A Step-by-Step Professional Guide for Choosing Hyperparameters

Type	@datasciencebrain

Mastering Hyperparameter Tuning: A Step-by-Step Professional Guide

1. Understanding Hyperparameters

Hyperparameters are settings used to control the behavior and performance of machine learning algorithms. Unlike model parameters (weights learned from data), hyperparameters are specified manually or set through optimization processes before training begins.

Common hyperparameters include:

- **Learning rate:** Determines the step size during optimization.
- **Regularization terms:** (e.g., L1, L2) penalize overly complex models.
- **Number of estimators/trees:** (Random Forest, XGBoost, etc.) affects model complexity.
- **Depth of tree:** Controls complexity and risk of overfitting in decision trees.
- **Batch size:** Number of samples processed before updating model parameters in neural networks.
- **Epochs:** Number of passes through the entire training dataset.
- **Kernel type and parameters:** (SVM models) define boundaries between classes.
- **Hidden layers/nodes:** Structure of neural networks.

---

2. Initial Analysis and Preparation

Step-by-step approach:

- **Clearly Define Problem and Metrics:**
  - Identify whether the problem is classification, regression, clustering, etc.
  - Select evaluation metrics (accuracy, precision, recall, ROC-AUC, F1-score, RMSE, MAE, etc.) that align with your business objectives.
- **Understand Your Dataset:**
  - Explore dataset size, dimensions, number of classes, feature distributions, missing values, and outliers.
  - Determine data splitting strategy (train-validation-test split, cross-validation).
- **Benchmarking:**
  - Establish baseline models with default hyperparameters.
  - Record baseline performance for reference.

```
# Step 2: Initial Analysis and Preparation

# Import necessary libraries
import pandas as pd
from sklearn.model_selection import train_test_split

# Load your dataset
df = pd.read_csv('your_dataset.csv')

# Basic exploration of the dataset
print("Dataset Information:")
print(df.info())

print("\nDataset Statistics:")
print(df.describe())

# Define feature matrix X and target vector y
X = df.drop(columns=['target'])
y = df['target']
```

```
# Split the data into training (70%), validation (15%), and test sets (15%)
X_train, X_temp, y_train, y_temp = train_test_split(
    X, y, test_size=0.30, random_state=42, stratify=y
)

# Further split the temp dataset equally into validation and test sets
X_val, X_test, y_val, y_test = train_test_split(
    X_temp, y_temp, test_size=0.50, random_state=42, stratify=y_temp
)

# Output the shapes to confirm splits
print("\nData splits:")
print(f"Training Set: {X_train.shape}, {y_train.shape}")
print(f"Validation Set: {X_val.shape}, {y_val.shape}")
print(f"Test Set: {X_test.shape}, {y_test.shape}")
```

Replace `'your_dataset.csv'` **and** `'target'` with your dataset filename and actual target variable name. This step ensures your data is properly understood and split before proceeding with further analysis or hyperparameter tuning.

---

3. Identify Hyperparameters and Their Effects

Before you begin tuning hyperparameters, you must clearly identify and understand their potential impact on your model's performance. Different hyperparameters have distinct effects on the behavior, accuracy, generalization, and complexity of your model.

Common hyperparameters and their effects (examples):

1. Learning Rate *(Neural Networks, Gradient Boosting Models)*

- Controls how quickly or slowly a model learns from training data.
- **Low** values (e.g., `0.001`) lead to slow but stable convergence.
- **High** values (e.g., `0.1` or higher) risk unstable or divergent behavior.

2. Max Depth *(Decision Trees, Random Forests, XGBoost)*

- Controls the complexity of decision boundaries.
- **High** values (e.g., `10` or higher) increase model complexity, risk overfitting.
- **Low** values (e.g., `3`) simplify the model, risk underfitting.

3. Number of Estimators (Trees) *(Random Forest, XGBoost, Gradient Boosting)*

- Controls how many trees or estimators the algorithm builds.
- **Higher** values generally increase accuracy but slow training speed and may overfit.

