A Introduction to Scikit-Learn

Lab Objective: *Scikit-learn is the one of the fundamental tools in Python for machine learning. In this appendix we highlight and give examples of some popular scikit-learn tools for classification and regression, training and testing, data normalization, and constructing complex models.*

> **Note**
>
> This guide corresponds to scikit-learn version 0.20, which has a few significant differences from previous releases. See `http://scikit-learn.org/stable/whats_new.html` for current release notes. Install scikit-learn (the `sklearn` module) with `conda install scikit-learn`.

Base Classes and API

Many machine learning problems center on constructing a function $f : X \to Y$, called a *model* or *estimator*, that accurately represents properties of given data. The domain X is usually $\mathbb{R}^D$, and the range Y is typically either $\mathbb{R}$ (regression) or a subset of $\mathbb{Z}$ (classification). The model is trained on N *samples* $(\mathbf{x}_i)_{i=1}^N \subset X$ that usually (but not always) have N accompanying *labels* $(y_i)_{i=1}^N \subset Y$.

Scikit-learn [**?**, **?**] takes a highly object-oriented approach to machine learning models. Every major scikit-learn class inherits from `sklearn.base.BaseEstimator` and conforms to the following conventions:

1. The constructor `__init__()` receives *hyperparameters* for the classifier, which are parameters for the model f that are **not dependent on data**. Each hyperparameter must have a default value (i.e., every argument of `__init__()` is a keyword argument), and each argument must be saved as an instance variable of the **same name** as the parameter.

2. The `fit()` method constructs the model f. It receives an $N \times D$ matrix X and, optionally, a vector $\mathbf{y}$ with N entries. Each row $\mathbf{x}_i$ of X is one sample with corresponding label y_i. By convention, `fit()` always returns `self`.

Along with the `BaseEstimator` class, there are several other "mix in" base classes in `sklearn.base` that define specific kinds of models. The three listed below are the most common.[1]

[1] See `http://scikit-learn.org/stable/modules/classes.html#base-classes` for the complete list.

- `ClassifierMixin`: for *classifiers*, estimators that take on discrete values.
- `RegressorMixin`: for *regressors*, estimators that take on continuous values.
- `TransformerMixin`: for preprocessing data before estimation.

Classifiers and Regressors

The `ClassifierMixin` and `RegressorMixin` both require a `predict()` method that acts as the actual model f. That is, `predict()` receives an $N \times D$ matrix X and returns N predicted labels $(y_i)_{i=1}^N$, where y_i is the label corresponding to the ith row of X. Both of these base class have a predefined `score()` method that uses `predict()` to test the accuracy of the model. It accepts $N \times D$ test data and a vector of N corresponding labels, then reports either the classification accuracy (for classifiers) or the R^2 value of the regression (for regressors).

For example, a `KNeighborsClassifier` from `sklearn.neighbors` inherits from `BaseEstimator` and `ClassifierMixin`. This classifier uses a simple strategy: to classify a new piece of data $\mathbf{z}$, find the k training samples that are "nearest" to $\mathbf{z}$, then take the most common label corresponding to those nearest neighbors to be the label for $\mathbf{z}$. Its constructor accepts hyperparameters such as `n_neighbors`, for determining the number of neighbors k to search for, `algorithm`, which speficies the strategy to find the neighbors, and `n_jobs`, the number of parallel jobs to run during the neighbors search. Again, these hyperparameters are independent of any data, which is why they are set in the constructor (before fitting the model). Calling `fit()` organizes the data X into a data structure for efficient nearest neighbor searches (determined by `algorithm`). Calling `predict()` executes the search, determines the most common label of the neighbors, and returns that label.

```
>>> from sklearn.datasets import load_breast_cancer
>>> from sklearn.neighbors import KNeighborsClassifier
>>> from sklearn.model_selection import train_test_split

# Load the breast cancer dataset and split it into training and testing groups.
>>> cancer = load_breast_cancer()
>>> X_train, X_test, y_train, y_test = train_test_split(cancer.data,
...                                                     cancer.target)
>>> print(X_train.shape, y_train.shape)
(426, 30) (426,)        # There are 426 training points, each with 30 features.

# Train a KNeighborsClassifier object on the training data.
# fit() returns the object, so we can instantiate and train in a single line.
>>> knn = KNeighborsClassifier(n_neighbors=2).fit(X_train, y_train)
# The hyperparameter 'n_neighbors' is saved as an attribute of the same name.
>>> knn.n_neighbors
2

# Test the classifier on the testing data.
>>> knn.predict(X_test[:6])
array([0, 1, 0, 1, 1, 0])       # Predicted labels for the first 6 test points.
>>> knn.score(X_test, y_test)
0.8951048951048951              # predict() chooses 89.51% of the labels right.
```

