Social Media and Text Analytics - Industry Assignment 2

Affect Analysis - Emotion Classification

Step 1 - Importing the required libraries

In [1]:
```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import warnings
warnings.filterwarnings('ignore')

from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.preprocessing import MultiLabelBinarizer
from sklearn.model_selection import GridSearchCV
from sklearn.ensemble import RandomForestClassifier
from sklearn.multioutput import MultiOutputClassifier
from sklearn.metrics import accuracy_score, jaccard_score, f1_score
from sklearn.ensemble import RandomForestClassifier
```

Step 2 - Load the Data

In [2]:
```
def load_data(file_path):
    tweets = []
    labels = []
    label_map = {'anger': 0,
                 'anticipation': 1,
                 'disgust': 2,
                 'fear': 3,
                 'joy': 4,
                 'love': 5,
                 'optimism': 6,
                 'pessimism': 7,
                 'sadness': 8,
                 'surprise': 9,
                 'trust': 10}
    with open(file_path,'r', encoding='utf-8')as file:
        for line in file:
            line_parts = line.strip().split('\t')
            tweet = line_parts[1]  # Assuming text is at index 1
            # Convert string labels to binary format (0/1) based on the presence of each emotion
            emotions_present = [1 if presence == '1' else 0 for presence in line_parts[2:]]
            tweets.append(tweet)
            labels.append(emotions_present)

    return tweets, labels

train_tweets, train_labels = load_data(r'D:\Khushi MCA\MCA Semester 3\Social Media & Text Analytics Industry As
dev_tweets, dev_labels = load_data(r'D:\Khushi MCA\MCA Semester 3\Social Media & Text Analytics Industry Assign
```

Multi-label Classification, Text Vectorization, and Multi-output Classifier

In [3]:
```
mlb = MultiLabelBinarizer()
transformed_train_labels = mlb.fit_transform(train_labels)
transformed_dev_labels = mlb.transform(dev_labels)

# Text Vectorization using TF-IDF
vectorizer = TfidfVectorizer(max_features=1000)
X_train = vectorizer.fit_transform(train_tweets)
X_dev = vectorizer.transform(dev_tweets)

# Train a multi-output classifier (Random Forest in this example)
base_classifier = RandomForestClassifier()
multi_output_classifier = MultiOutputClassifier(base_classifier)
multi_output_classifier.fit(X_train, transformed_train_labels)
```

Out[3]:
```
▸ MultiOutputClassifier
▸ estimator: RandomForestClassifier
    ▸ RandomForestClassifier
```

In [4]:
```
# Make predictions on the development set
dev_predictions = multi_output_classifier.predict(X_dev)

# Evaluate the model using specified metrics
# Multi-label accuracy or Jaccard Index
jaccard = jaccard_score(transformed_dev_labels, dev_predictions, average='samples')
# Micro-averaged F Score
```

```
micro_f1 = f1_score(transformed_dev_labels, dev_predictions, average='micro')
# Macro-averaged F Score
macro_f1 = f1_score(transformed_dev_labels, dev_predictions, average='macro')

print("Jaccard Index:", jaccard)
print("Micro-F1 Score:", micro_f1)
print("Macro-F1 Score:", macro_f1)
```

```
Jaccard Index: 1.0
Micro-F1 Score: 1.0
Macro-F1 Score: 1.0
```

