Understanding the Movie Sentiment Analysis Script

1. Introduction

Purpose: This document provides a detailed breakdown of the provided Python script, designed to perform sentiment analysis on movie reviews. The script implements and allows comparison between three distinct methodologies: a lexicon-based approach, a machine learning approach using Principal Component Analysis (PCA) combined with Logistic Regression (LR), and a standard supervised machine learning approach using Term Frequency-Inverse Document Frequency (TF-IDF) features with Logistic Regression.
Report Objective: The primary goal of this report is to equip developers, especially those new to this codebase, with a thorough understanding of its structure, components, logic, and operational workflow. This knowledge is intended to facilitate easier maintenance, modification, and extension of the script's functionalities.
Key Features Overview: The script encompasses several key functionalities:
- Sophisticated text preprocessing tailored for sentiment analysis.
- A custom negation handling mechanism to improve accuracy.
- External configuration management via a JSON file (settings.json).
- Complete pipelines for training and testing two different types of machine learning models (PCA+LR and TF-IDF+LR).
- Comprehensive model evaluation using standard metrics.
- An interactive command-line menu for user operation, enhanced with GUI dialogs for file/directory selection.

Architectural Insight: It is important to recognize that this script is not a single sentiment analyzer but rather a framework offering three different techniques. The presence of the MovieSentimentAnalyzer class (handling the lexicon method and housing the PCA+LR logic) alongside separate functions for the directory-based TF-IDF+LR approach (train_directory_supervised, test_directory_supervised) clearly demonstrates this multi-pronged design. This architecture likely stems from a need to compare these methods or provide flexibility depending on the available data (e.g., presence of a good lexicon vs. labeled training text vs. unlabeled data). Understanding this multi-approach structure is fundamental for navigating and modifying the codebase effectively.

2. Getting Started: Setup and Dependencies

Essential Libraries: The script leverages several standard and specialized Python libraries to achieve its functionality. The core dependencies include:
- os: For interacting with the operating system, primarily for file path manipulation (checking existence, joining paths, listing directories).
- re: For regular expression operations, used here for cleaning text data (removing punctuation).
- string: Provides string constants (like punctuation).
- json: Used for reading from and writing to the settings.json configuration file.

- joblib: For efficiently saving and loading Python objects, specifically the trained machine learning models and vectorizers, enabling persistence.
- numpy: The fundamental package for numerical computation in Python, essential for handling the numerical feature vectors used in the machine learning models.
- pandas: A powerful data manipulation library, used here specifically for reading the sentiment lexicon from a CSV file.
- nltk (Natural Language Toolkit): A comprehensive library for various natural language processing tasks. This script uses it for accessing lists of stopwords, lemmatizing words, and potentially tokenization (though basic splitting is used in preprocessing).
- sklearn (Scikit-learn): A widely used machine learning library in Python. It provides implementations for PCA, Logistic Regression, TF-IDF vectorization, cross-validation, grid search, and various evaluation metrics.
- tkinter: Python's standard GUI (Graphical User Interface) toolkit. It's used here *not* for the main application interface (which is CLI-based) but specifically to invoke native file and directory selection dialog boxes, making it easier for users to specify input paths without manual editing.
- typing: Provides support for type hints, improving code readability and allowing for static analysis.

Library Roles: The following table summarizes the primary role of the key external libraries within this script:

Library Name	Primary Use in Script
nltk	Provides linguistic resources (stopwords, WordNet lemmatizer) and NLP functionalities.
sklearn	Implements ML algorithms (PCA, Logistic Regression), feature extraction (TF-IDF), model tuning (GridSearch, CV), metrics.
pandas	Reads and processes the sentiment lexicon from a CSV file (select_and_load_lexicon).
numpy	Handles numerical arrays and matrices for feature representation and ML computations.
joblib	Saves trained ML models (PCA, LogisticRegression, TfidfVectorizer) and loads them for later use (testing).
json	Manages loading and saving script configuration from/to settings.json.
tkinter	Provides GUI dialogs for selecting input files (lexicon, .feat files, text reviews) and directories (train/test sets).

NLTK Data Downloads: Upon first execution, the script attempts to download necessary data packages from NLTK using nltk.download(...). These packages include:

- stopwords: A list of common English words (like "the", "a", "is") often removed during text preprocessing.
- wordnet: A large lexical database of English, used here by the WordNetLemmatizer.
- punkt, punkt_tab: Pre-trained models used by NLTK for tokenization (splitting text into words or sentences). Although the script uses simple .split() for tokenization in preprocess_text, these downloads might be implicitly required by other NLTK components or were included for potential future use.

These downloads are crucial; the script will likely fail if they cannot be completed successfully due to network issues or permissions. This setup step highlights an external dependency that must be satisfied before the script can fully operate. The inclusion of tkinter for file dialogs within a command-line script is a specific design choice favoring ease of use for path selection over a purely text-based interface.

3. Configuration Management: The settings.json File

Purpose: The script utilizes an external file, settings.json, to manage its configuration parameters. This approach centralizes settings, making the script more flexible and easier to adapt to different environments or datasets without modifying the Python code itself. It stores file paths, operational flags for logging, and parameters for machine learning model training.
Loading (load_settings): When the script starts, the load_settings function is called. It attempts to open and read settings.json. If the file is found, it loads the settings from it. Crucially, if the file does not exist (FileNotFoundError), the function defines a default dictionary containing predefined paths (initially empty strings), logging flags (mostly True by default), and ML parameters (cv_folds, grid_C). This ensures the script can run "out of the box" on its first execution, prompting the user for necessary paths later via GUI dialogs. The log_settings_load flag controls whether the loaded settings are printed to the console.
Saving (save_settings): The save_settings function writes the current state of the settings dictionary back to settings.json in a nicely formatted way (indent=2). This typically happens after a user selects a file or directory path via a GUI dialog, persisting that selection for future runs. The log_settings_save flag controls console output upon saving.
Key Settings: The settings.json file (or the default dictionary if the file is absent) controls various aspects of the script's operation. The following table details the purpose of these settings:

Setting Key	Description	Default Value	Purpose
lexicon_path	Path to the sentiment lexicon CSV file.	""	Input data for Lexicon-Based Analysis.
pca_labeled	Path to the labeled .feat file for PCA+LR training.	""	Input data for PCA+LR Training (labeled features).
pca_unsup	Path to the unsupervised .feat file for PCA fitting.	""	Input data for PCA+LR Training (unlabeled features for PCA).
pca_vocab	Path to the vocabulary (.vocab) file for PCA+LR.	""	Input data for PCA+LR (defines feature indices).
pca_test	Path to the labeled .feat file for PCA+LR testing.	""	Input data for PCA+LR Testing.
dir_train	Path to the directory containing training text files.	""	Input data for Directory-Based Training.
dir_test	Path to the directory containing test text files.	""	Input data for Directory-Based Testing.

Setting Key	Description	Default Value	Purpose
verbose_tokens	Enable/disable detailed token-level scoring output.	False	Debugging flag for Lexicon-Based Analysis scoring.
log_init	Enable/disable initialization messages.	True	Debugging flag for class/script startup.
log_preprocess	Enable/disable detailed text preprocessing step logs.	True	Debugging flag for preprocess_text function.
log_negation	Enable/disable detailed negation handling step logs.	True	Debugging flag for apply_negation_handling function.
log_score_compute	Enable/disable detailed sentiment score computation logs.	True	Debugging flag for compute_sentiment_score function.
log_lexicon_debug	Enable/disable lexicon loading debug messages.	True	Debugging flag for select_and_load_lexicon function.
log_pca_training_debug	Enable/disable PCA+LR training debug messages.	True	Debugging flag for train_pca_lr function.
log_pca_test_debug	Enable/disable PCA+LR testing debug messages.	True	Debugging flag for test_pca_lr function.
log_dir_train_debug	Enable/disable Directory-Based training debug messages.	True	Debugging flag for train_directory_supervised function.
log_dir_test_debug	Enable/disable Directory-Based testing debug messages.	True	Debugging flag for test_directory_supervised function.
log_settings_load	Enable/disable message when settings are loaded.	True	Debugging flag for configuration loading.
log_settings_save	Enable/disable message when settings are saved.	True	Debugging flag for configuration saving.
cv_folds	Number of folds for cross-validation.	5	Parameter for ML model evaluation and tuning (cross_val_score, GridSearchCV).
grid_C	List of 'C' values (regularization strength) for GridSearch.	[0.01, 0.1, 1, 10]	Hyperparameter search space for Logistic Regression tuning.

The extensive use of log_* flags demonstrates a design focused on transparency and debuggability. Developers can enable detailed output for specific modules (preprocessing, negation, scoring, training phases) to diagnose issues without being overwhelmed by irrelevant information. This granular control is highly beneficial for maintenance and understanding potentially complex or error-prone sections of the code. Externalizing configuration avoids hardcoding paths or parameters, a crucial practice for creating maintainable and portable software.

4. Core Text Processing Pipeline (MovieSentimentAnalyzer)

The MovieSentimentAnalyzer class encapsulates the logic for the lexicon-based analysis and also contains the methods for training and testing the PCA+LR model. Its core text processing functions are fundamental to the lexicon approach.

Class Initialization (__init__):

- When an instance of MovieSentimentAnalyzer is created, the __init__ method is called.
- It requires a lexicon (a dictionary mapping words to sentiment scores) as an argument.
- It initializes several components used in processing:
  - self.lexicon: Stores the provided sentiment lexicon.
  - self.stop_words: Loads the standard English stopwords set from NLTK (stopwords.words('english')).
  - self.stemmer: Initializes a PorterStemmer object (Note: This stemmer object is initialized but does not appear to be used in the preprocess_text method as written; lemmatization is used instead).
  - self.lemmatizer: Initializes a WordNetLemmatizer object from NLTK, used to reduce words to their base or dictionary form.
  - self.negation_words: Defines a set of common negation words (e.g., "not", "no", "n't"). These are crucial for the negation handling logic.
  - self.negation_scope: Sets the window size (default 3) for the negation handling logic.
  - self.pca, self.clf: Initialized to None; these will store the trained PCA and Logistic Regression models if the PCA+LR training method is called.
- If the log_init setting is enabled, it prints a confirmation message.

Preprocessing (preprocess_text): This method takes a raw text string and applies a series of cleaning and normalization steps commonly used in NLP:

1. **Lowercasing:** Converts the entire text to lowercase (text.lower()) to ensure case-insensitive matching (e.g., "Good" and "good" are treated the same).
2. **Punctuation Removal:** Uses regular expressions (re.sub) to remove punctuation characters defined in string.punctuation. Notably, the apostrophe (') is explicitly *kept* by replacing it from the punctuation set before removal (string.punctuation.replace("'", "")). This likely prevents contractions like "don't" from becoming "dont", preserving them for the negation handling step.
3. **Tokenization:** Splits the cleaned text into a list of individual words (tokens) based on whitespace (cleaned.split()). This is a simple tokenization method; more advanced techniques might handle hyphens or other cases differently.
4. **Stop Word Removal & Lemmatization:** Iterates through the tokens. A word is kept if it is either in the self.negation_words set OR it is *not* in the self.stop_words set. This is a key step: it removes common words *unless* they are negation words. The kept words are then lemmatized using self.lemmatizer.lemmatize(w), reducing them to their base form (e.g.,

"running" -> "run", "better" -> "good" - depending on context/part-of-speech tagging, which isn't used here, so it might be simpler like "cars" -> "car"). Lemmatization helps group different forms of a word under a single concept.
5. **Logging:** If log_preprocess is enabled, intermediate results (lowercased, punctuation removed, tokenized, lemmatized) are printed, allowing developers to trace the transformation process.

