✏️

Fine-tuning Your Model

Created	@September 30, 2024 10:13 PM
Class	Supervised Learning with scikit-learn

Evaluating Classification Models

Key Details

- Accuracy alone is not always a useful metric for classification problems
- Class imbalance can lead to misleading accuracy scores
- Confusion matrix and derived metrics provide a more comprehensive evaluation

Class Imbalance

- Occurs when one class is more frequent than others
- Example: Fraudulent transaction detection (1% fraudulent, 99% legitimate)
- A model always predicting "legitimate" would have 99% accuracy but fail at its purpose

Confusion Matrix

- 2×2 matrix summarizing binary classifier performance
- Rows: Actual labels
- Columns: Predicted labels
- Components:
    - True Positives (TP)
    - True Negatives (TN)
    - False Positives (FP)
    - False Negatives (FN)

Important Metrics

Accuracy

- Sum of true predictions divided by total predictions
- Formula: $\text{Accuracy} = \frac{TP+TN}{TP+TN+FP+FN}$

Precision

- True positives divided by all positive predictions
- Also called Positive Predictive Value
- Formula: $\text{Precision} = \frac{TP}{TP+FP}$
- High precision means lower false positive rate

Recall

- True positives divided by all actual positives
- Also called Sensitivity
- Formula: $\text{Recall} = \frac{TP}{TP+FN}$
- High recall means lower false negative rate

F1-Score

- Harmonic mean of precision and recall
- Balances precision and recall
- Formula: $\text{F1-Score} = 2 \cdot \frac{\text{Precision} \cdot \text{Recall}}{\text{Precision} + \text{Recall}}$
- Useful when seeking a model with similar precision and recall

Implementation in Python

```
from sklearn.metrics import classification_report, confusion_
matrix

# Assume we have a classifier, X_train, X_test, y_train, y_te
```

```
st

# Fit the classifier
classifier.fit(X_train, y_train)

# Make predictions
y_pred = classifier.predict(X_test)

# Compute confusion matrix
conf_matrix = confusion_matrix(y_test, y_pred)
print("Confusion Matrix:")
print(conf_matrix)

# Compute classification report
class_report = classification_report(y_test, y_pred)
print("Classification Report:")
print(class_report)
```

Example Output Interpretation

- Confusion Matrix:
  - [[1106, ...], [..., ...]]
  - 1106 true negatives in the top left
- Classification Report:
  - Includes precision, recall, and F1-score for each class
  - For churn class: precision = 0.76, recall = 0.16
  - Low recall on churn class indicates poor model performance in identifying churned customers

Key Takeaways

- Accuracy alone can be misleading, especially with class imbalance
- Confusion matrix provides a comprehensive view of model performance

- Precision, recall, and F1-score offer deeper insights into model behavior
- Choose appropriate metrics based on the specific problem and goals
- Consider class imbalance when evaluating classification models

Logistic Regression and Model Evaluation

Key Details

- Logistic regression is used for classification despite its name
- Calculates probability of an observation belonging to a binary class
- Produces a linear decision boundary
- Default probability threshold is 0.5 in scikit-learn

Logistic Regression Overview

- Calculates probability (p) of belonging to a binary class
- Classification rule:
  - If $p \geq 0.5$, label as 1 (positive class)
  - If $p < 0.5$, label as 0 (negative class)
- Example: Diabetes prediction
  - $p \geq 0.5$: More likely to have diabetes (label 1)
  - $p < 0.5$: Less likely to have diabetes (label 0)

Implementation in Python

```
from sklearn.linear_model import LogisticRegression
from sklearn.model_selection import train_test_split

# Instantiate the classifier
clf = LogisticRegression()
```

```
# Split the data
X_train, X_test, y_train, y_test = train_test_split(X, y, tes
t_size=0.2, random_state=42)

# Fit the model
clf.fit(X_train, y_train)

# Predict on test set
y_pred = clf.predict(X_test)

# Predict probabilities
y_pred_probs = clf.predict_proba(X_test)[:, 1]  # Probabiliti
es for positive class
```

Probability Thresholds

- Default threshold: 0.5
- Can be applied to other models (e.g., KNN)
- Varying threshold affects model performance

Receiver Operating Characteristic (ROC) Curve

- Visualizes model performance across different thresholds
- Plots True Positive Rate (TPR) vs False Positive Rate (FPR)
- Dotted line represents a chance model (random guessing)
- Threshold extremes:
    - Threshold = 0: Predicts 1 for all observations (TPR = 1, FPR = 1)
    - Threshold = 1: Predicts 0 for all observations (TPR = 0, FPR = 0)

