Scikit-learn Cheatsheet

Essential operations for machine learning and data science

This cheatsheet provides a quick reference to fundamental scikit-learn operations, algorithms, and best practices, ideal for both beginners and experienced data scientists for efficient machine learning workflows.

Data Preprocessing
Prepare and transform data

Model Selection
Choose and configure algorithms

Model Training
Fit models to data

Model Evaluation
Assess model performance

Feature Engineering
Extract and select features

Installation & Imports

Installation: `pip install scikit-learn`

Install scikit-learn and common dependencies.

```
# Install scikit-learn
pip install scikit-learn
# Install with additional packages
pip install scikit-learn pandas numpy matplotlib
# Upgrade to latest version
pip install scikit-learn --upgrade
```

Essential Imports

Standard imports for scikit-learn workflows.

```
# Core imports
import numpy as np
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from sklearn.metrics import accuracy_score,
classification_report

# Common algorithms
from sklearn.linear_model import LinearRegression,
LogisticRegression
from sklearn.ensemble import RandomForestClassifier,
GradientBoostingRegressor
from sklearn.svm import SVC
from sklearn.tree import DecisionTreeClassifier
```

Check Version

Verify your scikit-learn installation.

```
import sklearn
print(sklearn.__version__)
# Show build configuration
sklearn.show_versions()
```

Dataset Loading

Load built-in datasets for practice.

```
from sklearn.datasets import load_iris, load_boston,
make_classification
# Load sample datasets
iris = load_iris()
X, y = iris.data, iris.target
# Generate synthetic data
X_synth, y_synth = make_classification(n_samples=1000,
n_features=20, n_informative=10, random_state=42)
```

Data Preprocessing

Transform and prepare data for machine learning algorithms.

01

Train-Test Split: `train_test_split()`

Divide data into training and testing sets.

```
# Basic split (80% train, 20% test)
X_train, X_test, y_train, y_test =
train_test_split(X, y, test_size=0.2,
random_state=42)
# Stratified split for classification
X_train, X_test, y_train, y_test =
train_test_split(X, y, test_size=0.2,
stratify=y, random_state=42)
# Multiple splits
X_train, X_temp, y_train, y_temp =
train_test_split(X, y, test_size=0.4,
random_state=42)
X_val, X_test, y_val, y_test =
train_test_split(X_temp, y_temp,
test_size=0.5, random_state=42)
```

02

Feature Scaling: `StandardScaler()` / `MinMaxScaler()`

Normalize features to similar scales.

```
# Standardization (mean=0, std=1)
from sklearn.preprocessing import
StandardScaler, MinMaxScaler
scaler = StandardScaler()
X_scaled =
scaler.fit_transform(X_train)
X_test_scaled =
scaler.transform(X_test)

# Min-Max scaling (0-1 range)
minmax_scaler = MinMaxScaler()
X_minmax =
minmax_scaler.fit_transform(X_train)
X_test_minmax =
minmax_scaler.transform(X_test)
```

03

Encoding: `LabelEncoder()` / `OneHotEncoder()`

Convert categorical variables to numerical format.

```
# Label encoding for target variable
from sklearn.preprocessing import
LabelEncoder, OneHotEncoder
label_encoder = LabelEncoder()
y_encoded =
label_encoder.fit_transform(y)

# One-hot encoding for categorical
features
from sklearn.preprocessing import
OneHotEncoder
encoder =
OneHotEncoder(sparse=False,
drop='first')
X_encoded =
encoder.fit_transform(X_categorical)
# Get feature names
feature_names =
encoder.get_feature_names_out()
```

Supervised Learning - Classification

Logistic Regression: `LogisticRegression()`

Linear model for binary and multiclass classification.

```
# Basic logistic regression
from sklearn.linear_model import LogisticRegression
log_reg = LogisticRegression(random_state=42)
log_reg.fit(X_train, y_train)
y_pred = log_reg.predict(X_test)
y_proba = log_reg.predict_proba(X_test)

# With regularization
log_reg_l2 = LogisticRegression(C=0.1, penalty='l2')
log_reg_l1 = LogisticRegression(C=0.1, penalty='l1',
solver='liblinear')
```

