Supervised learning consists in learning the link between two datasets: the observed data `X` and an external variable `y` that we are trying to predict, usually called “target” or “labels”. Most often, `y` is a 1D array of length `n_samples`.

All supervised estimators in scikit-learn implement a `fit(X, y)` method to fit the model and a `predict(X)` method that, given unlabeled observations `X`, returns the predicted labels `y`.

Vocabulary: classification and regression

If the prediction task is to classify the observations in a set of finite labels, in other words to “name” the objects observed, the task is said to be a **classification** task. On the other hand, if the goal is to predict a continuous target variable, it is said to be a **regression** task.

When doing classification in scikit-learn, `y` is a vector of integers or strings.

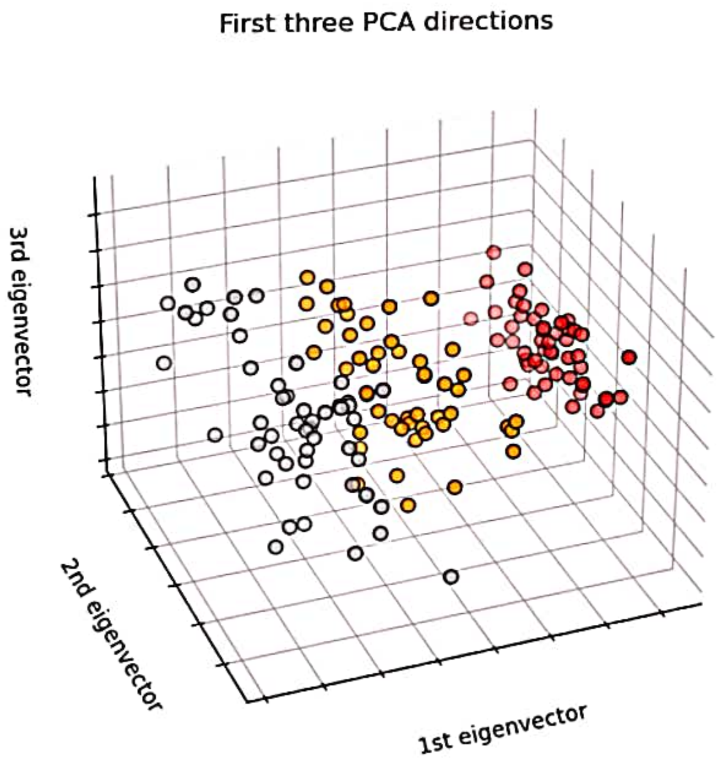
Note: See the Introduction to machine learning with scikit-learn Tutorial for a quick run-through on the basic machine learning vocabulary used within scikit-learn.

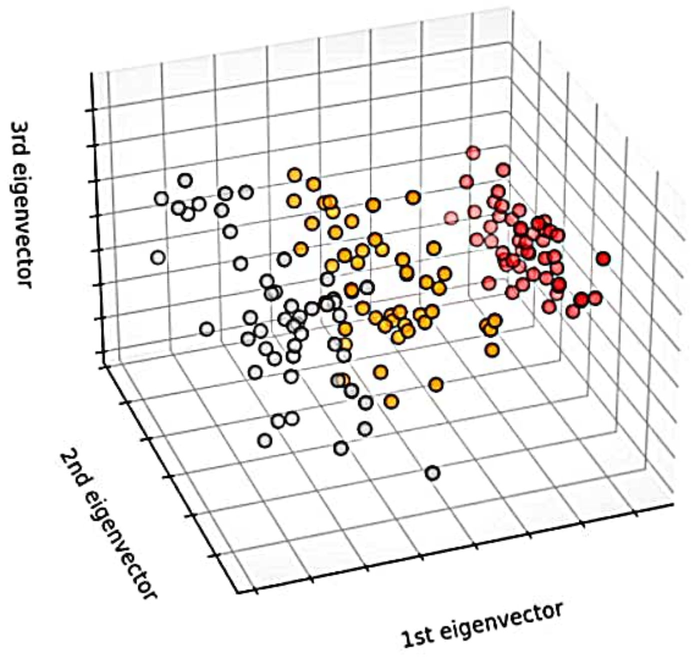
Nearest neighbor and the curse of dimensionality

