1. Overview

This notebook focuses on building and fine-tuning machine learning models to classify the sentiment of tweets as positive or not. It explores several models, including Logistic Regression, Random Forest, Neural Networks, and XGBoost.

Key processes include:

- Splitting data into training and test sets.
- Applying weighted metrics (Precision, Recall, F1 Score) to handle class imbalance.
- Hyperparameter tuning using GridSearchCV for optimal performance, with a focus on maximizing precision to reduce false positives, as required by the business case.
- Comparison of model performance to select the best model.
- The final model is saved using pickle for future use.

The notebook concludes that XGBoost is the most suitable model due to its balanced performance and minimal overfitting.

2. Import Necessary Libraries

In [1]:
```
# pip install keras_tuner
```

In [2]:
```python
import pandas as pd
import numpy as np
import seaborn as sns
import matplotlib.pyplot as plt
import tensorflow as tf
import keras_tuner as kt
import xgboost as xgb
import pickle
import warnings
from sklearn.model_selection import GridSearchCV
from sklearn.metrics import precision_score, make_scorer, recall_score, f1
from tensorflow.keras.metrics import Precision
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Dropout
from sklearn.ensemble import RandomForestClassifier
from sklearn.metrics import confusion_matrix
from tensorflow.keras.utils import to_categorical
from xgboost import XGBClassifier
from sklearn.linear_model import LogisticRegression
from tensorflow.keras.metrics import Accuracy
from tensorflow.keras.optimizers import Adam
from tensorflow.keras.losses import CategoricalCrossentropy
from tensorflow.keras.regularizers import l2
from tensorflow.keras import layers, regularizers

# Delete the warnings
warnings.filterwarnings('ignore')
```

3. Define Global Variables

In [3]:
```python
input_X_train_path = '../data/train_processed.csv'
input_X_test_path = '../data/test_processed.csv'
```

4. Functions

In [4]:

```
def plot_confusion_matrix_and_metrics(y_test, y_pred, title='Confusion Mat
    """
    This function plots a confusion matrix and calculates the weighted F1

    Parameters:
    y_test (array-like): True labels
    y_pred (array-like): Predicted labels
    title (str): Title for the confusion matrix plot

    Returns:
    tuple: The weighted F1 score, precision score, and recall score
    """

    # Predefined labels list
    labels_list = ["Not Positive emotion", "Positive emotion"]

    # Generate the confusion matrix
    conf_matrix = confusion_matrix(y_test, y_pred, labels=[0, 1])

    # Calculate percentages
    conf_matrix_percent = conf_matrix.astype('float') / conf_matrix.sum(ax

    # Combine the count and the percentage into one annotation
    labels = [f"{count}\n{percent:.2f}%" for count, percent in zip(conf_ma
    labels = np.asarray(labels).reshape(2, 2)

    # Plot the confusion matrix without the color bar
    plt.figure(figsize=(8, 6))
    sns.heatmap(conf_matrix, annot=labels, fmt="", cmap="Blues", cbar=Fals
                xticklabels=labels_list,
                yticklabels=labels_list)

    plt.xlabel(r'$\bf{Predicted\ labels}$')
    plt.ylabel(r'$\bf{True\ labels}$')
    plt.title(title)
    plt.show()

    # Calculate weighted precision, recall, and F1 score
    precision = precision_score(y_test, y_pred, average='weighted')
    recall = recall_score(y_test, y_pred, average='weighted')
    f1 = f1_score(y_test, y_pred, average='weighted')

    # Print metrics
    print(f"Weighted Precision: {precision:.2f}")
    print(f"Weighted Recall: {recall:.2f}")
    print(f"Weighted F1 Score: {f1:.2f}")

    return f1, precision, recall
```

In [5]:
```
def precision_results_table(y_train, y_test,
                            y_pred_train_lr, y_pred_test_lr,
                            y_pred_train_rf, y_pred_test_rf, y_pred_train_
                            predicted_classes_train, predicted_classes_tes
                            y_pred_train_xgb, y_pred_test_xgb, y_pred_trai
    """
    Function to calculate and return a table with precision scores for tra
    across different models.

    Parameters:
    y_train: Ground truth labels for the training set
    y_test: Ground truth labels for the test set
    y_pred_train_lr: Predictions for Logistic Regression (train set)
    y_pred_test_lr: Predictions for Logistic Regression (test set)
    y_pred_train_rf: Predictions for Random Forest (train set)
    y_pred_test_rf: Predictions for Random Forest (test set)
    y_pred_train_rf_2: Second set of predictions for Random Forest (train
    y_pred_test_rf_2: Second set of predictions for Random Forest (test se
    predicted_classes_train: Predictions for Neural Networks (train set)
    predicted_classes_test: Predictions for Neural Networks (test set)
    y_pred_train_xgb: Predictions for XGBoost (train set)
    y_pred_test_xgb: Predictions for XGBoost (test set)
    y_pred_train_xgb_2: Second set of predictions for XGBoost (train set)
    y_pred_test_xgb_2: Second set of predictions for XGBoost (test set)

    Returns:
    A Pandas DataFrame containing precision scores for both train and test
    """

    # Calculate precision for Logistic Regression
    precision_train_lr = precision_score(y_train, y_pred_train_lr, average
    precision_test_lr = precision_score(y_test, y_pred_test_lr, average='w

    # Calculate precision for Random Forest (first set of predictions)
    precision_train_rf = precision_score(y_train, y_pred_train_rf, average
    precision_test_rf = precision_score(y_test, y_pred_test_rf, average='w

    # Calculate precision for Random Forest (second set of predictions)
    precision_train_rf_2 = precision_score(y_train, y_pred_train_rf_2, ave
    precision_test_rf_2 = precision_score(y_test, y_pred_test_rf_2, averag

    # Calculate precision for Neural Networks
    precision_train_nn = precision_score(y_train, predicted_classes_train,
    precision_test_nn = precision_score(y_test, predicted_classes_test, av

    # Calculate precision for XGBoost (first set of predictions)
    precision_train_xgb = precision_score(y_train, y_pred_train_xgb, avera
    precision_test_xgb = precision_score(y_test, y_pred_test_xgb, average=

    # Calculate precision for XGBoost (second set of predictions)
    precision_train_xgb_2 = precision_score(y_train, y_pred_train_xgb_2, a
    precision_test_xgb_2 = precision_score(y_test, y_pred_test_xgb_2, aver

    # Create a DataFrame to store the precision results
    results = pd.DataFrame({
```

```
        'Model': ['Logistic Regression', 'Random Forest 1', 'Random Forest
        'Precision (Train)': [precision_train_lr, precision_train_rf, prec
        'Precision (Test)': [precision_test_lr, precision_test_rf, precisi

    })

    # Creating another column to see the difference between the Train's ar
    results['Difference (Train-Test)'] = results['Precision (Train)'] - re

    return results
```

5. Choosing the metric

Taking into consideration the Business case that was defined in the notebook 01_data_understanding, we are going to use the Precision score because we consider that the more false positives that we have, the more harmful it will be to the nature of our business because it will affect our analysis that we will do in the future to determine the positive features of the technological products.

6. Code

We are going to open both csv files from the notebook 02_data_preprocessing

In [6]:

```
df_train = pd.read_csv(input_X_train_path)
df_test = pd.read_csv(input_X_test_path)
```

In [7]:

```
df_train.head()
```

Out[7]:

	aapl	aaron	ab	abacus	abba	abc	aber	ability	able	abnormal	...	zms	zombie	zomg
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
1	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
2	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
3	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
4	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

5 rows × 5921 columns

In [8]:
```
df_test.head()
```

Out[8]:

	aapl	aaron	ab	abacus	abba	abc	aber	ability	able	abnormal	...	zms	zombie	zor
0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.379029	0.0	...	0.0	0.0	(
1	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.000000	0.0	...	0.0	0.0	(
2	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.000000	0.0	...	0.0	0.0	(
3	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.000000	0.0	...	0.0	0.0	(
4	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.000000	0.0	...	0.0	0.0	(

5 rows × 5921 columns

Let's do the separation between X_train, y_train, X_test, y_test

In [9]:
```
# For train
X_train, y_train = df_train.drop('emotion_type_encoded', axis=1), df_trair

# For test
X_test, y_test = df_test.drop('emotion_type_encoded', axis=1), df_test['em
```

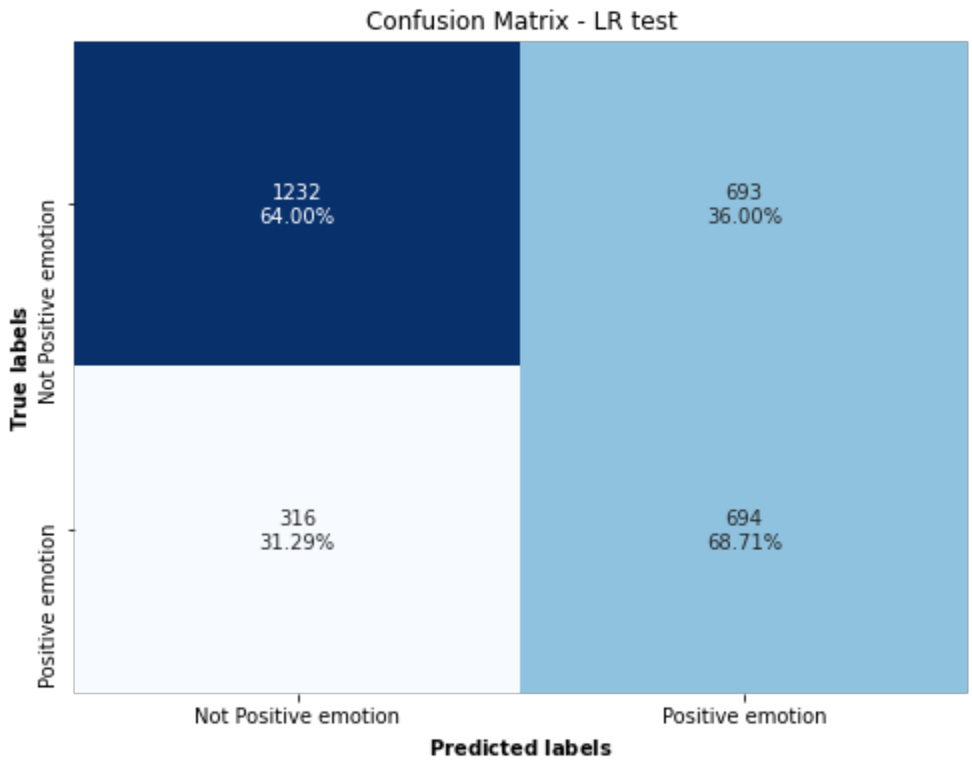
We will now proceed to train different models

6.1 LR

In [10]:
```
# Initialize the Multinomial Logistic Regression model
lr = LogisticRegression(solver='liblinear', random_state=12)

# Fit the model on the training data
lr.fit(X_train, y_train)

# Make predictions on the test data
y_pred_proba_test_lr = lr.predict_proba(X_test)[:, 1]

# Let's apply a threshold to the probabilities of y_pred_proba_test_lr to
y_pred_test_lr = np.where(y_pred_proba_test_lr >= 0.34, 1, 0)

# Make predictions on the training data to check for overfitting
y_pred_proba_train_lr = lr.predict_proba(X_train)[:, 1]

# Let's apply a threshold to the probabilities of y_pred_proba_train_lr tc
y_pred_train_lr = np.where(y_pred_proba_train_lr >= 0.34, 1, 0)
```

Let's look at the confusion matrix

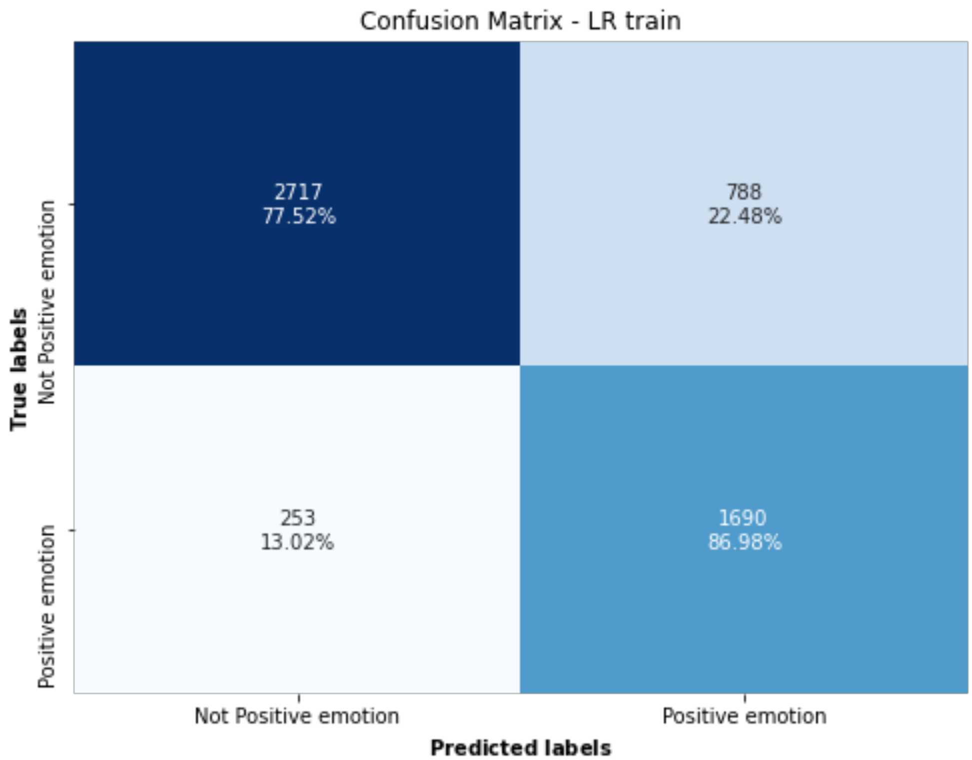
In [11]:
```
1 plot_confusion_matrix_and_metrics(y_test, y_pred_test_lr, title='Confusior
```




```
Weighted Precision: 0.69
Weighted Recall: 0.66
Weighted F1 Score: 0.66
```

