Machine Learning Algorithms: Complete Guide

1. Linear Regression

Mathematical Foundation

Linear regression models the relationship between a dependent variable y and independent variables X using a linear equation:

Simple Linear Regression:

$$y = \beta_0 + \beta_1 x + \varepsilon$$

Multiple Linear Regression:

$$y = \beta_0 + \beta_1 x_1 + \beta_2 x_2 + \ldots + \beta_n x_n + \varepsilon$$

Where:

- y = dependent variable (target)
- β_0 = y-intercept (bias term)
- $\beta_1, \beta_2, \ldots, \beta_n$ = coefficients (weights)
- $x_1, x_2, \ldots, x_n$ = independent variables (features)
- ε = error term

Cost Function

The algorithm minimizes the Mean Squared Error (MSE):

$$MSE = (1/n) \sum_{i=1}^{n} (y_i - \hat{y}_i)^2$$

How It Works

1. **Initialize** coefficients randomly
2. **Calculate predictions** using current coefficients
3. **Compute cost** using MSE
4. **Update coefficients** using gradient descent:

   $$\beta_1 = \beta_1 - \alpha \times (\partial MSE / \partial \beta_1)$$

5. **Repeat** until convergence

Key Assumptions

- Linear relationship between variables
- Independence of residuals
- Homoscedasticity (constant variance)
- Normal distribution of residuals

Evaluation Metrics

- **R^2 Score**: Coefficient of determination (0-1, higher is better)
- **Mean Absolute Error (MAE)**: Average absolute difference
- **Root Mean Square Error (RMSE)**: Square root of MSE
- **Adjusted R^2**: R^2 adjusted for number of predictors

Advantages & Disadvantages

Pros:

- Simple and interpretable
- Fast training and prediction
- No hyperparameter tuning needed
- Works well with linear relationships

Cons:

- Assumes linear relationship
- Sensitive to outliers
- Requires feature scaling
- Poor performance with non-linear data

---

2. Logistic Regression

Mathematical Foundation

Logistic regression uses the sigmoid function to map any real number to a probability between 0 and 1:

Sigmoid Function:

```
σ(z) = 1 / (1 + e^(-z))
```

Linear Combination:

$$z = \beta_0 + \beta_1 x_1 + \beta_2 x_2 + \ldots + \beta_n x_n$$

Probability:

$$P(y=1|x) = \sigma(z) = 1 / (1 + e^{-(\beta_0 + \beta_1 x_1 + \ldots + \beta_n x_n)})$$

Cost Function

Uses Maximum Likelihood Estimation with log-likelihood:

$$\text{Cost} = -[y \log(p) + (1-y) \log(1-p)]$$

How It Works

1. **Calculate linear combination** $z = \beta_0 + \beta_1 x_1 + \ldots + \beta_n x_n$
2. **Apply sigmoid function** to get probabilities
3. **Make predictions** (threshold typically 0.5)
4. **Calculate cost** using log-likelihood
5. **Update weights** using gradient descent
6. **Repeat** until convergence

Types

- **Binary Classification**: Two classes (0 or 1)
- **Multinomial**: Multiple classes (one-vs-rest or softmax)
- **Ordinal**: Ordered categories

Evaluation Metrics

- **Accuracy**: Correct predictions / Total predictions
- **Precision**: TP / (TP + FP)
- **Recall**: TP / (TP + FN)
- **F1-Score**: 2 × (Precision × Recall) / (Precision + Recall)
- **AUC-ROC**: Area under ROC curve
- **Confusion Matrix**: Detailed breakdown of predictions

Advantages & Disadvantages

Pros:

- Probabilistic output

- No assumptions about distribution
- Less prone to overfitting
- Interpretable coefficients

Cons:

- Assumes linear relationship between features and log-odds
- Sensitive to outliers
- Requires large sample sizes
- Can struggle with complex relationships

---

3. Decision Tree

Mathematical Foundation

Decision trees use information theory concepts:

Entropy (measure of impurity):

$$H(S) = -\sum_{i=1}^{c} p_i \log_2(p_i)$$

Information Gain:

$$IG(S,A) = H(S) - \sum_{v \in Values(A)} (|S_v|/|S|) \times H(S_v)$$

Gini Impurity:

$$Gini(S) = 1 - \sum_{i=1}^{c} p_i^2$$

Where:

- S = dataset
- c = number of classes
- p_i = proportion of class i
- A = attribute/feature

How It Works

1. **Start with root node** containing all training data
2. **Calculate impurity** for current node
3. **For each feature**, calculate information gain or Gini gain

4. **Select best feature** that maximizes information gain
5. **Split data** based on selected feature
6. **Create child nodes** and repeat recursively
7. **Stop when** stopping criteria met (max depth, min samples, etc.)

Tree Construction Algorithm (ID3/C4.5/CART)

```
function BuildTree(dataset, features):
    if all examples have same class:
        return leaf node with that class
    if no features remaining:
        return leaf node with majority class

    best_feature = select_best_feature(dataset, features)
    tree = create_node(best_feature)

    for each value v of best_feature:
        subset = examples with best_feature = v
        subtree = BuildTree(subset, features - best_feature)
        add subtree to tree

    return tree
```

Pruning Techniques

- **Pre-pruning**: Stop early (max depth, min samples)
- **Post-pruning**: Build full tree, then remove branches
- **Cost Complexity Pruning**: Balance tree complexity and accuracy

Evaluation Metrics

- **Accuracy**: Overall correctness
- **Precision, Recall, F1**: Class-specific performance
- **Feature Importance**: How much each feature contributes
- **Tree Depth**: Complexity measure
- **Number of Leaves**: Model complexity

Advantages & Disadvantages

Pros:

- Easy to understand and visualize
- Requires little data preparation

- Handles both numerical and categorical data
- Can capture non-linear relationships
- Provides feature importance

Cons:

- Prone to overfitting
- Unstable (small data changes = different tree)
- Biased toward features with more levels
- Can create overly complex trees

---

4. Support Vector Machine (SVM)

Mathematical Foundation

SVM finds the optimal hyperplane that separates classes with maximum margin.

Linear SVM Objective:

$$\text{Minimize: } (1/2)||w||^2 + C \sum_{i=1}^{n} \xi_i$$

Subject to:

$$y_i(w \cdot x_i + b) \geq 1 - \xi_i, \ \xi_i \geq 0$$

Decision Function:

$$f(x) = \text{sign}(w \cdot x + b) = \text{sign}(\sum_{i=1}^{n} \alpha_i y_i K(x_i, x) + b)$$

Where:

- w = weight vector
- b = bias term
- C = regularization parameter
- ξ_i = slack variables
- α_i = Lagrange multipliers
- $K(x_i, x)$ = kernel function

Kernel Functions

Linear Kernel:

$$K(x_i, x_j) = x_i \cdot x_j$$

Polynomial Kernel:

$$K(x_i, x_j) = (\gamma x_i \cdot x_j + r)^d$$

RBF (Gaussian) Kernel:

$$K(x_i, x_j) = \exp(-\gamma ||x_i - x_j||^2)$$

Sigmoid Kernel:

$$K(x_i, x_j) = \tanh(\gamma x_i \cdot x_j + r)$$

How It Works

1. **Map data** to higher dimensional space (if using kernel)
2. **Find support vectors** (data points closest to decision boundary)
3. **Solve quadratic optimization** problem to find optimal hyperplane
4. **Construct decision function** using support vectors
5. **Classify new points** based on which side of hyperplane they fall

Key Concepts

- **Margin**: Distance between hyperplane and nearest points
- **Support Vectors**: Training points that define the margin
- **Kernel Trick**: Implicit mapping to higher dimensions
- **Soft Margin**: Allows some misclassification (C parameter)

Hyperparameters

- **C**: Regularization (low = wider margin, high = stricter)
- **γ (gamma)**: Kernel coefficient (low = far influence, high = close influence)
- **kernel**: Type of kernel function
- **degree**: Degree for polynomial kernel

Evaluation Metrics

- **Accuracy**: Overall performance
- **Precision, Recall, F1**: Per-class metrics

