Function Code

```
def clean_text(text):
    nopunc = [w for w in text if w not in string.punctuation]
    nopunc = ''.join(nopunc)
    return  ' '.join([word for word in nopunc.split() if word.lower() not in
stopwords.words('english')])
```

Explanation

This function `clean_text` seems to be aimed at cleaning and processing text data. Let's break it down step by step:

1. **Removing Punctuation:**
   - `nopunc = [w for w in text if w not in string.punctuation]`: This line creates a list `nopunc` containing all characters from the input text `text` that are not punctuation marks. `string.punctuation` is a string containing all the punctuation marks.

   - `nopunc = ''.join(nopunc)`: Then, it joins all the characters in `nopunc` back together into a single string. So, this line essentially removes all punctuation from the text.

2. **Removing Stopwords:**
   - `[word for word in nopunc.split() if word.lower() not in stopwords.words('english')]`: This part splits the text into words using `.split()`, then iterates over each word. For each word, it checks if the lowercase version of the word is not in the list of English stopwords. Stopwords are commonly used words (e.g., "the", "is", "and") that are often removed from text because they don't carry significant meaning.

3. **Returning Cleaned Text:**
   - `return ' '.join(...)`: Finally, it joins the filtered words back together into a single string with spaces between them, effectively removing any stopwords.

To use this function, you would pass a piece of text into it, and it would return the text with punctuation removed and stopwords filtered out. However, it seems to be missing an import statement for `string` and `stopwords` from the NLTK library. You need to ensure you've imported these libraries before using this function.

Function Code	`def preprocess(text):`
	`    return ' '.join([word for word in word_tokenize(text) if word not in stopwords.words('english') and not word.isdigit() and word not in string.punctuation])`

Explanation	This function, `preprocess`, appears to be a text preprocessing function designed to prepare text data for further analysis or processing. Let's break down its functionality: 1. **Tokenization:** - `word_tokenize(text)`: This part tokenizes the input text `text`, splitting it into individual words. `word_tokenize` is a function from the NLTK library that performs word tokenization, which essentially means breaking down a text into its component words. 2. **Removing Stopwords, Digits, and Punctuation:** - `[word for word in word_tokenize(text) if word not in stopwords.words('english') and not word.isdigit() and word not in string.punctuation]`: This list comprehension iterates over each word produced by `word_tokenize(text)`. For each word: - `word not in stopwords.words('english')`: It checks if the word is not a stopword (i.e., a common word like "the", "is", "and" that may not carry significant meaning). - `not word.isdigit()`: It checks if the word is not a digit. `isdigit()` is a method that returns True if all characters in the string are digits. - `word not in string.punctuation`: It checks if the word is not a punctuation mark. This ensures that punctuation symbols are removed from the text. 3. **Joining Tokens Back into a String:** - `' '.join(...)`: Finally, it joins the filtered words back together into a single string with spaces between them. This reconstructed string is then returned. In summary, this function takes a piece of text, tokenizes it into words, and then filters out stopwords, digits, and punctuation marks. It's a common preprocessing step in natural language processing tasks to clean and prepare text data for further analysis or modeling.

Function Code	`stemmer = PorterStemmer()`
	`def stem_words(text):`
	`    return ' '.join([stemmer.stem(word) for word in text.split()])`
	`df['text_'] = df['text_'].apply(lambda x: stem_words(x))`