Let's look at the confusion matrix of the train dataset to check for overfitting

In [12]:
```python
plot_confusion_matrix_and_metrics(y_train, y_pred_train_lr, title='Confusi
```



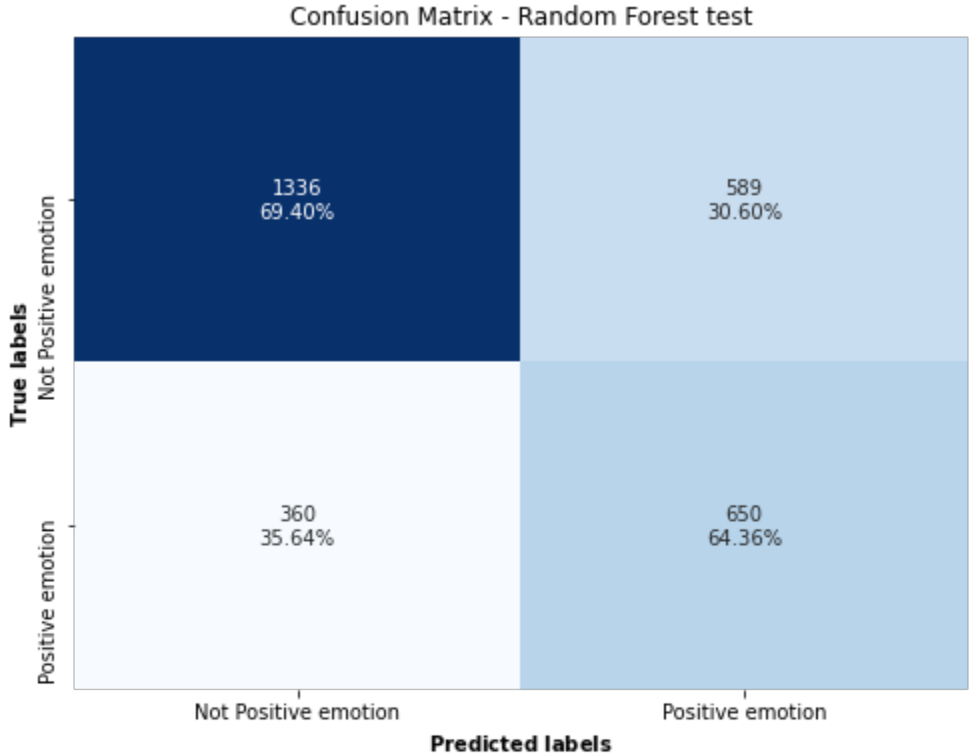

```
Weighted Precision: 0.83
Weighted Recall: 0.81
Weighted F1 Score: 0.81
```

As can be seen. Here the overfitting is very low when we look at the different F1-Scores of the test and train dataset.

6.2 Random Forest

In [13]:
```python
# Initializing the RandomForestClassifier
rf = RandomForestClassifier(random_state=12)

# Let's do a fit on X_train and y_train
rf.fit(X_train, y_train)

# Let's do the predict of X_test
y_pred_proba_test_rf = rf.predict_proba(X_test)[:,1]

# Let's apply a threshold to the probabilities of y_pred_proba_test_rf to
y_pred_test_rf = np.where(y_pred_proba_test_rf >= 0.34, 1, 0)
```

In [14]:
```
plot_confusion_matrix_and_metrics(y_test, y_pred_test_rf, title='Confusior
```



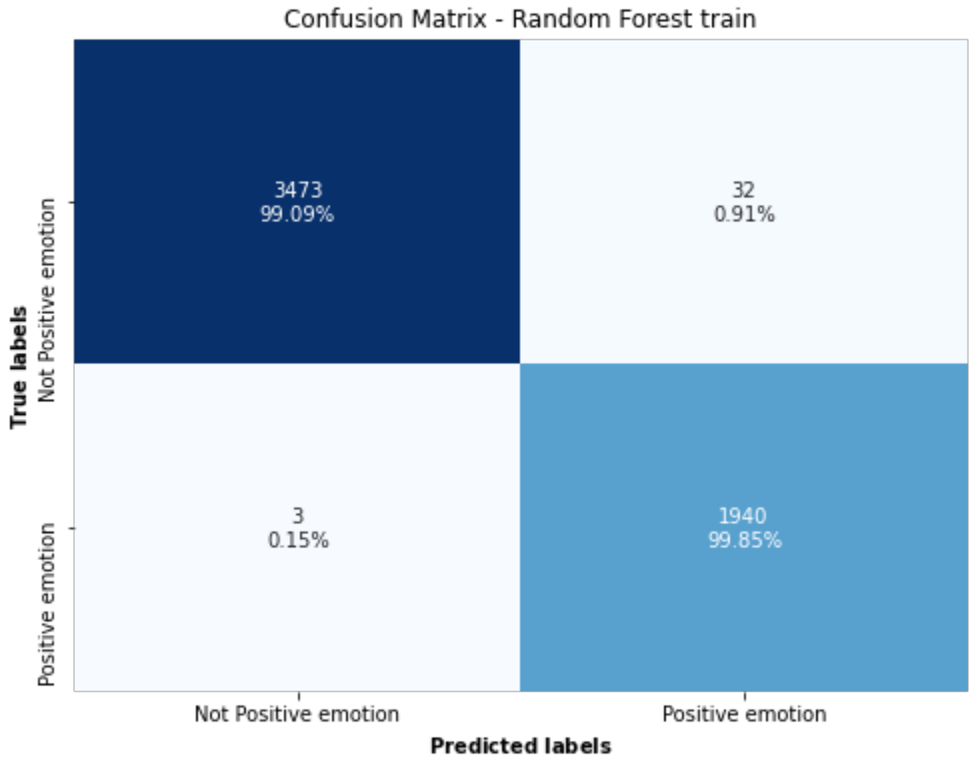

```
Weighted Precision: 0.70
Weighted Recall: 0.68
Weighted F1 Score: 0.68
```

We are going to check the level of overfitting in the Random Forest Model

In [15]:
```
# Let's do the predict of X_train
y_pred_proba_train_rf = rf.predict_proba(X_train)[:,1]

# Let's apply a threshold to the probabilities of y_pred_proba_train_rf tc
y_pred_train_rf = np.where(y_pred_proba_train_rf >= 0.34, 1, 0)
```

In [16]:

```
plot_confusion_matrix_and_metrics(y_train, y_pred_train_rf, title='Confusi
```




```
Weighted Precision: 0.99
Weighted Recall: 0.99
Weighted F1 Score: 0.99
```

As we can see, there is a very significant overfitting as the F1 score of the train dataset is 0.99 whereas the F1 score of the test dataset is 0.63

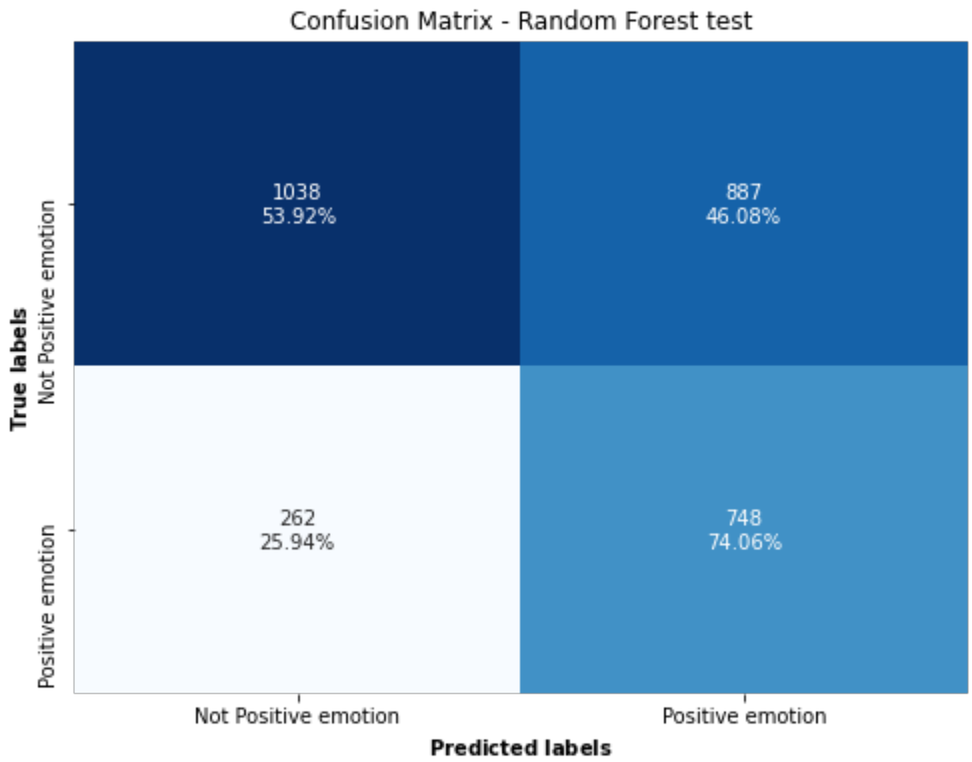
We are now going to do another Random Forest model to try to eliminate the overfitting.

In [17]:

```
# Initializing the RandomForestClassifier
rf_2 = RandomForestClassifier(n_estimators=60, max_depth=20, random_state=

# Let's do a fit on X_train and y_train
rf_2.fit(X_train, y_train)

# Let's do the predict of X_test
y_pred_proba_test_rf_2 = rf_2.predict_proba(X_test)[:,1]

# Let's apply a threshold to the probabilities of y_pred_proba_test_rf_2 t
y_pred_test_rf_2 = np.where(y_pred_proba_test_rf_2 >= 0.34, 1, 0)
```

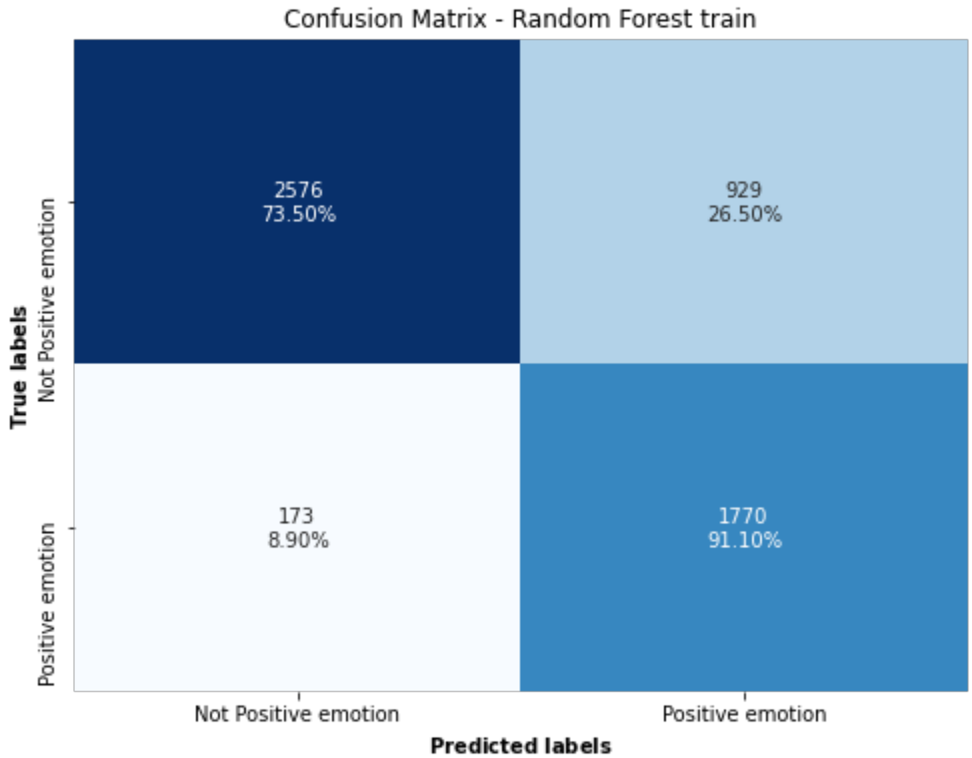
Let's take a look at the overfitting again

In [18]:
```
1 plot_confusion_matrix_and_metrics(y_test, y_pred_test_rf_2, title='Confusi
```




```
Weighted Precision: 0.68
Weighted Recall: 0.61
Weighted F1 Score: 0.62
```

In [19]:
```
# Let's do the predict of X_train
y_pred_train_rf_2 = rf_2.predict_proba(X_train)[:,1]

# Let's apply a threshold to the probabilities of y_pred_train_rf_2 to det
y_pred_train_rf_2 = np.where(y_pred_train_rf_2 >= 0.34, 1, 0)

plot_confusion_matrix_and_metrics(y_train, y_pred_train_rf_2, title='Confu
```




```
Weighted Precision: 0.84
Weighted Recall: 0.80
Weighted F1 Score: 0.80
```

We have been able to mitigate the overfitting, however the results are not satisfactory and so we decide to not use this model from now onwards.

