Jump-Start Scikit-Learn

Apply Machine Learning with Scikit-Learn Now

Jason Brownlee

MACHINE LEARNING MASTERY

Machine Learning Mastery
Web: http://MachineLearningMastery.com
Email: jason@MachineLearningMastery.com

Jump-Start Scikit-Learn
Apply Machine Learning with Scikit-Learn Now
by Jason Brownlee, PhD

Copyright © 2014 Jason Brownlee, All Rights Reserved.

Share this Guide
If you know someone who can benefit from this guide, just send them this link:

http://MachineLearningMastery.com

Table of Contents

Introduction

Welcome to the guide **Jump-Start Scikit-Learn**. You have taken that first step and decided that you want to learn machine learning bad enough that you are willing to invest in yourself.

I want to make sure you get the most out of that investment. I'm here to help. If you are struggling or need some advice, email me anytime at jason@MachineLearningMastery.com

This Path is Right For You!

This guide will teach you how to get up and running with the scikit-learn library for machine learning in Python.

This guide is for Python developers looking to get started or make the most of the scikit-learn library without getting bogged down with the mathematics and theory of the algorithms or having to implement each algorithm themselves.

Specifically, you are a Python developer that:

- Has or knows how to setup a Python development environment on a workstation.
- Knows their way around the SciPy stack, or can figure it out quickly.
- Has already installed or can figure out how to install scikit-learn.
- Is new to scikit-learn or is looking to get the most out of the library.
- Has some knowledge of machine learning or knows where to look to find some.
- This book is not intended to developers new to Python or SciPy.

Guide Overview

This guide provides standalone recipes in Python to use the scikit-learn library for specific use cases. A focus is given to classification and regression problems.

The recipes in this guide are divided into 6 sections:

- **What is scikit-learn**: An overview of the scikit-learn library including where it came from and the objectives of the project.
- **Handling Data**: Recipes for opening and loading datasets and preprocessing data ready for modeling.
- **Supervised Learning**: Recipes for common machine learning algorithms for classification and regression problems.
- **Regularization**: Recipes for machine learning algorithms that focus on regularization.
- **Ensemble Methods**: Recipes for machine learning algorithms that combine predictions from multiple weak models.
- **Advanced**: Recipes on subjects required when applying machine learning such as cross validation, model parameter tuning and feature selection.

Recipes use standard datasets such as the Iris flower dataset to demonstrate classification algorithms, and the diabetes dataset to demonstrate regression algorithms. Only the most common model parameters and model features are demonstrated and references are provided to learn about those parameters and features not described.

Each recipe has been tested with Python 2.7 and scikit-learn 0.14. These are the recommended versions, although the libraries are generally backwards compatible so you should be able to use all of the recipes with future versions.

How to Use This Guide

This book was designed for you to look-up like a reference or to browse for interesting methods to try.

The recipes in this book were designed for you to copy and paste (with the supplied source code) or to type in and have a working version of the library feature from which you can adapt and customize to your specific problem.

Below are some ways that you could use this guide:

- You want to use a specific common algorithm on your problem. You locate the recipe for that algorithm and copy-paste or type the examples into your project.
- You are looking for one or a few algorithms to try on your problem. You browse through the guide and select algorithms that can handle the supervised learning task you are working on and copy-paste or type the examples into your project.
- You are starting on a new problem and want to use scikit-learn. You copy the recipes for loading the dataset, feature selection, a powerful algorithm and cross validation and paste them into your project as a quick-start.
- You are new to scikit-learn and are interested in the specific capabilities of the library. You browse the recipes, trying a few examples as you do.

About This Guide

This guide will not teach you machine learning or about machine learning algorithms. There are plenty of excellent books available for that. Nevertheless, if you are new to machine learning, it will give you examples of algorithms and problems with which you can tinker.

Features

The scikit-learn documentation is great. It provides a tutorial, user manual and API documentation.

A problem that I had with the documentation is that the explanation of the library and its usage was fused with a terse introduction to machine learning. A given library feature may be demonstrated with an advanced or obscure case rather than the average case.

This recipe book was created because I saw the need for a simple and lightweight reference for using the scikit-learn library.