4. Regularization Parameters (alpha, lambda, L1, L2) *(Linear/Logistic Regression, Ridge/Lasso)*

- Penalizes overly complex models to reduce overfitting.
- **Higher regularization strength** simplifies models, potentially underfitting data.
- **Lower strength** allows models to fit data more closely, risking overfitting.

5. Kernel Type *(Support Vector Machines)*

- Determines the decision boundary shape.
- **Linear**: simple datasets.
- **RBF (radial basis function)**: non-linear, complex relationships.

Practical Python code to list hyperparameters and effects:

Here's how to practically inspect hyperparameters available for different models using Python (scikit-learn example):

```
# Step 3: Identify Hyperparameters and Their Effects

from sklearn.ensemble import RandomForestClassifier
from sklearn.linear_model import LogisticRegression
from sklearn.svm import SVC
from xgboost import XGBClassifier
```

```
# Instantiate models with default parameters
rf = RandomForestClassifier()
lr = LogisticRegression()
svc = SVC()
xgb = XGBClassifier(use_label_encoder=False, eval_metric='logloss')

# Print default hyperparameters for Random Forest
print("Random Forest Hyperparameters:\n", rf.get_params(), "\n")

# Logistic Regression Hyperparameters
print("Logistic Regression Hyperparameters:\n", lr.get_params(), "\n")

# Support Vector Classifier Hyperparameters
print("SVC Hyperparameters:\n", svc.get_params(), "\n")

# XGBoost Classifier Hyperparameters
print("XGBoost Hyperparameters:\n", xgb.get_params(), "\n")
```

Output Interpretation:

- Run this code to clearly see all available hyperparameters.
- Investigate each hyperparameter and document its potential effects as described above.
- Understanding these clearly will guide effective tuning.

Practical Example (Interpreting Hyperparameter Effects in XGBoost):

```
# Example: Effect of hyperparameters in XGBoost
hyperparameter_effects = {
    'learning_rate': 'Low values slow learning, high values risk instability.',
    'max_depth': 'Controls complexity. Higher values risk overfitting.',
    'n_estimators': 'Number of boosting rounds; more improves accuracy but may o
verfit.',
    'subsample': 'Fraction of data randomly sampled per tree; reduces overfitting if
<1.',
    'colsample_bytree': 'Fraction of features randomly sampled per tree; reduces c
```

```
orrelation among trees.',
}

# Print effects
print("XGBoost Hyperparameter Effects:")
for param, effect in hyperparameter_effects.items():
    print(f"- {param}: {effect}")
```

Action items (recommended approach):

- List and document all hyperparameters of your chosen model(s).
- Describe each hyperparameter clearly (as demonstrated above).
- Note potential effects on model complexity, accuracy, and generalization explicitly.
- Use these notes as a foundation to structure your hyperparameter tuning strategy.

4. Select the Right Hyperparameter Search Method

Choosing the appropriate hyperparameter search method depends on several factors: computational resources, size of hyperparameter space, and the complexity of the model. Here's a detailed guide on selecting the optimal method for hyperparameter tuning, along with clear Python code implementations.

Hyperparameter Search Methods:

1. Grid Search

Explanation:

- Grid search exhaustively evaluates all combinations within a predefined hyperparameter grid.
- Recommended for small, manageable hyperparameter spaces.

Pros:

- Thorough exploration; guaranteed evaluation of all parameter combinations.
- Simple to implement and interpret.

Cons:

- Computationally expensive, especially with large search spaces.

Python Implementation Example:

```
from sklearn.model_selection import GridSearchCV
from sklearn.ensemble import RandomForestClassifier

# Define the hyperparameter grid
param_grid = {
    'n_estimators': [100, 200],
    'max_depth': [3, 5, 7],
    'max_features': ['auto', 'sqrt']
}

# Initialize model
rf_model = RandomForestClassifier(random_state=42)

# Setup GridSearchCV
grid_search = GridSearchCV(
    estimator=rf_model,
    param_grid=param_grid,
    scoring='accuracy',
    cv=5,
    n_jobs=-1
)

# Fit GridSearchCV
grid_search.fit(X_train, y_train)

# Best parameters
print("Best Grid Search parameters:", grid_search.best_params_)
```

2. Randomized Search

Explanation:

- Randomized search samples a fixed number of random combinations from the hyperparameter space.
- Efficient and practical for larger search spaces.