Explanation	This function, `stem_words`, appears to be aimed at stemming words in a piece of text. Stemming is the process of reducing words to their root or base form. Let's break down how this function works: 1. **Initializing Stemmer:** - `stemmer = PorterStemmer()`: This line initializes a stemmer object using the Porter stemming algorithm. The Porter stemmer is a widely used algorithm for stemming English words. 2. **Stemming Words:** - `[stemmer.stem(word) for word in text.split()]`: This list comprehension iterates over each word in the input text `text`, which is split into individual words using `.split()`. For each word, `stemmer.stem(word)` is called, which returns the stem of the word using the Porter stemming algorithm. 3. **Joining Stemmed Words Back into a String:** - `' '.join(...)`: After stemming each word, the stemmed words are joined back together into a single string with spaces between them. 4. **Applying the Function to a DataFrame Column:** - `df['text_'].apply(lambda x: stem_words(x))`: This line applies the `stem_words` function to each element in the 'text_' column of the DataFrame `df`. It uses the `apply` function along with a lambda function to pass each element of the column to the `stem_words` function. In summary, this function takes a piece of text, stems each word in the text using the Porter stemming algorithm, and returns the stemmed text. It's commonly used as a text preprocessing step to reduce words to their base forms before further analysis or modeling.

Function Code	`lemmatizer = WordNetLemmatizer()`
	`def lemmatize_words(text):`
	`    return ' '.join([lemmatizer.lemmatize(word) for word in text.split()])`
	`df["text_"] = df["text_"].apply(lambda text: lemmatize_words(text))`

Explanation	This function, `lemmatize_words`, is designed to lemmatize words in a piece of text. Lemmatization is the process of reducing words to their base or dictionary form, known as lemmas. Here's how this function works: 1. **Initializing Lemmatizer:** - `lemmatizer = WordNetLemmatizer()`: This line initializes a lemmatizer object using the WordNet Lemmatizer. WordNet is a lexical database for the English language that includes lemmas for words. 2. **Lemmatizing Words:** - `[lemmatizer.lemmatize(word) for word in text.split()]`: This list comprehension iterates over each word in the input text `text`, which is split into individual words using `.split()`. For each word, `lemmatizer.lemmatize(word)` is called, which returns the lemma of the word using the WordNet Lemmatizer. 3. **Joining Lemmatized Words Back into a String:** - `' '.join(...)`: After lemmatizing each word, the lemmatized words are joined back together into a single string with spaces between them. 4. **Applying the Function to a DataFrame Column:** - `df["text_"].apply(lambda text: lemmatize_words(text))`: This line applies the `lemmatize_words` function to each element in the 'text_' column of the DataFrame `df`. It uses the `apply` function along with a lambda function to pass each element of the column to the `lemmatize_words` function. In summary, this function takes a piece of text, lemmatizes each word in the text using the WordNet Lemmatizer, and returns the lemmatized text. Lemmatization is a common text preprocessing step to reduce words to their base forms, which can help with tasks like text analysis and natural language understanding.

Function Code	`df[df['label']=='OR'][['text_','length']].sort_values(by='length',ascending=False).head().iloc[0].text_`
Explanation	Let's break down the provided code step by step: 1. \`df[df['label']=='OR']\`: This part filters the DataFrame \`df\` to select only rows where the value in the 'label' column is equal to 'OR'. This creates a subset of the DataFrame containing only rows with the label 'OR'. 2. \`[['text_', 'length']]\`: After filtering, this part selects only the 'text_' and 'length' columns from the filtered DataFrame. It creates a new DataFrame with only these two columns. 3. \`.sort_values(by='length', ascending=False)\`: This sorts the DataFrame based on the values in the 'length' column in descending order. This means that rows with longer values in the 'length' column will appear first. 4. \`.head()\`: This selects the first few rows of the sorted DataFrame. By default, it selects the first 5 rows. 5. \`.iloc[0].text_\`: Finally, \`.iloc[0]\` selects the first row from the resulting DataFrame, and \`.text_\` selects the value in the 'text_' column of that row. So, it retrieves the value of the 'text_' column from the first row of the sorted DataFrame. In summary, the provided code fetches the text data ('text_') from the row with the longest text length among the rows where the label is 'OR' in the DataFrame \`df\`.