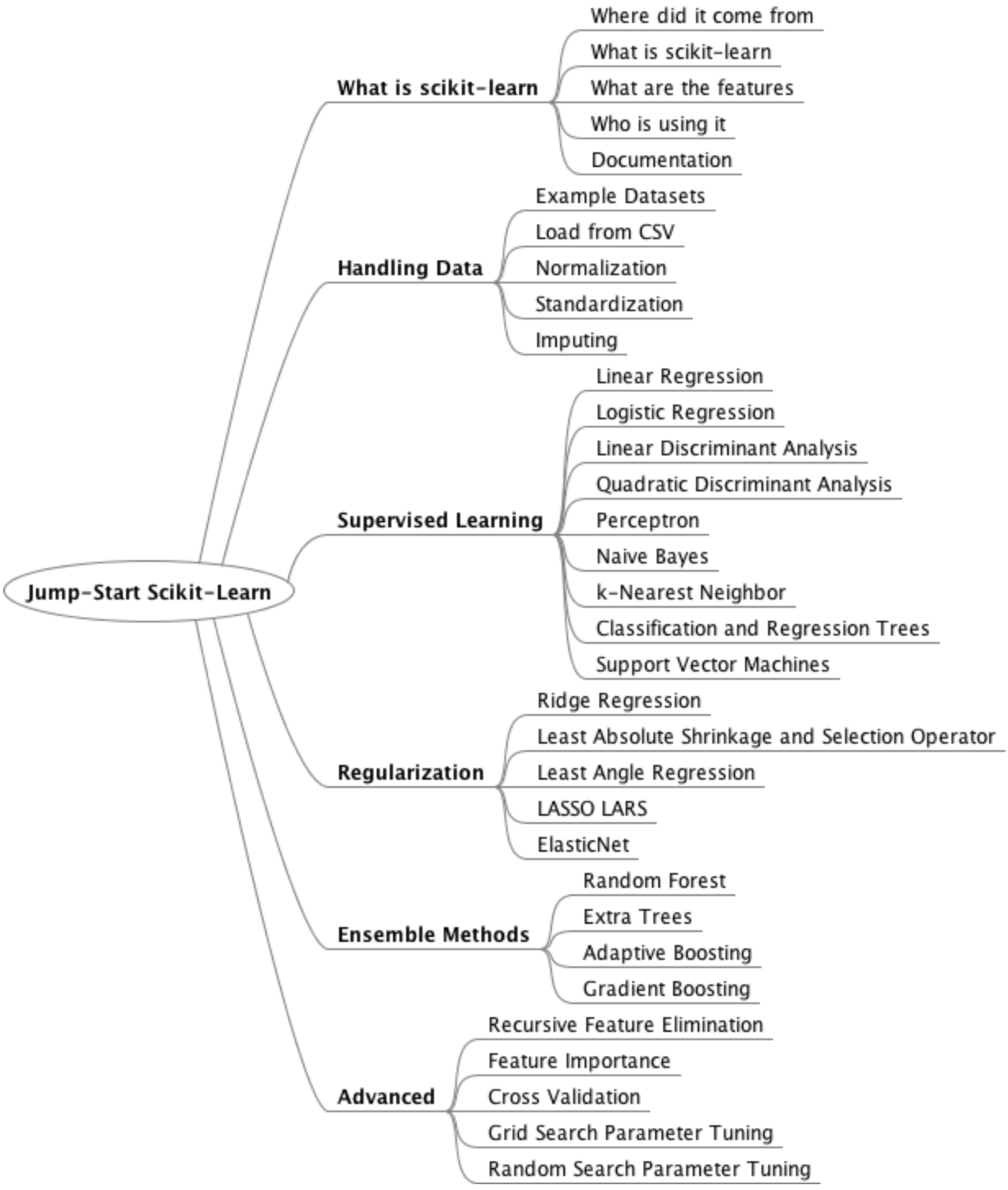
The goals of this guide are:

- **Standalone**: Each code example is a standalone, complete and executable recipe.
- **Just Code**: Focuses on the code with minimal exposition on machine learning theory.
- **Simplicity**: Recipes are presented in the most common use case, which is probably what you are looking to do.
- **Portable**: Recipes are provided in a single reference that can be searched and printed, browsed and looked up.
- **Consistent**: All code examples are presented consistently and follow the same code pattern and style conventions.

Benefits

Wield the scikit-learn library and solve complex problems.

- Apply algorithms and features directly.
- Discover the code you need.
- Understand what is going on with a glance.
- Own the recipes and use and organize them the way you want.
- Get the most out of the algorithms and features.


Jump-Start Scikit-Learn
What is scikit-learn
Where did it come from
What is scikit-learn
What are the features
Who is using it
Documentation
Handling Data
Example Datasets
Load from CSV
Normalization
Standardization
Imputing
Supervised Learning
Linear Regression
Logistic Regression
Linear Discriminant Analysis
Quadratic Discriminant Analysis
Perceptron
Naive Bayes
k-Nearest Neighbor
Classification and Regression Trees
Support Vector Machines
Regularization
Ridge Regression
Least Absolute Shrinkage and Selection Operator
Least Angle Regression
LASSO LARS
ElasticNet
Ensemble Methods
Random Forest
Extra Trees
Adaptive Boosting
Gradient Boosting
Advanced
Recursive Feature Elimination
Feature Importance
Cross Validation
Grid Search Parameter Tuning
Random Search Parameter Tuning

What is Scikit-Learn

If you are a Python programmer or you are looking for a robust library you can use to bring machine learning into a production system then a library that you will want to seriously consider is scikit-learn.

In this section you will find an overview of the scikit-learn library and useful references of where you can learn more.

Where did it come from?

Scikit-learn was initially developed by David Cournapeau as a Google summer of code project in 2007. Later Matthieu Brucher joined the project and started to use it as apart of his thesis work. In 2010 INRIA got involved and the first public release (v0.1 beta) was published in late January 2010.

The project now has more than 30 active contributors and has had paid sponsorship from INRIA, Google, Tinyclues and the Python Software Foundation.

What is scikit-learn?

Scikit-learn provides a range of supervised and unsupervised learning algorithms via a consistent interface in Python. It is licensed under a permissive simplified BSD license and is distributed under many Linux distributions, encouraging academic and commercial use.

The library is built upon the SciPy stack (Scientific Python) that must be installed before you can use scikit-learn. This is stack that includes:

- **NumPy**: Base n-dimensional array package
- **SciPy**: Fundamental library for scientific computing
- **Matplotlib**: Comprehensive 2D/3D plotting
- **IPython**: Enhanced interactive console
- **Sympy**: Symbolic mathematics
- **Pandas**: Data structures and analysis

Extensions or modules for SciPy are conventionally named SciKits. As such, the module provides learning algorithms and is named scikit-learn.

The vision for the library is a level of robustness and support required for use in production systems. This means a deep focus on concerns such as ease of use, code quality, collaboration, documentation and and performance.

Although the interface is Python, c-libraries are leveraged for performance such as numpy for arrays and matrix operations, LAPACK, LibSVM and the careful use of cython.

What are the features?

The library is focused on modeling data. It is not focused on loading, manipulating and summarizing data. For these features, refer to NumPy and Pandas.

