Objective:

Develop machine learning models to classify emotions in text samples.

1. Loading and Preprocessing

1. Load the dataset and perform necessary preprocessing steps. This should include text cleaning, tokenization, and removal of stopwords. Explain the preprocessing techniques used and their impact on model performance

In [71]:
```
# import the liberaries
import pandas as pd
import nltk
from nltk.tokenize import word_tokenize
from nltk.corpus import stopwords
import string
from nltk.stem import PorterStemmer
from nltk.stem import WordNetLemmatizer
from sklearn.feature_extraction.text import CountVectorizer
from sklearn.model_selection import train_test_split
from sklearn.svm import SVC
from sklearn.naive_bayes import GaussianNB
from sklearn.metrics import classification_report,confusion_matrix,accuracy_sc
```

In [72]:
```
# Load the dataset
data = pd.read_csv("C:/Users/PWORLD/Downloads/nlp_dataset.csv")
df = data
df
```

Out[72]:

	Comment	Emotion
0	i seriously hate one subject to death but now ...	fear
1	im so full of life i feel appalled	anger
2	i sit here to write i start to dig out my feel...	fear
3	ive been really angry with r and i feel like a...	joy
4	i feel suspicious if there is no one outside l...	fear
...	...	...
5932	i begun to feel distressed for you	fear
5933	i left feeling annoyed and angry thinking that...	anger
5934	i were to ever get married i d have everything...	joy
5935	i feel reluctant in applying there because i w...	fear
5936	i just wanted to apologize to you because i fe...	anger

5937 rows × 2 columns

In [73]:

```
# Download necessary NLTK resources
nltk.download('punkt')
nltk.download('stopwords')
nltk.download('wordnet')



# Initialize the stemmer and lemmatizer
stemmer = PorterStemmer()
lemmatizer = WordNetLemmatizer()

# Function to preprocess the text data
def preprocess_text(Comment, use_stemming=False, use_lemmatization=False):
    # 1. Convert text to lowercase
    comment = Comment.lower()

    # 2. Remove punctuation
    comment = ''.join([char for char in comment if char not in string.punctuati

    # 3. Tokenize the text
    tokens = word_tokenize(Comment)

    # 4. Remove stopwords
    stop_words = set(stopwords.words('english'))
    tokens = [word for word in tokens if word not in stop_words and word.isalph

    # 5. Apply stemming if requested
    if use_stemming:
        tokens = [stemmer.stem(word) for word in tokens]

    # 6. Apply lemmatization if requested
    if use_lemmatization:
        tokens = [lemmatizer.lemmatize(word) for word in tokens]

    return tokens

# Apply preprocessing with stemming
df['processed_comment_stemmed'] = df['Comment'].apply(lambda x: preprocess_text

# Apply preprocessing with lemmatization
df['processed_comment_lemmatized'] = df['Comment'].apply(lambda x: preprocess_t

# Display the dataframe with original and processed text
print(df[['Comment', 'processed_comment_stemmed', 'processed_comment_lemmatized
```

```
[nltk_data] Downloading package punkt to
[nltk_data]     C:\Users\PWORLD\AppData\Roaming\nltk_data...
[nltk_data]   Package punkt is already up-to-date!
[nltk_data] Downloading package stopwords to
[nltk_data]     C:\Users\PWORLD\AppData\Roaming\nltk_data...
[nltk_data]   Package stopwords is already up-to-date!
[nltk_data] Downloading package wordnet to
[nltk_data]     C:\Users\PWORLD\AppData\Roaming\nltk_data...
[nltk_data]   Package wordnet is already up-to-date!
```

```
                                              Comment  \
0  i seriously hate one subject to death but now ...
1                im so full of life i feel appalled
2  i sit here to write i start to dig out my feel...
3  ive been really angry with r and i feel like a...
4  i feel suspicious if there is no one outside l...

                           processed_comment_stemmed  \
0  [serious, hate, one, subject, death, feel, rel...
1                      [im, full, life, feel, appal]
2  [sit, write, start, dig, feel, think, afraid, ...
3  [ive, realli, angri, r, feel, like, idiot, tru...
4  [feel, suspici, one, outsid, like, raptur, hap...

                         processed_comment_lemmatized
0  [seriously, hate, one, subject, death, feel, r...
1                  [im, full, life, feel, appalled]
2  [sit, write, start, dig, feeling, think, afrai...
3  [ive, really, angry, r, feel, like, idiot, tru...
4  [feel, suspicious, one, outside, like, rapture...
```