Pros:

- More efficient than grid search; covers broader search space.
- Usually achieves similar or better results compared to grid search.

Cons:

- Not exhaustive; may miss optimal combinations if number of iterations is low.

Python Implementation Example:

```
from sklearn.model_selection import RandomizedSearchCV
from xgboost import XGBClassifier

# Define hyperparameter distribution
param_dist = {
    'max_depth': [3, 5, 7, 10],
    'learning_rate': [0.01, 0.05, 0.1, 0.2],
    'n_estimators': [100, 200, 300, 500],
    'subsample': [0.7, 0.8, 0.9, 1.0],
}

# Model initialization
xgb_model = XGBClassifier(random_state=42, use_label_encoder=False, eval_met
ric='logloss')

# RandomizedSearchCV setup
random_search = RandomizedSearchCV(
    estimator=xgb_model,
    param_distributions=param_dist,
    n_iter=20,
    scoring='accuracy',
    cv=5,
    n_jobs=-1,
    random_state=42
```

```
)

# Fit model
random_search.fit(X_train, y_train)

# Best parameters
print("Best Random Search parameters:", random_search.best_params_)
```

---

3. Bayesian Optimization

Explanation:

- Bayesian optimization uses a probabilistic model to guide the search toward promising hyperparameters based on previous evaluations.
- Efficient for expensive models with complex hyperparameter spaces.

Pros:

- Highly efficient, faster convergence.
- Adaptively searches optimal hyperparameter regions.

Cons:

- Slightly more complex implementation.
- Requires specialized libraries.

Python Implementation (Optuna example):

```
import optuna
from sklearn.metrics import accuracy_score
from xgboost import XGBClassifier

# Objective function for Optuna
def objective(trial):
    params = {
        'max_depth': trial.suggest_int('max_depth', 3, 10),
        'learning_rate': trial.suggest_loguniform('learning_rate', 0.01, 0.3),
        'n_estimators': trial.suggest_int('n_estimators', 100, 500),
        'subsample': trial.suggest_uniform('subsample', 0.6, 1.0),
```

```
    }
    model = XGBClassifier(**params, random_state=42, use_label_encoder=False,
eval_metric='logloss')
    model.fit(X_train, y_train)
    preds = model.predict(X_val)
    accuracy = accuracy_score(y_val, preds)
    return accuracy

# Run Bayesian optimization using Optuna
study = optuna.create_study(direction='maximize')
study.optimize(objective, n_trials=50)

# Best parameters
print("Best Bayesian parameters (Optuna):", study.best_params)
```

4. Hyperband/TPE (Tree-structured Parzen Estimator)

- Efficiently explores large hyperparameter spaces by adaptively allocating resources.
- **Pros:** Optimal resource allocation; effective for complex models.
- **Cons:** Slightly complex; may require specialized libraries.

Choosing the Right Method:

Situation	Recommended Method
Small hyperparameter space, abundant computational power	**Grid Search**
Large hyperparameter space, limited computation	**Randomized Search**
Complex model, expensive evaluations, requires efficiency	**Bayesian Optimization**

Summary of Recommended Usage:

- **Grid Search:**

  When thorough exploration is necessary, small search spaces.

- **Randomized Search:**

  General-purpose, efficient exploration, large parameter spaces.

- **Bayesian Optimization (e.g., Optuna):**

  When hyperparameter evaluation is computationally expensive and optimal resource utilization is critical.

Action Items:

- Consider computational resources and size of hyperparameter space.
- Choose a method that best balances exploration, efficiency, and practicality.
- Proceed confidently with your selected method into detailed hyperparameter tuning.

---

5. Define Hyperparameter Search Space

Defining a hyperparameter search space involves clearly specifying a range of values or options for each hyperparameter you want to tune. The goal is to identify meaningful and efficient boundaries or distributions within which hyperparameters will be optimized.

