STATS 507
Data Analysis in Python

Lecture 23: scikit-learn

scikit-learn

Open-source Python machine learning library

    Built atop numpy, scipy and matplotlib

Makes many common ML/stats models easily available

    API supports simple model fitting, prediction, cross-validation, etc.

Installation:

    `pip install scikit-learn` (or `conda install scikit-learn`)

    ...or install from source (still not recommended)

Example: classifiers in scikit-learn

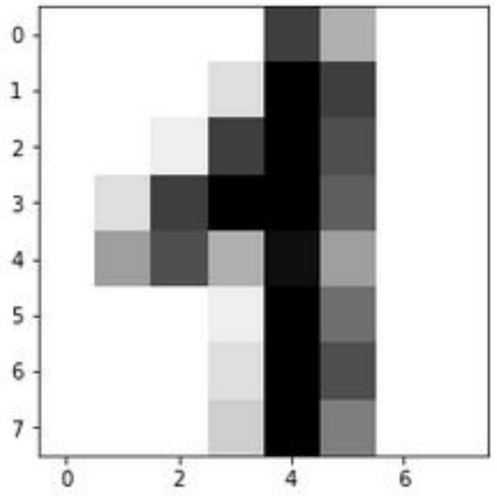
```
from sklearn import datasets
digits = datasets.load_digits()
digits.target[42]
```

```
1
```

`sklearn` includes a number of built-in data sets, among which is a version of the famous MNIST digits data set. We'll see more of this when we discuss TensorFlow.

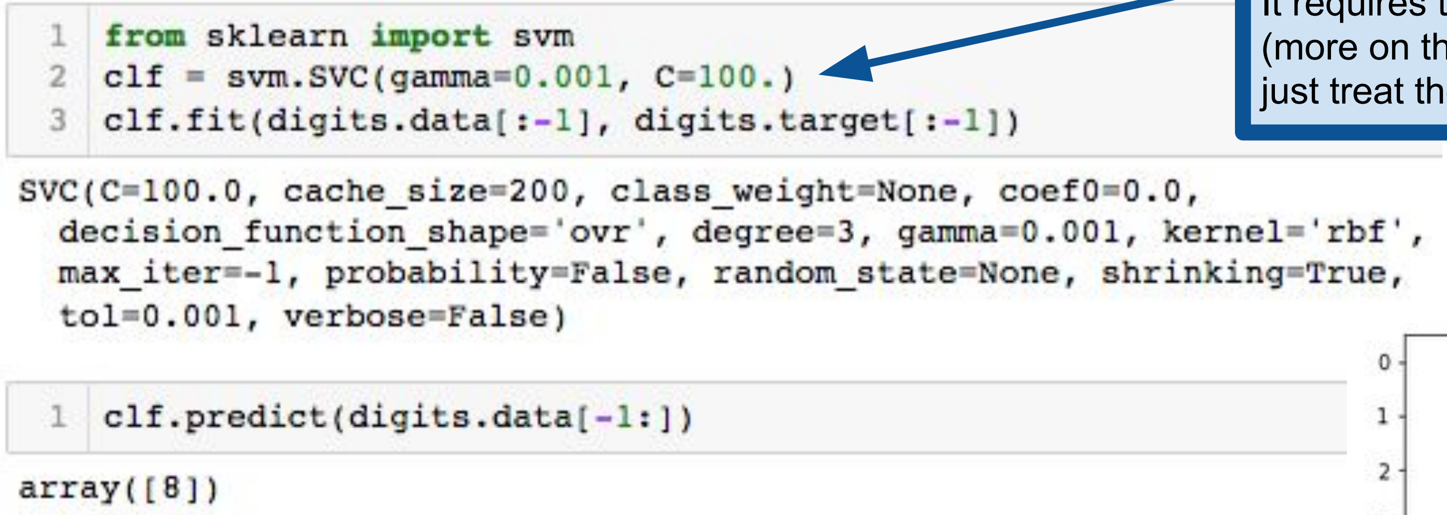
```
plt.imshow(np.reshape(digits.data[42], (8,8)), cmap='binary')
```

```
<matplotlib.image.AxesImage at 0x109815748>
```



`digits.data` is an array, entries of which are 64-dimensional vectors, which correspond to images. To display them, we have to reshape them to 8-by-8. The `cmap` argument specifies a color map. See https://matplotlib.org/users/colormaps.html

Example: classifiers in scikit-learn



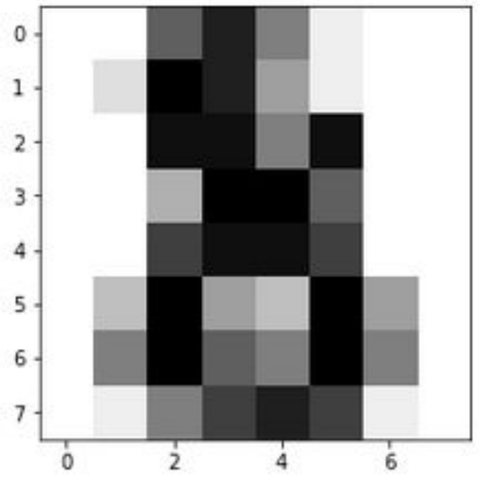
```
from sklearn import svm
clf = svm.SVC(gamma=0.001, C=100.)
clf.fit(digits.data[:-1], digits.target[:-1])
```

```
SVC(C=100.0, cache_size=200, class_weight=None, coef0=0.0,
  decision_function_shape='ovr', degree=3, gamma=0.001, kernel='rbf',
  max_iter=-1, probability=False, random_state=None, shrinking=True,
  tol=0.001, verbose=False)
```

```
clf.predict(digits.data[-1:])
```

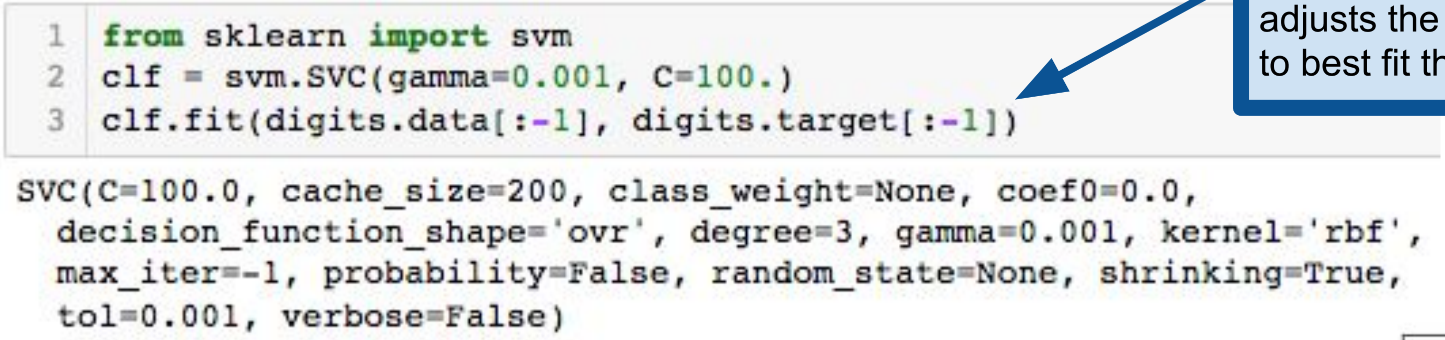
```
array([8])
```

SVC is a support vector machine (SVM) classifier, one of many classifiers that `sklearn` provides. It requires two hyperparameters (more on these soon, but for now just treat them as magic).



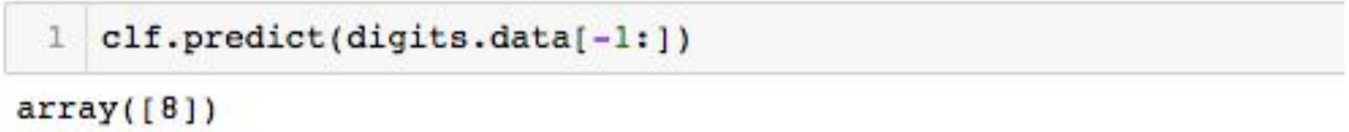
Example: classifiers in scikit-learn

Every classifier object supports a `fit` method, which takes observations and labels and adjusts the model parameters to best fit that data.



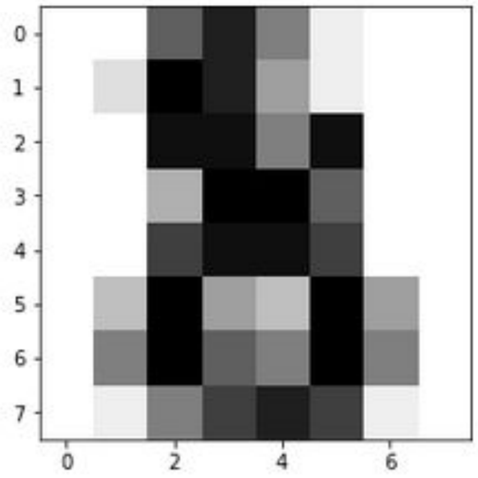
```
from sklearn import svm
clf = svm.SVC(gamma=0.001, C=100.)
clf.fit(digits.data[:-1], digits.target[:-1])
```

```
SVC(C=100.0, cache_size=200, class_weight=None, coef0=0.0,
  decision_function_shape='ovr', degree=3, gamma=0.001, kernel='rbf',
  max_iter=-1, probability=False, random_state=None, shrinking=True,
  tol=0.001, verbose=False)
```



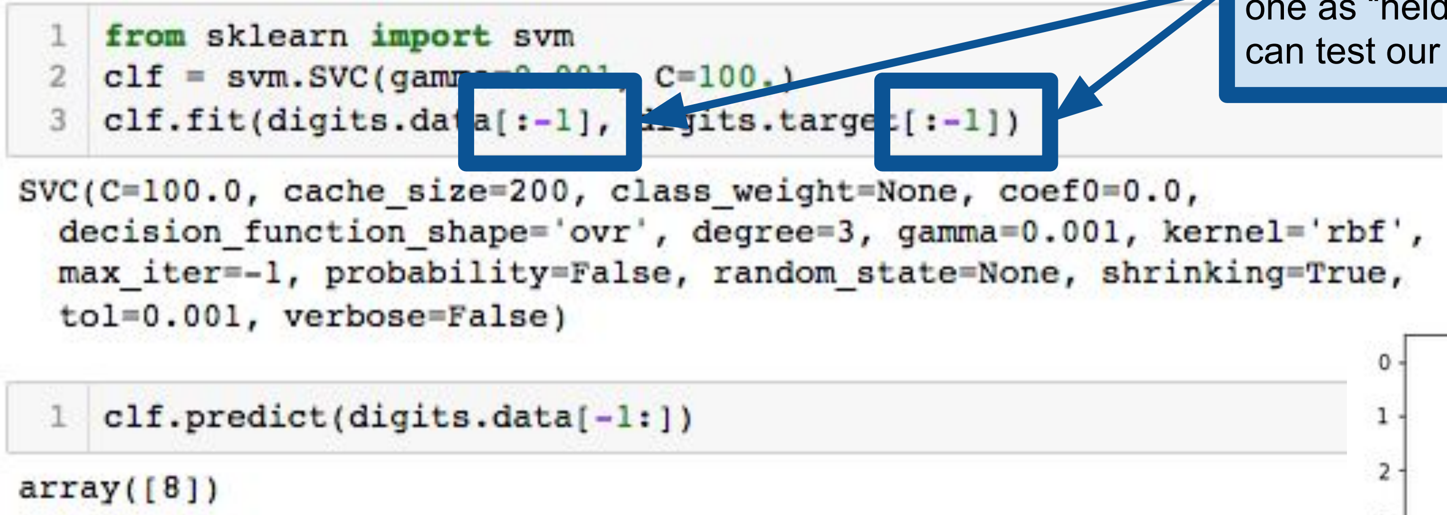
```
clf.predict(digits.data[-1:])
```

```
array([8])
```



Example: classifiers in scikit-learn

We are training on all but one of the digits in the collection, keeping one as “held out” data on which we can test our classifier.

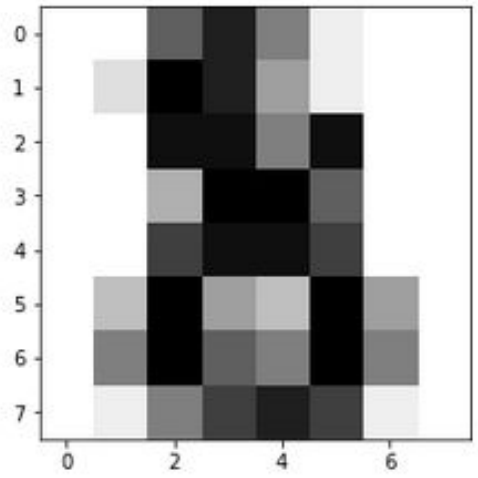


```
from sklearn import svm
clf = svm.SVC(gamma=0.001, C=100.)
clf.fit(digits.data[:-1], digits.target[:-1])
```

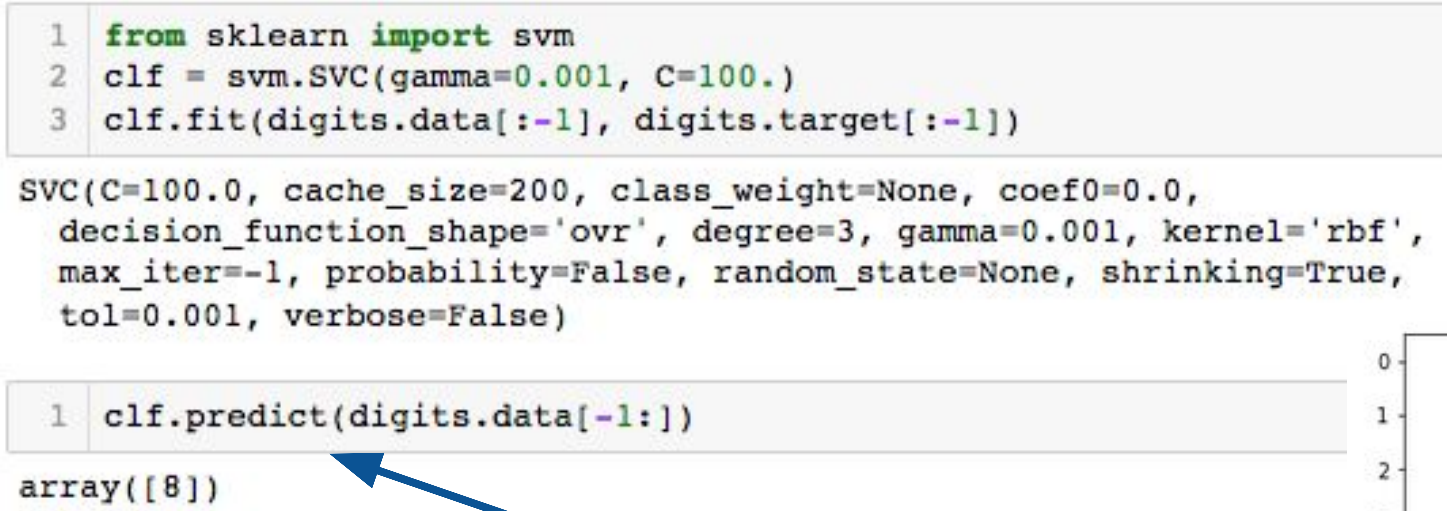
```
SVC(C=100.0, cache_size=200, class_weight=None, coef0=0.0,
  decision_function_shape='ovr', degree=3, gamma=0.001, kernel='rbf',
  max_iter=-1, probability=False, random_state=None, shrinking=True,
  tol=0.001, verbose=False)
```

```
clf.predict(digits.data[-1:])
```

```
array([8])
```



Example: classifiers in scikit-learn



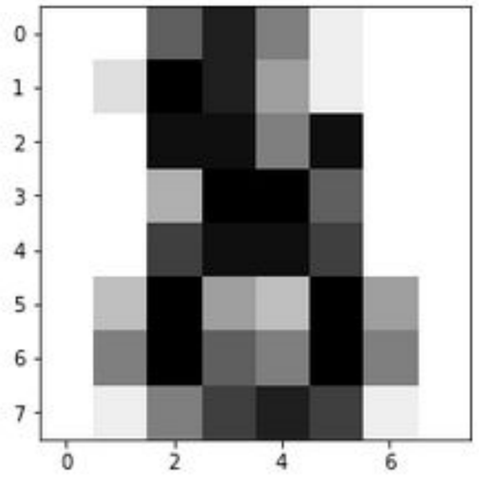
```
from sklearn import svm
clf = svm.SVC(gamma=0.001, C=100.)
clf.fit(digits.data[:-1], digits.target[:-1])
```

```
SVC(C=100.0, cache_size=200, class_weight=None, coef0=0.0,
  decision_function_shape='ovr', degree=3, gamma=0.001, kernel='rbf',
  max_iter=-1, probability=False, random_state=None, shrinking=True,
  tol=0.001, verbose=False)
```

```
clf.predict(digits.data[-1:])
```

```
array([8])
```

Every classifier object also supports a `predict` method, which takes an observation and tries to guess the “best” label for it, based on the model parameters.



Supervised learning example: LASSO in `sklearn`

Review: linear regression

$$y = X\beta + \epsilon$$

Predictors $X \in \mathbb{R}^{n \times d}$

Coefficients $\beta \in \mathbb{R}^d$

Noise $\epsilon \in \mathbb{R}^n$

Response $y \in \mathbb{R}^n$

Ordinary least squares (OLS)

$$\min_{\beta} \frac{1}{n} \sum_{i=1}^{n} (y_i - \beta^T x_i)^2$$

Minimizes the sum of the squared residuals between the response and the model prediction. OLS is computationally convenient. The solution can be expressed as an expression of X.

Supervised learning example: LASSO in `sklearn`

Review: linear regression

$$y = X\beta + \epsilon$$

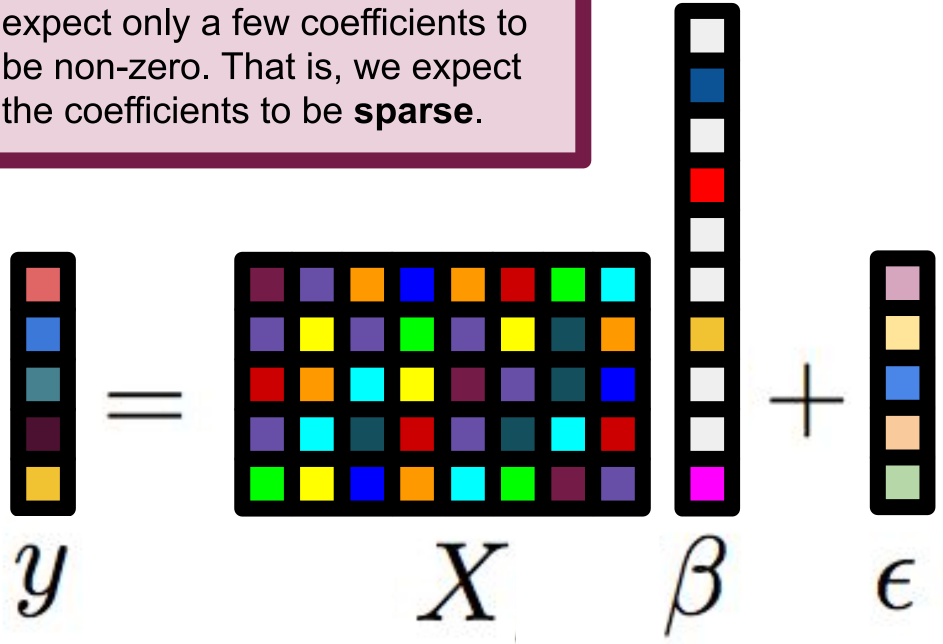
Predictors $X \in \mathbb{R}^{n\times d}$

Coefficients $\beta \in \mathbb{R}^{d}$

Noise $\epsilon \in \mathbb{R}^{n}$

Response $y \in \mathbb{R}^{n}$

In many applications (e.g., audio, genomics, medical imaging), we expect only a few coefficients to be non-zero. That is, we expect the coefficients to be **sparse**.



