Classification – Chapter 3 Summary

Hands-On Machine Learning with Scikit-Learn and TensorFlow

Study Notes Summary

June 29, 2025

Contents

1 The Classification Problem

Classification is a supervised learning task where the goal is to predict discrete categories or classes for given input data.

1.1 Types of Classification

- **Binary Classification**: Distinguishing between two classes (e.g., spam vs. not spam)
- **Multiclass Classification**: Distinguishing between more than two classes (e.g., digit recognition 0-9)
- **Multilabel Classification**: Each instance can belong to multiple classes simultaneously
- **Multioutput Classification**: Multiple outputs, each with multiple possible classes

1.2 MNIST Dataset Example

The MNIST dataset contains 70,000 images of handwritten digits (0-9), each 28×28 pixels. It's a classic benchmark for classification algorithms.

Listing 1: Loading MNIST Dataset

```
from sklearn.datasets import fetch_openml
import numpy as np

# Load MNIST dataset
mnist = fetch_openml('mnist_784', version=1, as_frame=False)
X, y = mnist["data"], mnist["target"]

# Convert target to integers
y = y.astype(np.uint8)

# Split into train and test sets
X_train, X_test, y_train, y_test = X[:60000], X[60000:], y[:60000], y[60000:]
```

Tip

The MNIST dataset is already shuffled, but it's good practice to shuffle your data before training to ensure the learning algorithm doesn't get biased by the order of instances.

2 Training a Binary Classifier

Let's start with a binary classification problem: detecting whether a digit is a 5 or not.

Listing 2: Creating Binary Classification Target

```
# Create binary target: True for 5, False for all other digits
y_train_5 = (y_train == 5)
y_test_5 = (y_test == 5)
```

2.1 SGD Classifier

The Stochastic Gradient Descent (SGD) classifier is efficient for large datasets and can handle various loss functions.

Listing 3: Training SGD Classifier

```
from sklearn.linear_model import SGDClassifier

# Create and train SGD classifier
sgd_clf = SGDClassifier(max_iter=1000, tol=1e-3, random_state=42)
sgd_clf.fit(X_train, y_train_5)

# Make predictions
some_digit = X[0]
prediction = sgd_clf.predict([some_digit])
print(f"Prediction: {prediction[0]}")  # True or False
```

2.2 Decision Functions vs Predicted Classes

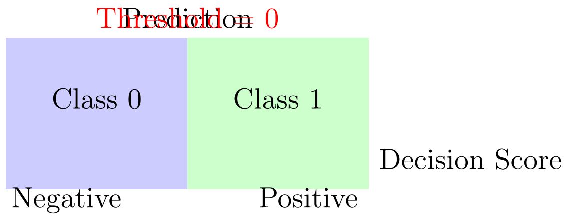

Figure 1: Decision Function Threshold

Listing 4: Decision Function vs Prediction

```
# Get decision scores
decision_scores = sgd_clf.decision_function([some_digit])
print(f"Decision score: {decision_scores[0]}")

# Prediction is based on whether score > 0
prediction = sgd_clf.predict([some_digit])
print(f"Prediction: {prediction[0]}")
```

3 Performance Measures

3.1 Accuracy and Its Limitations

Accuracy is the ratio of correct predictions to total predictions, but it can be misleading with imbalanced datasets.

Listing 5: Measuring Accuracy

```
from sklearn.metrics import accuracy_score
from sklearn.model_selection import cross_val_score

# Cross-validation accuracy
accuracy_scores = cross_val_score(sgd_clf, X_train, y_train_5,
                                  cv=3, scoring="accuracy")
print(f"Accuracy: {accuracy_scores.mean():.4f}")
```

Warning

For imbalanced datasets (like our 5-detector where only 10% are 5s), accuracy can be misleading. A classifier that always predicts "not 5" would achieve 90% accuracy!

3.2 Confusion Matrix

A confusion matrix provides detailed breakdown of correct and incorrect predictions.