Negation Handling (apply_negation_handling): This method implements a custom rule-based approach to handle negation, which often inverts the sentiment of subsequent words.

- **Goal:** Identify negation words and mark the words immediately following them as "negated".
- **Input:** A list of lemmatized tokens from preprocess_text.
- **Process:**
  - It iterates through the tokens using a while loop and an index i.
  - If the current token tokens[i] is found in the self.negation_words set:
    - It marks the *next* self.negation_scope (default 3) tokens as negated. It does this by appending tuples (token, True) to the result list for each token within the scope.
    - The loop index i is then advanced past the negation word and the scope (i += self.negation_scope + 1) to avoid processing the scoped words again.
  - If the current token is *not* a negation word:
    - It appends the token to the result list marked as *not* negated: (token, False).
    - The loop index i is incremented by 1.
- **Output:** A list of tuples, where each tuple contains a token and a boolean flag indicating whether it should be treated as negated (True) or not (False).
- **Logging:** If log_negation is enabled, the token list before and after processing, along with messages indicating when a negator is found, are printed.

This preprocessing pipeline is fairly standard, but the explicit preservation of negation words during stopword removal, followed by the custom window-based apply_negation_handling logic, represents a specific strategy to address the challenge of negation in sentiment analysis. The fixed negation_scope is a heuristic – a simplified rule – that might not capture all nuances of negation in complex sentences but provides a mechanism beyond simply ignoring negation words.

5. Sentiment Analysis Approach 1: Lexicon-Based Method (MovieSentimentAnalyzer)

This approach determines sentiment directly by looking up word scores in a predefined lexicon, modified by the negation handling logic. It does not involve machine learning model training.

Lexicon Loading (select_and_load_lexicon):

- **Purpose:** This standalone function is responsible for loading the sentiment lexicon data required by the MovieSentimentAnalyzer.
- **Path Handling:** It first checks if lexicon_path is set in settings.json and if the file exists.
  - If the path is valid, it uses it directly (logging the path if log_lexicon_debug is True).
  - If the path is missing or invalid, it uses tkinter.filedialog.askopenfilename to display a standard GUI window, prompting the user to select a CSV file. If a file is selected, its path is stored in settings["lexicon_path"] and saved via save_settings(). If the

user cancels, an empty dictionary is returned.
- **CSV Reading:** It attempts to read the selected CSV file using pandas.read_csv(). Error handling is included for potential reading issues.
- **Format Validation:** It explicitly checks if the loaded DataFrame contains columns named 'stemmed_word' and 'sentiment_score'. If not, it prints an error and returns an empty dictionary. This enforces the required input format.
- **Dictionary Creation:** It iterates through the 'stemmed_word' and 'sentiment_score' columns of the DataFrame. For each pair, it lemmatizes the word using WordNetLemmatizer (note the potential mismatch with the column name 'stemmed_word' – the code expects lemmatized forms or lemmatizes on the fly) and converts the score to a float. These word-score pairs are stored in a dictionary (lex).
- **Output:** Returns the lex dictionary mapping lemmatized words to their sentiment scores. A success message indicating the number of loaded entries is printed.

Sentiment Scoring (compute_sentiment_score):
- **Input:** Takes the list of (word, is_negated) tuples generated by apply_negation_handling.
- **Logic:**
    - Initializes score to 0.0.
    - Iterates through each (word, neg) tuple in the input list.
    - For each word, it looks up its base sentiment score in the self.lexicon dictionary using self.lexicon.get(word, 0.0). The .get() method safely returns 0.0 if the word is not found in the lexicon (i.e., unknown words don't contribute to the score).
    - If the neg flag is True (meaning the word was marked as negated by apply_negation_handling), the base score is *subtracted* from the total score.
    - If the neg flag is False, the base score is *added* to the total score.
- **Logging:** If log_score_compute is enabled, it prints the list of tokens being scored, the base score and negation status for each word, and the final total score. If verbose_tokens is also enabled within the analyze_review method call, it prints the contribution of each individual token.
- **Output:** Returns the final calculated sentiment score (a float).

Classification (classify_sentiment):
- **Input:** Takes the final sentiment score (float) from compute_sentiment_score.
- **Logic:** Applies a simple threshold: if the score is greater than or equal to 0, it classifies the sentiment as "positive"; otherwise, it classifies it as "negative".
- **Output:** Returns the sentiment label string ("positive" or "negative").

Overall Workflow (analyze_review):
- This method orchestrates the lexicon-based analysis for a given input text string.
- It calls preprocess_text to clean and normalize the text.
- It passes the result to apply_negation_handling to get tokens marked with negation status.
- It sends these tuples to compute_sentiment_score to calculate the final score.
- It calls classify_sentiment to get the final label based on the score.
- **Output:** Returns a tuple containing the classified sentiment label (string) and the calculated sentiment score (float).

This entire approach is rule-based and interpretable. Its effectiveness hinges directly on the quality, coverage, and relevance of the sentiment lexicon provided in the CSV file and the accuracy of the negation handling heuristic. It requires no machine learning training phase but depends entirely on this external linguistic resource. The discrepancy between the expected column name 'stemmed_word' and the use of a lemmatizer in select_and_load_lexicon might require clarification or adjustment depending on the actual format of the lexicon CSV being used.