Decision Tree: `DecisionTreeClassifier()`

Tree-based model for classification tasks.

```
# Decision tree classifier
from sklearn.tree import DecisionTreeClassifier
tree_clf = DecisionTreeClassifier(max_depth=5,
random_state=42)
tree_clf.fit(X_train, y_train)
y_pred = tree_clf.predict(X_test)

# Feature importance
importances = tree_clf.feature_importances_
# Visualize tree
from sklearn.tree import plot_tree
plot_tree(tree_clf, max_depth=3, filled=True)
```

Random Forest: `RandomForestClassifier()`

Ensemble method combining multiple decision trees.

```
# Random forest classifier
from sklearn.ensemble import RandomForestClassifier
rf_clf = RandomForestClassifier(n_estimators=100,
random_state=42)
rf_clf.fit(X_train, y_train)
y_pred = rf_clf.predict(X_test)

# Hyperparameter tuning
rf_clf = RandomForestClassifier(
    n_estimators=200,
    max_depth=10,
    min_samples_split=5,
    min_samples_leaf=2,
    random_state=42
)
```

Support Vector Machine: `SVC()`

Powerful classifier using kernel methods.

```
# SVM classifier
from sklearn.svm import SVC
svm_clf = SVC(kernel='rbf', C=1.0, gamma='scale',
random_state=42)
svm_clf.fit(X_train, y_train)
y_pred = svm_clf.predict(X_test)

# Different kernels
svm_linear = SVC(kernel='linear')
svm_poly = SVC(kernel='poly', degree=3)
svm_rbf = SVC(kernel='rbf', gamma=0.1)
```

Supervised Learning - Regression

Linear Regression: `LinearRegression()`

Basic linear model for continuous target variables.

```
# Simple linear regression
from sklearn.linear_model import LinearRegression
lin_reg = LinearRegression()
lin_reg.fit(X_train, y_train)
y_pred = lin_reg.predict(X_test)

# Get coefficients and intercept
coefficients = lin_reg.coef_
intercept = lin_reg.intercept_
print(f"R² score: {lin_reg.score(X_test, y_test)}")
```

Ridge Regression: `Ridge()`

Linear regression with L2 regularization.

```
# Ridge regression (L2 regularization)
from sklearn.linear_model import Ridge
ridge_reg = Ridge(alpha=1.0)
ridge_reg.fit(X_train, y_train)
y_pred = ridge_reg.predict(X_test)

# Cross-validation for alpha selection
from sklearn.linear_model import RidgeCV
ridge_cv = RidgeCV(alphas=[0.1, 1.0, 10.0])
ridge_cv.fit(X_train, y_train)
```

Lasso Regression: `Lasso()`

Linear regression with L1 regularization for feature selection.

```
# Lasso regression (L1 regularization)
from sklearn.linear_model import Lasso
lasso_reg = Lasso(alpha=0.1)
lasso_reg.fit(X_train, y_train)
y_pred = lasso_reg.predict(X_test)

# Feature selection (non-zero coefficients)
selected_features = X.columns[lasso_reg.coef_ != 0]
print(f"Selected features: {len(selected_features)}")
```

Random Forest Regression: `RandomForestRegressor()`

Ensemble method for regression tasks.

```
# Random forest regressor
from sklearn.ensemble import RandomForestRegressor
rf_reg = RandomForestRegressor(n_estimators=100,
random_state=42)
rf_reg.fit(X_train, y_train)
y_pred = rf_reg.predict(X_test)

# Feature importance for regression
feature_importance = rf_reg.feature_importances_
```

Model Evaluation

Assess model performance using various metrics.

Classification Metrics

Evaluate classification model performance.

```
# Basic accuracy
from sklearn.metrics import accuracy_score,
precision_score, recall_score, f1_score
accuracy = accuracy_score(y_test, y_pred)
precision = precision_score(y_test, y_pred,
average='weighted')
recall = recall_score(y_test, y_pred, average='weighted')
f1 = f1_score(y_test, y_pred, average='weighted')

# Detailed classification report
from sklearn.metrics import classification_report
print(classification_report(y_test, y_pred))

# Confusion matrix
from sklearn.metrics import confusion_matrix
cm = confusion_matrix(y_test, y_pred)
```

ROC Curve & AUC

Plot ROC curve and calculate Area Under Curve.

```
# ROC curve for binary classification
from sklearn.metrics import roc_curve, auc
fpr, tpr, thresholds = roc_curve(y_test, y_proba[:, 1])
roc_auc = auc(fpr, tpr)

# Plot ROC curve
import matplotlib.pyplot as plt
plt.figure(figsize=(8, 6))
plt.plot(fpr, tpr, label=f'ROC Curve (AUC = {roc_auc:.2f})')
plt.plot([0, 1], [0, 1], 'k--')
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')
plt.legend()
```

