Model Evaluation and Selection

Hyperparameter Tuning: Grid Search, Random Search, and Validation Curves

Introduction to Machine Learning

Department of Computer Science
Georgia State University

Optimizing Model Performance

Today's Learning Journey

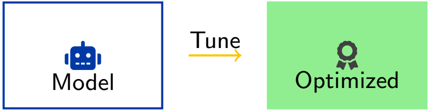
Why Model Evaluation Matters

Key Questions:

- How well does our model perform?
- Which model architecture is best?
- What parameter settings optimize performance?
- How do we avoid overfitting?

The Challenge:

- Models have hyperparameters
- Need systematic approach to find optimal values
- Balance between bias and variance



Parameters vs. Hyperparameters

Parameters

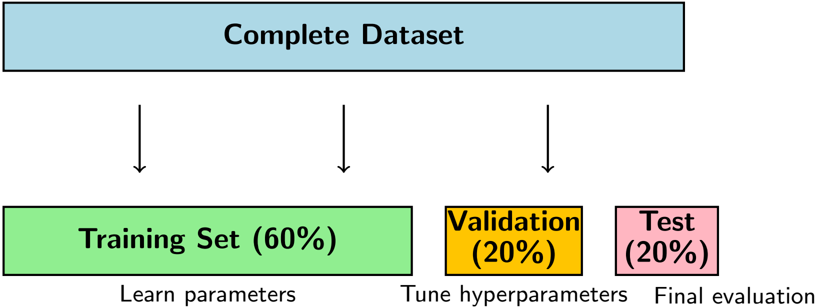
- Learned from training data
- Updated during training
- Examples:
  - Weights in neural networks
  - Coefficients in linear regression
  - Split thresholds in decision trees

Hyperparameters

- Set before training begins
- Control the learning process
- Examples:
  - Learning rate
  - Number of trees in random forest
  - Regularization strength
  - Kernel type in SVM

Hyperparameter tuning finds optimal configuration for best performance

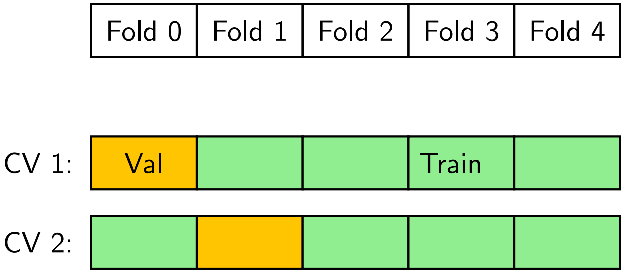
Train-Validation-Test Split Strategy



Process:

1. Train model on training set with different hyperparameters
2. Evaluate each configuration on validation set
3. Select best hyperparameters based on validation performance
4. Final assessment on test set (use only once!)

Cross-Validation for Hyperparameter Tuning



Average validation scores across all folds

Benefits:

- More robust estimate of model performance
- Better use of available data
- Reduces impact of particular train/validation split

Grid Search: Exhaustive Parameter Exploration

Concept: Systematically test all combinations of hyperparameter values

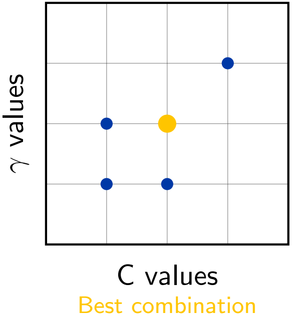
Example: SVM Hyperparameters

- C (regularization): [0.1, 1, 10, 100]
- γ (kernel): [0.001, 0.01, 0.1, 1]
- Total combinations: $4 \times 4 = 16$

Algorithm:

1. Define parameter grid
2. For each combination:
    - Train model with parameters
    - Evaluate using cross-validation
    - Record performance
3. Select best performing combination

Parameter Grid



Grid Search Implementation

```python
from sklearn.model_selection import GridSearchCV
from sklearn.svm import SVC
from sklearn.datasets import load_iris

# Load data
X, y = load_iris(return_X_y=True)

# Define parameter grid
param_grid = {
    'C': [0.1, 1, 10, 100],
    'gamma': [0.001, 0.01, 0.1, 1],
    'kernel': ['rbf', 'poly']
}

# Create grid search object
grid_search = GridSearchCV(
    estimator=SVC(),
    param_grid=param_grid,
    cv=5,                    # 5-fold cross-validation
    scoring='accuracy',      # evaluation metric
    n_jobs=-1               # parallel processing
)

# Fit grid search
grid_search.fit(X, y)

# Results
print(f"Best parameters: {grid_search.best_params_}")
print(f"Best CV score: {grid_search.best_score_:.3f}")
```