Classifying irises:

The iris dataset is a classification task consisting in identifying 3 different types of irises (Setosa, Versicolour, and Virginica) from their petal and sepal length and width:

```
>>> import numpy as np
>>> from sklearn import datasets
>>> iris_X, iris_y = datasets.load_iris(re
>>> np.unique(iris_y)
array([0, 1, 2])
```





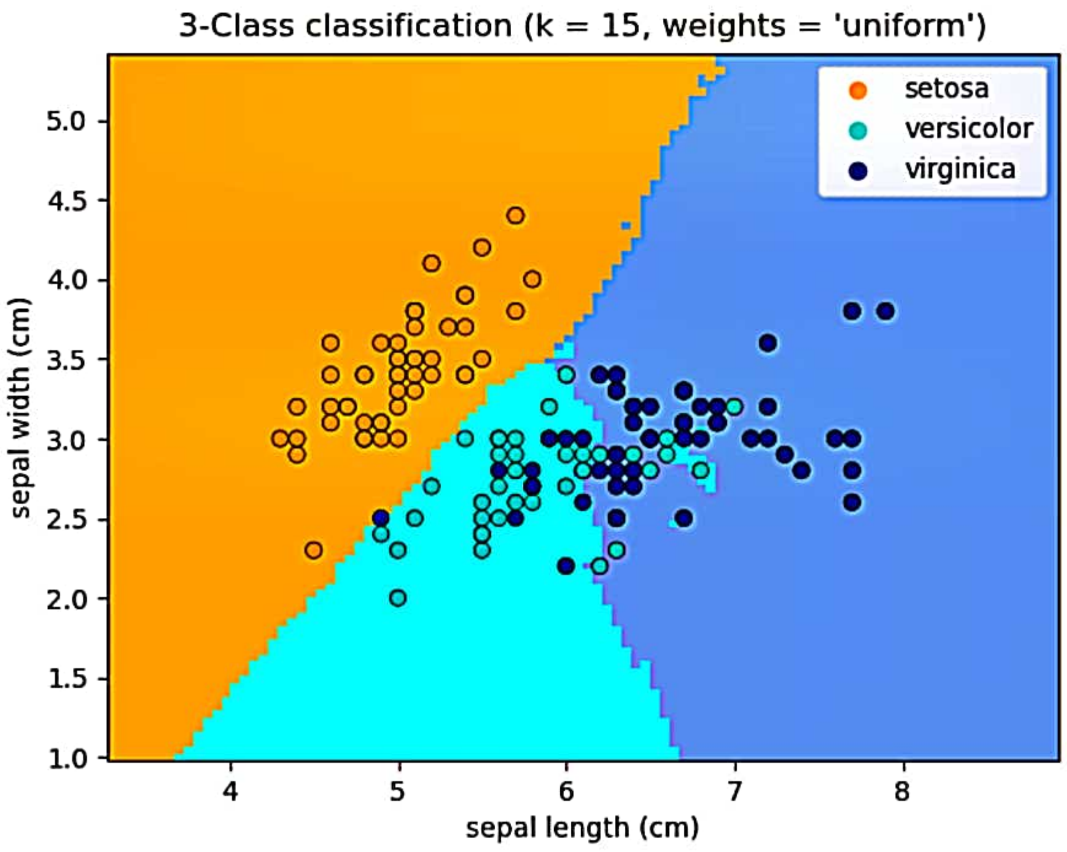
k-Nearest neighbors classifier

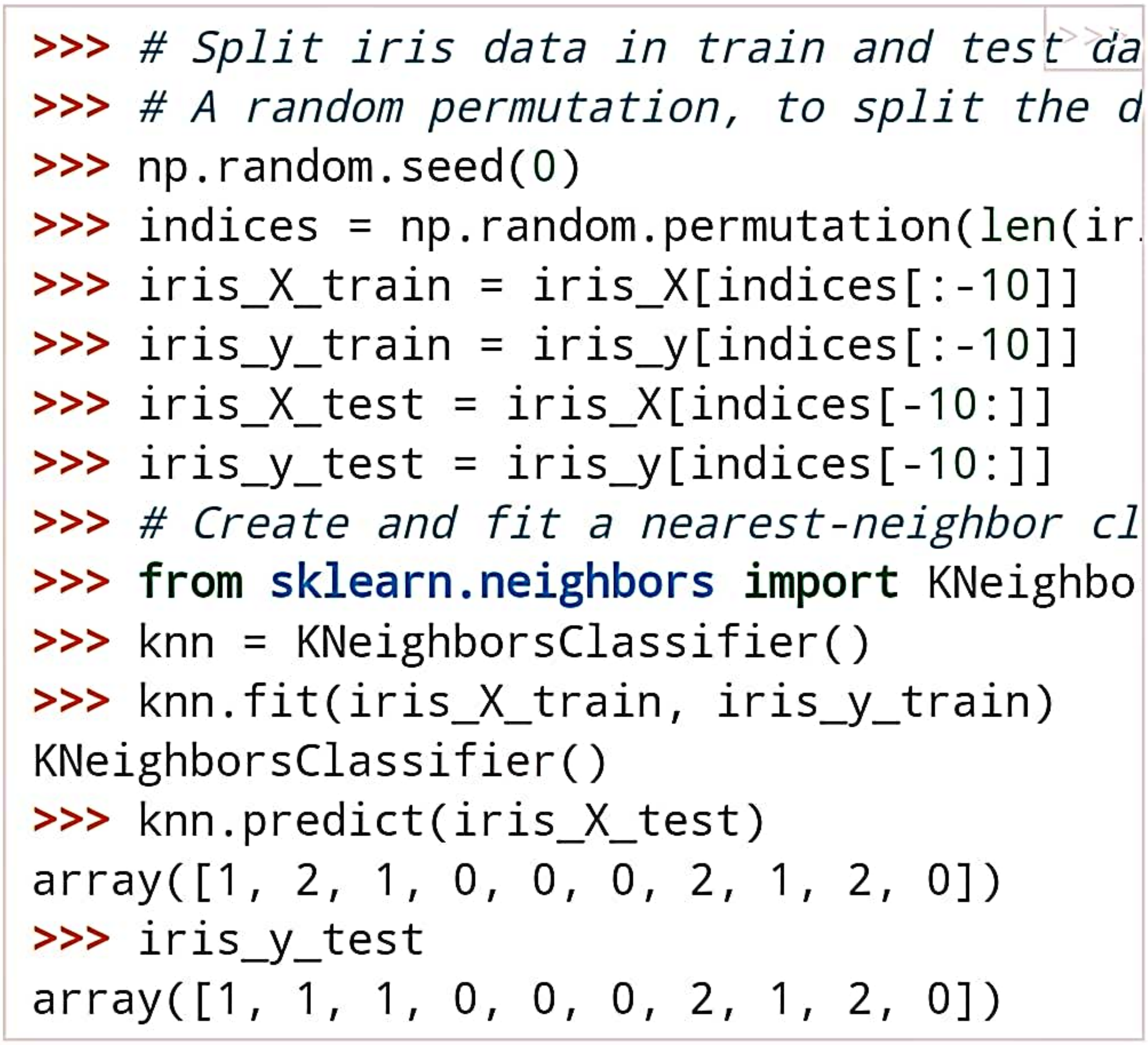
The simplest possible classifier is the nearest neighbor: given a new observation `X_test`, find in the training set (i.e. the data used to train the estimator) the observation with the closest feature vector. (Please see the Nearest Neighbors section of the online Scikit-learn documentation for more information about this type of classifier.)

Training set and testing set

While experimenting with any learning algorithm, it is important not to test the prediction of an estimator on the data used to fit the estimator as this would not be evaluating the performance of the estimator on **new data**. This is why datasets are often split into *train* and *test* data.

