHandler.cleanTempFiles()
* root.mainloop()

Handles graphical user interface interactions for selecting and visualizing data.

**Constructor (\_\_init\_\_ Method)**

* **Parameters:**
  + visualizer: An instance of the Visualizer class used for data visualization.
* **Functionality:**
  + Stores the visualizer instance.
  + Initializes selection parameters (selection, sortBy, threshold, countVal, graphType).

**Method: loadDynamicColumns(selection)**

* **Functionality:**
  + Determines the appropriate CSV file based on the selected data type (texts or words).
  + Loads column names from the selected CSV file.
  + Returns a list of available columns or an empty list in case of an error.

**Method: updateSortByOptions(event, sortByMenu, selectionVar)**

* **Functionality:**
  + Updates the sortBy dropdown menu dynamically based on the selected data type.
  + Calls loadDynamicColumns() to retrieve relevant column names.

**Method: onSubmit(selectionVar, sortByVar, thresholdVar, countVar, graphTypeVar)**

* **Functionality:**
  + Captures user input values for selection parameters.
  + Converts countVar to an integer and handles errors.
  + Calls groupAndSummarizeData() from Visualizer to process the dataset.
  + Calls sliceData() to filter the dataset based on the threshold.
  + Calls plotData() to generate a visualization if valid data is available.

**Method: launchGUI()**

* **Functionality:**
  + Creates a Tkinter-based graphical interface with dropdowns and input fields for selecting data visualization preferences.
  + Binds user selection events to update dropdown menus dynamically.
  + Provides a submit button that triggers onSubmit() for processing.
  + Calls FileHandler.cleanTempFiles() to remove temporary files.
  + Runs the Tkinter main event loop to display the interface.

