Classification - Advanced Handout

Machine Learning for Smarter Innovation

1 Classification - Advanced Handout

Target Audience: Data scientists and ML engineers **Duration**: 90 minutes reading **Level**: Advanced (mathematical foundations and production systems)

1.1 Mathematical Foundations of Classification

1.1.1 Decision Theory Framework

Classification assigns observations to categories by partitioning the feature space. For input x in R^d and labels Y = {1, . . . , K}, we seek decision function h: R^d -> Y minimizing expected loss:

R(h) = E[L(Y, h(X))] = sum_k P(Y=k) E[L(k, h(X)) | Y=k]

The Bayes optimal classifier minimizes this risk:

h*(x) = argmax_k P(Y=k|X=x)

For 0-1 loss L(y, y') = I(y != y'), the Bayes risk is:

R* = 1 - E[max_k P(Y=k|X)]

This irreducible error represents the noise floor that no classifier can surpass.

1.1.2 Discriminative vs Generative Models

Generative models learn joint distribution P(X, Y) = P(X|Y)P(Y), then apply Bayes rule:

P(Y|X) = P(X|Y)P(Y) / P(X)

Examples: Naive Bayes, Linear Discriminant Analysis, Gaussian Mixture Models.

Discriminative models learn P(Y|X) directly or learn decision boundary without probabilistic interpretation.

Examples: Logistic Regression, SVM, Neural Networks.

Discriminative models often achieve lower asymptotic error, but generative models can be more data-efficient and handle missing data naturally.

1.1.3 Information-Theoretic View

Entropy measures uncertainty in class distribution:

H(Y) = -sum_k P(Y=k) log P(Y=k)

Conditional entropy after observing feature X:

H(Y|X) = -sum_x P(x) sum_k P(k|x) log P(k|x)

Information gain (mutual information) measures predictive value:

I(Y; X) = H(Y) - H(Y|X)

Decision trees maximize information gain at each split. For continuous features, information gain evaluates candidate split thresholds.

1.2 Decision Trees: Theory and Optimization

1.2.1 Recursive Partitioning

Decision trees partition feature space into axis-aligned regions. At node t with samples S_t, we seek split s = (j, theta) on feature j at threshold theta maximizing:

Delta I(s, t) = I(t) - [p_L I(t_L) + p_R I(t_R)]

where p_L, p_R are proportions going left/right, and I(t) is impurity at node t.

Gini Impurity: G(t) = 1 - sum_k p_kˆ2

Entropy: H(t) = -sum_k p_k log p_k

where p_k is proportion of class k at node t. Both measure "impurity" but Gini is computationally cheaper.

1.2.2 Pruning Theory

Minimal cost-complexity pruning balances accuracy and tree size. Define:

R_alpha(T) = R(T) + alpha |T|

where R(T) is training error (or cross-entropy), |T| is number of leaves, and alpha >= 0 is complexity parameter.

For increasing alpha, the sequence of optimal subtrees is nested: T_max > T_1 > T_2 > ... > {root}

Cross-validation selects optimal alpha. This prevents overfitting without early stopping heuristics.

1.2.3 Optimal Decision Trees

Finding the globally optimal tree is NP-hard. CART's greedy splitting is locally optimal. Recent work uses mixed-integer optimization for small trees:

min_{tree T} R(T) + alpha |T| subject to: path constraints, leaf purity constraints

This yields provably optimal trees but scales only to thousands of samples.

1.3 Ensemble Methods: Theory

1.3.1 Bootstrap Aggregating (Bagging)

Bagging reduces variance by averaging B bootstrap models:

h_bag(x) = (1/B) sum_{b=1}ˆB h_b(x)

For regression with independent base learners of variance sigmaˆ2:

Var(h_bag) = sigmaˆ2 / B

For classification, majority voting achieves error rate:

P(error) = sum_{k > B/2} C(B,k) pˆk (1-p)ˆ{B-k}

where p is individual classifier error rate. By law of large numbers, as B -> inf, ensemble error approaches Bayes error if base classifiers are accurate ($p < 0.5$) and diverse.

1.3.2 Random Forests

Random Forests combine bagging with feature randomization. At each split, sample m features from d total (typically m = sqrt(d) for classification).

