Azercell

Model Selection

Azercell DataMinds Bootcamp

DS4

Content

Introduction

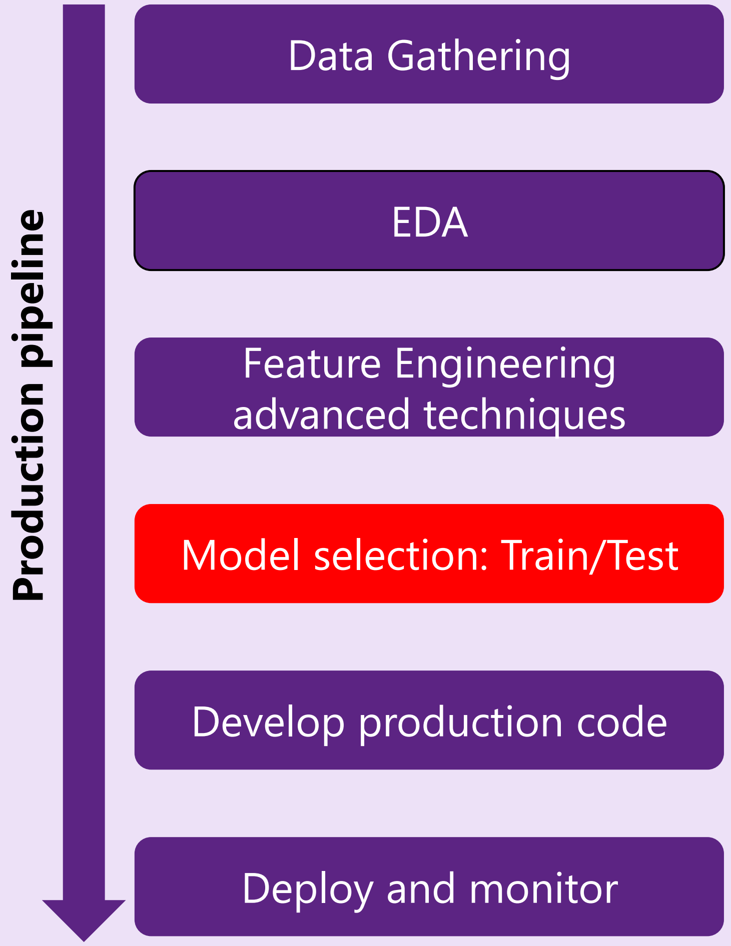
Model Selection Train/Test

Key important features

- Bias/Variance tradeoff
- Efficiency
- Evaluation and comparison

Model Model selection is a **critical step in machine learning** because not all models perform equally well on a given dataset.

Reason	Explanation
Better performance	Different models fit different data well
Generalization	Avoid underfitting and overfitting
Computational cost	Choose efficient models for production
Task-specific needs	Match model to data and problem type
Evaluation rigor	Use metrics and validation to choose reliably



OPTUNA

Azercell

Linear Models

Introduction to Linear and Logistic Regression

Overview
•Both are **supervised learning** algorithms.
•They differ in **output type** and **use case**.

Feature	Linear Regression	Logistic Regression
Target Variable	Continuous	Categorical (binary/multiclass)
Output	Any real number	Probability (0 to 1)
Used For	Regression tasks	Classification tasks

Common Use Cases:
•Linear Regression: Predicting housing prices, stock trends.
•Logistic Regression: Spam detection, disease prediction.

Azercell

Linear Regression – Formula & Intuition

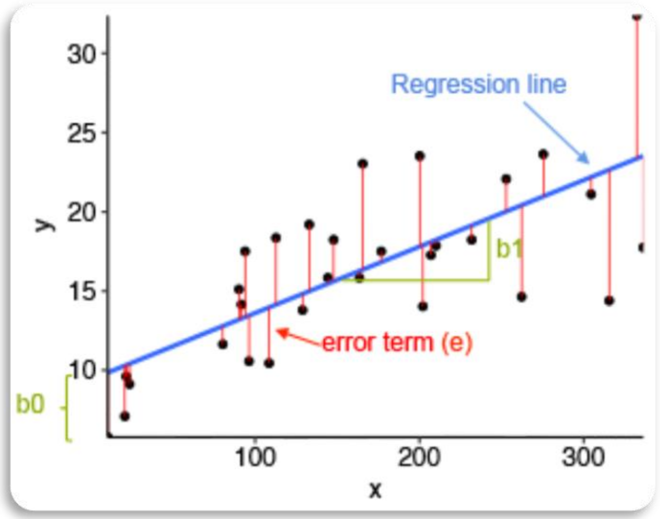
Model Equation:

$$\hat{y} = \beta_0 + \beta_1 x_1 + \beta_2 x_2 + \cdots + \beta_n x_n$$

- $\hat{y}$: Predicted value
- β_i: Coefficients/weights
- x_i: Feature variables

Loss Function (MSE):

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



Objective: Minimize MSE by adjusting weights using **Gradient Descent** or **Closed-form solution**.

Gradient Descent – Pseudocode.

```
Initialize weights $\beta_0, \beta_1, \ldots, \beta_n$ randomly
Repeat until convergence:
    Predict:  y_hat = X · β
    Compute gradients: ∇ = -2/n * Xᵗ · (y - y_hat)
    Update weights:    β = β - α * ∇
```

- α: Learning rate
- X: Input matrix
- y: Target vector
- Convergence = when loss stops decreasing significantly.

```python
from sklearn.linear_model import LinearRegression
from sklearn.datasets import make_regression
from sklearn.model_selection import train_test_split

# Generate sample data
X, y = make_regression(
  n_samples=100,
  n_features=1,
  noise=10)

X_train, X_test, y_train, y_test = train_test_split(
  X,
  y,
  test_size=0.2)

# Fit linear regression model
model = LinearRegression()
model.fit(X_train, y_train)

# Coefficients
print("Coefficient:", model.coef_)
print("Intercept:", model.intercept_)

# Predict
y_pred = model.predict(X_test)
```

Interpretation:
The coefficient tells how much y changes for one unit increase in X. The intercept is the expected y when all features are 0.

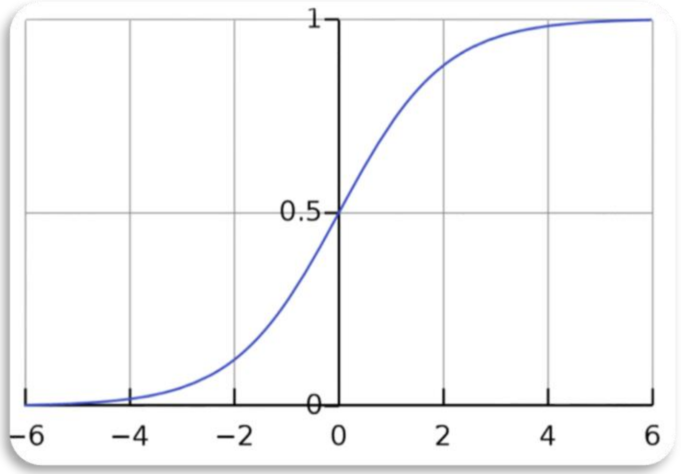
Logistic Regression – Formula & Intuition

Model Equation (Sigmoid Function):

$$\hat{p} = \sigma(z) = \frac{1}{1+e^{-z}}$$

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

- $\hat{p}$: Predicted probability of class 1
- $\sigma(z)$ Sigmoid function maps z to $[0,1]$



Decision Rule:

- If $p > 0.5$: predict class 1 (by default)
- Else: predict class 0

Loss Function (Binary Cross-Entropy):

$$\mathcal{L} = -\frac{1}{n}\sum_{i=1}^{n}[y_i \log(\widehat{p_i}) + (1-y_i)\log(1-\widehat{p_i})]$$

```
Initialize weights $\beta_0, \beta_1, \dots, \beta_n$ randomly
Repeat until convergence:
    Predict:          p_hat = sigmoid(X · β)
    Compute error: error = p_hat - y
    Compute gradients: ∇ = (1/n) * $X^t$ · error
    Update weights:     β = β - α * ∇
```

- Gradient is from the derivative of the log-loss with respect to weights.
- Learning rate α must be tuned to ensure convergence.

```
from sklearn.linear_model import LogisticRegression
from sklearn.datasets import make_classification
from sklearn.model_selection import train_test_split
from sklearn.metrics import classification_report

# Generate binary classification data
X, y = make_classification(
  n_samples=100,
  n_features=2,
  n_classes=2)

X_train, X_test, y_train, y_test = train_test_split(
  X,
  y,
  test_size=0.2)

# Fit logistic regression model
model = LogisticRegression()
model.fit(X_train, y_train)

# Predict
y_pred = model.predict(X_test)
print(classification_report(y_test, y_pred))
```

Interpretation:

- `model.predict_proba(X_test)` gives class probabilities.
- Useful for threshold tuning, ROC/AUC analysis.