The `KNeighborsClassifier` object could easily be replaced with a different classifier, such as a `GaussianNB` object from `sklearn.naive_bayes`. Since `GaussianNB` also inherits from `BaseEstimator` and `ClassifierMixin`, it has `fit()`, `predict()`, and `score()` methods that take in the same kinds of inputs as the corresponding methods for the `KNeighborsClassifier`. The only difference, from an external perspective, is the hyperparameters that the constructor accepts.

```
>>> from sklearn.naive_bayes import GaussianNB

>>> gnb = GaussianNB().fit(X_train, y_train)
>>> gnb.predict(X_test[:6])
array([1, 1, 0, 1, 1, 0])
>>> gnb.score(X_test, y_test)
0.9440559440559441
```

Roughly speaking, the `GaussianNB` classifier assumes all features in the data are independent and normally distributed, then uses Bayes' rule to compute the likelihood of a new point belonging to a label for each of the possible labels. To do this, the `fit()` method computes the mean and variance of each feature, grouped by label. These quantities are saved as the attributes `theta_` (the means) and `sigma_` (the variances), then used in `predict()`. Parameters like these that **are dependent on data** are only defined in `fit()`, not the constructor, and they are always named with a trailing underscore. These "non-hyper" parameters are often simply called *model parameters*.

```
>>> gnb.classes_        # The collection of distinct training labels.
array([0, 1])
>>> gnb.theta_[:,0]     # The means of the first feature, grouped by label.
array([17.55785276, 12.0354981 ])
# The samples with label 0 have a mean of 17.56 in the first feature.
```

The `fit()` method should do all of the "heavy lifting" by calculating the model parameters. The `predict()` method should then use these parameters to choose a label for test data.

	Hyperparameters	Model Parameters
Data dependence	No	Yes
Initialization location	`__init__()`	`fit()`
Naming convention	Same as argument name	Ends with an underscore
Examples	`n_neighbors`, `algorithm`, `n_jobs`	`classes_`, `theta_`, `sigma_`

Table A.1: Naming and initialization conventions for scikit-learn model parameters.

Building Custom Estimators

The consistent conventions in the various scikit-learn classes makes it easy to use a wide variety of estimators with near-identical syntax. These conventions also makes it possible to write custom estimators that behave like native scikit-learn objects. This usually only involves writing `fit()` and `predict()` methods and inheriting from the appropriate base classes. As a simple (though poorly performing) example, consider an estimator that either always predicts the same user-provided label, or that always predicts the most common label in the training data. Which strategy to use is independent of the data, so we encode that behavior with hyperparameters; the most common label must be calculated from the data, so that is a model parameter.

```
>>> import numpy as np
>>> from collections import Counter
>>> from sklearn.base import BaseEstimator, ClassifierMixin

>>> class PopularClassifier(BaseEstimator, ClassifierMixin):
...     """Classifier that always guesses the most common training label."""
...     def __init__(self, strategy="most_frequent", constant=None):
...         self.strategy = strategy    # Store the hyperparameters, using
...         self.constant = constant    # the same names as the arguments.
...
...     def fit(self, X, y):
...         """Find and store the most common label."""
...         self.popular_label_ = Counter(y).most_common(1)[0][0]
...         return self                 # fit() always returns 'self'.
...
...     def predict(self, X):
...         """Always guess the most popular training label."""
...         M = X.shape[0]
...         if self.strategy == "most_frequent":
...             return np.full(M, self.popular_label_)
...         elif self.strategy == "constant":
...             return np.full(M, self.constant)
...         else:
...             raise ValueError("invalid value for 'strategy' param")
...
# Train a PopularClassifier on the breast cancer training data.
>>> pc = PopularClassifier().fit(X_train, y_train)
>>> pc.popular_label_
1
# Score the model on the testing data.
>>> pc.score(X_test, y_test)
0.6573426573426573                  # 65.73% of the testing data is labeled 1.

# Change the strategy to always guess 0 by changing the hyperparameters.
>>> pc.strategy = "constant"
>>> pc.constant = 0
>>> pc.score(X_test, y_test)
0.34265734265734266                  # 34.27% of the testing data is labeled 0.
```

This is a terrible classifier, but it is actually implemented as `sklearn.dummy.DummyClassifier` because any legitimate machine learning algorithm should be able to beat it, so it is useful as a baseline comparison.

