Natural Language Processing using Python Programming

Notebook 08.3: Evaluating Model Performance (Metrics and Matrices)

Python 3.8+ NLTK Latest SpaCy Latest Scikit-learn Latest License MIT

Part of the comprehensive learning series: Natural Language Processing using Python Programming

Learning Objectives:

- Master comprehensive model evaluation beyond simple accuracy metrics
- Understand and implement confusion matrix analysis for multi-class classification
- Learn precision, recall, and F1-score calculations and their real-world applications
- Apply cross-validation techniques for robust model assessment
- Implement hyperparameter tuning using GridSearchCV for optimal model performance

---

- This notebook explores comprehensive model evaluation techniques that go beyond simple accuracy metrics, which can be misleading especially in classification problems with class imbalance.

- We'll dive deep into essential evaluation metrics including the **Confusion Matrix**, **Precision**, **Recall**, and **F1-Score** to thoroughly understand classifier performance and make informed decisions about model selection.

1. Setting up: Re-running the Champion Model

We reload the data and train the Linear SVM model (our typical champion from 8.2) for evaluation.

In [1]:
```python
# Import necessary libraries
from sklearn.datasets import fetch_20newsgroups
from sklearn.model_selection import train_test_split
from sklearn.pipeline import Pipeline
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.svm import LinearSVC
from sklearn.metrics import classification_report, confusion_matrix
import pandas as pd
```

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

# 1. Load Data
categories = ['alt.atheism', 'soc.religion.christian', 'comp.graphics', 'rec.
newsgroups_train = fetch_20newsgroups(subset='all', categories=categories, sh

X_train, X_test, y_train, y_test = train_test_split(
    newsgroups_train.data, newsgroups_train.target, test_size=0.3, random_sta
)

# 2. Define and Train the SVM Pipeline
svm_pipeline = Pipeline([
    ('tfidf', TfidfVectorizer()),
    ('clf', LinearSVC(random_state=42, dual=True)),
])

# Train the model
svm_pipeline.fit(X_train, y_train)

# 3. Make predictions
predictions = svm_pipeline.predict(X_test)
target_names = newsgroups_train.target_names

print("SVM Model trained and ready for comprehensive evaluation.")
```

```
SVM Model trained and ready for comprehensive evaluation.
```

---

2. The Confusion Matrix

- The **Confusion Matrix** provides a complete breakdown of correct and incorrect predictions for *each class*.
- It is the foundation for all other classification metrics.

Key Terms (Binary Case):

- **True Positive (TP):** Predicted Positive, Actual Positive (Correct)
- **True Negative (TN):** Predicted Negative, Actual Negative (Correct)
- **False Positive (FP):** Predicted Positive, Actual Negative (Type I Error)
- **False Negative (FN):** Predicted Negative, Actual Positive (Type II Error)

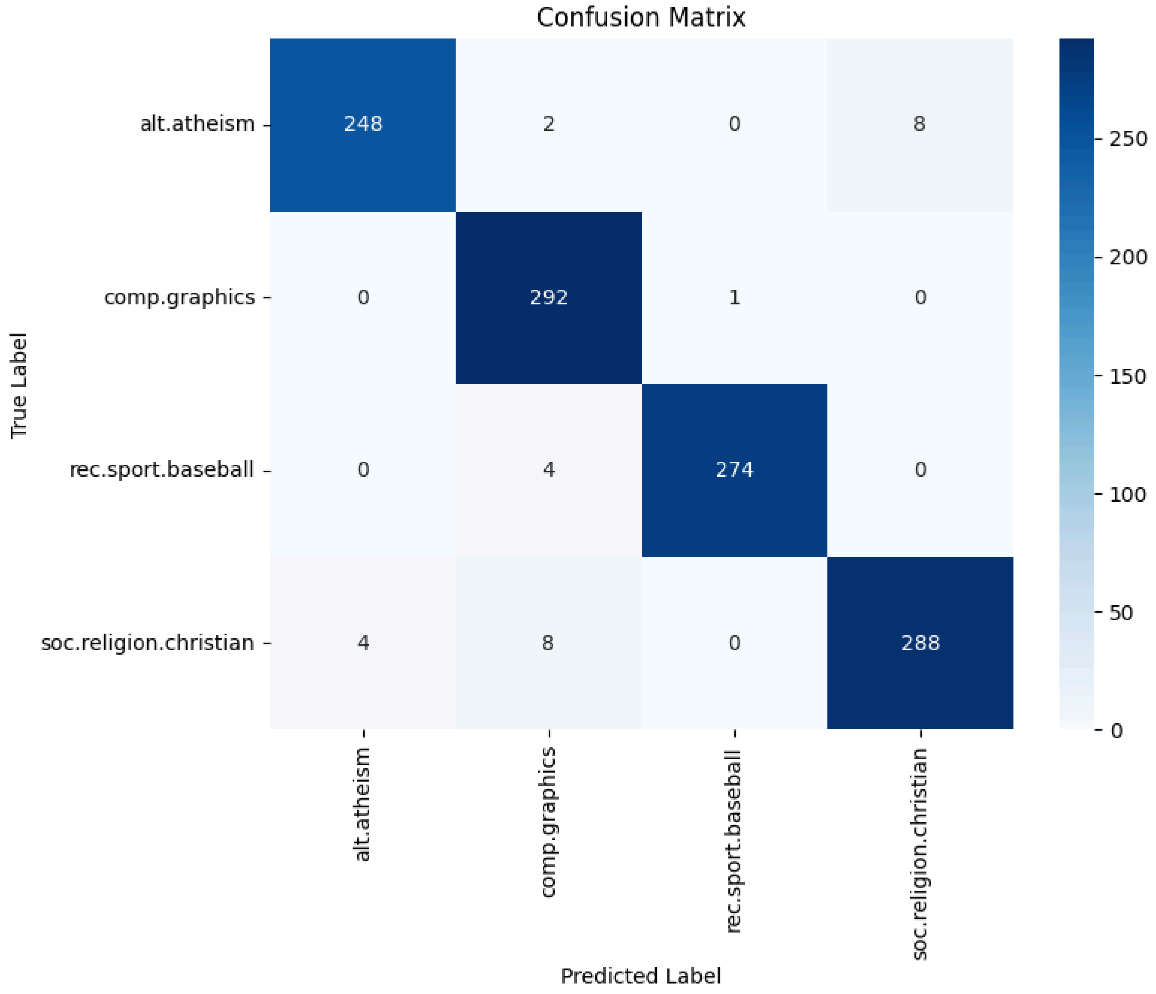
In [2]:

```python
# 1. Calculate the Confusion Matrix
cm = confusion_matrix(y_test, predictions)

# 2. Visualize the Confusion Matrix
plt.figure(figsize=(8, 6))
sns.heatmap(cm, annot=True, fmt='d', cmap='Blues',
            xticklabels=target_names, yticklabels=target_names)
plt.title('Confusion Matrix')
```

```
plt.xlabel('Predicted Label')
plt.ylabel('True Label')
plt.show()

# Observation: Diagonal values (TPs) should be high; off-diagonal values (FPs
```



---

3. Core Evaluation Metrics

- The **Classification Report** bundles the most important metrics, calculated directly from the Confusion Matrix.

In [3]:
```
print("--- Classification Report ---")
print(classification_report(y_test, predictions, target_names=target_names))
```

```
--- Classification Report ---
                        precision    recall  f1-score   support

           alt.atheism       0.98      0.96      0.97       258
         comp.graphics       0.95      1.00      0.97       293
    rec.sport.baseball       1.00      0.99      0.99       278
soc.religion.christian       0.97      0.96      0.97       300

              accuracy                           0.98      1129
             macro avg       0.98      0.98      0.98      1129
          weighted avg       0.98      0.98      0.98      1129
```

3.1 Precision (Minimizing False Positives)

- **Precision** answers: *Of all the documents the model predicted as X, how many were actually X?*

$$\text{Precision} = \frac{\text{True Positives}}{\text{True Positives} + \text{False Positives}}$$

> **Real-World Use:** Crucial for applications where a **False Positive** is very costly (e.g., **Spam Filter** - You don't want a legitimate email (TP) marked as spam (FP)). You want the model to be *sure* about its positive claims.

3.2 Recall (Minimizing False Negatives)

- **Recall** answers: *Of all the documents that were actually X, how many did the model correctly identify?*

$$\text{Recall} = \frac{\text{True Positives}}{\text{True Positives} + \text{False Negatives}}$$

> **Real-World Use:** Crucial for applications where a **False Negative** is very costly (e.g., **Medical Diagnosis** - You don't want to miss a disease (FN)). You want the model to capture *all* relevant examples.

3.3 F1-Score (The Harmonic Mean)

- The **F1-Score** is the harmonic mean of Precision and Recall.
- It provides a single score that summarizes the model's predictive power when balancing both metrics, making it a reliable overall benchmark.

$$\text{F1-Score} = 2 \times \frac{\text{Precision} \times \text{Recall}}{\text{Precision} + \text{Recall}}$$

Scenario	Best Metric	Reason
Spam Detection	**Precision**	Minimizing FPs (labeling non-spam as spam) is paramount.
Information Retrieval	**Recall**	Minimizing FNs (missing relevant documents) is paramount.
General Benchmarking	**F1-Score**	Provides a balanced view of performance.

---

4. Cross-Validation and Hyperparameter Tuning

- For reliable results, evaluation must be systematic.
- **Cross-Validation (CV)** and **Hyperparameter Tuning** are necessary to prevent overfitting and maximize performance.

4.1 Cross-Validation (CV)

- CV ensures the model performance is not dependent on a specific train/test split.
- It repeatedly splits the training data into k folds, trains the model k times, and averages the results.

In [4]:
```
# Import necessary libraries for cross-validation
from sklearn.model_selection import cross_val_score
import numpy as np