Supervised learning example: LASSO in `sklearn`

Review: linear regression

$$y = X\beta + \epsilon$$

Predictors $X \in \mathbb{R}^{n \times d}$

Coefficients $\beta \in \mathbb{R}^d$

Noise $\epsilon \in \mathbb{R}^n$

Response $y \in \mathbb{R}^n$

Ordinary least squares (OLS)

$$\min_{\beta} \frac{1}{n} \sum_{i=1}^{n} (y_i - \beta^T x_i)^2$$

LASSO

$$\min_{\beta} \frac{1}{n} \sum_{i=1}^{n} (y_i - \beta^T x_i)^2 + \boxed{\alpha \sum_{i=1}^{n} |\beta_i|}$$

This penalty term discourages non-zero coefficients. The larger α is, the more we are penalized for having non-zero coefficients.

Supervised learning example: LASSO in `sklearn`

Review: linear regression

$$y = X\beta + \epsilon$$

Predictors $X \in \mathbb{R}^{n \times d}$

Coefficients $\beta \in \mathbb{R}^{d}$

Noise $\epsilon \in \mathbb{R}^{n}$

Response $y \in \mathbb{R}^{n}$

LASSO (equivalent formulation)

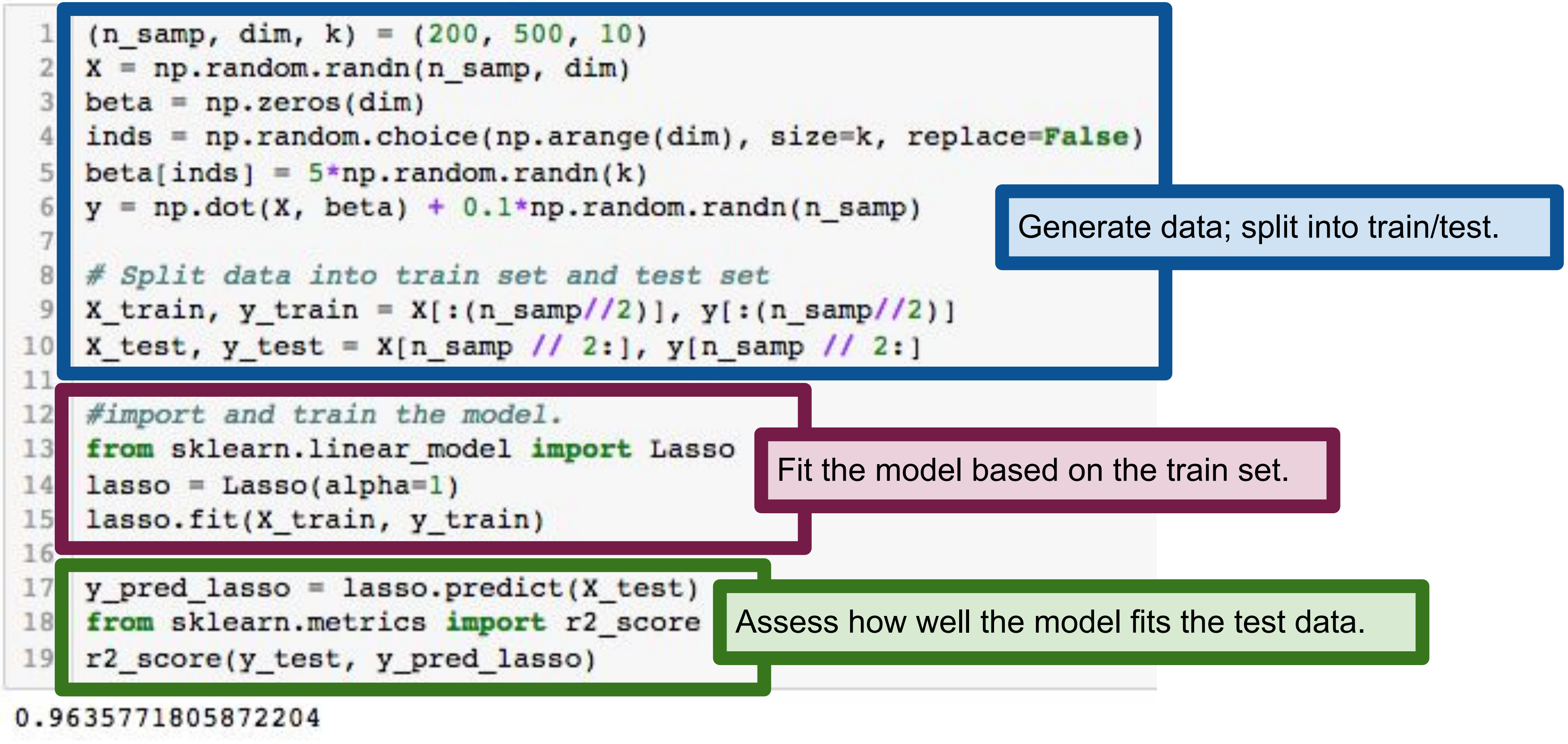
$$\min_{\beta} \underbrace{\boxed{\frac{1}{n}\|y - \beta^T X\|^2}}_{\text{L2 objective}} + \underbrace{\boxed{\alpha\|\beta\|_1}}_{\text{L1 penalty}}$$

L2 objective

L1 penalty

The key tradeoff here is that whereas OLS had a nice closed-form solution, we have to find a solution to the LASSO using optimization techniques, but that's okay, because sklearn will solve the optimization for us.

Supervised learning example: LASSO in `sklearn`



```
(n_samp, dim, k) = (200, 500, 10)
X = np.random.randn(n_samp, dim)
beta = np.zeros(dim)
inds = np.random.choice(np.arange(dim), size=k, replace=False)
beta[inds] = 5*np.random.randn(k)
y = np.dot(X, beta) + 0.1*np.random.randn(n_samp)

# Split data into train set and test set
X_train, y_train = X[:(n_samp//2)], y[:(n_samp//2)]
X_test, y_test = X[n_samp // 2:], y[n_samp // 2:]

#import and train the model.
from sklearn.linear_model import Lasso
lasso = Lasso(alpha=1)
lasso.fit(X_train, y_train)

y_pred_lasso = lasso.predict(X_test)
from sklearn.metrics import r2_score
r2_score(y_test, y_pred_lasso)
```

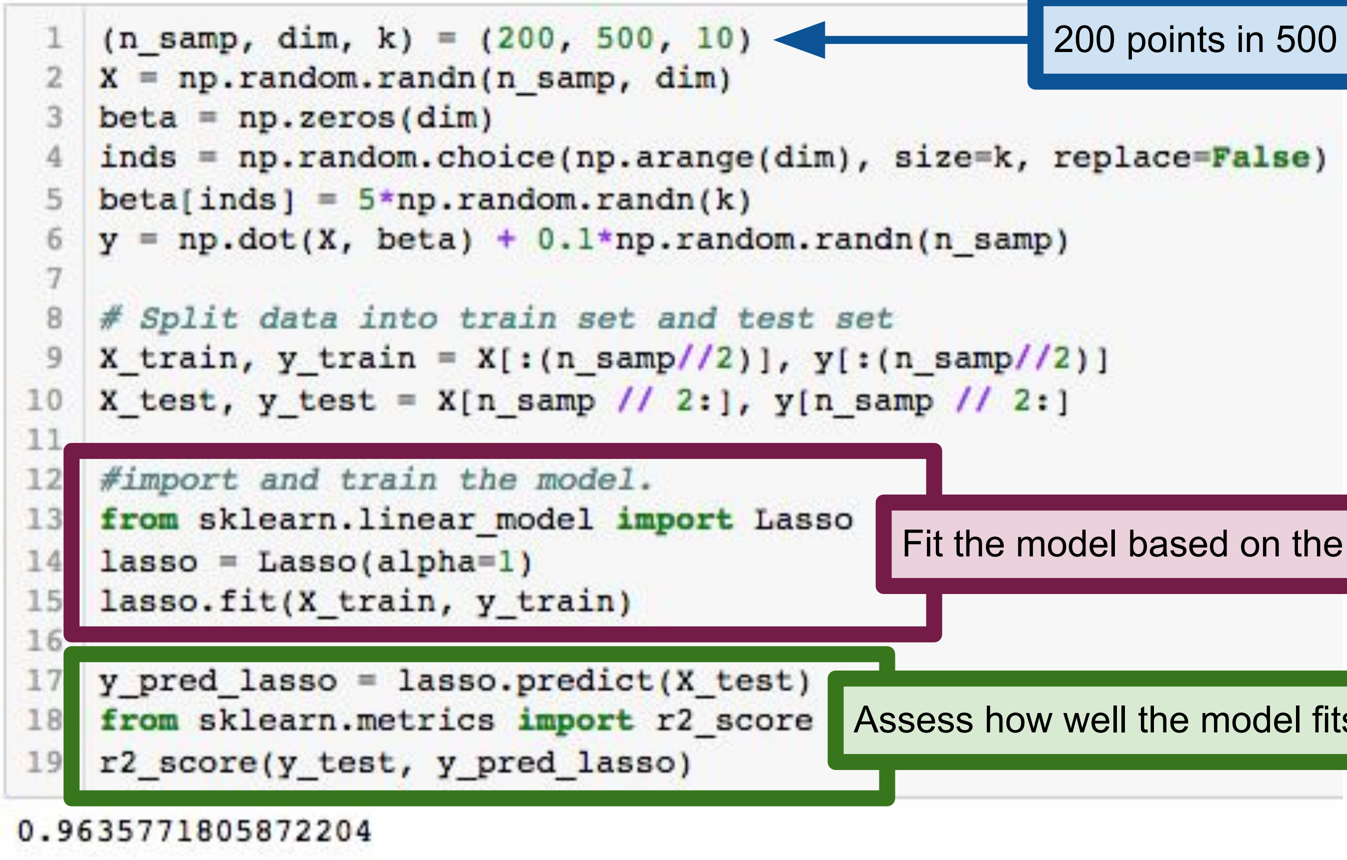
```
0.9635771805872204
```

Generate data; split into train/test.

Fit the model based on the train set.

Assess how well the model fits the test data.

Supervised learning example: LASSO in `sklearn`



```
(n_samp, dim, k) = (200, 500, 10)
X = np.random.randn(n_samp, dim)
beta = np.zeros(dim)
inds = np.random.choice(np.arange(dim), size=k, replace=False)
beta[inds] = 5*np.random.randn(k)
y = np.dot(X, beta) + 0.1*np.random.randn(n_samp)

# Split data into train set and test set
X_train, y_train = X[:(n_samp//2)], y[:(n_samp//2)]
X_test, y_test = X[n_samp // 2:], y[n_samp // 2:]

#import and train the model.
from sklearn.linear_model import Lasso
lasso = Lasso(alpha=1)
lasso.fit(X_train, y_train)

y_pred_lasso = lasso.predict(X_test)
from sklearn.metrics import r2_score
r2_score(y_test, y_pred_lasso)
```

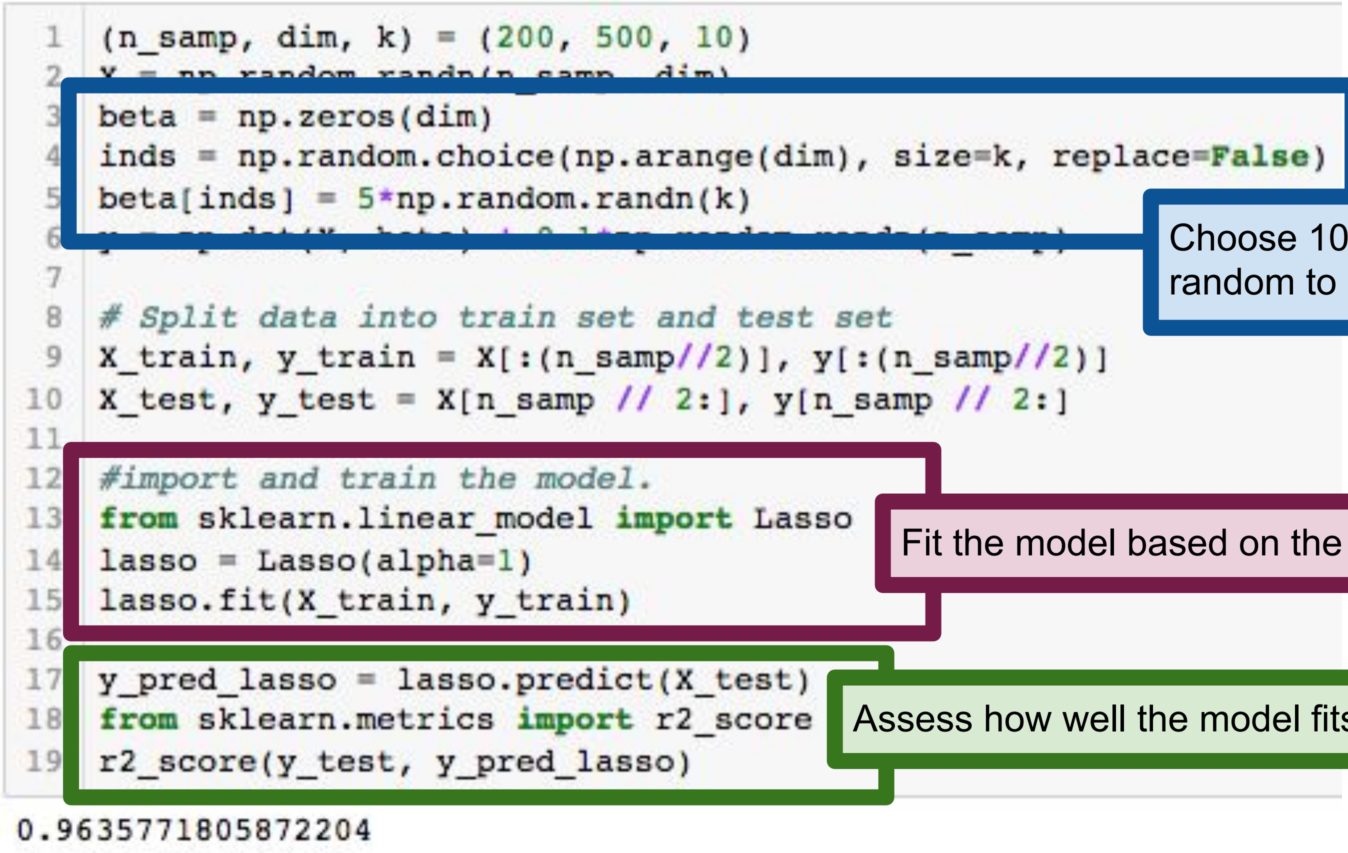
```
0.9635771805872204
```

200 points in 500 dimensions. Sparsity `k=10`.

Fit the model based on the train set.

Assess how well the model fits the test data.

Supervised learning example: LASSO in `sklearn`



```python
(n_samp, dim, k) = (200, 500, 10)
X = np.random.randn(n_samp, dim)
beta = np.zeros(dim)
inds = np.random.choice(np.arange(dim), size=k, replace=False)
beta[inds] = 5*np.random.randn(k)
y = np.dot(X, beta) + 0.1*np.random.randn(n_samp)

# Split data into train set and test set
X_train, y_train = X[:(n_samp//2)], y[:(n_samp//2)]
X_test, y_test = X[n_samp // 2:], y[n_samp // 2:]

#import and train the model.
from sklearn.linear_model import Lasso
lasso = Lasso(alpha=1)
lasso.fit(X_train, y_train)

y_pred_lasso = lasso.predict(X_test)
from sklearn.metrics import r2_score
r2_score(y_test, y_pred_lasso)
```

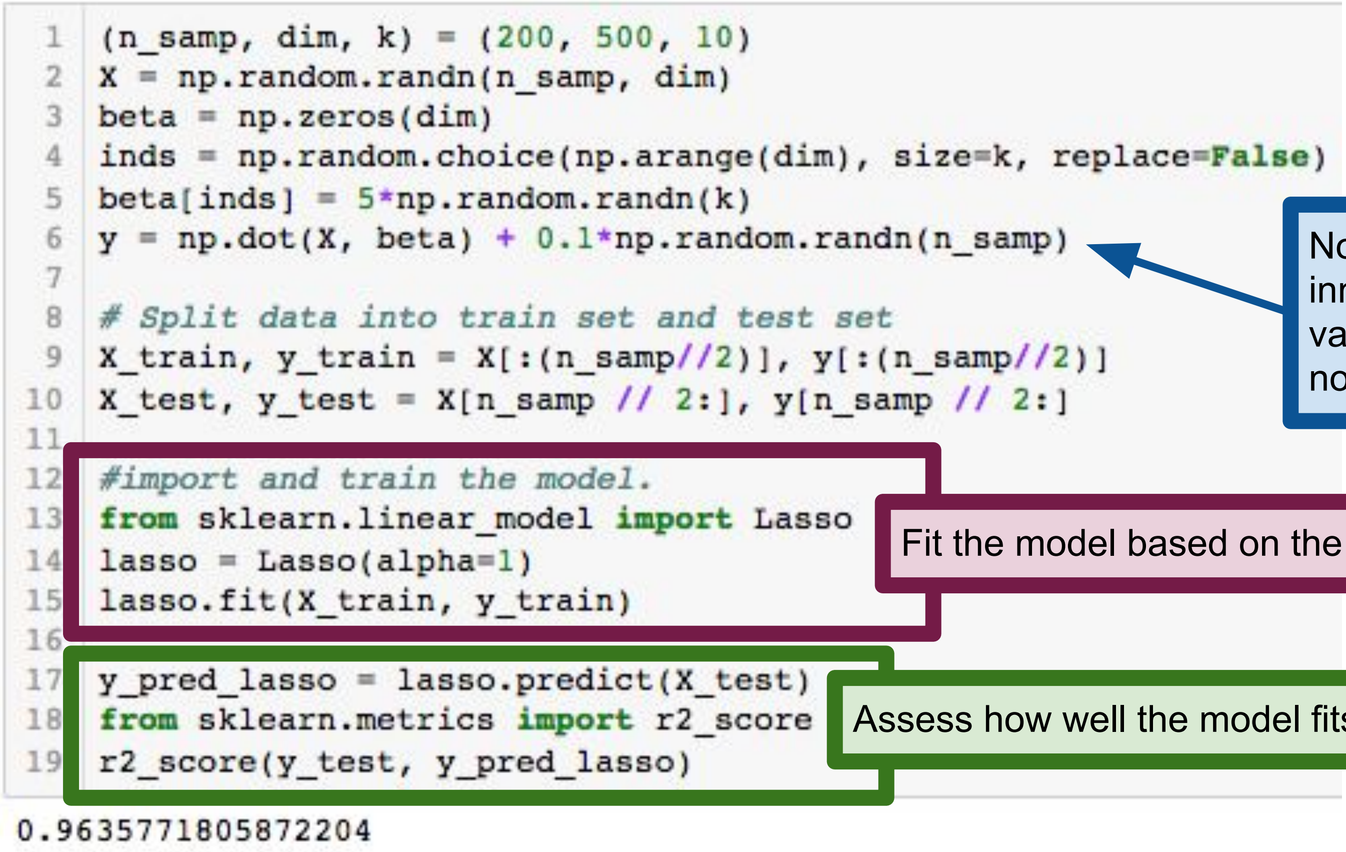
```
0.9635771805872204
```

Choose 10 coefficients at random to be nonzero.

Fit the model based on the train set.

Assess how well the model fits the test data.