Note that `score()` was inherited from `ClassifierMixin` (it isn't defined explicitly), so it returns a classification rate. In the next example, a slight simplification of the equally unintelligent `sklearn.dummy.DummyRegressor`, the `score()` method is inherited from `RegressorMixin`, so it returns an R^2 value.

```
>>> from sklearn.base import RegressorMixin

>>> class ConstRegressor(BaseEstimator, RegressorMixin):
...     """Regressor that always predicts a mean or median of training data."""
...     def __init__(self, strategy="mean", constant=None):
...         self.strategy = strategy    # Store the hyperparameters, using
...         self.constant = constant    # the same names as the arguments.
...
...     def fit(self, X, y):
...         self.mean_, self.median_ = np.mean(y), np.median(y)
...         return self                 # fit() always returns 'self'.
...
...     def predict(self, X):
...         """Always predict the middle of the training data."""
...         M = X.shape[0]
...         if self.strategy == "mean":
...             return np.full(M, self.mean_)
...         elif self.strategy == "median":
...             return np.full(M, self.median_)
...         elif self.strategy == "constant":
...             return np.full(M, self.constant)
...         else:
...             raise ValueError("invalid value for 'strategy' param")
...
# Train on the breast cancer data (treating it as a regression problem).
>>> cr = ConstRegressor(strategy="mean").fit(X_train, y_train)
>>> print("mean:", cr.mean_, " median:", cr.median_)
mean: 0.6173708920187794  median: 1.0

# Get the R^2 score of the regression on the testing data.
>>> cr.score(X_train, y_train)
0                           # Unsurprisingly, no correlation.
```

Achtung!

Both `PopularClassifier` and `ConstRegressor` wait until `predict()` to validate the `strategy` hyperparameter. The check could easily be done in the constructor, but that goes against scikit-learn conventions: in order to cooperate with automated validation tools, the constructor of any class inheriting from `BaseEstimator` must store the arguments of `__init__()` as attributes—with the same names as the arguments—and do nothing else.

Note

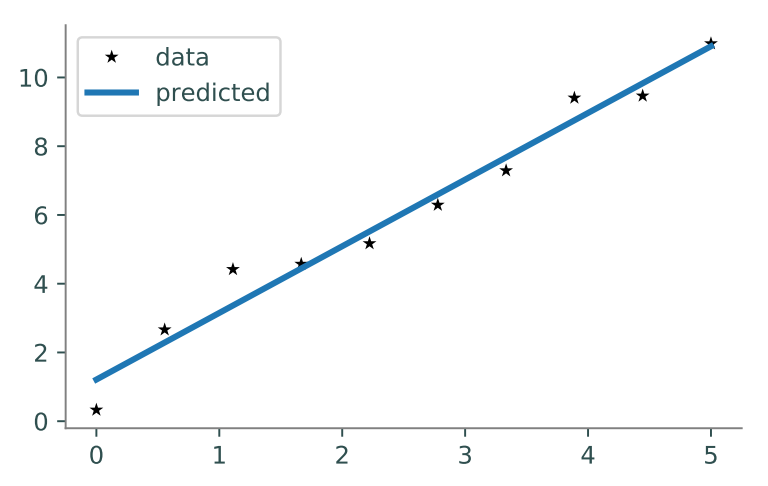
The first input to `fit()` and `predict()` are **always** two-dimensional $N \times D$ NumPy arrays, where N is the number of observations and D is the number of features. To fit or predict on one-dimensional data ($D = 1$), reshape the input array into a "column vector" before feeding it into the estimator. One-dimensional problems are somewhat rare in machine learning, but the following example shows how to do a simple one-dimensional linear regression.

```
>>> from matplotlib import pyplot as plt
>>> from sklearn.linear_model import LinearRegression

# Generate data for a 1-dimensional regression problem.
>>> X = np.linspace(0, 5, 10)
>>> Y = 2*X + 1 + np.random.normal(size=10)

# Reshape the training data into a column vector.
>>> lr = LinearRegression().fit(X.reshape((-1,1)), Y)

# Define another set of points to do predictions on.
>>> x = np.linspace(0, 5, 20)
>>> y = lr.predict(x.reshape((-1,1)))   # Reshape before predicting.
>>> plt.plot(X, Y, 'k*', label="data")
>>> plt.plot(x, y, label="predicted")
>>> plt.legend(loc="upper left")
>>> plt.show()
```



Transformers

A scikit-learn *transformer* processes data to make it better suited for estimation. This may involve shifting and scaling data, dropping columns, replacing missing values, and so on.

