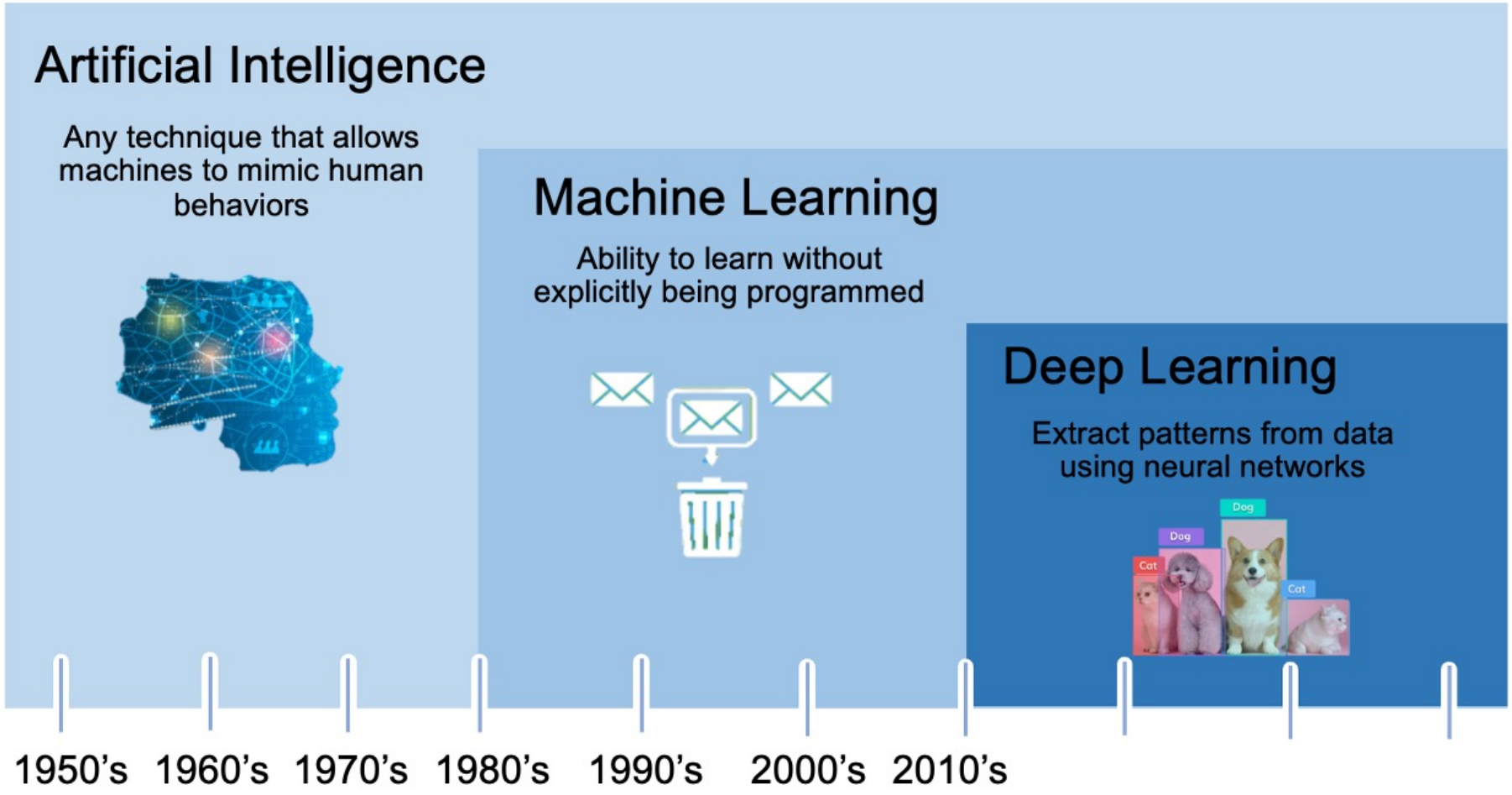
STATS 507
Data Analysis in Python

Week9-1: scikit learn in practice

Dr. Xian Zhang

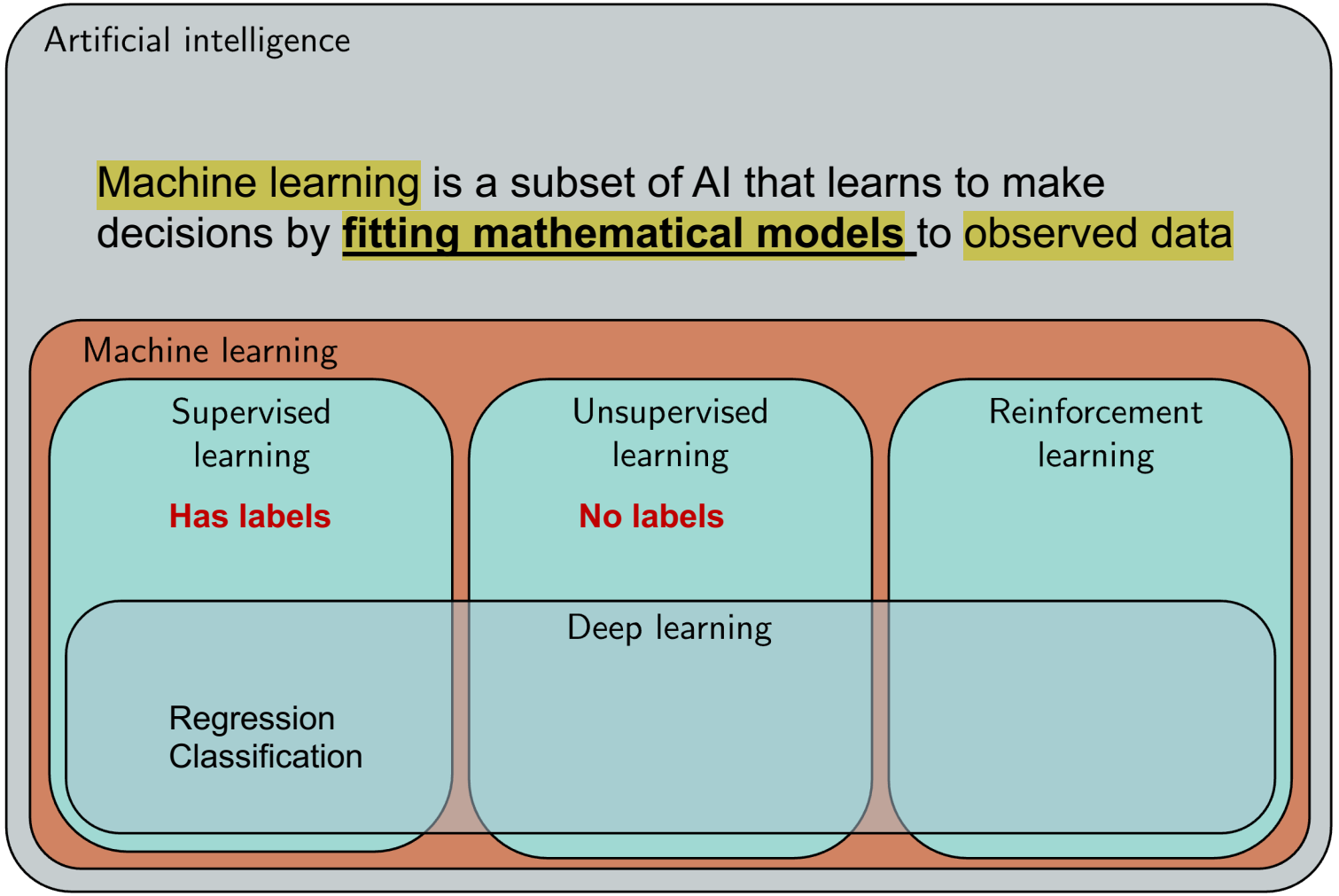
Recap: What is machine learning?



Slide Credit: Alexander Amini

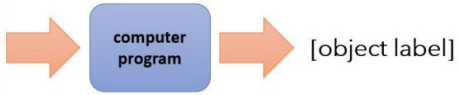
Modified from MIT open course: 6.S191 and Nvidia blog

Recap: What is machine learning?



Machine learning

supervised learning



input: $\mathbf{x}$

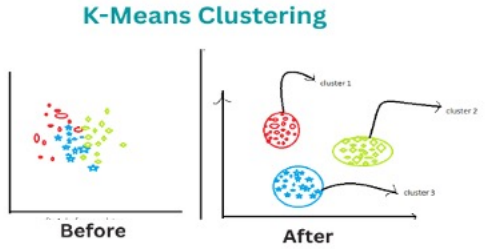
output: $\mathbf{y}$

data: $\mathcal{D} = \{(\mathbf{x}_i, \mathbf{y}_i)\}$

goal: $f_\theta(\mathbf{x}_i) \approx \mathbf{y}_i$

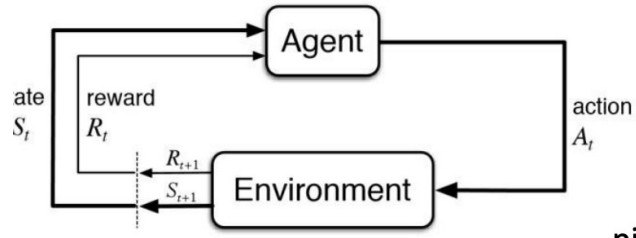
someone gives this to you

unsupervised learning



input: unlabeled data
output: Hidden structure of the data
data: $\boldsymbol{D} = \{\boldsymbol{x_i}\}$
goal: learn some hidden or underlying structure of the data

reinforcement learning



pick your own actions

input: $\mathbf{s}_t$ *at each time step*

output: $\mathbf{a}_t$ *at each time step*

data: $(\mathbf{s}_1, \mathbf{a}_1, r_1, \ldots, \mathbf{s}_T, \mathbf{a}_T, r_T)$

goal: learn $\pi_\theta : \mathbf{s}_t \rightarrow \mathbf{a}_t$
to maximize $\sum_t r_t$

