Classification - Intermediate Handout

Machine Learning for Smarter Innovation

1 Classification - Intermediate Handout

Target Audience: Practitioners with Python knowledge **Duration**: 60 minutes reading + coding **Level**: Intermediate (implementation focused)

1.1 Setup

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from sklearn.model_selection import (
    train_test_split, cross_val_score, cross_validate,
    StratifiedKFold, GridSearchCV
)
from sklearn.linear_model import LogisticRegression
from sklearn.tree import DecisionTreeClassifier, plot_tree
from sklearn.ensemble import RandomForestClassifier,
    GradientBoostingClassifier
from sklearn.svm import SVC
from sklearn.preprocessing import StandardScaler
from sklearn.metrics import (
    classification_report, confusion_matrix, roc_auc_score,
    roc_curve, precision_recall_curve, make_scorer
)
from sklearn.feature_selection import SelectKBest, f_classif, RFE
import warnings
warnings.filterwarnings('ignore')

# Visualization settings
plt.rcParams.update({
    'font.size': 10,
    'axes.labelsize': 10,
    'figure.figsize': (10, 6)
})
```

This handout covers practical implementation of classification algorithms for binary and multiclass prediction problems. Classification assigns categorical labels to observations based on features. The techniques apply across domains including fraud detection, customer churn prediction, medical diagnosis, and innovation success forecasting.

1.2 1. Logistic Regression with Regularization

1.2.1 Concept Overview

Logistic regression estimates class probabilities using a linear decision boundary transformed by the sigmoid function. Despite its name, it solves classification problems. Regularization (controlled by parameter C) prevents overfitting by penalizing large coefficients.

1.2.2 Implementation

```
# Generate sample classification data
from sklearn.datasets import make_classification

X, y = make_classification(
    n_samples=1000, n_features=20, n_informative=10,
    n_redundant=5, n_classes=2, random_state=42
)
feature_names = [f'feature_{i}' for i in range(X.shape[1])]
X = pd.DataFrame(X, columns=feature_names)

# Split data
X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.2, random_state=42, stratify=y
)

# Scale features (important for logistic regression)
scaler = StandardScaler()
X_train_scaled = scaler.fit_transform(X_train)
X_test_scaled = scaler.transform(X_test)

# Compare regularization strengths
C_values = [0.001, 0.01, 0.1, 1, 10, 100]
results = []

for C in C_values:
    lr = LogisticRegression(C=C, max_iter=1000, random_state=42)
    cv_scores = cross_val_score(lr, X_train_scaled, y_train, cv=5, scoring='
    roc_auc')
    results.append({
        'C': C,
        'mean_auc': cv_scores.mean(),
        'std_auc': cv_scores.std()
    })
    print(f"C={C}: ROC-AUC = {cv_scores.mean():.3f} (+/- {cv_scores.std():.3f
    })")

# Train best model
best_C = results[np.argmax([r['mean_auc'] for r in results])]['C']
best_lr = LogisticRegression(C=best_C, max_iter=1000, random_state=42)
best_lr.fit(X_train_scaled, y_train)

# Evaluate
y_pred_proba = best_lr.predict_proba(X_test_scaled)[:, 1]
print(f"\nBest model (C={best_C}) Test ROC-AUC: {roc_auc_score(y_test,
    y_pred_proba):.3f}")
```

1.3 2. Decision Trees with Pruning

1.3.1 Concept Overview

Decision trees split data recursively based on feature values, creating interpretable if-then rules. Without pruning, trees overfit by memorizing training data. Control complexity through max_depth, min_samples_split, and min_samples_leaf.

1.3.2 Implementation with Hyperparameter Tuning

```
# Define parameter grid
param_grid = {
    'max_depth': [3, 5, 7, 10, None],
    'min_samples_split': [2, 5, 10, 20],
    'min_samples_leaf': [1, 2, 4, 8]
}

dt = DecisionTreeClassifier(random_state=42)
grid_search = GridSearchCV(
    dt, param_grid, cv=5, scoring='roc_auc', n_jobs=-1
)
grid_search.fit(X_train, y_train)

print(f"Best parameters: {grid_search.best_params_}")
print(f"Best CV ROC-AUC: {grid_search.best_score_:.3f}")

# Visualize the best tree
best_tree = grid_search.best_estimator_
plt.figure(figsize=(16, 8))
plot_tree(best_tree, feature_names=feature_names,
          class_names=['Class 0', 'Class 1'],
          filled=True, rounded=True, fontsize=8, max_depth=3)
plt.title('Decision Tree (showing first 3 levels)')
plt.tight_layout()
plt.savefig('decision_tree.pdf', bbox_inches='tight')
plt.show()