In [ ]:

In [74]:
```
X = df.processed_comment_lemmatized
y = df.Emotion
```

In [75]:
```
X
```

Out[75]:
```
0       [seriously, hate, one, subject, death, feel, r...
1                         [im, full, life, feel, appalled]
2       [sit, write, start, dig, feeling, think, afrai...
3       [ive, really, angry, r, feel, like, idiot, tru...
4       [feel, suspicious, one, outside, like, rapture...
                              ...
5932                        [begun, feel, distressed]
5933    [left, feeling, annoyed, angry, thinking, cent...
5934    [ever, get, married, everything, ready, offer,...
5935    [feel, reluctant, applying, want, able, find, ...
5936    [wanted, apologize, feel, like, heartless, bitch]
Name: processed_comment_lemmatized, Length: 5937, dtype: object
```

In [76]:
```
y
```

Out[76]:
```
0        fear
1       anger
2        fear
3         joy
4        fear
        ...
5932     fear
5933    anger
5934      joy
5935     fear
5936    anger
Name: Emotion, Length: 5937, dtype: object
```

Explanation of Preprocessing Techniques

1. Lowercasing Purpose: Convert all text to lowercase to ensure that words are compared uniformly.

Impact on Model: Lowercasing prevents the model from treating the same word as different due to case sensitivity. This step reduces the vocabulary size and helps the model generalize better.

2. Removing Punctuation Purpose: Remove punctuation marks (like ., !, ?, etc.) as they generally do not contribute useful information for most NLP tasks (unless you're analyzing sentiment or specific symbols).

Impact on Model: Removing punctuation reduces noise and avoids unnecessary token variations. For example, "hello!" becomes "hello", which is consistent for modeling.

3. Tokenization Purpose: Split the text into individual tokens (words or symbols). Tokenization is a foundational step in NLP as it breaks down the text into manageable units that the model can work with.

Impact on Model: Tokenization is crucial because it structures the raw text into discrete components that can be further analyzed. The resulting tokens are the units the model uses for learning.

4. Stopword Removal Purpose: Remove common words like "the", "a", "is", etc., that don't carry much meaningful information in many text analysis tasks.

Impact on Model: Removing stopwords reduces the dimensionality of the feature space, allowing the model to focus on more meaningful words. This can improve performance by making the model focus on the more important words that carry specific meaning for the task (e.g., sentiment or classification).

5. Filtering Non-Alphabetic Tokens Purpose: The line word.isalpha() ensures that we only keep alphabetic words, removing numbers or special characters that do not contribute to the analysis.

Impact on Model: This step removes noisy tokens like "123", "@", or "!!!", which are typically irrelevant for many text analysis tasks like sentiment classification or topic modeling.

Impact of Preprocessing on Model Performance:

Reduction in Noise: Preprocessing helps to remove irrelevant information (like punctuation and stopwords), which can reduce noise in the model's input and improve accuracy.

Feature Reduction: By removing stopwords and non-alphabetic tokens, you reduce the size of the feature space, which can make models more efficient and focused on more informative features.

Improved Generalization: Lowercasing ensures uniformity, preventing the model from overfitting to variations of the same word.

More Efficient Training: By reducing the number of features (tokens), preprocessing can make the training process faster and more computationally efficient, which is especially important

2.Feature Extraction :

Implement feature extraction using CountVectorizer or TfidfVectorizer. Describe how the chosen method transforms the text data into numerical features.

