Handout 2: Intermediate Classification - Building Robust Models

Machine Learning for Smarter Innovation

1 Handout 2: Intermediate Classification - Building Robust Models

1.1 Learning Objectives

By the end of this handout, you will: - Implement all major classification algorithms - Master cross-validation techniques - Engineer features for better performance - Handle imbalanced datasets effectively - Tune hyperparameters systematically - Evaluate models with multiple metrics

1.2 Part 1: Comprehensive Algorithm Implementation

1.2.1 1.1 Logistic Regression with Regularization

```
import numpy as np
import pandas as pd
from sklearn.linear_model import LogisticRegression, LogisticRegressionCV
from sklearn.preprocessing import StandardScaler
from sklearn.model_selection import train_test_split, cross_val_score
from sklearn.metrics import classification_report, roc_auc_score
import matplotlib.pyplot as plt

# Load and prepare data
data = pd.read_csv('innovation_data.csv')
X = data.drop('success', axis=1)
y = data['success']

# Scale features for logistic regression
scaler = StandardScaler()
X_scaled = scaler.fit_transform(X)

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

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

for C in C_values:
    lr = LogisticRegression(C=C, max_iter=1000)
    cv_scores = cross_val_score(lr, X_train, y_train, cv=5, scoring='roc_auc')
```

```
    scores.append(cv_scores.mean())
    print(f"C={C}: ROC-AUC = {cv_scores.mean():.3f} (+/- {cv_scores.std():.3f
    })")

# Plot regularization effect
plt.figure(figsize=(10, 6))
plt.semilogx(C_values, scores, marker='o')
plt.xlabel('Regularization Parameter C')
plt.ylabel('Cross-Validated ROC-AUC')
plt.title('Logistic Regression: Effect of Regularization')
plt.grid(True, alpha=0.3)
plt.show()

# Use best model
best_C = C_values[np.argmax(scores)]
best_lr = LogisticRegression(C=best_C, max_iter=1000)
best_lr.fit(X_train, y_train)

# Detailed evaluation
y_pred = best_lr.predict(X_test)
y_pred_proba = best_lr.predict_proba(X_test)[:, 1]

print("\nBest Model Performance:")
print(classification_report(y_test, y_pred))
print(f"ROC-AUC Score: {roc_auc_score(y_test, y_pred_proba):.3f}")
```

1.2.2 1.2 Decision Trees with Pruning

```
from sklearn.tree import DecisionTreeClassifier, plot_tree
from sklearn.model_selection import GridSearchCV

# Hyperparameter tuning for decision tree
param_grid = {
    'max_depth': [3, 5, 7, 10, None],
    'min_samples_split': [2, 5, 10, 20],
    'min_samples_leaf': [1, 2, 4, 8],
    'criterion': ['gini', 'entropy']
}

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

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

# Visualize the best tree
best_tree = grid_search.best_estimator_
plt.figure(figsize=(20, 10))
plot_tree(best_tree, feature_names=X.columns,
          class_names=['Fail', 'Success'],
          filled=True, rounded=True, fontsize=10)
plt.title('Optimized Decision Tree for Innovation Classification')
plt.show()

# Feature importance
importances = best_tree.feature_importances_
feature_importance_df = pd.DataFrame({
    'feature': X.columns,
```

```
    'importance': importances
}).sort_values('importance', ascending=False)

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

1.2.3 1.3 Random Forest with Out-of-Bag Evaluation

```
from sklearn.ensemble import RandomForestClassifier

# Random Forest with OOB score
rf = RandomForestClassifier(
    n_estimators=100,
    oob_score=True,  # Enable out-of-bag evaluation
    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 analysis
feature_importances = pd.DataFrame({
    'feature': X.columns,
    'importance': rf.feature_importances_
}).sort_values('importance', ascending=False)

# Visualize feature importances
plt.figure(figsize=(10, 8))
plt.barh(feature_importances['feature'][:10],
         feature_importances['importance'][:10])
plt.xlabel('Importance')
plt.title('Top 10 Most Important Features - Random Forest')
plt.gca().invert_yaxis()
plt.tight_layout()
plt.show()

# Analyze individual tree predictions
tree_predictions = np.array([tree.predict(X_test)
                             for tree in rf.estimators_])
tree_accuracies = [accuracy_score(y_test, pred)
                   for pred in tree_predictions]

print(f"\nIndividual tree accuracy: {np.mean(tree_accuracies):.3f} "
      f"(+/- {np.std(tree_accuracies):.3f})")
print(f"Ensemble accuracy: {rf.score(X_test, y_test):.3f}")
```

1.2.4 1.4 Support Vector Machines with Different Kernels