The forest correlation rho between trees affects variance:

Var(h_RF) = rho * sigmaˆ2 + (1-rho) * sigmaˆ2 / B

Lower correlation (more randomness) reduces variance but increases bias. The sqrt(d) heuristic balances this tradeoff.

Out-of-Bag Error: Each tree excludes ~37% of data (out-of-bag). OOB predictions provide unbiased error estimate without separate validation set:

OOB_error = (1/n) sum_i I(argmax_k avg_{b: i in OOB_b} h_bˆk(x_i) != y_i)

1.3.3 Boosting: Theoretical Framework

Boosting combines weak learners sequentially, focusing on previously misclassified examples.

AdaBoost: For binary classification with weak learners h_t: X -> {-1, +1}, define weighted error:

epsilon_t = sum_i w_i I(h_t(x_i) != y_i)

Update weights:

alpha_t = 0.5 * log((1 - epsilon_t) / epsilon_t) w_i <- w_i * exp(-alpha_t * y_i * h_t(x_i))

Final classifier:

H(x) = sign(sum_t alpha_t h_t(x))

Training Error Bound: training_error <= prod_t sqrt(4 * epsilon_t * (1 - epsilon_t))

If each weak learner achieves epsilon_t <= 0.5 - gamma for some gamma > 0, training error decays exponentially.

Margin Theory: AdaBoost maximizes the margin rho_i = y_i * F(x_i) / ||alpha||_1. Large margins correlate with generalization. The margin distribution, not just minimum margin, affects test error.

1.3.4 Gradient Boosting

Gradient boosting views boosting as gradient descent in function space. For loss L(y, F(x)), at iteration m:

1. Compute pseudo-residuals: r_im = -[dL(y_i, F(x_i))/dF(x_i)]{F=F{m-1}}
2. Fit base learner h_m to residuals
3. Update: F_m(x) = F_{m-1}(x) + nu * h_m(x)

For squared loss, residuals are actual residuals. For log-loss (classification):

r_im = y_i - p_{m-1}(x_i)

where p(x) = 1/(1 + exp(-F(x))).

Regularization: - Shrinkage: nu < 1 (learning rate) - Subsampling: fit each tree on fraction of data - Tree constraints: max_depth, min_samples_leaf

XGBoost/LightGBM extensions: - Second-order approximation: uses Hessian in splitting criterion - Histogram-based splitting: discretize features for speed - Leaf-wise growth: grow deepest leaf first - Regularization: L1/L2 on leaf weights

---

1.4 Support Vector Machines

1.4.1 Maximum Margin Classification

For linearly separable data, SVM finds the hyperplane w.x + b = 0 maximizing margin:

margin = 2 / ||w||

This formulation:

min_{w,b} (1/2) ||w||^2 subject to: y_i(w.x_i + b) >= 1, for all i

The constraint y_i(w.x_i + b) >= 1 defines the margin. Points on the margin (support vectors) satisfy equality.

Dual Formulation: Using Lagrange multipliers alpha_i:

max_alpha sum_i alpha_i - (1/2) sum_{i,j} alpha_i alpha_j y_i y_j x_i.x_j subject to: alpha_i >= 0, sum_i alpha_i y_i = 0

Solution: w* = sum_i alpha_i* y_i x_i (sparse: only support vectors have alpha_i > 0)

1.4.2 Soft Margin and Regularization

For non-separable data, introduce slack variables xi_i:

min_{w,b,xi} (1/2) ||w||^2 + C sum_i xi_i subject to: y_i(w.x_i + b) >= 1 - xi_i, xi_i >= 0

Parameter C controls regularization: - Large C: narrow margin, low training error - Small C: wide margin, more misclassifications allowed

This is equivalent to minimizing hinge loss:

min_w (1/2) ||w||^2 + C sum_i max(0, 1 - y_i f(x_i))

1.4.3 Kernel Methods

Replace dot products with kernel function k(x, x'):

K(x, x') = phi(x).phi(x')

where phi maps to (possibly infinite-dimensional) feature space.

Mercer's Theorem: k is a valid kernel iff for any set of points, the Gram matrix K_ij = k(x_i, x_j) is positive semi-definite.