Grid Search: Advantages and Limitations

Advantages

- **Comprehensive:** Tests all combinations
- **Deterministic:** Reproducible results
- **Simple:** Easy to understand and implement
- **Parallel:** Can distribute computation
- **Guaranteed:** Finds global optimum in grid

Limitations

- **Computational cost:** Exponential growth
- **Curse of dimensionality:** Many parameters
- **Grid dependency:** May miss optimal values
- **Inefficient:** Equal time to all combinations
- **Memory intensive:** Stores all results

Best for: Few parameters, small search spaces, when thoroughness is critical

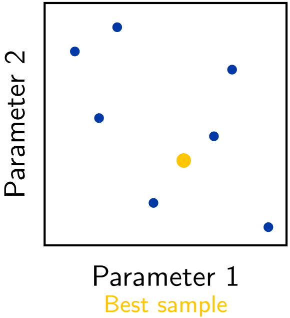
Random Search: Probabilistic Exploration

Key Insight: Not all hyperparameters are equally important

Algorithm:

1. Define parameter distributions
2. Randomly sample parameter combinations
3. Evaluate each sample using CV
4. Continue for fixed number of iterations
5. Select best performing combination

Random Sampling

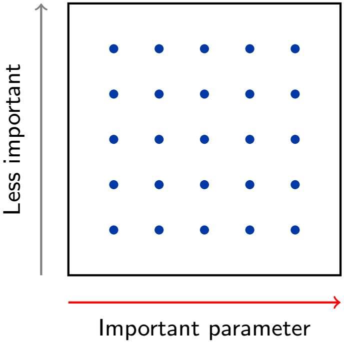


Theoretical Advantage:

- Higher probability of finding good values for important parameters
- More efficient exploration of parameter space

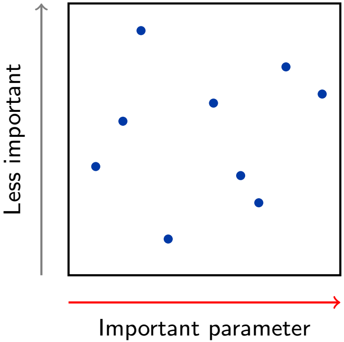
Why Random Search Often Outperforms Grid Search

Grid Search



9 unique values

Random Search



9 unique values

Key Insight: Random search explores more unique values along important dimensions

Random Search Implementation

```python
from sklearn.model_selection import RandomizedSearchCV
from scipy.stats import uniform, randint
import numpy as np
# Define parameter distributions
param_distributions = {
    'C': uniform(0.1, 100),              # Continuous uniform
    'gamma': uniform(0.001, 1),          # Continuous uniform
    'kernel': ['rbf', 'poly', 'sigmoid'] # Discrete choice
}
# Alternative: using specific distributions
param_distributions_alt = {
    'C': [0.1, 1, 10, 100, 1000],
    'gamma': np.logspace(-4, 0, 50),     # Log-uniform distribution
    'kernel': ['rbf', 'poly']
}
# Create random search object
random_search = RandomizedSearchCV(
    estimator=SVC(),
    param_distributions=param_distributions,
    n_iter=50,              # number of random samples
    cv=5,
    scoring='accuracy',
    random_state=42,        # for reproducibility
    n_jobs=-1)
# Fit random search
random_search.fit(X, y)
print(f"Best parameters: {random_search.best_params_}")
print(f"Best CV score: {random_search.best_score_:.3f}")
```

Random Search: Advantages and Best Practices

Advantages

- **Efficient:** Better parameter space coverage
- **Scalable:** Linear in number of trials
- **Flexible:** Works with continuous distributions
- **Anytime:** Can stop early if needed
- **Robust:** Less sensitive to irrelevant parameters

Best Practices

- Use appropriate distributions (log-uniform for learning rates)
- Set reasonable bounds
- Start with more iterations than grid points
- Monitor convergence
- Use cross-validation consistently

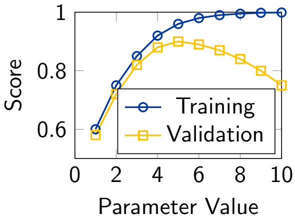
Rule of Thumb: Use random search when you have > 4 hyperparameters or large search spaces

Understanding Model Behavior with Validation Curves

Purpose: Visualize how a single hyperparameter affects model performance

What Validation Curves Show:

- Training vs. validation performance
- Underfitting vs. overfitting regions
- Optimal parameter range
- Bias-variance tradeoff



Key Patterns:

- Underfitting: Both curves low, gap small
- Good fit: High validation, small gap
- Overfitting: Large gap between curves

Creating Validation Curves

```
from sklearn.model_selection import validation_curve
from sklearn.ensemble import RandomForestClassifier
import matplotlib.pyplot as plt
import numpy as np
# Parameter range to test
param_range = [1, 5, 10, 20, 50, 100, 200]
# Generate validation curve
train_scores, val_scores = validation_curve(
    RandomForestClassifier(random_state=42),
    X, y,
    param_name='n_estimators',    # parameter to vary
    param_range=param_range,
    cv=5,                         # cross-validation folds
    scoring='accuracy',
    n_jobs=-1)
# Calculate means and standard deviations
train_mean = np.mean(train_scores, axis=1)
train_std = np.std(train_scores, axis=1)
val_mean = np.mean(val_scores, axis=1)
val_std = np.std(val_scores, axis=1)
# Plot validation curve
plt.figure(figsize=(10, 6))
plt.plot(param_range, train_mean, 'o-', color='blue', label='Training')
plt.fill_between(param_range, train_mean - train_std, train_mean + train_std, alpha=0.1, color='blue')
plt.plot(param_range, val_mean, 'o-', color='red', label='Validation')
plt.fill_between(param_range, val_mean - val_std, val_mean + val_std, alpha=0.1, color='red')
plt.xlabel('Number of Estimators')
plt.ylabel('Accuracy')
plt.legend()
plt.title('Validation Curve: Random Forest')
```

Interpreting Validation Curves: Common Patterns

Interpretation Guide:

- **Pattern 1:** Increase complexity (more estimators, deeper trees)
- **Pattern 2:** Reduce complexity or add regularization
- **Pattern 3:** Sweet spot in middle (optimal regularization)

Learning Curves vs. Validation Curves

Learning Curves

- Plot performance vs. training set size
- Fixed hyperparameters
- Shows if more data helps
- Diagnose high bias vs. high variance

Use when:

- Deciding if more data needed
- Model selection between algorithms
- Understanding data requirements

Validation Curves

- Plot performance vs. hyperparameter
- Fixed training set size
- Shows optimal parameter values
- Diagnose under/overfitting for parameters

Use when:

- Tuning specific hyperparameters
- Understanding parameter effects
- Visualizing bias-variance tradeoff

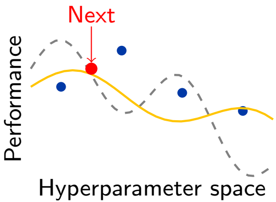
Both are essential tools for model diagnostics and optimization

Bayesian Optimization: Smart Search Strategy

Key Idea: Use probabilistic model to guide hyperparameter search

How it works:

1. Build probabilistic model of objective function
2. Use acquisition function to select next point
3. Evaluate objective at selected point
4. Update model with new observation
5. Repeat until budget exhausted



Popular Libraries:

- `scikit-optimize` (skopt)
- `hyperopt`
- `optuna`
- `ax-platform` (Facebook)

Bayesian Optimization Example

```python
from skopt import gp_minimize
from skopt.space import Real, Integer, Categorical
from skopt.utils import use_named_args
from sklearn.ensemble import RandomForestClassifier
from sklearn.model_selection import cross_val_score
# Define search space
dimensions = [Integer(10, 200, name='n_estimators'),
    Integer(1, 20, name='max_depth'),
    Real(0.1, 1.0, name='max_features'),
    Categorical(['gini', 'entropy'], name='criterion')]
# Objective function to minimize (we minimize, so return -accuracy)
@use_named_args(dimensions)
def objective(n_estimators, max_depth, max_features, criterion):
    model = RandomForestClassifier(
        n_estimators=n_estimators,
        max_depth=max_depth,
        max_features=max_features,
        criterion=criterion,
        random_state=42)
    # Return negative accuracy (since we minimize)
    scores = cross_val_score(model, X, y, cv=5, scoring='accuracy')
    return -np.mean(scores)
# Run Bayesian optimization
result = gp_minimize(
    func=objective,
    dimensions=dimensions,
    n_calls=50,            # number of evaluations
    random_state=42)
print(f"Best parameters: {result.x}")
print(f"Best accuracy: {-result.fun:.3f}")
```

Multi-Objective Optimization

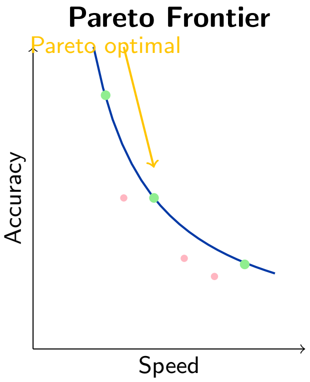
Challenge: Often need to optimize multiple objectives simultaneously