6.3 Neural Networks

In [20]:

```
# Define the neural network model
model = Sequential()

# Adding the input layer and the first hidden layer with dropout
model.add(Dense(units=128, input_dim=X_train.shape[1], activation='relu'))
model.add(Dropout(rate=0.5))  # Dropout with 50% rate

# Adding the second hidden layer with dropout
model.add(Dense(units=64, activation='relu'))
model.add(Dropout(rate=0.5))  # Dropout with 50% rate

# Adding the third hidden layer with dropout
model.add(Dense(units=32, activation='relu'))
model.add(Dropout(rate=0.5))  # Dropout with 50% rate

# Adding the output layer
model.add(Dense(units=2, activation='softmax'))

# Compile the model
model.compile(optimizer='adam', loss='categorical_crossentropy', metrics=[
```

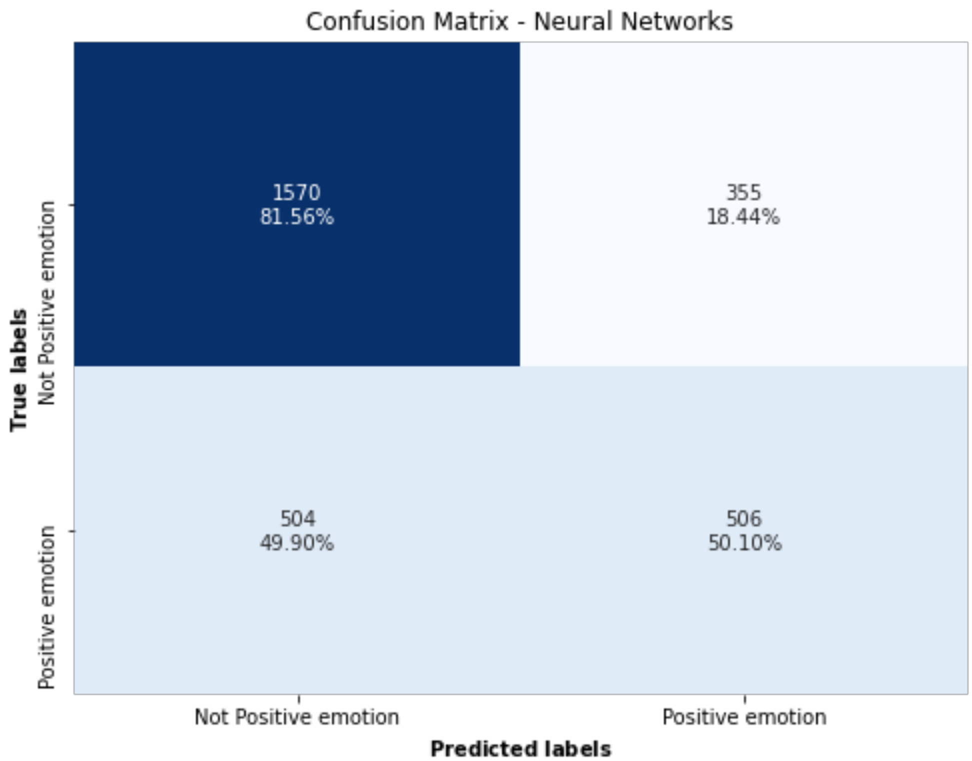
In [21]:
```
# Training the model

# Convert to NumPy arrays
X_train_np = X_train.to_numpy()  # Convert features to a NumPy array
y_train_np = to_categorical(y_train.to_numpy())  # Convert labels to a one

# Doing the same conversion to the test data
X_test_np = X_test.to_numpy()
y_test_np = to_categorical(y_test.to_numpy())

# Doing the fit
model.fit(
    X_train_np,       # Input data (features)
    y_train_np,       # Target data (one-hot encoded labels)
    epochs=10,        # Number of times the model will see the entire data
    batch_size=64,    # Number of samples per gradient update
    validation_data=(X_test_np, y_test_np)  # Validation data (optional)
)
```

```
Epoch 1/10
86/86 [==============================] - 1s 8ms/step - loss: 0.6545 - precisi
on: 0.6360 - val_loss: 0.6241 - val_precision: 0.6559
Epoch 2/10
86/86 [==============================] - 0s 5ms/step - loss: 0.6000 - precisi
on: 0.6529 - val_loss: 0.5797 - val_precision: 0.7077
Epoch 3/10
86/86 [==============================] - 0s 5ms/step - loss: 0.5007 - precisi
on: 0.7599 - val_loss: 0.5814 - val_precision: 0.7128
Epoch 4/10
86/86 [==============================] - 0s 5ms/step - loss: 0.3901 - precisi
on: 0.8489 - val_loss: 0.6304 - val_precision: 0.7080
Epoch 5/10
86/86 [==============================] - 0s 5ms/step - loss: 0.3134 - precisi
on: 0.8798 - val_loss: 0.6894 - val_precision: 0.6906
Epoch 6/10
86/86 [==============================] - 0s 5ms/step - loss: 0.2550 - precisi
on: 0.9099 - val_loss: 0.8006 - val_precision: 0.7039
Epoch 7/10
86/86 [==============================] - 0s 5ms/step - loss: 0.2178 - precisi
on: 0.9211 - val_loss: 0.8458 - val_precision: 0.7015
Epoch 8/10
86/86 [==============================] - 0s 5ms/step - loss: 0.2035 - precisi
on: 0.9269 - val_loss: 0.8625 - val_precision: 0.7002
Epoch 9/10
86/86 [==============================] - 0s 5ms/step - loss: 0.1763 - precisi
on: 0.9325 - val_loss: 0.9105 - val_precision: 0.7056
Epoch 10/10
86/86 [==============================] - 0s 5ms/step - loss: 0.1734 - precisi
on: 0.9341 - val_loss: 0.9359 - val_precision: 0.7073
```

Out[21]:
```
<keras.callbacks.History at 0x1aeffb29f10>
```

In [22]:
```
# Make predictions
predictions_test = model.predict(X_test_np)

# Since the model outputs probabilities, we want to convert these to class
# Find the index of the maximum probability for each sample, which corresp
predicted_classes_test = np.argmax(predictions_test, axis=1)
```

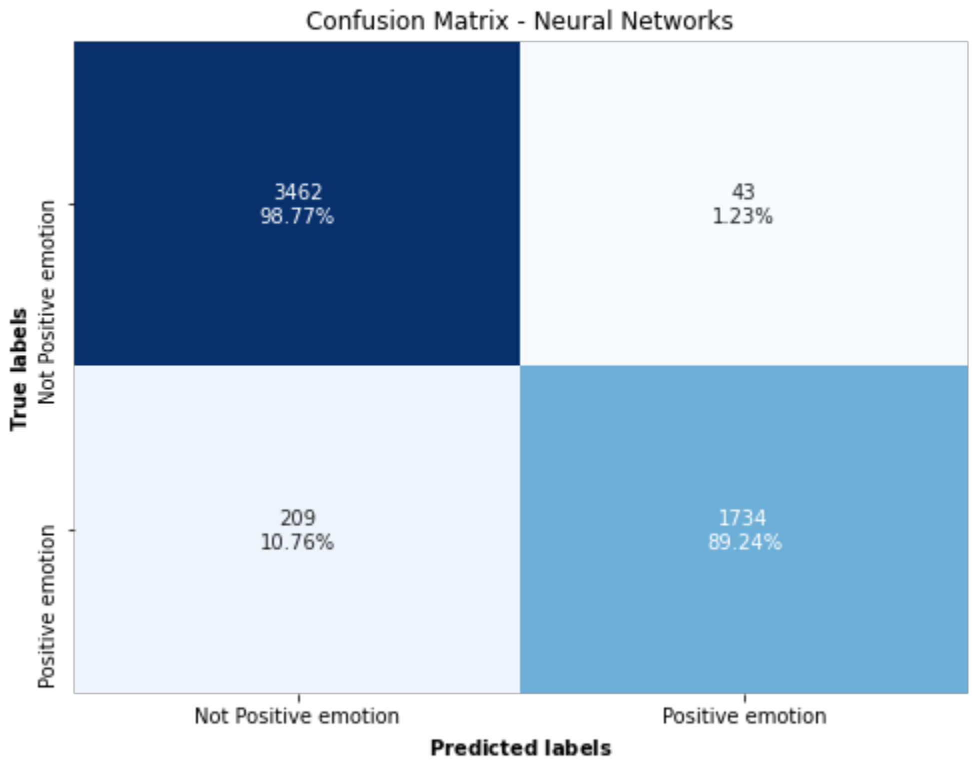
In [23]:
```
plot_confusion_matrix_and_metrics(y_test, predicted_classes_test, title='C
```




```
Weighted Precision: 0.70
Weighted Recall: 0.71
Weighted F1 Score: 0.70
```

In [24]:
```
# Make predictions
predictions_train = model.predict(X_train_np)

# Since the model outputs probabilities, we want to convert these to class
# Find the index of the maximum probability for each sample, which corresp
predicted_classes_train = np.argmax(predictions_train, axis=1)
```

In [25]:
```
plot_confusion_matrix_and_metrics(y_train, predicted_classes_train, title=
```



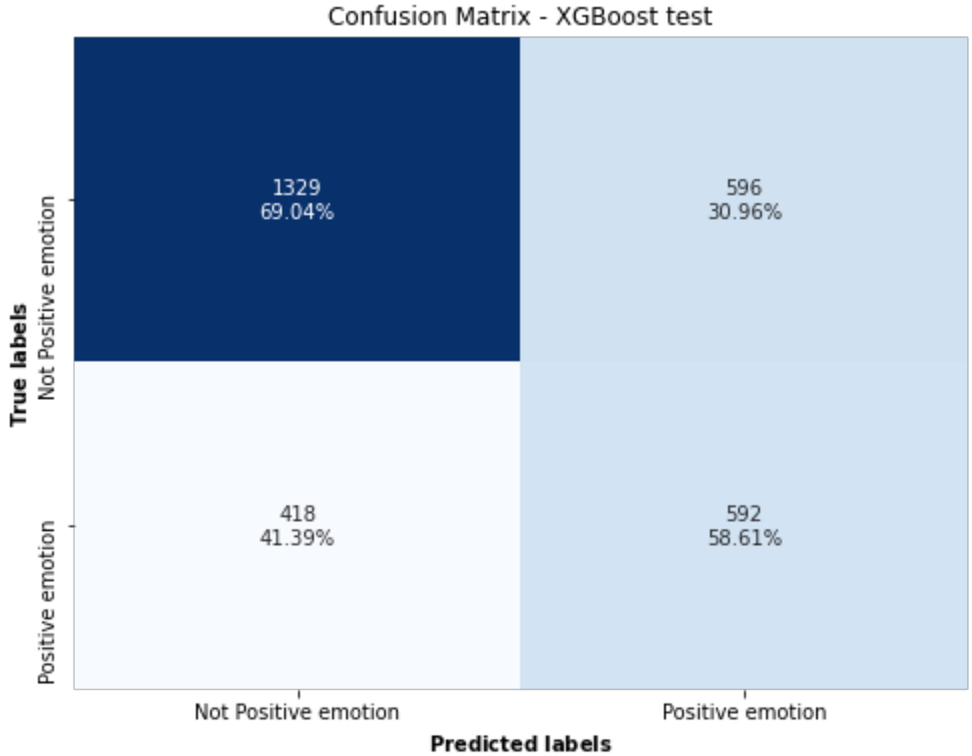

```
Weighted Precision: 0.95
Weighted Recall: 0.95
Weighted F1 Score: 0.95
```

Based on the metrics that are printed by the neural network, we can see there is some overfitting but not too exagerated. As we iterate over the model, we will proceed to mitigate the overfitting.

