

Machine Learning Cookbook: Step-by-Step Guide for Training Supervised Models

Step 1: Split the Data

Before training a model, it's essential to properly prepare your dataset.

1. Separate Features (X) and Labels (y)

- **Features (X)**: All input columns used to make predictions.
- **Labels (y)**: The target variable you want to predict.

2. Choose a Splitting Strategy

Option A: Train / Validation / Test

Split	Purpose	Typical Size
Train Set	Used to train the model	70-80%
Validation	Tune hyperparameters and check performance	10-15%
Test Set	Final evaluation on unseen data	10-15%

! Important! Don't forget to use **stratify** when splitting classification dataset!!!

Option B: Train / Test + Cross-Validation

Split	Purpose	Typical Size
Train Set	Used to train the model	~80-90%
Test Set	Final evaluation after cross-validation	~10-20%

Cross-Validation Summary

Cross-validation provides a more robust estimate of model performance by splitting the training set into multiple folds:

- **K-Fold CV**: Divide the data into k parts. Train on $k-1$ parts and validate on the remaining one. Repeat k times.

- **Common k values:** 5 or 10.
 - **Stratified K-Fold:** Maintains class distribution (important for classification).
 - **Why Use It?**
 - Reduces overfitting risk.
 - Gives a more accurate estimate of how the model will perform on unseen data.
 - Better use of data, especially with smaller datasets.
-

Step 2: Handle Missing Data

Before feeding your data into a model, you must address missing values. There are several strategies depending on the nature and amount of missing data.

♦ Option 1: Drop Missing Data

- **Drop Rows:** If only a small portion of rows have missing values.
- **Drop Columns:** If an entire feature is mostly missing or unimportant.

⚠ Be cautious: Dropping too much data can lead to information loss or bias.

♦ Option 2: Impute Missing Data

Numeric Features

Method	Description
Mean Imputation	Replace missing values with the column mean. Ideal for normally distributed data.
Median Imputation	Use the column median. Better for skewed data.
Mode Imputation	Use the most frequent value (for discrete numeric variables).
KNN Imputation	Use values from the k-nearest neighbors based on feature similarity.
Model-Based Imputation	Use a regression model to predict missing values based on other features.

🔒 **Important:** If using **model-based imputation**, you must **remove the target variable y** before training the imputation model.

Otherwise, you introduce **data leakage** — the model could "cheat" by learning from future information.

Categorical Features

Method	Description
Mode Imputation	Replace with the most frequent category.
Random Imputation	Randomly sample from existing values in the column.
"Missing" Category	Add a new category like "Unknown" or "Missing" to preserve info.




Step 3: Normalize Features

Normalization ensures that all input features contribute equally to the model — especially important when features are on different scales.

Why Normalize?

- Prevents features with larger scales (e.g., income in \$\$\$) from dominating smaller ones (e.g., age).
- Helps optimization algorithms converge faster and more reliably.
- Improves model performance, especially for distance-based models (like KNN) and models using gradient descent.

When to Normalize

-  Features have different units or scales
-  You're using models sensitive to scale (e.g., KNN, Logistic Regression, Linear Regression)
-  Not needed for tree-based models (e.g., Decision Trees, Random Forest)

Common Normalization Techniques

Technique	Formula	Use When
Min-Max Scaling	$(x - \min) / (\max - \min)$	You want values between 0 and 1. Sensitive to outliers.
Z-Score (Standardization)	$(x - \text{mean}) / \text{std}$	You want normally distributed features. Robust to outliers.

Rules of Thumb

- **Normalize only the input features (X).** Never normalize the target variable (y).
- Always **fit the scaler on the training set only.**
- Apply the **same transformation to the validation and test sets** using the training set's parameters (to avoid data leakage).

Note on Imputation

If you're using **KNN imputation** or **model-based imputation (e.g., linear regression)** for missing values:

- **Normalize the relevant features before imputation** to ensure meaningful distances and accurate predictions.
- Make sure to **exclude the target variable y** during imputation to avoid data leakage.

Step 3.1: Unbalanced Dataset

Check whether the dataset is unbalanced.