```
from sklearn.svm import SVC
from sklearn.preprocessing import StandardScaler

# SVM requires scaled features
scaler = StandardScaler()
X_train_scaled = scaler.fit_transform(X_train)
X_test_scaled = scaler.transform(X_test)
```

```
# Compare different kernels
kernels = ['linear', 'poly', 'rbf', 'sigmoid']
results = {}

for kernel in kernels:
    svm = SVC(kernel=kernel, probability=True, random_state=42)
    cv_scores = cross_val_score(svm, X_train_scaled, y_train,
                                cv=5, scoring='roc_auc')
    results[kernel] = {
        'mean': cv_scores.mean(),
        'std': cv_scores.std()
    }
    print(f"{kernel}: {cv_scores.mean():.3f} (+/- {cv_scores.std():.3f})")

# Train best kernel with hyperparameter tuning
param_grid = {
    'C': [0.1, 1, 10, 100],
    'gamma': ['scale', 'auto', 0.001, 0.01, 0.1]
}

svm_rbf = SVC(kernel='rbf', probability=True, random_state=42)
grid_search = GridSearchCV(svm_rbf, param_grid, cv=5,
                           scoring='roc_auc', n_jobs=-1)
grid_search.fit(X_train_scaled, y_train)

print(f"\nBest RBF-SVM parameters: {grid_search.best_params_}")
print(f"Best ROC-AUC: {grid_search.best_score_:.3f}")
```

1.2.5 1.5 Gradient Boosting with Early Stopping

```
from sklearn.ensemble import GradientBoostingClassifier
import matplotlib.pyplot as plt

# Gradient Boosting with validation-based 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_}")

# Plot training progress
test_scores = []
train_scores = []

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

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

plt.figure(figsize=(10, 6))
plt.plot(range(1, len(test_scores) + 1), test_scores, label='Test', alpha=0.8)
plt.plot(range(1, len(train_scores) + 1), train_scores, label='Train', alpha
    =0.8)
plt.xlabel('Boosting Iterations')
```

```
plt.ylabel('ROC-AUC Score')
plt.title('Gradient Boosting Learning Curves')
plt.legend()
plt.grid(True, alpha=0.3)
plt.show()
```

1.3 Part 2: Advanced Cross-Validation Techniques

1.3.1 2.1 Stratified K-Fold with Multiple Metrics

```
from sklearn.model_selection import StratifiedKFold, cross_validate
from sklearn.metrics import make_scorer, precision_score, recall_score,
    f1_score

# Define multiple scoring metrics
scoring = {
    'accuracy': 'accuracy',
    'precision': make_scorer(precision_score),
    'recall': make_scorer(recall_score),
    'f1': make_scorer(f1_score),
    'roc_auc': 'roc_auc'
}

# Stratified K-Fold ensures class balance in each fold
skf = StratifiedKFold(n_splits=5, shuffle=True, random_state=42)

# Evaluate multiple models
models = {
    'Logistic Regression': LogisticRegression(max_iter=1000),
    'Random Forest': RandomForestClassifier(n_estimators=100),
    'Gradient Boosting': GradientBoostingClassifier(n_estimators=100)
}

results_df = []
for name, model in models.items():
    cv_results = cross_validate(model, X, y, cv=skf,
                                scoring=scoring, n_jobs=-1)

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

results_table = pd.DataFrame(results_df)
print(results_table.to_string(index=False))
```

1.3.2 2.2 Time Series Split for Temporal Data

```
from sklearn.model_selection import TimeSeriesSplit
```

```
# When data has temporal component (e.g., innovations over time)
# Sort data by date if not already sorted
data_sorted = data.sort_values('launch_date')
X_sorted = data_sorted.drop(['success', 'launch_date'], axis=1)
y_sorted = data_sorted['success']

# Time series cross-validation
tscv = TimeSeriesSplit(n_splits=5)

for fold, (train_idx, test_idx) in enumerate(tscv.split(X_sorted)):
    print(f"Fold {fold + 1}:")
    print(f"  Train: index {train_idx[0]} to {train_idx[-1]}")
    print(f"  Test:  index {test_idx[0]} to {test_idx[-1]}")

    # Train and evaluate
    X_train_fold = X_sorted.iloc[train_idx]
    X_test_fold = X_sorted.iloc[test_idx]
    y_train_fold = y_sorted.iloc[train_idx]
    y_test_fold = y_sorted.iloc[test_idx]

    model = RandomForestClassifier(n_estimators=100, random_state=42)
    model.fit(X_train_fold, y_train_fold)
    score = model.score(X_test_fold, y_test_fold)
    print(f"  Accuracy: {score:.3f}\n")
```

1.4 Part 3: Feature Engineering

1.4.1 3.1 Creating Interaction Features