Function Code

```python
def text_process(review):
    nopunc = [char for char in review if char not in string.punctuation]
    nopunc = ''.join(nopunc)
    return [word for word in nopunc.split() if word.lower() not in
stopwords.words('english')]
```

Explanation

This function, `text_process`, seems to be a simple text preprocessing function designed to clean and tokenize a piece of text. Let's break down its functionality:

1. **Removing Punctuation:**
   - `nopunc = [char for char in review if char not in string.punctuation]`: This line creates a list called `nopunc` containing all characters from the input `review` that are not punctuation marks. `string.punctuation` is a string containing all the punctuation marks in Python.

   - `nopunc = ".join(nopunc)`: Then, it joins all the characters in `nopunc` back together into a single string. So, this line essentially removes all punctuation from the text.

2. **Tokenization and Removing Stopwords:**
   - `return [word for word in nopunc.split() if word.lower() not in stopwords.words('english')]`: This line splits the text into words using `.split()`, then iterates over each word. For each word:
     - It checks if the lowercase version of the word is not in the list of English stopwords. Stopwords are common words (e.g., "the", "is", "and") that are often removed from text because they don't carry significant meaning.

3. **Returning Tokenized Text without Stopwords:**
   - The function returns a list of words obtained after removing stopwords from the tokenized text.

In summary, this function takes a piece of text (`review`), removes punctuation, tokenizes the text into words, and removes stopwords, returning a list of words without stopwords. This type of preprocessing is common in natural language processing tasks to clean and prepare text data for further analysis or modeling.