If yes - decide on the strategy to tackle unbalanced data

1. Weighted Learning
2. Resampling
3. Combination of both

If resampling is needed, use one of these methods:

Method	Type	Description	When to Use	Key Libraries / Tools
Random Oversampling	Oversampling	Duplicates samples from the minority class to balance the dataset.	When you want a quick, simple approach without losing data.	<code>imblearn.over_sampling.RandomOverSampler</code>
SMOTE	Oversampling	Synthesizes new samples for the minority class using feature-space interpolation.	When you want more variety than duplication; works best with continuous features.	<code>imblearn.over_sampling.SMOTE</code>
SMOTENC	Oversampling	Variant of SMOTE that handles both categorical and numerical features.	When your dataset includes categorical features.	<code>imblearn.over_sampling.SMOTENC</code>
ADASYN	Oversampling	Focuses on generating synthetic data in difficult regions (near decision boundary).	When you want to emphasize "harder" examples.	<code>imblearn.over_sampling.ASYN</code>

Random Undersampling	Undersampling	Randomly removes samples from the majority class.	When your dataset is large and loss of data is acceptable.	<code>imblearn.under_sampling.RandomUnderSampler</code>
-----------------------------	---------------	---	--	---

! Note: Most resampling techniques (especially oversampling methods like SMOTE, ADASYN, SMOTENC) work by computing distances between samples in the feature space.

If your data isn't scaled before SMOTE, features with larger numeric ranges will dominate the distance calculation, leading to biased synthetic samples.



Best Practice (Pipeline Style)

In a pipeline, you should:

1. **Split** your data into train/test.
2. **Scale the training data first** (e.g., using `StandardScaler`).
3. **Apply SMOTE (or other distance-based resampler)** on the scaled data.
4. **Train your model** on the resampled + scaled data.

Apply the same scaler to the test set.

Step 4: Feature Selection & Feature Engineering

Carefully selecting and transforming features improves model performance and interpretability. This step covers common techniques for both regression and classification tasks, with notes on when and how to use each one.

◆ For Regression Models

1. Correlation Analysis

- Use **Pearson correlation** to assess linear relationships between features and the target.
- Remove highly correlated features to reduce **multicollinearity** and improve model robustness.

2. Distribution of the Target Variable

Some regression models (like linear regression) assume normally distributed residuals. If the target is skewed:

Transformation	Use When	Example
Log	Right-skewed data	Prices, salaries
Box-Cox	Any non-normal distribution	Optimizes normality
Square Root	Mild right skew	Counts

💡 *Note: it is possible to check whether the distribution is normal both by reviewing histplot or by plotting Q-Q plot.*

✅ Apply the **inverse transformation** on predictions when needed.

3. Feature Transformations

- **Logarithmic**: Handle skewed features
 - **Polynomial**: Add non-linear effects (e.g., x^2 , x^3)
 - **Binning**: Convert continuous values to categories (e.g., age → age group)
 - **Interaction Terms**: Combine features multiplicatively (e.g., $income * age$)
-

◆ For Classification Models

1. Log-Odds Interpretation (Logistic Regression)

- Coefficients represent the change in **log-odds** for a unit increase in a feature.
- Use **odds ratio = $\exp(\text{coefficient})$** for interpretation:
 - 1 → increases likelihood of positive class
 - <1 → decreases likelihood


2. Feature Transformations

- **Binarization**: Convert to binary indicators (e.g., is_premium_user)
 - **Scaling**: Apply normalization if needed (see Step 3)
 - **Frequency Encoding**: Replace categories with how frequently they occur
 - **Text to Numeric**: Use TF-IDF, bag-of-words, embeddings, etc.
-

Categorical Encoding (for Both Tasks)

Most models require numeric input, so categorical features must be encoded:

Encoding Method	Use Case	Notes
One-Hot Encoding	Non-ordinal categories (e.g., city, color)	Adds one column per category. Can explode dimensions
Ordinal Encoding	Ordered categories (e.g., education level: low → high)	Order must be meaningful
Target Encoding	High-cardinality categories (e.g., zip code, product ID)	Replace category with mean of target — watch for leakage!
Frequency Encoding	Categories with no inherent meaning	Replace category with frequency count

 For **target encoding**, always use cross-validation or hold-out sets to avoid data leakage.