In [77]:
```
# Ensure that X is a list of strings
X = df['Comment'].astype(str).tolist()

cv = CountVectorizer()
x1 = cv.fit_transform(X)
x1
```

Out[77]:
```
<5937x8954 sparse matrix of type '<class 'numpy.int64'>'
        with 93020 stored elements in Compressed Sparse Row format>
```

In [78]:
```
x1.toarray()
```

Out[78]:
```
array([[0, 0, 0, ..., 0, 0, 0],
       [0, 0, 0, ..., 0, 0, 0],
       [0, 0, 0, ..., 0, 0, 0],
       ...,
       [0, 0, 0, ..., 0, 0, 0],
       [0, 0, 0, ..., 0, 0, 0],
       [0, 0, 0, ..., 0, 0, 0]], dtype=int64)
```

In [79]:
```
cv.get_feature_names_out()
```

Out[79]:
```
array(['aa', 'aac', 'aaron', ..., 'zonisamide', 'zq', 'zumba'],
      dtype=object)
```

In [80]:
```
pd.DataFrame(x1.toarray(),columns=cv.get_feature_names_out())
```

Out[80]:

	aa	aac	aaron	ab	abandon	abandoned	abandonment	abbigail	abc	abdomen	...	zend
0	0	0	0	0	0	0	0	0	0	0	...	
1	0	0	0	0	0	0	0	0	0	0	...	
2	0	0	0	0	0	0	0	0	0	0	...	
3	0	0	0	0	0	0	0	0	0	0	...	
4	0	0	0	0	0	0	0	0	0	0	...	
...	...	...	...	...	...	...	...	...	...	...	...	
5932	0	0	0	0	0	0	0	0	0	0	...	
5933	0	0	0	0	0	0	0	0	0	0	...	
5934	0	0	0	0	0	0	0	0	0	0	...	
5935	0	0	0	0	0	0	0	0	0	0	...	
5936	0	0	0	0	0	0	0	0	0	0	...	

5937 rows × 8954 columns

In [81]:
```
x_train,x_test,y_train,y_test =train_test_split(x1,y,test_size=0.2)
```

In [82]:
```
x_train
```

Out[82]:
```
<4749x8954 sparse matrix of type '<class 'numpy.int64'>'
        with 74594 stored elements in Compressed Sparse Row format>
```

In [83]:
```
x_test
```

Out[83]:
```
<1188x8954 sparse matrix of type '<class 'numpy.int64'>'
        with 18426 stored elements in Compressed Sparse Row format>
```

In [84]:
```
x_train.shape
```

Out[84]:
```
(4749, 8954)
```

In [85]:
```
x_test.shape
```

Out[85]:
```
(1188, 8954)
```

CountVectorizer: We initialize CountVectorizer() and fit/transform the data. This results in a sparse matrix, which we convert to a dense format using .toarray() and then turn it into a DataFrame for easy visualization. It transforms the text data into a bag-of-words representation.

Each word in the corpus is treated as a separate feature, and the value in the matrix represents the count of the word in each document.

Advantages:

Simple to implement and understand. Effective when the frequency of words matters. Disadvantages:

Doesn't capture the importance of words in the context of the entire corpus (e.g., common words like "the", "a" can dominate the features). x1.toarray(): Converts the sparse matrix x1 (created by CountVectorizer) into a dense NumPy array. This dense matrix contains the term frequency (TF) for each word in each document. cv.get_feature_names_out(): This method returns a list of all the unique words (features) in the corpus, which are used as column headers in the resulting DataFrame.

3. Model Development

Train the following machine learning models a)Naive Bayesb)Support Vector Machine

1.Support Vector Machine

A Support Vector Machine (SVM) is a supervised machine learning algorithm used for classification and regression tasks. It is particularly effective in high-dimensional spaces and for datasets that are not linearly separable.

Training Phase: SVM takes the training data and finds the optimal hyperplane that separates the classes with the maximum margin. This is done by solving an optimization problem.

Prediction Phase: Once the hyperplane is defined, new data points are classified based on which side of the hyperplane they lie on.