# Feature importance
importance_df = pd.DataFrame({
    'feature': feature_names,
    'importance': best_tree.feature_importances_
}).sort_values('importance', ascending=False)

print("\nTop 5 Most Important Features:")
print(importance_df.head())
```

1.4 3. Random Forest Ensemble

1.4.1 Concept Overview

Random Forest combines many decision trees, each trained on a bootstrap sample with random feature subsets. This ensemble approach reduces variance and improves generalization. Out-of-bag (OOB) samples provide free validation estimates.

1.4.2 Implementation

```
# Random Forest with OOB evaluation
rf = RandomForestClassifier(
    n_estimators=100,
    oob_score=True,
    random_state=42,
    n_jobs=-1
)
rf.fit(X_train, y_train)

print(f"Out-of-Bag Score: {rf.oob_score_:.3f}")
print(f"Test Score: {rf.score(X_test, y_test):.3f}")

# Feature importance with confidence intervals
importances = rf.feature_importances_
std = np.std([tree.feature_importances_ for tree in rf.estimators_], axis=0)

importance_df = pd.DataFrame({
    'feature': feature_names,
    'importance': importances,
    'std': std
}).sort_values('importance', ascending=False)

# Visualize top features
plt.figure(figsize=(10, 6))
top_n = 10
plt.barh(range(top_n), importance_df['importance'][:top_n],
         xerr=importance_df['std'][:top_n], alpha=0.8)
plt.yticks(range(top_n), importance_df['feature'][:top_n])
plt.xlabel('Importance')
plt.title('Random Forest Feature Importance (with std)')
plt.gca().invert_yaxis()
plt.tight_layout()
plt.savefig('rf_importance.pdf', bbox_inches='tight')
plt.show()
```

1.5 4. Gradient Boosting

1.5.1 Concept Overview

Gradient Boosting builds trees sequentially, with each tree correcting errors from previous trees. It often achieves top performance but requires careful tuning to prevent overfitting. Use early stopping based on validation performance.

1.5.2 Implementation with Early Stopping

```
# Gradient Boosting with early stopping
gb = GradientBoostingClassifier(
    n_estimators=500,
    learning_rate=0.1,
    max_depth=5,
    validation_fraction=0.2,
    n_iter_no_change=10,
    random_state=42
)
gb.fit(X_train, y_train)
```

```
print(f"Stopped at iteration: {gb.n_estimators_}")
print(f"Test ROC-AUC: {roc_auc_score(y_test, gb.predict_proba(X_test)[:, 1])
    :.3f}")

# Plot learning curves
train_scores = []
test_scores = []

for i, y_pred in enumerate(gb.staged_predict_proba(X_test)):
    test_scores.append(roc_auc_score(y_test, y_pred[:, 1]))

for i, y_pred in enumerate(gb.staged_predict_proba(X_train)):
    train_scores.append(roc_auc_score(y_train, y_pred[:, 1]))

plt.figure(figsize=(10, 6))
plt.plot(train_scores, label='Train', alpha=0.8)
plt.plot(test_scores, label='Test', alpha=0.8)
plt.xlabel('Boosting Iterations')
plt.ylabel('ROC-AUC')
plt.title('Gradient Boosting Learning Curves')
plt.legend()
plt.grid(True, alpha=0.3)
plt.tight_layout()
plt.savefig('gb_learning_curve.pdf', bbox_inches='tight')
plt.show()
```

1.6 5. Cross-Validation and Model Comparison

1.6.1 Stratified K-Fold with Multiple Metrics

```
# Define scoring metrics
scoring = {
    'accuracy': 'accuracy',
    'precision': 'precision',
    'recall': 'recall',
    'f1': 'f1',
    'roc_auc': 'roc_auc'
}

# Models to compare
models = {
    'Logistic Regression': LogisticRegression(max_iter=1000),
    'Decision Tree': DecisionTreeClassifier(max_depth=5, random_state=42),
    'Random Forest': RandomForestClassifier(n_estimators=100, random_state=42)
    ,
    'Gradient Boosting': GradientBoostingClassifier(n_estimators=100,
    random_state=42)
}

# Evaluate all models
skf = StratifiedKFold(n_splits=5, shuffle=True, random_state=42)
comparison_results = []

for name, model in models.items():
    cv_results = cross_validate(
        model, X_train_scaled, y_train,
        cv=skf, scoring=scoring, n_jobs=-1
    )
```