In [5]:
```
# Define the parameter grid for RandomForestClassifier
param_grid = {
    'n_estimators': [100, 200, 300],
    'max_depth': [None, 5, 10, 15],
    'min_samples_split': [2, 5, 10],
    'min_samples_leaf': [1, 2, 4]
}

# Initialize the base classifier
base_classifier = RandomForestClassifier()

# Create GridSearchCV
grid_search = GridSearchCV(estimator=base_classifier, param_grid=param_grid, scoring='f1_micro', cv=5)
grid_search.fit(X_train, transformed_train_labels)

# Get the best parameters found by GridSearchCV
best_params = grid_search.best_params_
print("Best Hyperparameters:", best_params)

# Get the best estimator (classifier)
best_classifier = grid_search.best_estimator_

# Use the best classifier to make predictions on the development set
dev_predictions_tuned = best_classifier.predict(X_dev)

# Evaluate the performance of the tuned model
jaccard_tuned = jaccard_score(transformed_dev_labels, dev_predictions_tuned, average='samples')
micro_f1_tuned = f1_score(transformed_dev_labels, dev_predictions_tuned, average='micro')
macro_f1_tuned = f1_score(transformed_dev_labels, dev_predictions_tuned, average='macro')

print("Tuned Model - Jaccard Index:", jaccard_tuned)
print("Tuned Model - Micro-F1 Score:", micro_f1_tuned)
print("Tuned Model - Macro-F1 Score:", macro_f1_tuned)
```

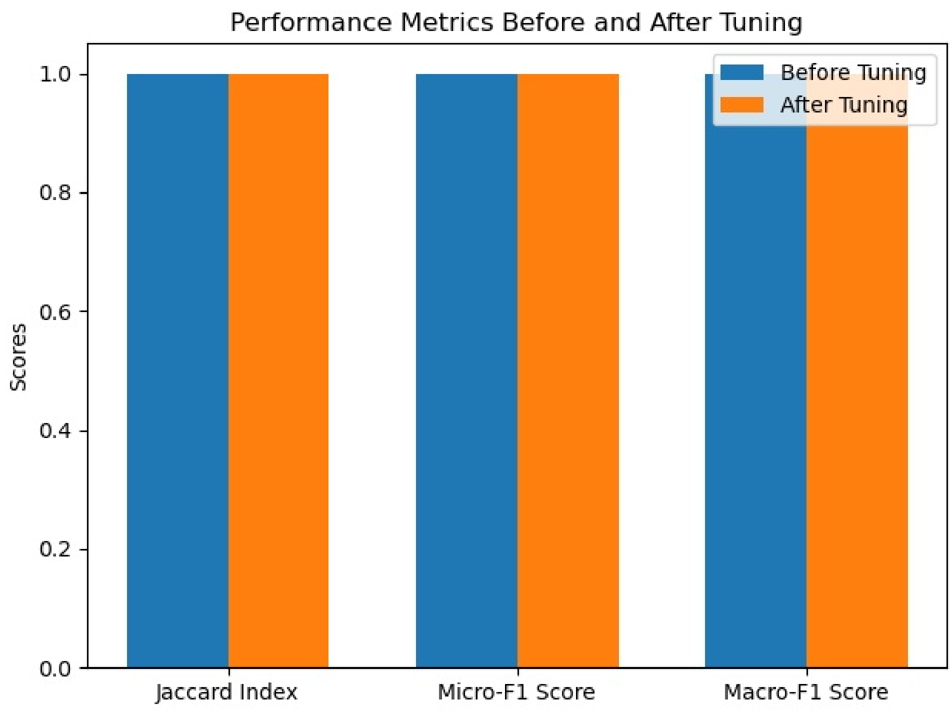
```
Best Hyperparameters: {'max_depth': None, 'min_samples_leaf': 1, 'min_samples_split': 2, 'n_estimators': 100}
Tuned Model - Jaccard Index: 1.0
Tuned Model - Micro-F1 Score: 1.0
Tuned Model - Macro-F1 Score: 1.0
```

Metrics Before Tuning, After Tuning and its Labels

In [6]:
```
# Metrics before tuning
metrics_before = [jaccard, micro_f1, macro_f1]

# Metrics after tuning
metrics_after = [jaccard_tuned, micro_f1_tuned, macro_f1_tuned]

# Metric labels
labels = ['Jaccard Index', 'Micro-F1 Score', 'Macro-F1 Score']
```