```
from sklearn.preprocessing import PolynomialFeatures

# Create interaction features
poly = PolynomialFeatures(degree=2, include_bias=False)
X_poly = poly.fit_transform(X)

# Get feature names
feature_names = poly.get_feature_names_out(X.columns)
X_poly_df = pd.DataFrame(X_poly, columns=feature_names)

print(f"Original features: {X.shape[1]}")
print(f"Polynomial features: {X_poly_df.shape[1]}")

# Select most important polynomial features
rf_poly = RandomForestClassifier(n_estimators=100, random_state=42)
rf_poly.fit(X_poly_df, y)

# Get top interaction features
importance_df = pd.DataFrame({
    'feature': feature_names,
    'importance': rf_poly.feature_importances_
}).sort_values('importance', ascending=False)

# Show top interaction features (containing ' ')
interaction_features = importance_df[importance_df['feature'].str.contains(' '
    )]
print("\nTop 5 Interaction Features:")
print(interaction_features.head())
```

1.4.2 3.2 Feature Selection Techniques

```
from sklearn.feature_selection import (
    SelectKBest, f_classif, RFE, SelectFromModel
)

# Method 1: Univariate Selection
selector_univariate = SelectKBest(score_func=f_classif, k=10)
X_selected_uni = selector_univariate.fit_transform(X, y)
selected_features_uni = X.columns[selector_univariate.get_support()]
print(f"Univariate selection: {list(selected_features_uni)}")

# Method 2: Recursive Feature Elimination
estimator = LogisticRegression(max_iter=1000)
selector_rfe = RFE(estimator, n_features_to_select=10)
X_selected_rfe = selector_rfe.fit_transform(X, y)
selected_features_rfe = X.columns[selector_rfe.support_]
print(f"RFE selection: {list(selected_features_rfe)}")

# Method 3: Tree-based Selection
selector_tree = SelectFromModel(
    RandomForestClassifier(n_estimators=100, random_state=42),
    threshold='median'
)
selector_tree.fit(X, y)
X_selected_tree = selector_tree.transform(X)
selected_features_tree = X.columns[selector_tree.get_support()]
print(f"Tree-based selection: {list(selected_features_tree)}")

# Compare performance with different feature sets
feature_sets = {
    'All': X,
    'Univariate': X[selected_features_uni],
    'RFE': X[selected_features_rfe],
    'Tree-based': X[selected_features_tree]
}

for name, X_features in feature_sets.items():
    scores = cross_val_score(
        RandomForestClassifier(n_estimators=100, random_state=42),
        X_features, y, cv=5, scoring='roc_auc'
    )
    print(f"{name}: {scores.mean():.3f} ({X_features.shape[1]} features)")
```

1.5 Part 4: Handling Imbalanced Data

1.5.1 4.1 Resampling Techniques

```
from imblearn.over_sampling import SMOTE, RandomOverSampler
from imblearn.under_sampling import RandomUnderSampler
from imblearn.combine import SMOTETomek

# Check class imbalance
print(f"Class distribution: {y.value_counts().to_dict()}")
print(f"Imbalance ratio: {y.value_counts()[0] / y.value_counts()[1]:.2f}:1")

# Method 1: Random Oversampling
ros = RandomOverSampler(random_state=42)
X_ros, y_ros = ros.fit_resample(X_train, y_train)
print(f"After ROS: {pd.Series(y_ros).value_counts().to_dict()}")
```

```
# Method 2: SMOTE (Synthetic Minority Oversampling)
smote = SMOTE(random_state=42)
X_smote, y_smote = smote.fit_resample(X_train, y_train)
print(f"After SMOTE: {pd.Series(y_smote).value_counts().to_dict()}")

# Method 3: Combined approach
smt = SMOTETomek(random_state=42)
X_combined, y_combined = smt.fit_resample(X_train, y_train)
print(f"After SMOTETomek: {pd.Series(y_combined).value_counts().to_dict()}")

# Compare methods
methods = {
    'Original': (X_train, y_train),
    'Oversampling': (X_ros, y_ros),
    'SMOTE': (X_smote, y_smote),
    'Combined': (X_combined, y_combined)
}

for name, (X_resampled, y_resampled) in methods.items():
    model = RandomForestClassifier(n_estimators=100, random_state=42)
    model.fit(X_resampled, y_resampled)
    y_pred = model.predict(X_test)

    print(f"\n{name}:")
    print(classification_report(y_test, y_pred, digits=3))
```

1.5.2 4.2 Cost-Sensitive Learning

```
# Adjust class weights to handle imbalance
class_weights = {
    0: 1,
    1: len(y[y==0]) / len(y[y==1])  # Inverse of class frequency
}

print(f"Class weights: {class_weights}")