Common Kernels: - Linear: k(x, x') = x.x' - Polynomial: k(x, x') = (gamma * x.x' + r)^d - RBF: k(x, x') = exp(-gamma ||x - x'||^2) - Sigmoid: k(x, x') = tanh(gamma * x.x' + r)

RBF with gamma -> inf overfits (each point becomes its own support vector). Small gamma underfits (decision boundary too smooth).

Dual with Kernels: max_alpha sum_i alpha_i - (1/2) sum_{i,j} alpha_i alpha_j y_i y_j K(x_i, x_j)

Prediction: f(x) = sum_i alpha_i y_i K(x_i, x) + b

1.5 Implementation

1.5.1 Setup

```
import numpy as np
import pandas as pd
from sklearn.model_selection import (
    train_test_split, cross_val_score, StratifiedKFold, GridSearchCV
```

```
)
from sklearn.ensemble import (
    RandomForestClassifier, GradientBoostingClassifier,
    AdaBoostClassifier, VotingClassifier, StackingClassifier
)
from sklearn.tree import DecisionTreeClassifier
from sklearn.svm import SVC
from sklearn.linear_model import LogisticRegression
from sklearn.preprocessing import StandardScaler
from sklearn.metrics import (
    accuracy_score, classification_report, roc_auc_score,
    brier_score_loss, log_loss
)
from sklearn.calibration import CalibratedClassifierCV, calibration_curve
import matplotlib.pyplot as plt
import warnings
warnings.filterwarnings('ignore')
```

1.5.2 Custom Gradient Boosting Classifier

```
class GradientBoostingFromScratch:
    """Gradient boosting for binary classification from scratch."""

    def __init__(self, n_estimators=100, learning_rate=0.1, max_depth=3):
        self.n_estimators = n_estimators
        self.learning_rate = learning_rate
        self.max_depth = max_depth
        self.trees = []
        self.initial_prediction = None

    def _sigmoid(self, x):
        return 1 / (1 + np.exp(-np.clip(x, -500, 500)))

    def _log_loss_gradient(self, y, p):
        """Gradient of log loss: y - p."""
        return y - p

    def fit(self, X, y):
        """Fit gradient boosting classifier."""
        n_samples = X.shape[0]

        # Initialize with log-odds
        p = y.mean()
        self.initial_prediction = np.log(p / (1 - p))
        F = np.full(n_samples, self.initial_prediction)

        for m in range(self.n_estimators):
            # Compute probabilities
            p = self._sigmoid(F)

            # Compute pseudo-residuals (gradient)
            residuals = self._log_loss_gradient(y, p)

            # Fit tree to residuals
            tree = DecisionTreeClassifier(max_depth=self.max_depth)
            tree.fit(X, (residuals > 0).astype(int))

            # Get leaf assignments
            leaf_ids = tree.apply(X)

            # Compute optimal leaf values (Newton step)
```

```
            leaf_values = {}
            for leaf_id in np.unique(leaf_ids):
                mask = leaf_ids == leaf_id
                # Optimal value for log loss
                numerator = residuals[mask].sum()
                denominator = (p[mask] * (1 - p[mask])).sum()
                if denominator > 0:
                    leaf_values[leaf_id] = numerator / denominator
                else:
                    leaf_values[leaf_id] = 0

            # Store tree and leaf values
            self.trees.append((tree, leaf_values))

            # Update predictions
            updates = np.array([leaf_values[lid] for lid in leaf_ids])
            F += self.learning_rate * updates

        return self

    def predict_proba(self, X):
        """Predict class probabilities."""
        F = np.full(X.shape[0], self.initial_prediction)

        for tree, leaf_values in self.trees:
            leaf_ids = tree.apply(X)
            updates = np.array([leaf_values.get(lid, 0) for lid in leaf_ids])
            F += self.learning_rate * updates

        p = self._sigmoid(F)
        return np.column_stack([1 - p, p])

    def predict(self, X):
        """Predict class labels."""
        return (self.predict_proba(X)[:, 1] > 0.5).astype(int)
```