Supervised learning example: LASSO in `sklearn`



```python
(n_samp, dim, k) = (200, 500, 10)
X = np.random.randn(n_samp, dim)
beta = np.zeros(dim)
inds = np.random.choice(np.arange(dim), size=k, replace=False)
beta[inds] = 5*np.random.randn(k)
y = np.dot(X, beta) + 0.1*np.random.randn(n_samp)

# Split data into train set and test set
X_train, y_train = X[:(n_samp//2)], y[:(n_samp//2)]
X_test, y_test = X[n_samp // 2:], y[n_samp // 2:]

#import and train the model.
from sklearn.linear_model import Lasso
lasso = Lasso(alpha=1)
lasso.fit(X_train, y_train)

y_pred_lasso = lasso.predict(X_test)
from sklearn.metrics import r2_score
r2_score(y_test, y_pred_lasso)
```

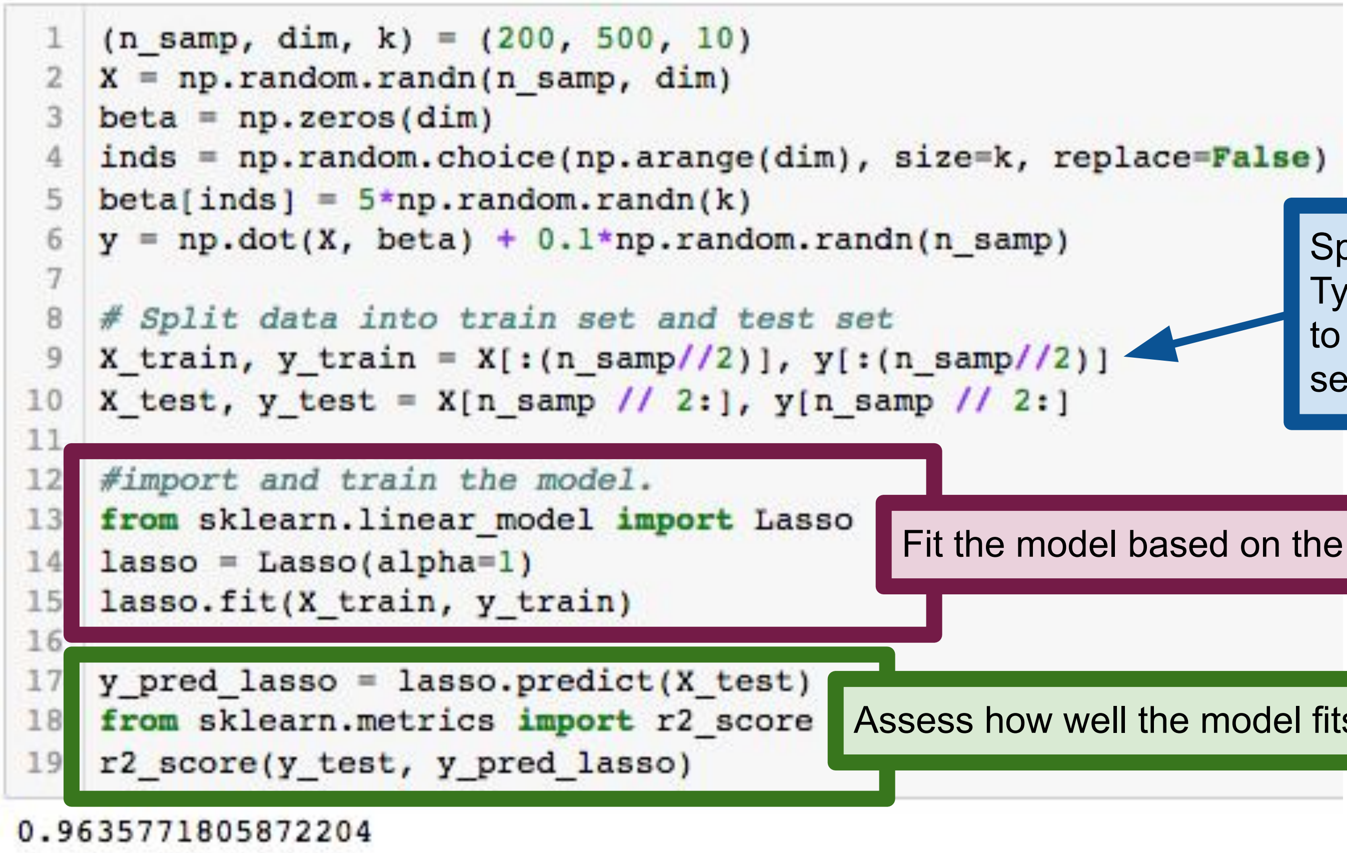
```
0.9635771805872204
```

Now generate the responses: inner product of independent variable with coefficients, plus normal noise.

Fit the model based on the train set.

Assess how well the model fits the test data.

Supervised learning example: LASSO in `sklearn`



```
(n_samp, dim, k) = (200, 500, 10)
X = np.random.randn(n_samp, dim)
beta = np.zeros(dim)
inds = np.random.choice(np.arange(dim), size=k, replace=False)
beta[inds] = 5*np.random.randn(k)
y = np.dot(X, beta) + 0.1*np.random.randn(n_samp)

# Split data into train set and test set
X_train, y_train = X[:(n_samp//2)], y[:(n_samp//2)]
X_test, y_test = X[n_samp // 2:], y[n_samp // 2:]

#import and train the model.
from sklearn.linear_model import Lasso
lasso = Lasso(alpha=1)
lasso.fit(X_train, y_train)

y_pred_lasso = lasso.predict(X_test)
from sklearn.metrics import r2_score
r2_score(y_test, y_pred_lasso)
```

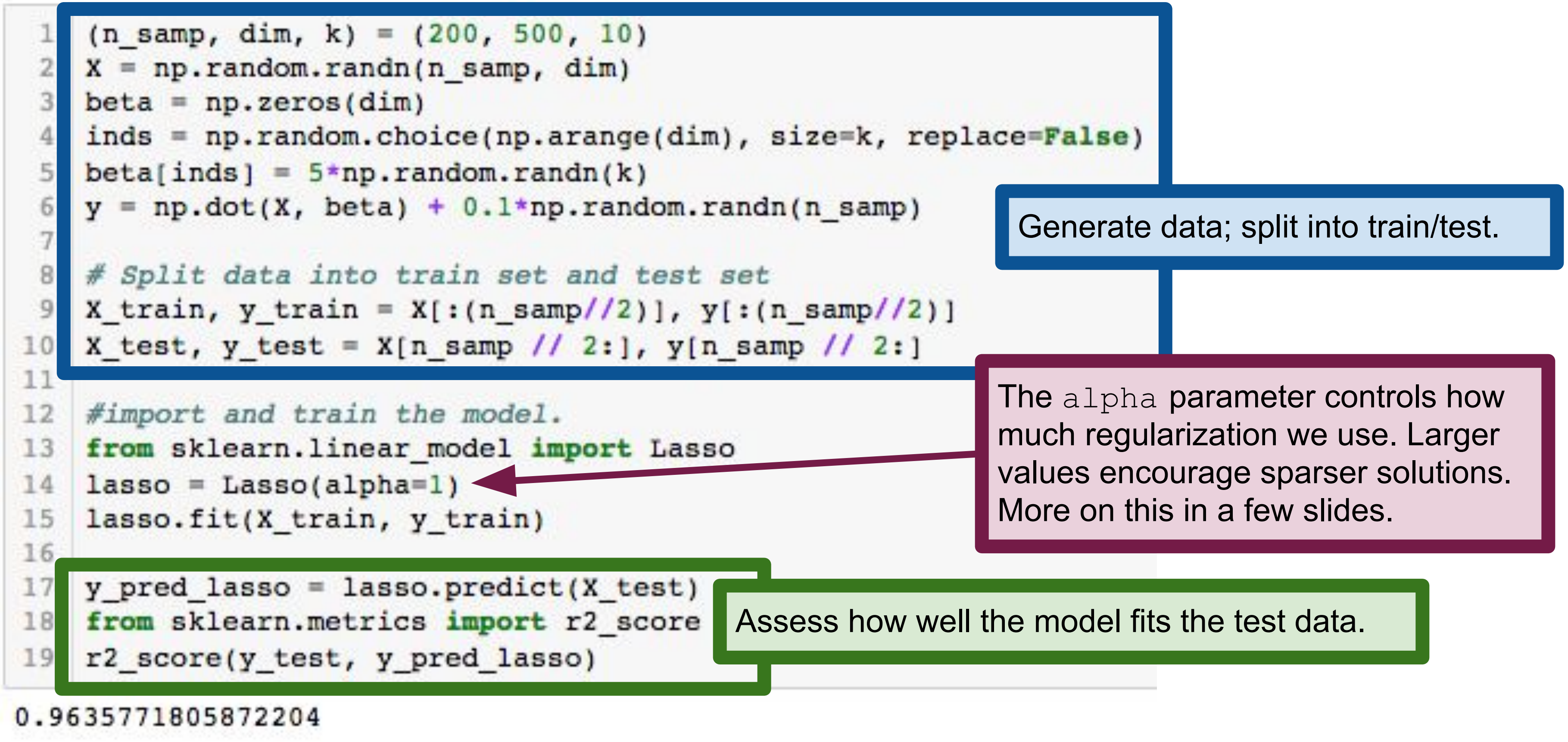
```
0.9635771805872204
```

Split into train and test sets. Typically the train set is chosen to be much larger than the test set, but this is just demo code.

Fit the model based on the train set.

Assess how well the model fits the test data.

Supervised learning example: LASSO in `sklearn`



```
(n_samp, dim, k) = (200, 500, 10)
X = np.random.randn(n_samp, dim)
beta = np.zeros(dim)
inds = np.random.choice(np.arange(dim), size=k, replace=False)
beta[inds] = 5*np.random.randn(k)
y = np.dot(X, beta) + 0.1*np.random.randn(n_samp)

# Split data into train set and test set
X_train, y_train = X[:(n_samp//2)], y[:(n_samp//2)]
X_test, y_test = X[n_samp // 2:], y[n_samp // 2:]

#import and train the model.
from sklearn.linear_model import Lasso
lasso = Lasso(alpha=1)
lasso.fit(X_train, y_train)

y_pred_lasso = lasso.predict(X_test)
from sklearn.metrics import r2_score
r2_score(y_test, y_pred_lasso)
```

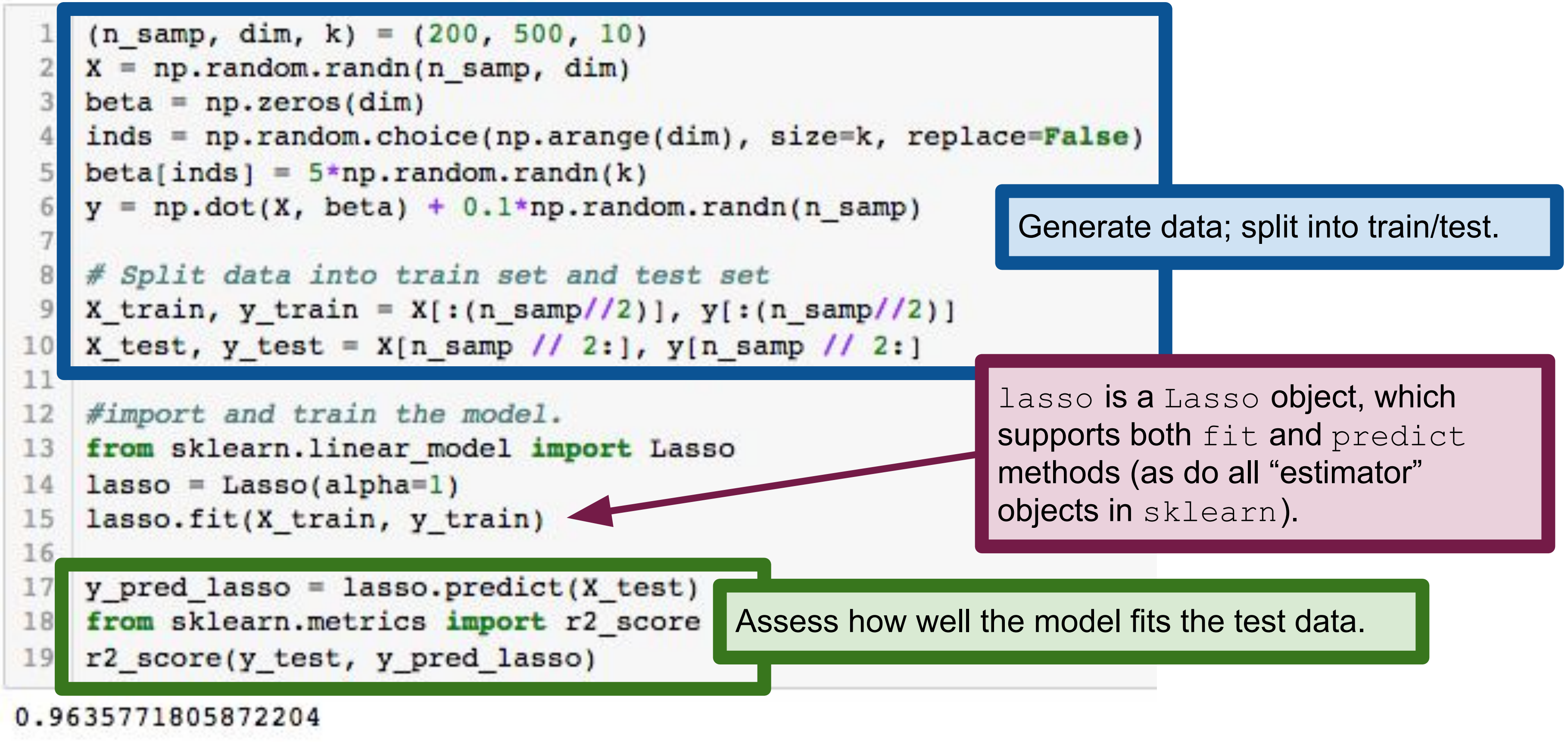
```
0.9635771805872204
```

Generate data; split into train/test.

The `alpha` parameter controls how much regularization we use. Larger values encourage sparser solutions. More on this in a few slides.

Assess how well the model fits the test data.

Supervised learning example: LASSO in `sklearn`



```
(n_samp, dim, k) = (200, 500, 10)
X = np.random.randn(n_samp, dim)
beta = np.zeros(dim)
inds = np.random.choice(np.arange(dim), size=k, replace=False)
beta[inds] = 5*np.random.randn(k)
y = np.dot(X, beta) + 0.1*np.random.randn(n_samp)

# Split data into train set and test set
X_train, y_train = X[:(n_samp//2)], y[:(n_samp//2)]
X_test, y_test = X[n_samp // 2:], y[n_samp // 2:]

#import and train the model.
from sklearn.linear_model import Lasso
lasso = Lasso(alpha=1)
lasso.fit(X_train, y_train)

y_pred_lasso = lasso.predict(X_test)
from sklearn.metrics import r2_score
r2_score(y_test, y_pred_lasso)
```

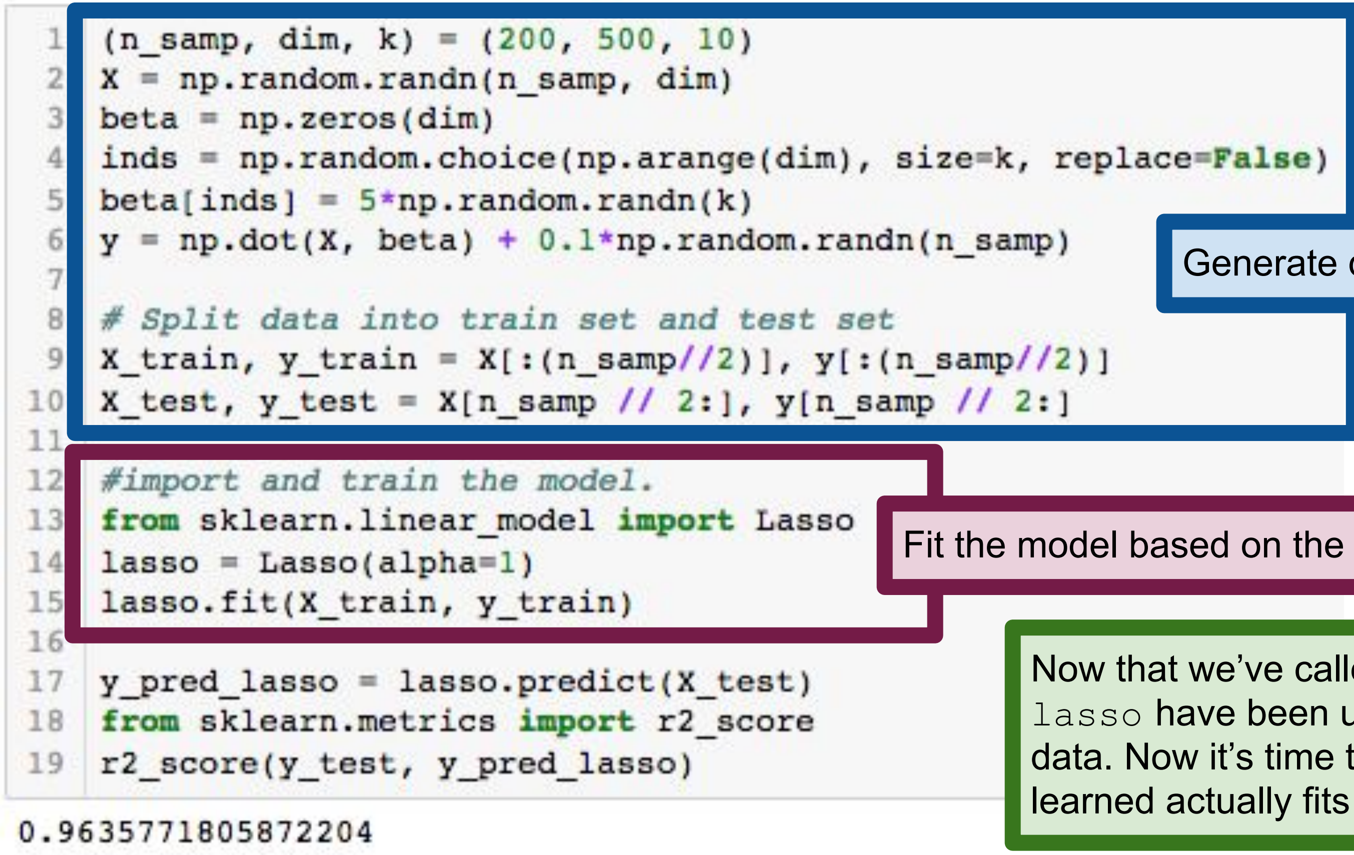
```
0.9635771805872204
```

Generate data; split into train/test.

`lasso` is a `Lasso` object, which supports both `fit` and `predict` methods (as do all “estimator” objects in `sklearn`).

Assess how well the model fits the test data.

Supervised learning example: LASSO in `sklearn`



```
(n_samp, dim, k) = (200, 500, 10)
X = np.random.randn(n_samp, dim)
beta = np.zeros(dim)
inds = np.random.choice(np.arange(dim), size=k, replace=False)
beta[inds] = 5*np.random.randn(k)
y = np.dot(X, beta) + 0.1*np.random.randn(n_samp)

# Split data into train set and test set
X_train, y_train = X[:(n_samp//2)], y[:(n_samp//2)]
X_test, y_test = X[n_samp // 2:], y[n_samp // 2:]

#import and train the model.
from sklearn.linear_model import Lasso
lasso = Lasso(alpha=1)
lasso.fit(X_train, y_train)

y_pred_lasso = lasso.predict(X_test)
from sklearn.metrics import r2_score
r2_score(y_test, y_pred_lasso)
```

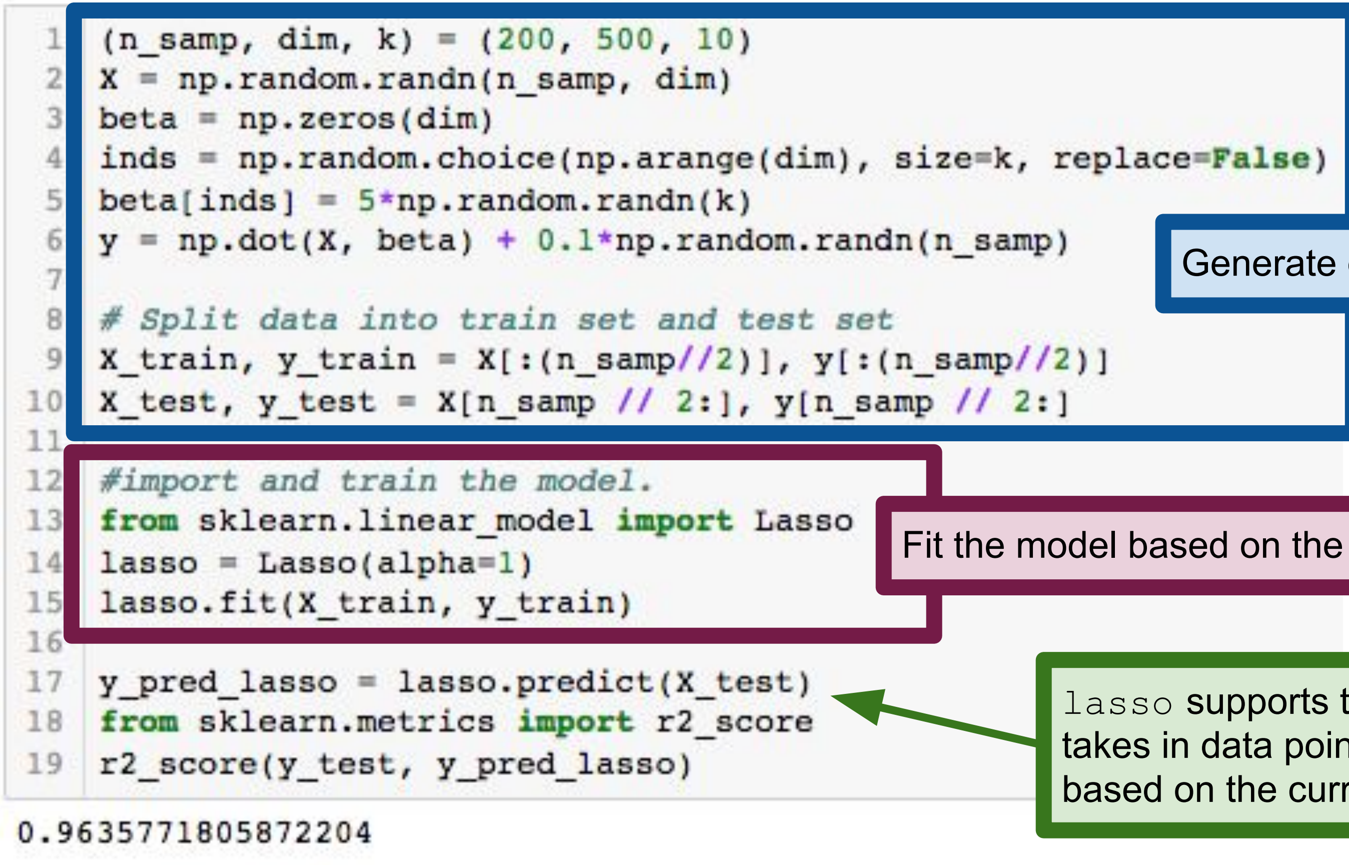
```
0.9635771805872204
```

Generate data; split into train/test.