Listing 6: Confusion Matrix

```python
from sklearn.metrics import confusion_matrix
from sklearn.model_selection import cross_val_predict

# Get predictions using cross-validation
y_train_pred = cross_val_predict(sgd_clf, X_train, y_train_5, cv=3)

# Generate confusion matrix
cm = confusion_matrix(y_train_5, y_train_pred)
print("Confusion Matrix:")
print(cm)
```

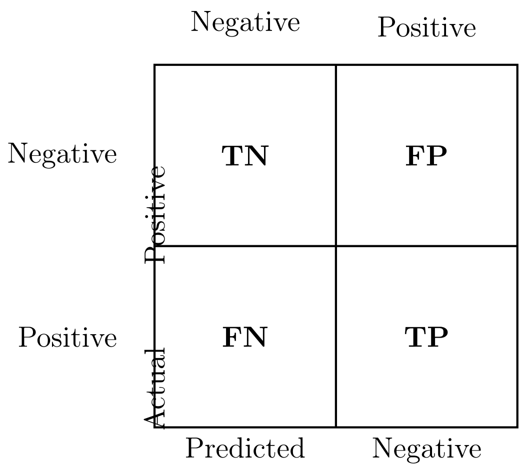

Figure 2: Confusion Matrix Structure

3.3 Precision, Recall, and F1 Score

$$\text{Precision} = \frac{TP}{TP + FP} \tag{1}$$

$$\text{Recall (Sensitivity)} = \frac{TP}{TP + FN} \tag{2}$$

$$\text{F1 Score} = \frac{2 \times \text{Precision} \times \text{Recall}}{\text{Precision} + \text{Recall}} \tag{3}$$

Listing 7: Precision

```python
from sklearn.metrics import precision_score, recall_score, f1_score

# Calculate metrics
precision = precision_score(y_train_5, y_train_pred)
recall = recall_score(y_train_5, y_train_pred)
f1 = f1_score(y_train_5, y_train_pred)

print(f"Precision: {precision:.4f}")
print(f"Recall: {recall:.4f}")
print(f"F1 Score: {f1:.4f}")
```

3.4 Precision/Recall Tradeoff

There's always a tradeoff between precision and recall. Adjusting the decision threshold affects both metrics.

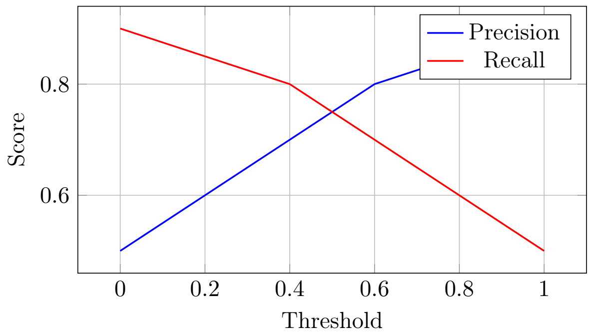

Figure 3: Precision/Recall Tradeoff

Listing 8: Adjusting Decision Threshold

```
from sklearn.metrics import precision_recall_curve

# Get decision scores for all training instances
y_scores = cross_val_predict(sgd_clf, X_train, y_train_5, cv=3,
                             method="decision_function")

# Calculate precision and recall for different thresholds
precisions, recalls, thresholds = precision_recall_curve(y_train_5, y_scores)

# Find threshold for 90% precision
threshold_90_precision = thresholds[np.argmax(precisions >= 0.90)]
print(f"Threshold for 90% precision: {threshold_90_precision}")

# Make predictions with custom threshold
y_train_pred_90 = (y_scores >= threshold_90_precision)
precision_90 = precision_score(y_train_5, y_train_pred_90)
recall_90 = recall_score(y_train_5, y_train_pred_90)
print(f"Precision: {precision_90:.4f}, Recall: {recall_90:.4f}")
```

3.5 ROC Curve and AUC

The Receiver Operating Characteristic (ROC) curve plots True Positive Rate vs False Positive Rate.