This search space can be categorized into three types of hyperparameters:

- **Categorical hyperparameters:** A discrete set of options (e.g., kernel type in SVM).
- **Integer hyperparameters:** Whole-number ranges (e.g., max_depth in decision trees).
- **Continuous hyperparameters:** Numerical values within a range (e.g., learning rate).

How to Define a Hyperparameter Search Space

Below are explicit Python examples demonstrating clearly defined search spaces for different ML models. You can directly adapt these to your tasks.

---

1. Example: Hyperparameter Search Space for Random Forest

```
# Hyperparameter search space for Random Forest
rf_param_space = {
    'n_estimators': [100, 200, 300, 500],           # Number of trees
    'max_depth': [None, 5, 10, 15, 20],             # Depth of each tree
    'min_samples_split': [2, 5, 10],                # Minimum samples required to split a
node
    'min_samples_leaf': [1, 2, 4],                  # Minimum samples required at each le
af node
    'max_features': ['auto', 'sqrt', 'log2']        # Features considered at each split
}
```

2. Example: Hyperparameter Search Space for XGBoost

```
# Hyperparameter search space for XGBoost
xgb_param_space = {
    'max_depth': [3, 4, 5, 6, 7, 8],                # Tree depth
    'learning_rate': [0.01, 0.05, 0.1, 0.2],        # Step size shrinkage to prevent overf
itting
    'n_estimators': [100, 200, 300, 500],           # Number of boosting rounds
    'subsample': [0.6, 0.7, 0.8, 0.9, 1.0],         # Data sampling for each tree
    'colsample_bytree': [0.6, 0.7, 0.8, 0.9, 1.0],    # Column sampling (features) per t
ree
    'gamma': [0, 0.1, 0.2, 0.3, 0.5]                # Minimum loss reduction required to
make further partition
}
```

3. Example: Hyperparameter Search Space for Logistic Regression

```
# Hyperparameter search space for Logistic Regression
logreg_param_space = {
    'penalty': ['l1', 'l2', 'elasticnet'],          # Type of regularization
    'C': [0.01, 0.1, 1, 10, 100],                   # Inverse of regularization strength
    'solver': ['saga'],                             # Solver compatible with all penalties
```

```
    'l1_ratio': [0, 0.25, 0.5, 0.75, 1]             # Elastic-net mixing parameter
}
```

4. Example: Hyperparameter Search Space for SVM

```
# Hyperparameter search space for SVM
svm_param_space = {
    'C': [0.1, 1, 10, 100],                      # Regularization parameter
    'kernel': ['linear', 'rbf', 'poly', 'sigmoid'],   # Kernel type
    'gamma': ['scale', 'auto', 0.01, 0.1, 1],        # Kernel coefficient
    'degree': [2, 3, 4]                           # Degree for poly kernel
}
```

5. Example: Hyperparameter Search Space for Neural Networks

```
# Hyperparameter search space for a simple neural network
nn_param_space = {
    'hidden_layer_sizes': [(32,), (64,), (128,), (64, 32), (128, 64)], # Network archite
cture
    'activation': ['relu', 'tanh', 'logistic'],                  # Activation function
    'solver': ['adam', 'sgd'],                                  # Optimization algorithm
    'learning_rate_init': [0.001, 0.005, 0.01, 0.05],                # Initial learning rate
    'batch_size': [32, 64, 128],                                # Size of mini-batches
    'alpha': [0.0001, 0.001, 0.01]                               # L2 regularization strength
}
```

Defining Hyperparameter Space with Distributions (Randomized Search)

Instead of discrete lists, sometimes distributions are more efficient:

Example (Randomized Search with distributions):

```
from scipy.stats import randint, uniform
```

```
xgb_param_dist = {
    'max_depth': randint(3, 11),                # integer between 3 and 10
    'learning_rate': uniform(0.01, 0.29),       # continuous between 0.01 and 0.3
    'n_estimators': randint(100, 501),          # integer between 100 and 500
    'subsample': uniform(0.6, 0.4),             # continuous between 0.6 and 1.0
    'colsample_bytree': uniform(0.6, 0.4)       # continuous between 0.6 and 1.0
}
```

Practical Recommendations (Best Practices):

- **Categorical hyperparameters:**

  Use explicit lists (e.g., `kernel=['linear', 'rbf']`).

