

Cheat Sheet: Building Supervised Learning Models

Common supervised learning models

Process Name	Brief Description	Code Syntax
One vs One classifier (using logistic regression)	Process: This method trains one classifier for each pair of classes. Key hyperparameters: - 'estimator': Base classifier (e.g., logistic regression) Pros: Can work well for small datasets. Cons: Computationally expensive for large datasets. Common applications: Multiclass classification problems where the number of classes is relatively small.	<pre>from sklearn.multiclass import OneVsOneClassifier from sklearn.linear_model import LogisticRegression model = OneVsOneClassifier(LogisticRegression())</pre>
One vs All classifier (using logistic regression)	Process: Trains one classifier per class, where each classifier distinguishes between one class and the rest. Key hyperparameters: - 'estimator': Base classifier (e.g., Logistic Regression) - 'multi_class': Strategy to handle multiclass classification ('ovr') Pros: Simpler and more scalable than One vs One. Cons: Less accurate for highly imbalanced classes. Common applications: Common in multiclass classification problems such as image classification.	<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	Process: A tree-based classifier that splits data into smaller subsets based on feature values. Key hyperparameters: - 'max_depth': Maximum depth of the tree Pros: Easy to interpret and visualize. Cons: Prone to overfitting if not pruned properly. Common applications: Classification tasks, such as credit risk assessment.	<pre>from sklearn.tree import DecisionTreeClassifier model = DecisionTreeClassifier(max_depth=5)</pre>
Decision tree regressor	Process: Similar to the decision tree classifier, but used for regression tasks to predict continuous values. Key hyperparameters: - 'max_depth': Maximum depth of the tree Pros: Easy to interpret, handles nonlinear data. Cons: Can overfit and perform poorly on noisy data. Common applications: Regression tasks, such as predicting housing prices.	<pre>from sklearn.tree import DecisionTreeRegressor model = DecisionTreeRegressor(max_depth=5)</pre>
Linear SVM classifier	Process: A linear classifier that finds the optimal hyperplane separating classes with a maximum margin. Key hyperparameters: - 'C': Regularization parameter - 'kernel': Type of kernel function ('linear', 'poly', 'rbf', etc.) - 'gamma': Kernel coefficient (only for 'rbf', 'poly', etc.) Pros: Effective for high-dimensional spaces. Cons: Not ideal for nonlinear problems without kernel tricks. Common applications: Text classification and image recognition.	<pre>from sklearn.svm import SVC model = SVC(kernel='linear', C=1.0)</pre>
K-nearest neighbors classifier	Process: Classifies data based on the majority class of its nearest neighbors. Key hyperparameters: - 'n_neighbors': Number of neighbors to use	<pre>from sklearn.neighbors import KNeighborsClassifier model = KNeighborsClassifier(n_neighbors=5, weights='uniform')</pre>

Process Name	Brief Description	Code Syntax
	<p>- <code>'weights'</code>: Weight function used in prediction (<code>'uniform'</code> or <code>'distance'</code>)</p> <p>- <code>'algorithm'</code>: Algorithm used to compute the nearest neighbors (<code>'auto'</code>, <code>'ball_tree'</code>, <code>'kd_tree'</code>, <code>'brute'</code>)</p> <p>Pros: Simple and effective for small datasets.</p> <p>Cons: Computationally expensive as the dataset grows.</p> <p>Common applications: Recommendation systems, image recognition.</p>	
Random Forest regressor	<p>Process: An ensemble method using multiple decision trees to improve accuracy and reduce overfitting.</p> <p>Key hyperparameters:</p> <p>- <code>'n_estimators'</code>: Number of trees in the forest</p> <p>- <code>'max_depth'</code>: Maximum depth of each tree</p> <p>Pros: Less prone to overfitting than individual decision trees.</p> <p>Cons: Model complexity increases with the number of trees.</p> <p>Common applications: 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>Process: A gradient boosting method that builds trees sequentially to correct errors from previous trees.</p> <p>Key hyperparameters:</p> <p>- <code>'n_estimators'</code>: Number of boosting rounds</p> <p>- <code>'learning_rate'</code>: Step size to improve accuracy</p> <p>- <code>'max_depth'</code>: Maximum depth of each tree</p> <p>Pros: High accuracy and works well with large datasets.</p> <p>Cons: Computationally intensive, complex to tune.</p> <p>Common applications: 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>
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>

Method Name	Brief Description	Code Syntax
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](#)