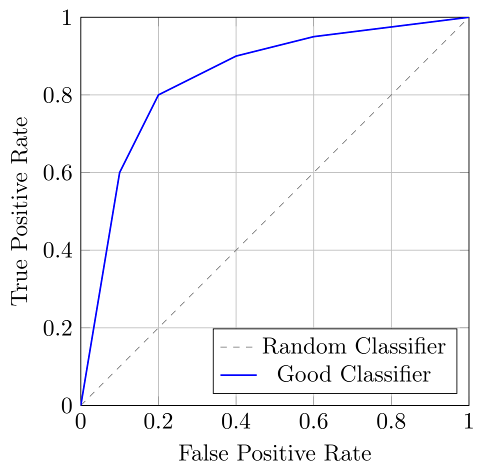

Figure 4: ROC Curve Example

Listing 9: ROC Curve and AUC

```
from sklearn.metrics import roc_curve, roc_auc_score

# Calculate ROC curve
fpr, tpr, thresholds = roc_curve(y_train_5, y_scores)

# Calculate AUC
auc_score = roc_auc_score(y_train_5, y_scores)
print(f"AUC Score: {auc_score:.4f}")
```

Note

Use ROC/AUC when classes are roughly balanced. Use Precision/Recall curves when dealing with imbalanced datasets or when false positives are more important than false negatives.

4 Using Cross-Validation

Cross-validation provides more robust performance estimates by training and testing on different data splits.

Listing 10: Cross-Validation for Classification

```
from sklearn.model_selection import StratifiedKFold

# Stratified K-Fold ensures balanced class distribution in each fold
skfolds = StratifiedKFold(n_splits=3, shuffle=True, random_state=42)

for train_index, test_index in skfolds.split(X_train, y_train_5):
    clone_clf = SGDClassifier(max_iter=1000, tol=1e-3, random_state=42)
    X_train_folds = X_train[train_index]
    y_train_folds = y_train_5[train_index]
    X_test_fold = X_train[test_index]
    y_test_fold = y_train_5[test_index]

    clone_clf.fit(X_train_folds, y_train_folds)
    y_pred = clone_clf.predict(X_test_fold)
    n_correct = sum(y_pred == y_test_fold)
    print(f"Accuracy: {n_correct/len(y_pred):.4f}")
```

Listing 11: Using cross$_{v}al_{p}redictforConfusionMatrix$

```python
# Get cross-validated predictions
y_train_pred = cross_val_predict(sgd_clf, X_train, y_train_5, cv=3)

# Calculate confusion matrix
cm = confusion_matrix(y_train_5, y_train_pred)
print("Confusion Matrix:")
print(cm)

# Generate classification report
from sklearn.metrics import classification_report
print("\nClassification Report:")
print(classification_report(y_train_5, y_train_pred))
```

5 Multiclass Classification

5.1 Strategies for Multiclass Classification

- **One-vs-Rest (OvR)**: Train one binary classifier per class
- **One-vs-One (OvO)**: Train one binary classifier for each pair of classes

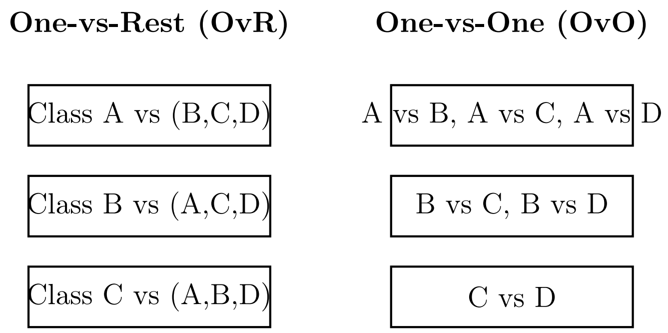

Figure 5: Multiclass Classification Strategies

Listing 12: Multiclass Classification with SGD

```python
# Train multiclass classifier (automatically uses OvR for SGD)
sgd_clf_multi = SGDClassifier(max_iter=1000, tol=1e-3, random_state=42)
sgd_clf_multi.fit(X_train, y_train)

# Make prediction
some_digit_pred = sgd_clf_multi.predict([some_digit])
print(f"Predicted digit: {some_digit_pred[0]}")

# Get decision scores for all classes
decision_scores = sgd_clf_multi.decision_function([some_digit])
print(f"Decision scores: {decision_scores}")
print(f"Predicted class: {np.argmax(decision_scores)}")
```