**Function: main()**

* def main():
* try:
* FileHandler.cleanData("sentiment\_dataset.csv", "cleaned\_dataset.csv")  # Clean data
* except FileNotFoundError:
* print("Error: 'sentiment\_dataset.csv' not found. Ensure the file is in the correct directory.")
* return
* except pd.errors.EmptyDataError:
* print("Error: 'cleaned\_dataset.csv' is empty. Provide a valid dataset.")
* return
* except Exception as e:
* print(f"Unexpected error loading dataset: {e}")
* return
* try:
* allStopwords = DataProcessor.getAllStopwords()
* except Exception as e:
* print(f"Error loading stopwords: {e}")
* return
* try:
* bestGpu = DataProcessor.getBestGpu()
* if bestGpu != -1:
* device = torch.device(f"cuda:{bestGpu}")
* print(f"Using GPU: {bestGpu}")
* else:
* device = torch.device("cuda" if torch.cuda.is\_available() else "cpu")
* print(f"Using device: {device}")
* except Exception as e:
* print(f"Error initializing device: {e}")
* return
* try:
* hf\_dataset = Dataset.from\_csv('cleaned\_dataset.csv')
* except FileNotFoundError:
* print("Error: 'cleaned\_dataset.csv' not found. Ensure the file is in the correct directory.")
* return
* except pd.errors.EmptyDataError:
* print("Error: 'cleaned\_dataset.csv' is empty. Provide a valid dataset.")
* return
* except Exception as e:
* print(f"Unexpected error loading dataset: {e}")
* return
* apiModels = [
* "j-hartmann/emotion-English-distilroberta-base",
* "bhadresh-savani/bert-base-go-emotion",
* "monologg/bert-base-cased-goemotions-original",
* ]
* try:
* emotions = []
* dataProcessor = DataProcessor(hf\_dataset, device, apiModels, emotions)
* fileHandler = FileHandler(hf\_dataset, allStopwords, emotions, dataProcessor)
* except Exception as e:
* print(f"Error initializing DataProcessor or FileHandler: {e}")
* return
* # Check and create necessary files
* if not os.path.exists("texts.csv") or not os.path.exists("words.csv"):
* try:
* print("Required files missing. Generating all necessary files...")
* fileHandler.createTextsCsv(dataProcessor.calculateToneImpact, dataProcessor)
* complexEmotionProcessor = ComplexEmotionProcessor(
* tempTextsPath="temp1texts.csv",
* outputTextsPath="temp2texts.csv",
* filteredEmotions = {
* "compassion": ["caring", "sadness"],
* "elation": ["joy", "excitement"],
* "affection": ["love", "approval"],
* "contentment": ["relief", "joy"],
* "playfulness": ["amusement", "joy"],
* "empathy": ["caring", "sadness"],
* "warmth": ["love", "caring"],
* "frustration": ["annoyance", "anger"],
* "shame": ["embarrassment", "disapproval"],
* "regret": ["remorse", "sadness"],
* "guilt": ["remorse", "grief"],
* "loneliness": ["sadness", "neutral"],
* "disdain": ["disapproval", "disgust"],
* "curiosity": ["confusion", "optimism"],
* "skepticism": ["confusion", "realization"],
* "uncertainty": ["confusion", "neutral"],
* "triumph": ["pride", "joy"],
* "reluctance": ["disapproval", "desire"],
* "apathy": ["neutral", "sadness"],
* "nostalgia": ["joy", "sadness"],
* "intrigue": ["curiosity", "desire"],
* "hopefulness": ["optimism", "joy"],
* "bliss": ["joy", "relief"],
* "fascination": ["curiosity", "admiration"],
* "passion": ["love", "desire"],
* "hopelessness": ["sadness", "disappointment"],
* "bitterness": ["sadness", "anger"],
* "gratefulness": ["gratitude", "relief"],
* "agitation": ["annoyance", "nervousness"],
* "yearning": ["desire", "sadness"],
* "sorrow": ["grief", "sadness"],
* "delight": ["joy", "amusement"],
* "trepidation": ["fear", "nervousness"],
* "amazement": ["surprise", "joy", "admiration"],
* "complacency": ["neutral", "relief", "approval"],
* "disillusionment": ["sadness", "disappointment", "realization"],
* "zeal": ["excitement", "pride", "desire"],
* "reverence": ["admiration", "gratitude"],
* "infatuation": ["love", "desire", "admiration"],
* "composure": ["relief", "neutral", "caring"],
* "ecstasy": ["joy", "excitement", "love"],
* "anticipation": ["excitement", "optimism", "curiosity"],
* "resignation": ["sadness", "relief"],
* "hostility": ["anger", "disgust", "annoyance"],
* "disorientation": ["confusion", "fear", "surprise"],
* "compunction": ["remorse", "sadness", "grief"],
* "humility": ["gratitude", "relief", "approval"],
* "serenity": ["joy", "relief", "caring"],
* "reconciliation": ["relief", "love", "gratitude"],
* "alienation": ["sadness", "disgust", "disapproval"],
* "exultation": ["pride", "joy", "excitement"],
* "affirmation": ["approval", "optimism", "pride"],
* "serendipity": ["joy", "surprise", "relief"],
* "acceptance": ["relief", "approval", "caring"],
* "resentment": ["sadness", "anger", "disapproval"],
* "cheerfulness": ["joy", "amusement", "optimism"],
