Instagram Fake vs Genuine Account Detection

This project aims to classify Instagram accounts as **fake** or **genuine** using various features extracted from profiles.
We will apply machine learning techniques to build models that can predict whether an account is fake based on attributes like:

- Profile picture presence
- Username patterns
- Follower and following counts
- Posts, description length, private/public status

🔰 Tools & Technologies:

- **Python**: Data manipulation and modeling
- **Pandas / NumPy**: Data handling
- **Matplotlib / Seaborn**: Visualization
- **Scikit-learn**: Machine learning

🔰 Goal:

- Build an accurate model for fake account detection
- Analyze key features contributing to fake/genuine prediction
- Visualize insights for better understanding

Step 2: Import Libraries

We import all the necessary Python libraries for:

- **Data manipulation:** `pandas`, `numpy`
- **Data visualization:** `matplotlib`, `seaborn`
- **Machine learning:** `scikit-learn` (for model building, evaluation, preprocessing)

This sets up our environment for the project.

In [ ]:
```
# Data manipulation
import pandas as pd
import numpy as np

# Visualization
```

```
import matplotlib.pyplot as plt
import seaborn as sns

# Machine learning
from sklearn.model_selection import train_test_split
from sklearn.tree import DecisionTreeClassifier
from sklearn.ensemble import RandomForestClassifier
from sklearn.preprocessing import StandardScaler
from sklearn.metrics import accuracy_score, confusion_matrix, classification_r

# Visualization style
sns.set(style='whitegrid')
```

Step 3: Upload & Load the Data

In this step, we will:

- Upload the `train.csv` and `test.csv` files directly into Colab.
- Load the uploaded CSV files into dataframes using **Pandas**.
- Preview the first few rows to understand the structure of the datasets.

In [ ]:
```
from google.colab import files

# Upload files
uploaded = files.upload()

# Load datasets into dataframes
import io
train = pd.read_csv(io.BytesIO(uploaded['train.csv']))
test = pd.read_csv(io.BytesIO(uploaded['test.csv']))

# Preview train dataset
train.head()
```

Browse...

Upload widget is only available when the cell has been executed in the current browser session. Please rerun this cell to enable.

```
Saving test.csv to test.csv
Saving train.csv to train.csv
```

Out[ ]:

	profile pic	nums/ length username	fullname words	nums/ length fullname	name==username	description length	exter U
0	1	0.27	0	0.0	0	53	
1	1	0.00	2	0.0	0	44	
2	1	0.10	2	0.0	0	0	
3	1	0.00	1	0.0	0	82	
4	1	0.00	2	0.0	0	0	

Step 4: Dataset Overview

In this step, we will:

- Display the structure and data types of the dataset.
- Generate summary statistics (mean, min, max, etc.).
- Check for any missing values.
- Examine the distribution of the target label (`fake`) to understand class balance.

In [ ]:
```
# Check dataset info
train.info()

# Summary statistics
train.describe()

# Missing values
print("\nMissing values in each column:\n", train.isnull().sum())

# Target label distribution
print("\nFake account distribution:\n", train['fake'].value_counts())
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 576 entries, 0 to 575
Data columns (total 12 columns):
 #   Column                Non-Null Count  Dtype
---  ------                --------------  -----
 0   profile pic           576 non-null    int64
 1   nums/length username  576 non-null    float64
 2   fullname words        576 non-null    int64
 3   nums/length fullname  576 non-null    float64
 4   name==username        576 non-null    int64
 5   description length    576 non-null    int64
 6   external URL          576 non-null    int64
 7   private               576 non-null    int64
 8   #posts                576 non-null    int64
 9   #followers            576 non-null    int64
 10  #follows              576 non-null    int64
 11  fake                  576 non-null    int64
dtypes: float64(2), int64(10)
memory usage: 54.1 KB

Missing values in each column:
 profile pic             0
nums/length username    0
fullname words          0
nums/length fullname    0
name==username          0
description length      0
external URL            0
private                 0
#posts                  0
#followers              0
#follows                0
fake                    0
dtype: int64

Fake account distribution:
 fake
0    288
1    288
Name: count, dtype: int64
```

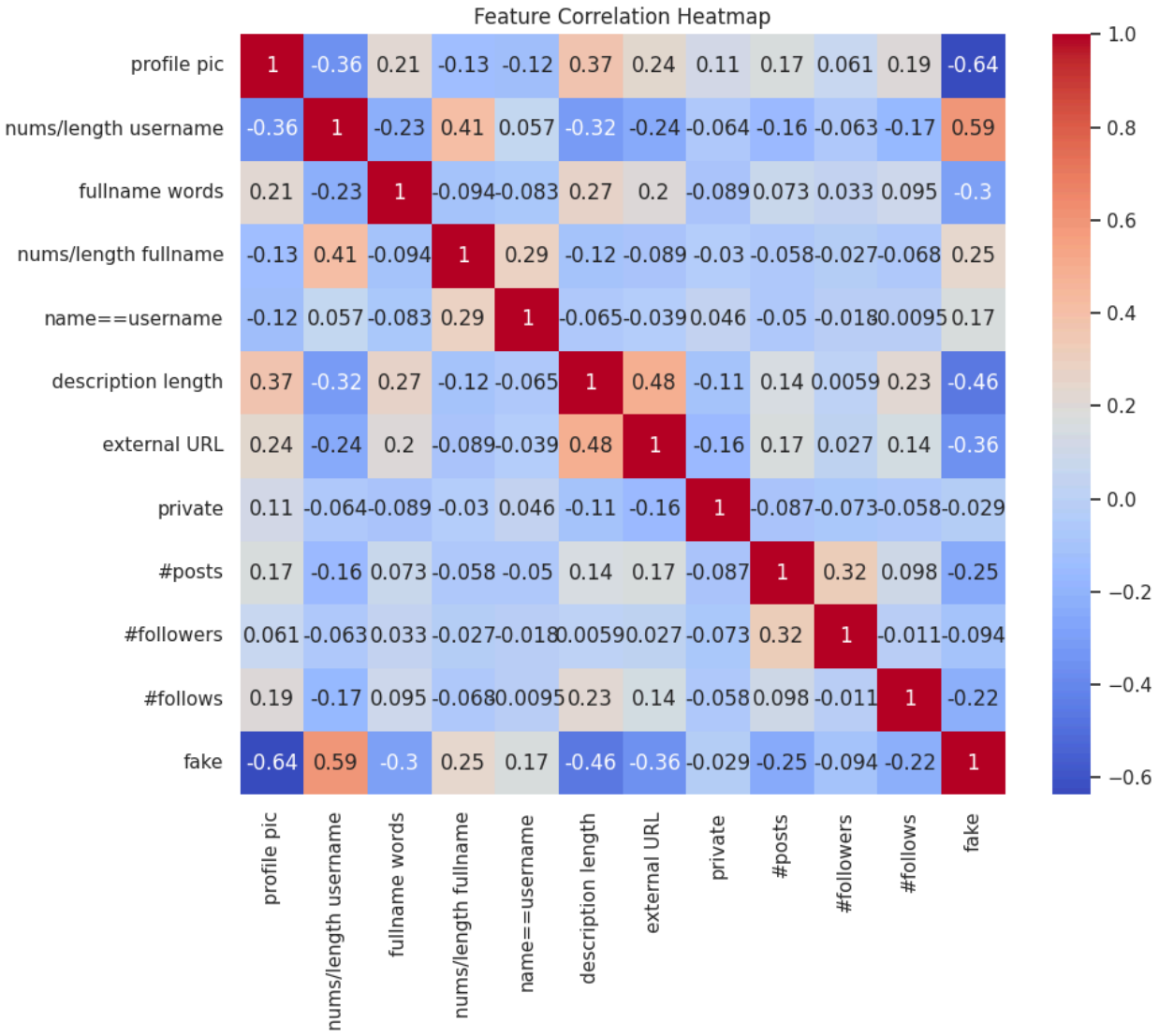
Step 5: Exploratory Data Analysis (EDA)