Linear vs Logistic – Summary & Tips

Aspect	Linear Regression	Logistic Regression
Output	Continuous	Probabilistic (0–1)
Loss Function	MSE	Cross-Entropy
Output Function	Identity	Sigmoid
Gradient Formula	$\nabla = -2X^T(y - \hat{y})$	$\nabla = X^T(\sigma(z) - \hat{y})$

Best Practices:
•Normalize features (important for GD convergence).
•Monitor learning curves during training.
•Use `learning_rate, max_iter,` and `tol` to control GD in custom implementations.

Metrics

Classification Metrics Overview

Metrics

Goal: Evaluate how well the model distinguishes between classes (binary or multiclass).

Key Metrics:

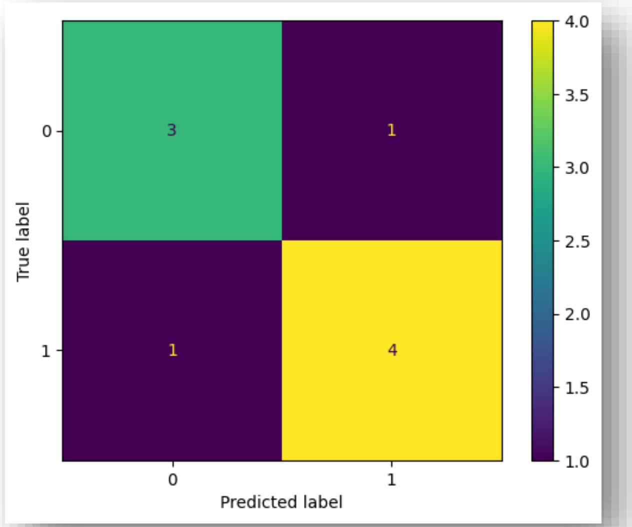
•**Precision** – How many selected items are relevant?
•**Recall** – How many relevant items are selected?
•**F1 Score** – Harmonic mean of precision and recall.
•**Confusion Matrix** – Summarizes predictions vs actuals.
•**ROC Curve & AUC** – Probability-based discrimination measure.

```
from sklearn.metrics import precision_score, recall_score, f1_score

y_true = [0, 1, 1, 1, 0, 1, 0, 0, 1]
y_pred = [0, 1, 0, 1, 0, 1, 1, 0, 1]

print("Precision:", precision_score(y_true, y_pred))
print("Recall:", recall_score(y_true, y_pred))
print("F1 Score:", f1_score(y_true, y_pred))
```

```
Precision: 0.8
Recall: 0.8
F1 Score: 0.8
```

TP (True Positive) – Correctly predicted positive.
FP (False Positive) – Incorrectly predicted positive.
TN (True Negative) – Correctly predicted negative.
FN (False Negative) – Incorrectly predicted negative.

```
from sklearn.metrics import (confusion_matrix,
  ConfusionMatrixDisplay)
import matplotlib.pyplot as plt

cm = confusion_matrix(y_true, y_pred)
disp = ConfusionMatrixDisplay(confusion_matrix=cm)
disp.plot()
plt.show()
```

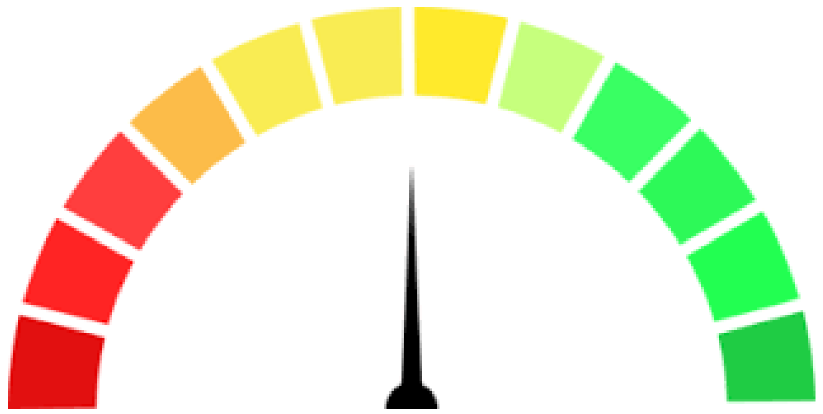


Classification Metrics definition

Metrics

Metric	Formula	Meaning
Accuracy	$\frac{TP + TN}{TP + TN + FP + FN}$	Overall correctness
Precision	$\frac{TP}{TP + FP}$	Correctly predicted positives
Recall	$\frac{TP}{TP + FN}$	Found all actual positives

TP (True Positive) – Correctly predicted positive.
FP (False Positive) – Incorrectly predicted positive.
TN (True Negative) – Correctly predicted negative.
FN (False Negative) – Incorrectly predicted negative.



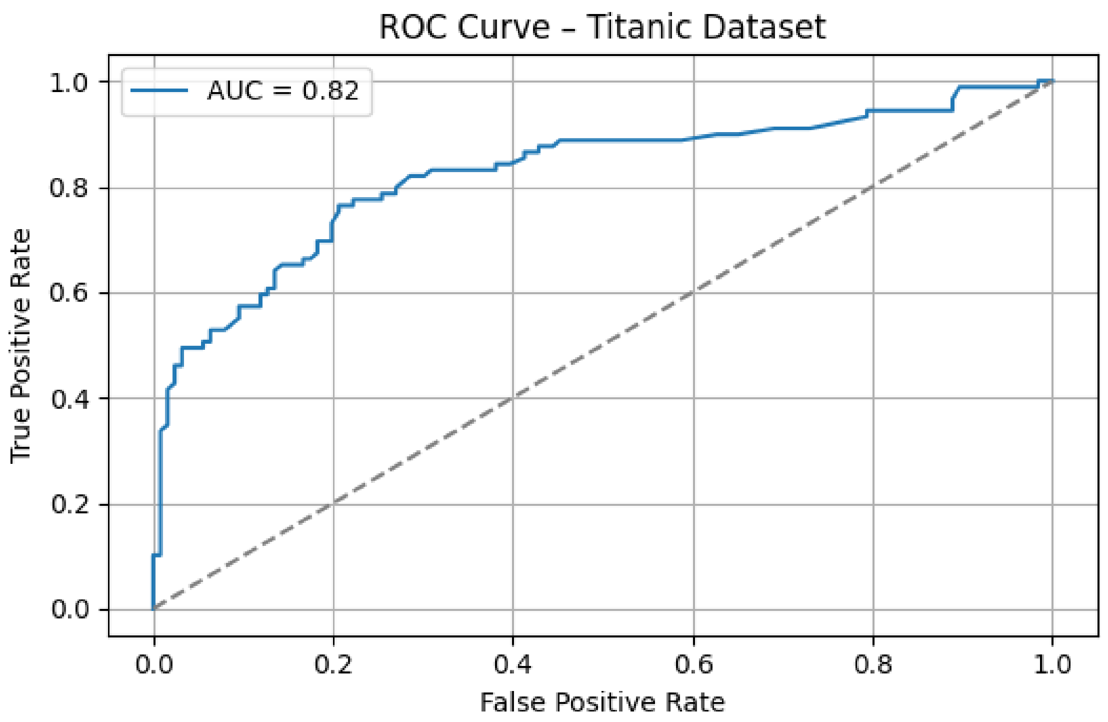
High Accuracy is misleading on imbalanced data.
Precision is crucial when **FP is costly** (e.g., spam filters).
Recall is vital when **FN is dangerous** (e.g., cancer screening).