Function Code	```bow_transformer = CountVectorizer(analyzer=text_process)```
	```bow_transformer```

Explanation	This code snippet is setting up a Bag-of-Words (BoW) transformation using the \`CountVectorizer\` from the scikit-learn library. Let's break it down: 1. **CountVectorizer Initialization:** - \`CountVectorizer(analyzer=text_process)\`: This line initializes a \`CountVectorizer\` object. \`CountVectorizer\` is a scikit-learn tool used to convert text data into a matrix of token counts. The \`analyzer\` parameter specifies the function that will be used to preprocess the text data. In this case, \`text_process\` function is used as the analyzer. 2. **\`text_process\` Function:** - \`text_process\` function seems to be a custom text preprocessing function that cleans and tokenizes text, as described in the previous explanation. It is likely defined elsewhere in the code and is used here to preprocess text data before building the BoW representation. 3. **BoW Transformation:** - The initialized \`CountVectorizer\` object is assigned to the variable \`bow_transformer\`. This object is now configured to transform raw text data into a Bag-of-Words representation using the \`text_process\` function for preprocessing. In summary, this code sets up a Bag-of-Words transformation using the \`CountVectorizer\` with a custom text preprocessing function \`text_process\`. The resulting \`bow_transformer\` object can now be used to convert raw text data into a matrix of token counts suitable for machine learning models.

Function Code	`bow_transformer.fit(df['text_'])`
	`print("Total Vocabulary:",len(bow_transformer.vocabulary_))`

Explanation	This code snippet is part of a process for creating a Bag-of-Words (BoW) representation of text data using the \`CountVectorizer\` from scikit-learn. Let's break it down: 1. **Fitting the Transformer to Data:** - \`bow_transformer.fit(df['text_'])\`: This line fits the \`CountVectorizer\` transformer (\`bow_transformer\`) to the text data stored in the 'text_' column of the DataFrame \`df\`. When you call \`.fit()\` on a \`CountVectorizer\` object, it learns the vocabulary from the text data and prepares the transformer for transforming new text data. 2. **Printing Vocabulary Size:** - \`print("Total Vocabulary:", len(bow_transformer.vocabulary_))\`: After fitting the \`CountVectorizer\`, this line prints the total size of the vocabulary learned by the transformer. The vocabulary is stored in the \`vocabulary_\` attribute of the \`CountVectorizer\` object. The vocabulary contains unique words found in the text data, and its size indicates how many unique words were encountered during fitting. In summary, this code snippet fits a \`CountVectorizer\` transformer to text data, learns the vocabulary from the data, and then prints the size of the learned vocabulary. The resulting vocabulary will be used to transform new text data into a Bag-of-Words representation, where each document (or piece of text) is represented as a vector of word counts based on this vocabulary.

Function Code	`bow_msg4 = bow_transformer.transform([review4])`
	`print(bow_msg4)`
	`print(bow_msg4.shape)`

Explanation	This code snippet is using the fitted `CountVectorizer` transformer (`bow_transformer`) to transform a new piece of text data (`review4`) into its Bag-of-Words (BoW) representation. Let's break it down: 1. **Transforming the Text:** - `bow_msg4 = bow_transformer.transform([review4])`: This line transforms the text `review4` into its BoW representation using the `transform()` method of the `bow_transformer`. The `transform()` method takes a list of strings as input, so `review4` is passed as a single-item list (`[review4]`). The output, `bow_msg4`, is a sparse matrix representing the BoW representation of the text. 2. **Printing the BoW Representation:** - `print(bow_msg4)`: This line prints the BoW representation of `review4`, which is stored in the `bow_msg4` variable. Since `bow_msg4` is a sparse matrix, it will print the non-zero elements along with their indices in the matrix. 3. **Printing the Shape of the BoW Representation:** - `print(bow_msg4.shape)`: This line prints the shape of the BoW representation stored in `bow_msg4`. The shape is a tuple indicating the dimensions of the matrix. In this case, it prints the number of documents (rows) and the number of unique words in the vocabulary (columns) represented in the BoW matrix. In summary, this code snippet transforms the text `review4` into its Bag-of-Words representation using the fitted `CountVectorizer` transformer (`bow_transformer`). It then prints the BoW representation itself and its shape to inspect the transformed data. This BoW representation can now be used as input for machine learning models.

Function Code	`bow_reviews = bow_transformer.transform(df['text_'])`