Ref: https://rail.eecs.berkeley.edu/deeprlcourse/

Scikit learn

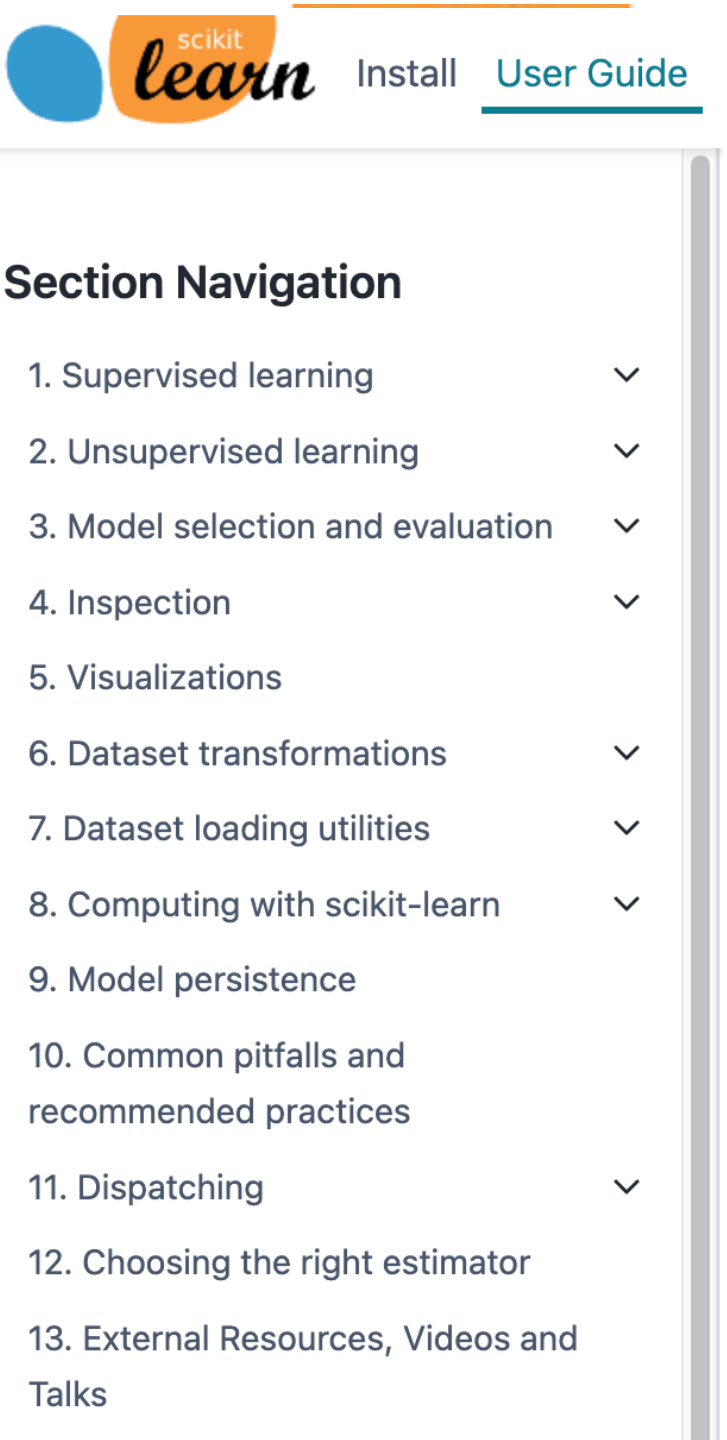
SciPy Tool**kit**
Open source machine learning library

- Built atop NumPy, SciPy and Matplotlib.

Makes many common ML/stats models easily available.

- Supervised learning example
  - Classification
  - Regression
- Unsupervised learning example

https://scikit-learn.org/stable/user_guide.html#user-guide



1. Classification example in scikit-learn

2. Regression example in scikit-learn

3. Unsupervised learning in scikit-learn
(if we have time…)

Problem: recognizing hand-writing digits

Real world input

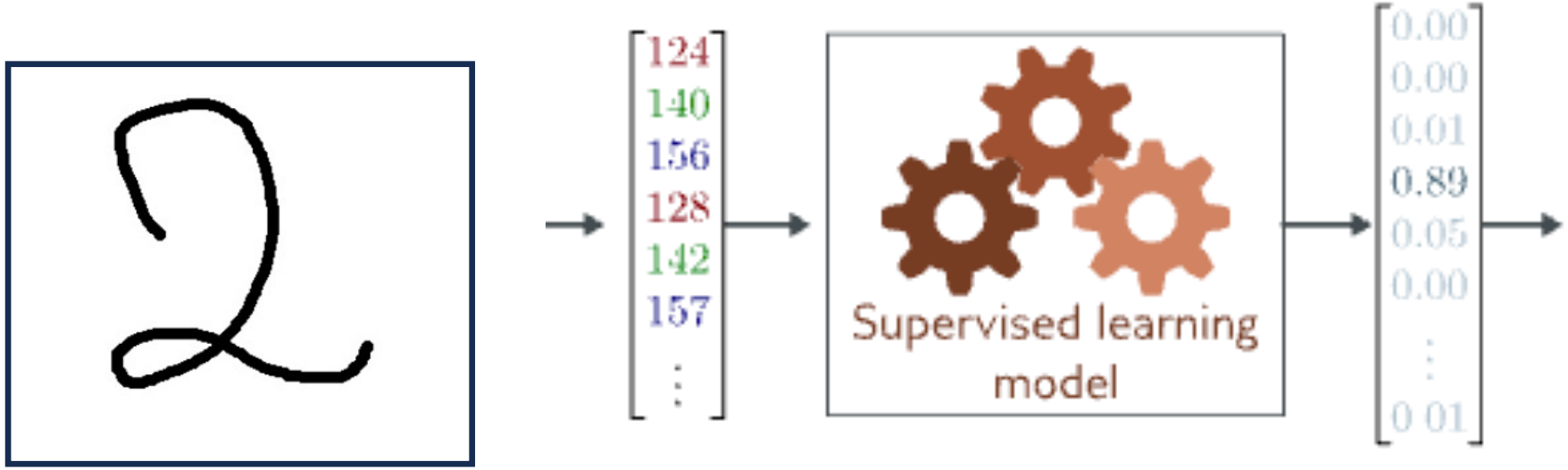
Model input

Model

SVM

Model output

Real world output



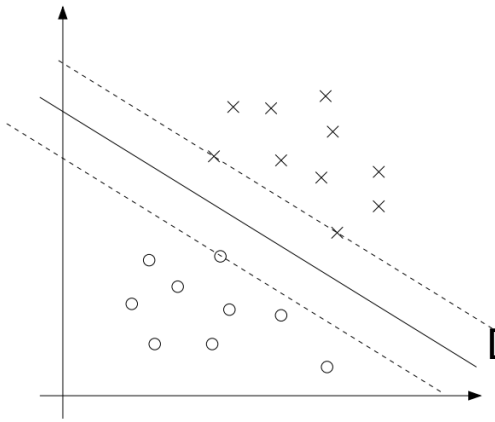
Classification or regression?

Binary or multi-class?

Which model to use? -> SVM

To apply a **classifier** on this data, we need to select/create a **classifier (model).**

Scikit-learn provides a wide range of classifiers for different types of machine learning tasks, in this example, we are going to look at support vector machine (SVM).

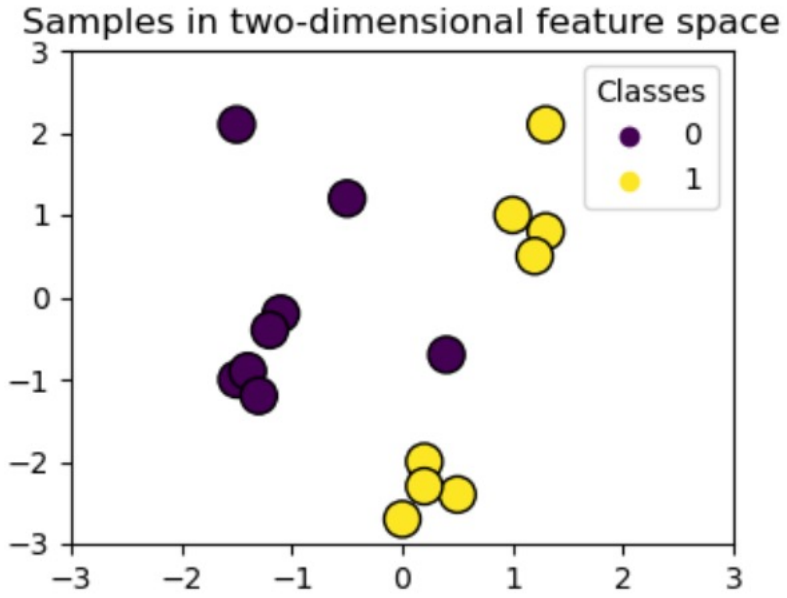


Decision boundary/a separating hyperplane

https://see.stanford.edu/materials/aimlcs229/cs229-notes3.pdf

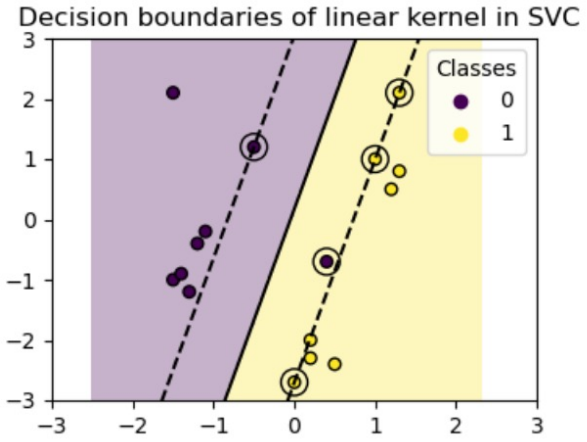
Kernels in SVM

Goal: separates the classes in their training data using SVM



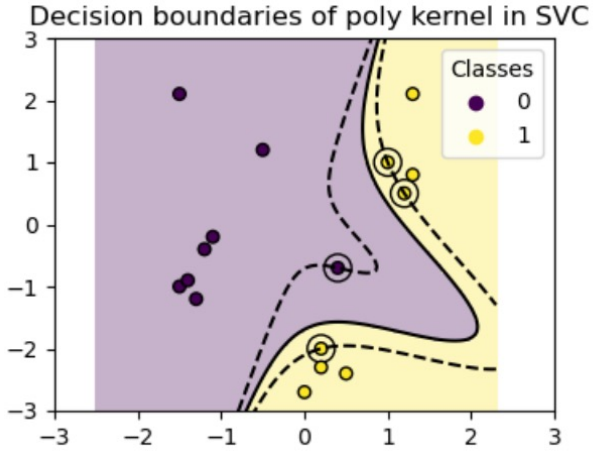
The samples are not clearly separable by a straight lin