6. Sentiment Analysis Approach 2: PCA + Logistic Regression (MovieSentimentAnalyzer)

This section details the first of the two machine learning approaches implemented in the script. It uses Principal Component Analysis (PCA) for dimensionality reduction, potentially leveraging unlabeled data, followed by a supervised Logistic Regression classifier trained on labeled data. The methods train_pca_lr and test_pca_lr are part of the MovieSentimentAnalyzer class.
Conceptual Overview: The goal is to classify sentiment using machine learning. The key idea here is a two-stage process:

1. **PCA:** Apply PCA, a technique that finds principal components (linear combinations of original features) that capture the most variance in the data. Here, PCA is fitted on *unsupervised* (unlabeled) data, potentially learning general structural patterns from a larger text corpus. This reduces the number of features (dimensionality reduction) from the full vocabulary size to a smaller number (n_components=100).
2. **Logistic Regression:** Train a standard Logistic Regression classifier using the *labeled* data, but represented in the lower-dimensional space defined by the principal components found in step 1.

This combination can sometimes improve model performance or generalization compared to training directly on high-dimensional sparse features, especially if labeled data is limited but unlabeled data is abundant.
Data Requirements (_load_feat, settings.json paths): This approach does not work with raw text files directly. It requires data pre-processed into specific formats:

- **Vocabulary File (pca_vocab):** A plain text file listing all unique words in the vocabulary, one word per line. The line number (0-indexed) corresponds to the feature index for that word. The _load_feat function uses the length of this list (vocab_size) to initialize feature vectors.
- **Feature Files (pca_labeled, pca_unsup, pca_test):** Text files in a sparse format, often called the "Bag-of-Words" (BoW) representation. Each line typically represents a document.
  - For *labeled* files (pca_labeled, pca_test): The line starts with a rating (e.g., "8"), followed by space-separated "index:count" pairs (e.g., "15:2 123:1 500:3"). index is the 0-based word index from the .vocab file, and count is the number of times that word appears in the document. The _load_feat function parses this, converts the initial rating to a binary label (1 if rating > 6, 0 if rating < 5, skipping ratings 5 and 6), and creates a dense NumPy vector of size vocab_size for each document, populating word counts at the appropriate indices.
  - For *unsupervised* files (pca_unsup): The format might omit the initial rating, containing only the "index:count" pairs. _load_feat handles this by setting labeled=False, skipping the label extraction and only creating the feature vectors (X).
- **Helper Function (_load_feat):** This internal function is crucial for parsing these specific .feat files. It takes the file path, vocabulary size, and a boolean labeled flag. It reads the file line by line, parses the rating (if labeled), extracts the "index:count" pairs, and constructs dense NumPy arrays (X for features, y for labels if applicable).

- **GUI Interaction:** Similar to lexicon loading, if the required paths (pca_labeled, pca_unsup, pca_vocab, pca_test) are not found in settings.json, the script uses tkinter.filedialog.askopenfilename to prompt the user for each file during the training or testing phase. Selected paths are saved back to settings.json.

Training (train_pca_lr):

1. **Path Acquisition:** Ensures paths to pca_labeled, pca_unsup, and pca_vocab are available, prompting the user via GUI if necessary.
2. **Load Data:** Reads the vocabulary list from pca_vocab. Calls _load_feat to load the labeled training data (Xl, yl) from pca_labeled and the unsupervised data (Xu) from pca_unsup, using the vocabulary size.
3. **PCA Initialization and Fitting:** Creates a PCA object from sklearn.decomposition, specifying n_components=100 (reducing features to 100 dimensions) and random_state=42 for reproducibility. Crucially, the PCA model is fit *only* on the unsupervised data (Xu). This step learns the principal components based on the structure of the potentially larger unlabeled dataset.
4. **Transform Labeled Data:** Applies the *trained* PCA model to the *labeled* feature data (Xl) using pca.transform(Xl). This projects the labeled data onto the 100 principal components learned from the unsupervised data, resulting in the reduced feature set Xr.
5. **Initial Logistic Regression Evaluation:**
    - Creates a LogisticRegression model with increased max_iter=1000 (to help convergence) and random_state=42.
    - Uses cross_val_score from sklearn.model_selection to perform k-fold cross-validation (where k is settings["cv_folds"], default 5) using the base LR model on the PCA-transformed labeled data (Xr, yl). This provides an initial estimate of the model's performance (mean accuracy and standard deviation) before hyperparameter tuning.
6. **Hyperparameter Tuning (Grid Search):**
    - Sets up a GridSearchCV object from sklearn.model_selection.
    - It uses the same LogisticRegression model definition.
    - param_grid={"C": settings.get("grid_C", [0.01,0.1,1,10])} specifies the hyperparameter to tune (C, the inverse of regularization strength) and the values to try.
    - cv=settings["cv_folds"] indicates that cross-validation should be used within the grid search to evaluate each C value.
    - n_jobs=-1 allows the grid search to use all available CPU cores for parallel processing.
    - grid.fit(Xr, yl) performs the search, finding the best value for C based on cross-validated performance.
    - The best C value and the corresponding mean cross-validated accuracy are printed.
7. **Store Best Model:** The best estimator found by GridSearchCV (the LR model with the optimal C) is stored in self.clf.
8. **Model Saving:** Uses joblib.dump to save the *trained PCA object* (self.pca), the *best LR classifier* (self.clf), and the *vocabulary list* (vocab_list) together into a single file named pca_lr_model.joblib. Saving all three is essential for later testing.
9. **Logging:** Debug messages are printed throughout the process if log_pca_training_debug is enabled.