# Apply to different algorithms
models_weighted = {
    'Logistic (weighted)': LogisticRegression(
        class_weight='balanced', max_iter=1000
    ),
    'RF (weighted)': RandomForestClassifier(
        n_estimators=100, class_weight='balanced', random_state=42
    ),
    'SVM (weighted)': SVC(
        class_weight='balanced', probability=True, random_state=42
    )
}

for name, model in models_weighted.items():
    scores = cross_val_score(model, X, y, cv=5, scoring='f1')
    print(f"{name}: F1 = {scores.mean():.3f}    {scores.std():.3f}")
```

1.6 Part 5: Model Evaluation Metrics

1.6.1 5.1 Comprehensive Evaluation

```
from sklearn.metrics import (
    confusion_matrix, classification_report,
    roc_curve, auc, precision_recall_curve
)

def evaluate_model(model, X_train, y_train, X_test, y_test, model_name):
    """Comprehensive model evaluation function"""

    # Train model
    model.fit(X_train, y_train)

    # Predictions
    y_pred = model.predict(X_test)
    y_proba = model.predict_proba(X_test)[:, 1]

    # Confusion Matrix
    cm = confusion_matrix(y_test, y_pred)

    # ROC Curve
    fpr, tpr, roc_thresholds = roc_curve(y_test, y_proba)
    roc_auc = auc(fpr, tpr)

    # Precision-Recall Curve
    precision, recall, pr_thresholds = precision_recall_curve(y_test, y_proba)

    # Create visualization
    fig, axes = plt.subplots(2, 2, figsize=(12, 10))
    fig.suptitle(f'{model_name} - Evaluation Metrics', fontsize=16)

    # Plot 1: Confusion Matrix
    axes[0, 0].imshow(cm, interpolation='nearest', cmap='Blues')
    axes[0, 0].set_title('Confusion Matrix')
    axes[0, 0].set_xlabel('Predicted')
    axes[0, 0].set_ylabel('Actual')
    for i in range(2):
        for j in range(2):
            axes[0, 0].text(j, i, str(cm[i, j]), ha='center', va='center')

    # Plot 2: ROC Curve
    axes[0, 1].plot(fpr, tpr, label=f'ROC (AUC = {roc_auc:.3f})')
    axes[0, 1].plot([0, 1], [0, 1], 'k--', label='Random')
    axes[0, 1].set_xlabel('False Positive Rate')
    axes[0, 1].set_ylabel('True Positive Rate')
    axes[0, 1].set_title('ROC Curve')
    axes[0, 1].legend()
    axes[0, 1].grid(True, alpha=0.3)

    # Plot 3: Precision-Recall Curve
    axes[1, 0].plot(recall, precision, label='PR Curve')
    axes[1, 0].set_xlabel('Recall')
    axes[1, 0].set_ylabel('Precision')
    axes[1, 0].set_title('Precision-Recall Curve')
    axes[1, 0].grid(True, alpha=0.3)

    # Plot 4: Feature Importance (if available)
    if hasattr(model, 'feature_importances_'):
        importances = model.feature_importances_
        indices = np.argsort(importances)[::-1][:10]
        axes[1, 1].barh(range(10), importances[indices])
        axes[1, 1].set_yticks(range(10))
        axes[1, 1].set_yticklabels([X.columns[i] for i in indices])
        axes[1, 1].set_xlabel('Importance')
        axes[1, 1].set_title('Top 10 Features')
```

```
        axes[1, 1].invert_yaxis()
    else:
        axes[1, 1].text(0.5, 0.5, 'Feature importance\nnot available',
                        ha='center', va='center', fontsize=12)
        axes[1, 1].set_title('Feature Importance')

    plt.tight_layout()
    plt.show()

    # Print classification report
    print(f"\n{model_name} - Classification Report:")
    print(classification_report(y_test, y_pred))

    return model

# Evaluate different models
models_to_evaluate = {
    'Random Forest': RandomForestClassifier(n_estimators=100, random_state=42)
    ,
    'Gradient Boosting': GradientBoostingClassifier(n_estimators=100,
    random_state=42),
    'Logistic Regression': LogisticRegression(max_iter=1000)
}

trained_models = {}
for name, model in models_to_evaluate.items():
    trained_models[name] = evaluate_model(
        model, X_train, y_train, X_test, y_test, name
    )
```

1.7 Summary and Best Practices

1. **Always use cross-validation** - Single train/test split can be misleading
2. **Scale features for distance-based algorithms** - SVM, KNN, Neural Networks
3. **Handle imbalanced data appropriately** - Don't just optimize for accuracy
4. **Engineer features thoughtfully** - Domain knowledge matters
5. **Compare multiple algorithms** - No single best algorithm for all problems
6. **Monitor for overfitting** - Use validation curves and learning curves
7. **Consider ensemble methods** - Often provide best performance

This intermediate guide provides the foundation for building robust classification systems. Practice with different datasets to build intuition!