1.5.3 Production Random Forest with Calibration

```
class CalibratedRandomForest:
    """Random Forest with probability calibration and monitoring."""

    def __init__(self, n_estimators=100, max_depth=None, calibration='isotonic
    '):
        self.n_estimators = n_estimators
        self.max_depth = max_depth
        self.calibration_method = calibration
        self.base_model = None
        self.calibrated_model = None
        self.feature_importances_ = None

    def fit(self, X, y, X_cal=None, y_cal=None):
        """Fit with optional separate calibration set."""
        if X_cal is None:
            # Split data for calibration
            X_train, X_cal, y_train, y_cal = train_test_split(
                X, y, test_size=0.2, stratify=y, random_state=42
            )
        else:
            X_train, y_train = X, y

        # Fit base model
```

```
        self.base_model = RandomForestClassifier(
            n_estimators=self.n_estimators,
            max_depth=self.max_depth,
            oob_score=True,
            random_state=42,
            n_jobs=-1
        )
        self.base_model.fit(X_train, y_train)

        # Calibrate probabilities
        self.calibrated_model = CalibratedClassifierCV(
            self.base_model, method=self.calibration_method, cv='prefit'
        )
        self.calibrated_model.fit(X_cal, y_cal)

        # Store feature importances
        self.feature_importances_ = self.base_model.feature_importances_

        return self

    def predict_proba(self, X):
        """Return calibrated probabilities."""
        return self.calibrated_model.predict_proba(X)

    def predict(self, X):
        """Return class predictions."""
        return self.calibrated_model.predict(X)

    def get_calibration_curve(self, X, y, n_bins=10):
        """Compute reliability diagram data."""
        probas = self.predict_proba(X)[:, 1]
        fraction_positives, mean_predicted = calibration_curve(
            y, probas, n_bins=n_bins, strategy='uniform'
        )
        return fraction_positives, mean_predicted

    def evaluate_calibration(self, X, y):
        """Comprehensive calibration evaluation."""
        probas = self.predict_proba(X)[:, 1]
        preds = self.predict(X)

        metrics = {
            'accuracy': accuracy_score(y, preds),
            'roc_auc': roc_auc_score(y, probas),
            'brier_score': brier_score_loss(y, probas),
            'log_loss': log_loss(y, probas),
            'oob_score': self.base_model.oob_score_
        }

        # Expected Calibration Error
        fraction_pos, mean_pred = self.get_calibration_curve(X, y)
        ece = np.mean(np.abs(fraction_pos - mean_pred))
        metrics['ece'] = ece

        return metrics
```

1.5.4 Advanced Stacking Ensemble

```
class ProductionStackingClassifier:
    """Production-ready stacking with cross-validation meta-features."""
```

```
    def __init__(self, base_models, meta_model, cv=5, use_features=False):
        self.base_models = base_models
        self.meta_model = meta_model
        self.cv = cv
        self.use_features = use_features
        self.fitted_base_models_ = []

    def _create_meta_features(self, X, y=None, training=True):
        """Generate meta-features using cross-validation or fitted models."""
        n_samples = X.shape[0]
        n_models = len(self.base_models)
        n_classes = len(np.unique(y)) if y is not None else 2

        meta_features = np.zeros((n_samples, n_models * n_classes))

        if training:
            skf = StratifiedKFold(n_splits=self.cv, shuffle=True, random_state
=42)

            for fold_idx, (train_idx, val_idx) in enumerate(skf.split(X, y)):
                X_train_fold, X_val_fold = X[train_idx], X[val_idx]
                y_train_fold = y[train_idx]

                for model_idx, (name, model) in enumerate(self.base_models):
                    model_clone = clone(model)
                    model_clone.fit(X_train_fold, y_train_fold)

                    probas = model_clone.predict_proba(X_val_fold)
                    start_col = model_idx * n_classes
                    end_col = start_col + n_classes
                    meta_features[val_idx, start_col:end_col] = probas

        else:
            for model_idx, (name, model) in enumerate(self.fitted_base_models_
):
                probas = model.predict_proba(X)
                start_col = model_idx * n_classes
                end_col = start_col + n_classes
                meta_features[:, start_col:end_col] = probas

        return meta_features

    def fit(self, X, y):
        """Fit stacking classifier."""
        from sklearn.base import clone

        # Create meta-features via CV
        meta_features = self._create_meta_features(X, y, training=True)

        # Fit base models on full data
        self.fitted_base_models_ = []
        for name, model in self.base_models:
            model_clone = clone(model)
            model_clone.fit(X, y)
            self.fitted_base_models_.append((name, model_clone))

        # Prepare meta input
        if self.use_features:
            meta_X = np.hstack([meta_features, X])
        else:
            meta_X = meta_features

        # Fit meta model
```

```
        self.meta_model.fit(meta_X, y)

        return self

    def predict_proba(self, X):
        """Predict class probabilities."""
        meta_features = self._create_meta_features(X, training=False)

        if self.use_features:
            meta_X = np.hstack([meta_features, X])
        else:
            meta_X = meta_features

        return self.meta_model.predict_proba(meta_X)

    def predict(self, X):
        """Predict class labels."""
        return np.argmax(self.predict_proba(X), axis=1)


from sklearn.base import clone
```

