

# Cheat Sheet: Building Supervised Learning Models

## Common supervised learning models

Process Name	Brief Description	Code Syntax
One vs One classifier (using logistic regression)	<p><b>Process:</b> This method trains one classifier for each pair of classes.</p> <p><b>Key hyperparameters:</b></p> <ul style="list-style-type: none"> <li>- `estimator`: Base classifier (e.g., logistic regression)</li> </ul> <p><b>Pros:</b> Can work well for small datasets.</p> <p><b>Cons:</b> Computationally expensive for large datasets.</p> <p><b>Common applications:</b> Multiclass classification problems where the number of classes is relatively small.</p>	<pre>from sklearn.multiclass import OneVsOneClassifier from sklearn.linear_model import LogisticRegression model = OneVsOneClassifier(LogisticRegression())</pre>
One vs All classifier (using logistic regression)	<p><b>Process:</b> Trains one classifier per class, where each classifier distinguishes between one class and the rest.</p> <p><b>Key hyperparameters:</b></p> <ul style="list-style-type: none"> <li>- `estimator`: Base classifier (e.g., Logistic Regression)</li> <li>- `multi_class`: Strategy to handle multiclass classification ('ovr')</li> </ul> <p><b>Pros:</b> Simpler and more scalable than One vs One.</p> <p><b>Cons:</b> Less accurate for highly imbalanced classes.</p> <p><b>Common applications:</b> Common in multiclass classification problems such as image classification.</p>	<pre>from sklearn.multiclass import OneVsRestClassifier from sklearn.linear_model import LogisticRegression model = OneVsRestClassifier(LogisticRegression())</pre> <p>or</p> <pre>from sklearn.linear_model import LogisticRegression model_ova = LogisticRegression(multi_class='ovr')</pre>
Decision tree classifier	<p><b>Process:</b> A tree-based classifier that splits data into smaller subsets based on feature values.</p> <p><b>Key hyperparameters:</b></p> <ul style="list-style-type: none"> <li>- `max_depth`: Maximum depth of the tree</li> </ul> <p><b>Pros:</b> Easy to interpret and visualize.</p> <p><b>Cons:</b> Prone to overfitting if not pruned properly.</p> <p><b>Common applications:</b> Classification tasks, such as credit risk assessment.</p>	<pre>from sklearn.tree import DecisionTreeClassifier model = DecisionTreeClassifier(max_depth=5)</pre>
Decision tree regressor	<p><b>Process:</b> Similar to the decision tree classifier, but used for regression tasks to predict continuous values.</p> <p><b>Key hyperparameters:</b></p> <ul style="list-style-type: none"> <li>- `max_depth`: Maximum depth of the tree</li> </ul> <p><b>Pros:</b> Easy to interpret, handles nonlinear data.</p> <p><b>Cons:</b> Can overfit and perform poorly on noisy data.</p> <p><b>Common applications:</b> Regression tasks, such as predicting housing prices.</p>	<pre>from sklearn.tree import DecisionTreeRegressor model = DecisionTreeRegressor(max_depth=5)</pre>
Linear SVM classifier	<p><b>Process:</b> A linear classifier that finds the optimal hyperplane separating classes with a maximum margin.</p> <p><b>Key hyperparameters:</b></p> <ul style="list-style-type: none"> <li>- `C`: Regularization parameter</li> <li>- `kernel`: Type of kernel function ('linear', 'poly', 'rbf', etc.)</li> <li>- `gamma`: Kernel coefficient (only for 'rbf', 'poly', etc.)</li> </ul> <p><b>Pros:</b> Effective for high-dimensional spaces.</p> <p><b>Cons:</b> Not ideal for nonlinear problems without kernel tricks.</p> <p><b>Common applications:</b> Text classification and image recognition.</p>	<pre>from sklearn.svm import SVC model = SVC(kernel='linear', C=1.0)</pre>