ROC Curve & AUC — Threshold-Free Performance

Metrics

- **TPR (Recall):** $\frac{TP}{TP+F}$
- **FPR:** $\frac{FP}{FP+TN}$
- **ROC Curve:** *TPR* vs *FPR* at different thresholds
- **AUC (Area Under Curve):** Scalar summary of classifier's discrimination ability
  - **AUC = 0.5** → Random guessing
  - **AUC = 1.0** → Perfect classifier

Why Use ROC AUC?
•Unaffected by class imbalance
•Evaluates performance **across all thresholds**



```
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import roc_curve, roc_auc_score

# Logistic Regression
model = LogisticRegression()
model.fit(X_train, y_train)
# Predict probabilities
y_scores = model.predict_proba(X_test)[:, 1]

# ROC and AUC
fpr, tpr, _ = roc_curve(y_test, y_scores)
auc_score = roc_auc_score(y_test, y_scores)

plt.figure(figsize=(6, 4))
plt.plot(fpr, tpr, label=f"AUC = {auc_score:.2f}")
plt.plot([0, 1], [0, 1], linestyle="--", color="gray")
plt.xlabel("False Positive Rate")
plt.ylabel("True Positive Rate")
plt.title("ROC Curve – Titanic Dataset")
plt.legend()
plt.grid(True)
plt.tight_layout()
plt.show()
```

Regression Metrics

Metrics

```
from sklearn.metrics import (
  mean_absolute_error,
  mean_squared_error,
  r2_score)
```

Metric	Formula	Description & Use Case
MAE	$MAE = \frac{1}{n}\Sigma$	$y_i - \hat{y}$
MSE	$MSE = \frac{1}{n}\sum_{i=1}^{n}(y_t - \hat{y}_i)$	Penalizes large errors more than MAE. Use when large deviations are critical.
RMSE	$RMSE = \sqrt{MSE}$	Same units as the target variable. Useful for interpretability.
R² Score	$R^2 = 1 - \frac{\Sigma(y_i - \hat{y}_i)^2}{\Sigma(y_i - \hat{y}_i)}$	Proportion of variance in target explained by model. Use to assess goodness of fit. $R^2 = 1$: perfect fit, $R = 0$, $R^2 = 0$: no explanatory power.

MAE: Simple interpretation, less sensitive to outliers
MSE: Heavier penalty on large errors
RMSE: Useful for expressing error in original units
R²: Good for comparing model fits, but can be misleading with non-linear data

Regularization

Why do we need Regularization?

Regularization

Problem: Overfitting in Machine Learning

•**Overfitting**: The model performs well on training data but poorly on unseen data.

•Happens when:

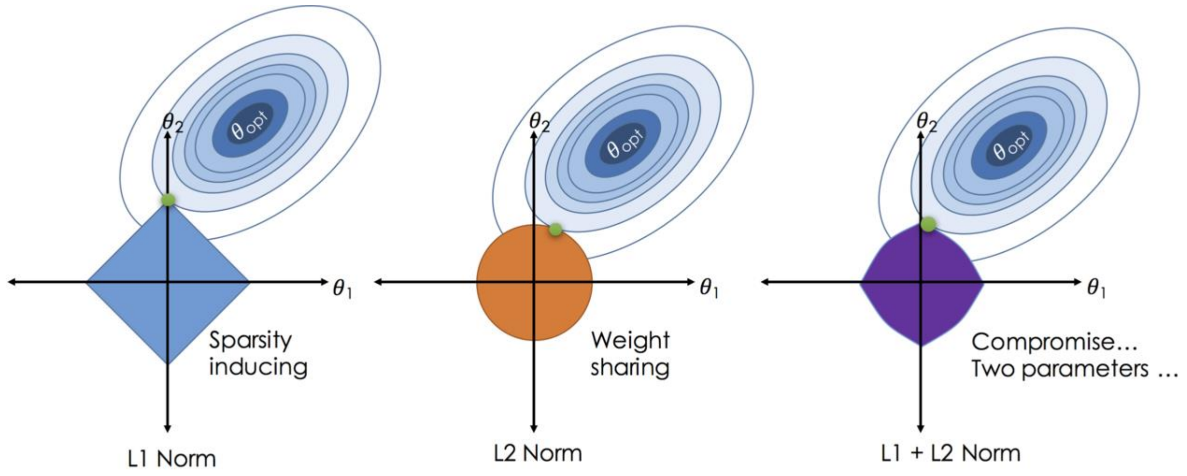
- Too complex model (e.g., high-degree polynomial)
- Too many parameters or not enough data

Solution: Regularization

- **Idea**: Penalize large model parameters to reduce complexity and prevent overfitting.

$$Cost\ Function\ (Basic) = Loss(y, \hat{y}) = \sum (y_i - \hat{y}_i)^2$$

•Regularization adds a **penalty** term to this cost function.



Types of Regularization (L1 & L2)

L2 Regularization (Ridge Regression)

$$J(\theta) = \sum (y_i - \hat{y}_i)^2 + \lambda \sum \left(\theta_j^2\right)$$

- Penalizes **squared** weights
- Shrinks weights smoothly toward zero

L1 Regularization (Lasso Regression)

$$J(\theta) = \sum (y_i - \hat{y}_i)^2 + \lambda \sum |\theta_j|$$

Can shrink some weights **exactly to zero** → feature selection

λ (lambda) controls penalty strength:

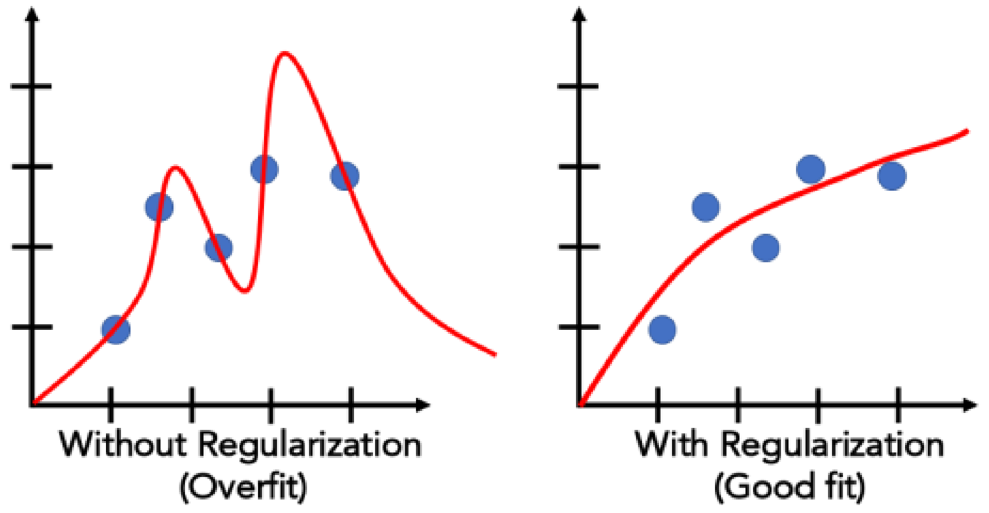
- λ = 0 → no regularization
- λ → ∞ → all weights shrink heavily

Regularization example

Regularization

```python
from sklearn.linear_model import Ridge, Lasso
from sklearn.datasets import make_regression
from sklearn.model_selection import train_test_split

X, y = make_regression(n_samples=100, n_features=20, noise=15)
X_train, X_test, y_train, y_test = train_test_split(X, y)

# Ridge (L2)
ridge = Ridge(alpha=1.0)
ridge.fit(X_train, y_train)
print("Ridge R^2:", ridge.score(X_test, y_test))

# Lasso (L1)
lasso = Lasso(alpha=0.1)
lasso.fit(X_train, y_train)
print("Lasso R^2:", lasso.score(X_test, y_test))
```

```
Ridge R^2: 0.9873922504549839
Lasso R^2: 0.9877459836451575
```



Model selection

Starting with a Baseline Model

Establishing the Baseline in Model Selection

Purpose of Baseline:

- A baseline model serves as a reference point to compare the performance of more complex models.
- It is simple, fast to train, and interpretable.

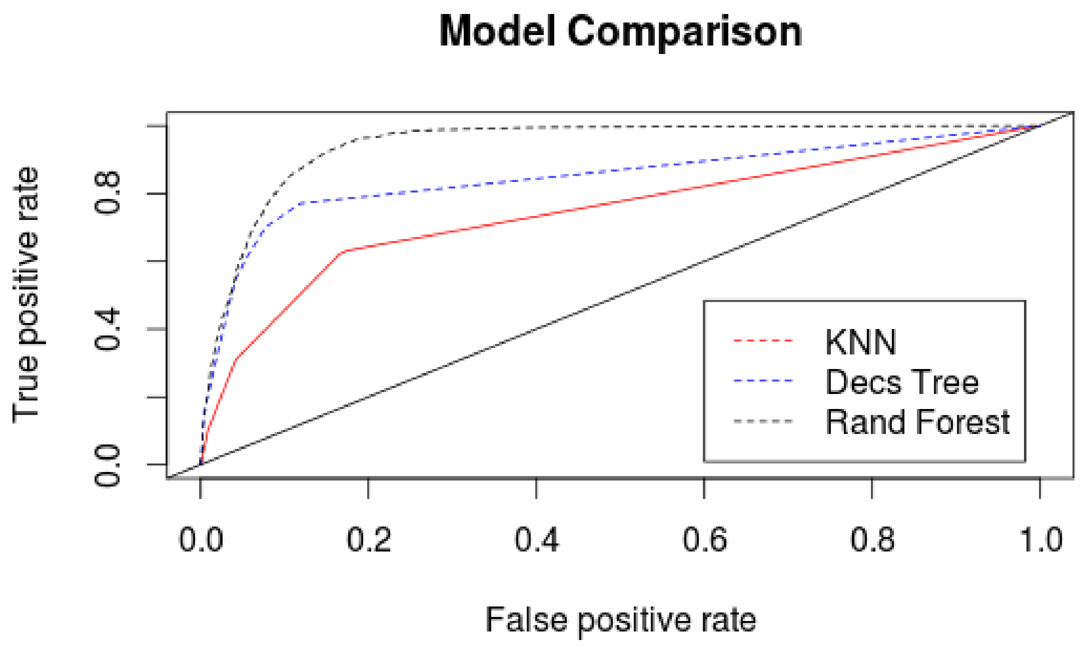
Examples of Baseline Models:

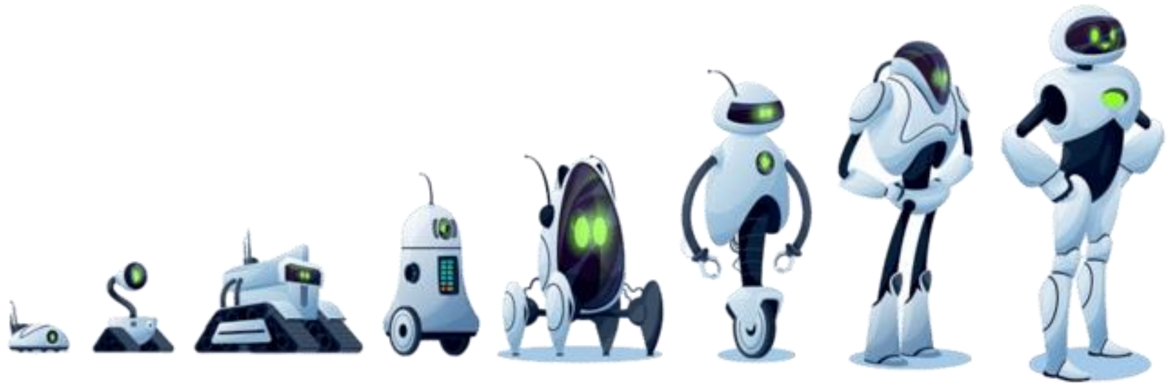
- **DummyClassifier** (e.g., always predicts the most frequent class).
- **Linear Regression** without feature scaling or regularization.
- **Mean or median predictor** for regression.

Why Start with a Baseline?

- **Sanity check:** Ensures your data pipeline is working correctly.
- **Benchmark:** Helps understand whether complex models actually add value.
- **Interpretability:** Provides initial insights into the task difficulty.

A good model should outperform the baseline significantly. If it doesn't, either the model is underfitting, or the data is not informative enough.





Classical ML Models and Performance Estimation

Exploring Classical Machine Learning Models

Candidate Models:

•**Logistic Regression** – linear decision boundaries, interpretable, good for linearly separable data.

•**K-Nearest Neighbors (KNN)** – non-parametric, sensitive to feature scaling.

•**Decision Trees** – interpretable, but prone to overfitting.

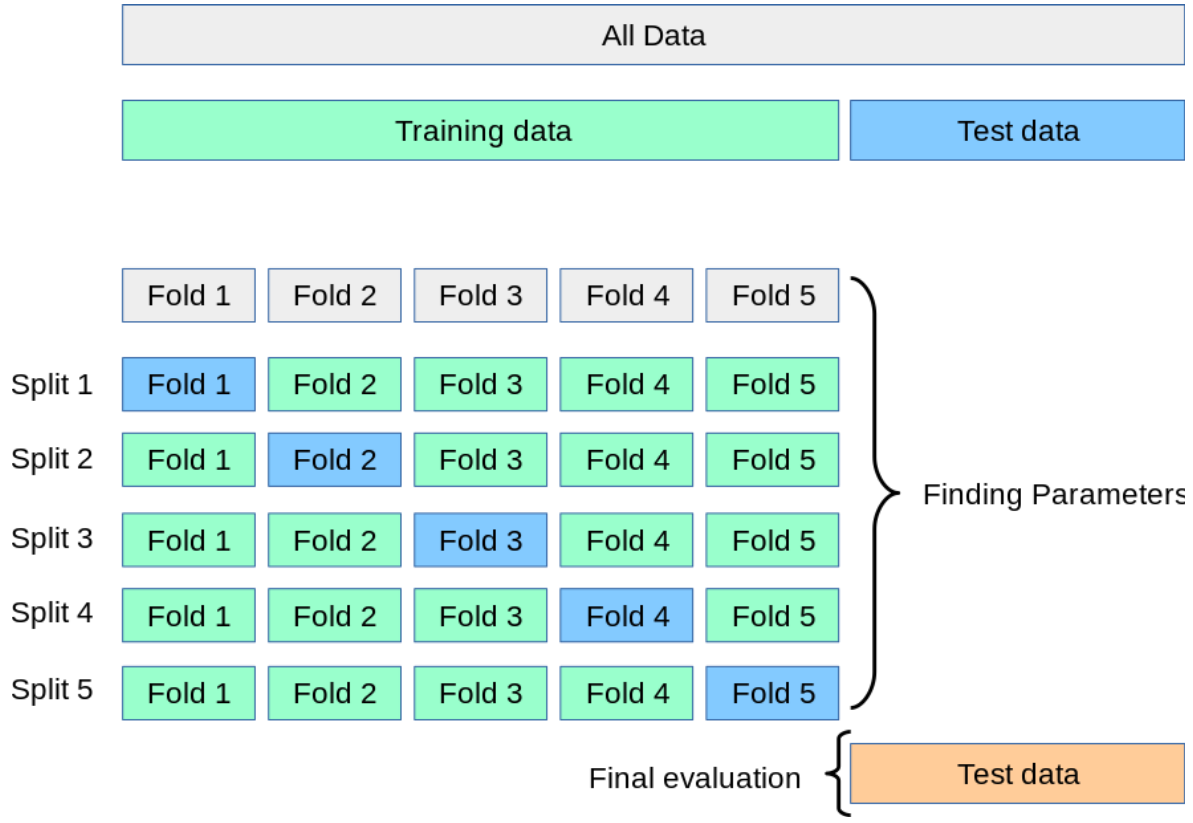
•**Random Forest** – ensemble of trees, reduces overfitting, less interpretable.

•**Support Vector Machines (SVM)** – effective in high-dimensional spaces, uses kernel tricks.

Model Training and Evaluation:

•**Cross-validation (e.g., k-fold)** ensures robust performance estimation.
•Evaluate with metrics:
- Classification: accuracy, precision, recall, F1-score, AUC-ROC.
- Regression: MSE, RMSE, MAE, R^2.

Bias-variance tradeoff guides model choice. Simpler models may underfit (high bias), complex ones may overfit (high variance). Regularization can help balance this.



Iterative Model Refinement and Selection

Refining, Tuning, and Selecting the Optimal Model

Model Tuning:

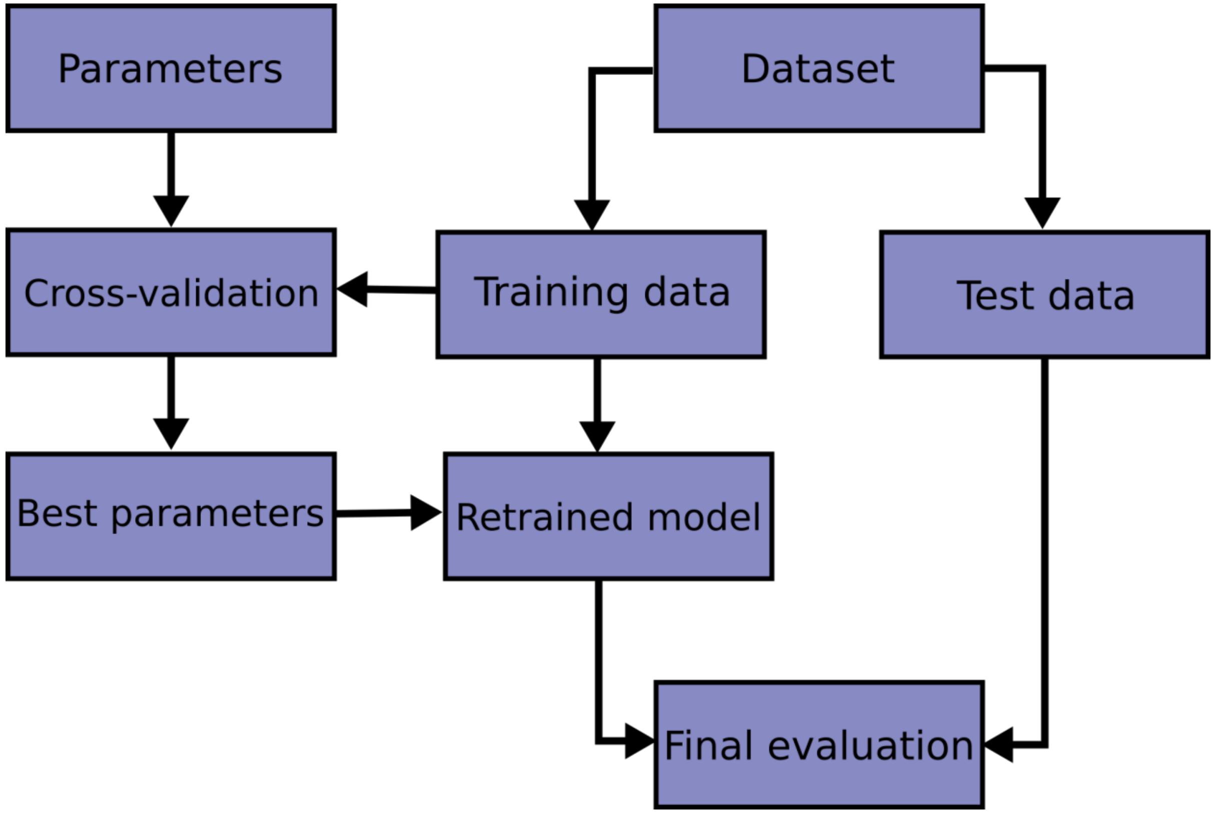
•**GridSearchCV / RandomizedSearchCV** to optimize hyperparameters.
•Use validation performance to avoid overfitting to training data.
•Apply **feature scaling**, **encoding**, or **dimensionality reduction** as needed.

Model Comparison:

•Compare performance using **statistical tests** (e.g., paired t-test) over cross-validation folds.
•Consider **interpretability vs. performance** tradeoff.

Final Selection Criteria:

•Best generalization performance on **unseen test data**.
•Meets application-specific constraints (e.g., inference speed, memory usage).
•Model stability and explainability.



•The no free lunch theorem tells us that no model is best for all tasks — model selection is inherently data-dependent.

Model selection example

Model selection

```
from sklearn.datasets import load_breast_cancer
from sklearn.model_selection import cross_val_score, StratifiedKFold
from sklearn.linear_model import LogisticRegression
from sklearn.ensemble import RandomForestClassifier
from scipy.stats import ttest_rel
import numpy as np

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

# Define models
model1 = LogisticRegression(max_iter=1000, solver='liblinear') # Simpler linear model
model2 = RandomForestClassifier(random_state=42) # More complex ensemble

# Stratified K-Folds
cv = StratifiedKFold(n_splits=10, shuffle=True, random_state=1)

# Evaluate both models
scores1 = cross_val_score(model1, X, y, cv=cv, scoring='accuracy')
scores2 = cross_val_score(model2, X, y, cv=cv, scoring='accuracy')

print("Logistic Regression Accuracies:", scores1)
print("Random Forest Accuracies:", scores2)
```

```
Logistic Regression Accuracies: [0.94736842
0.94736842 0.92982456 0.94736842 0.92982456
0.96491228 0.98245614 0.96491228 0.96491228
0.96428571]

Random Forest Accuracies: [0.94736842
0.94736842 1. 0.98245614 0.9122807
0.94736842 1. 0.94736842 0.94736842
0.92857143]
```

- **Paired t-test** offers a statistically principled way to compare models.
- **Cross-validation** ensures fair and robust performance estimation.
- Use **scientific significance (p-value)** alongside **practical significance (accuracy improvement)** when choosing the best model.

```
# Paired t-test
t_stat, p_value = ttest_rel(scores2, scores1)

print(f"t-statistic: {t_stat:.4f}")
print(f"p-value: {p_value:.4f}")
```

```
t-statistic: 0.1697
p-value: 0.8690
```

- **p-value < 0.05** → We **reject the null hypothesis**: there's a **statistically significant difference** between models.

- **t-statistic > 0** → RandomForestClassifier has **significantly higher accuracy** than LogisticRegression.

Why Random Forest Might Perform Better

Aspect	Logistic Regression	Random Forest
Model type	Linear classifier	Non-linear ensemble
Assumptions	Assumes linear decision boundaries	Captures complex interactions
Interpretability	High	Moderate to low
Overfitting risk	Low	Moderate (but mitigated via bagging)
Feature importance	Coefficients	Tree-based impurity scores

Azercell

Model selection Full pipeline

Model selection

```
from sklearn.datasets import load_breast_cancer
from sklearn.model_selection import KFold, cross_validate
from sklearn.pipeline import Pipeline
from sklearn.preprocessing import StandardScaler
from sklearn.impute import SimpleImputer
from sklearn.linear_model import LogisticRegression
from sklearn.compose import ColumnTransformer
import pandas as pd
import numpy as np

# Load dataset
X_raw, y = load_breast_cancer(return_X_y=True)
X = pd.DataFrame(X_raw, columns=load_breast_cancer().feature_names)

# Introduce missing values to simulate real-world data
X.iloc[::50, 0] = np.nan

# Preprocessing pipeline for numeric features
numeric_features = X.columns.tolist()
numeric_transformer = Pipeline([
  ('imputer', SimpleImputer(strategy='mean')),
  ('scaler', StandardScaler())
])

# ColumnTransformer to apply to all features
preprocessor = ColumnTransformer([
  ('num', numeric_transformer, numeric_features)
])

# Full pipeline: preprocessing + model
pipeline = Pipeline([
  ('preprocessor', preprocessor),
  ('classifier', LogisticRegression(max_iter=1000))
])
```

```
# Scoring metrics
scoring = [
  'accuracy',
  'precision',
  'recall',
  'f1',
  'roc_auc']

# KFold definition
kf = KFold(n_splits=5, shuffle=True, random_state=42)

# Evaluate multiple metrics
cv_results = cross_validate(
  pipeline,
  X,
  y,
  cv=kf,
  scoring=scoring,
  return_train_score=False)

# Print results
for metric in scoring:
  test_scores = cv_results[f'test_{metric}']
  print(f"{metric.capitalize()} scores: {test_scores}")
  print(f"Mean {metric}: {test_scores.mean():.4f}\n")
```

```
Accuracy scores: [0.97368421 0.98245614 0.96491228 0.99122807 0.97345133] Mean accuracy: 0.9771
Precision scores: [0.97222222 1. 0.95890411 0.98611111 0.97058824] Mean precision: 0.9776
Recall scores: [0.98591549 0.97402597 0.98591549 1. 0.98507463] Mean recall: 0.9862
F1 scores: [0.97902098 0.98684211 0.97222222 0.99300699 0.97777778] Mean f1: 0.9818
Roc_auc scores: [0.99737963 1. 0.98427776 0.99770717 0.99415964] Mean roc_auc: 0.9947
```