More kernel can be found in:
https://scikit-learn.org/dev/auto_examples/svm/



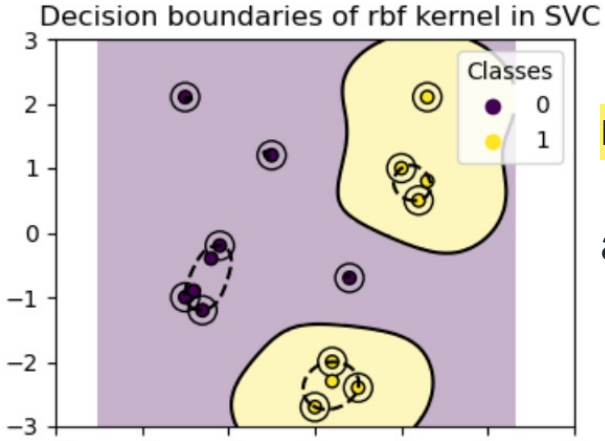
Hyperplane: straight line

Lack of expressivity



Hyperplane: straight line

adapts well to the training data



rbf: radial basis function (default)

adapts well to the training data

First: Load the dataset

```
1 from sklearn import datasets
2 digits = datasets.load_digits()
3 len(digits.data), len(digits.data[42])
```

```
(1797, 64)
```

`sklearn` includes a number of built-in data sets, among which is a version of the famous MNIST digits data set.

```
1 digits.data[42]
```

```
array([ 0.,  0.,  0.,  0., 12.,  5.,  0.,  0.,  0.,  0.,  0.,  2., 16.,
       12.,  0.,  0.,  0.,  0.,  1., 12., 16., 11.,  0.,  0.,  0.,  2.,
       12., 16., 16., 10.,  0.,  0.,  0.,  6., 11.,  5., 15.,  6.,  0.,
        0.,  0.,  0.,  0.,  1., 16.,  9.,  0.,  0.,  0.,  0.,  0.,  2.,
       16., 11.,  0.,  0.,  0.,  0.,  0.,  3., 16.,  8.,  0.,  0.])
```

```
1 digits.images[42]
```

```
array([[ 0.,  0.,  0.,  0., 12.,  5.,  0.,  0.],
       [ 0.,  0.,  0.,  2., 16., 12.,  0.,  0.],
       [ 0.,  0.,  1., 12., 16., 11.,  0.,  0.],
       [ 0.,  2., 12., 16., 16., 10.,  0.,  0.],
       [ 0.,  6., 11.,  5., 15.,  6.,  0.,  0.],
       [ 0.,  0.,  0.,  1., 16.,  9.,  0.,  0.],
       [ 0.,  0.,  0.,  2., 16., 11.,  0.,  0.],
       [ 0.,  0.,  0.,  3., 16.,  8.,  0.,  0.]])
```

`digits.data` is an array, entries of which are vectors of length 64, which correspond to images.
`digits.images` give us a 8x8 pixel images of digits.
`digits.target` gives us the digit that the images signifies

```
1 digits.target[42]
```

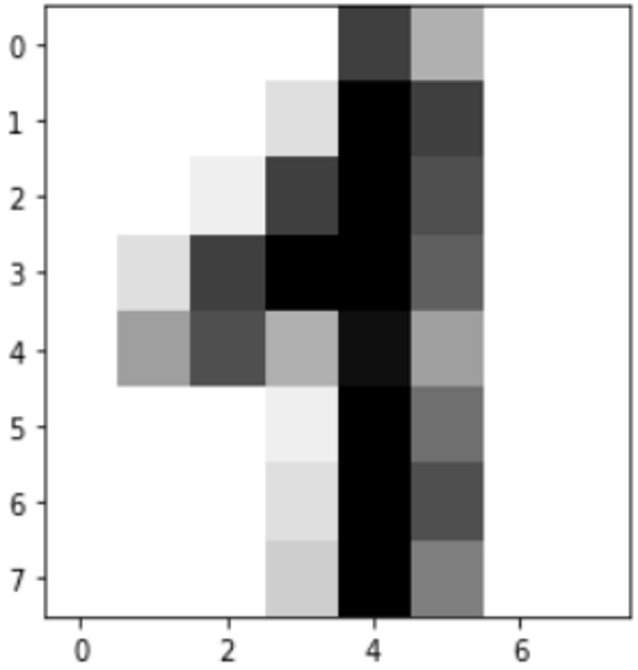
```
1
```

Reference:
https://scikit-learn.org/stable/auto_examples/classification/plot_digits_classification.html#digits-dataset

Plot the dataset

```
1 import matplotlib.pyplot as plt
2 plt.imshow(digits.images[42], cmap=plt.cm.gray_r)
```

```
<matplotlib.image.AxesImage at 0x7fe99715e590>
```




```
1 from collections import Counter
2 Counter(digits.target)
```

```
Counter({0: 178,
         1: 182,
         2: 177,
         3: 183,
         4: 181,
         5: 182,
         6: 181,
         7: 179,
         8: 174,
         9: 180})
```

Create a classifier model

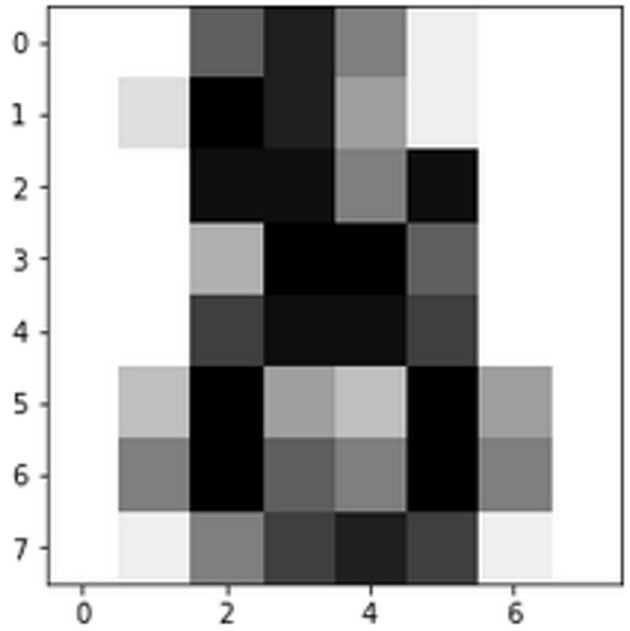
```
1 from sklearn import svm
2 clf = svm.SVC()  # support vector classifier
3 clf.fit(X[:-1], y[:-1])
```

```
SVC()
```

```
1 clf.predict(X[-1:])
```

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

SVC is a support vector machine (SVM) classifier, one of many classifiers that `sklearn` provides.



Reference:https://scikit-learn.org/stable/modules/svm.html

Train the classifier using `fit`

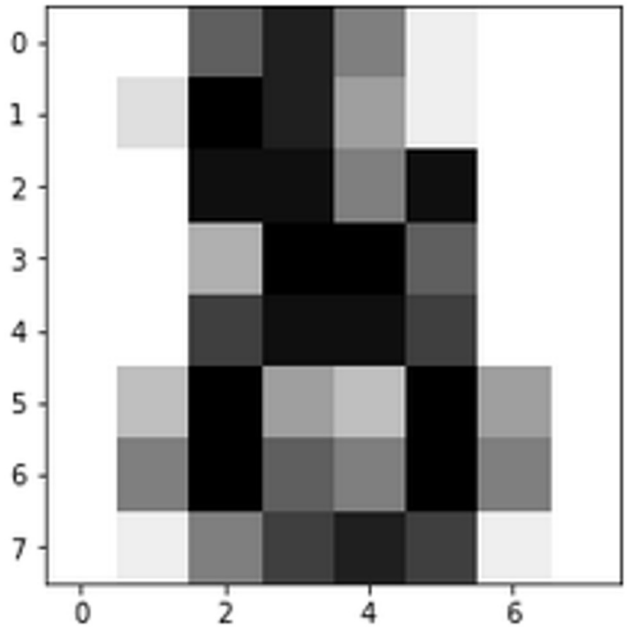
```
1 from sklearn import svm
2 clf = svm.SVC()  # support vector classifier
3 clf.fit(X[:-1], y[:-1])
```

```
SVC()
```

```
1 clf.predict(X[-1:])
```

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

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



Train the classifier using `fit`

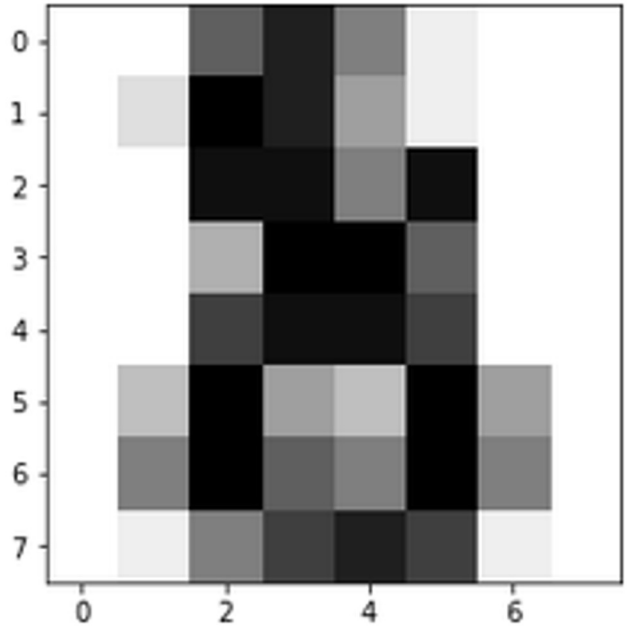
```
1 from sklearn import svm
2 clf = svm.SVC()  # support vector classifier
3 clf.fit(X[:-1], y[:-1])
```

```
SVC()
```

```
1 clf.predict(X[-1:])
```