Fit the model based on the train set.

Now that we've called `fit`, the coefficients of `lasso` have been updated to fit the training data. Now it's time to tell if the model we learned actually fits the held out data.

Supervised learning example: LASSO in `sklearn`



```
(n_samp, dim, k) = (200, 500, 10)
X = np.random.randn(n_samp, dim)
beta = np.zeros(dim)
inds = np.random.choice(np.arange(dim), size=k, replace=False)
beta[inds] = 5*np.random.randn(k)
y = np.dot(X, beta) + 0.1*np.random.randn(n_samp)

# Split data into train set and test set
X_train, y_train = X[:(n_samp//2)], y[:(n_samp//2)]
X_test, y_test = X[n_samp // 2:], y[n_samp // 2:]

#import and train the model.
from sklearn.linear_model import Lasso
lasso = Lasso(alpha=1)
lasso.fit(X_train, y_train)

y_pred_lasso = lasso.predict(X_test)
from sklearn.metrics import r2_score
r2_score(y_test, y_pred_lasso)
```

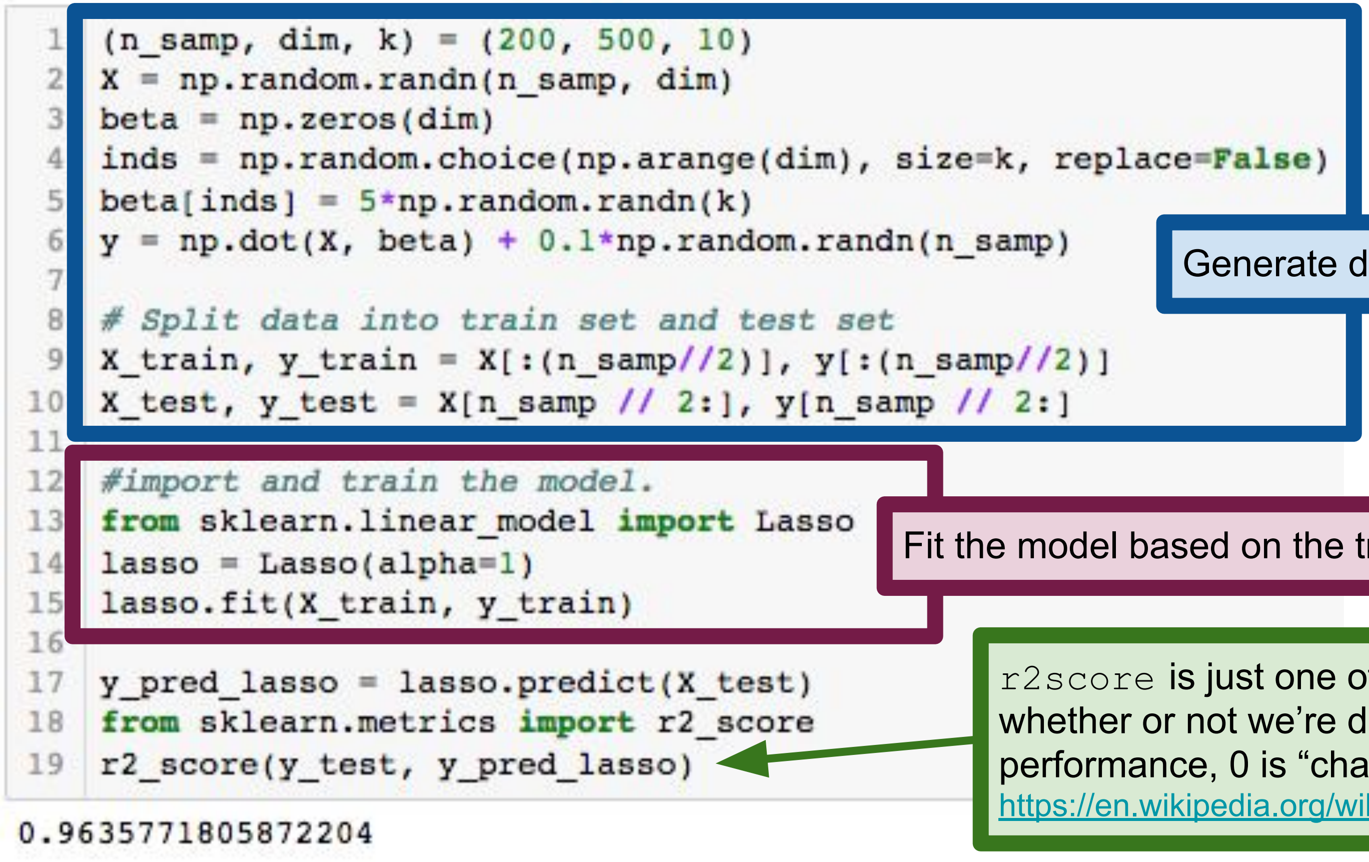
```
0.9635771805872204
```

Generate data; split into train/test.

Fit the model based on the train set.

`lasso` supports the `predict` method, which takes in data points and outputs responses based on the current estimate of `beta`.

Supervised learning example: LASSO in `sklearn`



```
(n_samp, dim, k) = (200, 500, 10)
X = np.random.randn(n_samp, dim)
beta = np.zeros(dim)
inds = np.random.choice(np.arange(dim), size=k, replace=False)
beta[inds] = 5*np.random.randn(k)
y = np.dot(X, beta) + 0.1*np.random.randn(n_samp)

# Split data into train set and test set
X_train, y_train = X[:(n_samp//2)], y[:(n_samp//2)]
X_test, y_test = X[n_samp // 2:], y[n_samp // 2:]

#import and train the model.
from sklearn.linear_model import Lasso
lasso = Lasso(alpha=1)
lasso.fit(X_train, y_train)

y_pred_lasso = lasso.predict(X_test)
from sklearn.metrics import r2_score
r2_score(y_test, y_pred_lasso)
```

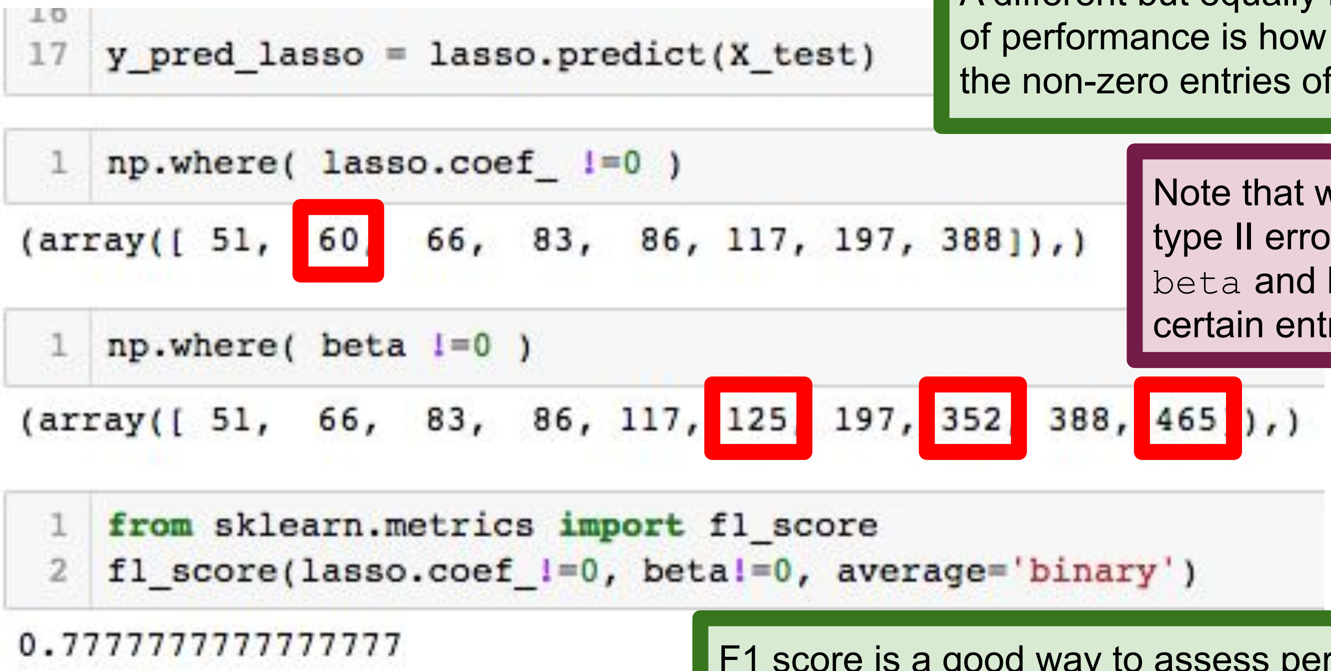
```
0.9635771805872204
```

Generate data; split into train/test.

Fit the model based on the train set.

`r2score` is just one of the many ways to assess whether or not we're doing well. 1 is perfect performance, 0 is "chance". https://en.wikipedia.org/wiki/Coefficient_of_determination

Supervised learning example: LASSO in `sklearn`



```
y_pred_lasso = lasso.predict(X_test)
```

```
np.where( lasso.coef_ !=0 )
```

```
(array([ 51,  60,  66,  83,  86, 117, 197, 388]),)
```

```
np.where( beta !=0 )
```

```
(array([ 51,  66,  83,  86, 117, 125, 197, 352, 388, 465]),)
```

```
from sklearn.metrics import f1_score
f1_score(lasso.coef_!=0, beta!=0, average='binary')
```

```
0.7777777777777777
```

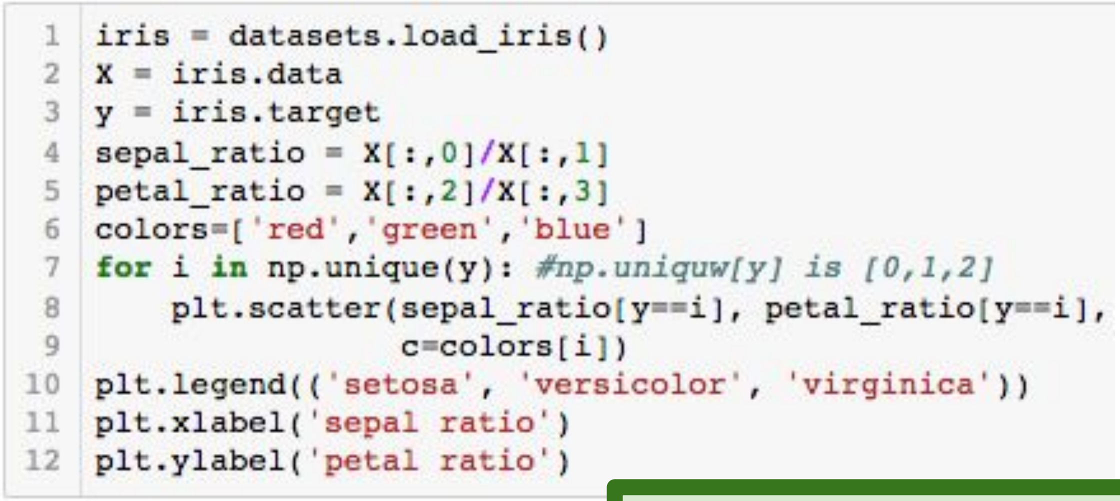
A different but equally important measure of performance is how well we recovered the non-zero entries of `beta`.

Note that we committed both type I and type II errors by missing some entries of `beta` and by incorrectly identifying certain entries as non-zero.

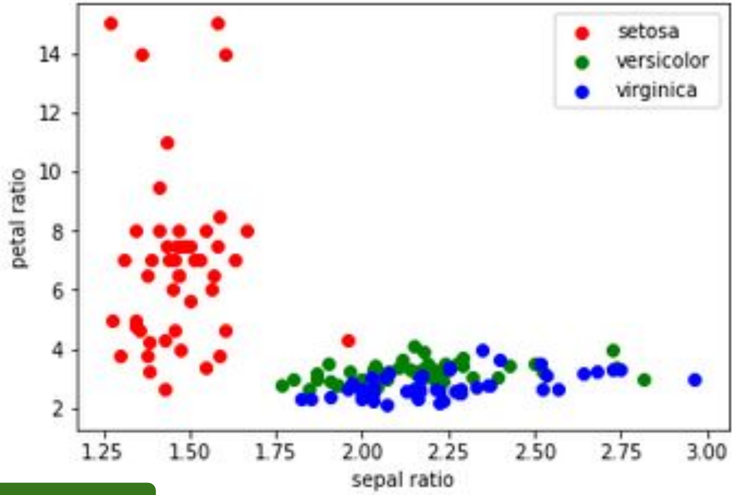
F1 score is a good way to assess performance on these kinds of problems. It is a harmonic mean between the recall and precision.
https://scikit-learn.org/stable/modules/generated/sklearn.metrics.f1_score.html

Gaussian Mixture Model

Unsupervised learning example: k-GMM in `sklearn`



```
iris = datasets.load_iris()
X = iris.data
y = iris.target
sepal_ratio = X[:,0]/X[:,1]
petal_ratio = X[:,2]/X[:,3]
colors=['red','green','blue']
for i in np.unique(y): #np.uniquw[y] is [0,1,2]
    plt.scatter(sepal_ratio[y==i], petal_ratio[y==i],
                c=colors[i])
plt.legend(('setosa', 'versicolor', 'virginica'))
plt.xlabel('sepal ratio')
plt.ylabel('petal ratio')
```



Here's the famous iris data set again. Clearly there's a cluster structure in the data. How can we discover it without using the label information?

https://scikit-learn.org/stable/auto_examples/datasets/plot_iris_dataset.html
https://en.wikipedia.org/wiki/Iris_flower_data_set

Unsupervised learning example: k-GMM in `sklearn`

Basic idea: model data as mixture of Gaussians

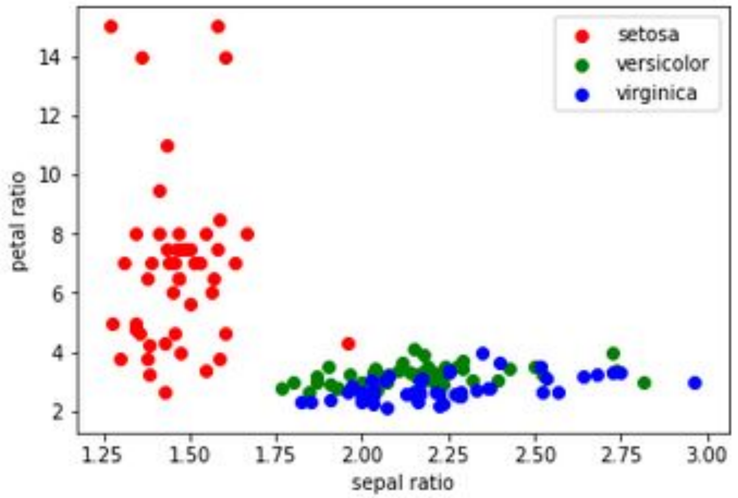
each Gaussian generates one cluster

For each cluster, estimate mean and covariance

Computationally hard...

...but can approximate via EM

https://en.wikipedia.org/wiki/Expectation-maximization_algorithm



Unsupervised learning example: k-GMM in `sklearn`

Basic idea: model data as mixture of Gaussians

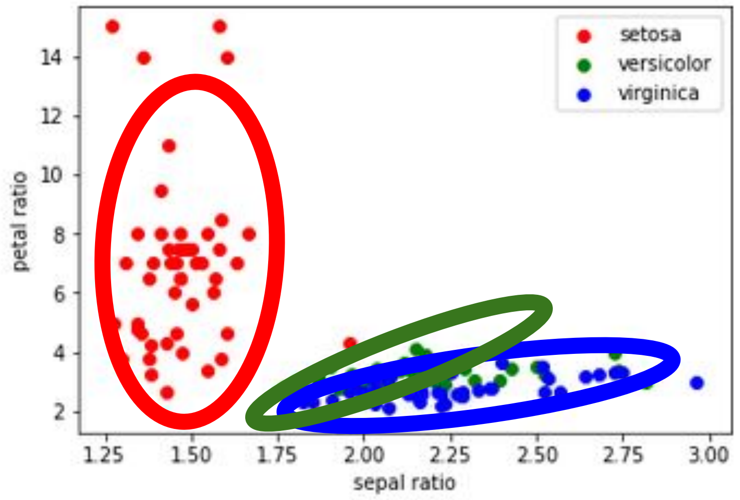
each Gaussian generates one cluster

For each cluster, estimate mean and covariance

Computationally hard...

...but can approximate via EM

https://en.wikipedia.org/wiki/Expectation-maximization_algorithm



Letting these ellipses represent the level sets of three Gaussians, we hope to see something like this picture.

Unsupervised learning example: k-GMM in `sklearn`

Basic idea: model data as mixture of Gaussians

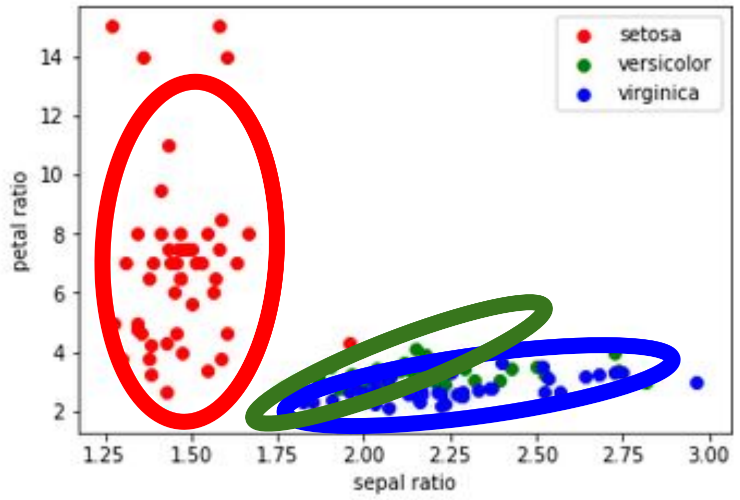
each Gaussian generates one cluster

For each cluster, estimate mean and covariance

Computationally hard...

...but can approximate via EM

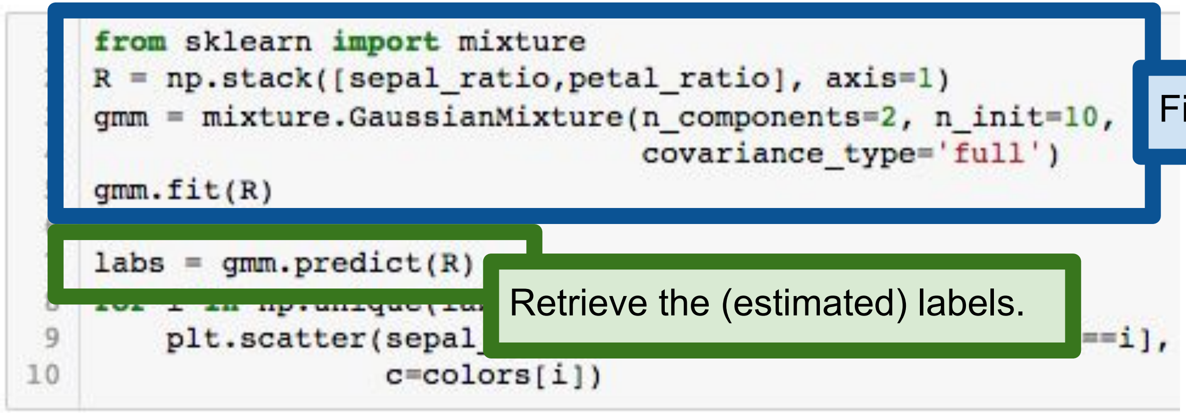
https://en.wikipedia.org/wiki/Expectation-maximization_algorithm



Of course, GMM is just one of many clustering algorithms we could choose from. For other options (though hardly an exhaustive list) and a good overview, see here: https://scikit-learn.org/stable/modules/clustering.html

Letting these ellipses represent the level sets of three Gaussians, we hope to see something like this picture.