KNN (k nearest neighbors) classification example:




```
>>> # Split iris data in train and test da
>>> # A random permutation, to split the d
>>> np.random.seed(0)
>>> indices = np.random.permutation(len(ir
>>> iris_X_train = iris_X[indices[:-10]]
>>> iris_y_train = iris_y[indices[:-10]]
>>> iris_X_test = iris_X[indices[-10:]]
>>> iris_y_test = iris_y[indices[-10:]]
>>> # Create and fit a nearest-neighbor cl
>>> from sklearn.neighbors import KNeighbo
>>> knn = KNeighborsClassifier()
>>> knn.fit(iris_X_train, iris_y_train)
KNeighborsClassifier()
>>> knn.predict(iris_X_test)
array([1, 2, 1, 0, 0, 0, 2, 1, 2, 0])
>>> iris_y_test
array([1, 1, 1, 0, 0, 0, 2, 1, 2, 0])
```

The curse of dimensionality

For an estimator to be effective, you need the distance between neighboring points to be less than some value d, which depends on the problem. In one dimension, this requires on average $n \sim 1/d$ points. In the context of the above k-NN example, if the data is described by just one feature with values ranging from 0 to 1 and with n training observations, then new data will be no further away than $1/n$. Therefore, the nearest neighbor decision rule will be efficient as soon as $1/n$ is small compared to the scale of between-class feature variations.

If the number of features is p, you now require $n \sim 1/d^p$ points. Let's say that we require 10 points in one dimension: now 10^p points are required in p dimensions to pave the $[0, 1]$ space. As p becomes large, the number of training points required for a good estimator grows exponentially.

For example, if each point is just a single number (8 bytes), then an effective k-NN estimator in a paltry $p \sim 20$ dimensions would require more training data than the current estimated size of the entire internet (±1000 Exabytes or so).

This is called the curse of dimensionality and is a core problem that machine learning addresses.

Linear model: from regression to sparsity

Diabetes dataset

The diabetes dataset consists of 10 physiological variables (age, sex, weight, blood pressure) measured on 442 patients, and an indication of disease progression after one year:

```
>>> diabetes_X, diabetes_y = datasets.load
>>> diabetes_X_train = diabetes_X[:-20]
>>> diabetes_X_test  = diabetes_X[-20:]
>>> diabetes_y_train = diabetes_y[:-20]
>>> diabetes_y_test  = diabetes_y[-20:]
```

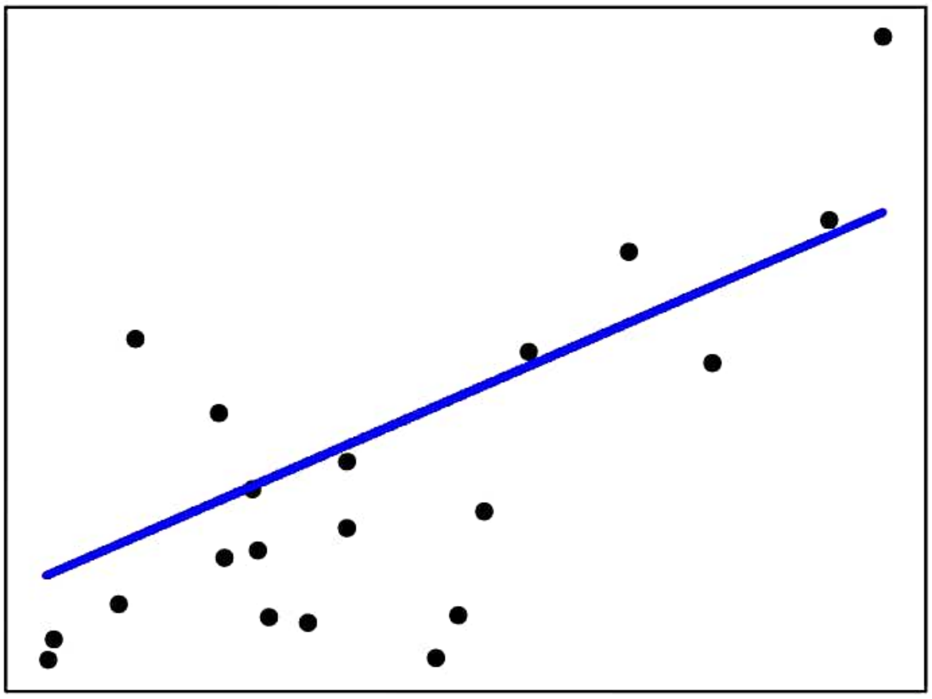
The task at hand is to predict disease progression from physiological variables.

Linear regression

`LinearRegression`, in its simplest form, fits a linear model to the data set by adjusting a set of parameters in order to make the sum of the squared residuals of the model as small as possible.