Ensemble methods

Introduction to Ensemble Learning

Ensemble methods

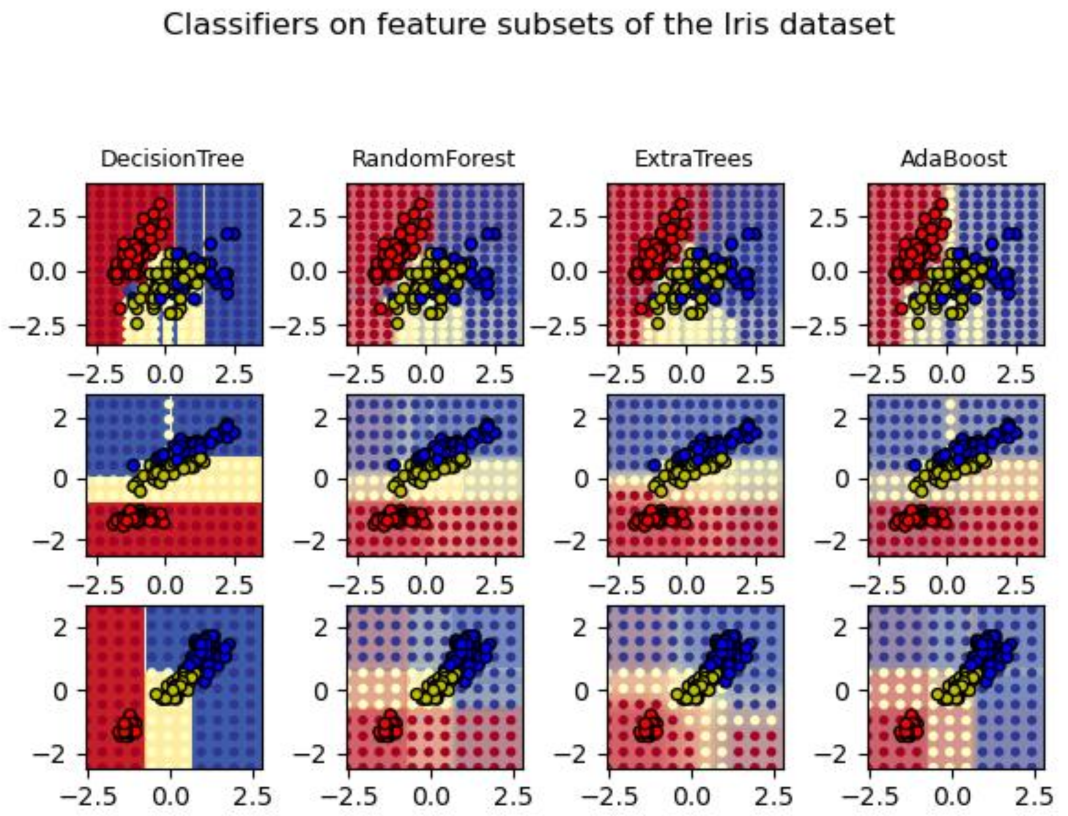
What is Ensemble Learning?
•Ensemble learning combines multiple models (often called **weak learners**) to build a more robust and accurate model.
•It leverages the **wisdom of the crowd**: aggregating multiple predictions typically outperforms a single model.

Types of Ensembles:

1.Bagging (Bootstrap Aggregating)
2.Boosting
3.Stacking

Why Use Ensembles?

•Reduce variance (Bagging)
•Reduce bias (Boosting)
•Leverage model diversity (Stacking)



Bagging (Bootstrap Aggregating)

Ensemble methods

Concept:

•Train multiple models on **bootstrapped samples** of data.
•Combine predictions using **voting (classification)** or **averaging (regression)**.

Pros:

•Reduces variance
•Handles overfitting well

Cons:

•Limited bias reduction

Popular Algorithm: RandomForestClassifier

```
from sklearn.ensemble import RandomForestClassifier
from sklearn.datasets import load_iris
from sklearn.model_selection import train_test_split

X, y = load_iris(return_X_y=True)
X_train, X_test, y_train, y_test = train_test_split(X, y)

model = RandomForestClassifier(
  n_estimators=100,
  random_state=42)
model.fit(X_train, y_train)
print(model.score(X_test, y_test))
```

Bagging pipeline

Ensemble methods

```
from sklearn.ensemble import BaggingClassifier
from sklearn.tree import DecisionTreeClassifier
from sklearn.metrics import accuracy_score

accuracy = make_scorer(accuracy_score)

cv = KFold(5, shuffle=True, random_state=0)

model = BaggingClassifier(
  estimator=DecisionTreeClassifier(),
  n_estimators=300,
  bootstrap=True,
  max_samples=1.0,
  max_features=1.0,
  random_state=0)

column_transform = ColumnTransformer([
   ('categories', categorical_onehot_encoding, low_card_categorical),
   ('numeric', numeric_passthrough, continuous)
  ],
  remainder='drop',
  verbose_feature_names_out=False,
  sparse_threshold=0.0)

model_pipeline = Pipeline([
  ('processing', column_transform),
  ('modeling', model)
])
```

```
cv_scores = cross_validate(
  estimator=model_pipeline,
  X=data,
  y=target_median,
  scoring=accuracy,
  cv=cv,
  return_train_score=True,
  return_estimator=True
)

mean_cv = np.mean(cv_scores['test_score'])
std_cv = np.std(cv_scores['test_score'])
fit_time = np.mean(cv_scores['fit_time'])
score_time = np.mean(cv_scores['score_time'])

print(f"{mean_cv:0.3f} ({std_cv:0.3f})",
      f"fit: {fit_time:0.2f}",
      f"secs pred: {score_time:0.2f} secs")
```

Azercell

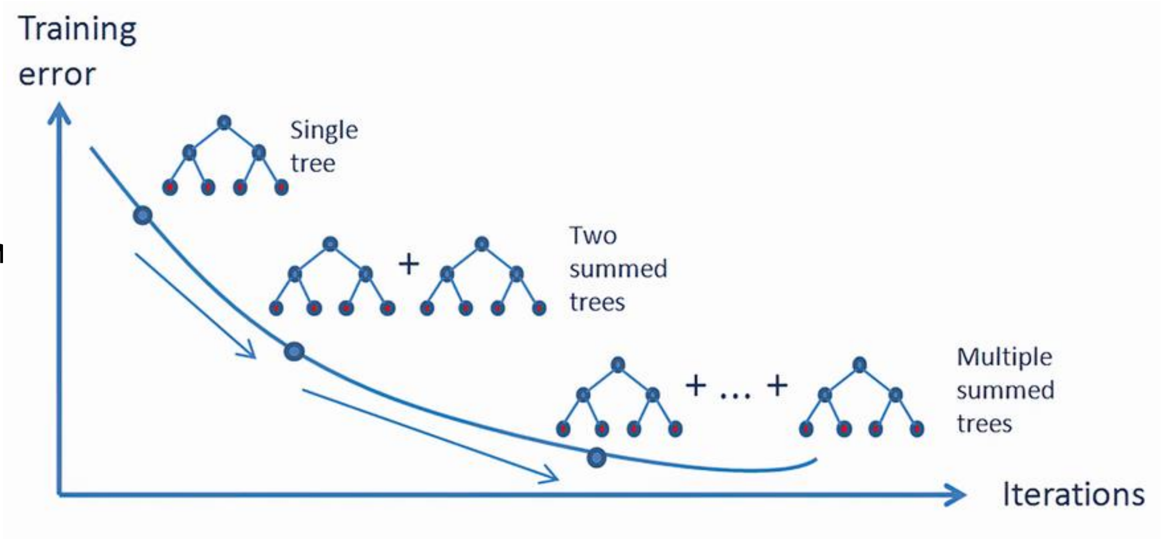
Boosting

Ensemble methods

Concept:
•Train models **sequentially**. Each new model focuses on **correcting th**
errors of the previous one.
•Models are **weighted** in the final prediction.
Pros:
•Reduces both bias and variance
•High accuracy
Cons:
•Can overfit if not properly regularized
•Training is sequential (slower)

Popular Algorithms:

- AdaBoost,
- GradientBoosting,
- XGBoost,
- LightGBM
- Catboost



```
from sklearn.ensemble import AdaBoostClassifier
from sklearn.tree import DecisionTreeClassifier

# Updated keyword: use 'estimator' instead of 'base_estimator'
model = AdaBoostClassifier(
  estimator=DecisionTreeClassifier(max_depth=1),
  n_estimators=50,
  learning_rate=1.0,
  random_state=42
)

model.fit(X_train, y_train)
print(model.score(X_test, y_test))
```

```
0.9473684210526315
```

Boosting algorithm

Ensemble methods

```
from sklearn.tree import DecisionTreeRegressor
import numpy as np

class GradientBoosting():
  def __init__(self, learning_rate=0.1, n_estimators=10, **params):
    self.learning_rate = learning_rate
    self.n_estimators = n_estimators
    self.params = params
    self.trees = list()

  def sigmoid(self, x):
    x = np.clip(x, -100, 100)
    return 1 / (1 + np.exp(-x))

  def logit(self, x, eps=1e-6):
    xp = np.clip(x, eps, 1-eps)
    return np.log(xp / (1 - xp))

  def gradient(self, y_true, y_pred):
    gradient = y_pred - y_true
    return gradient

  def fit(self, X, y):
    self.init = self.logit(np.mean(y))
    y_pred = self.init * np.ones((X.shape[0],))
    for k in range(self.n_estimators):
      gradient = self.gradient(self.logit(y), y_pred)
      tree = DecisionTreeRegressor(**self.params)
      tree.fit(X, -gradient)
      self.trees.append(tree)
      y_pred += (self.learning_rate * tree.predict(X))
```

```
def predict_proba(self, X):
  y_pred = self.init * np.ones((X.shape[0],))
  for tree in self.trees:
    y_pred += (self.learning_rate * tree.predict(X))
  return self.sigmoid(y_pred)

def predict(self, X, threshold=0.5):
  proba = self.predict_proba(X)
  return np.where(proba >= threshold, 1, 0)
```