In [86]:
```
# SVM
svm_model = SVC()
svm_model.fit(x_train,y_train)
```

Out[86]:
```
SVC
SVC()
```

In [87]:
```
y_pred=svm_model.predict(x_test)
y_pred
```

Out[87]:
```
array(['anger', 'anger', 'anger', ..., 'joy', 'fear', 'joy'], dtype=object)
```

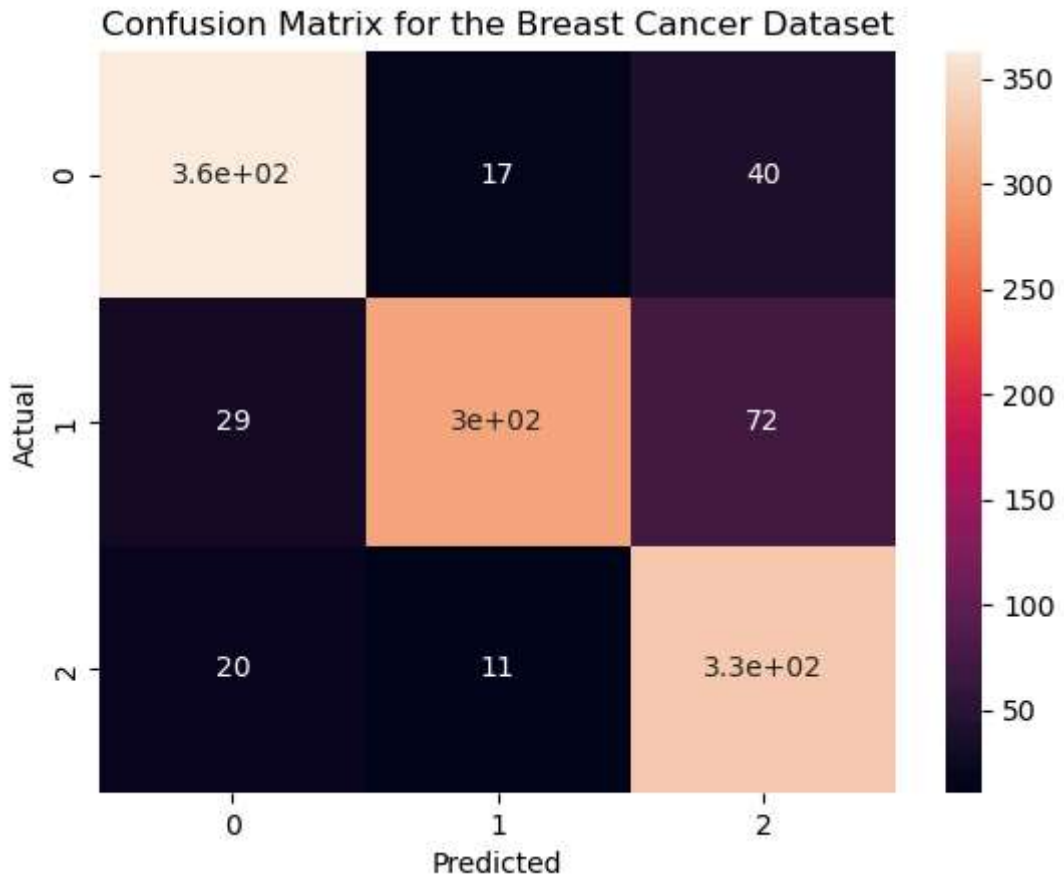
In [88]:
```
y_test
```

Out[88]:
```
5936    anger
1236     fear
1301    anger
3477      joy
2683    anger
        ...
1274     fear
293     anger
5850     fear
3720     fear
2189      joy
Name: Emotion, Length: 1188, dtype: object
```

In [89]:
```
print(confusion_matrix(y_test,y_pred))
```

```
[[363  17  40]
 [ 29 303  72]
 [ 20  11 333]]
```

In [90]:
```
con=confusion_matrix(y_test, y_pred)
sns.heatmap(con, annot=True)
plt.xlabel('Predicted')
plt.ylabel('Actual')
plt.title('Confusion Matrix for the Breast Cancer Dataset')
plt.show()
```