Linear models: $y = X\beta + \epsilon$

- X: data
- y: target variable
- β: Coefficients
- ϵ: Observation noise

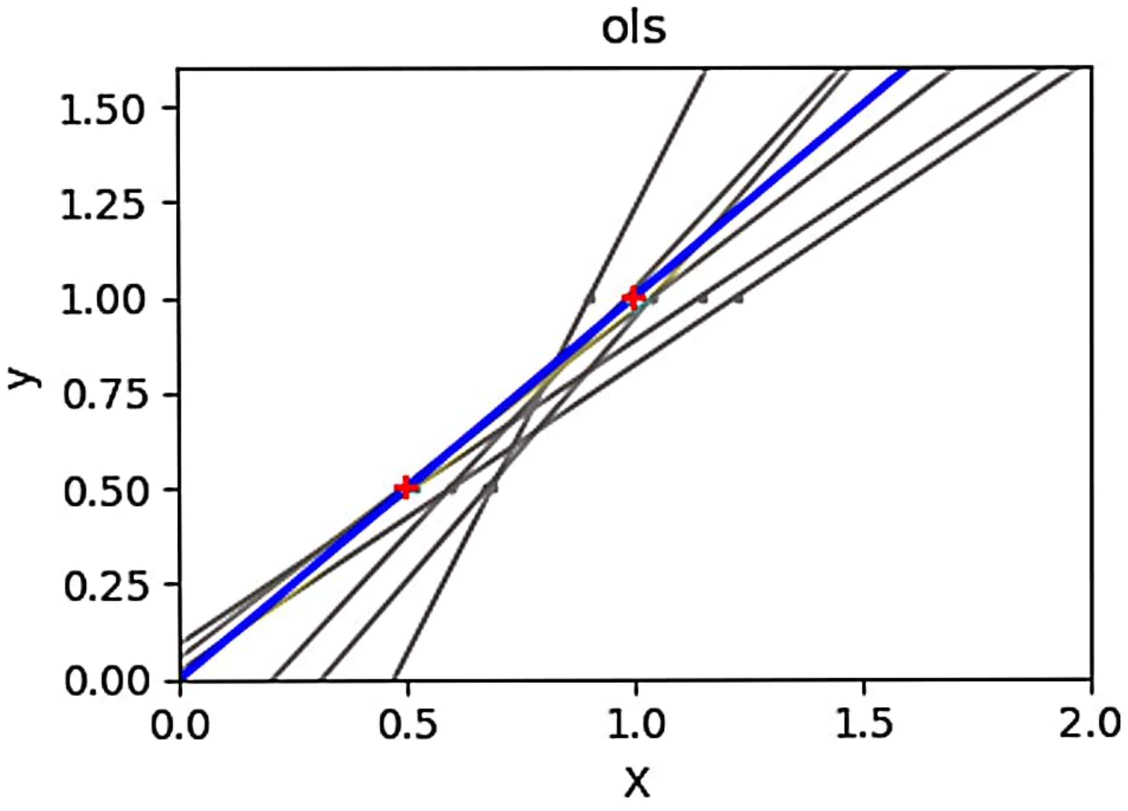


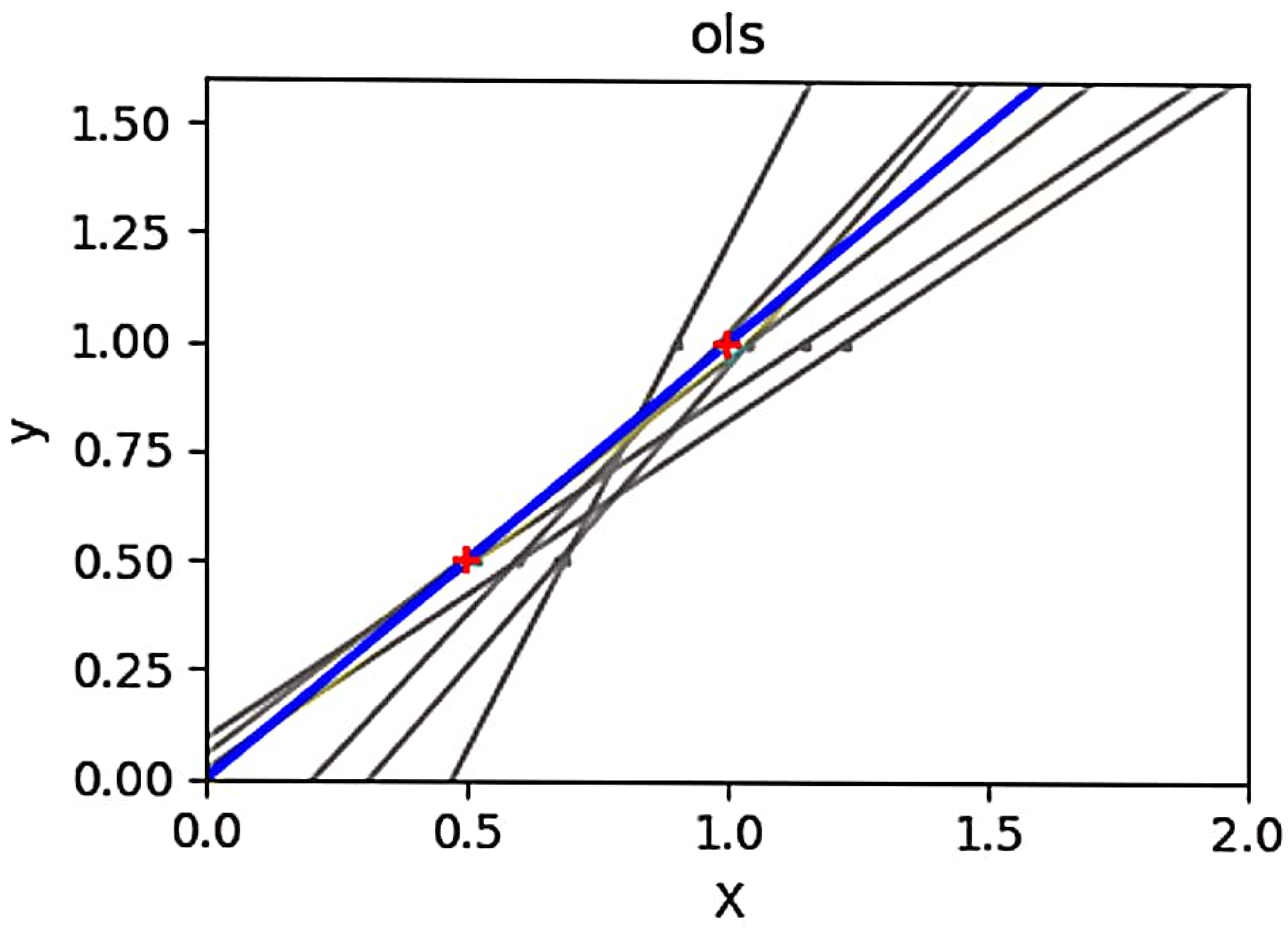
```
>>> from sklearn import linear_model                          >>>
>>> regr = linear_model.LinearRegression()
>>> regr.fit(diabetes_X_train, diabetes_y_
LinearRegression()
>>> print(regr.coef_)
[   0.30349955 -237.63931533  510.53060544
  492.81458798  102.84845219  184.60648906



>>> # The mean square error
>>> np.mean((regr.predict(diabetes_X_test)
2004.5...

>>> # Explained variance score: 1 is perfe
>>> # and 0 means that there is no linear
>>> # between X and y.
>>> regr.score(diabetes_X_test, diabetes_y
0.585...
```