Testing (test_pca_lr):

1. **Check for Model:** Verifies if the saved model file pca_lr_model.joblib exists. If not, it prints an error and returns.
2. **Path Acquisition:** Ensures the path to the test feature file (pca_test) is available, prompting the user via GUI if necessary.
3. **Model Loading:** Uses joblib.load to load the PCA object, the trained LR classifier (clf), and the vocabulary list from pca_lr_model.joblib.
4. **Load Test Data:** Reads the vocabulary size from the loaded vocab_list. Calls _load_feat to load the test features (X) and labels (y) from the file specified by pca_test.
5. **Transform Test Data:** Applies the *loaded* PCA object to the test features (X) using pca.transform(X) to get the reduced test features Xr. It is critical to use the *same* PCA object that was trained earlier.
6. **Prediction:** Uses the predict method of the *loaded* LR classifier (clf) on the transformed test data (Xr) to get sentiment predictions (preds). It also attempts to get prediction probabilities using predict_proba if the classifier supports it, which is useful for calculating the AUC score.
7. **Evaluation:** Calculates and prints standard performance metrics:
   - classification_report: Shows precision, recall, F1-score, and support for both 'neg' and 'pos' classes.
   - confusion_matrix: Shows the breakdown of true positives, true negatives, false positives, and false negatives.
   - roc_auc_score: Calculates the Area Under the ROC Curve, a measure of the model's ability to distinguish between classes (requires prediction probabilities). Includes error handling in case AUC calculation fails.
8. **Logging:** Debug messages related to path acquisition are printed if log_pca_test_debug is enabled.

This PCA+LR approach represents a more complex ML pipeline that attempts to leverage unlabeled data through dimensionality reduction before supervised classification. Its success depends on the quality of both the unlabeled and labeled data, the appropriateness of PCA for the task, and the tuning of both PCA (n_components) and LR (C). It critically relies on external tools or processes to generate the input .vocab and .feat files in the correct format.

7. Sentiment Analysis Approach 3: Directory-Based TF-IDF + Logistic Regression

This section describes the second machine learning approach, which is a more conventional supervised text classification pipeline. It reads raw text files directly from organized directories, uses TF-IDF for feature extraction, and trains/tests a Logistic Regression classifier. This logic is implemented in the standalone functions train_directory_supervised and test_directory_supervised.

Conceptual Overview: The goal is, again, to classify sentiment using machine learning, but via a different, widely-used pipeline for text data:

1. **Data Loading:** Read raw text documents (.txt files) directly from a directory structure where subdirectories indicate the class label (e.g., train/pos/, train/neg/).
2. **Feature Extraction (TF-IDF):** Convert the raw text documents into numerical feature vectors using the Term Frequency-Inverse Document Frequency (TF-IDF) method. TF-IDF weights words based on their frequency within a document (TF) and their rarity across the entire corpus of documents (IDF), giving higher importance to terms that are

characteristic of a particular document.
3. **Logistic Regression:** Train a Logistic Regression classifier using the labeled TF-IDF feature vectors.

This is a standard Bag-of-Words approach combined with TF-IDF weighting, common for text classification tasks. It works directly with raw text, making it potentially easier to apply to new datasets compared to the PCA+LR approach requiring pre-formatted .feat files.

Data Requirements (Directory Structure): This method expects a specific organization for the training and testing data:

- A main directory for training (dir_train) and another for testing (dir_test).
- Inside each main directory, there must be subdirectories named exactly 'pos' and 'neg'.
- Inside the 'pos' subdirectory should be all the positive review text files (e.g., .txt files).
- Inside the 'neg' subdirectory should be all the negative review text files (e.g., .txt files).
- The script iterates through these subdirectories, reads the content of each .txt file, and assigns a label (1 for 'pos', 0 for 'neg') based on the subdirectory name.
- **GUI Interaction:** If the paths dir_train or dir_test are not set in settings.json or are invalid, the script uses tkinter.filedialog.askdirectory to prompt the user to select the appropriate training or testing parent directory. Selected paths are saved to settings.json.

Training (train_directory_supervised):

1. **Path Acquisition:** Ensures the path to the training directory (dir_train) is available, prompting the user via GUI if necessary.
2. **Load Data:** Initializes empty lists texts and labels. Iterates through the 'neg' (label 0) and 'pos' (label 1) subdirectories within dir_train. For each .txt file found, it reads the content and appends it to the texts list, and appends the corresponding label (0 or 1) to the labels list.
3. **Feature Extraction (TF-IDF):**
   - Creates a TfidfVectorizer object from sklearn.feature_extraction.text. max_features=10000 limits the vocabulary considered to the 10,000 most frequent terms across the training corpus, which helps control dimensionality.
   - Calls vect.fit_transform(texts). This performs two actions:
     - fit: Learns the vocabulary and calculates the IDF weights from the training texts.
     - transform: Converts the training texts into a sparse TF-IDF matrix X.
4. **Initial Logistic Regression Evaluation:** Similar to the PCA+LR approach, it performs k-fold cross-validation (cross_val_score) using a base LogisticRegression model on the TF-IDF features (X, labels) to get an initial performance estimate.
5. **Hyperparameter Tuning (Grid Search):** Also similar to PCA+LR, it uses GridSearchCV to find the optimal C hyperparameter for LogisticRegression when trained on the TF-IDF features (X, labels). It uses the cv_folds and grid_C values from settings.json.
6. **Store Best Model:** The best estimator (LR model) found by GridSearchCV is stored in the clf variable.
7. **Model Saving:** Uses joblib.dump to save *both* the *fitted TfidfVectorizer object* (vect) and the *best LR classifier* (clf) together into a file named dir_sup_model.joblib. Saving the vectorizer is absolutely critical, as the exact same vocabulary and IDF weights must be used to process the test data.
8. **Logging:** Debug messages are printed if log_dir_train_debug is enabled.