Regression Metrics

Evaluate regression model performance.

```
# Regression metrics
from sklearn.metrics import mean_squared_error,
mean_absolute_error, r2_score
mse = mean_squared_error(y_test, y_pred)
rmse = np.sqrt(mse)
mae = mean_absolute_error(y_test, y_pred)
r2 = r2_score(y_test, y_pred)

print(f"MSE: {mse:.4f}")
print(f"RMSE: {rmse:.4f}")
print(f"MAE: {mae:.4f}")
print(f"R²: {r2:.4f}")
```

Cross-Validation

Robust model evaluation using cross-validation.

```
# K-fold cross-validation
from sklearn.model_selection import cross_val_score,
StratifiedKFold
cv_scores = cross_val_score(model, X, y, cv=5,
scoring='accuracy')
print(f"CV Accuracy: {cv_scores.mean():.4f} (+/-
{cv_scores.std() * 2:.4f})")

# Stratified K-fold for imbalanced datasets
skf = StratifiedKFold(n_splits=5, shuffle=True,
random_state=42)
cv_scores = cross_val_score(model, X, y, cv=skf,
scoring='f1_weighted')
```

Unsupervised Learning

K-Means Clustering: `KMeans()`

Partition data into k clusters.

```
# K-means clustering
from sklearn.cluster import KMeans
kmeans = KMeans(n_clusters=3, random_state=42)
cluster_labels = kmeans.fit_predict(X)
centroids = kmeans.cluster_centers_

# Determine optimal number of clusters (Elbow method)
inertias = []
K_range = range(1, 11)
for k in K_range:
    kmeans = KMeans(n_clusters=k, random_state=42)
    kmeans.fit(X)
    inertias.append(kmeans.inertia_)
```

Principal Component Analysis: `PCA()`

Dimensionality reduction technique.

```
# PCA for dimensionality reduction
from sklearn.decomposition import PCA
pca = PCA(n_components=2)
X_pca = pca.fit_transform(X)
explained_variance = pca.explained_variance_ratio_

# Find optimal number of components
pca_full = PCA()
pca_full.fit(X)
cumsum =
np.cumsum(pca_full.explained_variance_ratio_)
# Find components for 95% variance
n_components = np.argmax(cumsum >= 0.95) + 1
```

DBSCAN Clustering: `DBSCAN()`

Density-based clustering algorithm.

```
# DBSCAN clustering
from sklearn.cluster import DBSCAN
dbscan = DBSCAN(eps=0.5, min_samples=5)
cluster_labels = dbscan.fit_predict(X)
n_clusters = len(set(cluster_labels)) - (1 if -1 in
cluster_labels else 0)
n_noise = list(cluster_labels).count(-1)

print(f"Number of clusters: {n_clusters}")
print(f"Number of noise points: {n_noise}")
```

Hierarchical Clustering: `AgglomerativeClustering()`

Build hierarchy of clusters.

```
# Agglomerative clustering
from sklearn.cluster import AgglomerativeClustering
agg_clustering = AgglomerativeClustering(n_clusters=3,
linkage='ward')
cluster_labels = agg_clustering.fit_predict(X)

# Dendrogram visualization
from scipy.cluster.hierarchy import dendrogram, linkage
linked = linkage(X, 'ward')
plt.figure(figsize=(12, 8))
dendrogram(linked)
```

Model Selection & Hyperparameter Tuning

Find the best model and optimize hyperparameters.

Grid Search: `GridSearchCV()`

Exhaustive search over parameter grid.

```
# Grid search for hyperparameter tuning
from sklearn.model_selection import GridSearchCV
param_grid = {
    'n_estimators': [100, 200, 300],
    'max_depth': [3, 5, 7, None],
    'min_samples_split': [2, 5, 10]
}
grid_search = GridSearchCV(
    RandomForestClassifier(random_state=42),
    param_grid, cv=5, scoring='accuracy', n_jobs=-1
)
grid_search.fit(X_train, y_train)
best_model = grid_search.best_estimator_
best_params = grid_search.best_params_
```