Classes that inherit from the `TransformerMixin` base class have a `fit()` method that accepts an $N \times D$ matrix X (like an estimator) and an optional set of labels. The labels are not needed—in fact the `fit()` method should do nothing with them—but the parameter for the labels remains as a keyword argument to be consistent with the `fit(X,y)` syntax of estimators. Instead of a `predict()` method, the `transform()` method accepts data, modifies it (usually via a copy), and returns the result. The new data may or may not have the same number of columns as the original data.

One common transformation is shifting and scaling the features (columns) so that they each have a mean of 0 and a standard deviation of 1. The following example implements a basic version of this transformer.

```
>>> from sklearn.base import TransformerMixin

>>> class NormalizingTransformer(BaseEstimator, TransformerMixin):
...     def fit(self, X, y=None):
...         """Calculate the mean and standard deviation of each column."""
...         self.mu_ = np.mean(X, axis=0)
...         self.sig_ = np.std(X, axis=0)
...         return self
...
...     def transform(self, X):
...         """Center each column at zero and normalize it."""
...         return (X - self.mu_) / self.sig_
...
# Fit the transformer and transform the cancer data (both train and test).
>>> nt = NormalizingTransformer()
>>> Z_train = nt.fit_transform(X_train) # Or nt.fit(X_train).transform(X_train)
>>> Z_test = nt.transform(X_test)       # Transform test data (without fitting)

>>> np.mean(Z_train, axis=0)[:3]        # The columns of Z_train have mean 0...
array([-8.08951237e-16, -1.72006384e-17,  1.78678147e-15])
>>> np.std(Z_train, axis=0)[:3]         # ...and have unit variance.
array([1., 1., 1.])
>>> np.mean(Z_test, axis=0)[:3]         # The columns of Z_test each have mean
array([-0.02355067,  0.11665332, -0.03996177])                  # close to 0...
>>> np.std(Z_test, axis=0)[:3]          # ...and have close to unit deviation.
array([0.9263711 , 1.18461151, 0.91548103])

# Check to see if the classification improved.
>>> knn.fit(X_train, y_train).score(X_test, y_test)             # Old score.
0.8951048951048951
>>> knn.fit(Z_train, y_train).score(Z_test, y_test)             # New score.
0.958041958041958
```

This particular transformer is implemented as `sklearn.preprocessing.StandardScaler`. A close cousin is `sklearn.preprocessing.RobustScaler`, which ignores outliers when choosing the scaling and shifting factors.

Like estimators, transformers may have both hyperparameters (provided to the constructor) and model parameters (determined by `fit()`). Thus a transformer looks and acts like an estimator, with the exception of the `predict()` and `transform()` methods.

Achtung!

The `transform()` method should only rely on model parameters derived from the training data in `fit()`, **not** on the data that is worked on in `transform()`. For example, if the `NormalizingTransformer` is fit with the input $\widehat{X}$, then `transform()` should shift and scale any input X by the mean and standard deviation of $\widehat{X}$, not by the mean and standard deviation of X. Otherwise, the transformation is different for each input X.

Scikit-learn Module	**Classifier** Name	Notable Hyperparameters
discriminant_analysis	LinearDiscriminantAnalysis	solver, shrinkage, n_components
discriminant_analysis	QuadraticDiscriminantAnalysis	reg_param
ensemble	AdaBoostClassifier	n_estimators, learning_rate
ensemble	RandomForestClassifier	n_estimators, max_depth
linear_model	LogisticRegression	penalty, C
linear_model	SGDClassifier	loss, penalty, alpha
naive_bayes	GaussianNB	priors
naive_bayes	MultinomialNB	alpha
neighbors	KNeighborsClassifier	n_neighbors, weights
neighbors	RadiusNeighborsClassifier	radius, weights
neural_network	MLPClassifier	hidden_layer_size, activation
svm	SVC	C, kernel
tree	DecisionTreeClassifier	max_depth
Scikit-learn Module	**Regressor** Name	Notable Hyperparameters
ensemble	AdaBoostRegressor	n_estimators, learning_rate
ensemble	ExtraTreesRegressor	n_estimators, max_depth
ensemble	GradientBoostingRegressor	n_estimators, max_depth
ensemble	RandomForestRegressor	n_estimators, max_depth
isotonic	IsotonicRegression	y_min, y_max
kernel_ridge	KernelRidge	alpha, kernel
linear_model	LinearRegression	fit_intercept
neural_network	MLPRegressor	hidden_layer_size, activation
svm	SVR	C, kernel
tree	DecisionTreeRegressor	max_depth
Module	**Transformer** Name	Notable Hyperparameters
decomposition	PCA	n_components
preprocessing	Imputer	missing_values, strategy
preprocessing	MinMaxScaler	feature_range
preprocessing	OneHotEncoder	categorical_features
preprocessing	QuantileTransformer	n_quantiles, output_distribution
preprocessing	RobustScaler	with_centering, with_scaling
preprocessing	StandardScaler	with_mean, with_std