Some popular groups of models provided by scikit-learn include:

- **Clustering**: for grouping unlabeled data such as KMeans.
- **Cross Validation**: for estimating the performance of supervised models on unseen data.
- **Datasets**: for test datasets and for generating datasets with specific properties for investigating model behaviour.
- **Dimensionality Reduction**: for reducing the number of attributes in data for summarization, visualization and feature selection.
- **Ensemble methods**: for combining the predictions of multiple supervised models.
- **Feature extraction**: for defining attributes in image and text data.
- **Feature selection**: for identifying meaningful attributes from which to create supervised models.
- **Parameter Tuning**: for getting the most out of supervised models.
- **Manifold Learning**: for summarizing and depicting complex multi-dimensional data.
- **Supervised Models**: a vast array not limited to generalized linear models, discriminant analysis, naive bayes, lazy methods, neural networks, support vector machines and decision trees.

Who is using scikit-learn?

The scikit-learn testimonials page lists Inria, Mendeley, wise.io , Evernote, Telecom ParisTech and AWeber as users of the library. If this is a small indication of companies that have presented on their use, then there are very likely tens to hundreds of large organizations using the library.

It has good test coverage and managed releases and is suitable for prototype and production projects alike.

Handling Data

Data is key. It contains the hidden truth about the problem you are trying to solve. You need to collect enough if it and prepare it suitably for the algorithms and methods you plan to use. The recipes in this section focus on the loading and manipulation of data files.

Example Datasets

The scikit-learn library is packaged with five datasets. These datasets are useful for getting a handle on a given machine learning algorithm or library feature before using it in your own work.

This recipe demonstrates how to load each of the five packaged datasets. The Iris and Diabetes datasets from this recipe are used throughout the recipes in this guide.

```
# Load and display details of the packaged datasets
from sklearn.datasets import load_boston
from sklearn.datasets import load_iris
from sklearn.datasets import load_diabetes
from sklearn.datasets import load_digits
from sklearn.datasets import load_linnerud
# Boston house prices dataset (13x506, reals, regression)
boston = load_boston()
print(boston.data.shape)
# Iris flower dataset (4x150, reals, multi-label classification)
iris = load_iris()
print(iris.data.shape)
# Diabetes dataset (10x442, reals, regression)
diabetes = load_diabetes()
print(diabetes.data.shape)
# Hand-written digit dataset (64x1797, multi-label classification)
digits = load_digits()
print(digits.data.shape)
# Linnerud psychological and exercise dataset (3x20,3x20 multivariate regression)
linnerud = load_linnerud()
print(linnerud.data.shape)
```

Load from CSV

It is very common for you to have a dataset as a CSV file on your local workstation or on a remote server. This recipe show you how to load a CSV file from a URL, in this case the Pima Indians diabetes classification dataset from the UCI Machine Learning Repository. From the prepared *X* and *y* variables, you can train a machine learning model.

```
# Load the Pima Indians diabetes dataset from CSV URL
import numpy as np
import urllib
# URL for the Pima Indians Diabetes dataset (UCI Machine Learning Repository)
url = "http://goo.gl/j0Rvxq"
```

```
# download the file
raw_data = urllib.urlopen(url)
# load the CSV file as a numpy matrix
dataset = np.loadtxt(raw_data, delimiter=",")
print(dataset.shape)
# separate the data from the target attributes
X = dataset[:,0:7]
y = dataset[:,8]
```

Normalization

Normalization refers to rescaling real valued numeric attributes into the range 0 and 1. It is useful to scale the input attributes for a model that relies on the magnitude of values, such as distance measures used in k-nearest neighbors and in the preparation of coefficients in regression.

```
# Normalize the data attributes for the Iris dataset.
from sklearn.datasets import load_iris
from sklearn import preprocessing
# load the iris dataset
iris = load_iris()
print(iris.data.shape)
# separate the data from the target attributes
X = iris.data
y = iris.target
# normalize the data attributes
normalized_X = preprocessing.normalize(X)
```

Standardization

Standardization refers to shifting the distribution of each attribute to have a mean of 0 and a standard deviation of 1. It is useful to standardize attributes for a model that relies on the distribution of attributes such as Gaussian processes.

```
# Standardize the data attributes for the Iris dataset.
from sklearn.datasets import load_iris
from sklearn import preprocessing
# load the Iris dataset
iris = load_iris()
print(iris.data.shape)
# separate the data and target attributes
X = iris.data
y = iris.target
# standardize the data attributes
standardized_X = preprocessing.scale(X)
```

Imputing

Data can have missing values. These are values for attributes where a measurement could not be taken or is corrupt for some reason. It is important to identify and mark this missing data. Once marked, replacement values can be prepared. This is useful because some algorithms are unable to work with or exploit missing data.

This recipe demonstrates marking 0 values from the Pima Indians dataset as NaN and imputing the missing values with the mean of the attribute.

```
# Mark 0 values as missing and impute with the mean
import numpy as np
import urllib
from sklearn.preprocessing import Imputer
# Load the Pima Indians Diabetes dataset
url = "http://goo.gl/j0Rvxq"
raw_data = urllib.urlopen(url)
dataset = np.loadtxt(raw_data, delimiter=",")
print(dataset.shape)
# separate the data and target attributes
X = dataset[:,0:7]
y = dataset[:,8]
# Mark all zero values as 0
X[X==0]=np.nan
# Impute all missing values with the mean of the attribute
imp = Imputer(missing_values='NaN', strategy='mean')
imputed_X = imp.fit_transform(X)
```

Supervised Learning

Supervised learning refers to the modeling of a relationship (actually an underlying mathematical function) between observations and outcome. Both classification of instances to a category and predicting a numeric value for an instance (regression) are examples of supervised learning problems.

This section provides recipes for common supervised machine learning algorithms on classification and regression problems using the datasets provided with scikit-learn as examples.

Linear Regression

Linear regression fits a linear model (such as a line in two dimensions) to the data.

