# 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.<br>**Key hyperparameters:**<br>- `estimator`: Base classifier (e.g., logistic regression)<br>**Pros:** Can work well for small datasets.<br>**Cons:** Computationally expensive for large datasets.<br>**Common applications:** Multiclass classification problems where the number of classes is relatively small. | ```python
1. from sklearn.multiclass import OneVsOneClassifier
2. from sklearn.linear_model import LogisticRegression
3. model = OneVsOneClassifier(LogisticRegression())
``` |

| Process Name | Brief Description | Code Syntax |
|---|---|---|
| One vs All classifier (using logistic regression) | **Process:** Trains one classifier per class, where each classifier distinguishes between one class and the rest.<br>**Key hyperparameters:**<br>- `estimator`: Base classifier (e.g., Logistic Regression)<br>- `multi_class`: Strategy to handle multiclass classification (`ovr`)<br>**Pros:** Simpler and more scalable than One vs One.<br>**Cons:** Less accurate for highly imbalanced classes.<br>**Common applications:** Common in multiclass classification problems such as image classification. | ```
1. from sklearn.multiclass import
   OneVsRestClassifier
2. from sklearn.linear_model import
   LogisticRegression
3. model =
   OneVsRestClassifier(LogisticRegression())
```<br>or<br>```
1. from sklearn.linear_model import
   LogisticRegression
2. model_ova =
   LogisticRegression(multi_class='ovr')
``` |

| Process Name | Brief Description | Code Syntax |
|---|---|---|
| Decision tree classifier | **Process:** A tree-based classifier that splits data into smaller subsets based on feature values.<br>**Key hyperparameters:**<br>- `max_depth`: Maximum depth of the tree<br>**Pros:** Easy to interpret and visualize.<br>**Cons:** Prone to overfitting if not pruned properly.<br>**Common applications:** Classification tasks, such as credit risk assessment. | ```python\n1. from sklearn.tree import\n   DecisionTreeClassifier\n2. model = DecisionTreeClassifier(max_depth=5)\n``` |
| Decision tree regressor | **Process:** Similar to the decision tree classifier, but used for regression tasks to predict continuous values.<br>**Key hyperparameters:**<br>- `max_depth`: Maximum depth of the tree<br>**Pros:** Easy to interpret, handles nonlinear data.<br>**Cons:** Can overfit and perform poorly on noisy data.<br>**Common applications:** Regression tasks, such as predicting housing prices. | ```python\n1. from sklearn.tree import\n   DecisionTreeRegressor\n2. model = DecisionTreeRegressor(max_depth=5)\n``` |

| Process Name | Brief Description | Code Syntax |
|---|---|---|
| Linear SVM classifier | **Process:** A linear classifier that finds the optimal hyperplane separating classes with a maximum margin.<br>**Key hyperparameters:**<br>- `C`: Regularization parameter<br>- `kernel`: Type of kernel function (`linear`, `poly`, `rbf`, etc.)<br>- `gamma`: Kernel coefficient (only for `rbf`, `poly`, etc.)<br>**Pros:** Effective for high-dimensional spaces.<br>**Cons:** Not ideal for nonlinear problems without kernel tricks.<br>**Common applications:** Text classification and image recognition. | ```python<br>from sklearn.svm import SVC<br>model = SVC(kernel='linear', C=1.0)<br>``` |

| Process Name | Brief Description | Code Syntax |
|---|---|---|
| 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 - `weights`: Weight function used in prediction (`uniform` or `distance`) - `algorithm`: Algorithm used to compute the nearest neighbors (`auto`, `ball_tree`, `kd_tree`, `brute`) **Pros:** Simple and effective for small datasets. **Cons:** Computationally expensive as the dataset grows. **Common applications:** Recommendation systems, image recognition. | ```
1. from sklearn.neighbors import KNeighborsClassifier
2. model = KNeighborsClassifier(n_neighbors=5, weights='uniform')
``` |

| Process Name | Brief Description | Code Syntax |
|---|---|---|
| Random Forest regressor | **Process:** An ensemble method using multiple decision trees to improve accuracy and reduce overfitting.<br>**Key hyperparameters:**<br>- `n_estimators`: Number of trees in the forest<br>- `max_depth`: Maximum depth of each tree<br>**Pros:** Less prone to overfitting than individual decision trees.<br>**Cons:** Model complexity increases with the number of trees.<br>**Common applications:** Regression tasks such as predicting sales or stock prices. | ```python
1. from sklearn.ensemble import
   RandomForestRegressor
2. model =
   RandomForestRegressor(n_estimators=100,
   max_depth=5)
``` |

| Process Name | Brief Description | Code Syntax |
|---|---|---|
| XGBoost regressor | **Process:** A gradient boosting method that builds trees sequentially to correct errors from previous trees.<br>**Key hyperparameters:**<br>- `n_estimators`: Number of boosting rounds<br>- `learning_rate`: Step size to improve accuracy<br>- `max_depth`: Maximum depth of each tree<br>**Pros:** High accuracy and works well with large datasets.<br>**Cons:** Computationally intensive, complex to tune.<br>**Common applications:** Predictive modeling, especially in Kaggle competitions. | ```python<br>1. import xgboost as xgb<br>2. model = xgb.XGBRegressor(n_estimators=100,<br>   learning_rate=0.1, max_depth=5)``` |

## Associated functions used

| Method Name | Brief Description | Code Syntax |
|---|---|---|
| OneHotEncoder | Transforms categorical features into a one-hot encoded matrix. | ```python
1. from sklearn.preprocessing import OneHotEncoder
2. encoder = OneHotEncoder(sparse=False)
3. encoded_data = encoder.fit_transform(categorical_data)
``` |
| accuracy_score | Computes the accuracy of a classifier by comparing predicted and true labels. | ```python
1. from sklearn.metrics import accuracy_score
2. accuracy = accuracy_score(y_true, y_pred)
``` |
| LabelEncoder | Encodes labels (target variable) into numeric format. | ```python
1. from sklearn.preprocessing import LabelEncoder
2. encoder = LabelEncoder()
3. encoded_labels = encoder.fit_transform(labels)
``` |
| plot_tree | Plots a decision tree model for visualization. | ```python
1. from sklearn.tree import plot_tree
2. plot_tree(model, max_depth=3, filled=True)
``` |
| normalize | Scales each feature to have zero mean and unit variance (standardization). | ```python
1. from sklearn.preprocessing import normalize
2. normalized_data = normalize(data, norm='l2')
``` |

| Method Name | Brief Description | Code Syntax |
| --- | --- | --- |
| compute_sample_weight | Computes sample weights for imbalanced datasets. | ```python<br>from sklearn.utils.class_weight import compute_sample_weight<br>weights = compute_sample_weight(class_weight='balanced', y=y)<br>``` |
| roc_auc_score | Computes the Area Under the Receiver Operating Characteristic Curve (AUC-ROC) for binary classification models. | ```python<br>from sklearn.metrics import roc_auc_score<br>auc = roc_auc_score(y_true, y_score)<br>``` |