Random Search: `RandomizedSearchCV()`

Random sampling from parameter distributions.

```
# Randomized search (faster for large parameter spaces)
from sklearn.model_selection import
RandomizedSearchCV
from scipy.stats import randint
param_dist = {
    'n_estimators': randint(100, 500),
    'max_depth': [3, 5, 7, None],
    'min_samples_split': randint(2, 11)
}
random_search = RandomizedSearchCV(
    RandomForestClassifier(random_state=42),
    param_dist, n_iter=50, cv=5, scoring='accuracy',
n_jobs=-1, random_state=42
)
random_search.fit(X_train, y_train)
```

Pipeline: `Pipeline()`

Chain preprocessing and modeling steps.

```
# Create preprocessing and modeling pipeline
from sklearn.pipeline import Pipeline
pipeline = Pipeline([
    ('scaler', StandardScaler()),
    ('classifier', RandomForestClassifier(random_state=42))
])
pipeline.fit(X_train, y_train)
y_pred = pipeline.predict(X_test)

# Pipeline with grid search
param_grid = {
    'classifier__n_estimators': [100, 200],
    'classifier__max_depth': [3, 5, None]
}
grid_search = GridSearchCV(pipeline, param_grid, cv=5)
grid_search.fit(X_train, y_train)
```

Feature Selection: `SelectKBest()` / `RFE()`

Select the most informative features.

```
# Univariate feature selection
from sklearn.feature_selection import SelectKBest,
f_classif
selector = SelectKBest(score_func=f_classif, k=10)
X_selected = selector.fit_transform(X_train, y_train)
selected_features = X.columns[selector.get_support()]

# Recursive Feature Elimination
from sklearn.feature_selection import RFE
rfe = RFE(RandomForestClassifier(random_state=42),
n_features_to_select=10)
X_rfe = rfe.fit_transform(X_train, y_train)
```

Advanced Techniques

Sophisticated methods for complex machine learning tasks.

Ensemble Methods: `VotingClassifier()` / `BaggingClassifier()`

Combine multiple models for better performance.

```
# Voting classifier (ensemble of different algorithms)
from sklearn.ensemble import VotingClassifier
voting_clf = VotingClassifier(
    estimators=[
        ('lr', LogisticRegression(random_state=42)),
        ('rf', RandomForestClassifier(random_state=42)),
        ('svm', SVC(probability=True, random_state=42))
    ], voting='soft'
)
voting_clf.fit(X_train, y_train)
y_pred = voting_clf.predict(X_test)

# Bagging classifier
from sklearn.ensemble import BaggingClassifier
bagging_clf = BaggingClassifier(DecisionTreeClassifier(),
n_estimators=100, random_state=42)
bagging_clf.fit(X_train, y_train)
```

Gradient Boosting: `GradientBoostingClassifier()`

Sequential ensemble method with error correction.

```
# Gradient boosting classifier
from sklearn.ensemble import
GradientBoostingClassifier
gb_clf = GradientBoostingClassifier(n_estimators=100,
learning_rate=0.1, random_state=42)
gb_clf.fit(X_train, y_train)
y_pred = gb_clf.predict(X_test)

# Feature importance
importances = gb_clf.feature_importances_
# Learning curve
from sklearn.model_selection import learning_curve
train_sizes, train_scores, val_scores =
learning_curve(gb_clf, X, y, cv=5)
```

Handling Imbalanced Data: `SMOTE()` / Class Weights

Address class imbalance in datasets.

```
# Install imbalanced-learn: pip install imbalanced-learn
from imblearn.over_sampling import SMOTE
smote = SMOTE(random_state=42)
X_resampled, y_resampled = smote.fit_resample(X_train,
y_train)

# Using class weights
rf_balanced =
RandomForestClassifier(class_weight='balanced',
random_state=42)
rf_balanced.fit(X_train, y_train)

# Manual class weights
from sklearn.utils.class_weight import
compute_class_weight
class_weights = compute_class_weight('balanced',
classes=np.unique(y_train), y=y_train)
weight_dict = dict(zip(np.unique(y_train), class_weights))
```

Model Persistence: `joblib`

Save and load trained models.

```
# Save model
import joblib
joblib.dump(model, 'trained_model.pkl')
# Load model
loaded_model = joblib.load('trained_model.pkl')
y_pred = loaded_model.predict(X_test)

# Save entire pipeline
joblib.dump(pipeline, 'preprocessing_pipeline.pkl')
loaded_pipeline =
joblib.load('preprocessing_pipeline.pkl')

# Alternative using pickle
import pickle
with open('model.pkl', 'wb') as f:
    pickle.dump(model, f)
with open('model.pkl', 'rb') as f:
    loaded_model = pickle.load(f)
```