In [91]:
```
print(classification_report(y_test, y_pred))
```

```
              precision    recall  f1-score   support

       anger       0.88      0.86      0.87       420
        fear       0.92      0.75      0.82       404
         joy       0.75      0.91      0.82       364

    accuracy                           0.84      1188
   macro avg       0.85      0.84      0.84      1188
weighted avg       0.85      0.84      0.84      1188
```

In [92]:
```
Accuracy = accuracy_score(y_test, y_pred)
Accuracy
```

Out[92]:
```
0.8409090909090909
```

Naive Bayes

Naive Bayes is a simple yet powerful classification algorithm based on Bayes' Theorem, which applies conditional probability to predict the category or class of a data point. It is especially popular for text classification tasks such as spam filtering, sentiment analysis, and document categorization.

In text classification, it calculates how often each word appears in documents of each class. Prediction Phase: When predicting, Naive Bayes computes the posterior probability for each class using Bayes' Theorem and the product of likelihoods. The class with the highest posterior probability is chosen as the prediction.

In [93]:
```
# Naive Bayes
X_train=x_train.toarray()
model = GaussianNB()
model.fit(X_train,y_train)
```

Out[93]:
```
GaussianNB()
```

In [94]:
```
X_test=x_test.toarray()
y_pred= model.predict(X_test)
y_pred
```

Out[94]:
```
array(['anger', 'fear', 'fear', ..., 'fear', 'joy', 'anger'], dtype='<U5')
```