Explanation	This code snippet is using the fitted `CountVectorizer` transformer (`bow_transformer`) to transform all text data in the 'text_' column of the DataFrame `df` into its corresponding Bag-of-Words (BoW) representation. Let's break it down: 1. **Transforming Text Data:** - `bow_reviews = bow_transformer.transform(df['text_'])`: This line transforms all the text data in the 'text_' column of the DataFrame `df` into its BoW representation using the `transform()` method of the `bow_transformer`. The `transform()` method takes a list-like object containing text data as input, so `df['text_']` is passed directly. The output, `bow_reviews`, is a sparse matrix representing the BoW representations of all texts in the DataFrame. In summary, this code snippet transforms all text data in the 'text_' column of the DataFrame `df` into their corresponding Bag-of-Words representations using the fitted `CountVectorizer` transformer (`bow_transformer`). The resulting `bow_reviews` matrix can now be used as input for various machine learning models or other text analysis tasks.

Function Code	`print("Shape of Bag of Words Transformer for the entire reviews corpus:",bow_reviews.shape)`
	`print("Amount of non zero values in the bag of words model:",bow_reviews.nnz)`

Explanation	This code snippet prints out information about the Bag-of-Words (BoW) representation generated from the entire reviews corpus using the `CountVectorizer` transformer. 1. **Shape of BoW Transformer:** - `print("Shape of Bag of Words Transformer for the entire reviews corpus:", bow_reviews.shape)`: This line prints the shape of the BoW representation stored in the variable `bow_reviews`. The shape is a tuple containing two values: the number of documents (or reviews) and the number of unique words (or features) in the vocabulary. For example, if the shape is (1000, 5000), it means there are 1000 reviews in the corpus, and the BoW representation includes 5000 unique words. 2. **Amount of Non-Zero Values:** - `print("Amount of non-zero values in the bag of words model:", bow_reviews.nnz)`: This line prints the number of non-zero values in the BoW representation. Since the BoW representation is typically a sparse matrix, most of its elements are zeros. The `nnz` attribute returns the count of non-zero elements in the sparse matrix. This count indicates how many unique words are present across all the reviews in the corpus. In summary, these lines of code provide important information about the BoW representation generated from the entire reviews corpus: its shape (number of reviews and unique words) and the number of non-zero values (indicating the presence of words across the corpus). This information helps in understanding the size and density of the BoW representation, which is useful for further analysis or modeling tasks.

Function Code	`print("Sparsity:",np.round((bow_reviews.nnz/(bow_reviews.shape[0]*bow_reviews.shape[1]))*100,2))`
Explanation	This code snippet calculates and prints the sparsity of the Bag-of-Words (BoW) representation generated from the entire reviews corpus. Let's break it down: 1. **Calculating Sparsity:** - \`(bow_reviews.nnz / (bow_reviews.shape[0] \* bow_reviews.shape[1]))\`: This expression calculates the sparsity of the BoW representation. \`bow_reviews.nnz\` gives the number of non-zero values in the BoW matrix, \`bow_reviews.shape[0]\` gives the number of rows (representing the number of documents or reviews), and \`bow_reviews.shape[1]\` gives the number of columns (representing the number of unique words in the vocabulary). By dividing the number of non-zero values by the total number of elements in the BoW matrix (i.e., the product of the number of rows and columns), we get the proportion of non-zero values in the matrix, representing the proportion of words that are present in the reviews corpus. 2. **Multiplying by 100 and Rounding:** - \`np.round(... \* 100, 2)\`: This expression multiplies the calculated sparsity by 100 to convert it into a percentage and then rounds it to two decimal places using NumPy's \`round()\` function. This step is done to make the sparsity value easier to interpret. 3. **Printing Sparsity:** - \`print("Sparsity:", ...)\`: This line prints the calculated sparsity value along with a descriptive label ("Sparsity:"). The sparsity value represents the percentage of unique words in the vocabulary that are present in the reviews corpus. In summary, this code snippet calculates and prints the sparsity of the Bag-of-Words representation, providing insight into how dense or sparse the representation is. A higher sparsity value indicates a sparser representation, meaning that fewer unique words are present across the reviews corpus.

Function Code	`tfidf_transformer = TfidfTransformer().fit(bow_reviews)`
	`tfidf_rev4 = tfidf_transformer.transform(bow_msg4)`
	`print(bow_msg4)`