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

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. Non typical to use all but 1 for training, but this is just an example



Use the trained model to predict

```
1 from sklearn import svm
2 clf = svm.SVC()  # support vector classifier
3 clf.fit(X[:-1], y[:-1])
```

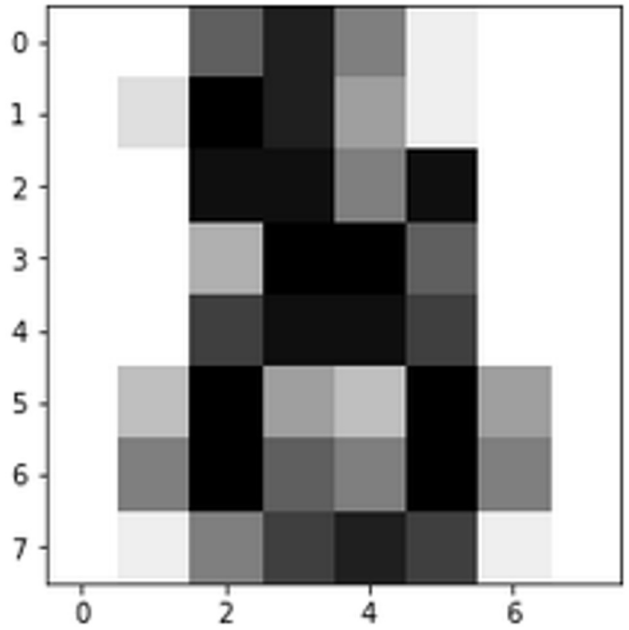
```
SVC()
```

```
1 clf.predict(X[-1:])
```

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

```
1 digits.target[-1]
```

```
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.

Evaluate model performance

- Given a set of labeled data and such a predictive model, every data point lies in one of four categories (for a classification problem)
  - True positive: “This message is spam, and we correctly predicted spam.”
  - False positive (Type 1 Error): “This message is not spam, but we predicted spam.”
  - False negative (Type 2 Error): “This message is spam, but we predicted not spam.”
  - True negative: “This message is not spam, and we correctly predicted not spam.”

$$\text{Precision} = \frac{\text{correctly classified actual positives}}{\text{everything classified as positive}} = \frac{TP}{TP + FP}$$

$$\text{Recall (or TPR)} = \frac{\text{correctly classified actual positives}}{\text{all actual positives}} = \frac{TP}{TP + FN}$$

$$\text{Accuracy} = \frac{\text{correct classifications}}{\text{total classifications}} = \frac{TP + TN}{TP + TN + FP + FN}$$

In-class practice

1. Classification example in scikit-learn

2. Regression example in scikit-learn

3. Unsupervised learning in scikit-learn

Supervised Learning: Linear Regression

Linear regression

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

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

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

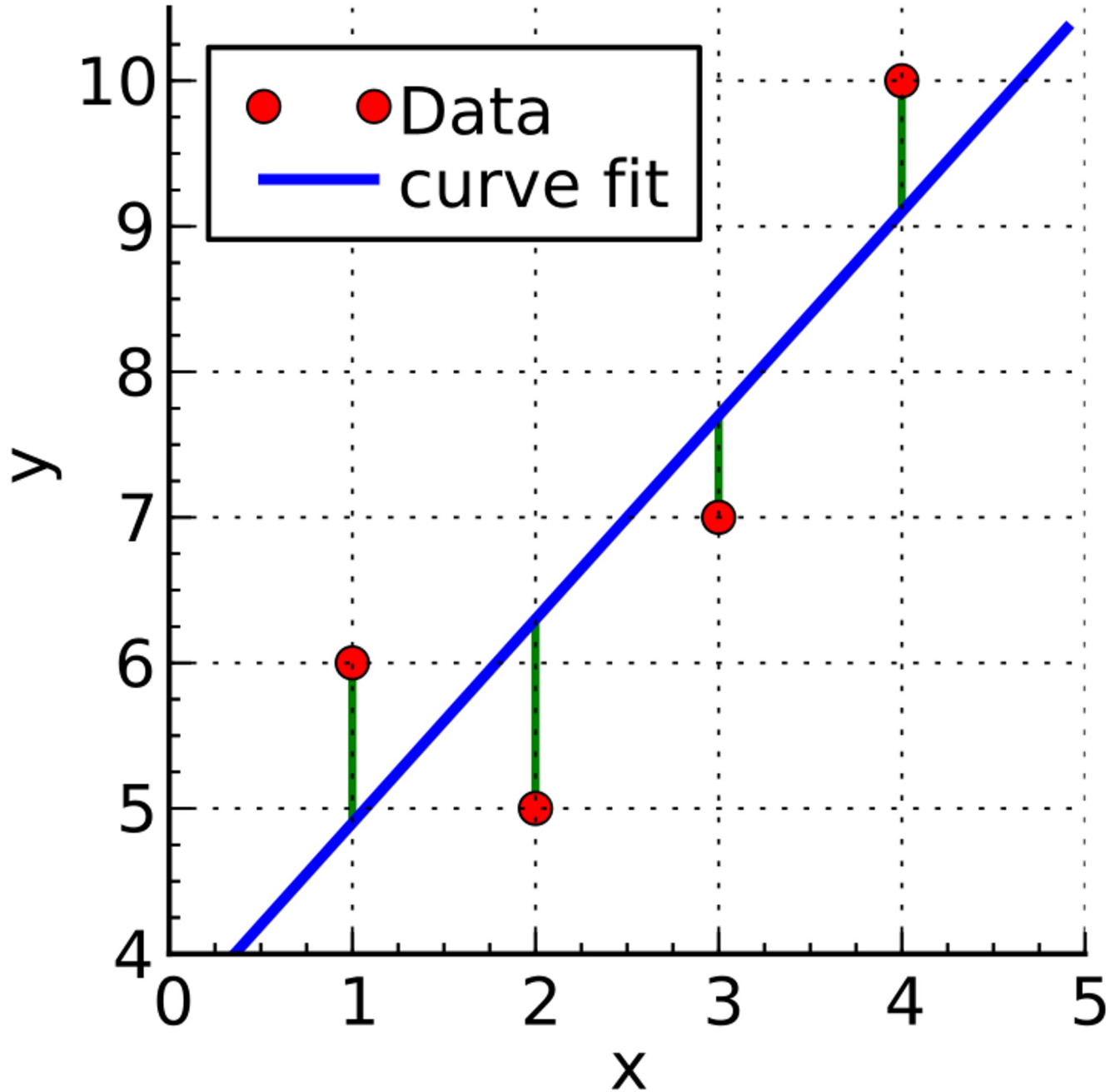
Noise $\epsilon \in \mathbb{R}^{n}$

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 and thereby derived the line of best fit. OLS is computationally convenient.


Data
curve fit
y
x
10
9
8
7
6
5
4
0
1
2
3
4
5

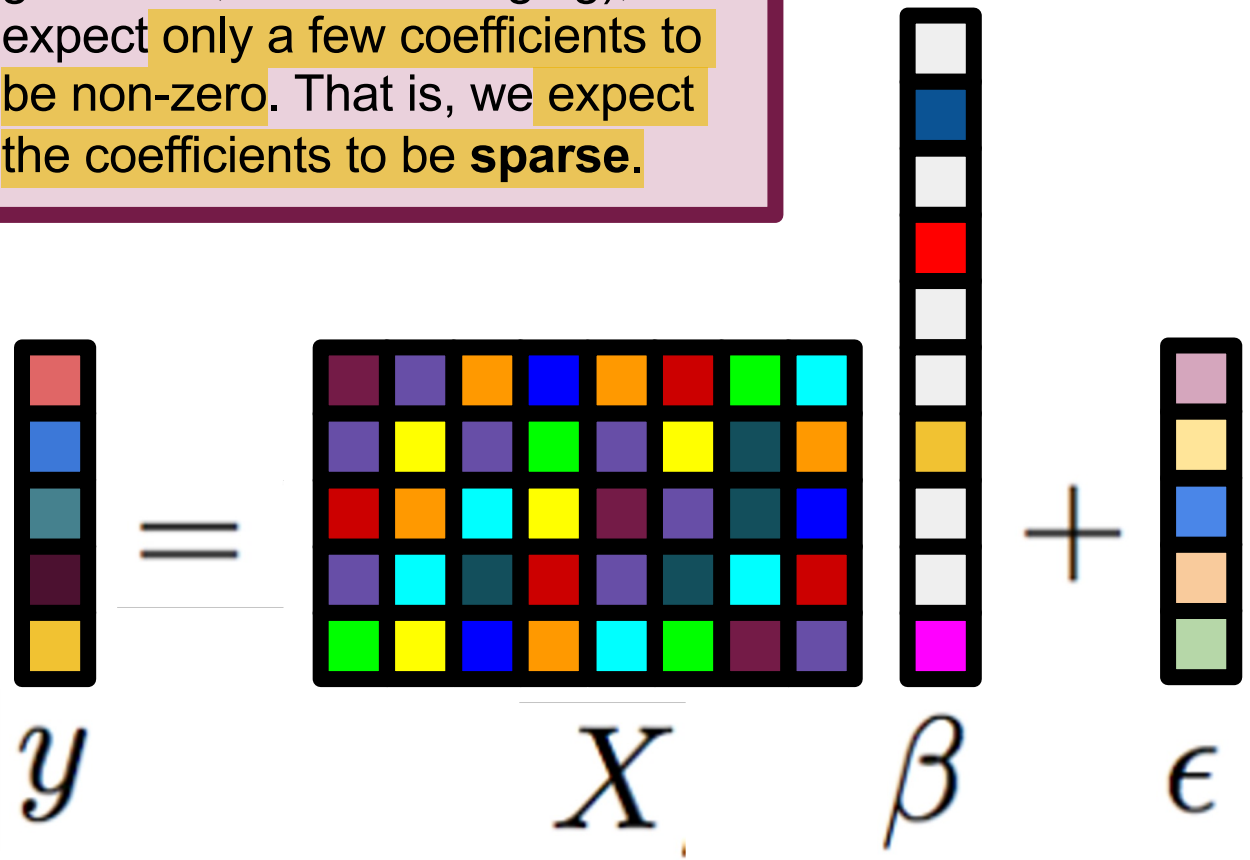
We want the coefficients to be sparse

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: LASSO Regression

Ordinary least squares (OLS)

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

LASSO

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

L2 objective

L1 penalty

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

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.

LASSO Regression in scikit learn

```
(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:]
```

Generate data; split into train/test.

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

Fit the model based on the train set.

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

Assess how well the model fits the test data.