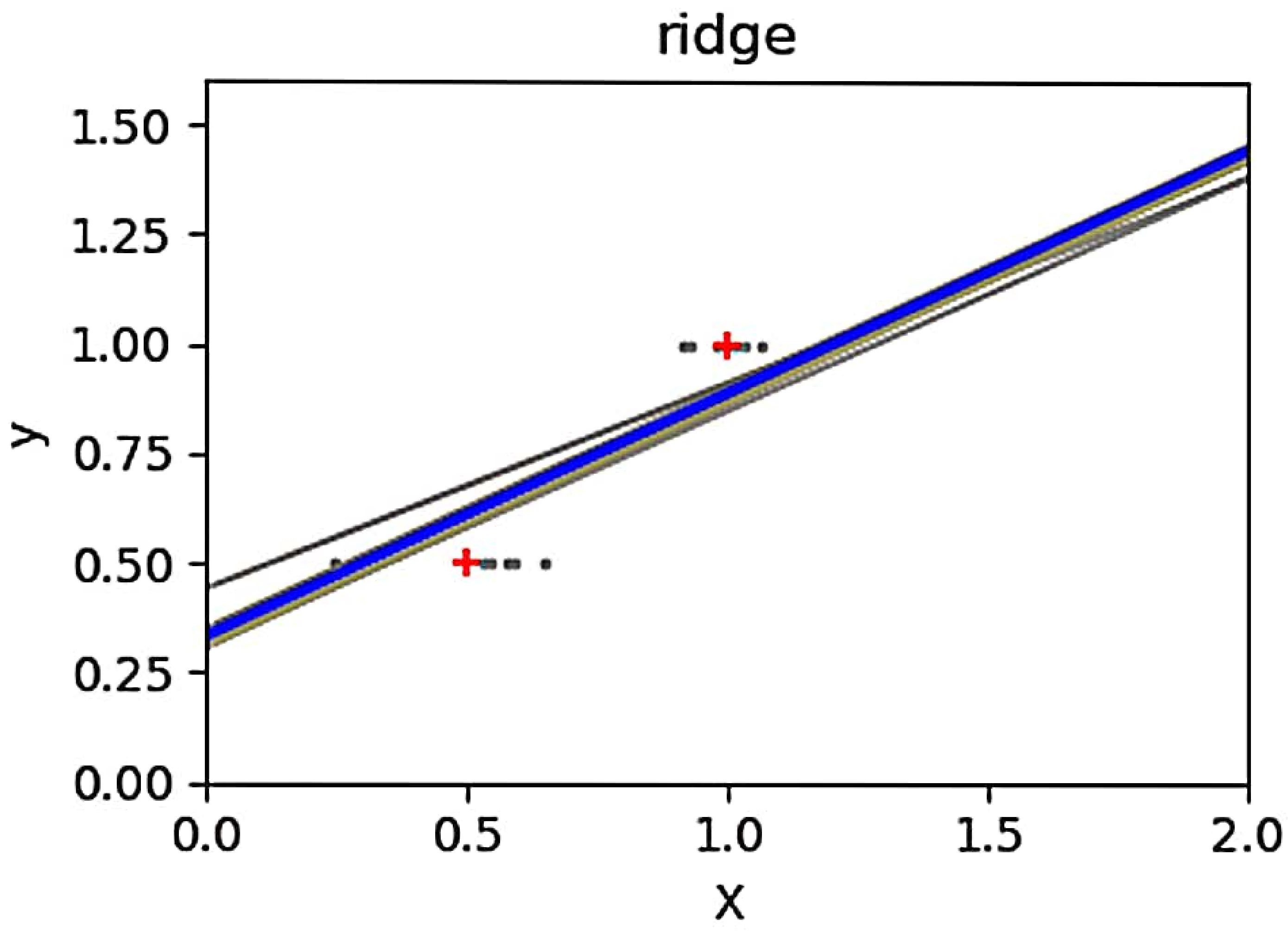
Shrinkage ¶

f there are few data points per dimension, noise in the observations induces high variance:

```
>>> X = np.c_[ .5, 1].T
>>> y = [.5, 1]
>>> test = np.c_[ 0, 2].T
>>> regr = linear_model.LinearRegression()

>>> import matplotlib.pyplot as plt
>>> plt.figure()
<...>
>>> np.random.seed(0)
>>> for _ in range(6):
...     this_X = .1 * np.random.normal(siz
...     regr.fit(this_X, y)
...     plt.plot(test, regr.predict(test))
...     plt.scatter(this_X, y, s=3)
LinearRegression...
```




ols
1.50
1.25
1.00
0.75
0.50
0.25
0.00
y
0.0
0.5
1.0
1.5
2.0
X


ridge
1.50
1.25
1.00
0.75
0.50
0.25
0.00
y
0.0
0.5
1.0
1.5
2.0
X