In this step, we will:

- Visualize the distribution of fake vs genuine accounts.
- Plot a correlation heatmap to see how features relate to each other and the target.
- Explore relationships between key features and the `fake` label (e.g. profile picture presence, followers, follows, posts, private status).

These visualizations help us identify important patterns in the data.

In [ ]:
```
# Distribution of fake vs genuine accounts
sns.countplot(x='fake', data=train)
plt.title("Distribution of Fake vs Genuine Accounts")
plt.show()

# Correlation heatmap
plt.figure(figsize=(10, 8))
sns.heatmap(train.corr(), annot=True, cmap='coolwarm')
plt.title("Feature Correlation Heatmap")
plt.show()
```



EDA: Feature-wise Visualizations

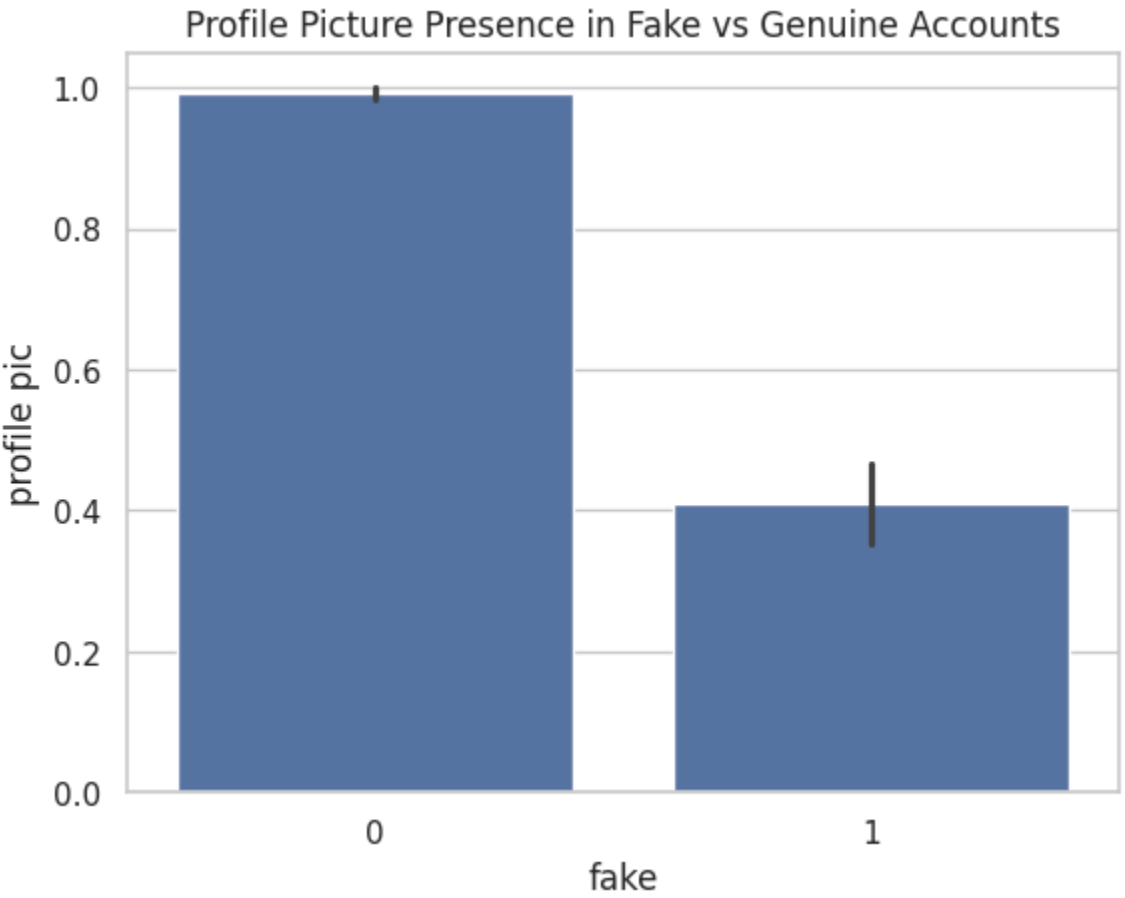
We will now explore how key features differ between fake and genuine accounts:

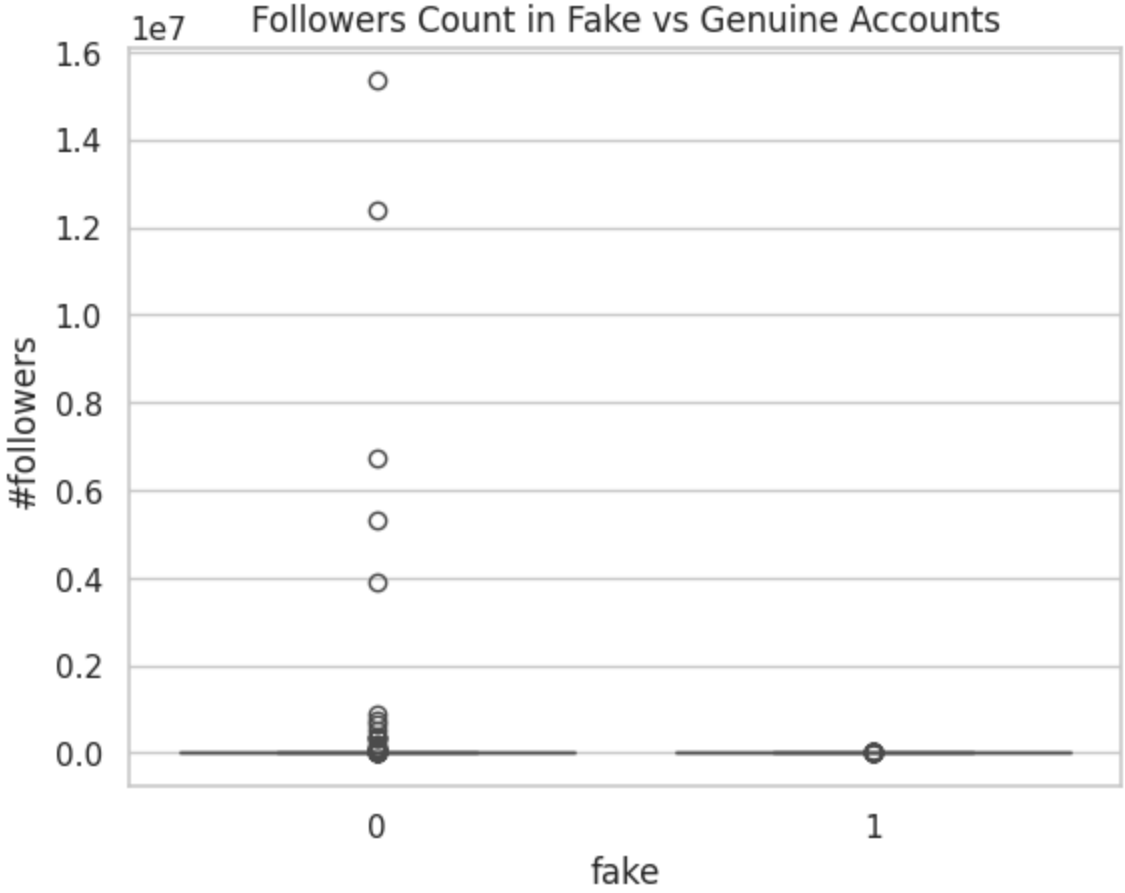
- Presence of profile picture
- Followers count
- Following count
- Posts count
- Private/public status

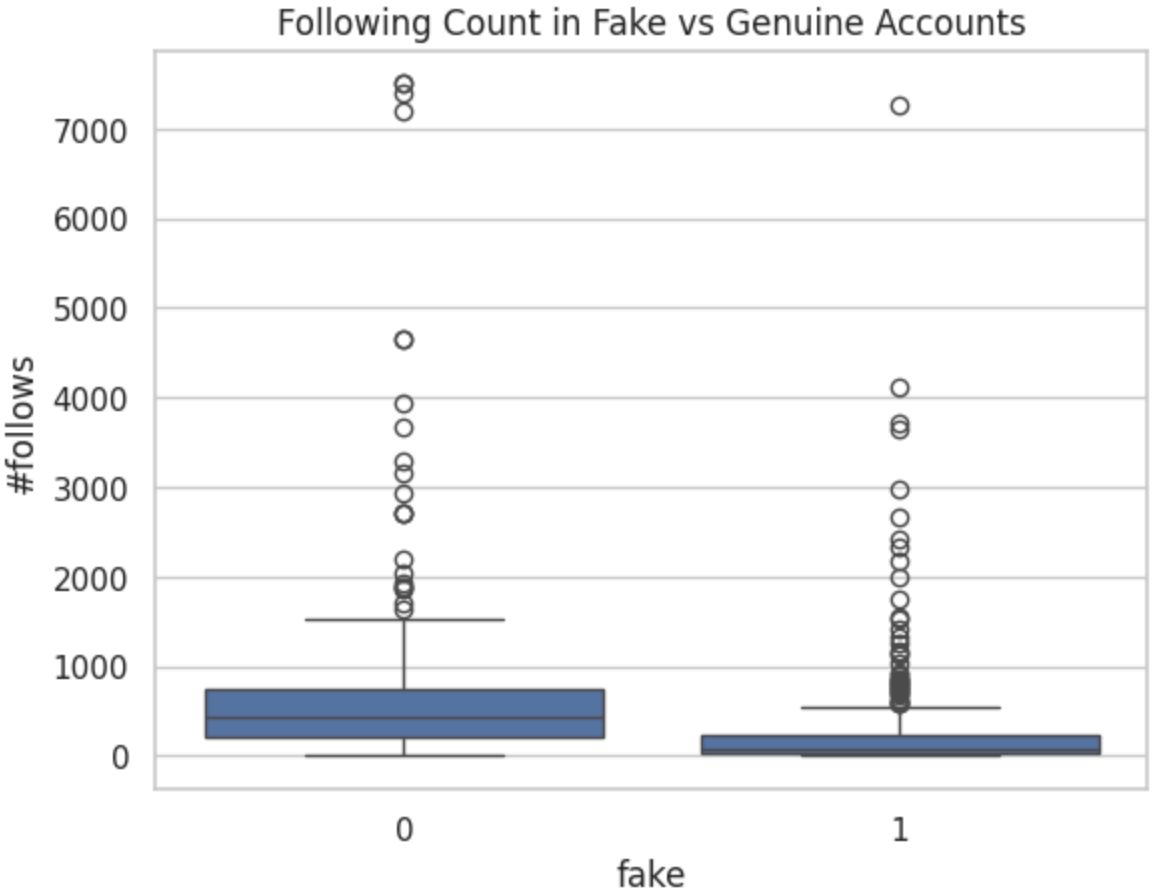
These charts will help identify features that could be useful for classification.