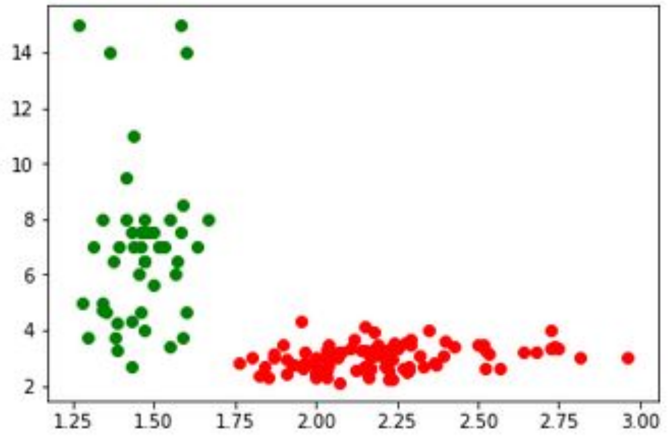
Unsupervised learning example: k-GMM in `sklearn`



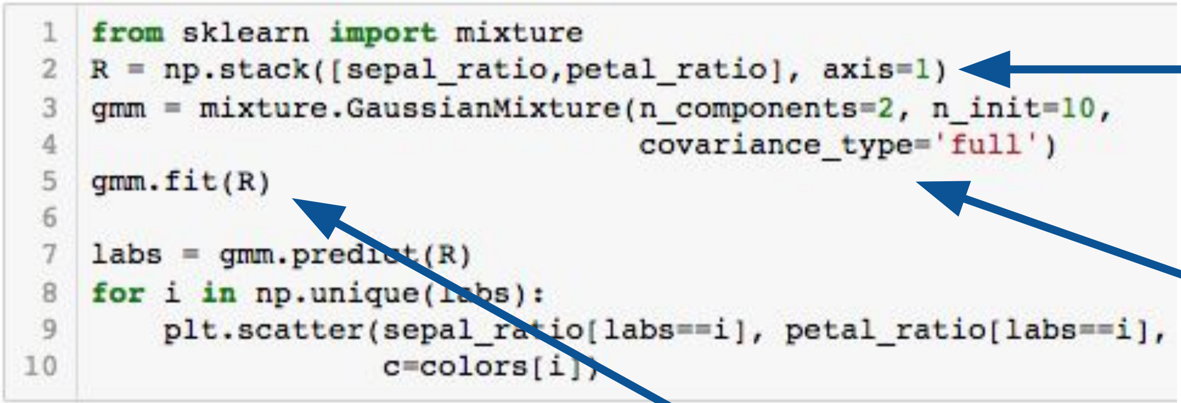
```
from sklearn import mixture
R = np.stack([sepal_ratio,petal_ratio], axis=1)
gmm = mixture.GaussianMixture(n_components=2, n_init=10,
                              covariance_type='full')
gmm.fit(R)

labs = gmm.predict(R)
for i in np.unique(la
    plt.scatter(sepal_                       ==i],
                c=colors[i])
```

Fit the model to the data.

Retrieve the (estimated) labels.



Unsupervised learning example: k-GMM in `sklearn`

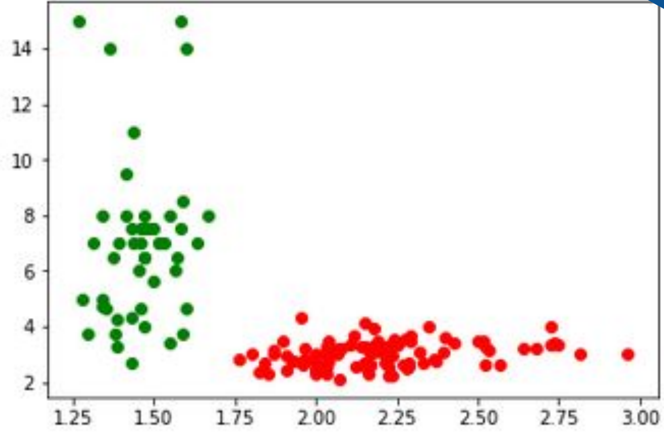


```python
from sklearn import mixture
R = np.stack([sepal_ratio,petal_ratio], axis=1)
gmm = mixture.GaussianMixture(n_components=2, n_init=10,
                              covariance_type='full')
gmm.fit(R)

labs = gmm.predict(R)
for i in np.unique(labs):
    plt.scatter(sepal_ratio[labs==i], petal_ratio[labs==i],
                c=colors[i])
```

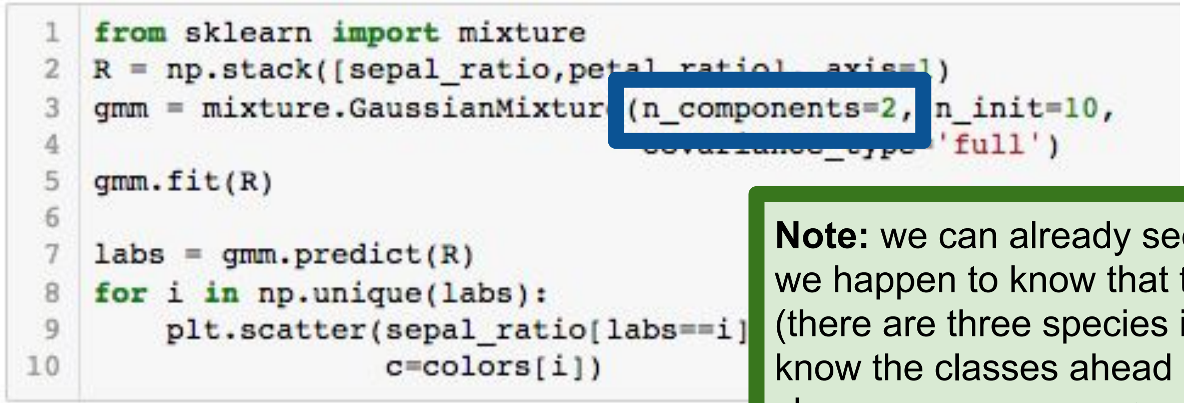
Gathering the sepal and petal ratios into a single array.

The `GaussianMixture` object has a number of attributes that specify how to go about finding a good fit. More about this in a moment.

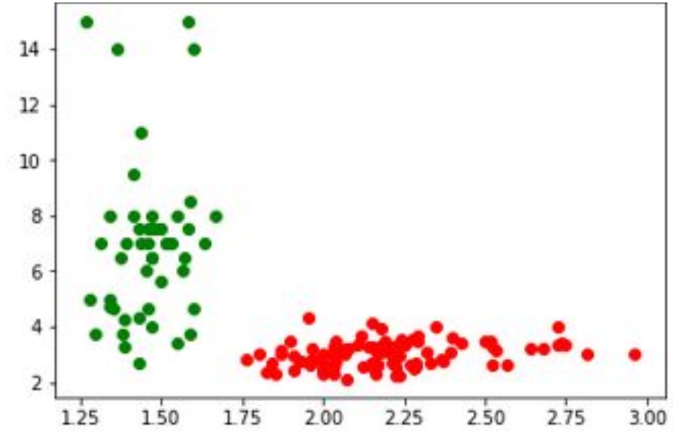
Every `sklearn` model supports the `fit` method. In this case, fitting consists of estimating the means and covariances of n=2 components.



Unsupervised learning example: k-GMM in `sklearn`



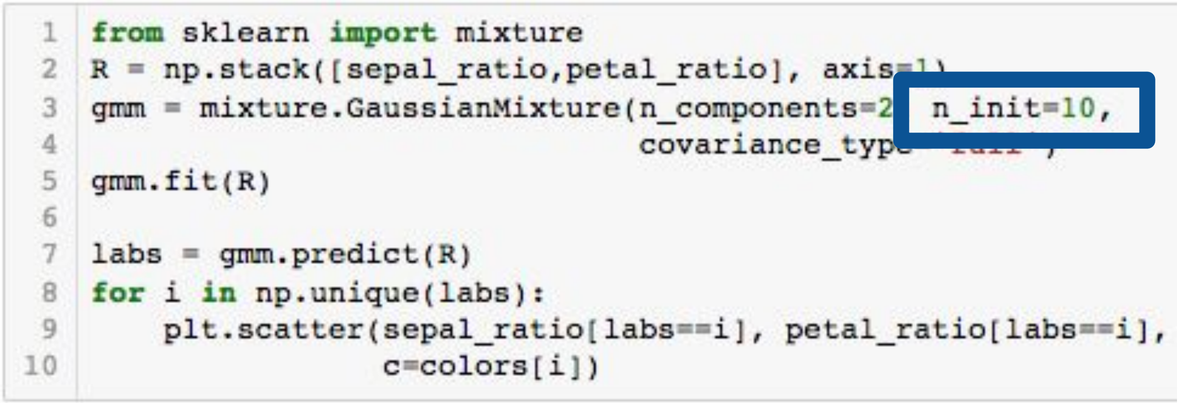
```
from sklearn import mixture
R = np.stack([sepal_ratio,petal_ratio], axis=1)
gmm = mixture.GaussianMixture(n_components=2, n_init=10,
                              covariance_type='full')
gmm.fit(R)

labs = gmm.predict(R)
for i in np.unique(labs):
    plt.scatter(sepal_ratio[labs==i]
                c=colors[i])
```



Fit a model with 2 components.

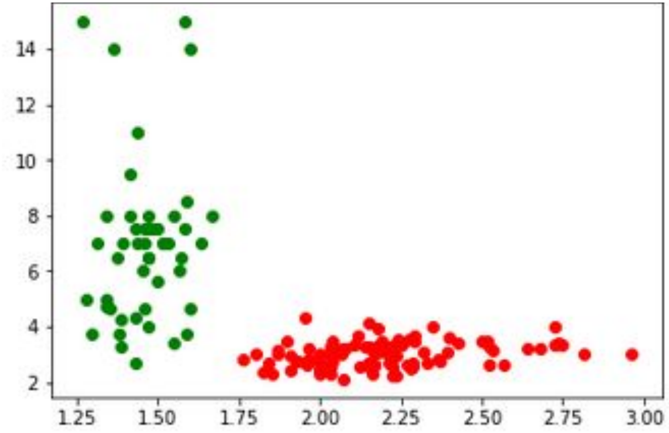
Note: we can already see a hard problem here. In this case, we happen to know that there are really three classes here (there are three species in the data), but typically, we don't know the classes ahead of time, so we don't know how to choose `n_components`. This is called **model selection**. More on this in a few slides.

Unsupervised learning example: k-GMM in `sklearn`

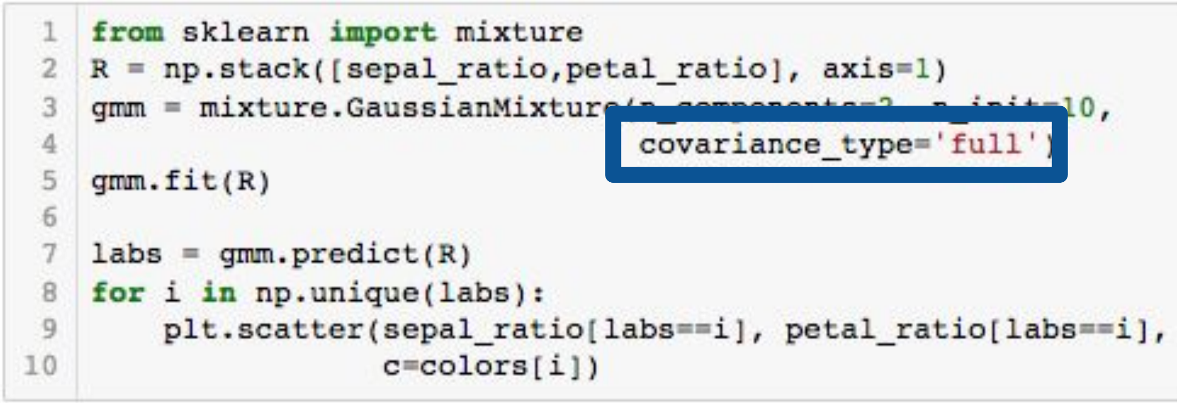


```
from sklearn import mixture
R = np.stack([sepal_ratio,petal_ratio], axis=1)
gmm = mixture.GaussianMixture(n_components=2, n_init=10,
                              covariance_type='full')
gmm.fit(R)

labs = gmm.predict(R)
for i in np.unique(labs):
    plt.scatter(sepal_ratio[labs==i], petal_ratio[labs==i],
                c=colors[i])
```

The EM algorithm is sensitive to its starting conditions, so we tell `sklearn` to run the EM algorithm multiple times (10, in this case), with different (random) starting conditions, and it keeps the one with the highest likelihood.

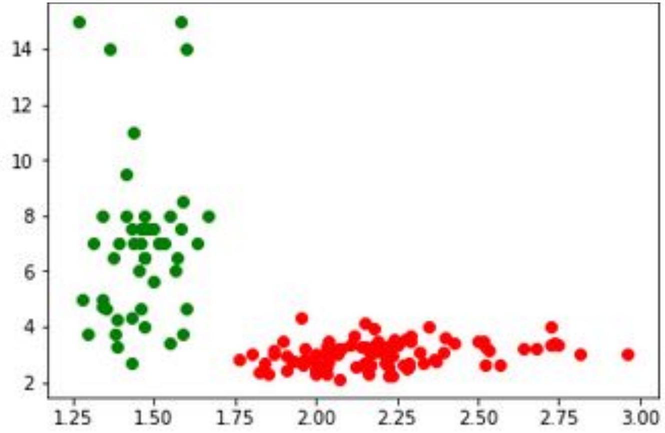


Unsupervised learning example: k-GMM in `sklearn`

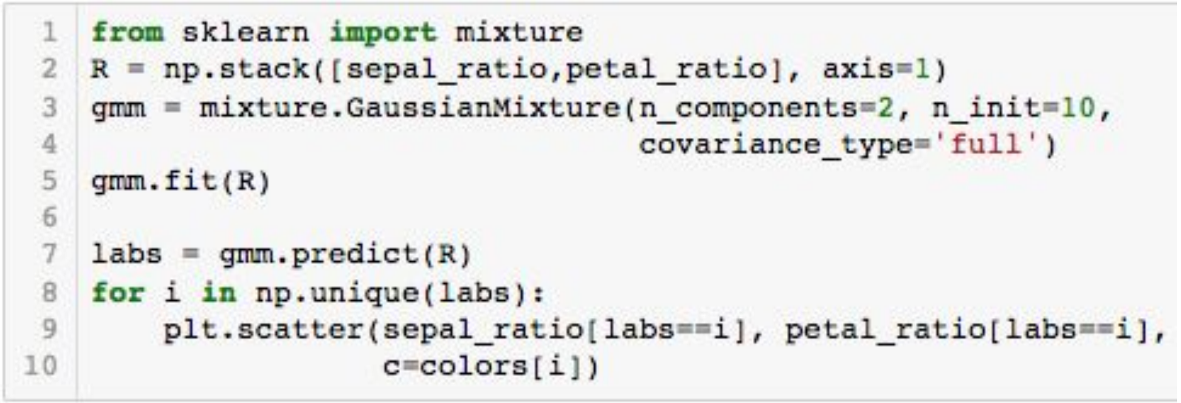


```
from sklearn import mixture
R = np.stack([sepal_ratio,petal_ratio], axis=1)
gmm = mixture.GaussianMixture(n_components=2, n_init=10,
                              covariance_type='full')
gmm.fit(R)

labs = gmm.predict(R)
for i in np.unique(labs):
    plt.scatter(sepal_ratio[labs==i], petal_ratio[labs==i],
                c=colors[i])
```

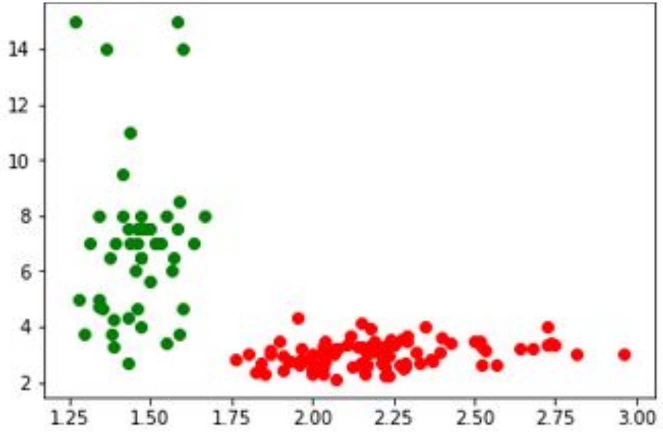
This tells `sklearn` to estimate a covariance matrix separately for each cluster. Other options include estimating one covariance shared across all clusters (`'tied'`) and estimating spherical covariances for each cluster (`'spherical'`).



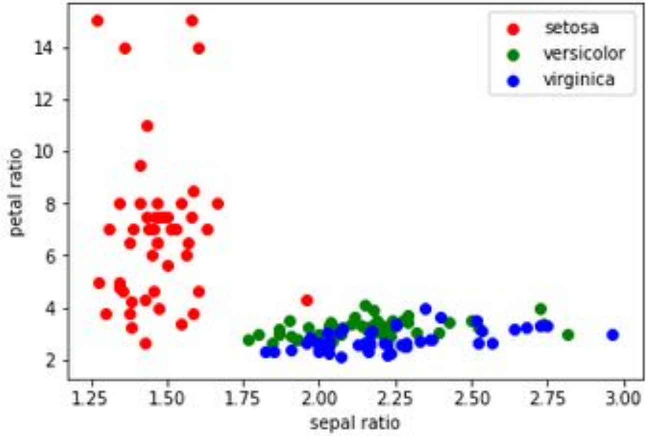
Unsupervised learning example: k-GMM in `sklearn`



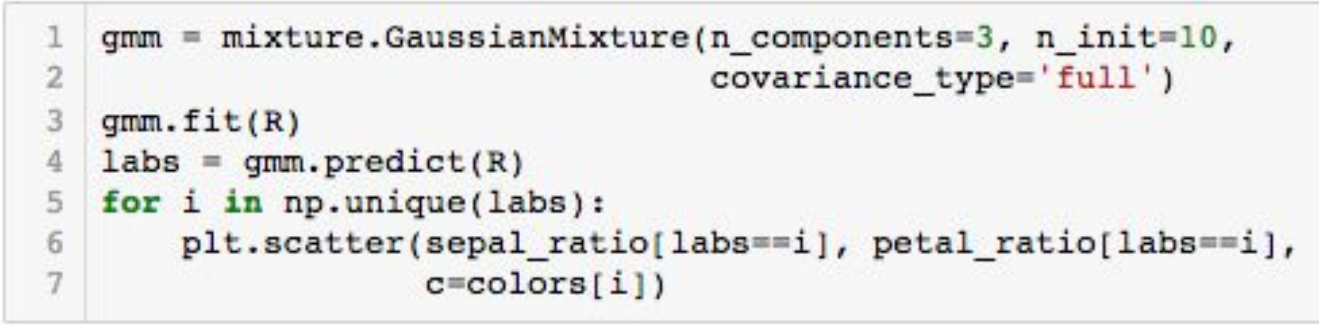
```python
from sklearn import mixture
R = np.stack([sepal_ratio,petal_ratio], axis=1)
gmm = mixture.GaussianMixture(n_components=2, n_init=10,
                              covariance_type='full')
gmm.fit(R)

labs = gmm.predict(R)
for i in np.unique(labs):
    plt.scatter(sepal_ratio[labs==i], petal_ratio[labs==i],
                c=colors[i])
```



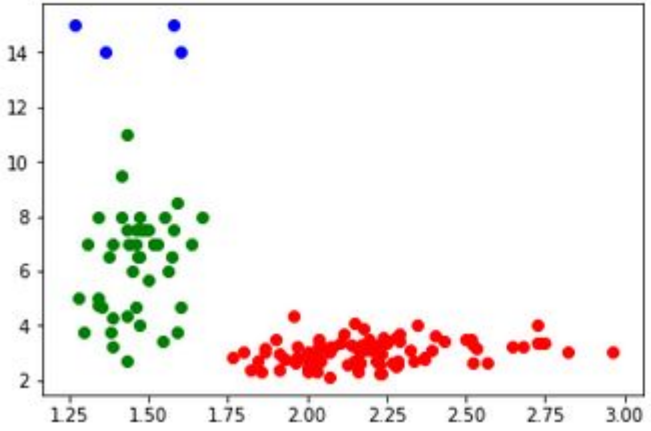
Of course, because we chose the wrong number of components, we fail to recover the true cluster structure of the data.