* "apprehension": ["fear", "nervousness", "curiosity"],
* "eagerness": ["excitement", "curiosity"],
* "clarity": ["relief", "realization", "caring"],
* "hesitation": ["fear", "nervousness", "confusion"],
* "grievance": ["anger", "sadness", "disappointment"],
* "outrage": ["anger", "disapproval", "disgust"],
* "pity": ["sadness", "caring"],
* "shock": ["surprise", "fear", "disgust"],
* "satisfaction": ["relief", "joy", "approval"]
* }
* )
* complexEmotionProcessor.processTexts()
* sentimentDataset = pd.read\_csv("temp2texts.csv", quotechar='"')
* politicalScoreProcessor = PoliticalScoreProcessor(
* sentimentDataset=sentimentDataset,
* outputTextsPath="temp3texts.csv"
* )
* politicalScoreProcessor.processTexts()
* except Exception as e:
* print(f"Error during file creation: {e}")
* return
* else:
* print("Required files found. Skipping file creation.")
* positiveEmotions = [
* "joy", "approval", "admiration", "optimism", "caring", "relief", "gratitude", "amusement", "pride",
* "excitement", "desire", "curiosity", "emotion\_compassion", "emotion\_elation", "emotion\_affection",
* "emotion\_contentment", "emotion\_playfulness", "emotion\_empathy", "emotion\_warmth", "emotion\_triumph",
* "emotion\_nostalgia", "emotion\_hopefulness", "emotion\_bliss", "emotion\_fascination", "emotion\_passion",
* "emotion\_delight", "emotion\_amazement", "emotion\_zeal", "emotion\_reverence", "emotion\_infatuation",
* "emotion\_composure", "emotion\_ecstasy", "emotion\_anticipation", "emotion\_serendipity", "emotion\_acceptance",
* "emotion\_cheerfulness", "emotion\_eagerness", "emotion\_clarity", "emotion\_gratefulness",
* "emotion\_joy", "emotion\_approval", "emotion\_excitement", "emotion\_admiration", "emotion\_caring",
* "emotion\_amusement", "emotion\_gratitude", "emotion\_optimism", "emotion\_pride", "emotion\_relief"
* ]
* negativeEmotions = [
* "surprise", "sadness", "neutral", "fear", "anger", "disgust", "realization", "disapproval", "annoyance",
* "disappointment", "confusion", "nervousness", "embarrassment", "remorse", "love", "grief",
* "emotion\_frustration", "emotion\_shame", "emotion\_regret", "emotion\_guilt", "emotion\_loneliness",
* "emotion\_disdain", "emotion\_skepticism", "emotion\_uncertainty", "emotion\_reluctance", "emotion\_apathy",
* "emotion\_bitterness", "emotion\_agitation", "emotion\_yearning", "emotion\_sorrow", "emotion\_trepidation",
* "emotion\_complacency", "emotion\_disillusionment", "emotion\_resignation", "emotion\_hostility",
* "emotion\_disorientation", "emotion\_compunction", "emotion\_humility", "emotion\_serenity", "emotion\_reconciliation",
* "emotion\_alienation", "emotion\_exultation", "emotion\_affirmation", "emotion\_resentment", "emotion\_hesitation",
* "emotion\_grievance", "emotion\_outrage", "emotion\_pity", "emotion\_shock", "emotion\_satisfaction",
* "emotion\_surprise", "emotion\_sadness", "emotion\_fear", "emotion\_anger", "emotion\_disgust",
* "emotion\_disapproval", "emotion\_disappointment", "emotion\_confusion", "emotion\_nervousness",
* "emotion\_embarrassment", "emotion\_grief", "emotion\_remorse"
* ]
* try:
* print("Applying tone adjustments for texts...")
* textsDf = pd.read\_csv("temp3texts.csv", quotechar='"')
* toneAdjuster = ToneAdjuster(positiveEmotions, negativeEmotions)
* adjustedTextsDf = toneAdjuster.adjustToneAndImpact(textsDf)
* adjustedTextsDf.to\_csv("temp3texts.csv", index=False, quoting=csv.QUOTE\_MINIMAL)
* print("Adjusted texts saved to 'temp3texts.csv'.")
* except Exception as e:
* print(f"Error applying tone adjustments for texts: {e}")
* try:
* print("Processing flagging parameters and impacts...")
* parameterProcessor = ImpactProcessor(
* inputFilePath="temp3texts.csv",
* outputFilePath="temp4texts.csv",
* modelContextFile="Flagging Prompts.txt",
* parameters=[
* 'Ableist', 'Abusive', 'Ageist', 'Aggressive', 'Alienating', 'Antisemitic', 'Belittling',
* 'Belligerent', 'Bullying', 'Caustic', 'Classist', 'Condescending', 'Containing\_slurs',
* 'Contemptful', 'Defamatory', 'Degrading', 'Demeaning', 'Demoralizing', 'Derisive',
* 'Derogatory', 'Despising', 'Destructive', 'Discriminatory', 'Disparaging', 'Disturbing',
* 'Enraging', 'Ethnocentric', 'Exclusionary', 'Harassing', 'Harmful', 'Hatespeech',
* 'Homophobic', 'Hostile', 'Hurtful', 'Incendiary', 'Inflammatory', 'Insulting',
* 'Intimidating', 'Intolerable', 'Intolerant', 'Islamophobic', 'Malicious', 'Marginalizing',
* 'Misogynistic', 'Mocking', 'Dehumanizing', 'Objectifying', 'Segregating', 'Nasty', 'Obscene',
* 'Offensive', 'Oppressive', 'Overbearing', 'Pejorative', 'Prejudiced', 'Profane', 'Racist',
* 'Sarcastic', 'Scornful', 'Sexist', 'Slanderous', 'Spiteful', 'Threatening', 'Toxic',