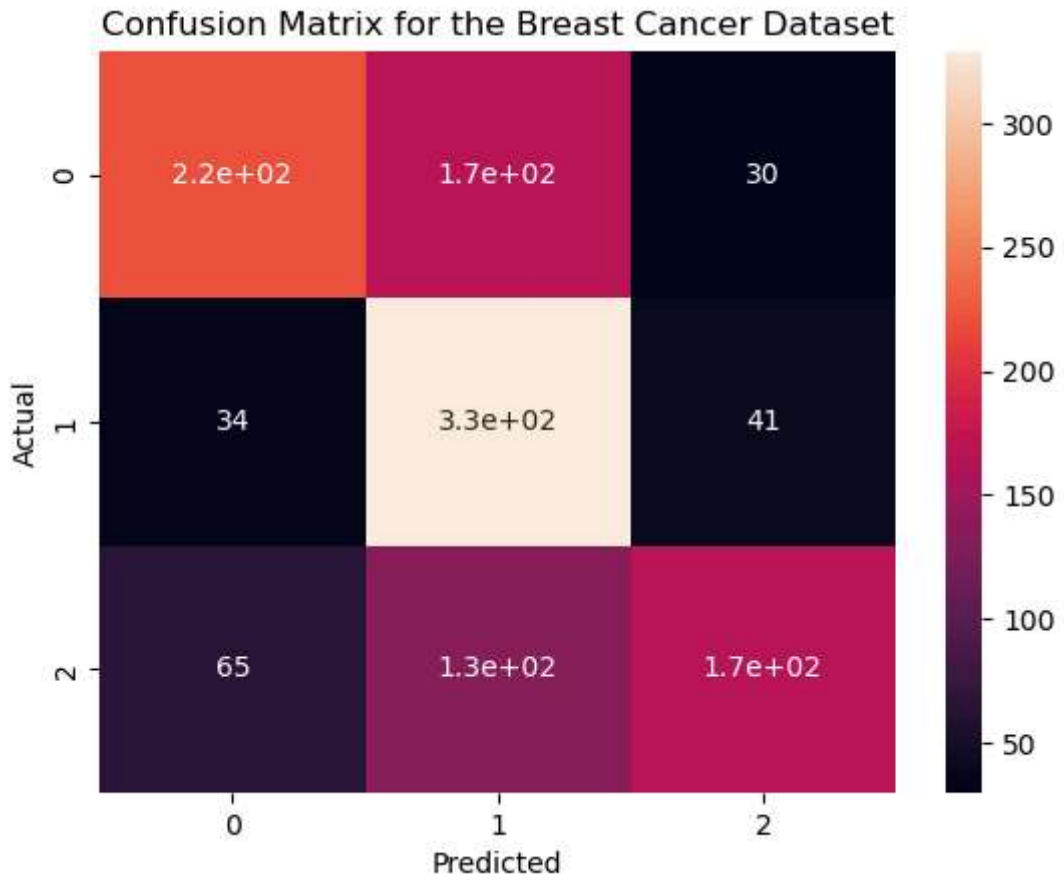
In [95]:
```
y_test
```

Out[95]:
```
5936    anger
1236     fear
1301    anger
3477      joy
2683    anger
        ...
1274     fear
293     anger
5850     fear
3720     fear
2189      joy
Name: Emotion, Length: 1188, dtype: object
```

In [96]:
```
print(confusion_matrix(y_test,y_pred))
```

```
[[220 170  30]
 [ 34 329  41]
 [ 65 133 166]]
```

In [97]:
```
con=confusion_matrix(y_test, y_pred)
sns.heatmap(con, annot=True)
plt.xlabel('Predicted')
plt.ylabel('Actual')
plt.title('Confusion Matrix for the Breast Cancer Dataset')
plt.show()
```



In [98]:
```
print(classification_report(y_test, y_pred))
```

```
              precision    recall  f1-score   support

       anger       0.69      0.52      0.60       420
        fear       0.52      0.81      0.64       404
         joy       0.70      0.46      0.55       364

    accuracy                           0.60      1188
   macro avg       0.64      0.60      0.59      1188
weighted avg       0.64      0.60      0.60      1188
```

In [99]:
```
Accuracy = accuracy_score(y_test, y_pred)
Accuracy
```

Out[99]:
```
0.6018518518518519
```

4. Model Comparison:

Evaluate the model using appropriate metrics (e.g., accuracy, F1-score). Provide a brief explanation of the chosen model and its suitability for emotion classification

In Naive Bayes, Accuracy (0.61): The overall accuracy of the model is 61%, which means the model correctly predicted the emotion for 61% of the total instances (1188). This is a moderate accuracy, which suggests the model could be improved. Macro Average: Macro Average (Precision = 0.64, Recall = 0.60, F1-Score = 0.60): The macro average is calculated by averaging the precision, recall, and F1-score across all classes without considering the class imbalance. The macro average precision and recall are lower than the accuracy, indicating that the model is not performing uniformly across all classes. Weighted Average: Weighted Average (Precision = 0.64, Recall = 0.61, F1-Score = 0.60): The weighted average takes the support (number of instances) of each class into account when calculating the average score. This gives us a slightly more realistic view of performance when the class distribution is imbalanced (which it seems to be, given the support values).

The Naive Bayes model performs moderately well, but there are noticeable weaknesses: Precision for fear is low, and recall for joy is quite low, meaning the model struggles to correctly predict these classes. Recall for fear is quite good, indicating that it does well in identifying fear, though the predictions are not as precise. Joy is a problematic class, with high precision but low recall, suggesting that the model often fails to identify joy despite being correct when it does.

In SVM, the overall accuracy of 86% means that 86% of the total instances (1188) in the dataset were classified correctly. This is a good accuracy, indicating that the SVM model performs well on the dataset as a whole. Macro Average: Precision (0.86): The average precision across all classes is 86%, suggesting that, on average, the model does a good job of accurately predicting each class. Recall (0.86): The average recall is also 86%, indicating that the model is similarly good at identifying all classes. F1-Score (0.86): The average F1-score is 86%, suggesting that the model achieves a good balance between precision and recall across all classes. Weighted

Average: The weighted average accounts for the number of instances in each class. Since the number of instances across classes is fairly balanced, the weighted average closely matches the macro average. It is also 86% for precision, recall, and F1-score.

Summary: SVM Model Performance: The model is performing well overall, with an accuracy of 86% and balanced metrics (precision, recall, and F1-score) across the three classes. The model performs best on the fear class, achieving high precision and recall. Anger also performs well with high precision and recall, though it's slightly less perfect than fear. Joy has a high recall (92%) but a lower precision (78%). The model is good at identifying joy but sometimes