Table A.2: Common scikit-learn classifiers, regressors, and transformers. For full documentation on these classes, see `http://scikit-learn.org/stable/modules/classes.html`.

Validation Tools

Knowing how to determine whether or not an estimator performs well is an essential part of machine learning. This often turns out to be a surprisingly sophisticated issue that largely depends on the type of problem being solved and the kind of data that is available for training. Scikit-learn has validation tools for many situations; for brevity, we restrict our attention to the simple (but important) case of *binary classification*, where the range of the desired model is $Y = \{0, 1\}$.

Evaluation Metrics

The `score()` method of a scikit-learn estimator representing the model $f : X \to \{0, 1\}$ returns the *accuracy* of the model, which is the percent of labels that are predicted correctly. However, accuracy isn't always the best measure of success. Consider the *confusion matrix* for a classifier, the matrix where the (i, j)th entry is the number of observations with actual label i but that are classified as label j. In binary classification, calling the class with label 0 the *negatives* and the class with label 1 the *positives*, this becomes the following.

$$\begin{array}{r} \\ \text{Actual: } 0 \\ \text{Actual: } 1 \end{array} \begin{array}{c} \begin{array}{cc} \text{Predicted: } 0 & \text{Predicted: } 1 \end{array} \\ \begin{bmatrix} \text{True Negatives } (TN) & \text{False Positives } (FP) \\ \text{False Negatives } (FN) & \text{True Positives } (TP) \end{bmatrix} \end{array}$$

With this terminology, we define the following metrics.

- *Accuracy*: $\dfrac{TN + TP}{TN + FN + FP + TP}$, the percent of labels predicted correctly.
- *Precision*: $\dfrac{TP}{TP + FP}$, the percent of predicted positives that are actually correct.
- *Recall*: $\dfrac{TP}{TP + FN}$, the percent of actual positives that are predicted correctly.

Precision is useful in situations where false positives are dangerous or costly, while recall is important when avoiding false negatives takes priority. For example, an email spam filter should avoid filtering out an email that isn't actually spam, so precision is a valuable metric for the filter. On the other hand, recall is more important in disease detection: it is better to test positive and not have the disease than to test negative when the disease is actually present. Focusing on a single metric often leads to skewed results (for example, always predicting the same label), so the following metric is also common.

- *F_β Score*: $(1 + \beta^2)\dfrac{\text{precision} \cdot \text{recall}}{(\beta^2 \cdot \text{precision}) + \text{recall}} = \dfrac{(1 + \beta^2)TP}{(1 + \beta^2)TP + FP + \beta^2 FN}$.

Choosing $\beta < 1$ weighs precision more than recall, while $\beta > 1$ prioritizes recall over precision. The choice of $\beta = 1$ yields the common F_1 score, which weighs precision and recall equally. This is an important alternative to accuracy when, for example, the training set is heavily unbalanced with respect to the class labels.

Scikit-learn implements these metrics in `sklearn.metrics`, as well as functions for evaluating regression, non-binary classification, and clustering models. The general syntax for such functions is `some_score(actual_labels, predicted_labels)`. For the complete list and further discussion, see `http://scikit-learn.org/stable/modules/model_evaluation.html`.

```
>>> from sklearn.metrics import (confusion_matrix, classification_report,
...                              accuracy_score, precision_score,
...                              recall_score, f1_score)

# Fit the esimator to training data and predict the test labels.
>>> knn.fit(X_train, y_train)
>>> knn_predicted = knn.predict(X_test)

# Compute the confusion matrix by comparing actual labels to predicted labels.
>>> CM = confusion_matrix(y_test, knn_predicted)
>>> CM
array([[44,  5],
       [10, 84]])

# Get accuracy (the "usual" score), precision, recall, and f1 scores.
>>> accuracy_score(y_test, knn_predicted)   # (CM[0,0] + CM[1,1]) / CM.sum()
0.8951048951048951
>>> precision_score(y_test, knn_predicted)  # CM[1,1] / CM[:,1].sum()
0.9438202247191011
>>> recall_score(y_test, knn_predicted)     # CM[1,1] / CM[1,:].sum()
0.8936170212765957
>>> f1_score(y_test, knn_predicted)
0.9180327868852459

# Get all of these scores at once with classification_report().
>>> print(classification_report(y_test, knn_predicted))
              precision    recall  f1-score   support

           0       0.81      0.90      0.85        49
           1       0.94      0.89      0.92        94

   micro avg       0.90      0.90      0.90       143
   macro avg       0.88      0.90      0.89       143
weighted avg       0.90      0.90      0.90       143
```