Testing (test_directory_supervised):

1. **Check for Model:** Verifies if the saved model file dir_sup_model.joblib exists.
2. **Path Acquisition:** Ensures the path to the test directory (dir_test) is available, prompting

the user via GUI if necessary.
3. **Model Loading:** Uses joblib.load to load the *saved TfidfVectorizer* (vect) and the *trained LR classifier* (clf) from dir_sup_model.joblib.
4. **Load Test Data:** Reads the raw text files and corresponding labels from the specified test directory structure (dir_test), similar to the training data loading process.
5. **Transform Test Data:** Uses the transform method of the *loaded* TfidfVectorizer (vect.transform(texts)) to convert the raw test texts into a TF-IDF matrix X. **Crucially, it uses transform, not fit_transform**. This ensures the test data is vectorized using the vocabulary and IDF weights learned *only* from the training data, preventing data leakage and ensuring consistency.
6. **Prediction:** Uses the predict method of the loaded LR classifier (clf) on the transformed test data (X) to get predictions (preds). It also attempts to get prediction probabilities (predict_proba) for AUC calculation.
7. **Evaluation:** Calculates and prints the same set of metrics as the PCA+LR approach: classification_report, confusion_matrix, and roc_auc_score.
8. **Logging:** Debug messages related to path acquisition are printed if log_dir_test_debug is enabled.

This directory-based TF-IDF + Logistic Regression approach provides a self-contained, standard supervised learning pipeline for text classification. It operates directly on commonly organized raw text data. Its performance depends on the quality and quantity of the labeled training data, the effectiveness of TF-IDF features for the task, and the tuning of the TfidfVectorizer and LogisticRegression models.

8. Training, Tuning, and Evaluating Models

The script incorporates standard machine learning practices for building robust models and evaluating them thoroughly, specifically within the PCA+LR and TF-IDF+LR approaches.

Cross-Validation (cross_val_score):

- **Purpose:** Instead of just splitting the training data once into a training and validation set, k-fold cross-validation (CV) provides a more reliable estimate of how the model is likely to perform on unseen data.
- **Mechanism:** The training data is divided into 'k' folds (where 'k' is specified by cv_folds in settings.json, defaulting to 5). The model is trained 'k' times. In each iteration, one fold is held out as a validation set, and the model is trained on the remaining k-1 folds. The model's performance (typically accuracy for classification) is recorded on the held-out fold.
- **Usage:** The script uses cross_val_score(base_clf, X, y, cv=cv) in both train_pca_lr (with Xr, yl) and train_directory_supervised (with TF-IDF X, labels). It calculates the accuracy for each fold and prints the mean accuracy and standard deviation across all folds. This gives an indication of the baseline performance and stability of the chosen classifier (LogisticRegression) on the respective feature sets *before* hyperparameter tuning.

Hyperparameter Tuning (GridSearchCV):

- **Purpose:** Most machine learning models have hyperparameters (parameters set *before* training, like the regularization strength C in Logistic Regression) that significantly affect performance. GridSearchCV automates the process of finding the best combination of these hyperparameters.
- **Mechanism:**
    - It takes a model (e.g., LogisticRegression), a param_grid (a dictionary where keys

are hyperparameter names and values are lists of values to try, e.g., {"C": [0.01, 0.1, 1, 10]}), and a cross-validation strategy (cv=cv_folds).
    - It systematically trains and evaluates the model using cross-validation for *every possible combination* of hyperparameters specified in the grid.
    - It identifies the combination that yields the best average cross-validation score.
- **Usage:** The script uses GridSearchCV in both train_pca_lr and train_directory_supervised to specifically tune the C parameter of LogisticRegression. The search space for C is defined by settings["grid_C"]. The n_jobs=-1 argument allows it to use multiple CPU cores, speeding up the potentially time-consuming search process. After fitting (grid.fit(X, y)), the script retrieves the best model (grid.best_estimator_) and prints the best C value found (grid.best_params_['C']) along with its corresponding mean CV score (grid.best_score_). This ensures the final saved model uses a data-driven optimal setting for the C hyperparameter.

Evaluation Metrics: Once a model is trained (either the best one from GridSearch or loaded from a file), its performance on a separate *test set* (data not used during training or tuning) is evaluated using several metrics provided by sklearn.metrics:

- **classification_report(y_true, y_pred, target_names=['neg','pos']):** Provides a text summary of key classification metrics:
    - **Precision:** For a given class (e.g., 'pos'), what proportion of instances predicted as 'pos' were actually 'pos'? (True Positives / (True Positives + False Positives)). High precision means fewer false positives.
    - **Recall (Sensitivity):** For a given class (e.g., 'pos'), what proportion of actual 'pos' instances were correctly identified? (True Positives / (True Positives + False Negatives)). High recall means fewer false negatives.
    - **F1-score:** The harmonic mean of precision and recall (2 * Precision * Recall / (Precision + Recall)). It provides a single metric balancing both concerns.
    - **Support:** The number of actual occurrences of the class in the test set. The report shows these metrics for each class ('neg', 'pos') and also provides averaged values (macro avg, weighted avg).
- **confusion_matrix(y_true, y_pred):** Generates a matrix that explicitly shows the counts of:
    - True Negatives (TN): Correctly predicted 'neg'.
    - False Positives (FP): Incorrectly predicted 'pos' (actually 'neg').
    - False Negatives (FN): Incorrectly predicted 'neg' (actually 'pos').
    - True Positives (TP): Correctly predicted 'pos'. This matrix gives a direct view of where the model is making errors.
- **roc_auc_score(y_true, y_pred_proba):** Calculates the Area Under the Receiver Operating Characteristic (ROC) Curve.
    - The ROC curve plots the True Positive Rate (Recall) against the False Positive Rate (FP / (FP + TN)) at various classification thresholds.
    - The AUC represents the likelihood that the model will rank a randomly chosen positive instance higher than a randomly chosen negative instance. An AUC of 1.0 is perfect classification, while 0.5 represents random guessing.
    - This metric requires the model's predicted probabilities (predict_proba), not just the final class labels. The script attempts to calculate this and includes error handling.

By employing cross-validation for robust estimation, grid search for hyperparameter optimization, and a comprehensive suite of evaluation metrics on a held-out test set, the script follows sound machine learning practices to build and assess its models, providing a nuanced understanding of their performance beyond simple accuracy.