- **Integer hyperparameters:**

  Define as discrete intervals or integer distributions (e.g., `randint(3, 10)`).

- **Continuous hyperparameters:**

  Define log-uniform distributions if spanning orders of magnitude (e.g., learning rates).

Action Items:

- Clearly define the hyperparameter search space for your selected model(s).
- Tailor ranges carefully, guided by past experiments or domain knowledge.
- Document your choices clearly for efficient tuning and reproducibility.

6. Choosing the Validation Strategy

1. Train-Validation-Test Split

A simple and effective strategy, best suited when you have a sufficiently large dataset.

Code Example:

```
from sklearn.model_selection import train_test_split

# Assuming X (features) and y (target) are already defined
X_train, X_temp, y_train, y_temp = train_test_split(
    X, y, test_size=0.3, random_state=42, stratify=y
)

# Split the temporary set equally into validation and test
X_val, X_test, y_val, y_test = train_test_split(
    X_temp, y_temp, test_size=0.5, random_state=42, stratify=y_temp
)

# Confirming dataset shapes
print(f"Train set: {X_train.shape}, {y_train.shape}")
print(f"Validation set: {X_val.shape}, {y_val.shape}")
print(f"Test set: {X_test.shape}, {y_test.shape}")
```

2. K-Fold Cross-Validation

Preferred for small to medium-sized datasets. Reduces variance by averaging results across multiple folds.

Code Example:

```
from sklearn.model_selection import cross_val_score, KFold
from sklearn.ensemble import RandomForestClassifier
import numpy as np

# Initialize model
model = RandomForestClassifier(random_state=42)

# Define K-Fold strategy
kf = KFold(n_splits=5, shuffle=True, random_state=42)

# Perform cross-validation
cv_scores = cross_val_score(model, X, y, cv=kf, scoring='accuracy')
```

```
# Results
print("K-Fold Cross-Validation Scores:", cv_scores)
print("Average Score:", np.mean(cv_scores))
```

3. Stratified K-Fold Cross-Validation

Optimal for imbalanced datasets, ensuring each fold retains the original class distribution.

Code Example:

```
from sklearn.model_selection import cross_val_score, StratifiedKFold
from sklearn.ensemble import RandomForestClassifier
import numpy as np

# Initialize model
model = RandomForestClassifier(random_state=42)

# Define Stratified K-Fold strategy
skf = StratifiedKFold(n_splits=5, shuffle=True, random_state=42)

# Perform stratified cross-validation
stratified_scores = cross_val_score(model, X, y, cv=skf, scoring='accuracy')

# Results
print("Stratified K-Fold Cross-Validation Scores:", stratified_scores)
print("Average Score:", np.mean(stratified_scores))
```

How to Choose:

Situation	Best Strategy
Large dataset, plenty of samples	Train-Validation-Test Split
Small-to-medium dataset, reduce variance	K-Fold Cross-Validation
Imbalanced dataset	Stratified K-Fold Cross-Validation

Action Items:

- Assess your dataset size and class balance.
- Choose a validation strategy accordingly.
- Document clearly to ensure reproducibility.

Using the right validation strategy ensures your hyperparameter tuning leads to robust, reliable, and generalizable results.

---

7. Execute the Hyperparameter Search

In this example, we'll perform hyperparameter tuning using **RandomizedSearchCV** from **Scikit-learn**, leveraging parallel computing for speed.