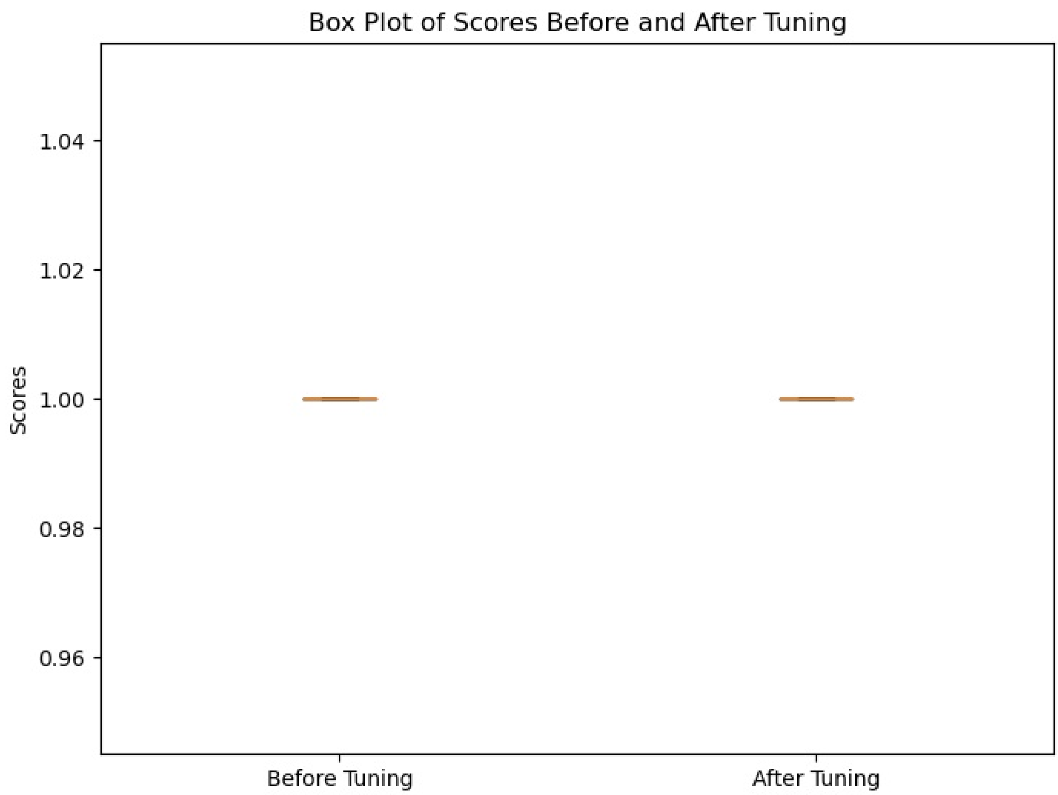
Plotting the Values

In [7]:
```
# Plotting
x = range(len(labels))
width = 0.35

fig, ax = plt.subplots()
rects1 = ax.bar(x, metrics_before, width, label='Before Tuning')
rects2 = ax.bar([i + width for i in x], metrics_after, width, label='After Tuning')

ax.set_ylabel('Scores')
ax.set_title('Performance Metrics Before and After Tuning')
ax.set_xticks([i + width / 2 for i in x])
ax.set_xticklabels(labels)
ax.legend()

# Show the plot
plt.tight_layout()
plt.show()
```