Performance & Debugging

Learning Curves: `learning_curve()`

Diagnose overfitting and underfitting.

```
# Plot learning curves
from sklearn.model_selection import learning_curve
train_sizes, train_scores, val_scores = learning_curve(
    model, X, y, cv=5, train_sizes=np.linspace(0.1, 1.0, 10)
)
plt.figure(figsize=(10, 6))
plt.plot(train_sizes, np.mean(train_scores, axis=1), 'o-',
label='Training Score')
plt.plot(train_sizes, np.mean(val_scores, axis=1), 'o-',
label='Validation Score')
plt.xlabel('Training Set Size')
plt.ylabel('Score')
plt.legend()
```

Validation Curves: `validation_curve()`

Analyze the effect of hyperparameters.

```
# Validation curve for single hyperparameter
from sklearn.model_selection import validation_curve
param_range = [10, 50, 100, 200, 500]
train_scores, val_scores = validation_curve(
    RandomForestClassifier(random_state=42), X, y,
    param_name='n_estimators',
param_range=param_range, cv=5
)
plt.figure(figsize=(10, 6))
plt.plot(param_range, np.mean(train_scores, axis=1), 'o-',
label='Training')
plt.plot(param_range, np.mean(val_scores, axis=1), 'o-',
label='Validation')
plt.xlabel('Number of Estimators')
plt.ylabel('Score')
```

Feature Importance Visualization

Understand which features drive model predictions.

```
# Plot feature importance
importances = model.feature_importances_
indices = np.argsort(importances)[::-1]
plt.figure(figsize=(12, 8))
plt.title("Feature Importance")
plt.bar(range(X.shape[1]), importances[indices])
plt.xticks(range(X.shape[1]), [X.columns[i] for i in indices],
rotation=90)

# SHAP values for model interpretability
# pip install shap
import shap
explainer = shap.TreeExplainer(model)
shap_values = explainer.shap_values(X_test)
shap.summary_plot(shap_values, X_test)
```

Model Comparison

Compare multiple algorithms systematically.

```
# Compare multiple models
from sklearn.model_selection import cross_val_score
models = {
    'Logistic Regression':
LogisticRegression(random_state=42),
    'Random Forest':
RandomForestClassifier(random_state=42),
    'SVM': SVC(random_state=42),
    'Gradient Boosting':
GradientBoostingClassifier(random_state=42)
}
results = {}
for name, model in models.items():
    scores = cross_val_score(model, X_train, y_train, cv=5,
scoring='accuracy')
    results[name] = scores.mean()
    print(f"{name}: {scores.mean():.4f} (+/- {scores.std() *
2:.4f})")
```

Configuration & Best Practices

Optimize workflow and maintain reproducible results.

Random State & Reproducibility

Ensure consistent results across runs.

```
# Set random state for
reproducibility
import numpy as np
np.random.seed(42)
# Set random_state in all
sklearn components
model =
RandomForestClassifier(random
_state=42)
train_test_split(X, y,
random_state=42)
# For cross-validation
cv = StratifiedKFold(n_splits=5,
shuffle=True, random_state=42)
```

Memory & Performance

Optimize for large datasets and computational efficiency.

```
# Use n_jobs=-1 for parallel
processing
model =
RandomForestClassifier(n_jobs=
-1)
grid_search =
GridSearchCV(model,
param_grid, n_jobs=-1)
# For large datasets, use
partial_fit when available
from sklearn.linear_model
import SGDClassifier
sgd = SGDClassifier()
# Process data in chunks
for chunk in chunks:
    sgd.partial_fit(chunk_X,
chunk_y)
```

Warnings & Debugging

Handle common issues and debug models.

```
# Suppress warnings (use
carefully)
import warnings
warnings.filterwarnings('ignore')
# Enable sklearn's set_config for
better debugging
from sklearn import set_config
set_config(display='diagram') #
Enhanced display in Jupyter
# Check for data leakage
from sklearn.model_selection
import cross_val_score
# Ensure preprocessing is done
inside CV loop
```

Reference: This cheatsheet covers essential scikit-learn commands and modern practices for efficient machine learning workflows in data science projects.