6.4 Xgboost

In [26]:
```
# Initializing the XGBClassifier
xgb = XGBClassifier(random_state=12)

# Let's do a fit on X_train and y_train
xgb.fit(X_train, y_train)

# Let's do the predict of X_test
y_pred_proba_test_xgb = xgb.predict_proba(X_test)[:,1]

# Let's apply a threshold to the probabilities of y_pred_proba_test_xgb to
y_pred_test_xgb = np.where(y_pred_proba_test_xgb >= 0.34, 1, 0)
```

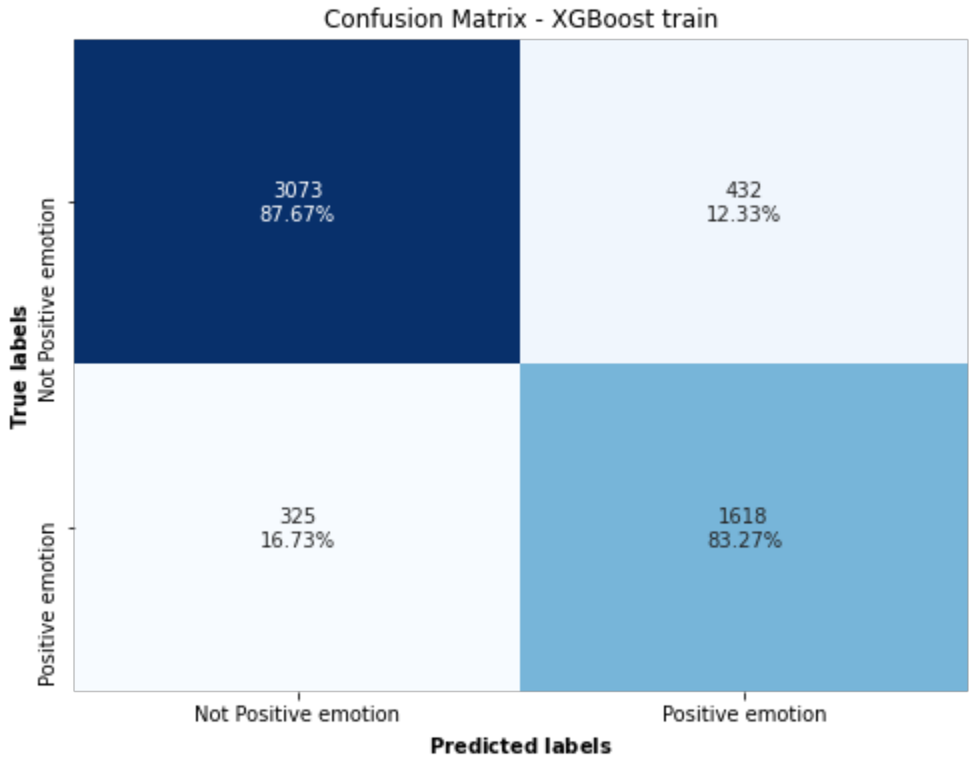
In [27]:
```
1 plot_confusion_matrix_and_metrics(y_test, y_pred_test_xgb, title='Confusic
```




```
Weighted Precision: 0.67
Weighted Recall: 0.65
Weighted F1 Score: 0.66
```

We are going to check the level of overfitting in the XGBoost

In [28]:
```
# Let's do the predict of X_train
y_pred_proba_train_xgb = xgb.predict_proba(X_train)[:,1]

# Let's apply a threshold to the probabilities of y_pred_proba_train_xgb t
y_pred_train_xgb = np.where(y_pred_proba_train_xgb >= 0.34, 1, 0)

plot_confusion_matrix_and_metrics(y_train, y_pred_train_xgb, title='Confus
```




```
Weighted Precision: 0.86
Weighted Recall: 0.86
Weighted F1 Score: 0.86
```

As we can see the overfitting is high for XGBoost judging the F1-scores of the train and test datasets

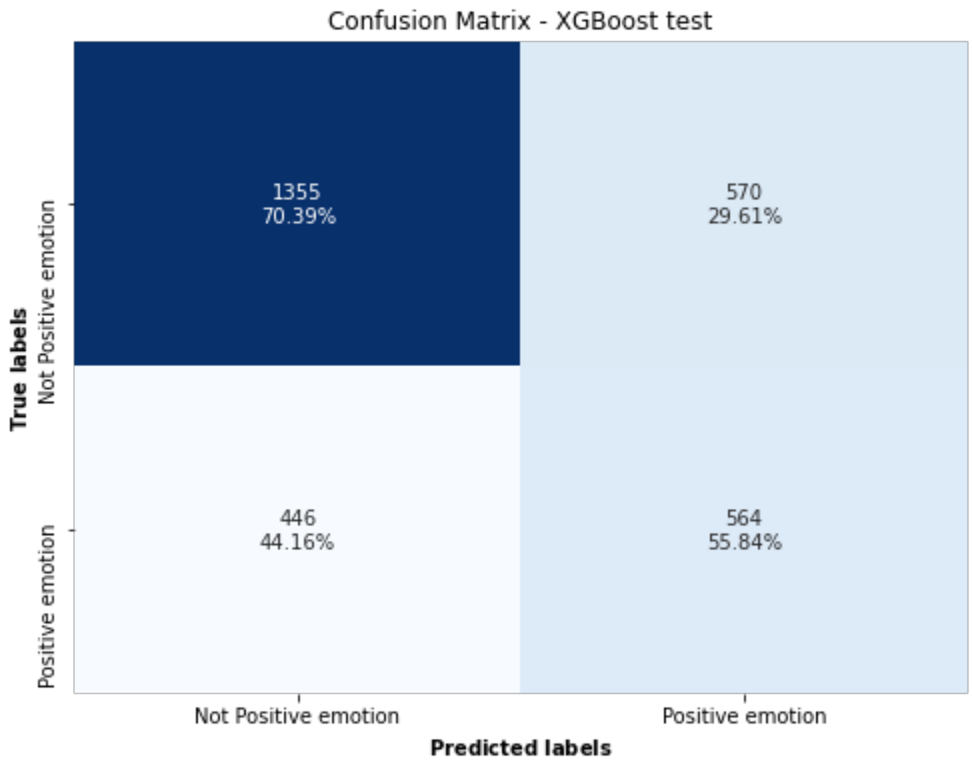
We are now going to create another XGBoost model to try to reduce the overfitting

In [29]:

```
# Initializing the XGBClassifier
xgb_2 = XGBClassifier(random_state=12, max_depth=3, max_leaves=4)

# Let's do a fit on X_train and y_train
xgb_2.fit(X_train, y_train)

# Let's do the predict of X_test
y_pred_proba_test_xgb_2 = xgb_2.predict_proba(X_test)[:,1]

# Let's apply a threshold to the probabilities of y_pred_proba_test_xgb_2
y_pred_test_xgb_2 = np.where(y_pred_proba_test_xgb_2 >= 0.34, 1, 0)
```

In [30]:

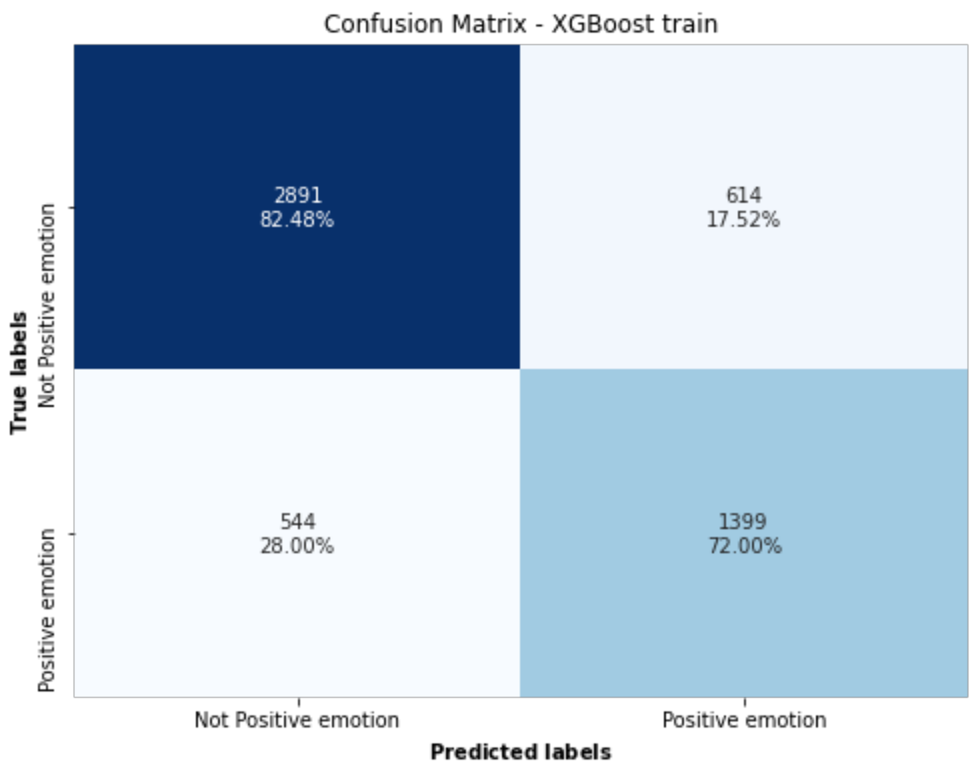
```
plot_confusion_matrix_and_metrics(y_test, y_pred_test_xgb_2, title='Confus
```




```
Weighted Precision: 0.66
Weighted Recall: 0.65
Weighted F1 Score: 0.66
```

Let's look at the overfitting now

In [31]:
```
# Let's do the predict of X_train
y_pred_proba_train_xgb_2 = xgb_2.predict_proba(X_train)[:,1]

# Let's apply a threshold to the probabilities of y_pred_proba_train_xgb_2
y_pred_train_xgb_2 = np.where(y_pred_proba_train_xgb_2 >= 0.34, 1, 0)

plot_confusion_matrix_and_metrics(y_train, y_pred_train_xgb_2, title='Conf
```




```
Weighted Precision: 0.79
Weighted Recall: 0.79
Weighted F1 Score: 0.79
```

6.5 Metric comparison

In [32]:
```
results_table = precision_results_table(y_train, y_test,
                                        y_pred_train_lr, y_pred_test_lr,
                                        y_pred_train_rf, y_pred_test_rf, y
                                        predicted_classes_train, predictec
                                        y_pred_train_xgb, y_pred_test_xgb,
results_table
```

Out[32]:

	Model	Precision (Train)	Precision (Test)	Difference (Train-Test)
0	Logistic Regression	0.831783	0.694176	0.137608
1	Random Forest 1	0.993657	0.697191	0.296467
2	Random Forest 2	0.836755	0.681126	0.155628
3	Neural Networks	0.954742	0.698730	0.256011
4	XGBoost 1	0.863310	0.670429	0.192881
5	XGBoost 2	0.789329	0.664607	0.124722

It seems that the best models when considering the Train's and Test's Precision are XGBoost, Logistic Regression and the Random Forest

6.6 Fine-Tunning

Please note that we have commented the codes of the hyper-tunning for the different models given that their times of execution are too long. We have added the code of the hyper-tunning with the best parameters already for every model.

6.6.1 XGBoost

Given the nature of our business case, we decide that the metric we wish to optimize is the precision. The reason being is that our business intends to deduce from the tweets, that we've classified as having a positive feeling, the features of the technological products that bring happy emotions.

The more false positives that we have, the more harmful it will be to the nature of our business because it will affect our analysis that we will do in the future to determine the positive features of the technological products. Thus, the metric we want to optimize is the precision to reduce our false positives.

Nonetheless, we will also look at the fine-tunning focusing on optimizing recall and the f1-score

Let's do a fine-tunning focusing on optimizing precision

In [33]:

```
# # Define the parameter grid for XGBoost
# param_grid = {
#     'max_depth': [3, 7, 10],  # Maximum depth of a tree
#     'n_estimators': [100, 150],  # Number of trees
#     'colsample_bytree': [0.8, 1.0],  # Fraction of features used for eac
#     'reg_lambda': [1, 2]  # L2 regularization term
# }

# # Create the XGBoost model
# xgb_model = XGBClassifier(use_label_encoder=False, eval_metric='logloss'

# # Define the custom weighted precision scoring function
# precision_scorer = make_scorer(precision_score, average='weighted')

# # Initialize GridSearchCV
# grid_search_precision = GridSearchCV(estimator=xgb_model, param_grid=par
#                              scoring=precision_scorer, cv=3, verbose=1, n_

# # Fit the grid search to the training data
# grid_search_precision.fit(X_train, y_train)

# # Get the best model from the grid search
# best_model_xgb_precision = grid_search_precision.best_estimator_

# # Make predictions on the train set (probabilities)
# y_pred_proba_train_xgb_precision = best_model_xgb_precision.predict_prob

# # Apply custom threshold to convert probabilities into binary prediction
# y_pred_train_xgb_tuned_precision = np.where(y_pred_proba_train_xgb_preci

# # Make predictions on the test set (probabilities)
# y_pred_proba_test_xgb_precision = best_model_xgb_precision.predict_proba

# # Apply custom threshold to convert probabilities into binary prediction
# y_pred_test_xgb_tuned_precision = np.where(y_pred_proba_test_xgb_precisi

# # Evaluate weighted precision on the test set using the custom threshold
# precision_test = precision_score(y_test, y_pred_test_xgb_tuned_precision
# print(f"Weighted Precision on test set: {precision_test:.4f}")

# # Display the best parameters found by the grid search
# print(f"Best parameters found by GridSearchCV: {grid_search_precision.be

```

After running the Gridsearch optimizing the precision metric, the best parameters that resulted where:

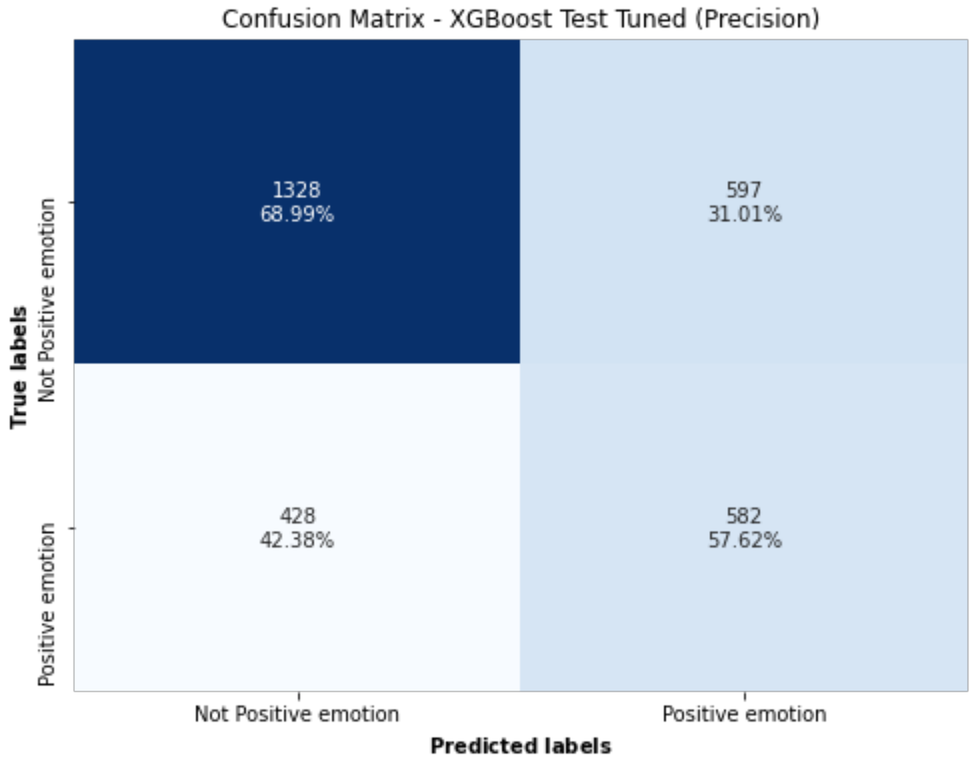
```
Fitting 3 folds for each of 24 candidates, totalling 72 fits
Weighted Precision on test set: 0.6659
Best parameters found by GridSearchCV: {'colsample_bytree': 0.8, 'max_depth': 3, 'n_estimators': 100, 'reg_lambda': 2}
```

In [34]:

```
# Define the parameter grid for XGBoost
param_grid = {
    'max_depth': [3],  # Maximum depth of a tree
    'n_estimators': [100],  # Number of trees
    'colsample_bytree': [0.8],  # Fraction of features used for each tree
    'reg_lambda': [2] # L2 regularization term
}

# Create the XGBoost model
xgb_model = XGBClassifier(use_label_encoder=False, eval_metric='logloss')

# Define the custom weighted precision scoring function
precision_scorer = make_scorer(precision_score, average='weighted')

# Initialize GridSearchCV
grid_search_precision = GridSearchCV(estimator=xgb_model, param_grid=param
                           scoring=precision_scorer, cv=3, verbose=1, n_jo

# Fit the grid search to the training data
grid_search_precision.fit(X_train, y_train)

# Get the best model from the grid search
best_model_xgb_precision = grid_search_precision.best_estimator_

# Make predictions on the train set (probabilities)
y_pred_proba_train_xgb_precision = best_model_xgb_precision.predict_proba(

# Apply custom threshold to convert probabilities into binary predictions
y_pred_train_xgb_tuned_precision = np.where(y_pred_proba_train_xgb_precisi

# Make predictions on the test set (probabilities)
y_pred_proba_test_xgb_precision = best_model_xgb_precision.predict_proba(X

# Apply custom threshold to convert probabilities into binary predictions
y_pred_test_xgb_tuned_precision = np.where(y_pred_proba_test_xgb_precision

# Evaluate weighted precision on the test set using the custom threshold
precision_test = precision_score(y_test, y_pred_test_xgb_tuned_precision,
print(f"Weighted Precision on test set: {precision_test:.4f}")

# Display the best parameters found by the grid search
print(f"Best parameters found by GridSearchCV: {grid_search_precision.best

```

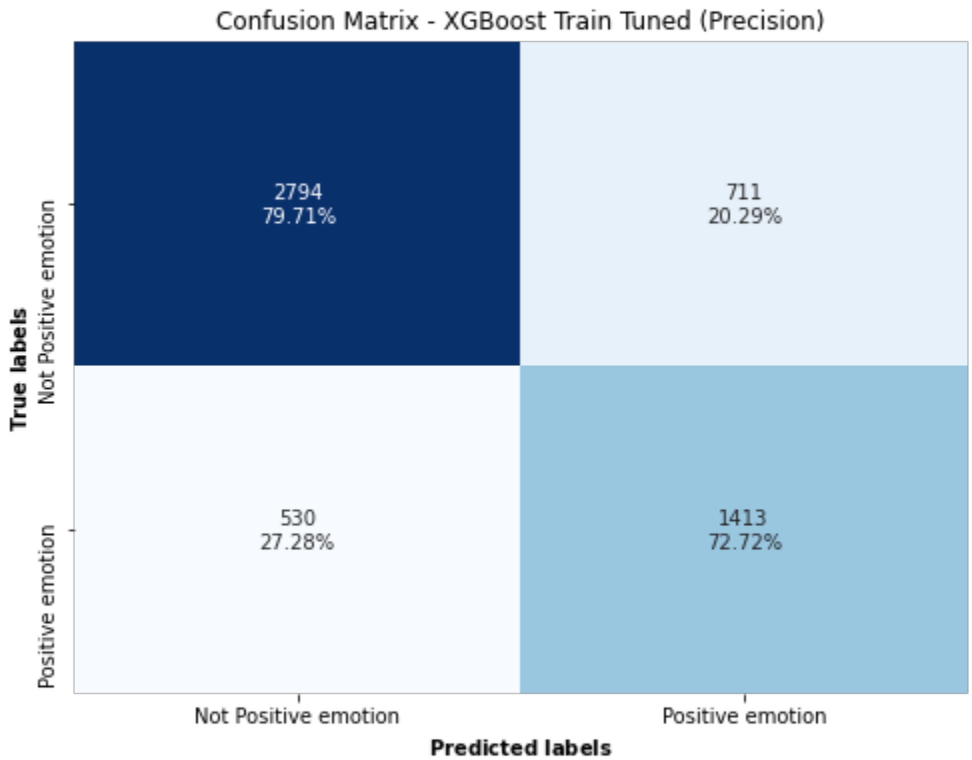
```
Fitting 3 folds for each of 1 candidates, totalling 3 fits
Weighted Precision on test set: 0.6659
Best parameters found by GridSearchCV: {'colsample_bytree': 0.8, 'max_depth':
3, 'n_estimators': 100, 'reg_lambda': 2}
```

In [35]:
```
1 plot_confusion_matrix_and_metrics(y_test, y_pred_test_xgb_tuned_precision,
```




```
Weighted Precision: 0.67
Weighted Recall: 0.65
Weighted F1 Score: 0.66
```

In [36]:
```
1 plot_confusion_matrix_and_metrics(y_train, y_pred_train_xgb_tuned_precisic
```




```
Weighted Precision: 0.78
Weighted Recall: 0.77
Weighted F1 Score: 0.77
```

We are going to look out for different metric scores to see what the metrics turn out being:

F1-Score

In [37]:

```
# # Define the parameter grid for XGBoost
# param_grid = {
#     'max_depth': [3, 7, 10],  # Maximum depth of a tree
#     'n_estimators': [100, 150],  # Number of trees
#     'colsample_bytree': [0.8, 1.0],  # Fraction of features used for eac
#     'reg_lambda': [1, 2]  # L2 regularization term
# }

# # Create the XGBoost model
# xgb_model = XGBClassifier(use_label_encoder=False, eval_metric='logloss'

# # Define the custom weighted F1 scoring function
# f1_scorer = make_scorer(f1_score, average='weighted')

# # Initialize GridSearchCV
# grid_search_f1 = GridSearchCV(estimator=xgb_model, param_grid=param_grid
#                              scoring=f1_scorer, cv=3, verbose=1, n_jobs=-1

# # Fit the grid search to the training data
# grid_search_f1.fit(X_train, y_train)

# # Get the best model from the grid search
# best_model_xgb_f1 = grid_search_f1.best_estimator_

# # Make predictions on the train set (probabilities)
# y_pred_proba_train_xgb_f1 = best_model_xgb_f1.predict_proba(X_train)[:,

# # Apply custom threshold to convert probabilities into binary prediction
# y_pred_train_xgb_tuned_f1 = np.where(y_pred_proba_train_xgb_f1 >= 0.34,

# # Evaluate weighted F1-score on the training set using the custom thresh
# f1_train = f1_score(y_train, y_pred_train_xgb_tuned_f1, average='weighte
# print(f"Weighted F1-score on training set: {f1_train:.4f}")

# # Make predictions on the test set (probabilities)
# y_pred_proba_test_xgb_f1 = best_model_xgb_f1.predict_proba(X_test)[:, 1]

# # Apply custom threshold to convert probabilities into binary prediction
# y_pred_test_xgb_tuned_f1 = np.where(y_pred_proba_test_xgb_f1 >= 0.34, 1,

# # Evaluate weighted F1-score on the test set using the custom threshold
# f1_test = f1_score(y_test, y_pred_test_xgb_tuned_f1, average='weighted')
# print(f"Weighted F1-score on test set: {f1_test:.4f}")

# # Display the best parameters found by the grid search
# print(f"Best parameters found by GridSearchCV: {grid_search_f1.best_para

```

After running the Gridsearch optimizing the f1-score metric, the best parameters that resulted where:

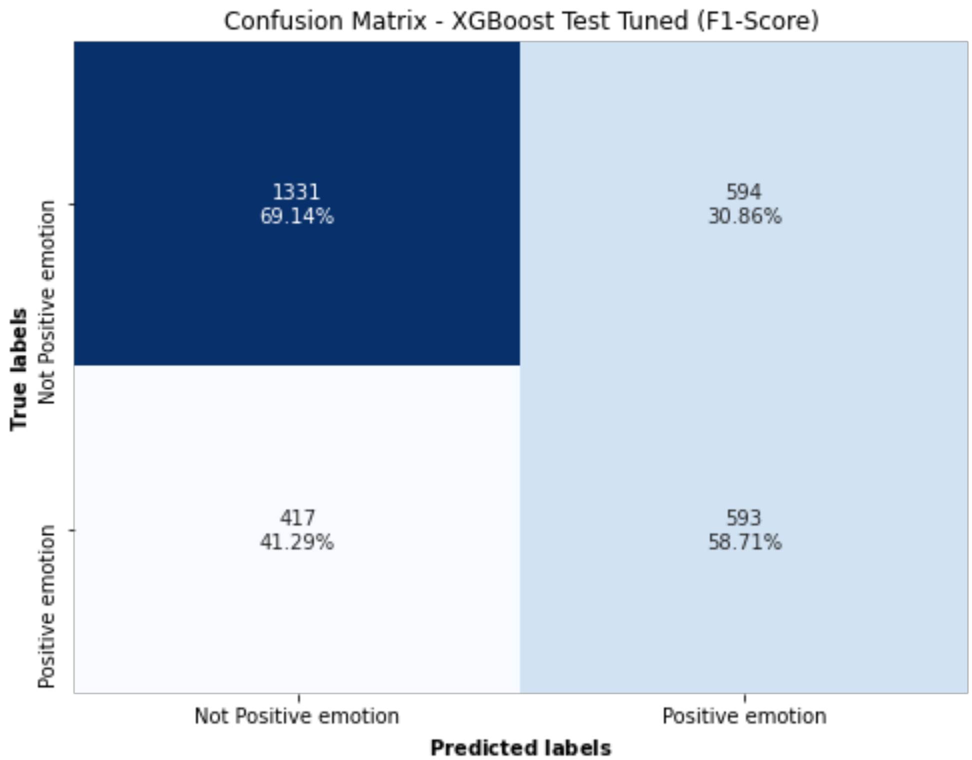
```
Fitting 3 folds for each of 24 candidates, totalling 72 fits
Weighted F1-score on training set: 0.8990
Weighted F1-score on test set: 0.6611
Best parameters found by GridSearchCV: {'colsample_bytree': 0.8, 'max_depth': 7, 'n_estimators': 150, 'reg_lambda': 1}
```