Cross Validation

The `sklearn.model_selection` module has utilities to streamline and improve model evaluation.

- `train_test_split()` randomly splits data into training and testing sets (we already used this).
- `cross_val_score()` randomly splits the data and trains and scores the model a set number of times. Each trial uses different training data and results in a different model. The function returns the score of each trial.
- `cross_validate()` does the same thing as `cross_val_score()`, but it also reports the time it took to fit, the time it took to score, and the scores for the test set as well as the training set.

Doing multiple evaluations with different testing and training sets is extremely important. If the scores on a cross validation test vary wildly, the model is likely overfitting to the training data.

```
>>> from sklearn.model_selection import cross_val_score, cross_validate

# Make (but do not train) a classifier to test.
>>> knn = KNeighborsClassifier(n_neighbors=3)

# Test the classifier on the training data 4 times.
>>> cross_val_score(knn, X_train, y_train, cv=4)
array([0.88811189, 0.92957746, 0.96478873, 0.92253521])

# Get more details on the train/test procedure.
>>> cross_validate(knn, X_train, y_train, cv=4,
...                return_train_score=False)
{'fit_time': array([0.00064683, 0.00042295, 0.00040913, 0.00040436]),
 'score_time': array([0.00115728, 0.00109601, 0.00105286, 0.00102782]),
 'test_score': array([0.88811189, 0.92957746, 0.96478873, 0.92253521])}

# Do the scoring with an alternative metric.
>>> cross_val_score(knn, X_train, y_train, scoring="f1", cv=4)
array([0.93048128, 0.95652174, 0.96629213, 0.93103448])
```

> **Note**
>
> Any estimator, even a user-defined class, can be evaluated with the scikit-learn tools presented in this section as long as that class conforms to the scikit-learn API discussed previously (i.e., inheriting from the correct base classes, having `fit()` and `predict()` methods, managing hyperparameters and parameters correctly, and so on). Any time you define a custom estimator, following the scikit-learn API gives you instant access to tools such as `cross_val_score()`.

Grid Search

Recall that the *hyperparameters* of a machine learning model are user-provided parameters that do not depend on the training data. Finding the optimal hyperparameters for a given model is a challenging and active area of research.[2] However, brute-force searching over a small hyperparameter space is simple in scikit-learn: a `sklearn.model_selection.GridSearchCV` object is initialized with an estimator, a dictionary of hyperparameters, and cross validation parameters (such as `cv` and `scoring`). When its `fit()` method is called, it does a cross validation test on the given estimator with every possible hyperparameter combination.

For example, a k-neighbors classifier has a few important hyperparameters that can have a significant impact on the speed and accuracy of the model: `n_neighbors`, the number of nearest neighbors allowed to vote; and `weights`, which specifies a strategy for weighting the distances between points. The following code tests various combinations of these hyperparameters.

[2] Intelligent hyperparameter selection is sometimes called metalearning. See, for example, [**?**].

```
>>> from sklearn.model_selection import GridSearchCV

>>> knn = KNeighborsClassifier()
# Specify the hyperparameters to vary and the possible values they should take.
>>> param_grid = {"n_neighbors": [2, 3, 4, 5, 6],
...               "weights": ["uniform", "distance"]}
>>> knn_gs = GridSearchCV(knn, param_grid, cv=4, scoring="f1", verbose=1)
>>> knn_gs.fit(X_train, y_train)
Fitting 4 folds for each of 5 candidates, totalling 20 fits
[Parallel(n_jobs=1)]: Using backend SequentialBackend with 1 concurrent worker.
[Parallel(n_jobs=1)]: Done  20 out of  20 | elapsed:    0.1s finished

# After fitting, the gridsearch object has data about the results.
>>> print(knn_gs.best_params_, knn_gs.best_score_)
{'n_neighbors': 5, 'weights': 'uniform'} 0.9532526583188765
```

The cost of a grid search rapidly increases as the hyperparameter space grows. However, the outcomes of each trial are completely independent of each other, so the problem of training each classifier is *embarassingly parallel*. To parallelize the grid search over n cores, set the `n_jobs` parameter to n, or set it to -1 to divide the labor between as many cores as are available.