1.5.5 Hyperparameter Optimization

```
class ClassifierOptimizer:
    """Bayesian optimization for classifier hyperparameters."""

    def __init__(self, model_type='rf', scoring='roc_auc', cv=5):
        self.model_type = model_type
        self.scoring = scoring
        self.cv = cv
        self.best_model = None
        self.best_params = None
        self.best_score = None

    def _get_param_grid(self):
        """Define parameter grids for different model types."""
        grids = {
            'rf': {
                'n_estimators': [50, 100, 200, 300],
                'max_depth': [5, 10, 15, 20, None],
                'min_samples_split': [2, 5, 10],
                'min_samples_leaf': [1, 2, 4],
                'max_features': ['sqrt', 'log2', 0.5]
            },
            'gb': {
                'n_estimators': [50, 100, 200],
                'learning_rate': [0.01, 0.05, 0.1, 0.2],
                'max_depth': [3, 5, 7],
                'subsample': [0.7, 0.8, 0.9, 1.0],
                'min_samples_split': [2, 5, 10]
            },
            'svm': {
                'C': [0.1, 1, 10, 100],
                'gamma': ['scale', 'auto', 0.01, 0.1],
                'kernel': ['rbf', 'poly'],
                'degree': [2, 3, 4]
            }
        }
        return grids.get(self.model_type, {})
```

```
    def _get_base_model(self):
        """Get base model for model type."""
        models = {
            'rf': RandomForestClassifier(random_state=42, n_jobs=-1),
            'gb': GradientBoostingClassifier(random_state=42),
            'svm': SVC(probability=True, random_state=42)
        }
        return models.get(self.model_type)

    def optimize(self, X, y):
        """Run hyperparameter optimization."""
        base_model = self._get_base_model()
        param_grid = self._get_param_grid()

        grid_search = GridSearchCV(
            base_model, param_grid, scoring=self.scoring,
            cv=self.cv, n_jobs=-1, verbose=1
        )
        grid_search.fit(X, y)

        self.best_model = grid_search.best_estimator_
        self.best_params = grid_search.best_params_
        self.best_score = grid_search.best_score_

        return self

    def get_results(self):
        """Return optimization results."""
        return {
            'best_params': self.best_params,
            'best_score': self.best_score,
            'model_type': self.model_type
        }
```

1.6 Calibration Theory

1.6.1 Probability Calibration

A classifier is well-calibrated if predicted probabilities match empirical frequencies:

P(Y=1 | p(X) = p) = p

Reliability Diagram: Plot empirical frequency vs predicted probability in bins. Perfect calibration lies on diagonal.

Expected Calibration Error (ECE):

ECE = sum_{m=1}^{M} (|B_m| / n) |acc(B_m) - conf(B_m)|

where B_m is bin m, acc is accuracy, and conf is average confidence.

1.6.2 Calibration Methods

Platt Scaling: Fit logistic regression on classifier outputs:

P(y=1|f) = 1 / (1 + exp(A*f + B))

Learn A, B on held-out data. Works well for SVMs.

Isotonic Regression: Non-parametric, monotonic fit. More flexible but needs more data.

Temperature Scaling: For neural networks, divide logits by temperature T:

p_i = exp(z_i/T) / sum_j exp(z_j/T)

Optimize T on validation set to minimize NLL.

1.7 Multi-Class Extensions

1.7.1 One-vs-Rest (OvR)

Train K binary classifiers, each separating class k from all others. Predict class with highest score:

y_pred = argmax_k f_k(x)

Calibration issues: scores not directly comparable across classifiers.