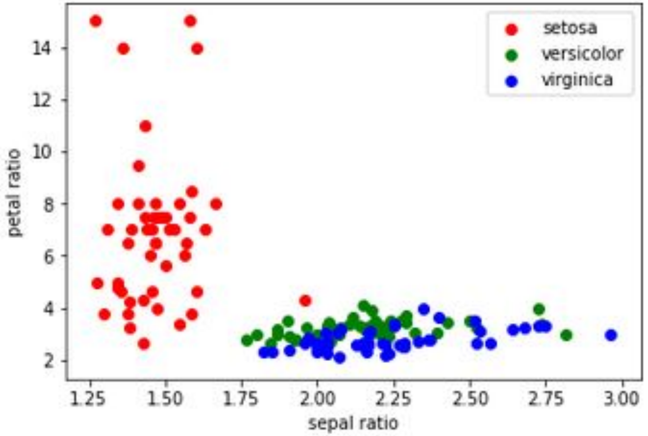
Unsupervised learning example: k-GMM in `sklearn`



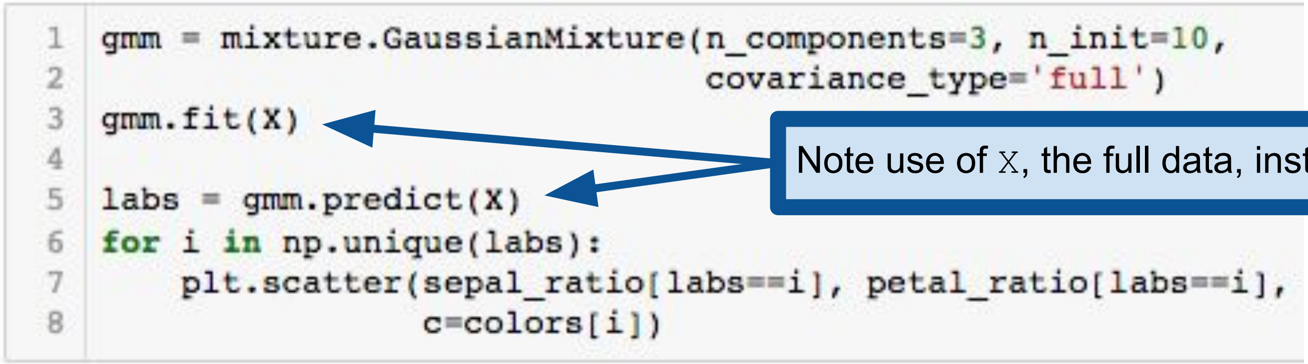
```python
gmm = mixture.GaussianMixture(n_components=3, n_init=10,
                              covariance_type='full')
gmm.fit(R)
labs = gmm.predict(R)
for i in np.unique(labs):
    plt.scatter(sepal_ratio[labs==i], petal_ratio[labs==i],
                c=colors[i])
```



But even if we choose the correct number of components, the “ratio” representation of the data collapses the versicolor and virginica species, and we get a weird solution.

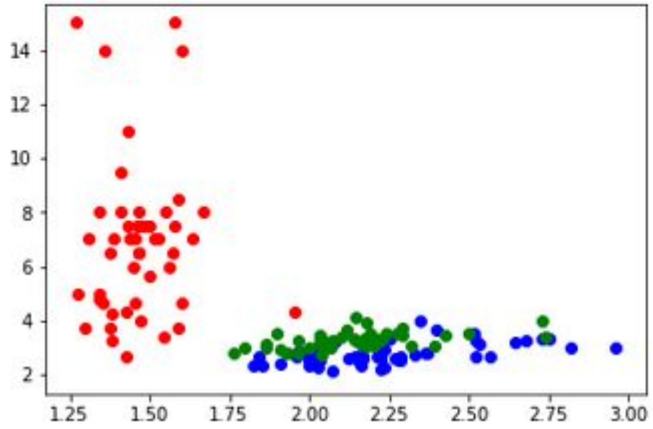


Unsupervised learning example: k-GMM in `sklearn`

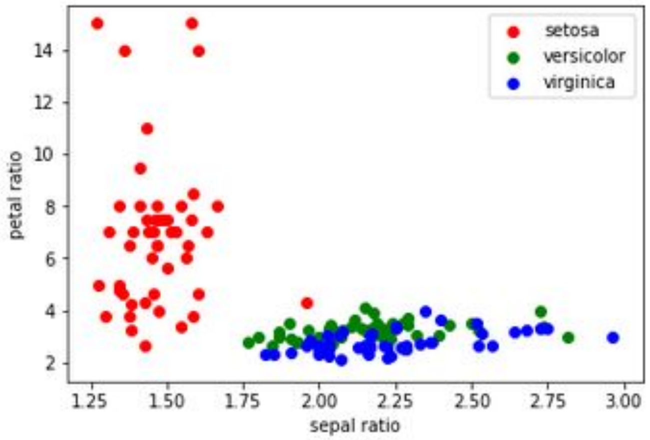


```python
gmm = mixture.GaussianMixture(n_components=3, n_init=10,
                              covariance_type='full')

gmm.fit(X)

labs = gmm.predict(X)
for i in np.unique(labs):
    plt.scatter(sepal_ratio[labs==i], petal_ratio[labs==i],
                c=colors[i])
```

Note use of `X`, the full data, instead of the “ratios” `R`.



Clustering with the correct number of components in the original 4-dimensional space recovers the truth.



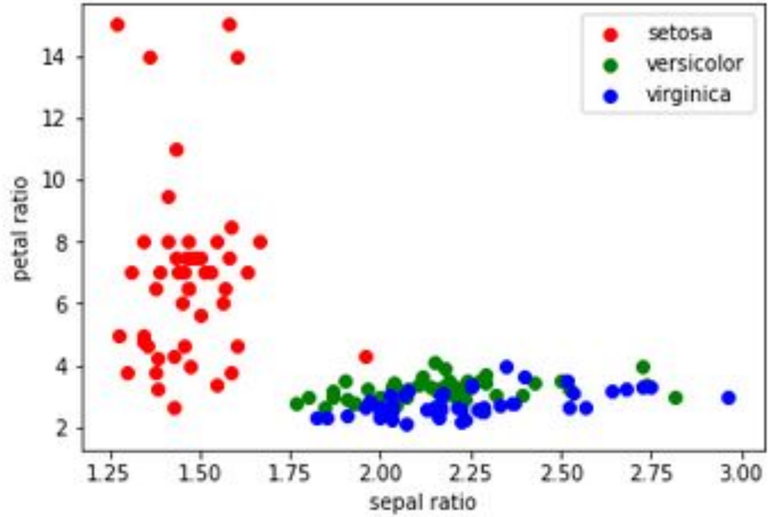
Model selection in `sklearn`

How should we choose the number of clusters in practice?

Again, typically we don't know, e.g., that there are three species in the data

One popular solution is to use an **information criterion**

- measures how well a model reflects data
- penalizes model complexity



Examples:

https://en.wikipedia.org/wiki/Bayesian_information_criterion

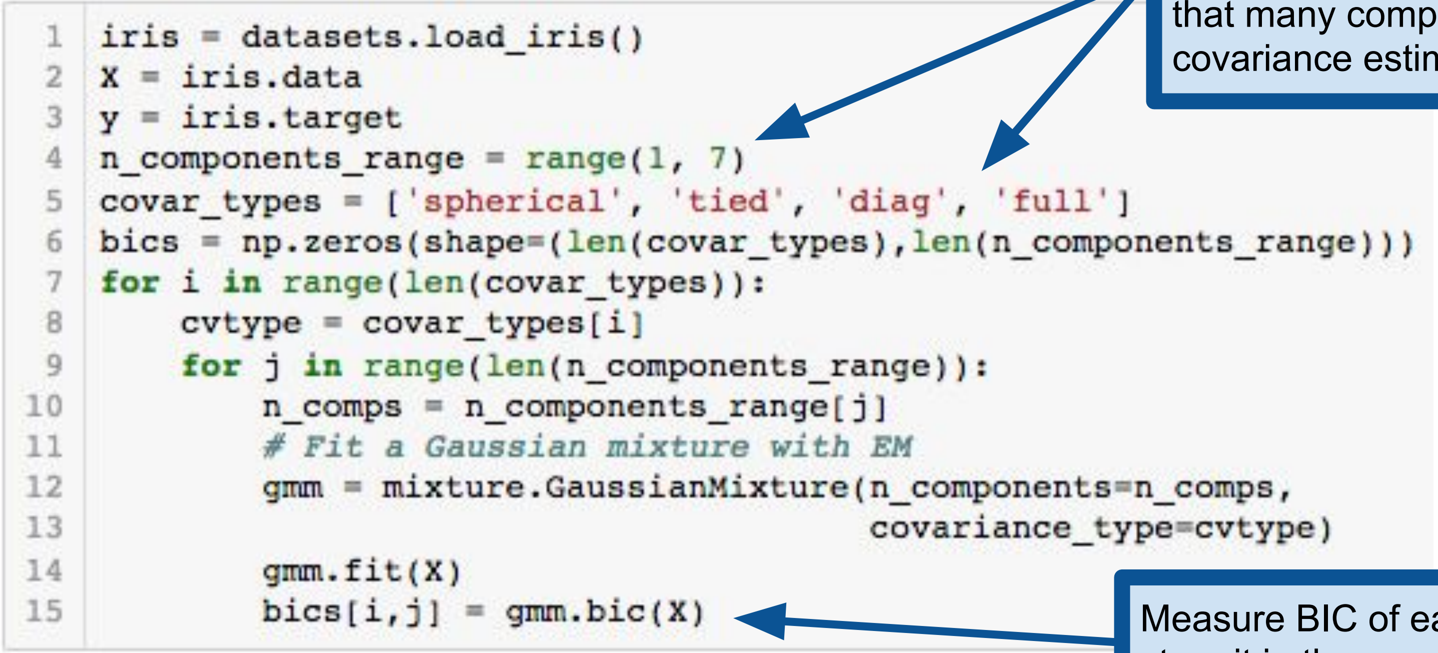
https://en.wikipedia.org/wiki/Akaike_information_criterion

https://en.wikipedia.org/wiki/Mallows%27s_Cp

See also

https://en.wikipedia.org/wiki/Minimum_description_length

Model selection in `sklearn`



```
iris = datasets.load_iris()
X = iris.data
y = iris.target
n_components_range = range(1, 7)
covar_types = ['spherical', 'tied', 'diag', 'full']
bics = np.zeros(shape=(len(covar_types),len(n_components_range)))
for i in range(len(covar_types)):
    cvtype = covar_types[i]
    for j in range(len(n_components_range)):
        n_comps = n_components_range[j]
        # Fit a Gaussian mixture with EM
        gmm = mixture.GaussianMixture(n_components=n_comps,
                                      covariance_type=cvtype)
        gmm.fit(X)
        bics[i,j] = gmm.bic(X)
```

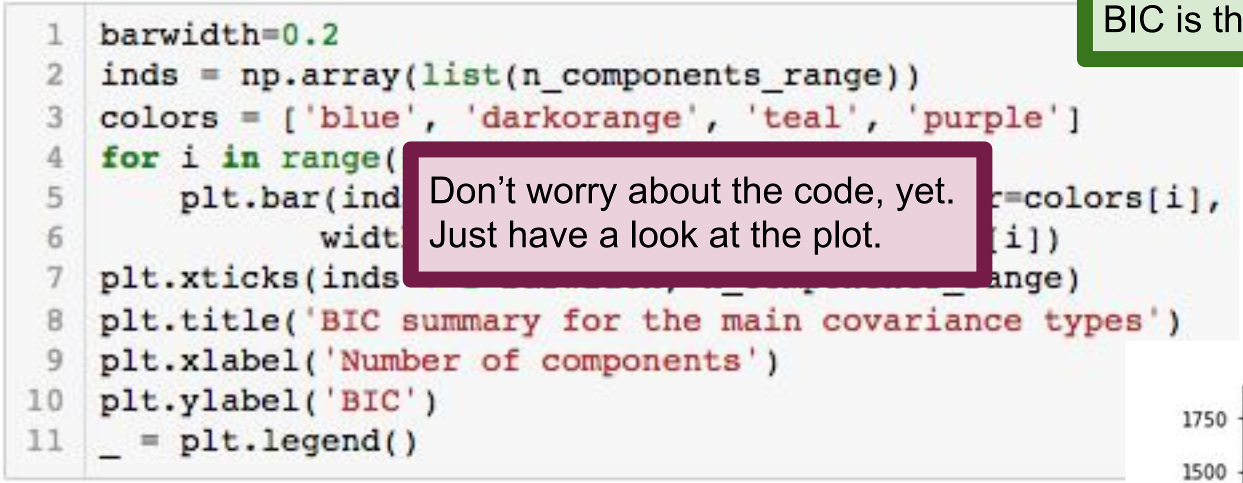
For different numbers of components and different covariance estimation methods, we're going to fit a GMM with that many components and using that covariance estimation method.

Measure BIC of each such choice; store it in the array `bics`.

This is a simplified version of the demo here: https://scikit-learn.org/stable/auto_examples/mixture/plot_gmm_selection.html

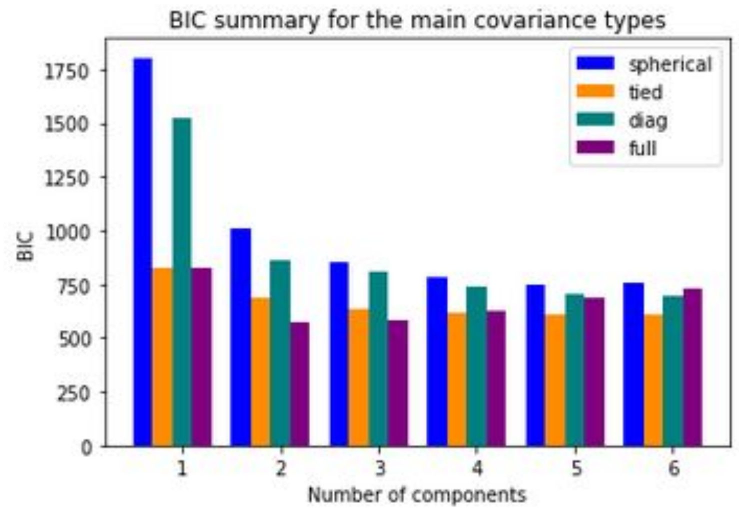
Model selection in `sklearn`

Now, let's have a look at the BIC scores. Within each covariance estimation method, the number of components with the lowest BIC is the one we should choose.



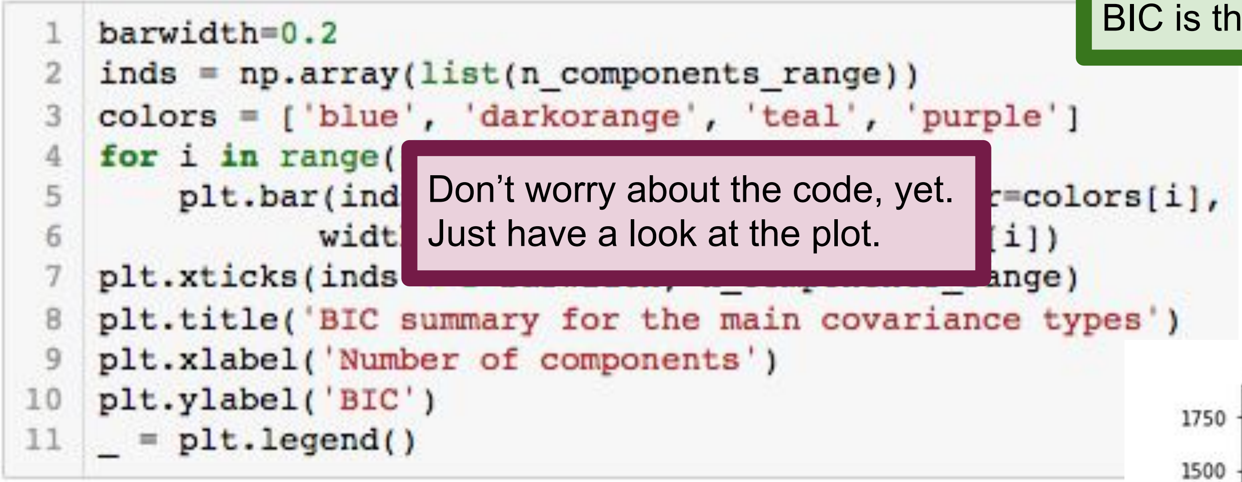
```
barwidth=0.2
inds = np.array(list(n_components_range))
colors = ['blue', 'darkorange', 'teal', 'purple']
for i in range(
    plt.bar(ind                                      r=colors[i],
            widt                                     [i])
plt.xticks(inds                                      ange)
plt.title('BIC summary for the main covariance types')
plt.xlabel('Number of components')
plt.ylabel('BIC')
_ = plt.legend()
```

Don't worry about the code, yet.
Just have a look at the plot.



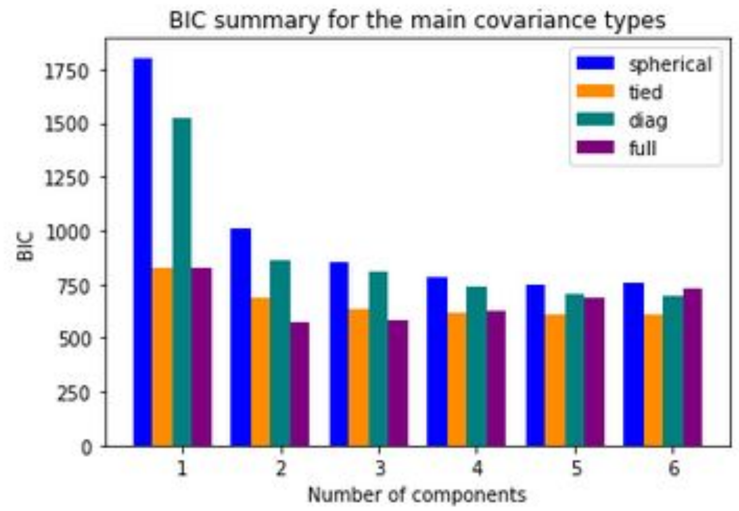
Model selection in `sklearn`

Now, let’s have a look at the BIC scores. Within each covariance estimation method, the number of components with the lowest BIC is the one we should choose.



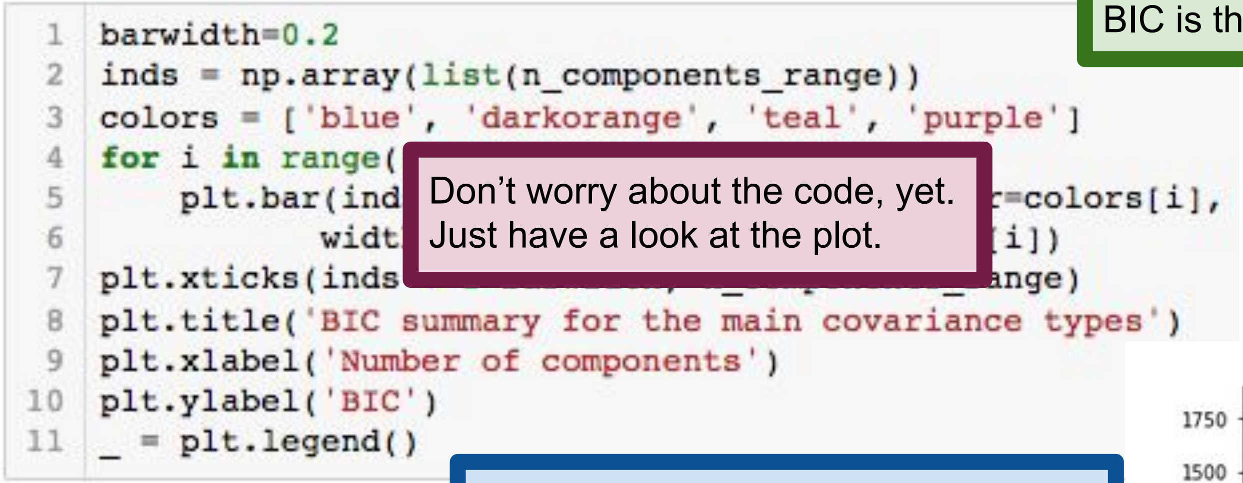
```
barwidth=0.2
inds = np.array(list(n_components_range))
colors = ['blue', 'darkorange', 'teal', 'purple']
for i in range(
    plt.bar(ind                                      r=colors[i],
            widt                                     [i])
plt.xticks(inds                                  ange)
plt.title('BIC summary for the main covariance types')
plt.xlabel('Number of components')
plt.ylabel('BIC')
_ = plt.legend()
```

Don’t worry about the code, yet.
Just have a look at the plot.