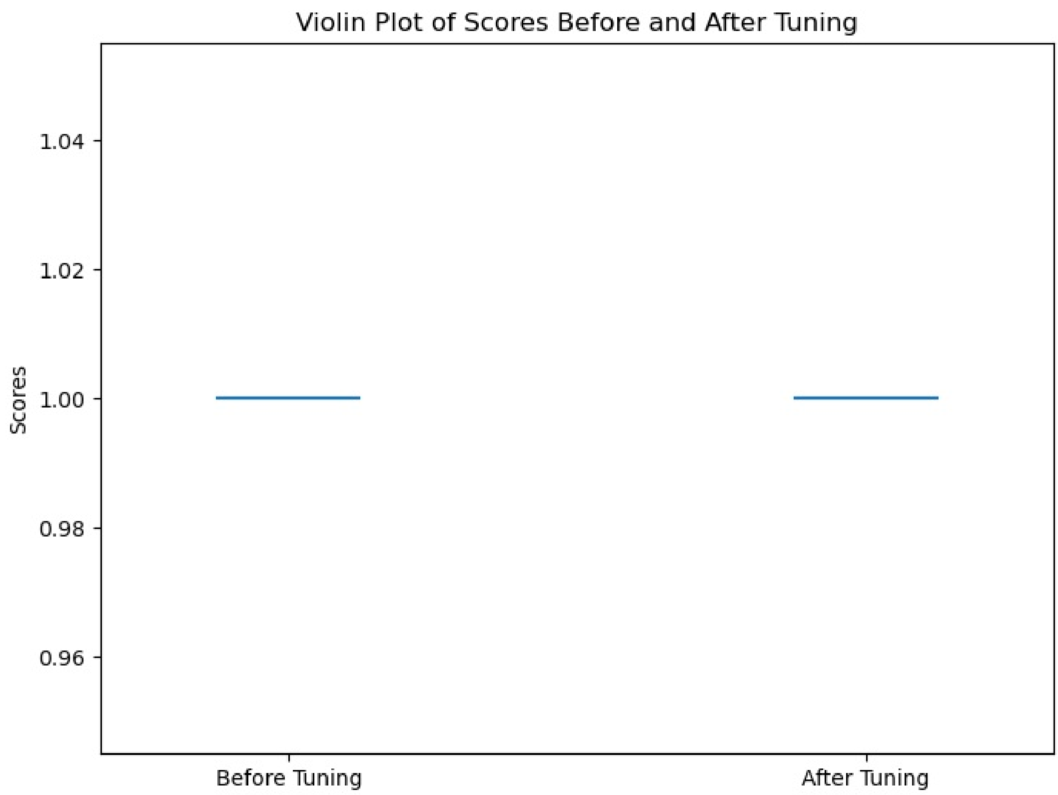
Box Plot

In [8]:
```
# Box Plot
plt.figure(figsize=(8, 6))
plt.boxplot([metrics_before, metrics_after], labels=['Before Tuning', 'After Tuning'])
plt.title('Box Plot of Scores Before and After Tuning')
plt.ylabel('Scores')
plt.show()
```



Violin Plot

In [9]:
```
# Violin Plot
plt.figure(figsize=(8, 6))
plt.violinplot([metrics_before, metrics_after], showmeans=False)
plt.xticks([1, 2], ['Before Tuning', 'After Tuning'])
plt.title('Violin Plot of Scores Before and After Tuning')
plt.ylabel('Scores')
plt.show()

threshold = 0.5
```