Step 5: Model Selection

◆ Regression Models

Model	Description	How It Works
Linear Regression	Predicts continuous numerical values. Assumes a linear relationship between features and the target.	Learns weights w and bias b to minimize squared error: $y = w \cdot x + b$
KNN (K-Nearest Neighbors)	Predicts a value based on the average of the target values of the nearest data points.	For a given input, find the k closest training points (based on distance) and average their target values.
Decision Tree	Predicts continuous values by learning decision rules from data features.	Recursively splits the data based on feature thresholds to minimize variance in target values at each node. Predictions are made using the average target value in the leaf node.
Random Forest	An ensemble method that improves accuracy and reduces overfitting.	Trains multiple decision trees on random subsets of the data and features. Final prediction is the average of all tree predictions.
Gradient Boosting	An ensemble method that builds strong models by correcting errors of previous ones.	Trains trees sequentially. Each new tree is trained to predict the residuals (errors) of the previous trees. Final prediction is a weighted sum of all tree outputs.
Voting Regressor	Combines predictions from multiple regressors to improve performance.	Trains multiple models independently and averages their predictions to produce the final result.
Stacking Regressor	A meta-model that learns how to best combine predictions from base regressors.	Trains several base regressors and then fits a meta-regressor on their outputs (typically on validation data).

SVR (Support Vector Regression)	A margin-based regression model that can capture both linear and non-linear patterns.	Tries to fit the best line (or curve) within a margin of tolerance (epsilon) using support vectors. Can use kernels for non-linear regression.
--	---	--

◆ Classification Models

Model	Description	How It Works
Logistic Regression	Binary classifier that outputs probabilities. Suitable for linear decision boundaries.	Computes $z = w \cdot x + b$, then applies the sigmoid function to get a probability: $P(y=1) = 1 / (1 + e^{-z})$
KNN (K-Nearest Neighbors)	Predicts the class of a sample based on the most common class among its neighbors.	Find the k closest training samples. Use majority vote to assign a class label.
Decision Tree	Non-linear model that splits the data into regions using learned decision rules.	Recursively splits the data based on feature thresholds to minimize impurity (e.g., Gini, entropy). The final class is the majority class in the leaf node.
Random Forest	An ensemble of decision trees that improves generalization and reduces overfitting.	Trains multiple decision trees on random subsets of data and features. Each tree votes, and the majority vote determines the final class.
Gradient Boosting	Builds an ensemble of weak learners that sequentially correct errors.	Each new tree is trained to predict the classification errors (log-loss gradient) of previous models. Final prediction is based on weighted majority voting or probability aggregation.
Voting Classifier	Combines predictions from multiple classifiers to improve accuracy.	Trains multiple different models. Each model votes on the class label, and the majority (hard voting) or averaged probability (soft voting) determines the final prediction.
Stacking Classifier	A meta-model that learns how to best combine base model predictions.	Trains several base classifiers and then trains a meta-classifier on their outputs (usually on validation predictions) to make the final prediction.
SVM (Support Vector Machine)	A powerful classifier that finds the best decision boundary between classes.	Finds the hyperplane that maximizes the margin between classes. Can use kernels to separate classes in non-linear feature spaces.

↔ Key Differences Between Models

Model Type	Learns Parameters?	Interpretable?	Handles Non-Linearities ?	Sensitive to Scaling?
Linear Regression	✓ Yes	✓ High	✗ No	✓ Yes
Logistic Regression	✓ Yes	✓ High	✗ No	✓ Yes
KNN	✗ No (lazy learner)	⚠ Low	✓ Yes (non-parametric)	✓ Yes
Decision Tree	✓ Yes	✓ Medium/High	✓ Yes	✗ No
Random Forest	✓ Yes (many trees)	⚠ Medium	✓ Yes	✗ No
Gradient Boosting	✓ Yes (many trees)	⚠ Medium	✓ Yes	⚠ Sometimes
Voting (Ensemble)	✗ No (uses others)	✗ Low	✓ Depends on base models	⚠ Depends
Stacking	✓ Meta-learner	✗ Low	✓ Depends on base models	⚠ Depends
SVM (Linear/Kernels)	✓ Yes	⚠ Medium (linear) / ✗ Low (kernel)	✓ Yes (with kernels)	✓ Yes

Step 6: Train, Evaluate, Tune

Once you've prepared the data and selected a model, it's time to train it, evaluate its performance, and tune it for better results.