In some circumstances, the parameter grid can be also organized in a way that eliminates redundancy. Consider an `SVC` classifier from `sklearn.svm`, an estimator that works by lifting the data into a high-dimensional space, then constructing a hyperplane to separate the classes. The `SVC` has a hyperparameter, `kernel`, that determines how the lifting into higher dimensions is done, and for each choice of kernel there are additional corresponding hyperparameters. To search the total hyperparameter space without redundancies, enter the parameter grid as a list of dictionaries, each of which defines a different section of the hyperparameter space. In the following code, doing so reduces the number of trials from $3 \times 2 \times 3 \times 4 = 72$ to only $1 + (1 \times 1 \times 3) + (1 \times 4) = 11$.

```
>>> from sklearn.svm import SVC

>>> svc = SVC(C=0.01, max_iter=100)
>>> param_grid = [
...     {"kernel": ["linear"]},
...     {"kernel": ["poly"], "degree": [2,3], "coef0": [0,1,5]},
...     {"kernel": ["rbf"], "gamma": [.01, .1, 1, 100]}]
>>> svc_gs = GridSearchCV(svc, param_grid,
                          cv=4, scoring="f1",
                          verbose=1, n_jobs=-1).fit(X_train, y_train)
Fitting 4 folds for each of 11 candidates, totalling 44 fits
[Parallel(n_jobs=-1)]: Using backend LokyBackend with 8 concurrent workers.
[Parallel(n_jobs=-1)]: Done  44 out of  44 | elapsed:    2.4s finished

>>> print(svc_gs.best_params_, svc_gs.best_score_)
{'gamma': 0.01, 'kernel': 'rbf'} 0.8909310239174055
```

See `https://scikit-learn.org/stable/modules/grid_search.html` for more details about `GridSearchCV` and its relatives.

Pipelines

Most machine learning problems require at least a little data preprocessing before estimation in order to get good results. A scikit-learn *pipeline* (`sklearn.pipeline.Pipeline`) chains together one or more transformers and one estimator into a single object, complete with `fit()` and `predict()` methods. For example, it is often a good idea to shift and scale data before feeding it into a classifier. The `StandardScaler` transformer can be combined with a classifier with a pipeline. Calling `fit()` on the resulting object calls `fit_transform()` on each successive transformer, then `fit()` on the estimator at the end. Likewise, calling `predict()` on the `Pipeline` object calls `transform()` on each transformer, then `predict()` on the estimator.

```
>>> from sklearn.preprocessing import StandardScaler
>>> from sklearn.pipeline import Pipeline

# Chain together a scaler transformer and a KNN estimator.
>>> pipe = Pipeline([("scaler", StandardScaler()),      # "scaler" is a label.
                     ("knn", KNeighborsClassifier())])  # "knn" is a label.
>>> pipe.fit(X_train, y_train)
>>> pipe.score(X_test, y_test)
0.972027972027972                       # Already an improvement!
```

Since `Pipeline` objects behaves like estimators (following the `fit()` and `predict()` conventions), they can be used with tools like `cross_val_score()` and `GridSearchCV`. To specify which hyperparameters belong to which steps of the pipeline, precede each hyperparameter name with `<stepname>__`. For example, `knn__n_neighbors` corresponds to the `n_neighbors` hyperparameter of the part of the pipeline that is labeled `knn`.

```
# Specify the possible hyperparameters for each step.
>>> pipe_param_grid = {"scaler__with_mean": [True, False],
...                    "scaler__with_std": [True, False],
...                    "knn__n_neighbors": [2,3,4,5,6],
...                    "knn__weights": ["uniform", "distance"]}

# Pass the Pipeline object to the GridSearchCV and fit it to the data.
>>> pipe = Pipeline([("scaler", StandardScaler()),
                     ("knn", KNeighborsClassifier())])
>>> pipe_gs = GridSearchCV(pipe, pipe_param_grid,
...                        cv=4, n_jobs=-1, verbose=1).fit(X_train, y_train)
Fitting 4 folds for each of 40 candidates, totalling 160 fits
[Parallel(n_jobs=-1)]: Using backend LokyBackend with 8 concurrent workers.
[Parallel(n_jobs=-1)]: Done 160 out of 160 | elapsed:    0.3s finished

>>> print(pipe_gs.best_params_, pipe_gs.best_score_, sep='\n')
{'knn__n_neighbors': 6, 'knn__weights': 'distance',
 'scaler__with_mean': True, 'scaler__with_std': True}
0.971830985915493
```