9. Running the Analyzer: Main Menu and User Interaction

The script provides a command-line interface (CLI) menu system to allow users to select and run the different sentiment analysis functionalities.
Entry Point (if __name__ == '__main__': main()): This is a standard Python idiom. It ensures that the main() function, which contains the primary application logic and menu loop, is called only when the script is executed directly (e.g., python your_script_name.py), not when it's imported as a module into another script.
Main Loop (main()):

- The main function initializes the analyzer variable to None.
- It enters an infinite while True loop, which continuously displays the main menu and prompts the user for input until they choose to exit.
- Inside the loop, it prints the main menu options.

Menu Options: The main menu presents five choices:

1. **Lexicon-based analysis:** If chosen, it first checks if the analyzer object (an instance of MovieSentimentAnalyzer) has been created. If not (i.e., on the first use or after restarting), it calls select_and_load_lexicon() to load the necessary sentiment lexicon. If a lexicon is successfully loaded, it creates the analyzer instance. It then enters a submenu specific to lexicon-based analysis.
2. **PCA + LR model:** This option leads to a submenu for training or testing the PCA+LR model. Similar to option 1, it ensures an analyzer instance exists before proceeding (even though the lexicon itself isn't used for PCA+LR training/testing, the methods train_pca_lr and test_pca_lr are defined within that class). This suggests a potential area for refactoring where PCA+LR logic could be separated if desired.
3. **Directory-based model:** Leads to a submenu for training or testing the TF-IDF+LR model using the standalone functions train_directory_supervised and test_directory_supervised.
4. **Settings:** Enters a submenu allowing the user to manage script settings stored in settings.json.
5. **Exit:** Prints "Goodbye!" and breaks out of the while True loop, terminating the script.

Submenus:

- **Lexicon-Based Submenu (Option 1):** Offers options to analyze text entered manually via input(), analyze text read from a .txt file (using tkinter.filedialog.askopenfilename to select the file), reload a different lexicon CSV file, or go back to the main menu.
- **PCA + LR Submenu (Option 2):** Offers options to call analyzer.train_pca_lr() or analyzer.test_pca_lr().
- **Directory-Based Submenu (Option 3):** Offers options to call train_directory_supervised() or test_directory_supervised().
- **Settings Submenu (Option 4):** Lists all the boolean log_* flags and verbose_tokens setting, showing their current state (ON/OFF). Allows the user to toggle any of these flags by entering its corresponding number. It also provides an option to clear all saved file/directory paths from the settings dictionary (setting them back to ""), forcing the script to ask for them again via GUI prompts on the next run. Changes are saved immediately using save_settings().

GUI for File/Directory Selection (tkinter): A notable feature of this CLI application is the integrated use of tkinter.filedialog. Instead of requiring users to manually find and type full file paths or edit settings.json, the script calls functions like askopenfilename (for files) or askdirectory (for directories) whenever an input path is needed and not already configured. This happens in:

- select_and_load_lexicon (for lexicon CSV)
- train_pca_lr (for .feat and .vocab files)
- test_pca_lr (for test .feat file)
- train_directory_supervised (for training directory)
- test_directory_supervised (for test directory)
- Lexicon submenu option 2 (for individual review .txt file)

This significantly improves usability, especially for users less comfortable with command-line operations, by providing a familiar graphical way to browse and select the necessary inputs. The root=Tk(); root.withdraw();... root.update() sequence is a standard way to use tkinter dialogs without showing the main tkinter window.

Overall, the main function acts as a controller, orchestrating calls to the appropriate analysis, training, testing, or configuration functions based on user choices navigated through a series of text menus, enhanced by GUI dialogs for path inputs.

10. Guidance for Future Modifications

This section provides pointers for developers looking to modify or extend the script's functionality, linking potential changes to the relevant code sections.

Adjusting Preprocessing (preprocess_text):

- **Location:** MovieSentimentAnalyzer.preprocess_text method.
- **Potential Changes:**
    - **Stemming vs. Lemmatization:** The code initializes PorterStemmer but uses WordNetLemmatizer. One could switch to stemming (self.stemmer.stem(w)) or use both. Stemming is faster but less linguistically accurate (e.g., 'studies', 'studying' -> 'studi'). Lemmatization aims for dictionary words ('studies', 'studying' -> 'study').
    - **Punctuation Handling:** The current code removes most punctuation but keeps apostrophes. Depending on the analysis goal, one might remove apostrophes as well, or try to handle punctuation differently (e.g., treating '!' as indicative of strong sentiment). Modify the re.sub pattern.
    - **Tokenization:** The current .split() is basic. For more complex text, consider using NLTK's word_tokenize, which handles contractions and punctuation more robustly: from nltk.tokenize import word_tokenize; tokens = word_tokenize(cleaned).
    - **Stop Words:** Modify the self.stop_words set or the logic for keeping/removing words (e.g., add domain-specific stopwords).

Modifying Negation Handling (apply_negation_handling):

- **Location:** MovieSentimentAnalyzer.apply_negation_handling method.
- **Potential Changes:**
    - **Negation Scope:** Adjust self.negation_scope (default 3) in __init__. A larger scope might capture longer-range negation effects but could also incorrectly negate unrelated words.
    - **Negation Word List:** Expand or refine self.negation_words in __init__.
    - **Advanced Negation:** Implement more sophisticated logic. This could involve

checking for intervening punctuation (if punctuation handling is changed) or using Part-of-Speech tagging or dependency parsing (using libraries like spaCy or NLTK's parser integrations) to understand sentence structure better. This would be a significant modification.