This recipes shows the fitting of a linear regression algorithm on the diabetes dataset.

```
# Linear Regression
import numpy as np
from sklearn import datasets
from sklearn.linear_model import LinearRegression
# load the diabetes datasets
dataset = datasets.load_diabetes()
# fit a linear regression model to the data
model = LinearRegression()
model.fit(dataset.data, dataset.target)
print(model)
# make predictions
expected = dataset.target
predicted = model.predict(dataset.data)
# summarize the fit of the model
mse = np.mean((predicted-expected)**2)
print(mse)
print(model.score(dataset.data, dataset.target))
```

Logistic Regression

Logistic regression fits a logistic model to data and makes predictions about the probability of an event (between 0 and 1).

This recipe shows the fitting of a logistic regression algorithm to the iris dataset. Because this is a mutliclass classification problem and logistic regression makes predictions between 0 and 1, a one-vs-all scheme is used (one model is created per class).

```
# Logistic Regression
from sklearn import datasets
from sklearn import metrics
```

```
from sklearn.linear_model import LogisticRegression
# load the iris datasets
dataset = datasets.load_iris()
# fit a logistic regression model to the data
model = LogisticRegression()
model.fit(dataset.data, dataset.target)
print(model)
# make predictions
expected = dataset.target
predicted = model.predict(dataset.data)
# summarize the fit of the model
print(metrics.classification_report(expected, predicted))
print(metrics.confusion_matrix(expected, predicted))
```

Linear Discriminant Analysis

Linear Discriminate Analysis (LDA) fits a conditional probability density function (Gaussian) to each attribute for the class, the discrimination function between the classes is linear.

This recipe shows the fitting of an LDA algorithm to the iris dataset.

```
# Linear Discriminant Analysis
from sklearn import datasets
from sklearn import metrics
from sklearn.lda import LDA
# load the iris datasets
dataset = datasets.load_iris()
# fit a LDA model to the data
model = LDA()
model.fit(dataset.data, dataset.target)
print(model)
# make predictions
expected = dataset.target
predicted = model.predict(dataset.data)
# summarize the fit of the model
print(metrics.classification_report(expected, predicted))
print(metrics.confusion_matrix(expected, predicted))
```

Quadratic Discriminant Analysis

Quadratic Discriminant Analysis (QDA) fits a conditional probability density function (Gaussian) to each attribute for the class, the discrimination function between the classes is quadratic.

This recipe shows the fitting of an QDA algorithm to the iris dataset.

```
# Quadratic Discriminant Analysis
from sklearn import datasets
from sklearn import metrics
```

```
from sklearn.qda import QDA
# load the iris datasets
dataset = datasets.load_iris()
# fit a QDA model to the data
model = QDA()
model.fit(dataset.data, dataset.target)
print(model)
# make predictions
expected = dataset.target
predicted = model.predict(dataset.data)
# summarize the fit of the model
print(metrics.classification_report(expected, predicted))
print(metrics.confusion_matrix(expected, predicted))
```

Perceptron

The Perceptron is a primitive type of neural network that learns weights for input attributes and transfers the weighted inputs into a network output or prediction.

This recipes shows the fitting of a Perceptron algorithm on the diabetes dataset.

```
# Perceptron
import numpy as np
from sklearn import datasets
from sklearn.linear_model import Perceptron
# load the diabetes datasets
dataset = datasets.load_diabetes()
# fit a Perceptron model to the data
model = Perceptron()
model.fit(dataset.data, dataset.target)
print(model)
# make predictions
expected = dataset.target
predicted = model.predict(dataset.data)
# summarize the fit of the model
mse = np.mean((predicted-expected)**2)
print(mse)
print(model.score(dataset.data, dataset.target))
```

Naive Bayes

Naive Bayes uses Bayes Theorem to model the conditional relationship of each attribute to the class variable.

This recipe shows the fitting of an Naive Bayes algorithm to the iris dataset.

```
# Gaussian Naive Bayes
from sklearn import datasets
```

```
from sklearn import metrics
from sklearn.naive_bayes import GaussianNB
# load the iris datasets
dataset = datasets.load_iris()
# fit a Naive Bayes model to the data
model = GaussianNB()
model.fit(dataset.data, dataset.target)
print(model)
# make predictions
expected = dataset.target
predicted = model.predict(dataset.data)
# summarize the fit of the model
print(metrics.classification_report(expected, predicted))
print(metrics.confusion_matrix(expected, predicted))
```

k-Nearest Neighbor

The k-Nearest Neighbor (kNN) method makes predictions by locating similar cases to a given data instance (using a similarity function) and returning the average or majority of the most similar data instances. The kNN algorithm can be used for classification or regression.

This recipe shows use of the kNN algorithm to make predictions for the iris dataset (classification).

```
# k-Nearest Neighbor Classification
from sklearn import datasets
from sklearn import metrics
from sklearn.neighbors import KNeighborsClassifier
# load iris the datasets
dataset = datasets.load_iris()
# fit a k-nearest neighbor model to the data
model = KNeighborsClassifier()
model.fit(dataset.data, dataset.target)
print(model)
# make predictions
expected = dataset.target
predicted = model.predict(dataset.data)
# summarize the fit of the model
print(metrics.classification_report(expected, predicted))
print(metrics.confusion_matrix(expected, predicted))
```

This recipe shows use of the kNN algorithm to make predictions for the diabetes dataset (regression).

```
# k-Nearest Neighbor Regression
import numpy as np
from sklearn import datasets
from sklearn.neighbors import KNeighborsRegressor
```

```
# load the diabetes datasets
dataset = datasets.load_diabetes()
# fit a model to the data
model = KNeighborsRegressor()
model.fit(dataset.data, dataset.target)
print(model)
# make predictions
expected = dataset.target
predicted = model.predict(dataset.data)
# summarize the fit of the model
mse = np.mean((predicted-expected)**2)
print(mse)
print(model.score(dataset.data, dataset.target))
```

Classification and Regression Trees

Classification and Regression Trees (CART) are constructed from a dataset by making splits that best separate the data for the classes or predictions being made. The CART algorithm can be used for classification or regression.