Stacking

Ensemble methods

Concept:

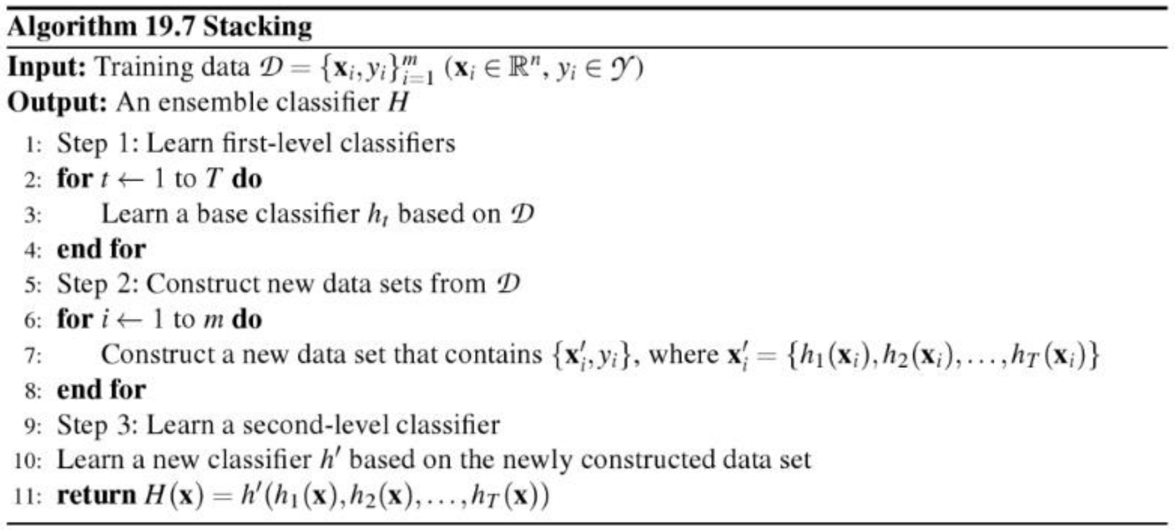
- Combine predictions of **multiple base models** using a **meta-model** (often logistic regression or linear regression).
- Base models learn the data, the meta-model learns from their predictions.

Pros:

- Can combine diverse models (tree + SVM + neural net)
- Captures both low and high bias/variance patterns

Cons:

- More complex, prone to overfitting if improperly validated



Algorithm 19.7 Stacking

Input: Training data $\mathcal{D} = \{\mathbf{x}_i, y_i\}_{i=1}^m$ ($\mathbf{x}_i \in \mathbb{R}^n$, $y_i \in \mathcal{Y}$)

Output: An ensemble classifier H

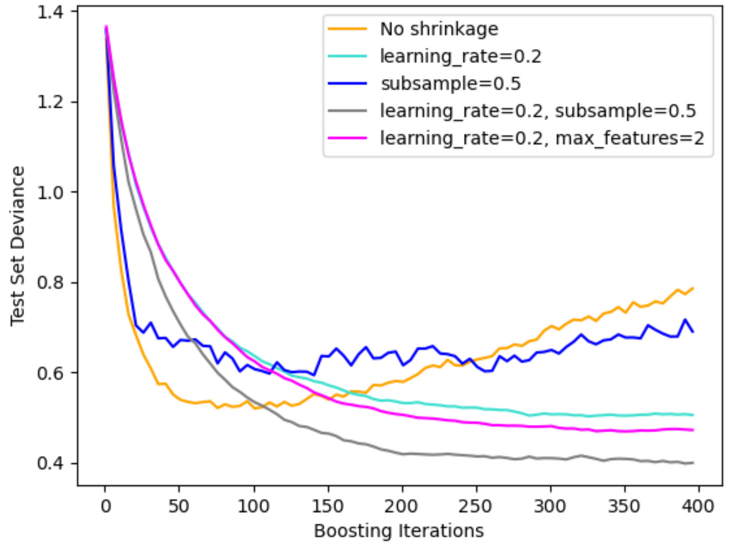
```
1:  Step 1: Learn first-level classifiers
2:  for $t \leftarrow 1$ to $T$ do
3:      Learn a base classifier $h_t$ based on $\mathcal{D}$
4:  end for
5:  Step 2: Construct new data sets from $\mathcal{D}$
6:  for $i \leftarrow 1$ to $m$ do
7:      Construct a new data set that contains $\{\mathbf{x}'_i, y_i\}$, where $\mathbf{x}'_i = \{h_1(\mathbf{x}_i), h_2(\mathbf{x}_i), \ldots, h_T(\mathbf{x}_i)\}$
8:  end for
9:  Step 3: Learn a second-level classifier
10: Learn a new classifier $h'$ based on the newly constructed data set
11: return $H(\mathbf{x}) = h'(h_1(\mathbf{x}), h_2(\mathbf{x}), \ldots, h_T(\mathbf{x}))$
```

```python
from sklearn.ensemble import StackingClassifier
from sklearn.linear_model import LogisticRegression
from sklearn.svm import SVC
from sklearn.naive_bayes import GaussianNB

estimators = [
  ('svc', SVC(probability=True)),
  ('nb', GaussianNB())
]

model = StackingClassifier(
  estimators=estimators,
  final_estimator=LogisticRegression()
)
model.fit(X_train, y_train)
print(model.score(X_test, y_test))
```

When to Use Which?

Ensemble methods

Strategy	Use When...	Key Focus
Bagging	High variance models (e.g., decision trees)	Variance
Boosting	High bias models, need higher accuracy	Bias + Errors
Stacking	Need to combine diverse models	Diversity



Bagging → Parallel training (good for speed)
Boosting → Sensitive to outliers (careful with noise)
Stacking → Needs good validation to avoid overfitting

Hyperparameters tuning

Grid Search – Exhaustive Hyperparameter Tuning

Hyperparameters tuning

Definition:
Grid Search tests **all possible combinations** of predefined hyperparameter values.

How it works:
- You define a **parameter grid** (e.g., `{'C': [0.1, 1, 10], 'kernel': ['linear', 'rbf']}`).
- For each combination, the model is trained and evaluated using **cross-validation**.
- The best combination is selected based on a **scoring metric** (e.g., accuracy, F1).

Pros:
- Simple and **deterministic**.
- Good for **small search spaces**.

Cons:
- **Computationally expensive** for large grids.
- Doesn't prioritize promising areas of search space.

```
from sklearn.model_selection import GridSearchCV
from sklearn.svm import SVC

param_grid = {
  'C': [0.1, 1, 10],
  'kernel': ['linear', 'rbf’]
}

grid = GridSearchCV(
  SVC(),
  param_grid,
  scoring='accuracy’,
  cv=5)

grid.fit(X_train, y_train)
print(grid.best_params_)
```

```
{'C': 1, 'kernel': 'linear'}
```

Random Search – Efficient Exploration

Hyperparameters tuning

Definition:
Random Search randomly samples hyperparameter combinations from given distributions.

How it works:
- You define **ranges or distributions** (e.g., uniform, log-uniform).
- Random combinations are sampled for a fixed number of iterations.

Pros:
- **More efficient** than Grid Search when some hyperparameters are more important.
- Useful when there are many parameters.

Cons:
- Still **requires many evaluations**.
- No feedback from previous trials (i.e., no learning from past evaluations).

```
from sklearn.model_selection import RandomizedSearchCV
from scipy.stats import uniform
from sklearn.ensemble import RandomForestClassifier
from scipy.stats import randint

param_dist = {
  'n_estimators': [50, 100, 200],
  'max_depth': randint(1, 10)
}
rand_search = RandomizedSearchCV(
  RandomForestClassifier(),
  param_dist,
  n_iter=10,
  scoring='accuracy',
  cv=5)
rand_search.fit(X_train, y_train)
print(rand_search.best_params_)
```

```
{'max_depth': 9,
 'n_estimators': 200}
```

Optuna – Intelligent Optimization

Hyperparameters tuning

Optuna is a **state-of-the-art automatic hyperparameter optimization** framework using **Bayesian optimization and pruning**.

Key concepts:

- Uses **TPE (Tree-structured Parzen Estimator)** for smarter search.
- Learns from past trials to **focus on promising regions**.
- Supports **early stopping (pruning)** to skip unpromising trials.