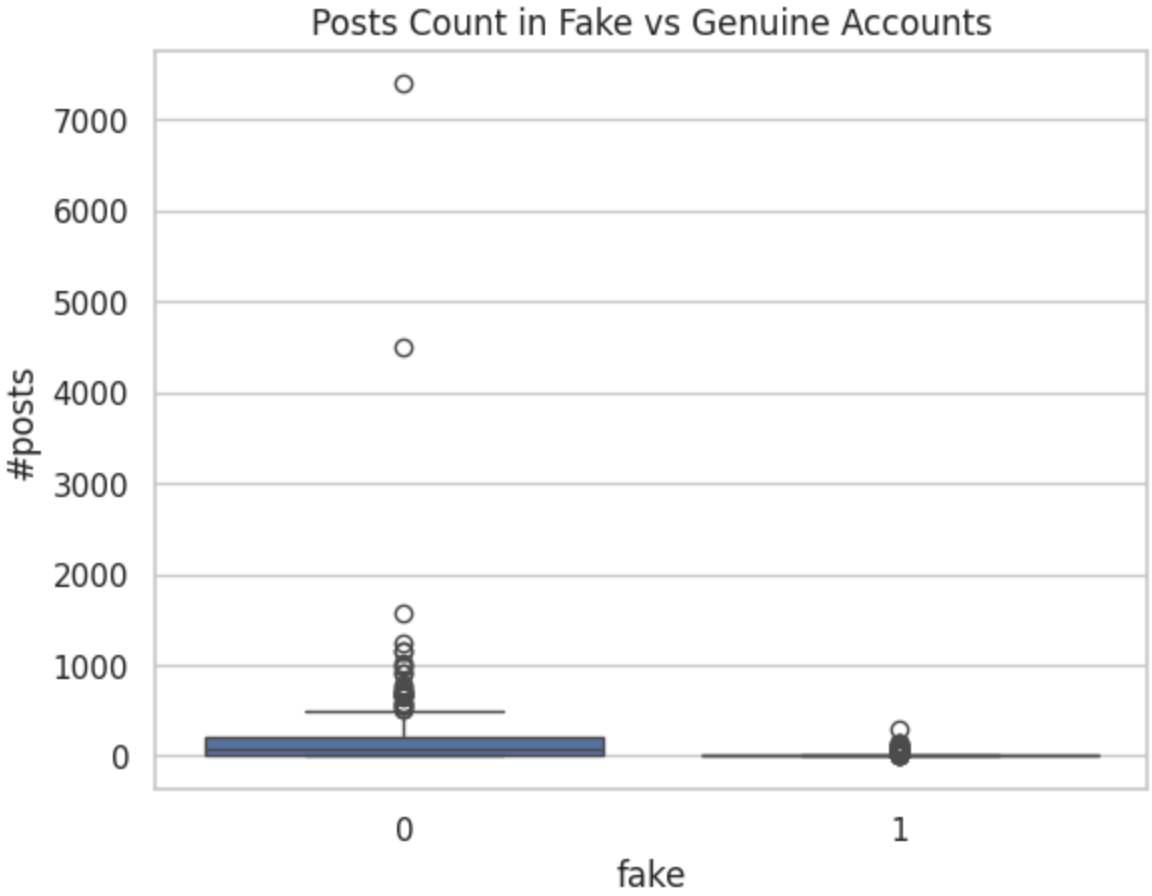
In [ ]:
```
# Profile pic presence vs fake/genuine
sns.barplot(x='fake', y='profile pic', data=train)
plt.title("Profile Picture Presence in Fake vs Genuine Accounts")
plt.show()
```

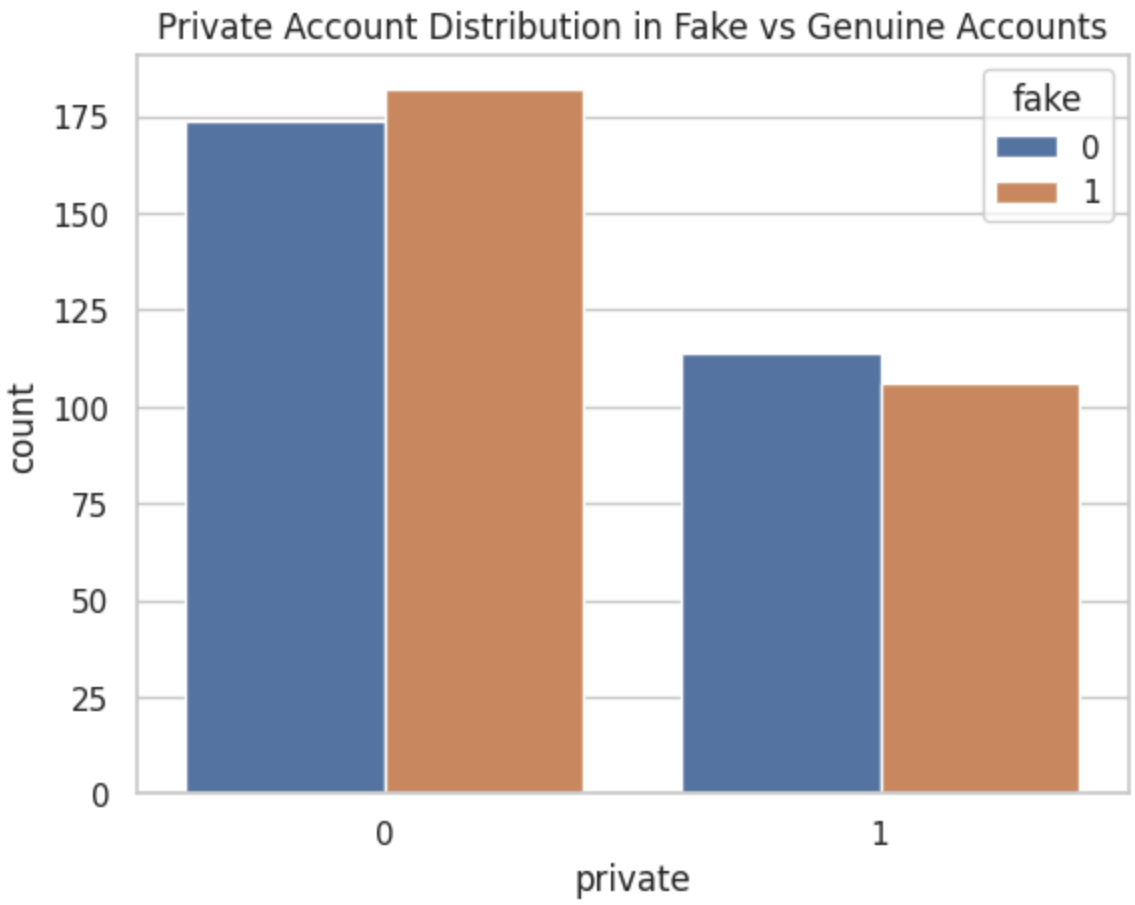
```
# Followers count
sns.boxplot(x='fake', y='#followers', data=train)
plt.title("Followers Count in Fake vs Genuine Accounts")
plt.show()

# Following count
sns.boxplot(x='fake', y='#follows', data=train)
plt.title("Following Count in Fake vs Genuine Accounts")
plt.show()

# Posts count
sns.boxplot(x='fake', y='#posts', data=train)
plt.title("Posts Count in Fake vs Genuine Accounts")
plt.show()

# Private account distribution
sns.countplot(x='private', hue='fake', data=train)
plt.title("Private Account Distribution in Fake vs Genuine Accounts")
plt.show()
```