Spherical and diagonal covariance both seem to think that more components is always better (at least up to 6, anyway). This is unsurprising given the data: it’s simple to check that the dimensions of the iris data are correlated.

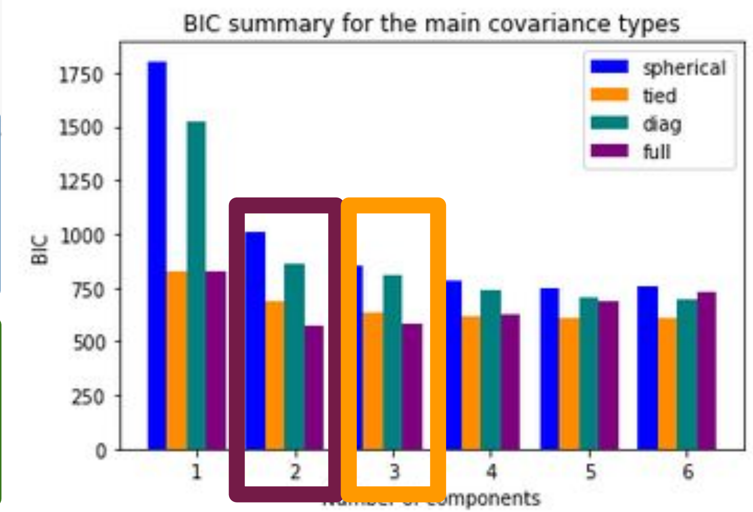
Model selection in `sklearn`



```
barwidth=0.2
inds = np.array(list(n_components_range))
colors = ['blue', 'darkorange', 'teal', 'purple']
for i in range(
    plt.bar(ind                                  r=colors[i],
            widt                                 [i])
plt.xticks(inds                                  ange)
plt.title('BIC summary for the main covariance types')
plt.xlabel('Number of components')
plt.ylabel('BIC')
_ = plt.legend()
```

Now, let's have a look at the BIC scores. Within each covariance estimation method, the number of components with the lowest BIC is the one we should choose.

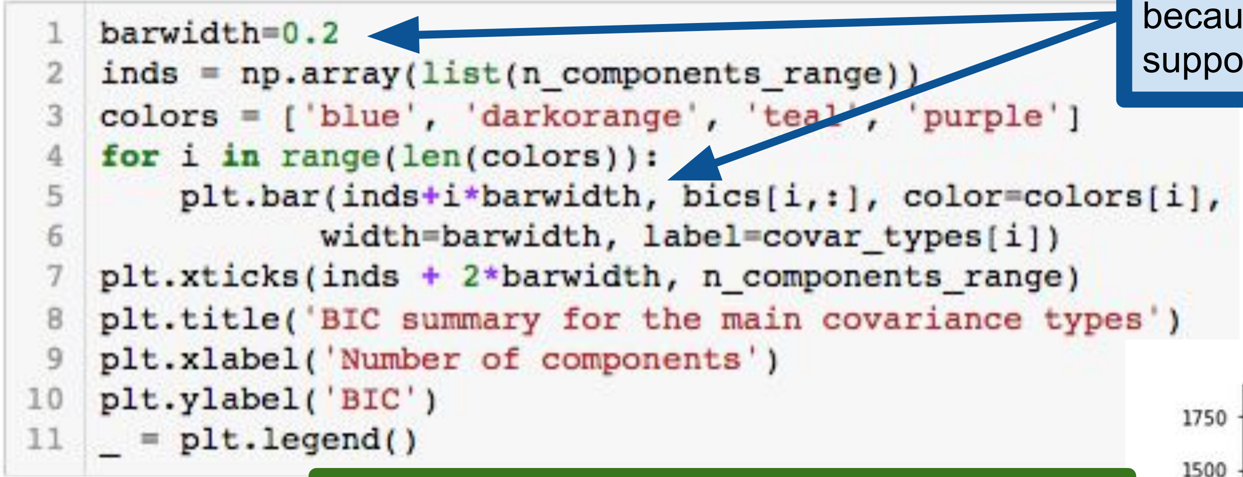
Don't worry about the code, yet. Just have a look at the plot.



Full covariance has lowest BIC at 2. Tied covariance selects (the correct) number of components to be 3.

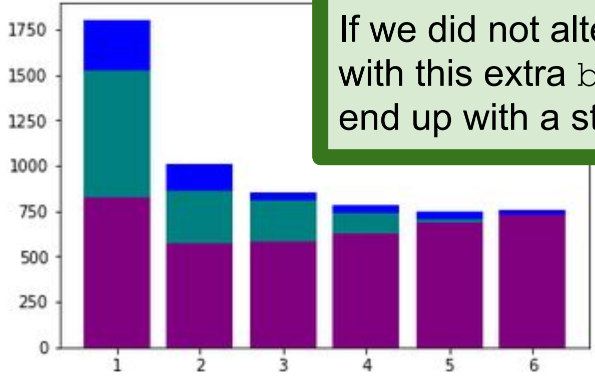
The lesson here is not that one of these methods will always be best, but that even a principled technique like BIC may sometimes give us the wrong answer.

Model selection in `sklearn`

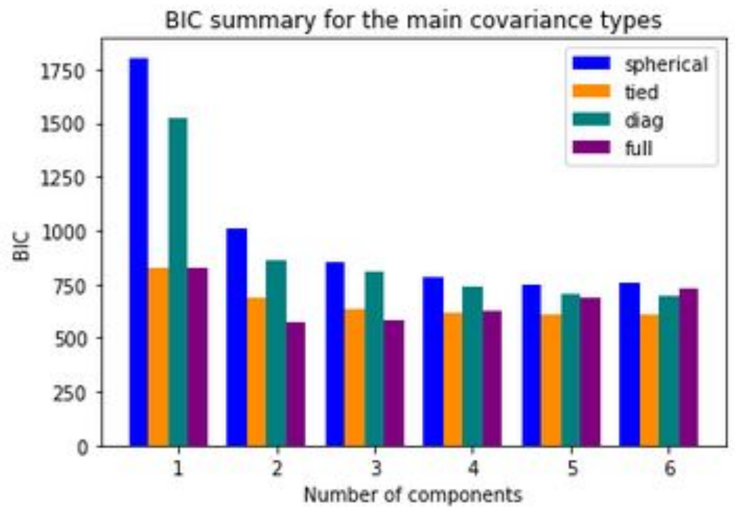


```python
barwidth=0.2
inds = np.array(list(n_components_range))
colors = ['blue', 'darkorange', 'teal', 'purple']
for i in range(len(colors)):
    plt.bar(inds+i*barwidth, bics[i,:], color=colors[i],
                width=barwidth, label=covar_types[i])
plt.xticks(inds + 2*barwidth, n_components_range)
plt.title('BIC summary for the main covariance types')
plt.xlabel('Number of components')
plt.ylabel('BIC')
_ = plt.legend()
```

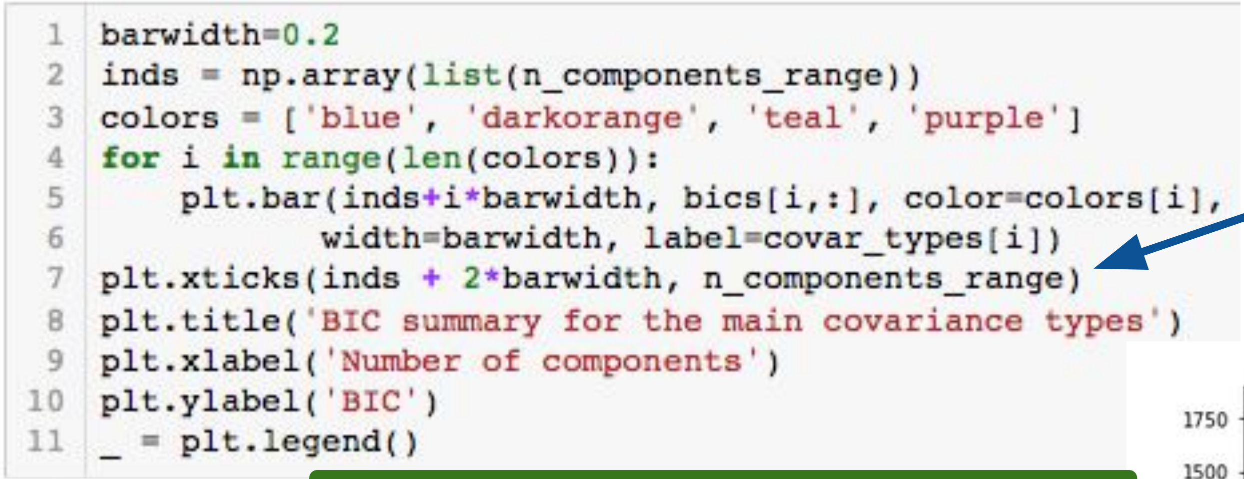
We need to do a bit of annoying work because matplotlib doesn't have good support for "grouped" bar plots like this.



If we did not alter the placement of the bars with this extra `barwidth` business, we would end up with a stacked bar graph like this one.



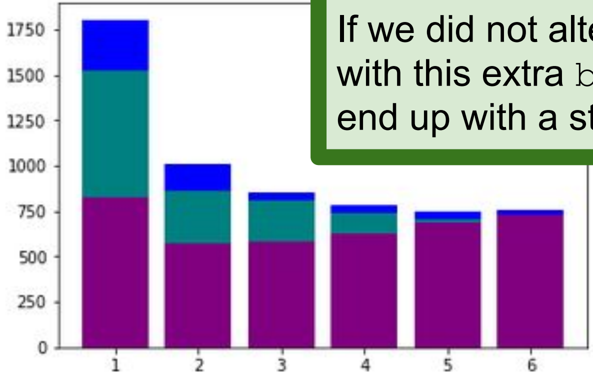
Model selection in `sklearn`

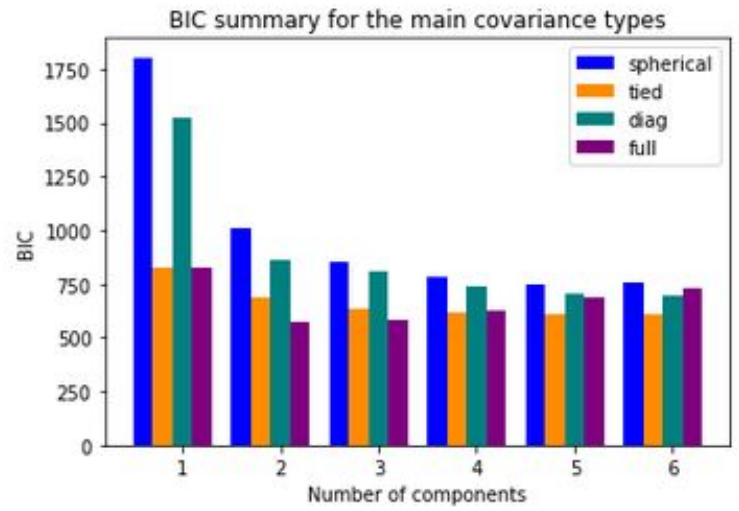


```python
barwidth=0.2
inds = np.array(list(n_components_range))
colors = ['blue', 'darkorange', 'teal', 'purple']
for i in range(len(colors)):
    plt.bar(inds+i*barwidth, bics[i,:], color=colors[i],
               width=barwidth, label=covar_types[i])
plt.xticks(inds + 2*barwidth, n_components_range)
plt.title('BIC summary for the main covariance types')
plt.xlabel('Number of components')
plt.ylabel('BIC')
_ = plt.legend()
```

We also have to alter the location of the ticks on the x-axis, which would otherwise be aligned to the first bar in each group.

If we did not alter the placement of the bars with this extra `barwidth` business, we would end up with a stacked bar graph like this one.





Cross-validation in `sklearn`

Similar to model selection, `sklearn` includes tools for cross-validation (CV)

CV is how we choose parameters like `alpha` in the LASSO

Basic idea:

Try many different choices of parameter

Keep the one that gives the best performance on the train data

There are many ways to do this, but we'll focus on K-fold CV

See https://en.wikipedia.org/wiki/Cross-validation_(statistics) for more

Cross-validation in `sklearn`: K-fold CV

We split the training data into K “folds” (K=5 in the example at right). For each fold, we train on the other K-1 folds and evaluate the trained model on the “held-out” fold.

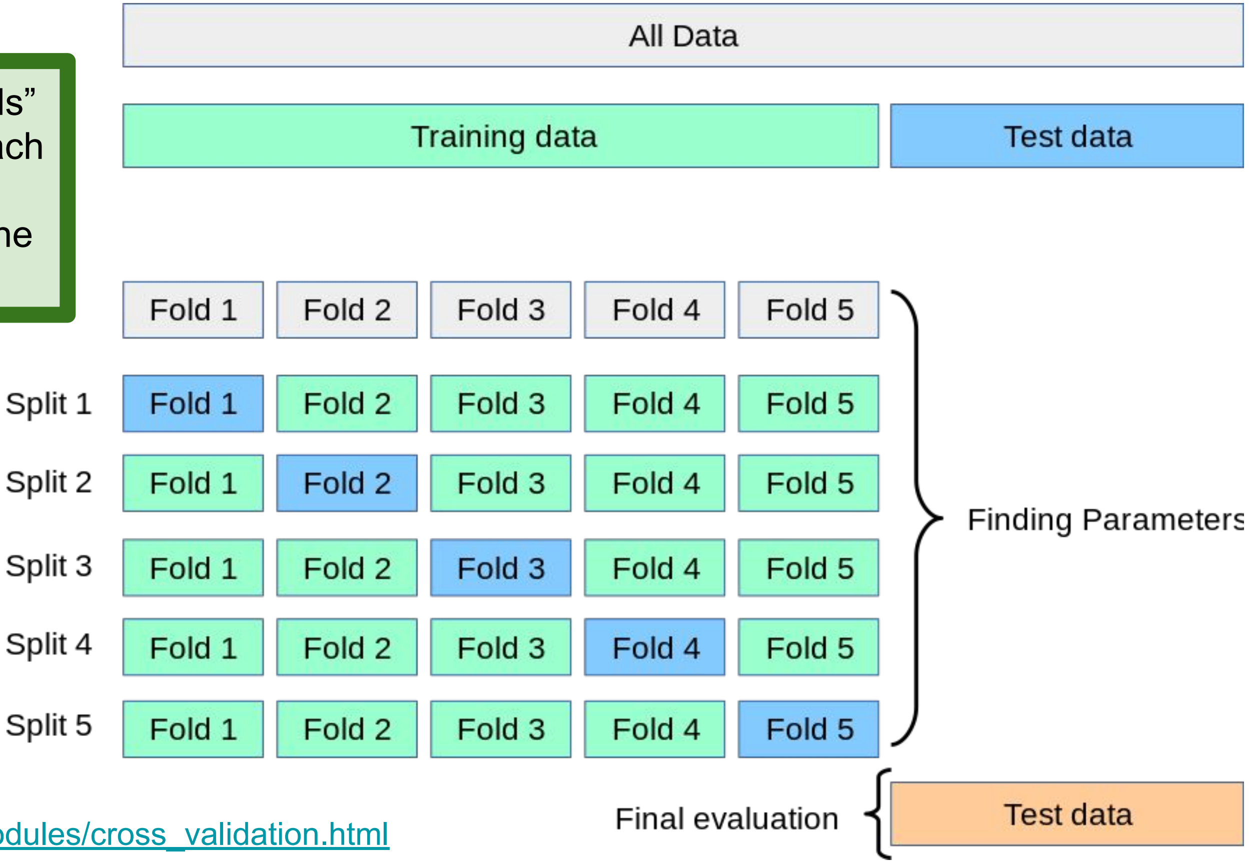


Image credit: https://scikit-learn.org/stable/modules/cross_validation.html

Cross-validation in `sklearn`: K-fold CV

We split the training data into K “folds” (K=5 in the example at right). For each fold, we train on the other K-1 folds and evaluate the trained model on the “held-out” fold.

On each fold, we evaluate all of the models that are under consideration. We then average each model over the folds and keep the model with the best average score.

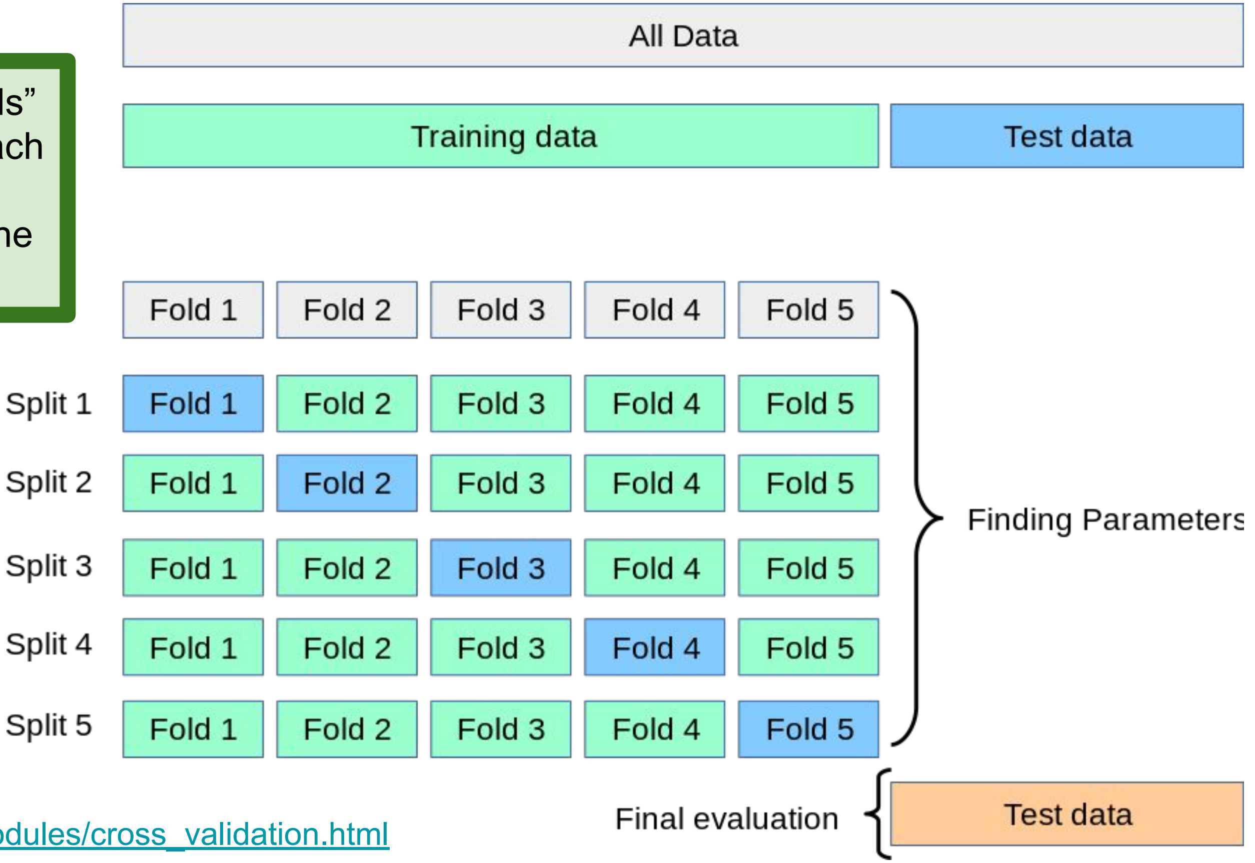
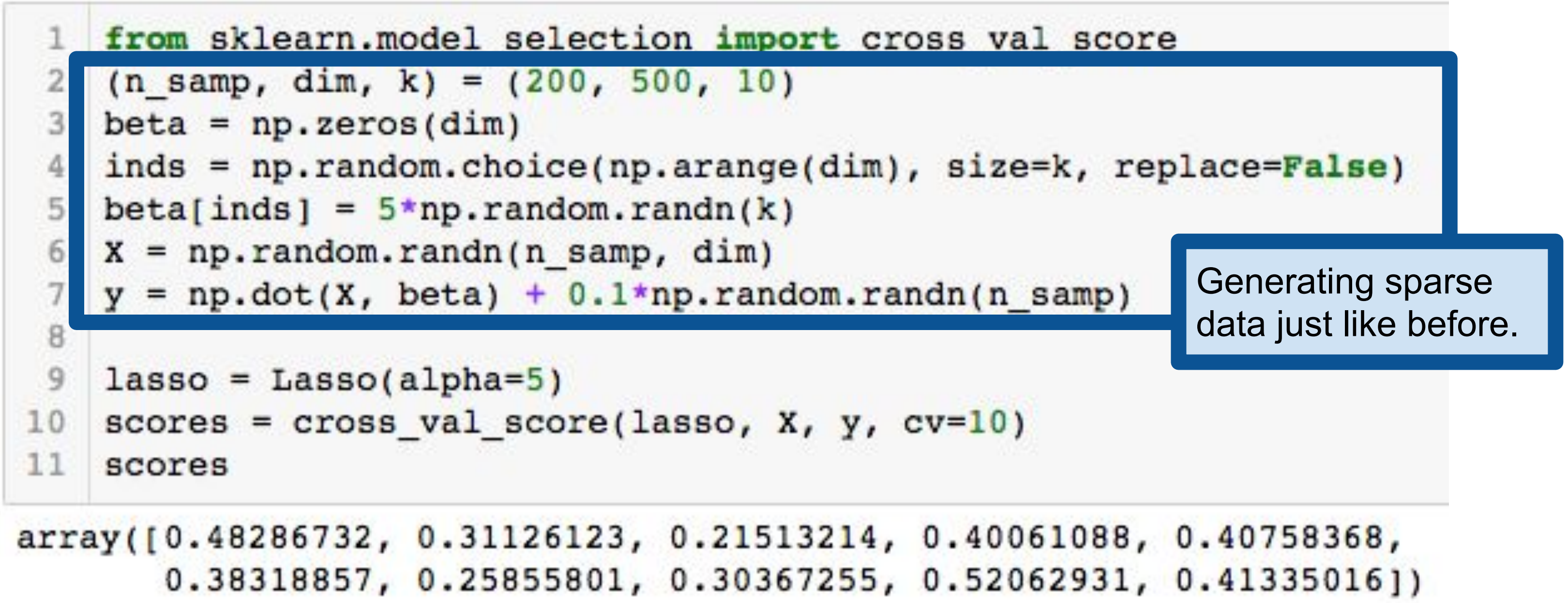