Explanation	This code snippet is performing TF-IDF (Term Frequency-Inverse Document Frequency) transformation on the Bag-of-Words (BoW) representation using the \`TfidfTransformer\` from scikit-learn. 1. **Initializing and Fitting the Transformer:** - \`tfidf_transformer = TfidfTransformer().fit(bow_reviews)\`: This line initializes a \`TfidfTransformer\` object and fits it to the BoW representation \`bow_reviews\` using the \`fit()\` method. Fitting the transformer calculates the inverse document frequency (IDF) for each term in the vocabulary based on its frequency across the entire corpus. 2. **Transforming a Specific Document:** - \`tfidf_rev4 = tfidf_transformer.transform(bow_msg4)\`: This line transforms the BoW representation of a specific document (likely \`review4\`) stored in \`bow_msg4\` into its TF-IDF representation using the fitted \`TfidfTransformer\`. The \`transform()\` method applies the IDF transformation to the document's BoW representation. 3. **Printing the BoW Representation:** - \`print(bow_msg4)\`: This line prints the original BoW representation of the specific document stored in \`bow_msg4\`. It provides a comparison between the original BoW representation and the TF-IDF representation obtained after transformation. In summary, this code snippet initializes a TF-IDF transformer, fits it to the BoW representation of the entire reviews corpus, transforms a specific document's BoW representation into its TF-IDF representation, and prints the original BoW representation of that document. TF-IDF transformation is used to weigh the importance of terms in a document relative to their frequency in the entire corpus.

Function Code	
	```
	tfidf_reviews = tfidf_transformer.transform(bow_reviews)
	print("Shape:",tfidf_reviews.shape)
	print("No. of Dimensions:",tfidf_reviews.ndim)
	```

Explanation	This code snippet is performing TF-IDF (Term Frequency-Inverse Document Frequency) transformation on the Bag-of-Words (BoW) representation using the \`TfidfTransformer\` from scikit-learn. 1. **Initializing and Fitting the Transformer:** - \`tfidf_transformer = TfidfTransformer().fit(bow_reviews)\`: This line initializes a \`TfidfTransformer\` object and fits it to the BoW representation \`bow_reviews\` using the \`fit()\` method. Fitting the transformer calculates the inverse document frequency (IDF) for each term in the vocabulary based on its frequency across the entire corpus. 2. **Transforming a Specific Document:** - \`tfidf_rev4 = tfidf_transformer.transform(bow_msg4)\`: This line transforms the BoW representation of a specific document (likely \`review4\`) stored in \`bow_msg4\` into its TF-IDF representation using the fitted \`TfidfTransformer\`. The \`transform()\` method applies the IDF transformation to the document's BoW representation. 3. **Printing the BoW Representation:** - \`print(bow_msg4)\`: This line prints the original BoW representation of the specific document stored in \`bow_msg4\`. It provides a comparison between the original BoW representation and the TF-IDF representation obtained after transformation. In summary, this code snippet initializes a TF-IDF transformer, fits it to the BoW representation of the entire reviews corpus, transforms a specific document's BoW representation into its TF-IDF representation, and prints the original BoW representation of that document. TF-IDF transformation is used to weigh the importance of terms in a document relative to their frequency in the entire corpus.

Function Code	
	```
	pipeline = Pipeline([
	    ('bow',CountVectorizer(analyzer=text_process)),
	    ('tfidf',TfidfTransformer()),
	    ('classifier',MultinomialNB())
	])
	```

Explanation	This code snippet creates a machine learning pipeline using scikit-learn's `Pipeline` class. Let's break down each component of the pipeline: 1. **CountVectorizer (BoW Transformation):** - `'bow', CountVectorizer(analyzer=text_process)`: This part of the pipeline is responsible for converting text data into a Bag-of-Words (BoW) representation. It uses the `CountVectorizer` class from scikit-learn. The `analyzer` parameter specifies the text processing function to be used for tokenization and cleaning. In this case, it uses the `text_process` function, which is likely a custom function for text preprocessing as described earlier. 2. **TfidfTransformer (TF-IDF Transformation):** - `'tfidf', TfidfTransformer()`: This part of the pipeline is responsible for transforming the BoW representation into TF-IDF (Term Frequency-Inverse Document Frequency) representation. It uses the `TfidfTransformer` class from scikit-learn. TF-IDF transformation weights the importance of terms in a document relative to their frequency in the entire corpus. 3. **Classifier (Multinomial Naive Bayes):** - `'classifier', MultinomialNB()`: This part of the pipeline is responsible for classification. It uses the Multinomial Naive Bayes classifier (`MultinomialNB`) from scikit-learn. Naive Bayes classifiers are commonly used for text classification tasks. Each component in the pipeline is defined as a tuple containing a name and an instance of the corresponding scikit-learn transformer or estimator class. The order of the components in the pipeline defines the sequence in which the data will be transformed or processed. In summary, this pipeline first converts text data into a BoW representation using `CountVectorizer`, then transforms the BoW representation into TF-IDF representation using `TfidfTransformer`, and finally applies classification using the Multinomial Naive Bayes classifier. This pipeline encapsulates the entire workflow of text preprocessing and classification into a single object, making it convenient to train and deploy machine learning models.

Function Code	`pipeline.fit(review_train,label_train)`