Followers Count in Fake vs Genuine Accounts
1e7
1.6
1.4
1.2
1.0
0.8
0.6
0.4
0.2
0.0
#followers
0
1
fake


Following Count in Fake vs Genuine Accounts
#follows
7000
6000
5000
4000
3000
2000
1000
0
0
1
fake


Posts Count in Fake vs Genuine Accounts
#posts
7000
6000
5000
4000
3000
2000
1000
0
0
1
fake



Step 6: Feature Engineering / Preprocessing

In this step, we will:

- Create additional features that might improve model performance (e.g., follower-to-following ratio).
- Decide whether scaling is needed for certain models.
- Prepare the dataset by selecting features and separating the target variable.

These actions help in improving model accuracy and ensuring better training.

In [ ]:
```
# Create follower-following ratio (handle division by zero)
train['follower_following_ratio'] = train['#followers'] / train['#follows'].re
test['follower_following_ratio'] = test['#followers'] / test['#follows'].repla

# Check new feature
train[['#followers', '#follows', 'follower_following_ratio']].head()
```

Out[ ]:

	#followers	#follows	follower_following_ratio
0	1000	955	1.047120
1	2740	533	5.140713
2	159	98	1.622449
3	414	651	0.635945
4	151	126	1.198413

Step 7: Train/Test Preparation

In this step, we will:

- Separate features (X) and target label (y) from the train dataset.
- Perform a train-validation split so we can train the model and evaluate it on unseen validation data.
- Ensure our train and validation sets are representative for model training.

In [ ]:
```
# Define X (features) and y (target)
X = train.drop('fake', axis=1)
y = train['fake']

# Split into training and validation sets (80/20 split)
X_train, X_val, y_train, y_val = train_test_split(X, y, test_size=0.2, random_

# Show shapes
print(f"Train shape: {X_train.shape}")
print(f"Validation shape: {X_val.shape}")
```

```
Train shape: (460, 12)
Validation shape: (116, 12)
```

Step 8: Baseline Model Building

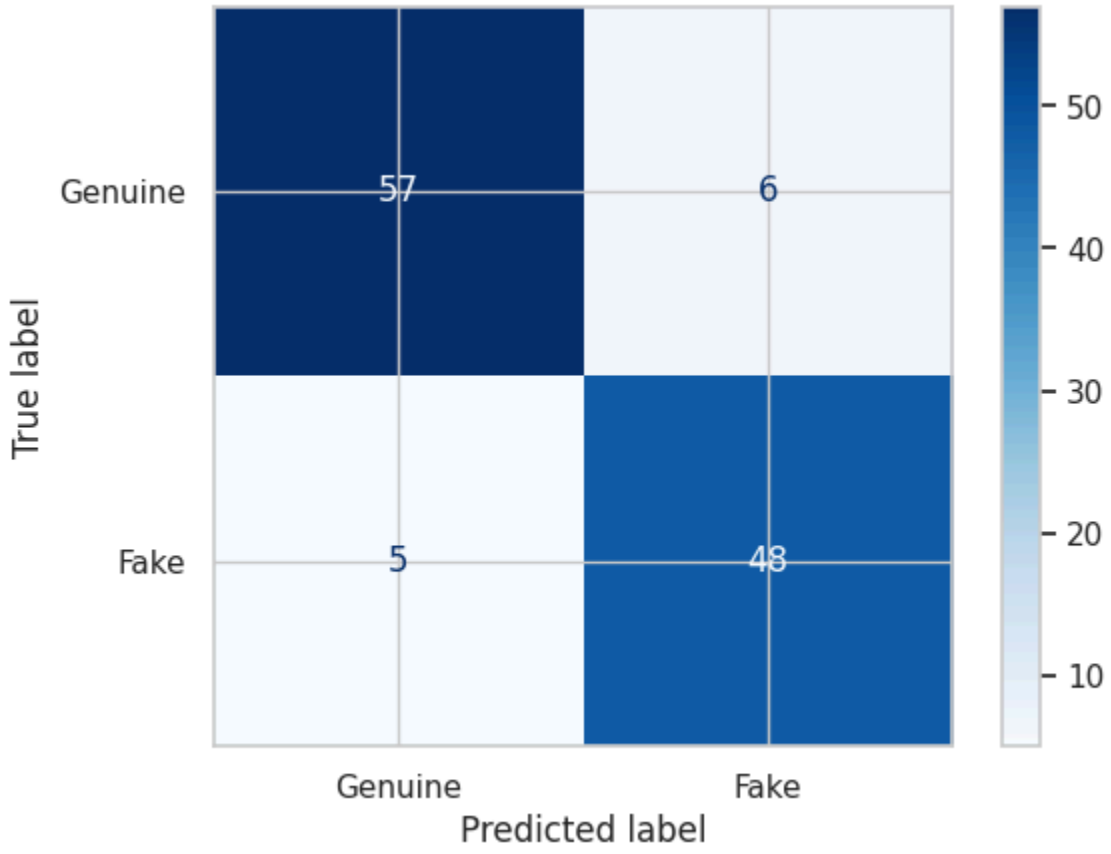
In this step, we will:

- Build and train basic machine learning models: **Decision Tree** and **Random Forest**.
- These models will serve as our baseline to evaluate performance.
- We will later try tuning and more advanced methods if needed.