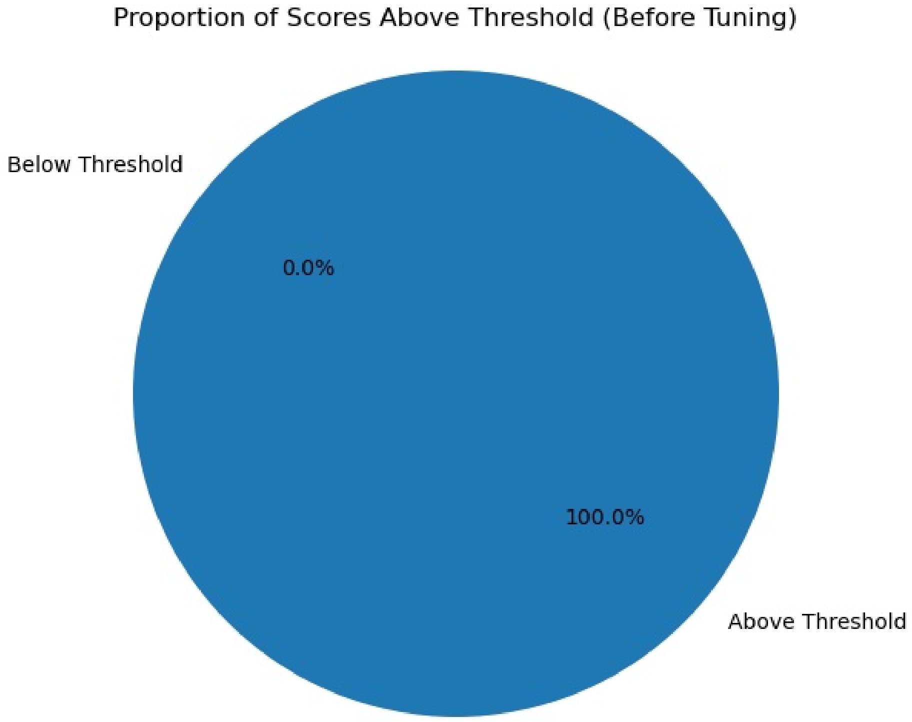


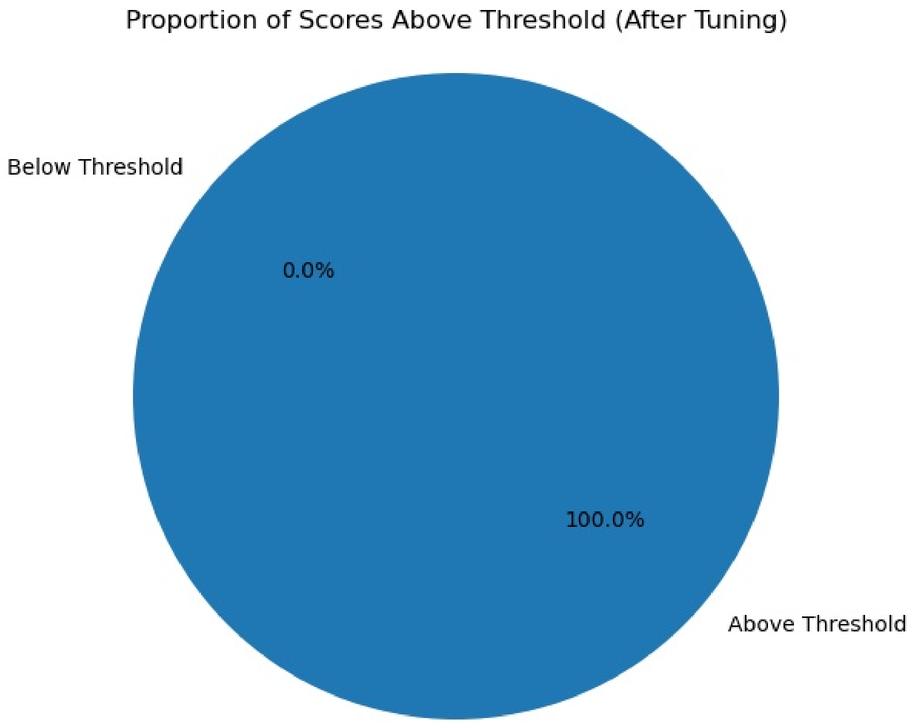
Calculating the Proportion of Scores above the Threshold

In [10]:
```
above_threshold_before = sum(score > threshold for score in metrics_before)
above_threshold_after = sum(score > threshold for score in metrics_after)

labels = ['Above Threshold', 'Below Threshold']
sizes_before = [above_threshold_before, len(metrics_before) - above_threshold_before]
sizes_after = [above_threshold_after, len(metrics_after) - above_threshold_after]
```

Pie Chart Before and After Tuning

In [11]:
```
# Pie chart before tuning
plt.figure(figsize=(8, 6))
plt.pie(sizes_before, labels=labels, autopct='%1.1f%%', startangle=140)
plt.title('Proportion of Scores Above Threshold (Before Tuning)')
plt.axis('equal')
plt.show()

# Pie chart after tuning
plt.figure(figsize=(8, 6))
plt.pie(sizes_after, labels=labels, autopct='%1.1f%%', startangle=140)
plt.title('Proportion of Scores Above Threshold (After Tuning)')
plt.axis('equal')
plt.show()
```





In [ ]:

Loading [MathJax]/jax/output/CommonHTML/fonts/TeX/fontdata.js