```
0.9635771805872204
```

LASSO Regression in scikit learn

```
1  (n_samp, dim, k) = (200, 500, 10)
2  X = np.random.randn(n_samp, dim)
3  beta = np.zeros(dim)
4  inds = np.random.choice(np.arange(dim), size=k, replace=False)
5  beta[inds] = 5*np.random.randn(k)
6  y = np.dot(X, beta) + 0.1*np.random.randn(n_samp)
7
8  # Split data into train set and test set
9  X_train, y_train = X[:(n_samp//2)], y[:(n_samp//2)]
10 X_test, y_test = X[n_samp // 2:], y[n_samp // 2:]
11
12 #import and train the model.
13 from sklearn.linear_model import Lasso
14 lasso = Lasso(alpha=1)
15 lasso.fit(X_train, y_train)
16
17 y_pred_lasso = lasso.predict(X_test)
18 from sklearn.metrics import r2_score
19 r2_score(y_test, y_pred_lasso)
```

200 points in 500 dimensions (**X**). Sparsity `k=10`.

Fit the model based on the train set.

Assess how well the model fits the test data.

```
0.9635771805872204
```

LASSO Regression in scikit learn

```
(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)
```

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.

```
0.9635771805872204
```

LASSO Regression in scikit learn

```
1  (n_samp, dim, k) = (200, 500, 10)
2  X = np.random.randn(n_samp, dim)
3  beta = np.zeros(dim)
4  inds = np.random.choice(np.arange(dim), size=k, replace=False)
5  beta[inds] = 5*np.random.randn(k)
6  y = np.dot(X, beta) + 0.1*np.random.randn(n_samp)
7
8  # Split data into train set and test set
9  X_train, y_train = X[:(n_samp//2)], y[:(n_samp//2)]
10 X_test, y_test = X[n_samp // 2:], y[n_samp // 2:]
11
12 #import and train the model.
13 from sklearn.linear_model import Lasso
14 lasso = Lasso(alpha=1)
15 lasso.fit(X_train, y_train)
16
17 y_pred_lasso = lasso.predict(X_test)
18 from sklearn.metrics import r2_score
19 r2_score(y_test, y_pred_lasso)
```

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.

```
0.9635771805872204
```

LASSO Regression in scikit learn

```
(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.

LASSO Regression in scikit learn

```
(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.

Assess how well the model fits the test data.

LASSO Regression in scikit learn

```
(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.

LASSO Regression in scikit learn

```
1  (n_samp, dim, k) = (200, 500, 10)
2  X = np.random.randn(n_samp, dim)
3  beta = np.zeros(dim)
4  inds = np.random.choice(np.arange(dim), size=k, replace=False)
5  beta[inds] = 5*np.random.randn(k)
6  y = np.dot(X, beta) + 0.1*np.random.randn(n_samp)
7
8  # Split data into train set and test set
9  X_train, y_train = X[:(n_samp//2)], y[:(n_samp//2)]
10 X_test, y_test = X[n_samp // 2:], y[n_samp // 2:]
11
12 #import and train the model.
13 from sklearn.linear_model import Lasso
14 lasso = Lasso(alpha=1)
15 lasso.fit(X_train, y_train)
16
17 y_pred_lasso = lasso.predict(X_test)
18 from sklearn.metrics import r2_score
19 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.

LASSO Regression in scikit learn

```
(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`.

LASSO Regression in scikit learn

```
(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:]
```

Generate data; split into train/test.

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

Fit the model based on the train set.

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

```
0.9635771805872204
```

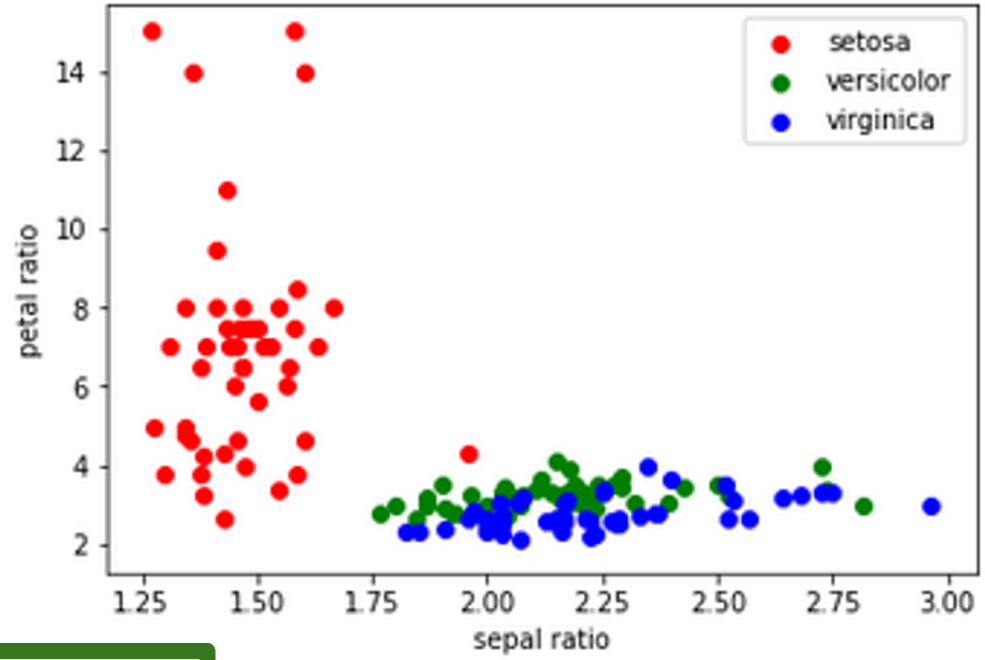
`r2score` is just one of the many ways to assess whether or not we're doing well. 1 is perfect performance, 0 is "chance".
https://en.wikipedia.org/wiki/Coefficient_of_determination

1. Classification example in scikit-learn
2. Regression example in scikit-learn
3. Unsupervised learning in scikit-learn

Gaussian Mixture Model

Unsupervised learning example: k-GMM in

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



Here's Fisher's famous iris data set. 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 : k-GMM in sklearn

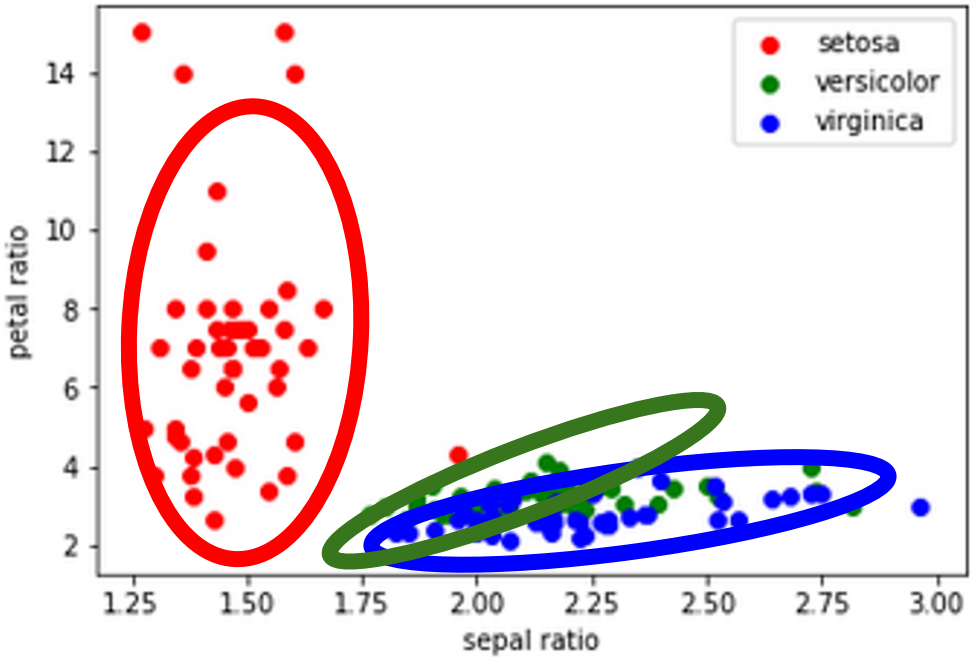
Basic idea: model data as mixture of Gaussians

each Gaussian generates one cluster

For each cluster, estimate mean and covariance

Computationally hard...