Detailed Python Implementation:

```
# Import necessary libraries
from sklearn.model_selection import RandomizedSearchCV, train_test_split
from sklearn.metrics import accuracy_score, classification_report
from xgboost import XGBClassifier
import pandas as pd
import numpy as np
import time

# Load your dataset
df = pd.read_csv('your_dataset.csv')

# Define features (X) and target (y)
X = df.drop('target', axis=1)
y = df['target']

# Split dataset into train and test (stratify if imbalanced)
X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.2, stratify=y, random_state=42
)

# Initialize model
xgb_model = XGBClassifier(use_label_encoder=False, eval_metric='logloss', rand
```

```
om_state=42)

# Define hyperparameter distributions
param_distributions = {
    'max_depth': np.arange(3, 11),                  # integer values from 3 to 10
    'learning_rate': np.linspace(0.01, 0.3, 30),      # 30 values between 0.01 and 0.3
    'n_estimators': np.arange(100, 501, 50),          # from 100 to 500 in steps of 50
    'subsample': np.linspace(0.6, 1.0, 5),            # values from 0.6 to 1.0
    'colsample_bytree': np.linspace(0.6, 1.0, 5),     # values from 0.6 to 1.0
}

# Configure RandomizedSearchCV
random_search = RandomizedSearchCV(
    estimator=xgb_model,
    param_distributions=param_distributions,
    n_iter=20,                # number of hyperparameter combinations to sample
    scoring='accuracy',       # or your appropriate metric
    cv=5,                     # 5-fold cross-validation
    n_jobs=-1,                # parallel computing (use all cores)
    verbose=2,
    random_state=42
)

# Execute hyperparameter tuning and monitor computational time
start_time = time.time()

random_search.fit(X_train, y_train)

end_time = time.time()

print(f"Hyperparameter tuning took {(end_time - start_time):.2f} seconds.")

# Best hyperparameters
print("Best Hyperparameters Found:", random_search.best_params_)

# Evaluate best model on validation data (cross-validation results)
print(f"Best cross-validation Accuracy: {random_search.best_score_:.4f}")
```

```
# Final evaluation on test data
y_pred = random_search.predict(X_test)
test_accuracy = accuracy_score(y_test, y_pred)
print(f"Test Accuracy: {test_accuracy:.4f}")

# Detailed classification report
print("Classification Report:")
print(classification_report(y_test, y_pred))
```

Explanation of Code:

- **RandomizedSearchCV**:
  - Samples a fixed number (`n_iter`) of random hyperparameter combinations from defined distributions.
  - Uses `cv=5` folds cross-validation to robustly estimate performance.
- **Parallel computing** (`n_jobs=-1`):
  - Utilizes all available CPU cores for faster computation.
- **Performance Monitoring**:
  - Records execution time (`time.time()`) to monitor computational resources used.
  - Reports cross-validation accuracy and final test accuracy clearly.
- **Evaluation**:
  - Provides detailed insights through `classification_report()` (precision, recall, F1-score, support).

Action Items:

- Replace `'your_dataset.csv'` and `'target'` with your dataset filename and target variable.
- Execute this code to perform systematic hyperparameter tuning.
- Record and document best hyperparameters and performance results clearly for future reference.

---

8. Analyze Results and Choose Optimal Hyperparameters

Step-by-step approach:

- **Evaluate best-performing hyperparameters:**
  - Performance on validation set, consistency across folds.
  - Stability (variance) of results.
- **Check Learning Curves:**
  - Detect overfitting or underfitting by observing training vs. validation performance.
- **Perform Error Analysis:**
  - Identify types of errors and reasons behind them.
- **Assess robustness:**
  - Slight variations in hyperparameters should not drastically affect performance.

---

9. Refine Hyperparameters (Iterative Process)

Hyperparameter tuning is often iterative:

- If optimal parameters are at boundaries, **expand your search space**.
- Use results from initial searches to **narrow down promising regions**.
- Run more refined searches around best parameters for precision optimization.

---

10. Final Validation on Test Set

Once hyperparameters are finalized:

- Evaluate final model performance on unseen test data.
- Confirm that chosen hyperparameters generalize effectively.

---

11. Document and Communicate Findings

Clearly document:

- Chosen hyperparameters and rationale.
- Results from different hyperparameter sets.
- Final model performance and comparison with baseline.
- Observations (convergence rates, computational efficiency, robustness).