Explanation	This code snippet fits the previously defined machine learning pipeline to the training data. 1. **Fitting the Pipeline:** - \`pipeline.fit(review_train, label_train)\`: This line fits the \`pipeline\` object to the training data. The \`fit()\` method of the pipeline sequentially fits each component of the pipeline to the data. Here, \`review_train\` represents the training text data, and \`label_train\` represents the corresponding labels or target values. - The \`fit()\` method first transforms the text data using the \`CountVectorizer\` to convert it into a Bag-of-Words (BoW) representation, then applies the \`TfidfTransformer\` to transform the BoW representation into TF-IDF representation, and finally fits the Multinomial Naive Bayes classifier (\`MultinomialNB\`) to the TF-IDF transformed data. - By fitting the pipeline to the training data, the model learns from the data and is trained to make predictions on new, unseen data. In summary, this code snippet fits the machine learning pipeline to the training data, which involves transforming the text data and training the classifier. After fitting, the pipeline is ready to make predictions on new text data.

Function Code	`predictions = pipeline.predict(review_test)`
	`predictions`

Explanation	This code snippet uses the fitted pipeline to make predictions on the test data. 1. **Making Predictions:** - `predictions = pipeline.predict(review_test)`: This line applies the trained pipeline (`pipeline`) to the test data (`review_test`) to make predictions. The `predict()` method of the pipeline applies each step of the pipeline (including text preprocessing and classification) to the test data and returns the predicted labels for each instance in the test data. 2. **Storing Predictions:** - `predictions`: The predicted labels are stored in the variable `predictions`. Each element in `predictions` corresponds to the predicted label for the corresponding instance in the test data. In summary, this code snippet uses the fitted machine learning pipeline to predict labels for the test data. The resulting predictions are stored in the variable `predictions` and can be further evaluated or analyzed.

Function Code	`print('Classification Report:',classification_report(label_test,predictions))`
	`print('Confusion Matrix:',confusion_matrix(label_test,predictions))`
	`print('Accuracy Score:',accuracy_score(label_test,predictions))`

Explanation	These lines of code evaluate the performance of the classification model by comparing the predicted labels (\`predictions\`) with the true labels (\`label_test\`). Let's break down each part: 1. **Classification Report:** - \`print('Classification Report:', classification_report(label_test, predictions))\`: This line prints a classification report generated by the \`classification_report\` function from scikit-learn. The classification report provides a comprehensive summary of the model's performance, including precision, recall, F1-score, and support for each class. It also calculates these metrics for both micro and macro average, providing insights into the overall performance of the model across different classes. 2. **Confusion Matrix:** - \`print('Confusion Matrix:', confusion_matrix(label_test, predictions))\`: This line prints the confusion matrix, which is a table showing the number of true positives, false positives, true negatives, and false negatives for each class. The confusion matrix provides a detailed breakdown of the model's performance and helps identify which classes are being confused with each other. 3. **Accuracy Score:** - \`print('Accuracy Score:', accuracy_score(label_test, predictions))\`: This line prints the accuracy score, which is the proportion of correctly predicted labels out of all the labels in the test set. The accuracy score provides a simple and intuitive measure of the overall performance of the model. However, it may not be the best metric to use in all cases, especially when dealing with imbalanced datasets or when different types of errors have different consequences. In summary, these lines of code provide a comprehensive evaluation of the classification model's performance, including detailed metrics such as precision, recall, F1-score, support, confusion matrix, and accuracy score. These metrics help assess how well the model is performing and identify areas for improvement.

Flow chart of code

1. Import Library		6. `lemmatize_words (text)` Function , (apply this function on dataset)		11. `TfidfTransformer().fit(bow_reviews)` Function , (apply this function on dataset)		16. Evaluate model.	
2. Read Dataset		7.Remove null value		12. Creating training and testing data		17 .Design user interface from Gradio	
3. `clean_text function` (apply this function on dataset)		8. Add length column		13.create Pipeline			
4. `preprocess (text)` Function , (apply this function on dataset)		9. `text_process (review)` Apply this function in step 10.		14.train model			
5. `stem_words (text)` Function , (apply this function on dataset)		10. `CountVectorizer (analyzer=text_process)` Function , (apply this function on dataset)		15. Get Prediction from Model .			