Common Trade-offs:

- Accuracy vs. Speed
- Accuracy vs. Memory usage
- Accuracy vs. Interpretability
- Precision vs. Recall
- Performance vs. Fairness

Approaches:

- **Weighted sum:** Combine objectives with weights
- **Pareto optimization:** Find Pareto-optimal solutions
- **Constraint optimization:** Optimize one subject to constraints
- **Sequential:** Optimize primary, then secondary



Hyperparameter Tuning Best Practices

Search Strategy

- Start with random search for exploration
- Use grid search for final refinement
- Consider Bayesian optimization for expensive models
- Always use cross-validation
- Set reasonable parameter bounds

Computational Efficiency

- Use parallel processing (`n_jobs=-1`)
- Start with smaller datasets for initial tuning
- Use early stopping when possible
- Cache expensive computations

Validation Strategy

- Never use test set for hyperparameter tuning
- Use stratified splits for imbalanced data
- Ensure consistent preprocessing in CV folds
- Use appropriate metrics for your problem
- Consider time-series specific validation

Common Pitfalls

- Data leakage in preprocessing
- Overfitting to validation set
- Ignoring computational constraints
- Not considering model interpretability

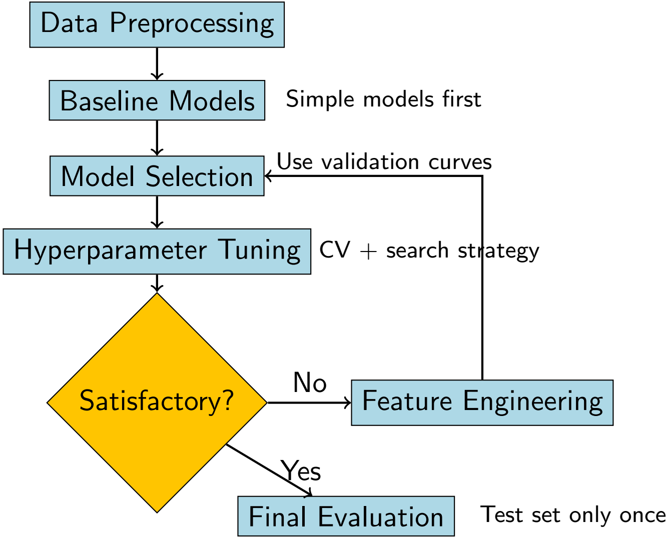
Parameter Search Guidelines by Model Type

Model	Key Parameters	Search Strategy	Considerations
SVM	C, gamma, kernel	Grid search on log scale	Expensive training, scale features
Random Forest	n_estimators, max_depth, max_features	Random search first, then grid	Fast training, can overfit
XGBoost	learning_rate, max_depth, n_estimators	Bayesian optimization	Many parameters, use early stopping
Neural Networks	learning_rate, batch_size, hidden_units	Random/Bayesian search	Expensive training, use validation curves
Logistic Regression	C, penalty, solver	Grid search	Fast training, simple search space

General Tips:

- Use log-uniform distributions for learning rates and regularization parameters
- Start with default values and expand search space gradually
- Consider domain-specific constraints (e.g., max_depth for interpretability)

Model Selection Workflow



Key Takeaways

Core Concepts

- Hyperparameters control learning process
- Validation strategy prevents overfitting
- Cross-validation provides robust estimates
- Visualization aids understanding

Search Strategies

- Grid search: thorough but expensive
- Random search: efficient exploration
- Bayesian optimization: intelligent search
- Validation curves: parameter effects

Best Practices

- Never tune on test set
- Use appropriate search spaces
- Consider computational budget
- Validate consistently
- Document your process

Remember

- Good hyperparameters $\neq$ good model
- Domain knowledge matters
- Interpretability vs. performance
- Generalization is the goal

Systematic hyperparameter tuning is key to achieving optimal model performance

Next Steps and Further Reading

Practical Next Steps:

- Implement grid and random search on your datasets
- Create validation curves for key hyperparameters
- Experiment with Bayesian optimization libraries
- Practice with different model types and search strategies

Advanced Topics to Explore:

- Multi-fidelity optimization (using subsets of data)
- Neural architecture search (NAS)
- Automated machine learning (AutoML)
- Population-based training
- Hyperparameter optimization for deep learning

Questions?

Thank you for your attention!

sali85@student.gsu.edu