...but can approximate via EM



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

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

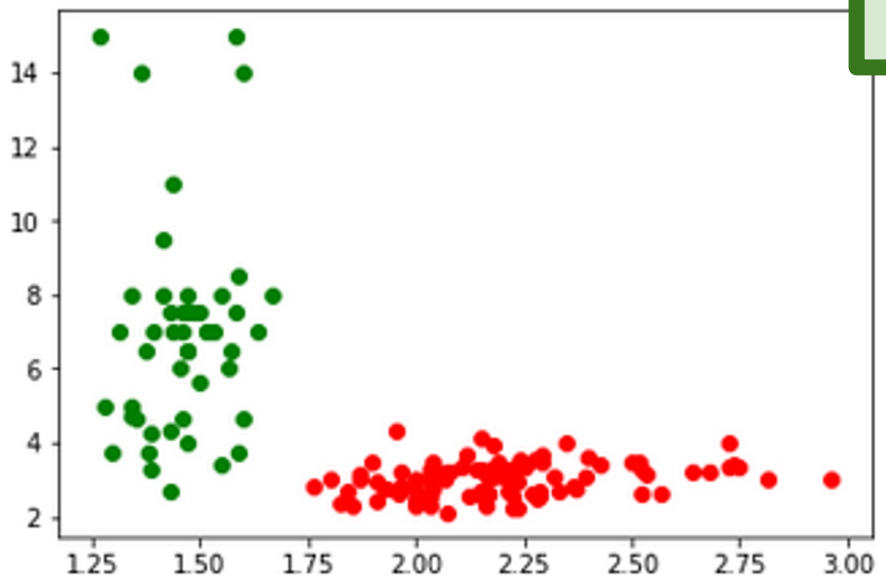
Unsupervised learning : k-GMM in sklearn

```
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])
```

Fit the model to the data.

Retrieve the (estimated) labels.



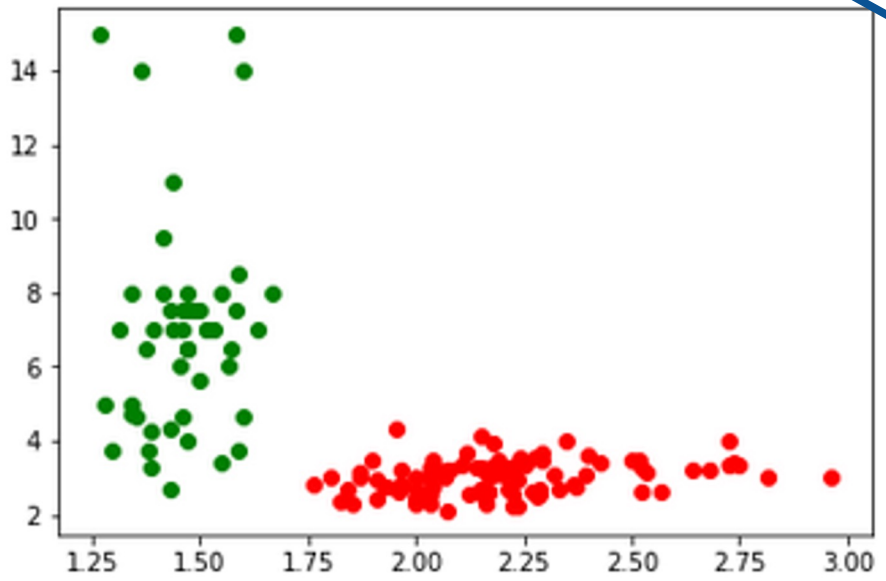
Unsupervised learning : k-GMM in sklearn

```
1  from sklearn import mixture
2  R = np.stack([sepal_ratio,petal_ratio], axis=1)
3  gmm = mixture.GaussianMixture(n_components=2, n_init=10,
4                                covariance_type='full')
5  gmm.fit(R)
6
7  labs = gmm.predict(R)
8  for i in np.unique(labs):
9      plt.scatter(sepal_ratio[labs==i], petal_ratio[labs==i],
10                 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.

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

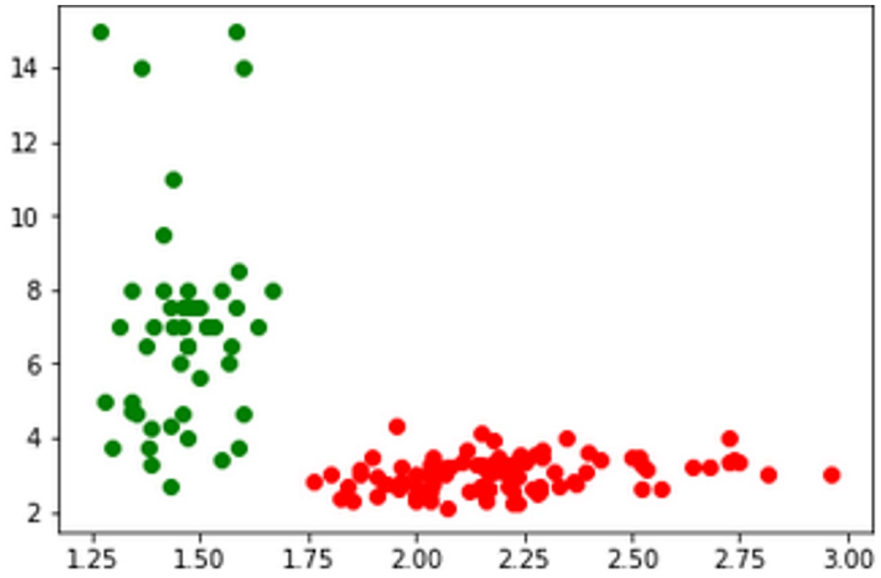


Unsupervised learning : k-GMM in sklearn

```
1  from sklearn import mixture
2  R = np.stack([sepal_ratio,petal_ratio], axis=1)
3  gmm = mixture.GaussianMixture(n_components=2, n_init=10,
4                                covariance_type='full')
5  gmm.fit(R)
6
7  labs = gmm.predict(R)
8  for i in np.unique(labs):
9      plt.scatter(sepal_ratio[labs==i],
10                 c=colors[i])
```

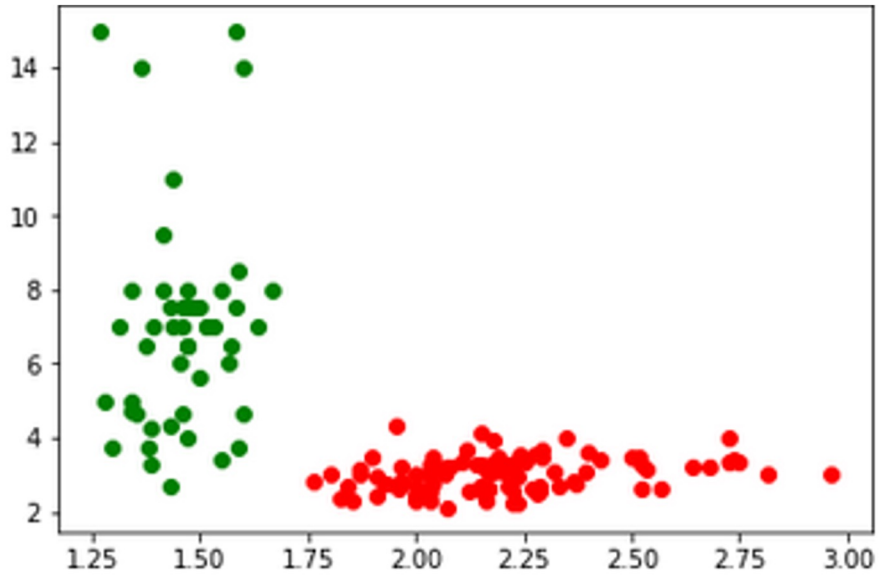
Fit a model with 2 components.

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**.



Unsupervised learning example: k-GMM

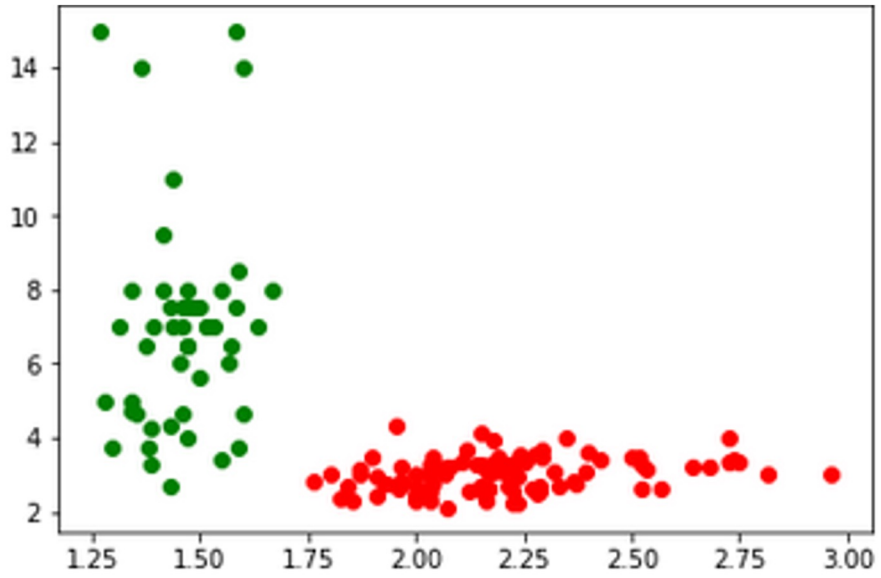
```
1  from sklearn import mixture
2  R = np.stack([sepal_ratio,petal_ratio], axis=1)
3  gmm = mixture.GaussianMixture(n_components=2, n_init=10,
4                                covariance_type='full')
5  gmm.fit(R)
6
7  labs = gmm.predict(R)
8  for i in np.unique(labs):
9      plt.scatter(sepal_ratio[labs==i], petal_ratio[labs==i],
10                 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

```
1  from sklearn import mixture
2  R = np.stack([sepal_ratio,petal_ratio], axis=1)
3  gmm = mixture.GaussianMixture(n_components=2, n_init=10,
4                                covariance_type='full')
5  gmm.fit(R)
6
7  labs = gmm.predict(R)
8  for i in np.unique(labs):
9      plt.scatter(sepal_ratio[labs==i], petal_ratio[labs==i],
10                 c=colors[i])
```

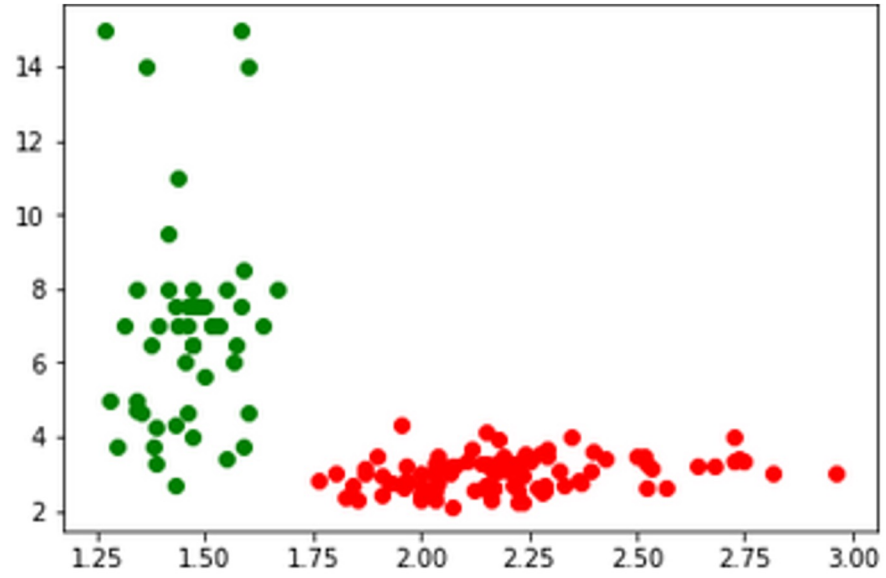


This tells `sklearn` to estimate a covariance matrix separately for each cluster. Other options include estimating one covariance shared across all clusters (`'tied'`) and estimating spherical covariances for each cluster (`'spherical'`).