In [38]:

```python
# Define the parameter grid for XGBoost
param_grid = {
    'max_depth': [7],  # Maximum depth of a tree
    'n_estimators': [150],  # Number of trees
    'colsample_bytree': [0.8],  # Fraction of features used for each tree
    'reg_lambda': [1]  # L2 regularization term
}

# Create the XGBoost model
xgb_model = XGBClassifier(use_label_encoder=False, eval_metric='logloss')

# Define the custom weighted F1 scoring function
f1_scorer = make_scorer(f1_score, average='weighted')

# Initialize GridSearchCV
grid_search_f1 = GridSearchCV(estimator=xgb_model, param_grid=param_grid,
                              scoring=f1_scorer, cv=3, verbose=1, n_jobs=-1)

# Fit the grid search to the training data
grid_search_f1.fit(X_train, y_train)

# Get the best model from the grid search
best_model_xgb_f1 = grid_search_f1.best_estimator_

# Make predictions on the train set (probabilities)
y_pred_proba_train_xgb_f1 = best_model_xgb_f1.predict_proba(X_train)[:, 1]

# Apply custom threshold to convert probabilities into binary predictions
y_pred_train_xgb_tuned_f1 = np.where(y_pred_proba_train_xgb_f1 >= 0.34, 1,

# Evaluate weighted F1-score on the training set using the custom threshol
f1_train = f1_score(y_train, y_pred_train_xgb_tuned_f1, average='weighted'
print(f"Weighted F1-score on training set: {f1_train:.4f}")

# Make predictions on the test set (probabilities)
y_pred_proba_test_xgb_f1 = best_model_xgb_f1.predict_proba(X_test)[:, 1]

# Apply custom threshold to convert probabilities into binary predictions
y_pred_test_xgb_tuned_f1 = np.where(y_pred_proba_test_xgb_f1 >= 0.34, 1, 0

# Evaluate weighted F1-score on the test set using the custom threshold
f1_test = f1_score(y_test, y_pred_test_xgb_tuned_f1, average='weighted')
print(f"Weighted F1-score on test set: {f1_test:.4f}")

# Display the best parameters found by the grid search
print(f"Best parameters found by GridSearchCV: {grid_search_f1.best_params
```

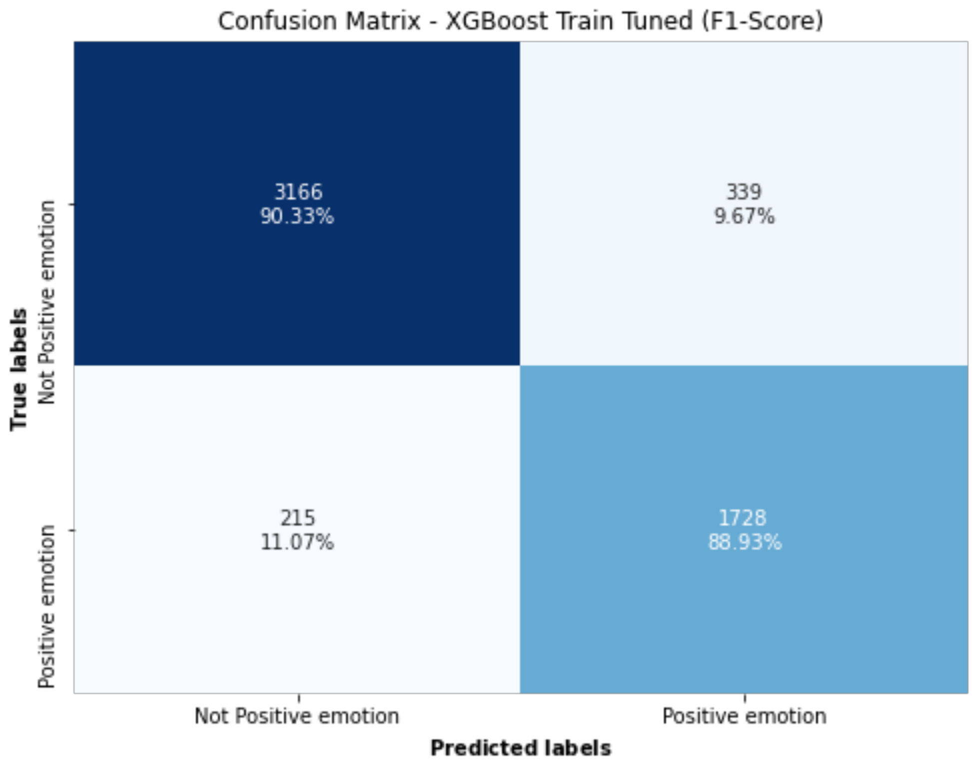
```
Fitting 3 folds for each of 1 candidates, totalling 3 fits
Weighted F1-score on training set: 0.8990
Weighted F1-score on test set: 0.6611
Best parameters found by GridSearchCV: {'colsample_bytree': 0.8, 'max_depth':
7, 'n_estimators': 150, 'reg_lambda': 1}
```

In [39]:
```
1 plot_confusion_matrix_and_metrics(y_test, y_pred_test_xgb_tuned_f1, title=
```




```
Weighted Precision: 0.67
Weighted Recall: 0.66
Weighted F1 Score: 0.66
```

In [40]:
```
1 plot_confusion_matrix_and_metrics(y_train, y_pred_train_xgb_tuned_f1, titl
```




```
Weighted Precision: 0.90
Weighted Recall: 0.90
Weighted F1 Score: 0.90
```

Recall

In [41]:

```
# # Define the parameter grid for XGBoost
# param_grid = {
#     'max_depth': [3, 7, 10],  # Maximum depth of a tree
#     'n_estimators': [100, 150],  # Number of trees
#     'colsample_bytree': [0.8, 1.0],  # Fraction of features used for eac
#     'reg_lambda': [1, 2]  # L2 regularization term
# }

# # Create the XGBoost model
# xgb_model = XGBClassifier(use_label_encoder=False, eval_metric='logloss'

# # Define the custom weighted recall scoring function
# recall_scorer = make_scorer(recall_score, average='weighted')

# # Initialize GridSearchCV
# grid_search_recall = GridSearchCV(estimator=xgb_model, param_grid=param_
#                              scoring=recall_scorer, cv=3, verbose=1, n_job

# # Fit the grid search to the training data
# grid_search_recall.fit(X_train, y_train)

# # Get the best model from the grid search
# best_model_xgb_recall = grid_search_recall.best_estimator_

# # Make predictions on the train set (probabilities)
# y_pred_proba_train_xgb_recall = best_model_xgb_recall.predict_proba(X_tr

# # Apply custom threshold to convert probabilities into binary prediction
# y_pred_train_xgb_tuned_recall = np.where(y_pred_proba_train_xgb_recall >

# # Evaluate weighted recall on the training set using the custom threshol
# recall_train = recall_score(y_train, y_pred_train_xgb_tuned_recall, aver
# print(f"Weighted Recall on training set: {recall_train:.4f}")

# # Make predictions on the test set (probabilities)
# y_pred_proba_test_xgb_recall = best_model_xgb_recall.predict_proba(X_tes

# # Apply custom threshold to convert probabilities into binary prediction
# y_pred_test_xgb_tuned_recall = np.where(y_pred_proba_test_xgb_recall >=

# # Evaluate weighted recall on the test set using the custom threshold
# recall_test = recall_score(y_test, y_pred_test_xgb_tuned_recall, average
# print(f"Weighted Recall on test set: {recall_test:.4f}")

# # Display the best parameters found by the grid search
# print(f"Best parameters found by GridSearchCV: {grid_search_recall.best_

```

After running the Gridsearch optimizing the recall metric, the best parameters that resulted where:

```
47
```

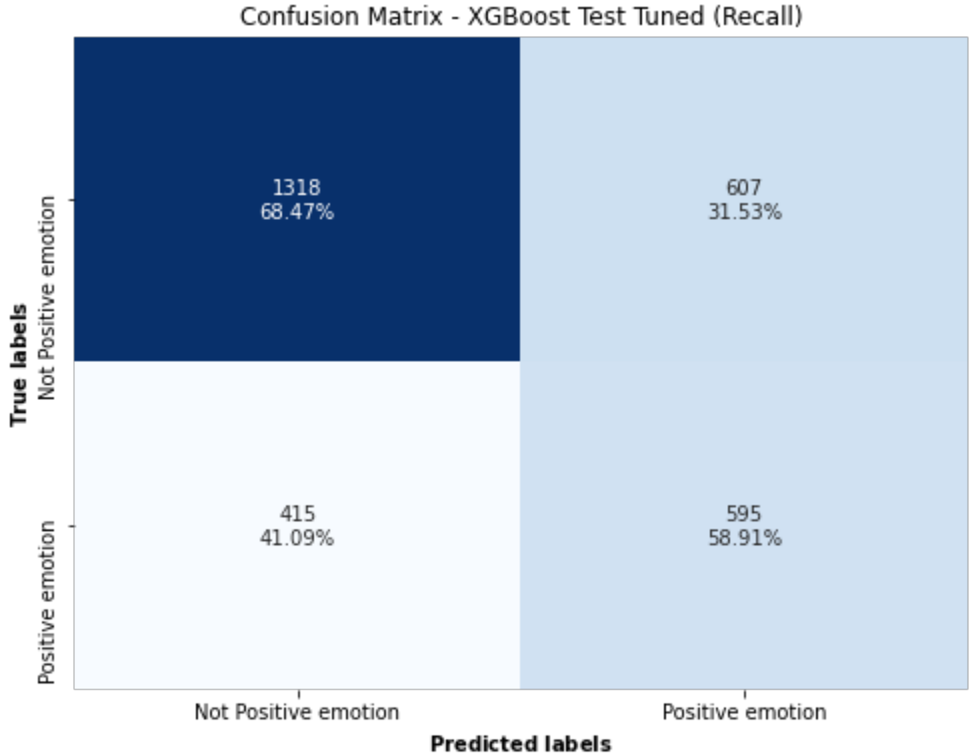
```
Fitting 3 folds for each of 24 candidates, totalling 72 fits
Weighted Recall on training set: 0.8724
Weighted Recall on test set: 0.6518
Best parameters found by GridSearchCV: {'colsample_bytree': 0.8, 'max_depth': 7, 'n_estimators': 100, 'reg_lambda': 1}
```

In [42]:

```
# Define the parameter grid for XGBoost
param_grid = {
    'max_depth': [7],  # Maximum depth of a tree
    'n_estimators': [100],  # Number of trees
    'colsample_bytree': [0.8],  # Fraction of features used for each tree
    'reg_lambda': [1]  # L2 regularization term
}

# Create the XGBoost model
xgb_model = XGBClassifier(use_label_encoder=False, eval_metric='logloss')

# Define the custom weighted recall scoring function
recall_scorer = make_scorer(recall_score, average='weighted')

# Initialize GridSearchCV
grid_search_recall = GridSearchCV(estimator=xgb_model, param_grid=param_gr
                           scoring=recall_scorer, cv=3, verbose=1, n_jobs=

# Fit the grid search to the training data
grid_search_recall.fit(X_train, y_train)

# Get the best model from the grid search
best_model_xgb_recall = grid_search_recall.best_estimator_

# Make predictions on the train set (probabilities)
y_pred_proba_train_xgb_recall = best_model_xgb_recall.predict_proba(X_trai

# Apply custom threshold to convert probabilities into binary predictions
y_pred_train_xgb_tuned_recall = np.where(y_pred_proba_train_xgb_recall >=

# Evaluate weighted recall on the training set using the custom threshold
recall_train = recall_score(y_train, y_pred_train_xgb_tuned_recall, averag
print(f"Weighted Recall on training set: {recall_train:.4f}")

# Make predictions on the test set (probabilities)
y_pred_proba_test_xgb_recall = best_model_xgb_recall.predict_proba(X_test)

# Apply custom threshold to convert probabilities into binary predictions
y_pred_test_xgb_tuned_recall = np.where(y_pred_proba_test_xgb_recall >= 0.

# Evaluate weighted recall on the test set using the custom threshold
recall_test = recall_score(y_test, y_pred_test_xgb_tuned_recall, average='
print(f"Weighted Recall on test set: {recall_test:.4f}")

# Display the best parameters found by the grid search
print(f"Best parameters found by GridSearchCV: {grid_search_recall.best_pa