Pipelines can also be used to compare different transformations or estimators. For example, a pipeline could end in either a `KNeighborsClassier()` or an `SVC()`, even though they have different hyperparameters. Like before, use a list of dictionaries to specify the hyperparameter space.

```
>>> pipe = Pipeline([("scaler", StandardScaler()),
                     ("classifier", KNeighborsClassifier())])
>>> pipe_param_grid = [
...     {"classifier": [KNeighborsClassifier()],    # Try a KNN classifier...
...      "classifier__n_neighbors": [2,3,4,5],
...      "classifier__weights": ["uniform", "distance"]},
...     {"classifier": [SVC(kernel="rbf")],         # ...and an SVM classifier.
...      "classifier__C": [.001, .01, .1, 1, 10, 100],
...      "classifier__gamma": [.001, .01, .1, 1, 10, 100]}]
>>> pipe_gs = GridSearchCV(pipe, pipe_param_grid,
...                        cv=5, scoring="f1",
...                        verbose = 1, n_jobs=-1).fit(X_train, y_train)
Fitting 5 folds for each of 44 candidates, totalling 220 fits
[Parallel(n_jobs=-1)]: Using backend LokyBackend with 8 concurrent workers.
[Parallel(n_jobs=-1)]: Done 220 out of 220 | elapsed:    0.6s finished

>>> params = pipe_gs.best_params_
>>> print("Best classifier:", params["classifier"])
Best classifier: SVC(C=10, cache_size=200, class_weight=None, coef0=0.0,
  decision_function_shape='ovr', degree=3, gamma=0.01, kernel='rbf',
  max_iter=-1, probability=False, random_state=None, shrinking=True,
  tol=0.001, verbose=False)

# Check the best classifier against the test data.
>>> confusion_matrix(y_test, pipe_gs.predict(X_test))
array([[48,  1],                        # Near perfect!
       [ 1, 93]])
```

Additional Material

Exercises

Problem 1. Writing custom scikit-learn transformers is a convenient way to organize the data cleaning process. Consider the data in `titanic.csv`, which contains information about passengers on the maiden voyage of the *RMS Titanic* in 1912. Write a custom transformer class to clean this data, implementing the `transform()` method as follows:

1. Extract a copy of data frame with just the `"Pclass"`, `"Sex"`, and `"Age"` columns.
2. Replace `NaN` values in the `"Age"` column (of the copied data frame) with the mean age. The mean age of the training data should be calculated in `fit()` and used in `transform()` (compare this step to using `sklearn.preprocessing.Imputer`).
3. Convert the `"Pclass"` column datatype to pandas categoricals (`pd.CategoricalIndex`).
4. Use `pd.get_dummies()` to convert the categorical columns to multiple binary columns (compare this step to using `sklearn.preprocessing.OneHotEncoder`).
5. Cast the result as a NumPy array and return it.

Ensure that your transformer matches scikit-learn conventions (it inherits from the correct base classes, `fit()` returns `self`, etc.).

Problem 2. Read the data from `titanic.csv` with `pd.read_csv()`. The `"Survived"` column indicates which passengers survived, so the entries of the column are the labels that we would like to predict. Drop any rows in the raw data that have `NaN` values in the `"Survived"` column, then separate the column from the rest of the data. Split the data and labels into training and testing sets. Use the training data to fit a transformer from Problem 1, then use that transformer to clean the training set, then the testing set. Finally, train a `LogisticRegressionClassifier` and a `RandomForestClassifier` on the cleaned training data, and score them using the cleaned test set.

Problem 3. Use `classification_report()` to score your classifiers from Problem 2. Next, do a grid search for each classifier (using only the cleaned training data), varying at least two hyperparameters for each kind of model. Use `classification_report()` to score the resulting best estimators with the cleaned test data. Try changing the hyperparameter spaces or scoring metrics so that each grid search yields a better estimator.

Problem 4. Make a pipeline with at least two transformers to further process the Titanic dataset. Do a gridsearch on the pipeline and report the hyperparameters of the best estimator.