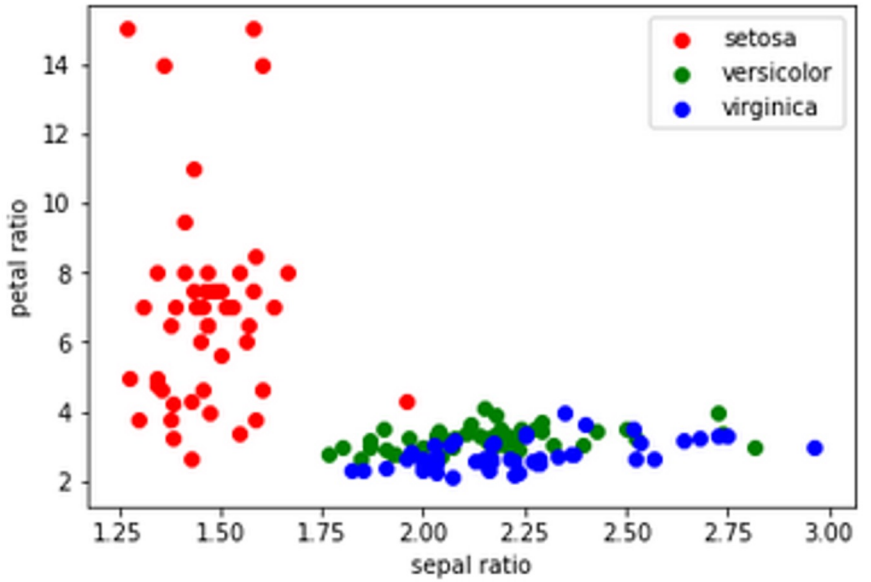
Unsupervised learning : k-GMM in sklearn

```
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])
```

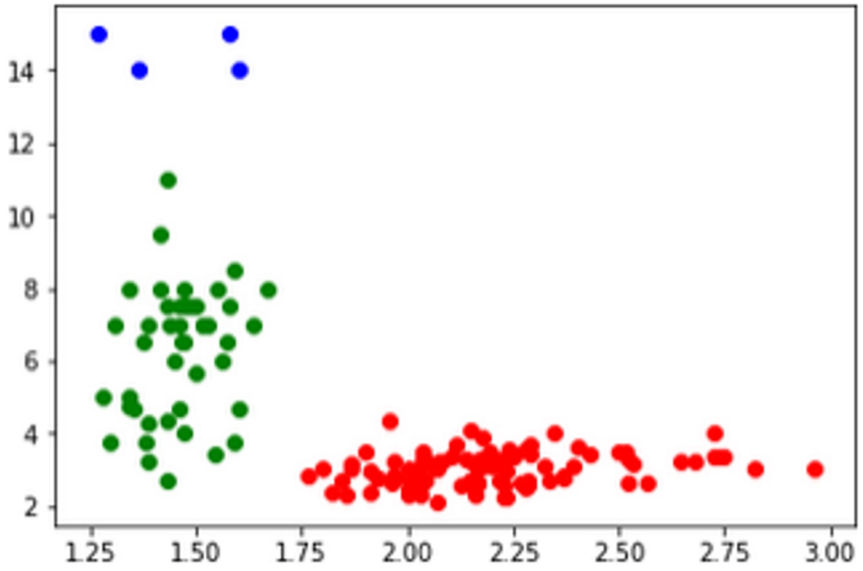


Of course, because we chose the wrong number of components, we fail to recover the true cluster structure of the data.

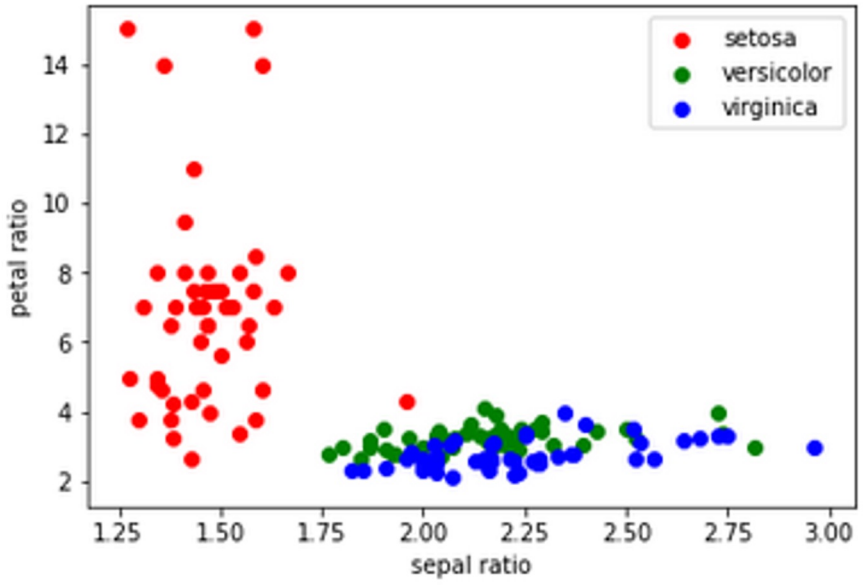


Unsupervised learning : k-GMM in sklearn

```
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])
```



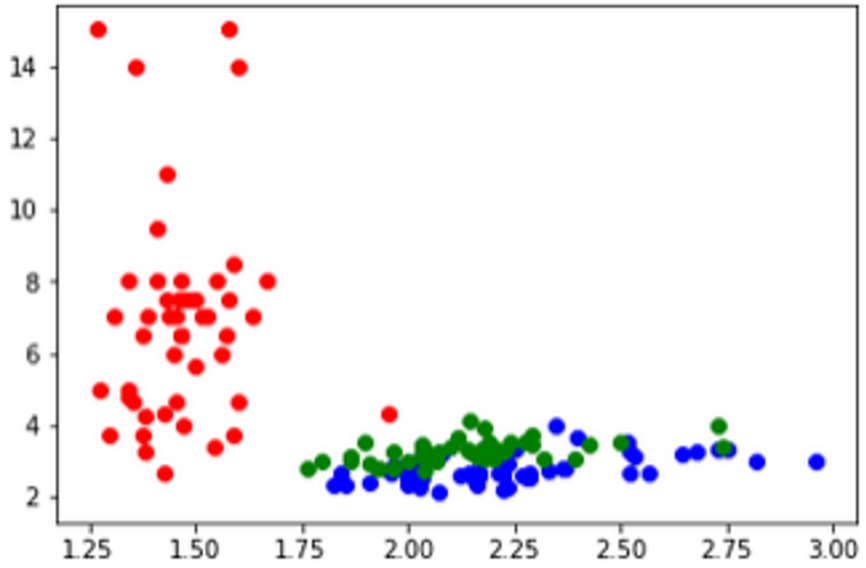
But even if we choose the correct number of components, the “ratio” representation of the data collapses the versicolor and virginica species, and we get a weird solution.