```

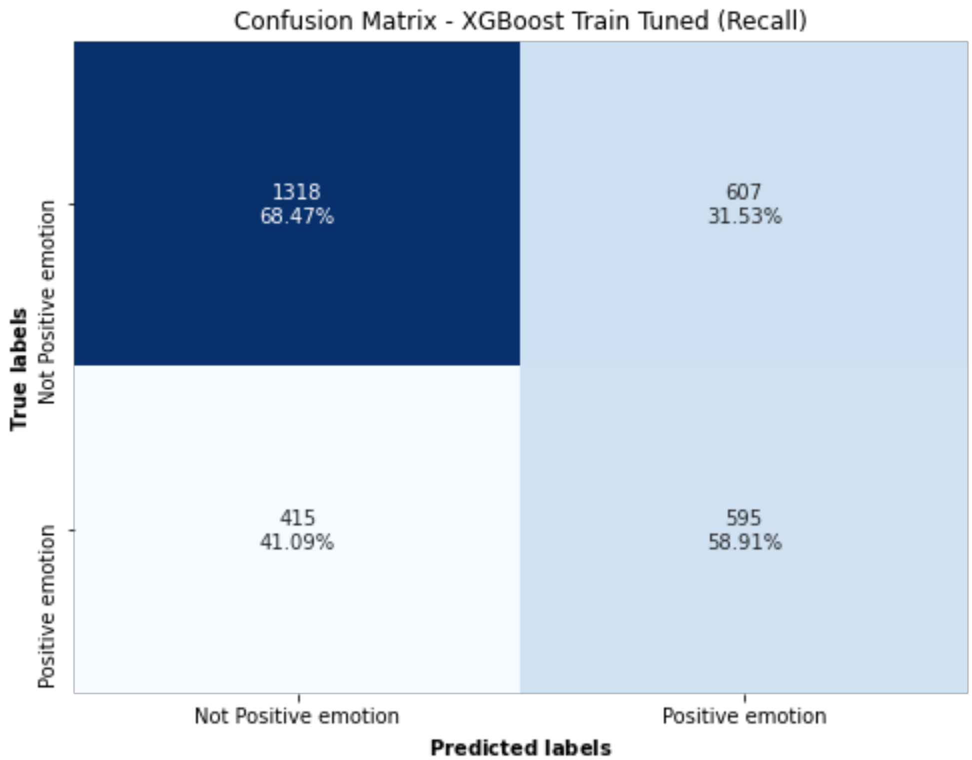
```
Fitting 3 folds for each of 1 candidates, totalling 3 fits
Weighted Recall on training set: 0.8724
Weighted Recall on test set: 0.6518
Best parameters found by GridSearchCV: {'colsample_bytree': 0.8, 'max_depth':
7, 'n_estimators': 100, 'reg_lambda': 1}
```

In [43]:
```
1 plot_confusion_matrix_and_metrics(y_test, y_pred_test_xgb_tuned_recall, ti
```




```
Weighted Precision: 0.67
Weighted Recall: 0.65
Weighted F1 Score: 0.66
```

In [44]:
```
1 plot_confusion_matrix_and_metrics(y_test, y_pred_test_xgb_tuned_recall, ti
```




```
Weighted Precision: 0.67
Weighted Recall: 0.65
Weighted F1 Score: 0.66
```

6.6.2 Random Forest

In [45]:
```
# # Define the parameter grid for Random Forest
# param_grid_rf = {
#     'n_estimators': [100, 150],  # Number of trees
#     'max_depth': [5, 10, 20],  # Maximum depth of the tree
#     'min_samples_leaf': [4, 6],  # Minimum number of samples required to
# }

# # Create the Random Forest model
# rf_model = RandomForestClassifier()

# # Define the custom precision scoring function
# precision_scorer = make_scorer(precision_score, average='weighted')

# # Initialize GridSearchCV
# grid_search_rf = GridSearchCV(estimator=rf_model, param_grid=param_grid_
#                                scoring=precision_scorer, cv=3, verbose=1,

# # Fit the grid search to the training data
# grid_search_rf.fit(X_train, y_train)

# # Get the best model from the grid search
# best_model_rf = grid_search_rf.best_estimator_

# # Make predictions on the train set
# y_pred_proba_train_rf = best_model_rf.predict_proba(X_train)[:, 1]  # Ge

# # Apply custom threshold to convert probabilities into binary prediction
# y_pred_train_rf_tuned = np.where(y_pred_proba_train_rf >= 0.34, 1, 0)

# # Make predictions on the test set
# y_pred_proba_test_rf = best_model_rf.predict_proba(X_test)[:, 1]  # Get

# # Apply custom threshold to convert probabilities into binary prediction
# y_pred_test_rf_tuned = np.where(y_pred_proba_test_rf >= 0.34, 1, 0)

# # Evaluate precision on the test set using the custom threshold
# precision_test_rf = precision_score(y_test, y_pred_test_rf_tuned, averag
# print(f"Precision on test set (Random Forest): {precision_test_rf:.4f}")

# # Display the best parameters found by the grid search
# print(f"Best parameters found by GridSearchCV (Random Forest): {grid_sea
```

After running the Gridsearch optimizing the precision metric, the best parameters that resulted where:

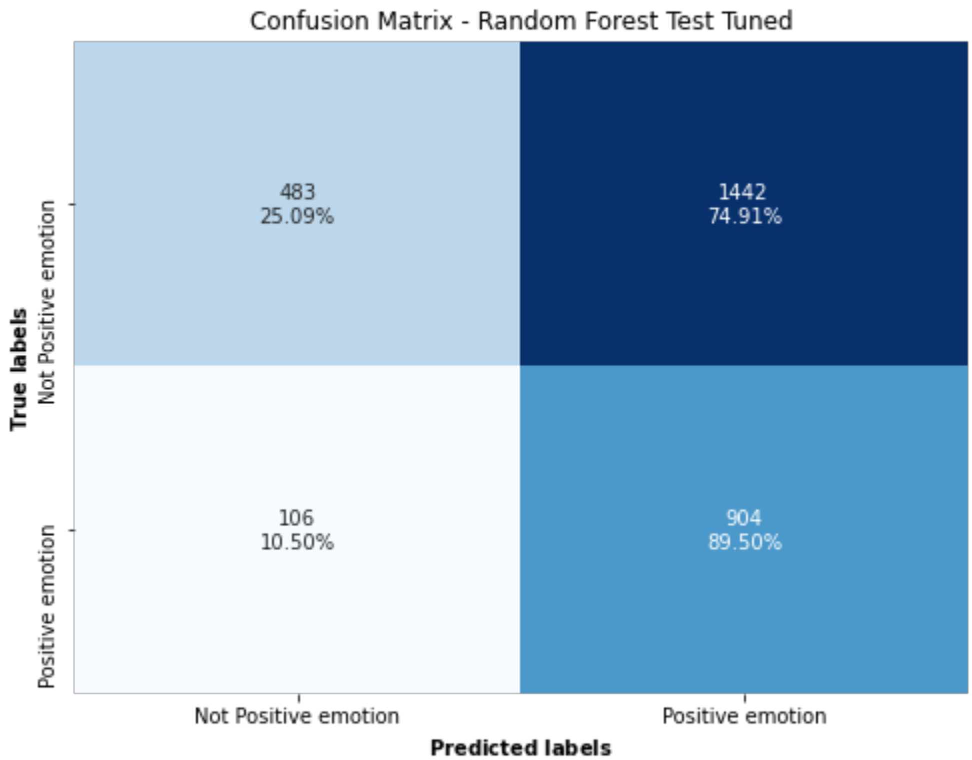
```
Fitting 3 folds for each of 12 candidates, totalling 36 fits
Precision on test set (Random Forest): 0.6700
Best parameters found by GridSearchCV (Random Forest): {'max_depth': 10, 'min_samples_leaf': 6, 'n_estimators': 150}
```

In [46]:

```python
# Define the parameter grid for Random Forest
param_grid_rf = {
    'n_estimators': [150],  # Number of trees
    'max_depth': [10],  # Maximum depth of the tree
    'min_samples_leaf': [6],  # Minimum number of samples required to be c
}

# Create the Random Forest model
rf_model = RandomForestClassifier()

# Define the custom precision scoring function
precision_scorer = make_scorer(precision_score, average='weighted')

# Initialize GridSearchCV
grid_search_rf = GridSearchCV(estimator=rf_model, param_grid=param_grid_rf
                              scoring=precision_scorer, cv=3, verbose=1, r

# Fit the grid search to the training data
grid_search_rf.fit(X_train, y_train)

# Get the best model from the grid search
best_model_rf = grid_search_rf.best_estimator_

# Make predictions on the train set
y_pred_proba_train_rf = best_model_rf.predict_proba(X_train)[:, 1]  # Get

# Apply custom threshold to convert probabilities into binary predictions
y_pred_train_rf_tuned = np.where(y_pred_proba_train_rf >= 0.34, 1, 0)

# Make predictions on the test set
y_pred_proba_test_rf = best_model_rf.predict_proba(X_test)[:, 1]  # Get pr

# Apply custom threshold to convert probabilities into binary predictions
y_pred_test_rf_tuned = np.where(y_pred_proba_test_rf >= 0.34, 1, 0)

# Evaluate precision on the test set using the custom threshold
precision_test_rf = precision_score(y_test, y_pred_test_rf_tuned, average=
print(f"Precision on test set (Random Forest): {precision_test_rf:.4f}")

# Display the best parameters found by the grid search
print(f"Best parameters found by GridSearchCV (Random Forest): {grid_searc
```

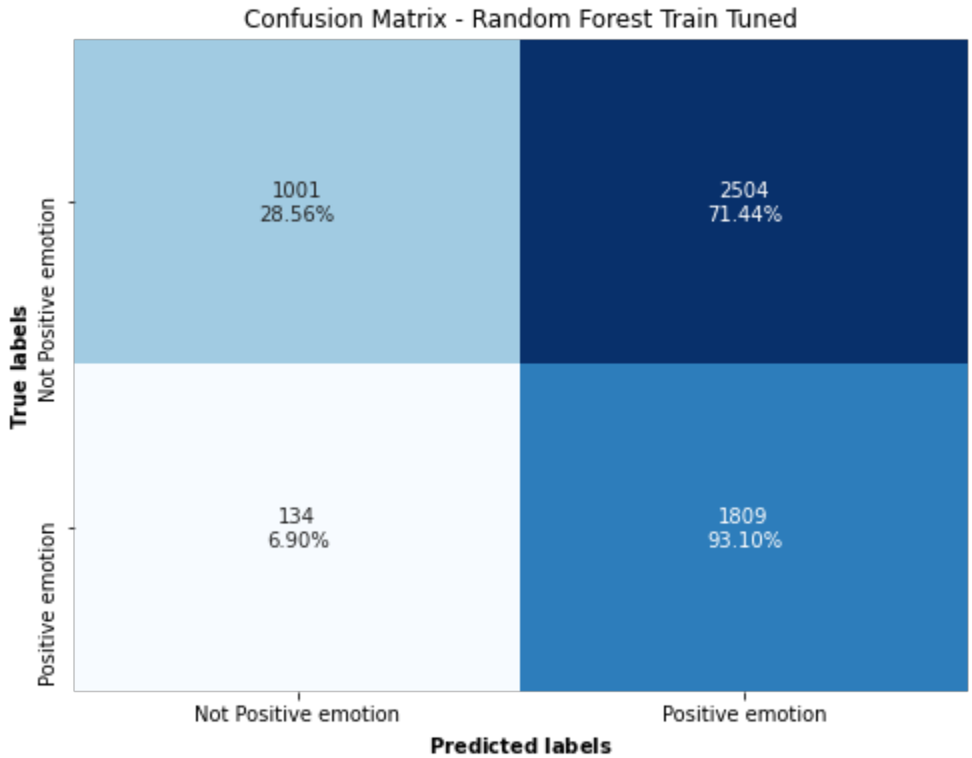
```
Fitting 3 folds for each of 1 candidates, totalling 3 fits
Precision on test set (Random Forest): 0.6704
Best parameters found by GridSearchCV (Random Forest): {'max_depth': 10, 'min
_samples_leaf': 6, 'n_estimators': 150}
```

In [47]:
```
plot_confusion_matrix_and_metrics(y_test, y_pred_test_rf_tuned, title='Cor
```




```
Weighted Precision: 0.67
Weighted Recall: 0.47
Weighted F1 Score: 0.44
```

In [48]:
```
1 plot_confusion_matrix_and_metrics(y_train, y_pred_train_rf_tuned, title='C
```




```
Weighted Precision: 0.72
Weighted Recall: 0.52
Weighted F1 Score: 0.48
```

6.6.3 Logistic Regression

In [49]:
```
# # Define the parameter grid for Logistic Regression
# param_grid = {
#     'penalty': ['l1', 'l2'],  # Regularization term
#     'C': [0.1, 1.0, 10],  # Inverse of regularization strength
#     'solver': ['liblinear', 'saga'],  # Solver for optimization
#     'max_iter': [100, 200]  # Maximum number of iterations
# }

# # Create the Logistic Regression model
# lr_model = LogisticRegression()

# # Define the custom precision scoring function
# precision_scorer = make_scorer(precision_score)

# # Initialize GridSearchCV
# grid_search_lr = GridSearchCV(estimator=lr_model, param_grid=param_grid,
#                                 scoring=precision_scorer, cv=3, verbose=1,

# # Fit the grid search to the training data
# grid_search_lr.fit(X_train, y_train)

# # Get the best model from the grid search
# best_model_lr = grid_search_lr.best_estimator_

# # Make predictions on the train set
# y_pred_proba_train_lr = best_model_lr.predict_proba(X_train)[:, 1]  # Ge

# # Apply custom threshold to convert probabilities into binary prediction
# y_pred_train_lr_tuned = np.where(y_pred_proba_train_lr >= 0.34, 1, 0)

# # Make predictions on the test set
# y_pred_proba_test_lr = best_model_lr.predict_proba(X_test)[:, 1]  # Get

# # Apply custom threshold to convert probabilities into binary prediction
# y_pred_test_lr_tuned = np.where(y_pred_proba_test_lr >= 0.34, 1, 0)

# # Evaluate precision on the test set using the custom threshold
# precision_test_lr = precision_score(y_test, y_pred_test_lr_tuned)
# print(f"Precision on test set (Logistic Regression): {precision_test_lr:

# # Display the best parameters found by the grid search
# print(f"Best parameters found by GridSearchCV (Logistic Regression): {gr
```