# Perform 5-fold cross-validation on the SVM pipeline
# Note: This uses the WHOLE dataset (X, y) for the cross-validation process
cv_scores = cross_val_score(svm_pipeline, newsgroups_train.data, newsgroups_t

print("5-Fold Cross-Validation Scores (F1-Macro):")
print(cv_scores.round(4))
print(f"Average F1-Score: {np.mean(cv_scores):.4f}")
```

```
5-Fold Cross-Validation Scores (F1-Macro):
[0.9811 0.9828 0.9775 0.9807 0.9907]
Average F1-Score: 0.9826
```

4.2 Hyperparameter Tuning with `GridSearchCV`

- Hyperparameters are model settings (like the `C` value in SVM or the `max_features` in TFIDF).
- **Grid Search** systematically tests various combinations of hyperparameters to find the optimal set that yields the highest score (often F1-Score) via cross-

validation.

In [5]:
```
# Import necessary libraries for Grid Search
from sklearn.model_selection import GridSearchCV

# Define the parameter grid to search over
parameters = {
    'tfidf__ngram_range': [(1, 1), (1, 2)],  # Test unigrams vs. unigrams+big
    'clf__C': [0.1, 1.0, 10.0]               # Test different regularization
}

# 1. Initialize GridSearchCV
grid_search = GridSearchCV(svm_pipeline, parameters, cv=2, n_jobs=-1, verbose

# 2. Run the search (Note: CV=2 and a small grid for speed)
print("Starting Grid Search (may take a moment)...")
grid_search.fit(X_train, y_train)
```

```
Starting Grid Search (may take a moment)...
Fitting 2 folds for each of 6 candidates, totalling 12 fits
```

Out[5]:
```
▸ GridSearchCV
▸ best_estimator_: Pipeline
    ▸ TfidfVectorizer
    ▸ LinearSVC
```

In [6]:
```
print("\nOptimal Parameters Found:")
print(grid_search.best_params_)
print(f"Best F1-Score: {grid_search.best_score_:.4f}")
```

```
Optimal Parameters Found:
{'clf__C': 10.0, 'tfidf__ngram_range': (1, 1)}
Best F1-Score: 0.9708
```

5. Summary and Next Steps

- We have moved beyond simple accuracy to analyze our model performance using the **Confusion Matrix**, **Precision**, **Recall**, and **F1-Score**.
- We also established the rigorous testing methods of **Cross-Validation** and **Hyperparameter Tuning**.
- You have now completed the entire classical Machine Learning (ML) section of the course (Chapters 6-8).
- In **Chapter 9**, we will explore the powerful world of modern NLP: **Word Embeddings** and **Semantic Similarity**, which is the first step toward deep learning and Transformer models.

Key Takeaways

- **Comprehensive Evaluation Mastery:** We moved beyond simple accuracy to implement robust evaluation using confusion matrices, precision, recall, and F1-score for thorough model assessment.
- **Metric Selection Wisdom:** We learned when to prioritize different metrics - precision for spam detection (minimize false positives), recall for medical diagnosis (minimize false negatives), and F1-score for balanced evaluation.
- **Rigorous Testing Methods:** We implemented cross-validation and hyperparameter tuning with GridSearchCV to ensure reliable, generalizable model performance.
- **Classical ML Completion:** We successfully completed the entire classical machine learning section (Chapters 6-8), establishing solid foundations for advanced NLP techniques.

Next Chapter Preview

- With classical ML mastery achieved, we're ready to enter the exciting world of **modern NLP**.
- Chapter 9 will introduce **Word Embeddings** and **Semantic Similarity** - the foundational concepts that bridge traditional NLP to deep learning and Transformer models.

About This Project

This notebook is part of the **Natural Language Processing using Python Programming for Beginners** repository - a comprehensive, beginner-friendly guide for mastering NLP using Python, NLTK, and SpaCy.

Repository: `NLP`

Author

Prakash Ukhalkar
GitHub prakash-ukhalkar

Built with ❤️ for the Python community