Training the Model

Training is the process of fitting the model to the training data by learning patterns that map features (**X**) to the target (**y**).

- **Analytical training:** Some models (e.g., Linear Regression) use closed-form mathematical solutions.
 - **Iterative training:** Other models (e.g., Logistic Regression, KNN) adjust parameters using optimization algorithms.
-

Model Parameters and Tuning

Each model comes with tunable hyperparameters that can affect performance. Here are key examples:

Model	Parameters	Notes	Default Params to Start With
Linear Regression	<code>fit_intercept</code> , <code>normalize</code> (deprecated), <code>copy_X</code> , <code>n_jobs</code> , <code>positive</code>	Basic model. Can be extended with Ridge/Lasso for regularization.	<code>fit_intercept=True</code> , <code>positive=False</code>
Logistic Regression	<code>C</code> , <code>penalty</code> , <code>solver</code> , <code>max_iter</code> , <code>class_weight</code> , <code>multi_class</code> , <code>l1_ratio</code>	<code>C</code> is inverse of regularization strength. Use <code>class_weight='balanced'</code> for imbalance.	<code>C=1.0</code> , <code>penalty='l2'</code> , <code>solver='lbfgs'</code> , <code>max_iter=100</code>
KNN	<code>n_neighbors</code> , <code>weights</code> , <code>metric</code> , <code>p</code> , <code>algorithm</code> , <code>leaf_size</code>	Try <code>distance</code> weight if classes are close. Use <code>p=1</code> for Manhattan distance.	<code>n_neighbors=5</code> , <code>weights='uniform'</code> , <code>metric='minkowski'</code>
Decision Tree	<code>max_depth</code> , <code>min_samples_split</code> , <code>min_samples_leaf</code> , <code>criterion</code> , <code>max_features</code> , <code>class_weight</code> , <code>random_state</code>	Control tree complexity to avoid overfitting.	<code>max_depth=None</code> , <code>criterion='gini'</code> , <code>random_state=0</code>
Random Forest	<code>n_estimators</code> , <code>max_depth</code> , <code>max_features</code> , <code>bootstrap</code> , <code>min_samples_split</code> , <code>class_weight</code> , <code>random_state</code>	Ensemble of trees with randomization to reduce variance.	<code>n_estimators=100</code> , <code>max_depth=None</code> , <code>random_state=0</code>
Gradient Boosting	<code>n_estimators</code> , <code>learning_rate</code> , <code>max_depth</code> , <code>subsample</code> , <code>min_samples_split</code> , <code>loss</code>	Low learning rate with more trees improves generalization.	<code>n_estimators=100</code> , <code>learning_rate=0.1</code> , <code>max_depth=3</code>

Voting Classifier/Regressor	<code>estimators, voting</code> (classifier), <code>weights</code> , <code>n_jobs</code> , <code>verbose</code>	Combine multiple models. Use soft voting for probabilistic averaging.	<code>voting='hard'</code> , pass 2–3 strong base models
Stacking Classifier/Regressor	<code>estimators</code> , <code>final_estimator</code> , <code>cv</code> , <code>passthrough</code> , <code>n_jobs</code> , <code>verbose</code>	Trains a meta-model on predictions of base learners.	<code>cv=5</code> , <code>final_estimator=LogisticRegression()</code>
SVM / SVR	<code>C</code> , <code>kernel</code> , <code>gamma</code> , <code>degree</code> , <code>coef0</code> , <code>shrinking</code> , <code>probability</code> (SVC), <code>epsilon</code> (SVR), <code>class_weight</code>	Use <code>rbf</code> for non-linear problems. Tune <code>C</code> and <code>gamma</code> together.	<code>C=1.0</code> , <code>kernel='rbf'</code> , <code>gamma='scale'</code>

Regularization

Regularization helps prevent overfitting by penalizing large model weights (coefficients). It reduces model complexity and improves generalization.