Using a Different Lexicon (select_and_load_lexicon):

- **Location:** Primarily affects the input to MovieSentimentAnalyzer. The loading happens in select_and_load_lexicon.
- **Process:** Simply prepare a new sentiment lexicon as a CSV file with the required columns: 'stemmed_word' (containing lemmatized or stemmable words) and 'sentiment_score' (numeric score). Use Option 1 -> 3 in the main menu ("Load new lexicon") or clear the lexicon_path in settings (Option 4 -> Clear paths) and select the new file when prompted upon restarting Option 1.

Tuning ML Models:

- **Locations:** train_pca_lr and train_directory_supervised functions/methods.
- **Potential Changes:**
  - **Hyperparameters (GridSearchCV):**
    - Modify the param_grid in the GridSearchCV call within the training functions. For LogisticRegression, explore different C values, add the penalty parameter ('l1', 'l2'), or solver options.
    - Replace LogisticRegression entirely with another classifier from sklearn (e.g., SVC, RandomForestClassifier) and adjust the param_grid accordingly for the new classifier's hyperparameters.
  - **Cross-Validation Folds:** Change the cv_folds value in settings.json.
  - **PCA Components:** In train_pca_lr, modify the n_components parameter passed to PCA(). Experimenting with different numbers of components can impact performance.
  - **TF-IDF Parameters:** In train_directory_supervised, adjust parameters of TfidfVectorizer, such as max_features, ngram_range (e.g., (1, 2) to include bigrams), min_df, max_df.

Changing Data Sources:

- **Lexicon-Based:** Prepare a new lexicon CSV as described above.
- **PCA+LR:** This requires generating new .vocab and .feat files (for labeled, unlabeled, and test sets) in the specific format expected by _load_feat. The process for generating these files is *external* to this script. Once generated, update the corresponding paths in settings.json or select them via the GUI prompts.
- **Directory-Based:** Organize the new raw text data (.txt files) into the required pos/neg subdirectory structure within new training and testing parent directories. Update dir_train and dir_test in settings.json or select the new directories via the GUI prompts.

Adding Features/Analysis:

- New analysis steps (e.g., aspect-based sentiment analysis, emotion detection) would likely require significant new code, potentially new classes or functions, and integration into the main menu structure.

Improving Efficiency:

- **Data Loading:** For very large datasets, _load_feat or reading many small files in train_directory_supervised could be slow. Consider optimized file reading, using sparse matrix formats more directly if memory becomes an issue, or data sampling.
- **Model Training:** PCA fitting and especially GridSearchCV can be computationally expensive. Ensure n_jobs=-1 is used in GridSearchCV. For very large datasets, consider

techniques like randomized search (RandomizedSearchCV) instead of exhaustive grid search, or using models that train faster.

Comparison of Approaches: The following table summarizes the key characteristics of the three sentiment analysis methods implemented in the script:

Approach	Core Technique	Data Input	Pros	Cons
Lexicon-Based	Lexicon lookup + Negation rules	CSV lexicon file	Transparent logic, No ML training needed	Heavily dependent on lexicon quality/coverage, Heuristic negation
PCA + LR	PCA (unsupervised) + Logistic Regression (supervised)	.vocab + .feat files (labeled & unlabeled)	May leverage unlabeled data, Dimensionality reduction	Requires specific pre-processed files, PCA difficult to interpret
TF-IDF + LR	TF-IDF (supervised) + Logistic Regression (supervised)	Raw .txt files in pos/neg dirs	Standard pipeline, Works directly on raw text	Ignores word order (BoW), Performance depends on data & tuning

This comparison helps in choosing the appropriate method for a given task or dataset and understanding the trade-offs involved when considering modifications or improvements.

11. Conclusion

Summary: This Python script provides a versatile framework for performing sentiment analysis on movie reviews using three distinct approaches: a rule-based lexicon method with custom negation handling, a machine learning pipeline combining PCA (potentially leveraging unlabeled data) with Logistic Regression, and a standard supervised pipeline using TF-IDF features and Logistic Regression on raw text files. It includes robust configuration management via settings.json, standard ML practices like cross-validation and hyperparameter tuning (GridSearchCV), comprehensive evaluation metrics, and a user-friendly CLI menu enhanced with GUI file/directory selection.

Strengths:

- **Flexibility:** Offers multiple analysis methods catering to different data availability scenarios (lexicon vs. labeled text vs. unlabeled text).
- **Configurability:** Externalizes paths, logging levels, and model parameters via settings.json, promoting adaptability and maintainability.
- **Standard Practices:** Incorporates established ML techniques like TF-IDF, PCA, Logistic Regression, cross-validation, and grid search.
- **Usability:** The CLI menu combined with tkinter dialogs for path selection makes the script relatively easy to operate.
- **Transparency:** Extensive optional logging allows for detailed tracing of text processing and model training/testing steps.

Potential Areas for Enhancement:

- **Negation Handling:** The current fixed-scope heuristic in apply_negation_handling could be refined using more context-aware techniques.
- **Preprocessing:** Explore alternative tokenization, stemming/lemmatization strategies, or

punctuation handling in preprocess_text.

- **Model Exploration:** Experiment with different sklearn classifiers or tune existing model hyperparameters beyond the current grid search settings.
- **Feature Engineering:** For the ML approaches, explore different feature representations beyond TF-IDF or PCA-reduced BoW (e.g., word embeddings).
- **Efficiency:** Optimize data loading or model training for very large datasets if performance becomes a bottleneck.
- **Code Structure:** Consider refactoring the PCA+LR methods out of the MovieSentimentAnalyzer class for better separation of concerns, as they don't rely on the class's lexicon state.

Final Thought: This report has aimed to provide a clear and comprehensive guide to the script's functionality and structure. By understanding the different components, their interactions, the underlying concepts (NLP preprocessing, lexicon scoring, PCA, TF-IDF, Logistic Regression, CV, GridSearch), and the specific implementation details, developers should be well-equipped to use, maintain, and effectively modify this sentiment analysis codebase.