In [ ]:
```
# Initialize and train Decision Tree
dt_model = DecisionTreeClassifier(random_state=42)
dt_model.fit(X_train, y_train)
```

```python
# Predict on validation set
y_val_pred = dt_model.predict(X_val)

# Evaluate performance
print("Decision Tree Performance on Validation Set:")
print(classification_report(y_val, y_val_pred))
ConfusionMatrixDisplay.from_predictions(y_val, y_val_pred, display_labels=['Ge
plt.show()
```

```
Decision Tree Performance on Validation Set:
              precision    recall  f1-score   support

           0       0.92      0.90      0.91        63
           1       0.89      0.91      0.90        53

    accuracy                           0.91       116
   macro avg       0.90      0.91      0.90       116
weighted avg       0.91      0.91      0.91       116
```

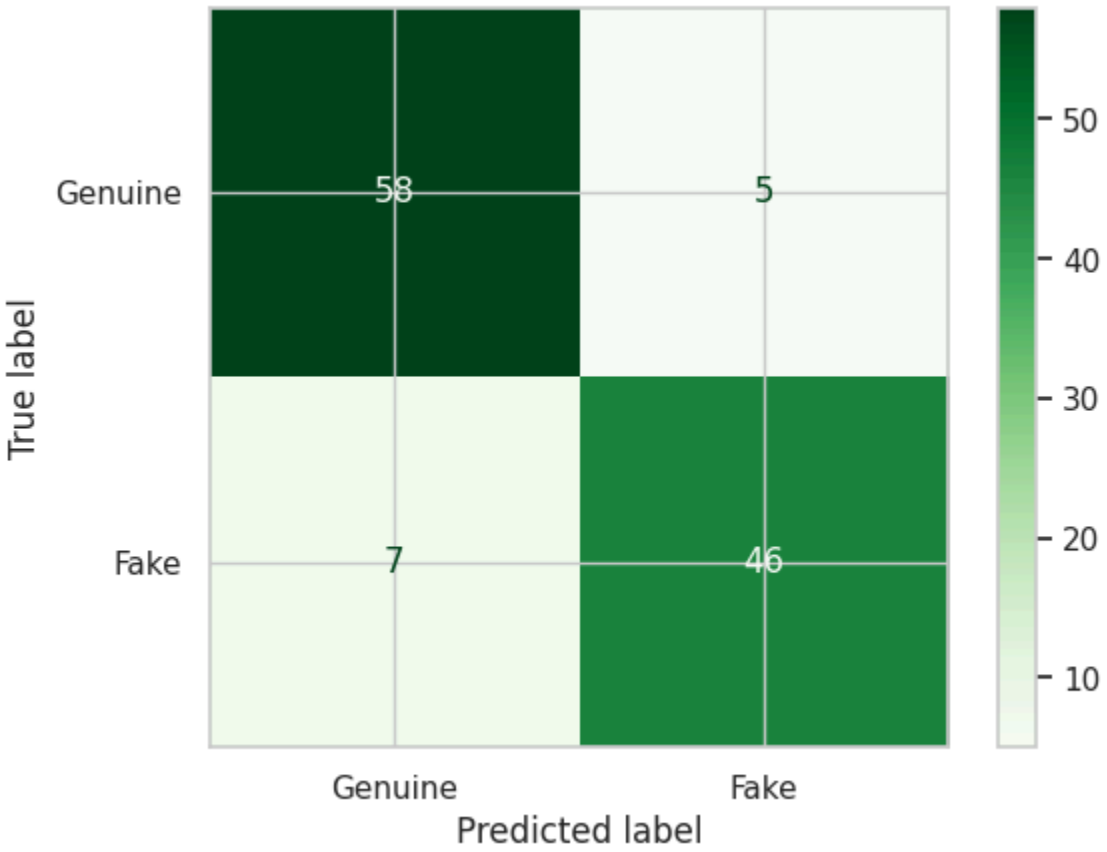


Baseline Random Forest Model

Next, we build a **Random Forest Classifier**.
Random Forest generally provides better performance due to ensemble learning.
We will train it on our training data and evaluate on the validation set.

In [ ]:
```
# Initialize and train Random Forest
rf_model = RandomForestClassifier(random_state=42)
rf_model.fit(X_train, y_train)

# Predict on validation set
y_val_pred_rf = rf_model.predict(X_val)

# Evaluate performance
print("Random Forest Performance on Validation Set:")
print(classification_report(y_val, y_val_pred_rf))
ConfusionMatrixDisplay.from_predictions(y_val, y_val_pred_rf, display_labels=[
plt.show()
```

```
Random Forest Performance on Validation Set:
              precision    recall  f1-score   support

           0       0.89      0.92      0.91        63
           1       0.90      0.87      0.88        53

    accuracy                           0.90       116
   macro avg       0.90      0.89      0.90       116
weighted avg       0.90      0.90      0.90       116
```



Step 9: Hyperparameter Tuning

In this step, we will:

- Apply **GridSearchCV** to search for the best hyperparameters for our Random Forest model.
- Tune parameters such as:
    - `n_estimators` (number of trees)
    - `max_depth`
    - `min_samples_split`
    - `min_samples_leaf`
- This process helps in improving model accuracy and generalization.