1.7.2 One-vs-One (OvO)

Train K(K-1)/2 binary classifiers, one per class pair. Predict by majority voting. More classifiers but each trained on smaller, balanced subset.

1.7.3 Multinomial Extension

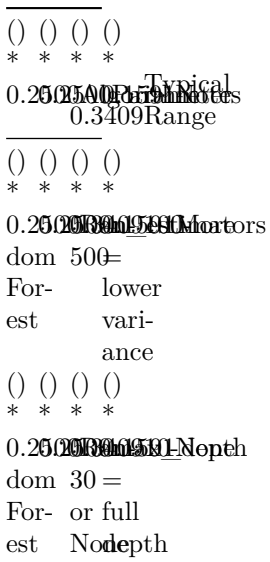
Logistic regression extends naturally:

P(Y=k|x) = exp(w_k.x) / sum_j exp(w_j.x)

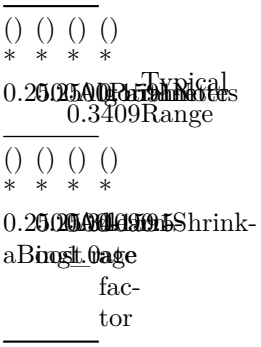
Loss: negative log-likelihood (cross-entropy).

For trees/ensembles, leaf predictions store class probability vectors. No architectural changes needed.

1.8 Common Parameters



() *	() *	() *	() *
0.25	5	0.1	Typical 0.3409 Range
() *	() *	() *	() *
0.25	5	0.1	2 (all)
dom For-est	log2	cor-re-la-tion	
() *	() *	() *	() *
0.25	5	0.1	Lower
di-ing-ent Boost-ing	0.2	more trees needed	
() *	() *	() *	() *
0.25	5	0.1	ators
di-ent Boost-ing	500	early stop-ping	
() *	() *	() *	() *
0.25	5	0.1	Sleph
di-ent Boost-ing	8	low trees pre-ferred	
() *	() *	() *	() *
0.25	5	0.1	In-
	100	verse reg-u-lar-iza-tion	
() *	() *	() *	() *
0.25	5	0.1	er-
(RB			
	1	band-width	
() *	() *	() *	() *
0.25	5	0.1	ators
aBoos	200	risk of over-fit-ting	



1.9 Practice Problems

1. **Margin Analysis:** Implement margin computation for AdaBoost. Plot margin distribution over boosting iterations. Verify that margins increase and relate to test error.
2. **Kernel Comparison:** Train SVMs with linear, polynomial (d=2,3), and RBF kernels on the same dataset. Visualize decision boundaries and compare generalization.
3. **Ensemble Diversity:** Create 10 Random Forest models with different random seeds. Measure pairwise correlation of predictions. Relate diversity to ensemble improvement.
4. **Calibration Study:** Compare uncalibrated Random Forest against Platt-scaled and isotonic-calibrated versions. Plot reliability diagrams and compute ECE for each.
5. **Feature Importance Stability:** Train 50 Random Forest models and record feature importances. Compute confidence intervals for each feature's importance rank.

1.10 References

1. Breiman, L. (2001). "Random Forests." Machine Learning, 45(1), 5-32.
2. Friedman, J. H. (2001). "Greedy Function Approximation: A Gradient Boosting Machine." Annals of Statistics, 29(5), 1189-1232.
3. Freund, Y., & Schapire, R. E. (1997). "A Decision-Theoretic Generalization of On-Line Learning and an Application to Boosting." Journal of Computer and System Sciences, 55(1), 119-139.
4. Cortes, C., & Vapnik, V. (1995). "Support-Vector Networks." Machine Learning, 20(3), 273-297.
5. Platt, J. (1999). "Probabilistic Outputs for Support Vector Machines." Advances in Large Margin Classifiers, MIT Press.
6. Niculescu-Mizil, A., & Caruana, R. (2005). "Predicting Good Probabilities with Supervised Learning." ICML.
7. Chen, T., & Guestrin, C. (2016). "XGBoost: A Scalable Tree Boosting System." KDD.

Classification theory provides the foundation for understanding when and why algorithms work. Mathematical insight into margins, regularization, and calibration enables principled model selection and reliable probability estimates for decision-making.