This recipe shows use of the CART algorithm to make predictions for the iris dataset (classification).

```
# Decision Tree Classifier
from sklearn import datasets
from sklearn import metrics
from sklearn.tree import DecisionTreeClassifier
# load the iris datasets
dataset = datasets.load_iris()
# fit a CART model to the data
model = DecisionTreeClassifier()
model.fit(dataset.data, dataset.target)
print(model)
# make predictions
expected = dataset.target
predicted = model.predict(dataset.data)
# summarize the fit of the model
print(metrics.classification_report(expected, predicted))
print(metrics.confusion_matrix(expected, predicted))
```

This recipe shows use of the CART algorithm to make predictions for the diabetes dataset (regression).

```
# Decision Tree Regression
import numpy as np
from sklearn import datasets
from sklearn.tree import DecisionTreeRegressor
# load the diabetes datasets
```

```
dataset = datasets.load_diabetes()
# fit a CART model to the data
model = DecisionTreeRegressor()
model.fit(dataset.data, dataset.target)
print(model)
# make predictions
expected = dataset.target
predicted = model.predict(dataset.data)
# summarize the fit of the model
mse = np.mean((predicted-expected)**2)
print(mse)
print(model.score(dataset.data, dataset.target))
```

Support Vector Machines

Support Vector Machines (SVM) are a method that uses points in a transformed problem space that best separate classes into two groups. Classification for multiple classes is supported by a one-vs-all method. SVM also supports regression by modeling the function with a minimum amount of allowable error.

This recipe shows use of the SVM algorithm to make predictions for the iris dataset (classification).

```
# SVM Classification
from sklearn import datasets
from sklearn import metrics
from sklearn.svm import SVC
# load the iris datasets
dataset = datasets.load_iris()
# fit a SVM model to the data
model = SVC()
model.fit(dataset.data, dataset.target)
print(model)
# make predictions
expected = dataset.target
predicted = model.predict(dataset.data)
# summarize the fit of the model
print(metrics.classification_report(expected, predicted))
print(metrics.confusion_matrix(expected, predicted))
```

This recipe shows use of the SVM algorithm to make predictions for the diabetes dataset (regression).

```
# SVM Regression
import numpy as np
from sklearn import datasets
from sklearn.svm import SVR
# load the diabetes datasets
```

```
dataset = datasets.load_diabetes()
# fit a SVM model to the data
model = SVR()
model.fit(dataset.data, dataset.target)
print(model)
# make predictions
expected = dataset.target
predicted = model.predict(dataset.data)
# summarize the fit of the model
mse = np.mean((predicted-expected)**2)
print(mse)
print(model.score(dataset.data, dataset.target))
```

Regularization

Regularization refers to methods that decrease over-fitting of a model, most commonly by introducing a penalty proportional to model complexity. It is most common to demonstrate regularization methods using regression algorithms as the base model.

This section provides recipes for regularization regression algorithms.

Ridge Regression

Ridge Regression (also known as Tikhonov regularization) penalizes a least squares regression model on the square of the absolute magnitude of the coefficients (called the L2 norm).

This recipe shows use of the Ridge Regression algorithm to make predictions for the diabetes dataset.

```
# Ridge Regression
import numpy as np
from sklearn import datasets
from sklearn.linear_model import Ridge
# load the diabetes datasets
dataset = datasets.load_diabetes()
# fit a ridge regression model to the data
model = Ridge(alpha=0.1)
model.fit(dataset.data, dataset.target)
print(model)
# make predictions
expected = dataset.target
predicted = model.predict(dataset.data)
# summarize the fit of the model
mse = np.mean((predicted-expected)**2)
print(mse)
print(model.score(dataset.data, dataset.target))
```

Least Absolute Shrinkage and Selection Operator

The Least Absolute Shrinkage and Selection Operator (LASSO) is a regularization method that penalizes a least squares regression model on the absolute magnitude of the coefficients (called the L1 norm).

This recipe shows use of the LASSO algorithm to make predictions for the diabetes dataset.

```
# Lasso Regression
import numpy as np
from sklearn import datasets
from sklearn.linear_model import Lasso
# load the diabetes datasets
```

```
dataset = datasets.load_diabetes()
# fit a LASSO model to the data
model = Lasso(alpha=0.1)
model.fit(dataset.data, dataset.target)
print(model)
# make predictions
expected = dataset.target
predicted = model.predict(dataset.data)
# summarize the fit of the model
mse = np.mean((predicted-expected)**2)
print(mse)
print(model.score(dataset.data, dataset.target))
```

Least Angle Regression

The Least Angle Regression (LARS) method is a computationally efficient algorithm for fitting a regression model. It is useful for high-dimensional data and is commonly used in conjunction with regularization (such as LASSO).

This recipe shows use of the LARS algorithm to make predictions for the diabetes dataset.

```
# LARS Regression
import numpy as np
from sklearn import datasets
from sklearn.linear_model import Lars
# load the diabetes datasets
dataset = datasets.load_diabetes()
# fit a LARS model to the data
model = Lars()
model.fit(dataset.data, dataset.target)
print(model)
# make predictions
expected = dataset.target
predicted = model.predict(dataset.data)
# summarize the fit of the model
mse = np.mean((predicted-expected)**2)
print(mse)
print(model.score(dataset.data, dataset.target))
```

LASSO LARS

The Least Angle Regression (LARS) can be used as an alternative method for calculating Least Absolute Shrinkage and Selection Operator (LASSO) fit.