In [ ]:
```
from sklearn.model_selection import GridSearchCV

# Define parameter grid
param_grid = {
    'n_estimators': [50, 100, 200],
    'max_depth': [None, 10, 20, 30],
    'min_samples_split': [2, 5, 10],
    'min_samples_leaf': [1, 2, 4]
}

# Initialize GridSearch
grid_search = GridSearchCV(estimator=RandomForestClassifier(random_state=42),
                           param_grid=param_grid,
                           cv=3,
                           n_jobs=-1,
                           verbose=2)

# Fit grid search
grid_search.fit(X_train, y_train)

# Best parameters
print("Best Hyperparameters:\n", grid_search.best_params_)
```

```
Fitting 3 folds for each of 108 candidates, totalling 324 fits
Best Hyperparameters:
 {'max_depth': None, 'min_samples_leaf': 1, 'min_samples_split': 2, 'n_estimato
rs': 100}
```

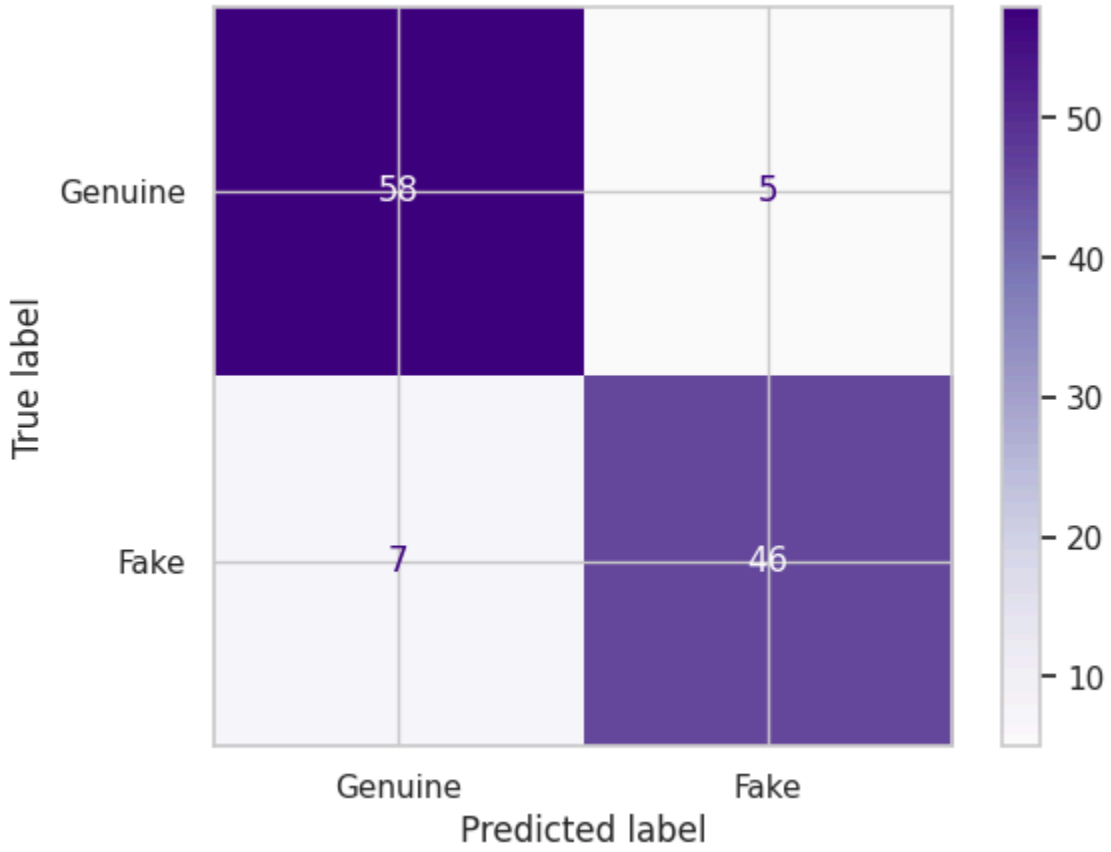
Step 10: Evaluate Tuned Model

Now that we have the best hyperparameters from GridSearchCV,
we will evaluate the tuned Random Forest model on our validation set.
This helps us confirm if tuning improved the model's accuracy, precision, and recall.

In [ ]:
```
# Get best model from grid search
best_rf_model = grid_search.best_estimator_

# Predict on validation set
```

```
y_val_pred_tuned = best_rf_model.predict(X_val)

# Evaluate performance
print("Tuned Random Forest Performance on Validation Set:")
print(classification_report(y_val, y_val_pred_tuned))
ConfusionMatrixDisplay.from_predictions(y_val, y_val_pred_tuned, display_label
plt.show()
```

```
Tuned Random Forest Performance on Validation Set:
              precision    recall  f1-score   support

           0       0.89      0.92      0.91        63
           1       0.90      0.87      0.88        53

    accuracy                           0.90       116
   macro avg       0.90      0.89      0.90       116
weighted avg       0.90      0.90      0.90       116
```



Step 11: Final Test Predictions

In this step, we will:

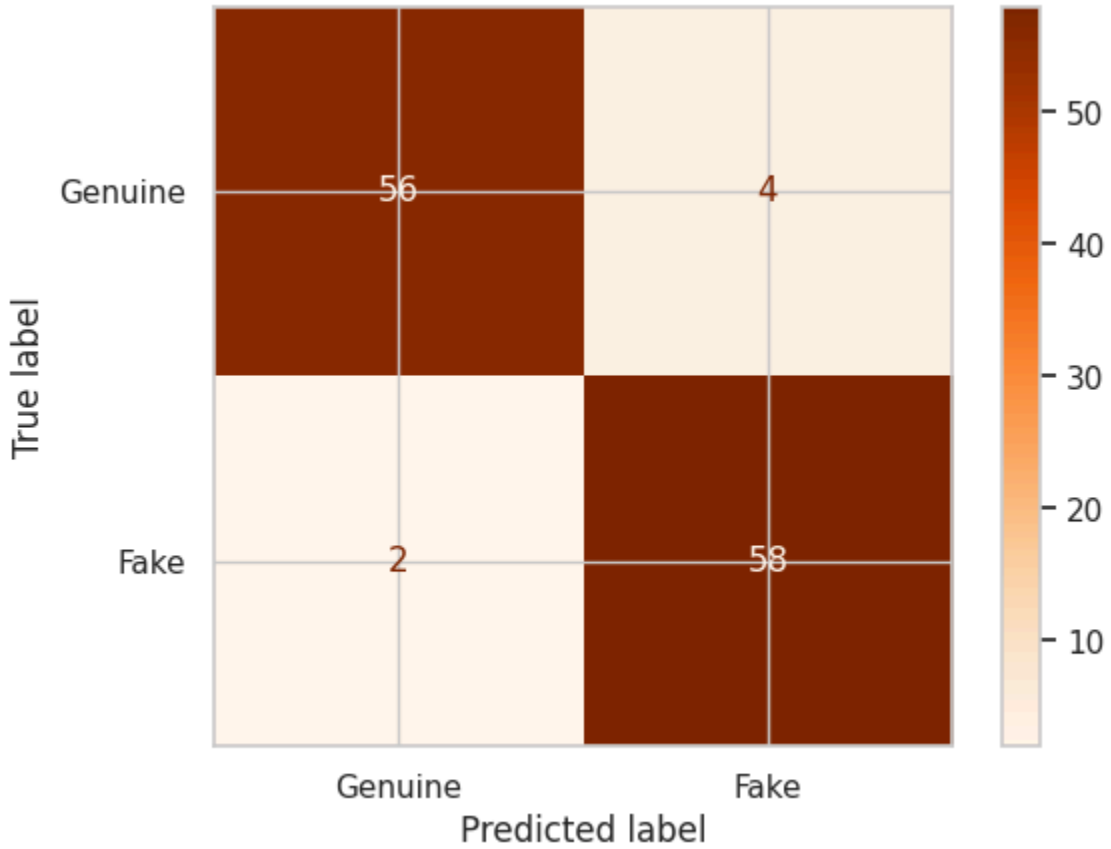
- Use the tuned Random Forest model to predict labels on the **test dataset**.
- If test labels are available, evaluate performance using accuracy,