* 'Transphobic', 'Traumatizing', 'Vindictive', 'Vulgar', 'Xenophobic', 'Manipulative',
* 'Exploitative', 'Gaslighting', 'Patronizing', 'Overcritical', 'Fearmongering', 'Shaming',
* 'Pathologizing', "Trolling", "Cyberbullying", "Dogpiling", "Sealioning", "Doxxing",
* "Brigading", "Spamming", "Clickbaiting", "Misinformation", "Disinformation",
* "Profanity", "Alarmist", "Hysterical", "Vindictive", "Shocking",
* "Overgeneralizing", "Narcissistic"
* ],
* metricName="toxicity"
* )
* parameterProcessor.processParameters()
* print("Flagging parameters processed and saved to 'temp4texts.csv'.")
* except Exception as e:
* print(f"Error processing flagging parameters: {e}")
* try:
* print("Processing psychological parameters and impacts...")
* mentalHealthProcessor = ImpactProcessor(
* inputFilePath="temp4texts.csv",
* outputFilePath="temp5texts.csv",
* modelContextFile="mental health prompts.txt",
* parameters=[
* 'Abandoned', 'Afraid', 'Alienated', 'Alone', 'Anguished', 'Annoyed', 'Anxious',
* 'Apathetic', 'Apologetic', 'Apprehensive', 'Ashamed', 'Awkward', 'Bitter', 'Blameworthy',
* 'Burned\_Out', 'Concerned', 'Dejected', 'Demoralized', 'Despondent', 'Detached',
* 'Disconnected', 'Disheartened', 'Dissociative', 'Distraught', 'Doubtful', 'Drained',
* 'Dread', 'Edgy', 'Embarrassed', 'Emptiness', 'Enraged', 'Excluded', 'Exposed', 'Fatigued',
* 'Fearful', 'Forsaken', 'Frustrated', 'Furious', 'Gloomy', 'Heartbroken', 'Helpless',
* 'Hesitant', 'Hopeless', 'Hypervigilant', 'Indifferent', 'Insecure', 'Irritable',
* 'Isolated', 'Judged', 'Lethargic', 'Longing', 'Lost', 'Melancholy', 'Miserable', 'Misunderstood',
* 'Mourning', 'Nervous', 'Numb', 'Overwhelmed', 'Panicked', 'Paranoid', 'Pressured',
* 'Regretful', 'Remorseful', 'Resentful', 'Restless', 'Sad', 'Sarcasm', 'Scared', 'Secluded',
* 'Self\_Critical', 'Shaky', 'Shy', 'Sorrowful', 'Startled', 'Stressed', 'Tense', 'Terrified',
* 'Tired', 'Triggered', 'Troubled', 'Uncertain', 'Uneasy', 'Unloved', 'Unmotivated',
* 'Unworthy', 'Vulnerable', 'Withdrawn', 'Worried', 'Worthless', 'Suicidal', 'Self\_harm'
* ],
* metricName="distress"
* )
* mentalHealthProcessor.processParameters()
* print("Psychological parameters processed and saved to 'temp5texts.csv'.")
* except Exception as e:
* print(f"Error processing psychological parameters: {e}")
* try:
* print("Processing healing emotions...")
* healingEmotionProcessor = ImpactProcessor(
* inputFilePath="temp5texts.csv",
* outputFilePath="texts.csv",
* modelContextFile="healing prompts.txt",
* parameters=[
* "Calming", "Relaxed", "Safe", "Motivated",
* "Empowered", "Peaceful", "Confident", "Trusting",
* "Comforted", "Reassured", "Inspired", "Nurtured",
* "Understanding", "Serene", "Fulfilled", "Energized",
* "Harmonious", "Appreciative", "Openness", "Sociable",
* "Gracious", "Altruistic", "Reflective","Enthusiastic","Adventurous"
* ],
* metricName="healing"
* )
* healingEmotionProcessor.processParameters()
* print("Healing emotions processed and saved to 'texts.csv'.")
* except Exception as e:
* print(f"Error processing healing emotions: {e}")
* fileHandler.createWordsCsv("texts.csv", "words.csv")
* try:
* # Initialize MySQL DatabaseHandler
* db\_handler = DatabaseHandler(
* host='localhost',  # Replace with your MySQL host, e.g., 'localhost'
* user='root',  # Replace with your MySQL username
* password='1Anurag2Basistha',  # Replace with your MySQL password
* database='SentimentAnalysis'  # Replace with your MySQL database name
* )
* # Connect to the MySQL database
* db\_handler.connect()
* try:
* with open("create\_texts\_database.txt", 'r') as file:
* texts\_columns = file.read().replace('\n', ' ')
* except Exception as e:
* print(f"Error reading file: {e}")
* try:
* with open("create\_words\_database.txt", 'r') as file:
* words\_columns = file.read().replace('\n', ' ')
* except Exception as e:
* print(f"Error reading file: {e}")
* # Create tables
* db\_handler.create\_tables(texts\_columns, words\_columns)
* # Insert data into tables from CSV files
* db\_handler.insert\_data('texts.csv', 'texts')
* db\_handler.insert\_data('words.csv', 'words')
* db\_handler.close()
* except Exception as e:
* print(f"Error in database handling: {e}")
* try:
* textsDf = pd.read\_csv("texts.csv", quotechar='"')
* wordsDf = pd.read\_csv("words.csv", quotechar='"')
* visualizer = Visualizer(textsDf, wordsDf)
* except Exception as e:
* print(f"Error loading data for Visualizer: {e}")
* return
* try:
* guiHandler = GUIHandler(visualizer)
* guiHandler.launchGUI()
* except Exception as e:
* print(f"Error launching GUI: {e}")
* return
* print("Pipeline completed successfully.")