This recipe shows use of the LASSO with LARS algorithm to make predictions for the diabetes dataset.

```
# LassoLars Regression
import numpy as np
from sklearn import datasets
from sklearn.linear_model import LassoLars
# load the iris datasets
dataset = datasets.load_diabetes()
# fit a LASSO using LARS model to the data
model = LassoLars(alpha=0.1)
model.fit(dataset.data, dataset.target)
print(model)
# make predictions
expected = dataset.target
predicted = model.predict(dataset.data)
# summarize the fit of the model
mse = np.mean((predicted-expected)**2)
print(mse)
print(model.score(dataset.data, dataset.target))
```

ElasticNet

The ElasticNet is a regularized form of regression that penalizes the model with both the L1 norm and the L2 norm.

This recipe shows use of the ElasticNet algorithm to make predictions for the diabetes dataset.

```
# ElasticNet Regression
import numpy as np
from sklearn import datasets
from sklearn.linear_model import ElasticNet
# load the diabetes datasets
dataset = datasets.load_diabetes()
# fit a model to the data
model = ElasticNet(alpha=0.1)
model.fit(dataset.data, dataset.target)
print(model)
# make predictions
expected = dataset.target
predicted = model.predict(dataset.data)
# summarize the fit of the model
mse = np.mean((predicted-expected)**2)
print(mse)
print(model.score(dataset.data, dataset.target))
```

Ensemble Methods

Ensemble methods take the predictions from two or more models and combine them into a single prediction. In this way, ensemble methods are models of models or meta-models. They are a powerful set of techniques for getting better results from existing models. Decision trees like CART and randomly created trees are a popular base model for creating ensembles.

This section provides recipes for ensemble algorithms.

Random Forest

The Random Forest ensemble method creates a number of decision trees, each on a different subset of the dataset. Generally the approach to creating an ensemble used by random forest is called bootstrap aggregation (or bagging for short).

This recipe shows use of the Random Forest algorithm to make predictions for the iris dataset (classification).

```
# Random Forest Classification
from sklearn import datasets
from sklearn import metrics
from sklearn.ensemble import RandomForestClassifier
# load iris the datasets
dataset = datasets.load_iris()
# fit a random forest model to the data
model = RandomForestClassifier()
model.fit(dataset.data, dataset.target)
print(model)
# make predictions
expected = dataset.target
predicted = model.predict(dataset.data)
# summarize the fit of the model
print(metrics.classification_report(expected, predicted))
print(metrics.confusion_matrix(expected, predicted))
```

This recipe shows use of the Random Forest model to make predictions for the diabetes dataset (regression).

```
# Random Forest Regression
import numpy as np
from sklearn import datasets
from sklearn.ensemble import RandomForestRegressor
# load the diabetes datasets
dataset = datasets.load_diabetes()
# fit a random forest model to the data
model = RandomForestRegressor()
model.fit(dataset.data, dataset.target)
print(model)
```

```
# make predictions
expected = dataset.target
predicted = model.predict(dataset.data)
# summarize the fit of the model
mse = np.mean((predicted-expected)**2)
print(mse)
print(model.score(dataset.data, dataset.target))
```

Extra Trees

The Extra Trees ensemble method creates a large number of random decision trees, each on a different subset of the dataset. Unlike Random Forest, the split points in the decision trees are selected randomly (i.e. random trees). Like Random Forest, this general ensemble method is called bagging.

This recipe shows use of the Extra Trees algorithm to make predictions for the iris dataset (classification).

```
# Extra Trees Classification
from sklearn import datasets
from sklearn import metrics
from sklearn.ensemble import ExtraTreesClassifier
# load the iris datasets
dataset = datasets.load_iris()
# fit a Extra Trees model to the data
model = ExtraTreesClassifier()
model.fit(dataset.data, dataset.target)
print(model)
# make predictions
expected = dataset.target
predicted = model.predict(dataset.data)
# summarize the fit of the model
print(metrics.classification_report(expected, predicted))
print(metrics.confusion_matrix(expected, predicted))
```

This recipe shows use of the Extra Trees algorithm to make predictions for the diabetes dataset (regression).

```
# Extra Trees Regression
import numpy as np
from sklearn import datasets
from sklearn.ensemble import ExtraTreesRegressor
# load the diabetes datasets
dataset = datasets.load_diabetes()
# fit a extra trees model to the data
model = ExtraTreesRegressor()
model.fit(dataset.data, dataset.target)
print(model)
```

```
# make predictions
expected = dataset.target
predicted = model.predict(dataset.data)
# summarize the fit of the model
mse = np.mean((predicted-expected)**2)
print(mse)
print(model.score(dataset.data, dataset.target))
```

Adaptive Boosting

Adaptive Boosting (AdaBoost) is an ensemble method that sums the predictions made by multiple decision trees. Additional models are added and trained on the data instances that were miss-classified by previous models in the ensemble. Generally the approach to creating an ensemble used by AdaBoost is called boosting.

This recipe shows use of the AdaBoost algorithm to make predictions for the iris dataset (classification).

```
# AdaBoost Classification
from sklearn import datasets
from sklearn import metrics
from sklearn.ensemble import AdaBoostClassifier
# load the iris datasets
dataset = datasets.load_iris()
# fit an AdaBoost model to the data
model = AdaBoostClassifier()
model.fit(dataset.data, dataset.target)
print(model)
# make predictions
expected = dataset.target
predicted = model.predict(dataset.data)
# summarize the fit of the model
print(metrics.classification_report(expected, predicted))
print(metrics.confusion_matrix(expected, predicted))
```

This recipe shows use of the AdaBoost algorithm to make predictions for the diabetes dataset (regression).

```
# Random Forest Regression
import numpy as np
from sklearn import datasets
from sklearn.ensemble import AdaBoostRegressor
# load the diabetes datasets
dataset = datasets.load_diabetes()
# fit an AdaBoost model to the data
model = AdaBoostRegressor()
model.fit(dataset.data, dataset.target)
print(model)
```

```
# make predictions
expected = dataset.target
predicted = model.predict(dataset.data)
# summarize the fit of the model
mse = np.mean((predicted-expected)**2)
print(mse)
print(model.score(dataset.data, dataset.target))
```

Gradient Boosting

Gradient Boosting is an ensemble method that uses boosted decision trees like AdaBoost, but is generalized to allow an arbitrary differentiable loss function (i.e. how model error is calculated).