Listing 13: Forcing OvO Strategy

```python
from sklearn.multiclass import OneVsOneClassifier

# Force OvO strategy
ovo_clf = OneVsOneClassifier(SGDClassifier(max_iter=1000, tol=1e-3,
    random_state=42))
```

```python
ovo_clf.fit(X_train, y_train)
prediction = ovo_clf.predict([some_digit])
print(f"OvO Prediction: {prediction[0]}")
print(f"Number of binary classifiers: {len(ovo_clf.estimators_)}")
```

Tip

Scikit-Learn automatically detects when you use a binary classifier for multiclass tasks and uses the appropriate strategy (OvR for most classifiers, OvO for SVM).

6 Error Analysis

Error analysis helps identify patterns in classification mistakes and guide improvements.

Listing 14: Multiclass Confusion Matrix

```python
# Get cross-validated predictions for multiclass
y_train_pred_multi = cross_val_predict(sgd_clf_multi, X_train, y_train, cv=3)

# Generate confusion matrix
cm_multi = confusion_matrix(y_train, y_train_pred_multi)
print("Multiclass Confusion Matrix:")
print(cm_multi)
```

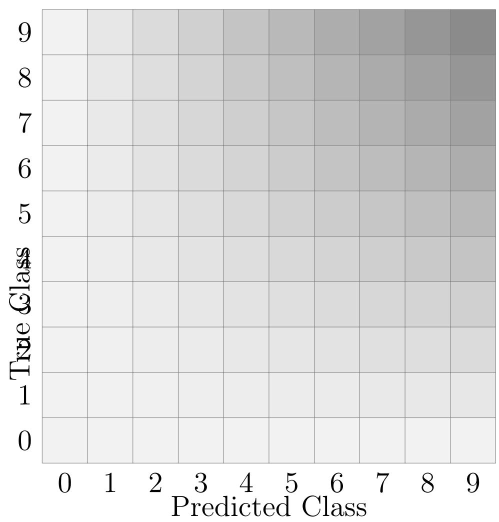

Figure 6: MNIST Confusion Matrix Visualization

Listing 15: Error Analysis - Normalized Confusion Matrix

```python
import matplotlib.pyplot as plt

# Normalize confusion matrix by row (true class)
row_sums = cm_multi.sum(axis=1, keepdims=True)
norm_cm = cm_multi / row_sums

# Fill diagonal with zeros to focus on errors
np.fill_diagonal(norm_cm, 0)

# Plot the error-focused confusion matrix
plt.figure(figsize=(8,8))
plt.matshow(norm_cm, cmap=plt.cm.gray)
```

```
plt.colorbar()
plt.title("Error Analysis - Normalized Confusion Matrix")
plt.xlabel("Predicted")
plt.ylabel("Actual")
plt.show()
```

Note

Look for patterns in the confusion matrix:

- Which classes are most often confused?
- Are there systematic errors (e.g., 8s mistaken for 3s)?
- Can you improve preprocessing or feature engineering?

7 Multilabel and Multioutput Classification

7.1 Multilabel Classification

In multilabel classification, each instance can belong to multiple classes simultaneously.

Listing 16: Multilabel Classification Example

```
from sklearn.neighbors import KNeighborsClassifier
import numpy as np

# Create multilabel targets (large digits >= 7, odd digits)
y_train_large = (y_train >= 7)
y_train_odd = (y_train % 2 == 1)
y_multilabel = np.c_[y_train_large, y_train_odd]

# Train KNN classifier for multilabel task
knn_clf = KNeighborsClassifier()
knn_clf.fit(X_train, y_multilabel)

# Make prediction
prediction = knn_clf.predict([some_digit])
print(f"Multilabel prediction: {prediction}")
# Output: [[False, True]] means not large (< 7) but odd
```

7.2 Multioutput Classification

Multioutput classification is a generalization where each output can have multiple classes.