```
    comparison_results.append({
        'Model': name,
        'Accuracy': f"{cv_results['test_accuracy'].mean():.3f}",
        'Precision': f"{cv_results['test_precision'].mean():.3f}",
        'Recall': f"{cv_results['test_recall'].mean():.3f}",
        'F1': f"{cv_results['test_f1'].mean():.3f}",
        'ROC-AUC': f"{cv_results['test_roc_auc'].mean():.3f}"
    })

print("\nModel Comparison:")
print(pd.DataFrame(comparison_results).to_string(index=False))
```

1.7 6. Handling Imbalanced Data

1.7.1 Resampling and Class Weights

```
# Simulate imbalanced data
from sklearn.datasets import make_classification

X_imb, y_imb = make_classification(
    n_samples=1000, n_features=20, n_informative=10,
    weights=[0.9, 0.1], random_state=42
)
print(f"Class distribution: {np.bincount(y_imb)}")

X_train_imb, X_test_imb, y_train_imb, y_test_imb = train_test_split(
    X_imb, y_imb, test_size=0.2, random_state=42, stratify=y_imb
)

# Method 1: Class weights
rf_weighted = RandomForestClassifier(
    n_estimators=100,
    class_weight='balanced',
    random_state=42
)

# Method 2: SMOTE (if imblearn installed)
try:
    from imblearn.over_sampling import SMOTE
    smote = SMOTE(random_state=42)
    X_train_smote, y_train_smote = smote.fit_resample(X_train_imb, y_train_imb
    )
    print(f"After SMOTE: {np.bincount(y_train_smote)}")

    rf_smote = RandomForestClassifier(n_estimators=100, random_state=42)
    rf_smote.fit(X_train_smote, y_train_smote)
except ImportError:
    print("imblearn not installed, skipping SMOTE")

# Compare approaches
rf_baseline = RandomForestClassifier(n_estimators=100, random_state=42)
rf_baseline.fit(X_train_imb, y_train_imb)
rf_weighted.fit(X_train_imb, y_train_imb)

print("\nBaseline (no weighting):")
print(classification_report(y_test_imb, rf_baseline.predict(X_test_imb)))
```

```
print("\nClass-weighted:")
print(classification_report(y_test_imb, rf_weighted.predict(X_test_imb)))
```

1.8 7. Comprehensive Model Evaluation

1.8.1 ROC and Precision-Recall Curves

```
def evaluate_classifier(model, X_train, y_train, X_test, y_test, model_name):
    """Complete evaluation with visualizations."""

    model.fit(X_train, y_train)
    y_pred = model.predict(X_test)
    y_proba = model.predict_proba(X_test)[:, 1]

    # Metrics
    cm = confusion_matrix(y_test, y_pred)
    fpr, tpr, _ = roc_curve(y_test, y_proba)
    precision, recall, _ = precision_recall_curve(y_test, y_proba)
    roc_auc = roc_auc_score(y_test, y_proba)

    # Visualizations
    fig, axes = plt.subplots(1, 3, figsize=(15, 4))

    # Confusion Matrix
    axes[0].imshow(cm, cmap='Blues')
    axes[0].set_title('Confusion Matrix')
    axes[0].set_xlabel('Predicted')
    axes[0].set_ylabel('Actual')
    for i in range(2):
        for j in range(2):
            axes[0].text(j, i, str(cm[i, j]), ha='center', va='center',
    fontsize=14)

    # ROC Curve
    axes[1].plot(fpr, tpr, label=f'AUC = {roc_auc:.3f}')
    axes[1].plot([0, 1], [0, 1], 'k--')
    axes[1].set_xlabel('False Positive Rate')
    axes[1].set_ylabel('True Positive Rate')
    axes[1].set_title('ROC Curve')
    axes[1].legend()
    axes[1].grid(True, alpha=0.3)

    # Precision-Recall Curve
    axes[2].plot(recall, precision)
    axes[2].set_xlabel('Recall')
    axes[2].set_ylabel('Precision')
    axes[2].set_title('Precision-Recall Curve')
    axes[2].grid(True, alpha=0.3)

    plt.suptitle(f'{model_name} Evaluation', fontsize=12)
    plt.tight_layout()
    plt.savefig(f'{model_name.lower().replace(" ", "_")}_eval.pdf',
    bbox_inches='tight')
    plt.show()

    print(f"\n{model_name} Classification Report:")
    print(classification_report(y_test, y_pred))
```