This recipe shows use of the Gradient Boosting algorithm to make predictions for the iris dataset (classification).

```
# Gradient Boosting Classification
from sklearn import datasets
from sklearn import metrics
from sklearn.ensemble import GradientBoostingClassifier
# load the iris dataset
dataset = datasets.load_iris()
# fit a Gradient Boosting model to the data
model = GradientBoostingClassifier()
model.fit(dataset.data, dataset.target)
print(model)
# make predictions
expected = dataset.target
predicted = model.predict(dataset.data)
# summarize the fit of the model
print(metrics.classification_report(expected, predicted))
print(metrics.confusion_matrix(expected, predicted))
```

This recipe shows use of the Gradient Boosting algorithm to make predictions for the diabetes dataset (regression).

```
# Gradient Boosting Regression
import numpy as np
from sklearn import datasets
from sklearn.ensemble import GradientBoostingRegressor
# load the diabetes dataset
dataset = datasets.load_diabetes()
# fit a Gradient Boosting model to the data
model = GradientBoostingRegressor()
model.fit(dataset.data, dataset.target)
print(model)
# make predictions
expected = dataset.target
```

```
predicted = model.predict(dataset.data)
# summarize the fit of the model
mse = np.mean((predicted-expected)**2)
print(mse)
print(model.score(dataset.data, dataset.target))
```

Advanced

There are many activities around preparing a model supported by scikit-learn such as automated methods to select data features, estimate the performance of a model on unseen data and tune the parameters of a model on a dataset.

This section provides recipes for advanced features on data and algorithms.

Recursive Feature Elimination

The Recursive Feature Elimination (RFE) method is a feature selection approach. It works by recursively removing attributes and building a model on those attributes that remain. It uses the model accuracy to identify which attributes (and combination of attributes) contribute the most to predicting the target attribute.

This recipe shows the use of RFE on the iris dataset to select 3 attributes.

```
# Recursive Feature Elimination
from sklearn import datasets
from sklearn.feature_selection import RFE
from sklearn.linear_model import LogisticRegression
# load the iris datasets
dataset = datasets.load_iris()
# create a base classifier used to evaluate a subset of attributes
model = LogisticRegression()
# create the RFE model and select 3 attributes
rfe = RFE(model, 3)
rfe = rfe.fit(dataset.data, dataset.target)
# summarize the selection of the attributes
print(rfe.support_)
print(rfe.ranking_)
```

Feature Importance

Methods that use ensembles of decision trees (like Random Forest and Extra Trees) can also compute the relative importance of each attribute. These importance values can be used to inform a feature selection process.

This recipe shows the construction of an Extra Trees ensemble of the iris dataset and the display of the relative feature importance.

```
# Feature Importance
from sklearn import datasets
from sklearn import metrics
from sklearn.ensemble import ExtraTreesClassifier
# load the iris datasets
dataset = datasets.load_iris()
```

```
# fit an Extra Trees model to the data
model = ExtraTreesClassifier()
model.fit(dataset.data, dataset.target)
# display the relative importance of each attribute
print(model.feature_importances_)
```

Cross Validation

Cross validation is a way of evaluating a model on a dataset. It provides an estimation of the accuracy of the model if it were to make predictions on previously unseen data. Cross validation estimations are used to aid in the selection of a robust model that is fit for purpose.

This recipe estimates the performance of logistic regression on the iris dataset using k-fold cross validation with 10 folds.

```
# Cross Validation Classification
import numpy as np
from sklearn import datasets
from sklearn.linear_model import LogisticRegression
from sklearn import cross_validation
# load the iris datasets
dataset = datasets.load_iris()
# prepare cross validation folds
num_folds = 10
num_instances = len(dataset.data)
kfold = cross_validation.KFold(n=num_instances, n_folds=num_folds)
# prepare a Logistic Regression model
model = LogisticRegression()
# evaluate the model k-fold cross validation
results = cross_validation.cross_val_score(model, dataset.data, dataset.target, cv=kfold)
# display the mean classification accuracy on each fold
print(results)
# display the mean and stdev of the classification accuracy
print(results.mean())
print(results.std())
```

Grid Search Parameter Tuning

Machine learning models are parameterized so that their behavior can be tuned for a given problem. Models can have many parameters and finding the best combination of parameters can be treated as a search problem.

Grid search is an approach to parameter tuning that will methodically build and evaluate a model for each combination of algorithm parameters specified in a grid.

This recipe evaluates different alpha values for the Ridge Regression algorithm on the diabetes dataset.

```
# Grid Search for Algorithm Tuning
import numpy as np
from sklearn import datasets
from sklearn.linear_model import Ridge
from sklearn.grid_search import GridSearchCV
# load the diabetes datasets
dataset = datasets.load_diabetes()
# prepare a range of alpha values to test
alphas = np.array([1,0.1,0.01,0.001,0.0001,0])
# create and fit a ridge regression model, testing each alpha
model = Ridge()
grid = GridSearchCV(estimator=model, param_grid=dict(alpha=alphas))
grid.fit(dataset.data, dataset.target)
print(grid)
# summarize the results of the grid search
print(grid.best_score_)
print(grid.best_estimator_.alpha)
```

Random Search Parameter Tuning

Random search is an approach to parameter tuning that will sample algorithm parameters from a random distribution (i.e. uniform) for a fixed number of iterations. A model is constructed and evaluated for each combination of parameters chosen.