Listing 17: Multioutput Classification - Noise Removal

```
# Add noise to MNIST images
noise = np.random.randint(0, 100, (len(X_train), 784))
X_train_mod = X_train + noise
noise = np.random.randint(0, 100, (len(X_test), 784))
X_test_mod = X_test + noise

# Target is the original clean image
y_train_mod = X_train
y_test_mod = X_test

# Train classifier to remove noise
knn_clf.fit(X_train_mod, y_train_mod)

```

```python
# Clean a noisy image
clean_digit = knn_clf.predict([X_test_mod[0]])
```

8 Key Tools and Code Summary

8.1 Essential Scikit-Learn Tools

Tool	Purpose
SGDClassifier	Stochastic Gradient Descent classifier
cross_val_score	Cross-validation scoring
cross_val_predict	Cross-validation predictions
confusion_matrix	Generate confusion matrix
classification_report	Comprehensive classification metrics
precision_score	Calculate precision
recall_score	Calculate recall
f1_score	Calculate F1 score
precision_recall_curve	Precision-recall curve data
roc_curve	ROC curve data
roc_auc_score	Area under ROC curve

Table 1: Key Classification Tools

Listing 18: Complete Classification Pipeline

```python
from sklearn.linear_model import SGDClassifier
from sklearn.model_selection import cross_val_score, cross_val_predict
from sklearn.metrics import (confusion_matrix, classification_report,
                             precision_score, recall_score, f1_score,
                             roc_auc_score, precision_recall_curve)

# 1. Train classifier
clf = SGDClassifier(max_iter=1000, tol=1e-3, random_state=42)
clf.fit(X_train, y_train)

# 2. Cross-validation evaluation
cv_scores = cross_val_score(clf, X_train, y_train, cv=3, scoring='accuracy')
print(f"CV Accuracy: {cv_scores.mean():.4f} (+/- {cv_scores.std() * 2:.4f})")

# 3. Get predictions and detailed metrics
y_pred = cross_val_predict(clf, X_train, y_train, cv=3)
print("\nConfusion Matrix:")
print(confusion_matrix(y_train, y_pred))
print("\nClassification Report:")
print(classification_report(y_train, y_pred))

# 4. For binary classification, get additional metrics
if len(np.unique(y_train)) == 2:
    y_scores = cross_val_predict(clf, X_train, y_train, cv=3,
                                 method="decision_function")
    auc = roc_auc_score(y_train, y_scores)
    print(f"\nAUC Score: {auc:.4f}")

# 5. Test set evaluation (final step)
test_accuracy = clf.score(X_test, y_test)
print(f"Test Accuracy: {test_accuracy:.4f}")
```

9 Key Takeaways

9.1 Chapter Summary

- **Classification Types**: Binary, multiclass, multilabel, and multioutput classification serve different purposes and require different approaches.
- **Performance Metrics**: Accuracy can be misleading with imbalanced data. Use precision, recall, F1-score, and confusion matrices for better evaluation.
- **Precision vs Recall**: There's always a tradeoff. Adjust decision thresholds based on your specific needs (false positives vs false negatives).
- **Cross-Validation**: Essential for robust performance estimation. Use stratified K-fold for classification to maintain class balance.
- **Multiclass Strategies**: Scikit-Learn automatically handles multiclass classification using OvR or OvO strategies.
- **Error Analysis**: Examine confusion matrices to identify patterns in misclassifications and guide improvements.

9.2 Best Practices

Tip

Classification Best Practices:

- Always use cross-validation for model evaluation
- Choose metrics appropriate for your problem (balanced vs imbalanced data)
- Examine confusion matrices to understand model behavior
- Consider the cost of different types of errors in your domain
- Use stratified sampling to maintain class distributions
- Start with simple models (like SGD) before moving to complex ones
- Always evaluate on a held-out test set for final performance assessment

9.3 When to Use Different Metrics

Metric	When to Use
Accuracy	Balanced datasets, equal cost for all errors
Precision	When false positives are costly (e.g., spam detection)
Recall	When false negatives are costly (e.g., medical diagnosis)
F1 Score	When you need a balance between precision and recall
ROC/AUC	Balanced datasets, when you need threshold-independent metric
PR Curve	Imbalanced datasets, when positive class is more important

Table 2: Choosing the Right Metric

Note

Remember: The choice of evaluation metric should always align with your business objectives and the costs associated with different types of errors in your specific domain.