Best Practices and Advanced Tips

- **Start simple, then increase complexity:** Always benchmark simpler models first.
- **Use automated hyperparameter tuning tools:** (e.g., Optuna, Hyperopt, Ray Tune).
- **Leverage domain knowledge:** Prior knowledge about the dataset or problem can inform your hyperparameter choices.
- **Consider constraints:** Balance between model accuracy, computational cost, inference time, and complexity.

Hyperparameters of Important ML Models

ML Model Type	Hyperparameter	Recommended Values / Ranges	Best Initial Choice	Comments & Tips
Linear Regression (Ridge/Lasso)	Alpha (Regularization Strength)	0.0001 – 10 (log scale)	1.0	Lower: less regularization; Higher: strong regularization
	Solver	`auto`, `svd`, `cholesky`, `saga`	`auto`	`auto` generally reliable
	Max iterations	100–10,000	1000	Increase if convergence warnings appear
Logistic Regression	C (Inverse Regularization Strength)	0.001 – 100 (log scale)	1.0	Smaller C → more regularization

	Penalty	`l1`, `l2`, `elasticnet`	`l2`	`l2` default, use `elasticnet` for hybrid regularization
	Solver	`lbfgs`, `liblinear`, `saga`	`lbfgs`	`saga` supports all penalties
Decision Trees	Max Depth	3–20	5	Increase depth carefully, higher risk of overfitting
	Min Samples Split	2–20	2 or 5	Higher → simpler tree
	Min Samples Leaf	1–20	1	Higher → smoother trees
Random Forest	Number of Estimators	50–500	100	More trees → better generalization but slower training
	Max Depth	5–30	10	Adjust based on validation error
	Max Features	`auto`, `sqrt`, `log2`	`sqrt`	`sqrt` generally recommended
Gradient Boosting (XGBoost/LightGBM)	Number of Estimators	100–1000	200	Larger values risk overfitting but may enhance accuracy
	Learning Rate	0.001–0.3	0.05	Lower values → slower but stable convergence
	Max Depth	3–12	6	Carefully tune, as depth significantly impacts performance

	Subsample	0.5–1.0	0.8	Prevents overfitting
Support Vector Machines (SVM)	Kernel	`linear`, `rbf`, `poly`, `sigmoid`	`rbf`	`rbf` usually performs best
	C (Regularization Parameter)	0.1–1000	1.0	Smaller C → more regularization
	Gamma (Kernel coefficient)	`scale`, `auto`, 0.0001–1	`scale`	`scale` typically robust
K-Nearest Neighbors (KNN)	Number of Neighbors (k)	3–30	5	Smaller → risk of noise; larger → smoother decision boundary
	Distance Metric	`euclidean`, `manhattan`, `minkowski`	`euclidean`	Euclidean is default and robust
Neural Networks (Deep Learning)	Learning Rate	0.00001–0.1	0.001	Crucial; lower values usually safer
	Number of Hidden Layers	1–10	2–3	More layers → complex model; risk overfitting
	Number of Neurons	32–1024 per layer	64–128 per layer	Adjust based on complexity of task
	Batch Size	16–512	32–64	Larger batch size → stable updates; smaller → noisier but faster convergence
	Activation Functions	`relu`, `tanh`, `sigmoid`	`relu`	`relu` recommended for hidden layers
Naive Bayes	Alpha (Laplace smoothing)	0.1–1.0	1.0	Smaller → sensitive to zero frequencies

K-Means Clustering	Number of Clusters (k)	2–20 (use Elbow method)	Determined by Elbow or Silhouette	Usually use Elbow/Silhouette method
	Initialization Method	`k-means++`, `random`	`k-means++`	Faster convergence, reliable initialization
Principal Component Analysis (PCA)	Number of Components	Explained variance: 90%–99%, or fixed number (2–50)	95% variance explained	Set based on desired explained variance
	Solver	`auto`, `full`, `randomized`	`auto`	Auto selects based on dataset size
DBSCAN	eps (Neighborhood Radius)	0.1–5.0 (highly dataset-specific)	Determined empirically	Sensitive; plot distances to choose best eps
	min_samples	3–10	5	Lower → more clusters, higher → fewer clusters