This recipe evaluates different alpha random values between 0 and 1 for the Ridge Regression algorithm on the diabetes dataset.

```
# Randomized Search for Algorithm Tuning
import numpy as np
from scipy.stats import uniform as sp_rand
from sklearn import datasets
from sklearn.linear_model import Ridge
from sklearn.grid_search import RandomizedSearchCV
# load the diabetes datasets
dataset = datasets.load_diabetes()
# prepare a uniform distribution to sample for the alpha parameter
param_grid = {'alpha': sp_rand()}
# create and fit a ridge regression model, testing random alpha values
model = Ridge()
rsearch = RandomizedSearchCV(estimator=model, param_distributions=param_grid, n_iter=100)
rsearch.fit(dataset.data, dataset.target)
print(rsearch)
# summarize the results of the random parameter search
print(rsearch.best_score_)
print(rsearch.best_estimator_.alpha)
```

Resources

This book has provided a snapshot of the capability of scikit-learn and the broader SciPy stack. In this section I want to share with you resources that you can use to extend your working knowledge of scikit-learn.

Website

The scikit-learn website has a lot of excellent documentation. There are three parts to the website that I recommend studying in more detail:

- **User Guide**: The user guide tours through all of the major features of the library. Along with code examples and summaries of the machine learning methods themselves, the user guide provides usage heuristics for some algorithms. Find these and memorize them, they are invaluable. Link
- **API Documentation**: The API documentation goes into the nuts and bolts of how to use each Python object and function call in the library. This is invaluable to see what your options are, the defaults assumed and the best way to tune the usage of the library to your needs. Link
- **Gallery**: A gallery is provided if small examples that use the library. This is a clever idea and almost all examples in this gallery also include some sort of pretty visualization. Find and study examples related to the problem you are working on or methods you are using. Link

Papers

There are research papers that describe the scikit-learn platform. These can be insightful to both get a handle on the design of the library itself and on the goals and objectives of the broader projects. I recommend reading:

- scikit-learn: Machine Learning in Python (2011)
- API design for machine learning software: experiences from the scikit-learn project (2013)

Books

There are books that include examples of using scikit-learn on larger and varied types of problems. Two books I recommend reading are:

- Building Machine Learning Systems with Python (2013)
- Learning scikit-learn: Machine Learning in Python (2013)

Next Steps

This guide covers a lot of recipes, but there were aspects of the scikit-learn library that were not covered. Three subjects that you may wish to study further yourself include:

- Unsupervised Learning such as clustering methods and manifold learning.
- Pipeline that allows the chaining of scikit-learn objects together.
- Computer Vision and Natural Language Processing problems that are supported by the library but were not covered in this guide.

Once you have mastered the basis of the scikit-learn library, the next step is to look at more complex problems. The books in the previous section provide an excellent starting point for learning how to apply scikit-learn on diverse and complex problems.

You may also like to look into machine learning competitions on open datasets that can help you practice and refine your usage of machine learning models.

Get Started Right Now!

This is a long guide and just reading it will not help you. You must take action.

Work through each tutorial and get to know the scikit-learn platform. Apply your new found skills on a machine learning problem. Work through the steps of the applied machine learning process using scikit-learn and present your findings. I'm excited to see what you can do.

Get serious and declare your intentions. Decide your next step and tell people about it on Twitter, Facebook, Google+, your blog or in person.

By declaring your intentions you will be more likely to finish and share your results.
You could write something like:

> "*I now know scikit-learn and my first project will be ...*"

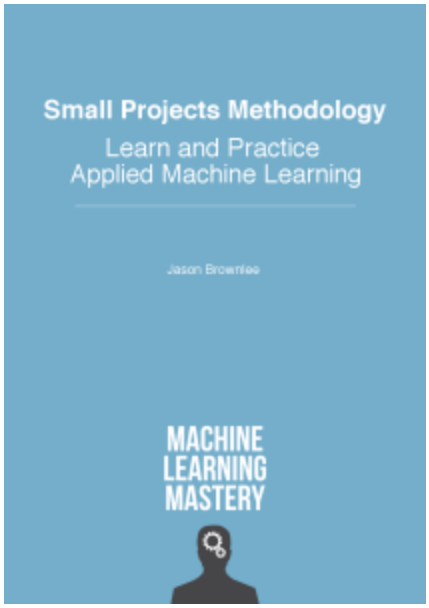
Good Luck!

I'd love to see and share what you come up with.
Please email me with links to your work at Jason@MachineLearningMastery.com

Jason Brownlee

Related Guides

This section lists of my related guides on machine learning that you might find interesting.

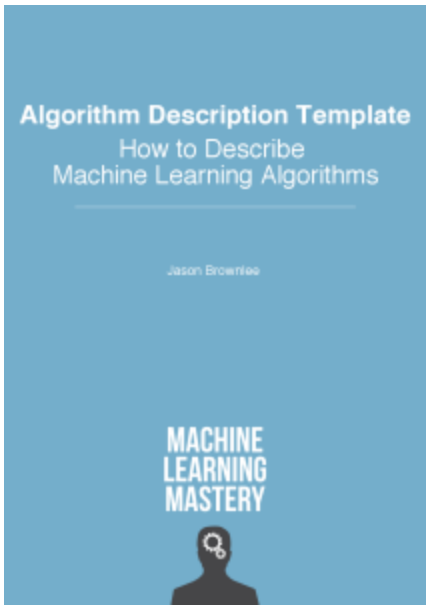


Small Projects Methodology: Learn and Practice Applied Machine Learning

If you are interested in some ideas for self-study projects, you might be interested in the Small Projects Methodology.

It provides 4 classes of project ideas with tactics and strategies you can use, as well as 90 ideas of projects that you could study

You can learn more at
http://SmallProjectsMethodology.com



Algorithm Description Template: How to Describe Machine Learning Algorithms

If you are interested in a deep learning of machine learning algorithms, you might be interested in the Algorithm Description Template.

It provides a structured step-by-step template to research and learn about machine learning algorithms.

You can learn more at
http://AlgorithmDescriptionTemplate.com