```
    return model

# Evaluate best model
best_model = evaluate_classifier(
    RandomForestClassifier(n_estimators=100, random_state=42),
    X_train, y_train, X_test, y_test, 'Random Forest'
)
```

1.9 Common Hyperparameters

() * 0.25	() * 0.3409	() *	() * Typical Range
() * 0.25	() *	() *	() * Lower
gis- tic Re- gres- sion	100 = more reg- u- lar- iza- tion		
() * 0.25	() *	() *	() *
gis- tic Re- gres- sion	l2, de- fault, ela- tic- net for		l1 spar- sity
() * 0.25	() *	() *	() * Depth
ci- sion Tree	20 or None	= fully grown	
() * 0.25	() *	() *	() * Min_samples_split
ci- sion Tree	20	i- mum to split node	

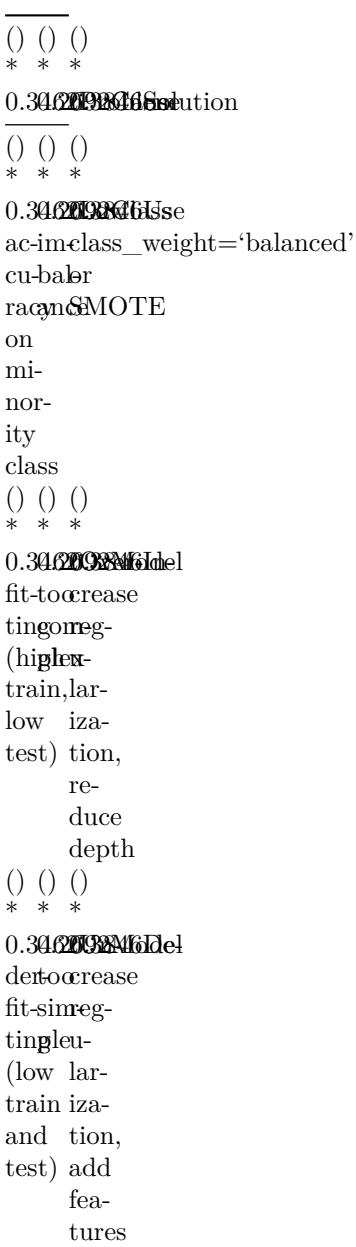
() *	() *	() *	() *
0.25	5	0.1	Typical 0.3409 Range
() *	() *	() *	() *
0.25	5	500	More trees = better but slower
() *	() *	() *	() *
0.25	5	log2, 0.7	log2, typical default
() *	() *	() *	() *
0.25	5	0.3	Lower rate = more trees needed
() *	() *	() *	() *
0.25	5	1000	Use early stopping
() *	() *	() *	() *
0.25	5	100	Regularization strength
() *	() *	() *	() *
0.25	5	auto, 0.01	RBF kernel width

1.10 Practice Projects

1. **Churn Prediction**: Build a customer churn classifier using subscription data. Handle class imbalance, identify key churn drivers, and optimize for recall to catch at-risk customers.

2. **Fraud Detection**: Create a fraud detection system with highly imbalanced transaction data. Focus on precision-recall tradeoffs and cost-sensitive evaluation.
3. **Medical Diagnosis**: Classify patient conditions from symptoms and test results. Prioritize interpretability with decision trees and feature importance analysis.
4. **Innovation Success Predictor**: Predict startup success from founding team, funding, and market characteristics. Compare ensemble methods and analyze feature interactions.

1.11 Troubleshooting



Problem	Cause	Solution
Logistic regression not converging	Too few iterations	Increase max_iter to 1000+
SVM slow on large data	Quadratic scaling	Use LinearSVC or subsample data
Random Forest slow	Too many trees/features	Reduce n_estimators, use max_features
Feature importance varies	Random forest variance	Average over multiple runs
Unstable validation	Small dataset or high variance	Use more folds, try different splits

1.12 Next Steps

- Read the advanced handout for ensemble stacking and calibration
- Experiment with XGBoost and LightGBM for improved performance
- Implement threshold tuning for cost-sensitive applications
- Explore model interpretability tools (SHAP, LIME)
- Build end-to-end classification pipelines with sklearn Pipeline

---

Classification maps features to categories. The best model depends on data characteristics, interpretability requirements, and business constraints. Always validate with cross-validation and evaluate with metrics aligned to business objectives.