Handles the entire processing pipeline, including data cleaning, analysis, impact calculation, database handling, and GUI launching.

**Step 1: Data Cleaning**

* Calls FileHandler.cleanData() to clean sentiment\_dataset.csv and save it as cleaned\_dataset.csv.
* Handles file errors such as missing or empty datasets.

**Step 2: Stopwords and Device Initialization**

* Retrieves stopwords using DataProcessor.getAllStopwords().
* Identifies the best available GPU using DataProcessor.getBestGpu().

**Step 3: Dataset Preparation**

* Loads cleaned\_dataset.csv into a Hugging Face Dataset.
* Initializes API models for text classification.
* Instantiates DataProcessor and FileHandler for further processing.

**Step 4: File Generation**

* Checks if texts.csv and words.csv exist; if not, regenerates them.
* Calls fileHandler.createTextsCsv() to process text data.
* Calls ComplexEmotionProcessor to compute complex emotions and their impacts.
* Calls PoliticalScoreProcessor to analyze political sentiment impact.

**Step 5: Tone Adjustment**

* Reads temp3texts.csv.
* Calls ToneAdjuster to refine sentiment scores based on positive and negative emotions.
* Saves adjusted results to temp3texts.csv.

**Step 6: Flagging Toxicity Parameters**

* Calls ImpactProcessor to evaluate texts for potentially harmful content.
* Saves results in temp4texts.csv.

**Step 7: Psychological Impact Processing**

* Calls ImpactProcessor to analyze psychological distress indicators.
* Saves results in temp5texts.csv.

**Step 8: Healing Emotion Analysis**

* Calls ImpactProcessor to assess texts for positive and healing emotions.
* Saves final processed text dataset as texts.csv.
* Calls fileHandler.createWordsCsv() to generate words.csv.

**Step 9: Database Handling**

* Initializes DatabaseHandler to manage MySQL operations.
* Reads and executes table creation scripts.
* Inserts processed text and word data into MySQL.
* Closes database connection.

**Step 10: Data Visualization & GUI**

* Loads texts.csv and words.csv into Pandas DataFrames.
* Instantiates Visualizer for graphical data representation.
* Launches the GUI using GUIHandler.launchGUI().

**Final Step: Completion Message**

* Prints confirmation that the entire processing pipeline has successfully completed.