Process Name	Brief Description	Code Syntax
K-nearest neighbors classifier	<p><b>Process:</b> Classifies data based on the majority class of its nearest neighbors.</p> <p><b>Key hyperparameters:</b></p> <ul style="list-style-type: none"> <li>- `n_neighbors`: Number of neighbors to use</li> <li>- `weights` : Weight function used in prediction ('uniform' or 'distance')</li> <li>- `algorithm`: Algorithm used to compute the nearest neighbors ('auto', 'ball_tree', 'kd_tree', 'brute')</li> </ul> <p><b>Pros:</b> Simple and effective for small datasets.</p> <p><b>Cons:</b> Computationally expensive as the dataset grows.</p> <p><b>Common applications:</b> Recommendation systems, image recognition.</p>	<pre>from sklearn.neighbors import KNeighborsClassifier model = KNeighborsClassifier(n_neighbors=5, weights='uniform')</pre>
Random Forest regressor	<p><b>Process:</b> An ensemble method using multiple decision trees to improve accuracy and reduce overfitting.</p> <p><b>Key hyperparameters:</b></p> <ul style="list-style-type: none"> <li>- `n_estimators`: Number of trees in the forest</li> <li>- `max_depth`: Maximum depth of each tree</li> </ul> <p><b>Pros:</b> Less prone to overfitting than individual decision trees.</p> <p><b>Cons:</b> Model complexity increases with the number of trees.</p> <p><b>Common applications:</b> Regression tasks such as predicting sales or stock prices.</p>	<pre>from sklearn.ensemble import RandomForestRegressor model = RandomForestRegressor(n_estimators=100, max_depth=5)</pre>
XGBoost regressor	<p><b>Process:</b> A gradient boosting method that builds trees sequentially to correct errors from previous trees.</p> <p><b>Key hyperparameters:</b></p> <ul style="list-style-type: none"> <li>- `n_estimators`: Number of boosting rounds</li> <li>- `learning_rate`: Step size to improve accuracy</li> <li>- `max_depth`: Maximum depth of each tree</li> </ul> <p><b>Pros:</b> High accuracy and works well with large datasets.</p> <p><b>Cons:</b> Computationally intensive, complex to tune.</p> <p><b>Common applications:</b> Predictive modeling, especially in Kaggle competitions.</p>	<pre>import xgboost as xgb model = xgb.XGBRegressor(n_estimators=100, learning_rate=0.1, max_depth=5)</pre>

## Associated functions used

Method Name	Brief Description	Code Syntax
OneHotEncoder	Transforms categorical features into a one-hot encoded matrix.	<pre>from sklearn.preprocessing import OneHotEncoder encoder = OneHotEncoder(sparse=False) encoded_data = encoder.fit_transform(categorical_data)</pre>
accuracy_score	Computes the accuracy of a classifier by comparing predicted and true labels.	<pre>from sklearn.metrics import accuracy_score accuracy = accuracy_score(y_true, y_pred)</pre>
LabelEncoder	Encodes labels (target variable) into numeric format.	<pre>from sklearn.preprocessing import LabelEncoder encoder = LabelEncoder() encoded_labels = encoder.fit_transform(labels)</pre>

Method Name	Brief Description	Code Syntax
plot_tree	Plots a decision tree model for visualization.	<pre>from sklearn.tree import plot_tree plot_tree(model, max_depth=3, filled=True)</pre>
normalize	Scales each feature to have zero mean and unit variance (standardization).	<pre>from sklearn.preprocessing import normalize normalized_data = normalize(data, norm='l2')</pre>
compute_sample_weight	Computes sample weights for imbalanced datasets.	<pre>from sklearn.utils.class_weight import compute_sample_weight weights = compute_sample_weight(class_weight='balanced', y=y)</pre>
roc_auc_score	Computes the Area Under the Receiver Operating Characteristic Curve (AUC-ROC) for binary classification models.	<pre>from sklearn.metrics import roc_auc_score auc = roc_auc_score(y_true, y_score)</pre>

## Author

[Jeff Grossman](#)  
[Abhishek Gagneja](#)



**Skills Network**