- **Support Vector Count**: Model complexity
- **Margin Width**: Generalization indicator

Advantages & Disadvantages

Pros:

- Effective in high dimensions
- Memory efficient (uses support vectors)
- Versatile (different kernels)
- Works well with small datasets

Cons:

- Slow on large datasets
- Sensitive to feature scaling
- No probabilistic output
- Difficult to interpret
- Choice of kernel and parameters crucial

---

5. Random Forest

Mathematical Foundation

Random Forest combines multiple decision trees using bagging and random feature selection:

Prediction (Classification):

$$\hat{y} = \text{mode}\{T_1(x), T_2(x), ..., T_n(x)\}$$

Prediction (Regression):

$$\hat{y} = (1/n) \Sigma_{i=1}^{n} T_i(x)$$

Out-of-Bag Error:

$$\text{OOB Error} = (1/n) \Sigma_{i=1}^{n} I(y_i \neq \hat{y}_i^{(OOB)})$$

Where:

- $T_i(x)$ = prediction from tree i
- n = number of trees

- $\hat{y}_i^{(OOB)}$ = prediction using only trees where x_i was out-of-bag

How It Works

1. **Bootstrap Sampling**: Create n bootstrap samples from training data
2. **Random Feature Selection**: At each split, select random subset of features
3. **Build Trees**: Train decision tree on each bootstrap sample
4. **Combine Predictions**:
   - Classification: Majority voting
   - Regression: Average predictions
5. **Calculate OOB Error**: Use out-of-bag samples for error estimation

Algorithm Steps

```
function RandomForest(dataset, n_trees, n_features):
    forest = []

    for i in range(n_trees):
        # Bootstrap sampling
        bootstrap_sample = sample_with_replacement(dataset)

        # Build tree with random feature selection
        tree = DecisionTree(bootstrap_sample, n_features)
        forest.append(tree)

    return forest

function Predict(forest, x):
    predictions = []
    for tree in forest:
        predictions.append(tree.predict(x))

    return majority_vote(predictions)  # or average for regression
```

Key Parameters

- **n_estimators**: Number of trees
- **max_features**: Number of features for each split
- **max_depth**: Maximum tree depth
- **min_samples_split**: Minimum samples to split
- **min_samples_leaf**: Minimum samples in leaf
- **bootstrap**: Whether to use bootstrap sampling

Feature Importance

Calculated based on how much each feature decreases impurity:

$$\text{Importance(feature)} = \Sigma(\text{trees}) \text{ (decrease in impurity)} / n\_trees$$

Evaluation Metrics

- **Accuracy/RMSE**: Overall performance
- **OOB Score**: Out-of-bag accuracy/error
- **Feature Importance**: Ranking of feature contributions
- **Precision, Recall, F1**: Classification metrics

Advantages & Disadvantages

Pros:

- Reduces overfitting compared to single trees
- Handles missing values
- Provides feature importance
- Works with both classification and regression
- Robust to outliers
- Requires minimal hyperparameter tuning

Cons:

- Less interpretable than single tree
- Can overfit with very noisy data
- Biased toward categorical variables with more categories
- Memory intensive
- Not optimal for linear relationships

---

6. Gradient Boosting

Mathematical Foundation

Gradient boosting builds models sequentially, where each model corrects errors of previous models:

Additive Model:

$$F(x) = \Sigma_{i=1}^{M} \gamma_i h_i(x)$$

Forward Stagewise Addition:

$$F_m(x) = F_{m-1}(x) + \gamma_m h_m(x)$$

Loss Function Minimization:

$$\gamma_m, h_m = \text{argmin} \sum_{i=1}^{n} L(y_i, F_{m-1}(x_i) + \gamma h(x_i))$$

Gradient Descent in Function Space:

$$r_{im} = -[\partial L(y_i, F(x_i))/\partial F(x_i)]_{F=F_{m-1}}$$

Where:

- $F(x)$ = final ensemble model
- $h_i(x)$ = individual weak learner
- γ_i = learning rate for model i
- L = loss function
- r_{im} = residuals (negative gradients)