After running the Gridsearch optimizing the precision metric, the best parameters that resulted where:

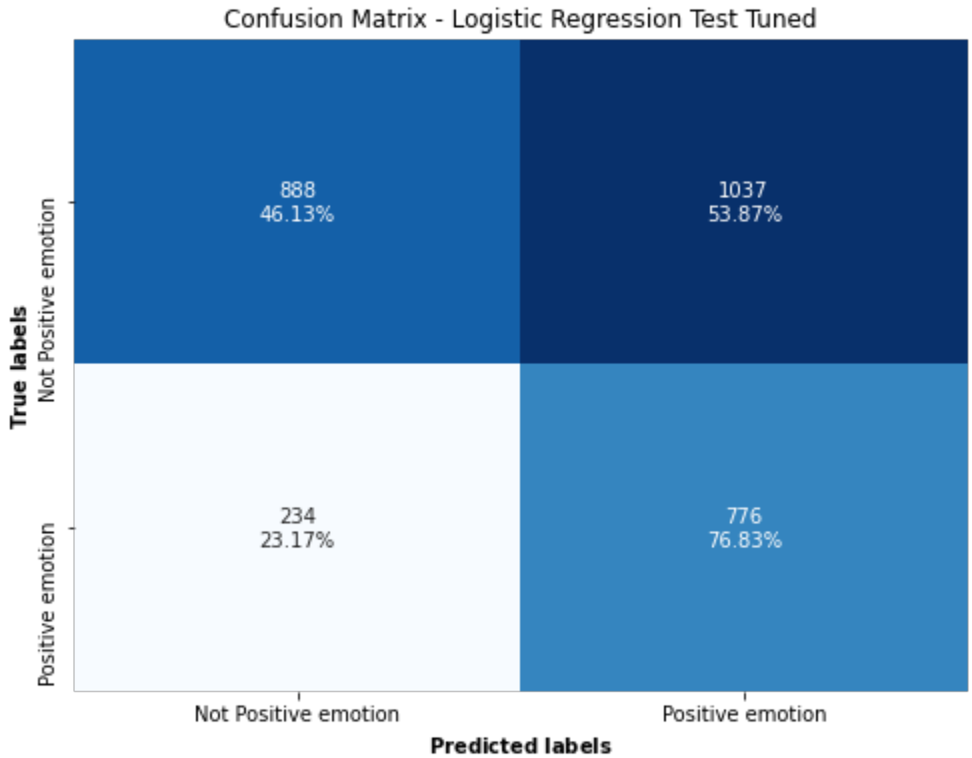
```
Fitting 3 folds for each of 24 candidates, totalling 72 fits
Precision on test set (Logistic Regression): 0.4280
Best parameters found by GridSearchCV (Logistic Regression): {'C': 0.1, 'max_iter': 100, 'penalty': 'l2', 'solver': 'liblinea
r'}
```

In [50]:

```
# Define the parameter grid for Logistic Regression
param_grid = {
    'penalty': ['l2'],  # Regularization term
    'C': [0.1],  # Inverse of regularization strength
    'solver': ['liblinear'],  # Solver for optimization
    'max_iter': [100]  # Maximum number of iterations
}

# Create the Logistic Regression model
lr_model = LogisticRegression()

# Define the custom precision scoring function
precision_scorer = make_scorer(precision_score)

# Initialize GridSearchCV
grid_search_lr = GridSearchCV(estimator=lr_model, param_grid=param_grid,
                              scoring=precision_scorer, cv=3, verbose=1, r

# Fit the grid search to the training data
grid_search_lr.fit(X_train, y_train)

# Get the best model from the grid search
best_model_lr = grid_search_lr.best_estimator_

# Make predictions on the train set
y_pred_proba_train_lr = best_model_lr.predict_proba(X_train)[:, 1]  # Get

# Apply custom threshold to convert probabilities into binary predictions
y_pred_train_lr_tuned = np.where(y_pred_proba_train_lr >= 0.34, 1, 0)

# Make predictions on the test set
y_pred_proba_test_lr = best_model_lr.predict_proba(X_test)[:, 1]  # Get pr

# Apply custom threshold to convert probabilities into binary predictions
y_pred_test_lr_tuned = np.where(y_pred_proba_test_lr >= 0.34, 1, 0)

# Evaluate precision on the test set using the custom threshold
precision_test_lr = precision_score(y_test, y_pred_test_lr_tuned)
print(f"Precision on test set (Logistic Regression): {precision_test_lr:.4

# Display the best parameters found by the grid search
print(f"Best parameters found by GridSearchCV (Logistic Regression): {gric
```

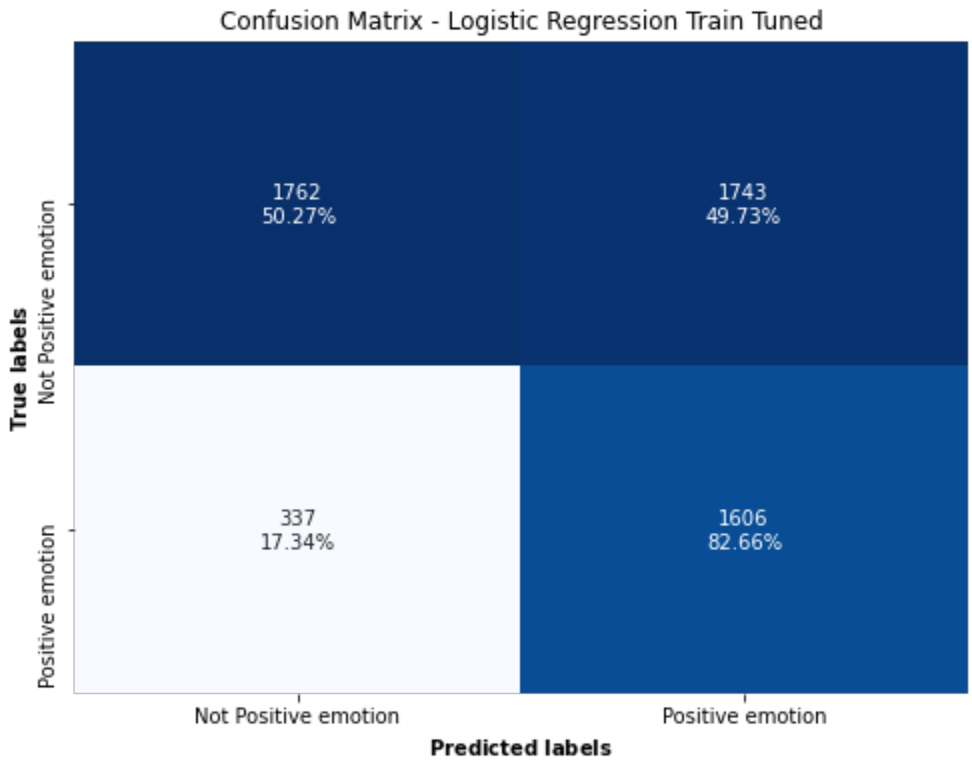
```
Fitting 3 folds for each of 1 candidates, totalling 3 fits
Precision on test set (Logistic Regression): 0.4280
Best parameters found by GridSearchCV (Logistic Regression): {'C': 0.1, 'max_
iter': 100, 'penalty': 'l2', 'solver': 'liblinear'}
```

In [51]:
```
1 plot_confusion_matrix_and_metrics(y_test, y_pred_test_lr_tuned, title='Cor
```




```
Weighted Precision: 0.67
Weighted Recall: 0.57
Weighted F1 Score: 0.57
```

In [52]:
```
plot_confusion_matrix_and_metrics(y_train, y_pred_train_lr_tuned, title='C
```




```
Weighted Precision: 0.71
Weighted Recall: 0.62
Weighted F1 Score: 0.62
```

6.7 Choosing the model

6.7.1 Explanation of the model chosen

After considering the different models that were run, we believe that XGBoost is the most adequate for different reasons: it has one of the highest precision, and the less overfitting maintaining the same metrics.

6.7.2 Saving the model in a pickle

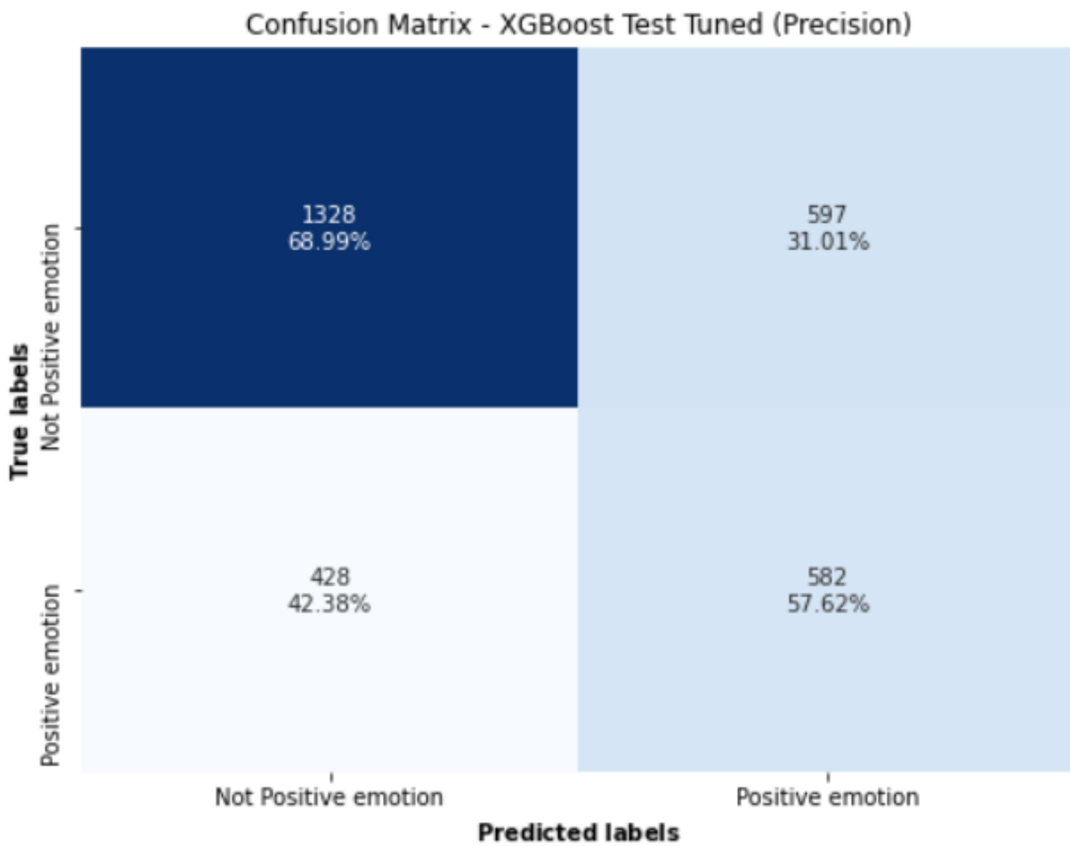
In [53]:
```
with open('..\pickle_objects\model.pkl', 'wb') as file:
    pickle.dump(best_model_xgb_precision, file)
```

7. Conclusion

We have built the best model to detect tweets with positive sentiment, achieving 58% success by optimizing for precision. Our focus is to avoid mistakes when identifying a tweet as having a positive emotion. The reason for this is tied to our business case and the next steps of this project, which involve distinguishing the characteristics of technological products that evoke positive emotions in the public. If we fail to minimize false positives, this would negatively impact our future model aimed at identifying the features of a technological product that trigger positive emotions.

This model enables us to gain reasonable insights when extracting positive features from tweets. By focusing on precision, we ensure that the positive sentiment tweets we identify are highly likely to reflect genuine positivity, which is crucial for the reliability of subsequent analyses.

In the future, the company's intention is that once the positive tweets are extracted, a model will be built to determine the characteristics of technological products that lead to positive sentiment towards the product. This approach will provide deeper insights into what aspects of these products resonate positively with the audience, further enhancing our overall analysis.



As seen in the confusion matrix, a false positive rate of 31.01% in predicting positive sentiment means that a notable portion of non-positive tweets are incorrectly classified as positive. This could impact companies' understanding of consumer feedback. However, with a true positive rate of 57.62%, the model effectively identifies over half of the positive sentiment tweets, providing valuable insights for marketing and product strategies.

To improve these results, we could focus on two key areas:

- First, increasing the size of the training dataset would allow the model to learn from more examples, improving its ability to generalize. It's also important to increase the number of tweets with positive sentiment, as the dataset is currently imbalanced with more non-positive tweets. A more balanced dataset would prevent the model from becoming biased towards the majority class, ensuring a fair representation of both positive and non-positive sentiments.

- Second, incorporating bigrams or trigrams (n-grams) would enable the model to capture word dependencies that are not evident with single tokens, thus enhancing its ability to identify sentiment patterns more accurately.