Image credit: https://scikit-learn.org/stable/modules/cross_validation.html

Cross-validation in `sklearn`

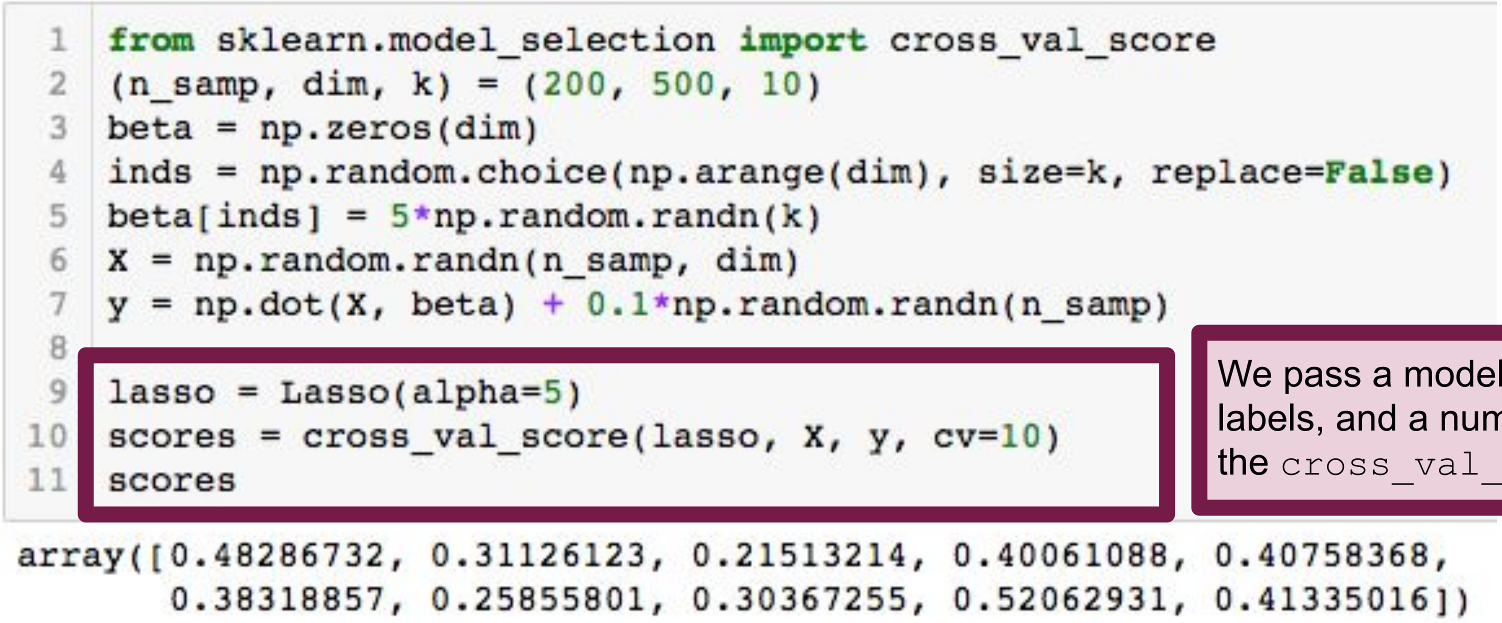


```
from sklearn.model_selection import cross_val_score
(n_samp, dim, k) = (200, 500, 10)
beta = np.zeros(dim)
inds = np.random.choice(np.arange(dim), size=k, replace=False)
beta[inds] = 5*np.random.randn(k)
X = np.random.randn(n_samp, dim)
y = np.dot(X, beta) + 0.1*np.random.randn(n_samp)

lasso = Lasso(alpha=5)
scores = cross_val_score(lasso, X, y, cv=10)
scores
```

```
array([0.48286732, 0.31126123, 0.21513214, 0.40061088, 0.40758368,
       0.38318857, 0.25855801, 0.30367255, 0.52062931, 0.41335016])
```

Generating sparse data just like before.

Cross-validation in `sklearn`



```python
from sklearn.model_selection import cross_val_score
(n_samp, dim, k) = (200, 500, 10)
beta = np.zeros(dim)
inds = np.random.choice(np.arange(dim), size=k, replace=False)
beta[inds] = 5*np.random.randn(k)
X = np.random.randn(n_samp, dim)
y = np.dot(X, beta) + 0.1*np.random.randn(n_samp)

lasso = Lasso(alpha=5)
scores = cross_val_score(lasso, X, y, cv=10)
scores
```

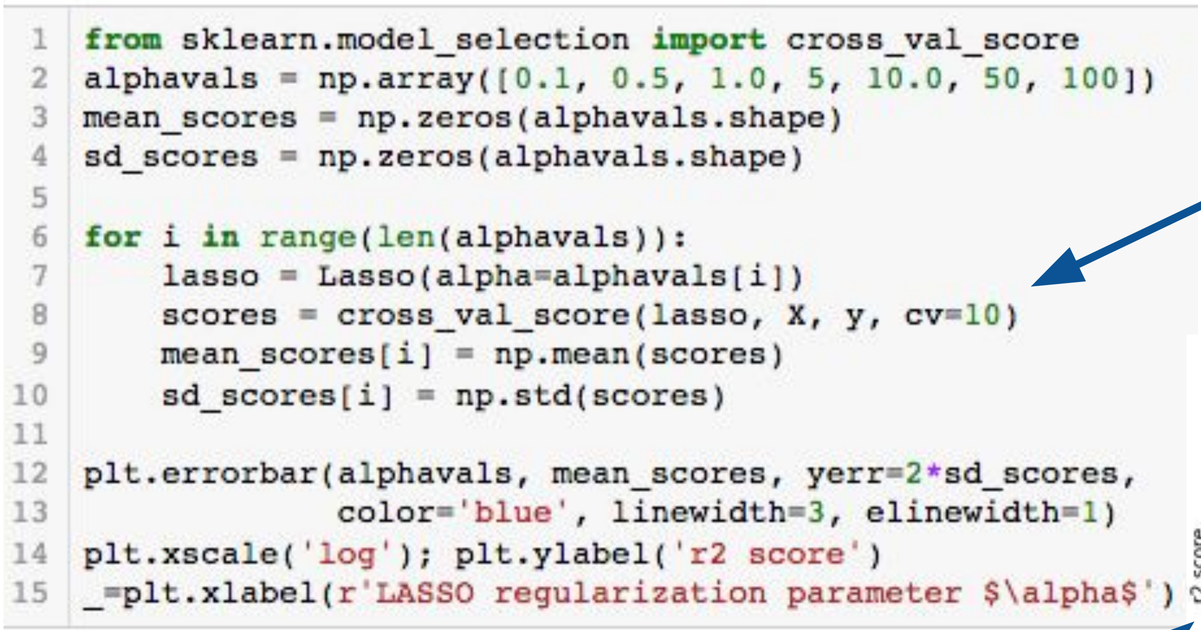
```
array([0.48286732, 0.31126123, 0.21513214, 0.40061088, 0.40758368,
       0.38318857, 0.25855801, 0.30367255, 0.52062931, 0.41335016])
```

We pass a model, observations, labels, and a number of folds to the `cross_val_score` function.

`cross_val_score` performs `cv` splits. On each split, we hold out one fold, train on the rest, and evaluate on the held-out fold. `cross_val_score` returns an array of the scores obtained in this way.

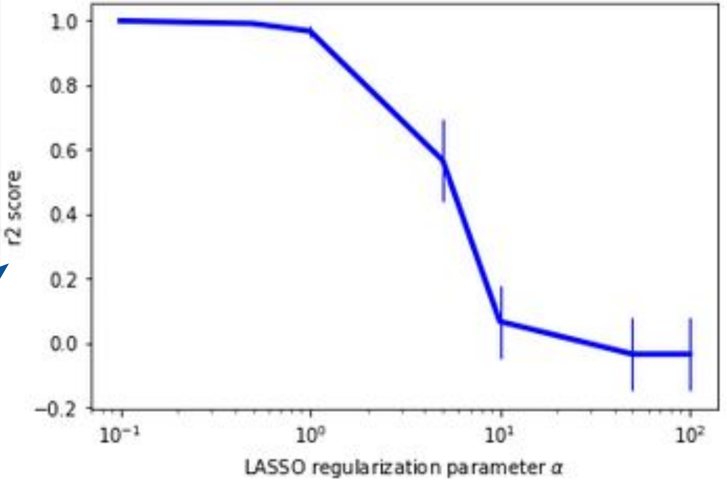
Cross-validation in `sklearn`

Now, we're going to do exactly the same thing, but for several different choices of `alpha`.



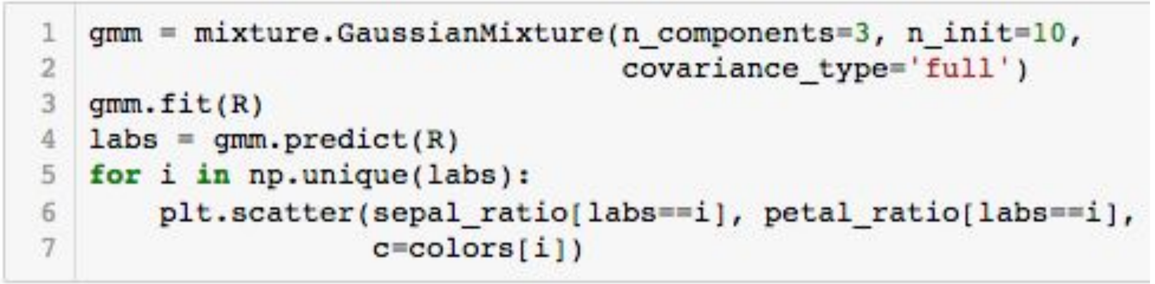
```python
from sklearn.model_selection import cross_val_score
alphavals = np.array([0.1, 0.5, 1.0, 5, 10.0, 50, 100])
mean_scores = np.zeros(alphavals.shape)
sd_scores = np.zeros(alphavals.shape)

for i in range(len(alphavals)):
    lasso = Lasso(alpha=alphavals[i])
    scores = cross_val_score(lasso, X, y, cv=10)
    mean_scores[i] = np.mean(scores)
    sd_scores[i] = np.std(scores)

plt.errorbar(alphavals, mean_scores, yerr=2*sd_scores,
             color='blue', linewidth=3, elinewidth=1)
plt.xscale('log'); plt.ylabel('r2 score')
_=plt.xlabel(r'LASSO regularization parameter $\alpha$')
```

By default, `cross_val_score` evaluates based on the `score` method supplied by the model. This can be changed by specifying the `scoring` parameter.



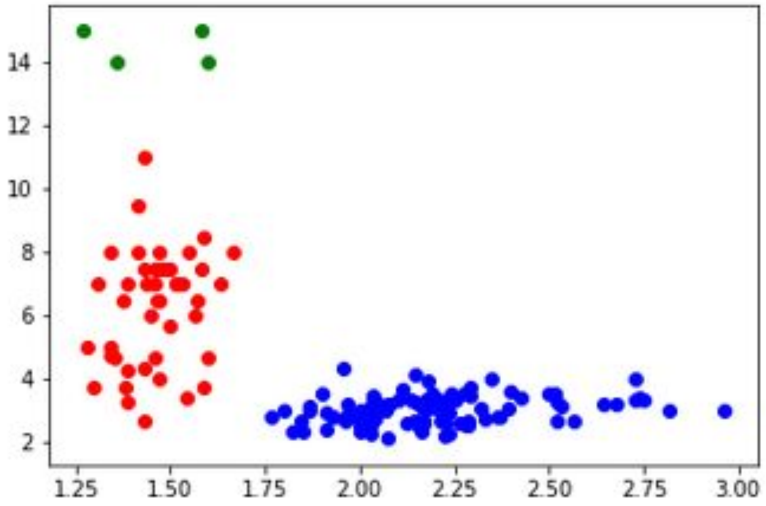
The `score` method of the `Lasso` model is the `r2score`, which we saw a few slides ago.

Assessing Models: `sklearn.metrics`

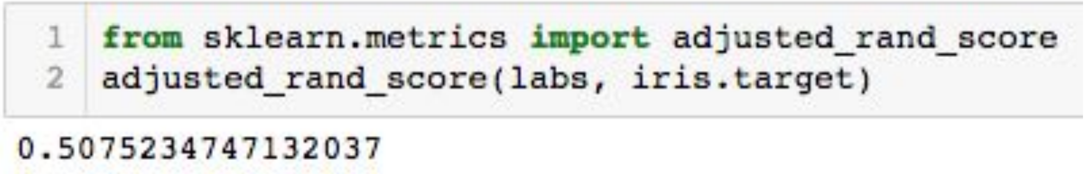


```python
gmm = mixture.GaussianMixture(n_components=3, n_init=10,
                              covariance_type='full')
gmm.fit(R)
labs = gmm.predict(R)
for i in np.unique(labs):
    plt.scatter(sepal_ratio[labs==i], petal_ratio[labs==i],
                c=colors[i])
```

`sklearn.metrics` contains a bunch of useful methods for evaluating models.



Example: the adjusted Rand index measures how well two clusterings agree. It’s a good measure of how well a clustering that we come up with agrees with the truth. ARI=1 is perfect, ARI=0 is random chance.

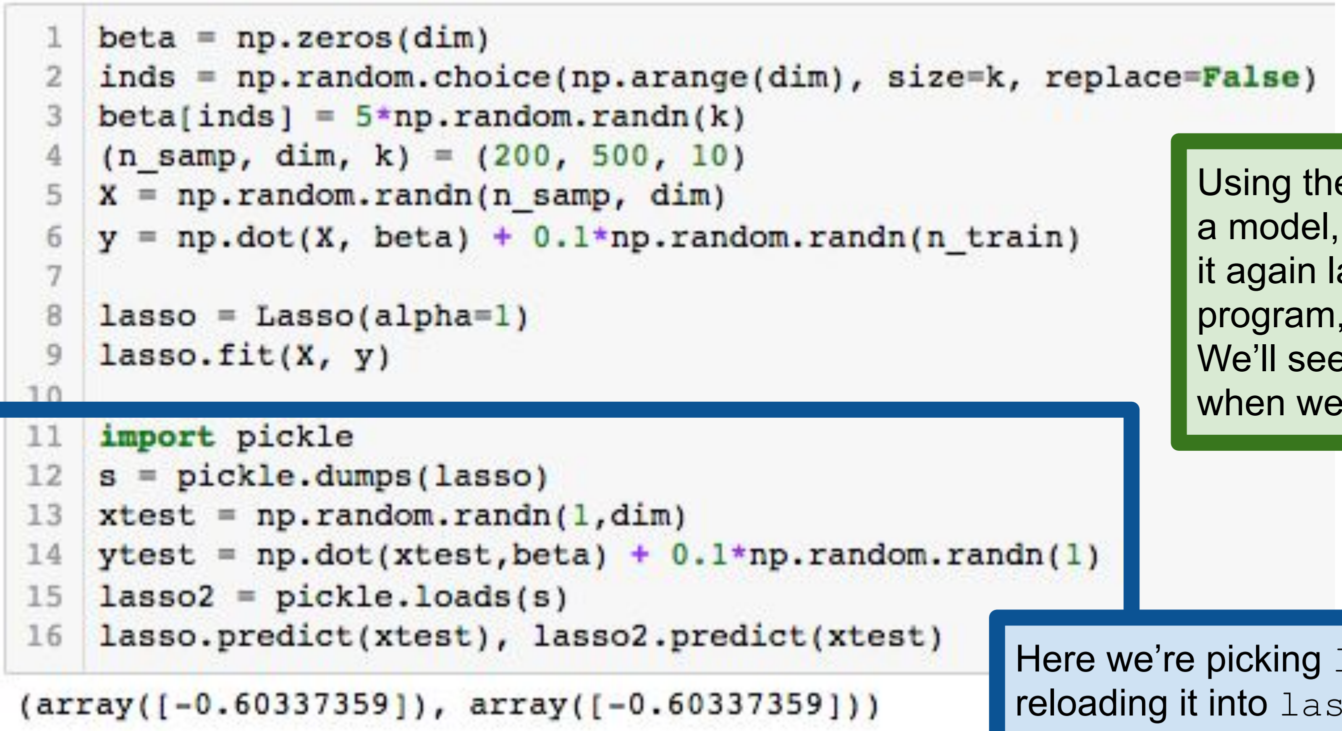


```python
from sklearn.metrics import adjusted_rand_score
adjusted_rand_score(labs, iris.target)
```

```
0.507523474713203 7
```

https://scikit-learn.org/stable/modules/model_evaluation.html

Model persistence: pickling model objects



```python
beta = np.zeros(dim)
inds = np.random.choice(np.arange(dim), size=k, replace=False)
beta[inds] = 5*np.random.randn(k)
(n_samp, dim, k) = (200, 500, 10)
X = np.random.randn(n_samp, dim)
y = np.dot(X, beta) + 0.1*np.random.randn(n_train)

lasso = Lasso(alpha=1)
lasso.fit(X, y)

import pickle
s = pickle.dumps(lasso)
xtest = np.random.randn(1,dim)
ytest = np.dot(xtest,beta) + 0.1*np.random.randn(1)
lasso2 = pickle.loads(s)
lasso.predict(xtest), lasso2.predict(xtest)
```

```
(array([-0.60337359]), array([-0.60337359]))
```

Using the pickle module, we can train a model, and save it in a file and load it again later (e.g., for use in a different program, on a different data set, etc.). We'll see a similar pattern again soon when we discuss TensorFlow.

Here we're picking `lasso`, and reloading it into `lasso2`. Note that the two models are indeed the same.