Regularization	Description	Effect
L1 (Lasso)	Shrinks some weights to exactly zero	Performs feature selection
L2 (Ridge)	Shrinks all weights but keeps them non-zero	Keeps all features, more stable for correlated inputs
Elastic Net	Combination of L1 and L2	Balances feature selection and weight shrinking

In **scikit-learn**, **C** controls regularization strength:

Smaller **C** → **stronger regularization**

Larger **C** → **weaker regularization**



Weighted Learning (especially for unbalanced classification dataset)

Weighted learning is a technique used to handle **imbalanced datasets** by assigning **different importance (weights)** to samples or classes during model training. Instead of changing the dataset itself (like resampling), the algorithm learns to pay more attention to minority classes by **penalizing their misclassification more heavily**.



Note, not all the algorithms support weighted learning

Model	Supports <code>class_weight</code>	Supports <code>sample_weight</code>	Notes
Logistic Regression	✓ Yes	✓ Yes	Use <code>class_weight='balanced'</code> or manual dict
SVM (SVC)	✓ Yes	✓ Yes	Scaling-sensitive; combine with normalization
Decision Tree	✓ Yes	✓ Yes	Works well with weighted learning
Random Forest	✓ Yes	✓ Yes	Use <code>class_weight</code> to handle imbalance
Gradient Boosting	✗ (native sklearn)	✓ Yes	Use <code>sample_weight</code> when calling <code>.fit()</code>
KNN	✗ No	✗ No	Doesn't support weighting internally

Underfitting vs Overfitting

Condition	Train Error	Test Error	Description
Underfitting	High	High	Model is too simple. Can't capture patterns.
Overfitting	Low	High	Model memorizes training data. Poor generalization.
Good Fit	Low	Low	Model generalizes well to new data.

Check training and validation curves (e.g., learning curves) to spot overfit/underfit.

Evaluation Metrics

Choose metrics based on the type of task:

Regression Metrics

Metric	Description	Preferred When
R²	Proportion of variance explained by the model	General goodness-of-fit
MAE	Mean of absolute errors	Interpretable, robust to outliers
RMSE	Square root of average squared errors	Penalizes large errors more than MAE

Classification Metrics

Metric	Description	Preferred When
Accuracy	% of correct predictions	When classes are balanced
Precision	$TP / (TP + FP)$	When False Positives are costly (e.g., spam)
Recall	$TP / (TP + FN)$	When False Negatives are costly (e.g., disease)
F1-Score	Harmonic mean of precision and recall	When balancing Precision and Recall is key
AUC-ROC	Probability model ranks positive > negative	For evaluating classifier confidence, thresholds

Step 7: Create a Reusable Pipeline and Store the Best Model

After training and evaluating different models, it's important to package everything into a consistent, reusable process. This ensures the model behaves the same way during both training and prediction — and can be reliably used in the future or in production.

Why Build a Pipeline?

A pipeline helps you **automate and standardize** the machine learning workflow. Instead of applying preprocessing and model training in separate steps, a pipeline combines them into one repeatable object.

Benefits:

- Ensures that the same preprocessing is applied during training and inference.
 - Reduces the risk of errors or forgetting steps when handling new data.
 - Makes it easier to deploy models into production environments.
 - Simplifies experimentation and model comparison.
-

What Should the Pipeline Include?

A well-structured pipeline typically includes:

1. **Imputation** – Filling in missing values (e.g., using mean, median, or mode).
 2. **Encoding** – Converting categorical variables into numeric form.
 3. **Normalization / Scaling** – Adjusting numeric feature values to the same scale.
 4. **Feature Selection (optional)** – Selecting only the most relevant features.
 5. **Model** – The machine learning model itself (e.g., Linear Regression, Logistic Regression, KNN).
-

Storing the Best Model

Once the model is trained and validated, you should **save it** so it can be reused for making predictions later without needing retraining.

Why **save the model**?

- Avoids retraining every time you want to use the model.
- Ensures consistency and reproducibility across sessions or deployments.
- Allows the model to be shared and integrated into applications (e.g., websites, APIs, automation tools).

✅ When saving a model, it's best to save the **entire pipeline**, not just the model. This way, all preprocessing steps are included, and raw input data can be passed directly to the saved pipeline for predictions.

Let me know if you'd like to move this into the doc now, or want to add a section about model versioning or deployment tips!