precision, recall, F1-score, and confusion matrix.

- This helps us understand how the model generalizes to unseen data.

In [ ]:
```
# Split test data into features and target (if labels are present)
X_test = test.drop('fake', axis=1)
y_test = test['fake']

# Predict on test data
y_test_pred = best_rf_model.predict(X_test)

# Evaluate performance
print("Tuned Random Forest Performance on Test Set:")
print(classification_report(y_test, y_test_pred))
ConfusionMatrixDisplay.from_predictions(y_test, y_test_pred, display_labels=['
plt.show()
```

```
Tuned Random Forest Performance on Test Set:
              precision    recall  f1-score   support

           0       0.97      0.93      0.95        60
           1       0.94      0.97      0.95        60

    accuracy                           0.95       120
   macro avg       0.95      0.95      0.95       120
weighted avg       0.95      0.95      0.95       120
```



Step 12: Insights & Feature Importance

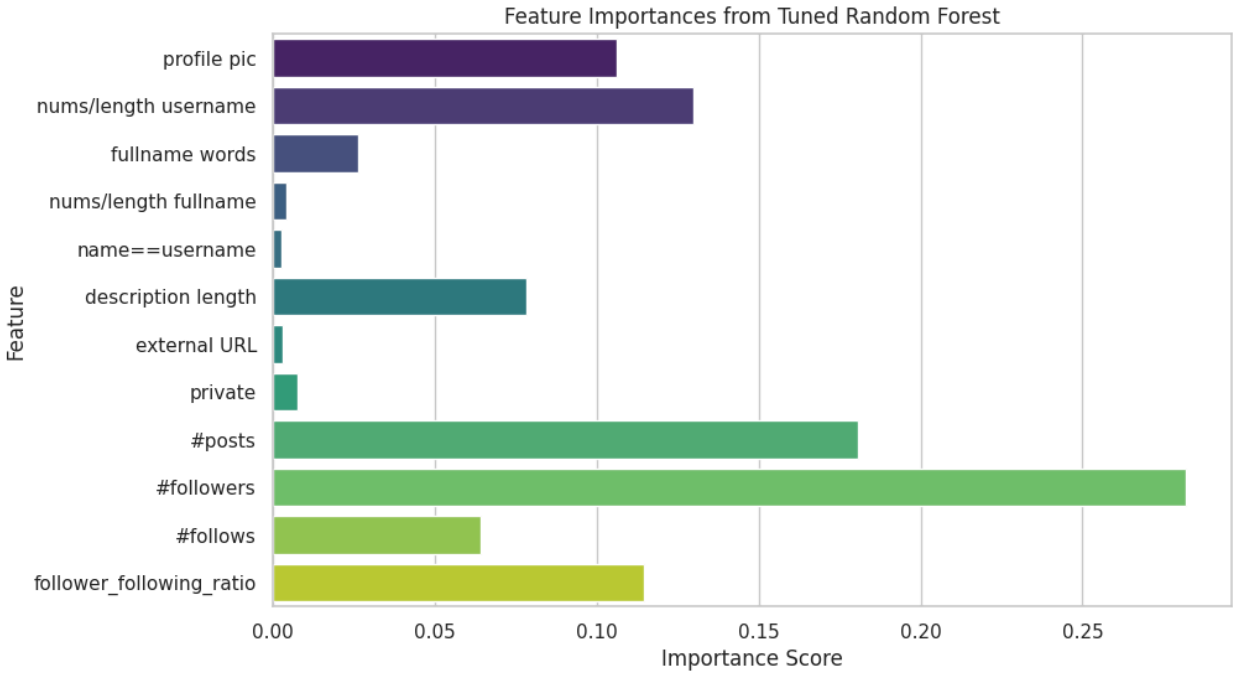
In this step, we will:

- Visualize the importance of each feature as determined by the Random Forest model.
- Summarize key insights about what drives the classification of fake vs genuine accounts.
- Identify patterns (e.g., profile picture, followers, following ratio) that were most influential.

This helps us understand how the model makes decisions.

In [ ]:
```
# Get feature importances from the best model
importances = best_rf_model.feature_importances_
features = X_train.columns

# Plot feature importances
plt.figure(figsize=(10, 6))
sns.barplot(x=importances, y=features, palette='viridis')
plt.title("Feature Importances from Tuned Random Forest")
plt.xlabel("Importance Score")
plt.ylabel("Feature")
plt.show()
```

```
/tmp/ipython-input-14-77774163.py:7: FutureWarning:

Passing `palette` without assigning `hue` is deprecated and will be removed in
v0.14.0. Assign the `y` variable to `hue` and set `legend=False` for the same e
ffect.

  sns.barplot(x=importances, y=features, palette='viridis')
```


Feature Importances from Tuned Random Forest
profile pic
nums/length username
fullname words
nums/length fullname
name==username
description length
external URL
private
#posts
#followers
#follows
follower_following_ratio
Feature
0.00
0.05
0.10
0.15
0.20
0.25
Importance Score