Algorithm Steps

```
function GradientBoosting(dataset, n_estimators, learning_rate):
    # Initialize with constant prediction
    F₀(x) = argmin Σᵢ L(yᵢ, γ)

    for m in range(1, n_estimators + 1):
        # Calculate residuals (negative gradients)
        for i in range(n):
            rᵢₘ = -∂L(yᵢ, Fₘ₋₁(xᵢ))/∂Fₘ₋₁(xᵢ)

        # Fit weak learner to residuals
        hₘ = fit_weak_learner(X, residuals)

        # Find optimal step size
        γₘ = argmin Σᵢ L(yᵢ, Fₘ₋₁(xᵢ) + γhₘ(xᵢ))

        # Update model
        Fₘ(x) = Fₘ₋₁(x) + learning_rate × γₘ × hₘ(x)

    return Fₘ
```

Common Loss Functions

Regression:

- Squared Error: $L(y, F(x)) = (y - F(x))^2/2$
- Absolute Error: $L(y, F(x)) = |y - F(x)|$
- Huber Loss: Combination of squared and absolute error

Classification:

- Logistic Loss: $L(y, F(x)) = \log(1 + \exp(-yF(x)))$
- Exponential Loss: $L(y, F(x)) = \exp(-yF(x))$

Regularization Techniques

1. **Learning Rate (η)**: Controls contribution of each tree
2. **Tree Constraints**: Max depth, min samples per leaf
3. **Subsampling**: Use random subset of training data
4. **Feature Subsampling**: Random subset of features per tree
5. **Early Stopping**: Stop when validation error stops improving

Popular Implementations

- **XGBoost**: Extreme Gradient Boosting
- **LightGBM**: Light Gradient Boosting Machine
- **CatBoost**: Categorical Boosting
- **Scikit-learn**: GradientBoostingClassifier/Regressor

Key Hyperparameters

- **n_estimators**: Number of boosting stages
- **learning_rate**: Shrinks contribution of each tree
- **max_depth**: Maximum depth of individual trees
- **subsample**: Fraction of samples for each tree
- **min_samples_split**: Minimum samples to split node

Evaluation Metrics

- **Training vs Validation Error**: Monitor overfitting
- **Feature Importance**: Based on splits and gain
- **Learning Curves**: Performance over iterations
- **Standard classification/regression metrics**

Advantages & Disadvantages

Pros:

- High predictive accuracy
- Handles different data types well
- Provides feature importance
- Robust to outliers
- No need for data preprocessing

Cons:

- Prone to overfitting
- Computationally intensive
- Many hyperparameters to tune
- Sensitive to noisy data
- Sequential nature makes it hard to parallelize
- Less interpretable than simpler models

Comparison Summary

Algorithm	Type	Interpretability	Overfitting Risk	Performance	Training Speed
Linear Regression	Regression	High	Low	Good for linear	Fast
Logistic Regression	Classification	High	Low	Good for linear	Fast
Decision Tree	Both	High	High	Variable	Fast
SVM	Both	Low	Medium	Good	Slow
Random Forest	Both	Medium	Low	Good	Medium
Gradient Boosting	Both	Low	High	Excellent	Slow

When to Use Each Algorithm

- **Linear/Logistic Regression**: Simple baseline, interpretability needed, linear relationships
- **Decision Trees**: Need interpretability, mixed data types, non-linear relationships
- **SVM**: High-dimensional data, small datasets, need robust boundaries
- **Random Forest**: Good all-around performer, need feature importance, avoid overfitting
- **Gradient Boosting**: Maximum accuracy needed, have time for tuning, competition/production

General Tips for Implementation

1. **Start Simple**: Begin with linear models, then increase complexity

2. **Cross-Validation**: Always use cross-validation for model selection
3. **Feature Engineering**: Often more important than algorithm choice
4. **Ensemble Methods**: Combine multiple algorithms for better performance
5. **Hyperparameter Tuning**: Use grid search or random search
6. **Monitor Overfitting**: Use validation curves and learning curves
7. **Scale Features**: Important for distance-based algorithms (SVM, logistic regression)
8. **Handle Missing Data**: Important preprocessing step
9. **Domain Knowledge**: Incorporate business understanding into model selection