Plotting ROC Curve

```
from sklearn.metrics import roc_curve
import matplotlib.pyplot as plt
```

```
# Calculate ROC curve
fpr, tpr, thresholds = roc_curve(y_test, y_pred_probs)

# Plot ROC curve
plt.figure()
plt.plot([0, 1], [0, 1], 'k--')  # Random guessing line
plt.plot(fpr, tpr)
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')
plt.title('ROC Curve')
plt.show()
```

Area Under the ROC Curve (AUC)

- Quantifies model performance based on ROC curve
- Ranges from 0 to 1, with 1 being ideal
- AUC of 0.5 represents a model making random guesses

Calculating AUC

```
from sklearn.metrics import roc_auc_score

auc_score = roc_auc_score(y_test, y_pred_probs)
print(f"AUC Score: {auc_score:.2f}")
```

Key Takeaways

- Logistic regression is a powerful tool for binary classification
- ROC curve and AUC provide comprehensive evaluation of model performance
- Varying probability thresholds can optimize model for specific use cases
- AUC score quantifies overall model performance across all thresholds

Hyperparameter Tuning

Key Details

- Hyperparameters are parameters specified before fitting a model (e.g., alpha in ridge/lasso regression, n_neighbors in KNN)
- Hyperparameter tuning is crucial for building successful models
- Cross-validation is used during tuning to avoid overfitting to the test set
- Two main approaches: Grid Search and Random Search

Hyperparameter Tuning Process

1. Split data into training and test sets
2. Perform cross-validation on the training set
3. Withhold test set for final evaluation of the tuned model

Grid Search

- Exhaustively searches through a specified grid of hyperparameter values
- Performs k-fold cross-validation for each combination of hyperparameters

Implementation in Python

```
from sklearn.model_selection import GridSearchCV, KFold
from sklearn.neighbors import KNeighborsRegressor

# Set up KFold cross-validation
kf = KFold(n_splits=5, shuffle=True, random_state=42)

# Define parameter grid
param_grid = {
    'n_neighbors': [2, 5, 8, 11],
    'metric': ['euclidean', 'manhattan']
}
```

```
# Instantiate the model
knn = KNeighborsRegressor()

# Perform Grid Search
grid_search = GridSearchCV(knn, param_grid, cv=kf)
grid_search.fit(X_train, y_train)

# Get best parameters and score
print("Best parameters:", grid_search.best_params_)
print("Best score:", grid_search.best_score_)
```

Limitations

- Number of fits = (number of hyperparameters) × (number of values) × (number of folds)
- Doesn't scale well with increasing hyperparameters or values

Random Search

- Picks random hyperparameter values instead of exhaustively searching all options
- Often more efficient than Grid Search, especially with high-dimensional hyperparameter spaces

Implementation in Python

```
from sklearn.model_selection import RandomizedSearchCV

# Perform Random Search
random_search = RandomizedSearchCV(knn, param_grid, n_iter=1
0, cv=kf, random_state=42)
random_search.fit(X_train, y_train)

# Get best parameters and score
```

```
print("Best parameters:", random_search.best_params_)
print("Best score:", random_search.best_score_)

# Evaluate on test set
test_score = random_search.score(X_test, y_test)
print("Test set score:", test_score)
```

Mathematical Concept: Cross-Validation in Hyperparameter Tuning

Cross-validation is used to estimate the generalization performance of a model with a given set of hyperparameters. The process can be described mathematically as follows:

1. Split the training data into K folds: $D = \{D_1, D_2, ..., D_K\}$
2. For each hyperparameter combination θ and each fold k:
   - Train on $\text{K-1 folds : } M_{\theta,k} = \text{train}(D \setminus D_k, \theta)$$
   - Validate on the held-out fold: $\text{score}k = \text{evaluate}(M\theta, k, D_k)$
3. Compute the average score: $\text{CV}\theta = \frac{1}{K} \sum k = 1^K \text{score}_k$
4. Select the best hyperparameters: $\theta^* = \arg\max_\theta \text{CV}_\theta$

This process helps in selecting hyperparameters that generalize well across different subsets of the data.

Key Takeaways

- Hyperparameter tuning is essential for optimizing model performance
- Grid Search provides an exhaustive search but can be computationally expensive
- Random Search offers a more efficient alternative, especially for high-dimensional hyperparameter spaces
- Cross-validation during tuning helps prevent overfitting to the test set
- The final tuned model should be evaluated on a held-out test set