Basic flow:

1. Define an objective(trial) function.
2. Use Optuna to minimize or maximize the objective.

```
pip install optuna
```

```
import optuna
from sklearn.ensemble import RandomForestClassifier
from sklearn.model_selection import cross_val_score

def objective(trial):
  n_estimators = trial.suggest_int('n_estimators', 50, 200)
  max_depth = trial.suggest_int('max_depth', 2, 32)
  clf = RandomForestClassifier(
    n_estimators=n_estimators,
    max_depth=max_depth)
return -cross_val_score(
  clf,
  X_train,
  y_train,
  cv=3,
   scoring='accuracy').mean()

study = optuna.create_study(direction='minimize')
study.optimize(objective, n_trials=50)
print(study.best_params)
```

Why do we use negative cross_val_score?

•cross_val_score() returns **positive values** for metrics like accuracy.
•But Optuna requires you to **minimize** or **maximize** your objective explicitly.
•If you want to **maximize accuracy**, you either:

- **maximize** the score directly, or
- **minimize the negative** of the score.

Optuna advanced features

Hyperparameters tuning

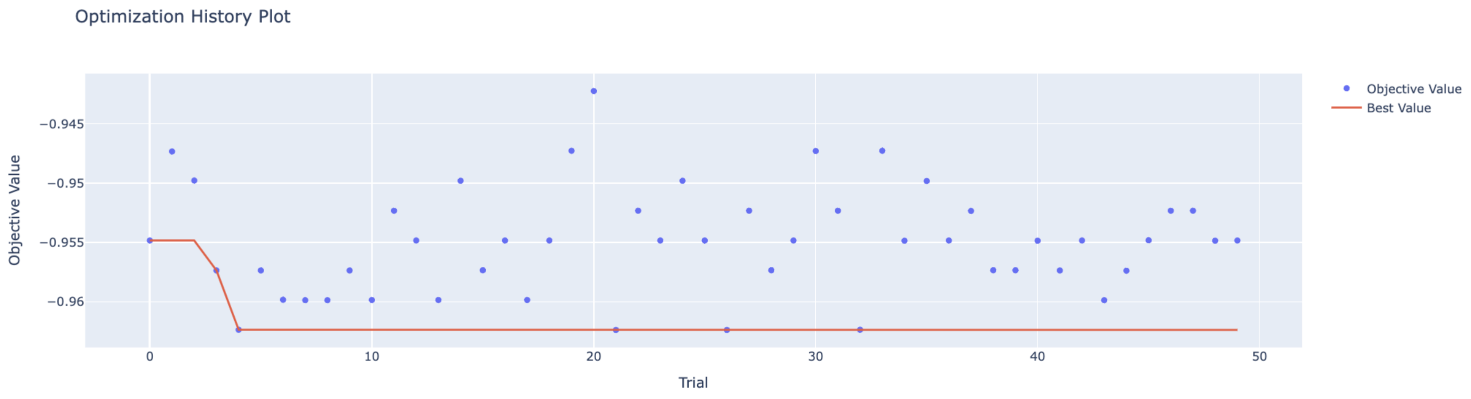
Advanced capabilities:

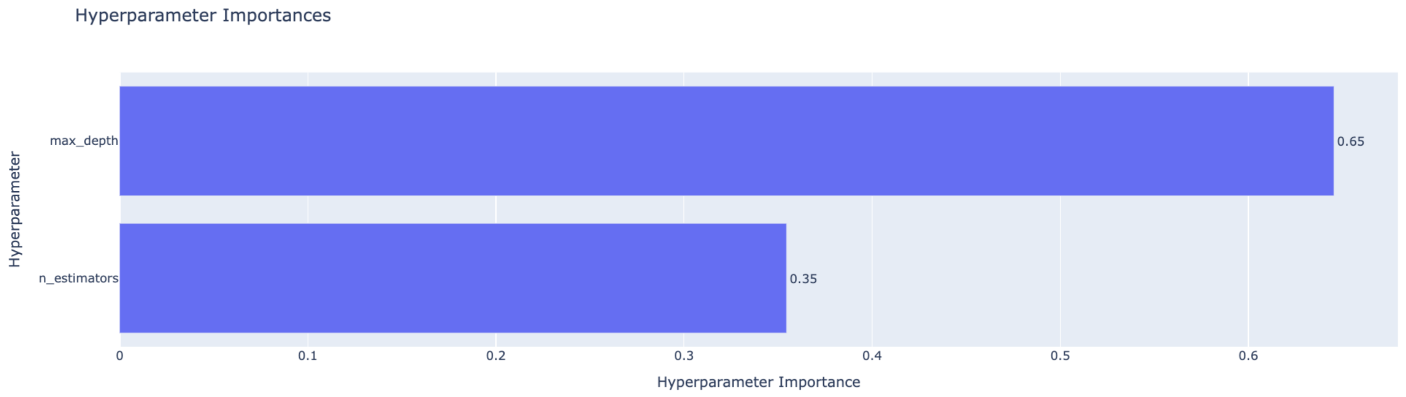
- **Pruning**: Stops poor trials early to save time.
- **Visualization**: Provides importance plots, optimization history, and parameter correlation plots.
- **Parallel execution**: Easily run multiple trials in parallel.
- **Sampler customization**: Use TPE, CMA-ES, or random sampling.

```
!pip install nbformat
```

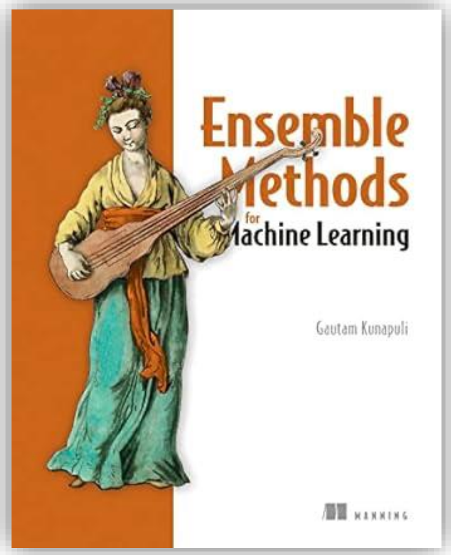
Visualization example:

```
optuna.visualization.plot_optimization_history(study).show()
optuna.visualization.plot_param_importances(study).show()
```

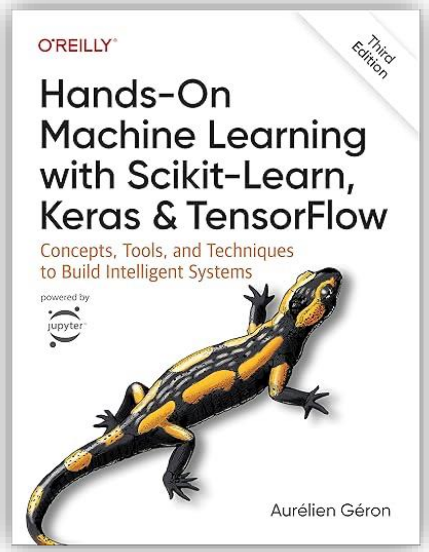




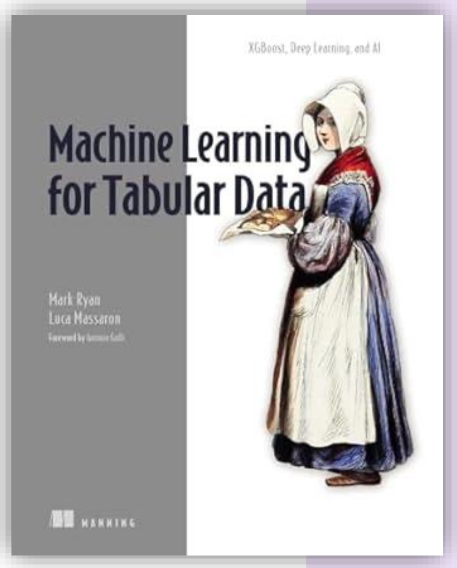
Literature and links



Ensemble Methods for Machine Learning
by Gautam Kunapuli



Hands-On Machine Learning with Scikit-Learn
by Aurélien Géron



Machine Learning for Tabular Data: XGBoost, Deep Learning, and AI
by Mark Ryan Luca Massaron

Kfold

https://scikit-learn.org/stable/modules/cross_validation.html

Hyperparameters tune

https://optuna.org/

Ensemble methods

stacking:

https://medium.com/@brijesh_soni/stacking-to-improve-model-performance-a-comprehensive-guide-on-ensemble-learning-in-python-9ed53c93ce28

https://rasbt.github.io/mlxtend/user_guide/classifier/StackingClassifier/.

bagging

https://www.datacamp.com/tutorial/what-bagging-in-machine-learning-a-guide-with-examples

Azercell

Thank you for your attention