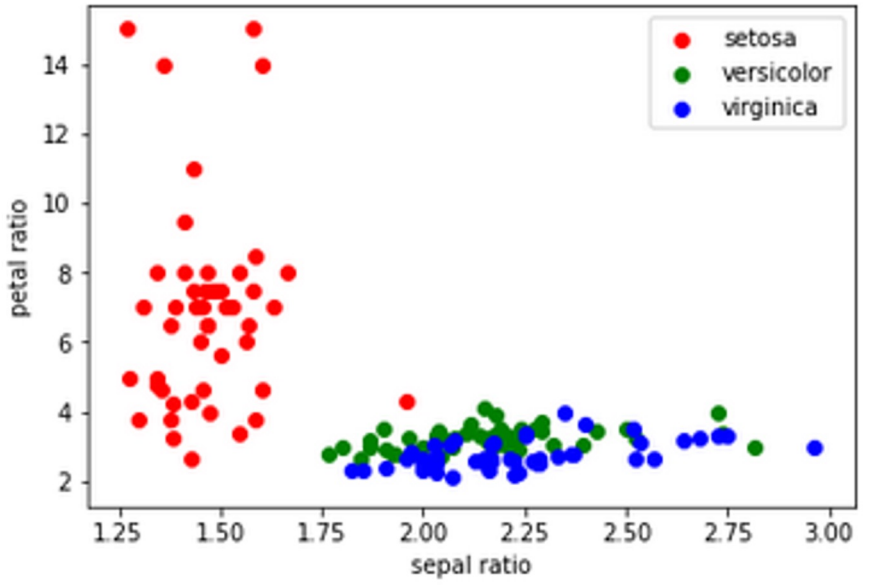
Unsupervised learning : k-GMM in sklearn

```
1  gmm = mixture.GaussianMixture(n_components=3, n_init=10,
2                                covariance_type='full')
3  gmm.fit(X)
4
5  labs = gmm.predict(X)
6  for i in np.unique(labs):
7      plt.scatter(sepal_ratio[labs==i], petal_ratio[labs==i],
8                  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:

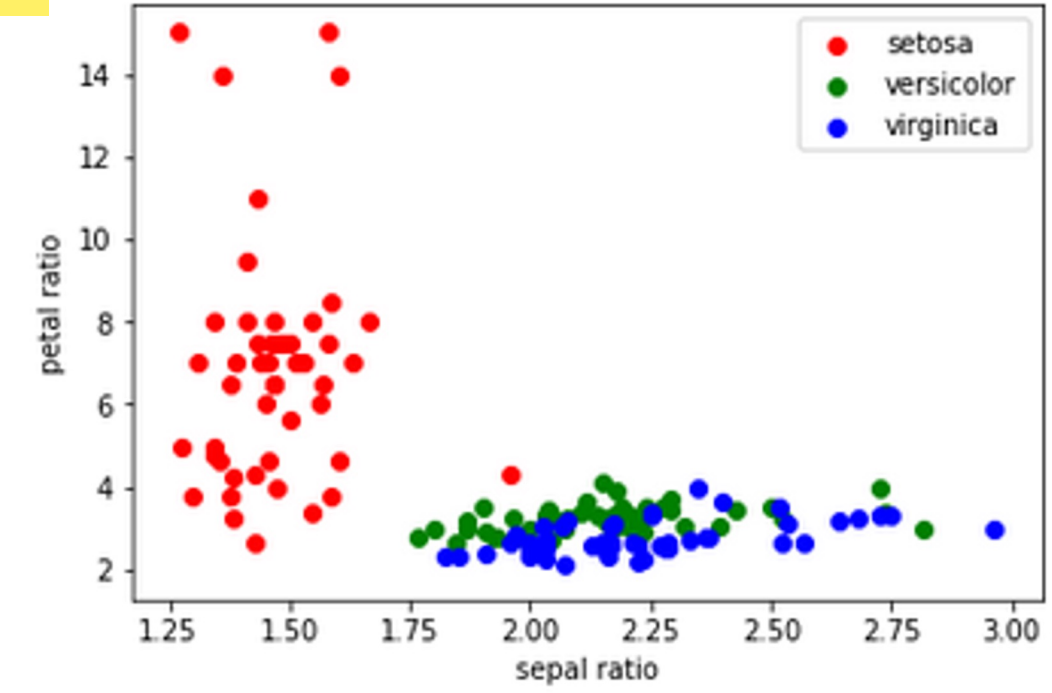
https://en.wikipedia.org/wiki/Bayesian_information_criterion

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

```
1  iris = datasets.load_iris()
2  X = iris.data
3  y = iris.target
4  n_components_range = range(1, 7)
5  covar_types = ['spherical', 'tied', 'diag', 'full']
6  bics = np.zeros(shape=(len(covar_types),len(n_components_range)))
7  for i in range(len(covar_types)):
8      cvtype = covar_types[i]
9      for j in range(len(n_components_range)):
10         n_comps = n_components_range[j]
11         # Fit a Gaussian mixture with EM
12         gmm = mixture.GaussianMixture(n_components=n_comps,
13                                       covariance_type=cvtype)
14         gmm.fit(X)
15         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.

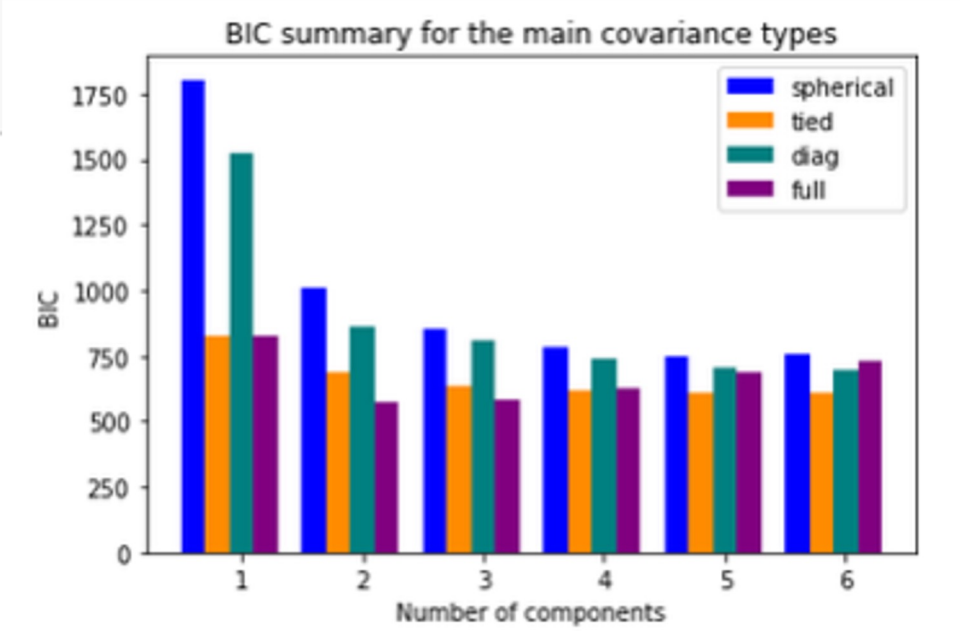
Measure BIC of each such choice; store it in the array `bics`.

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

```
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()
```

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.

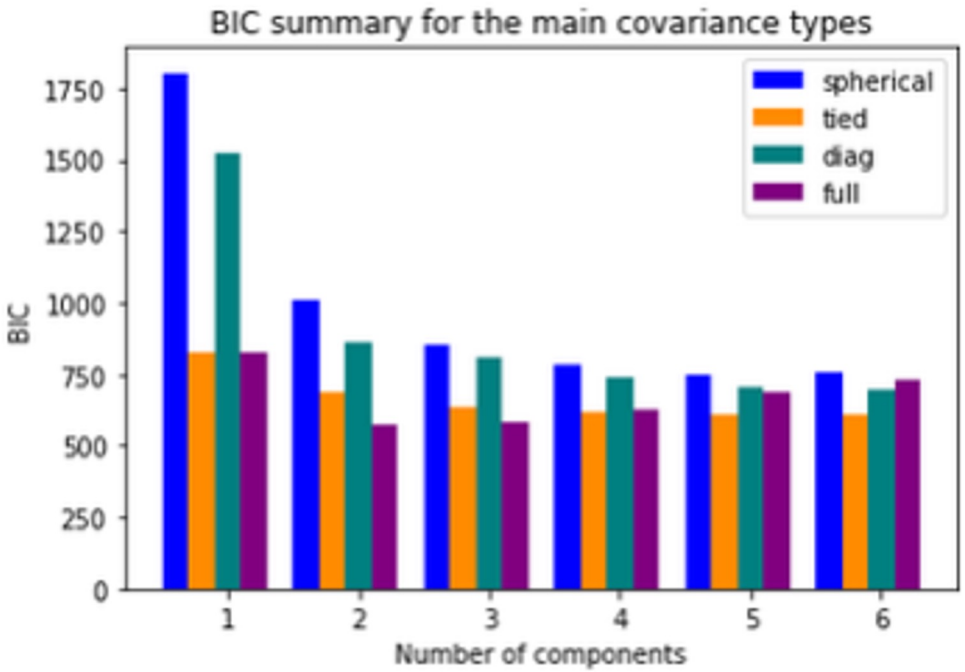


Model selection in sklearn

```
barwidth=0.2
inds = np.array(list(n_components_range))
colors = ['blue', 'darkorange', 'teal', 'purple']
for i in range(
    plt.bar(inds
            widt
plt.xticks(inds
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.

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.



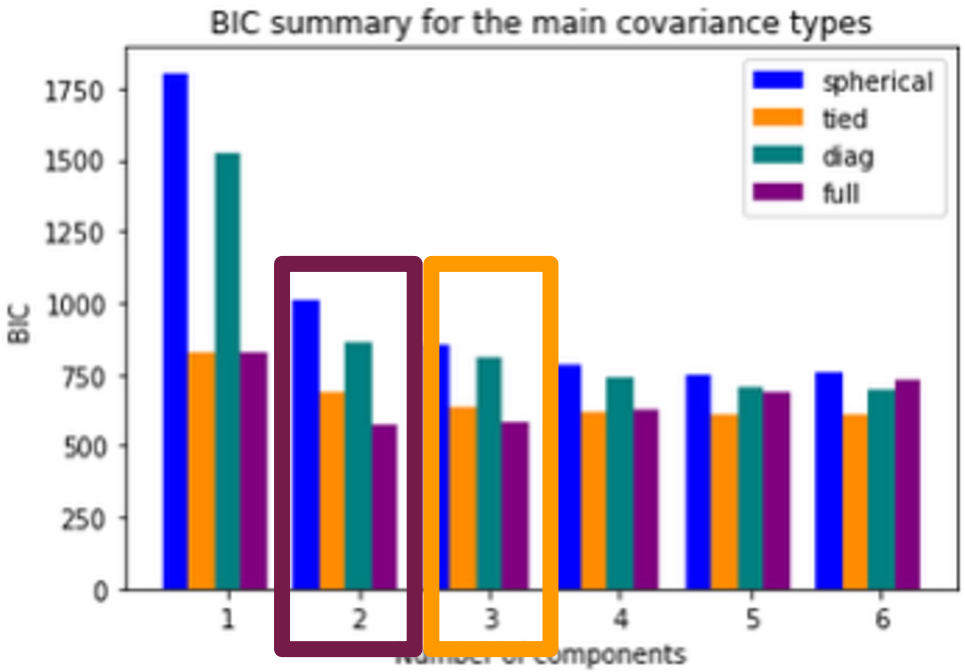
Spherical and diagonal covariance both seem to think that more components is always better (at least up to 6, anyway). This is unsurprising given the data: it's simple to check that the dimensions of the iris data are correlated.

Model selection in sklearn

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(inds
            widt
plt.xticks(inds
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.



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.

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.

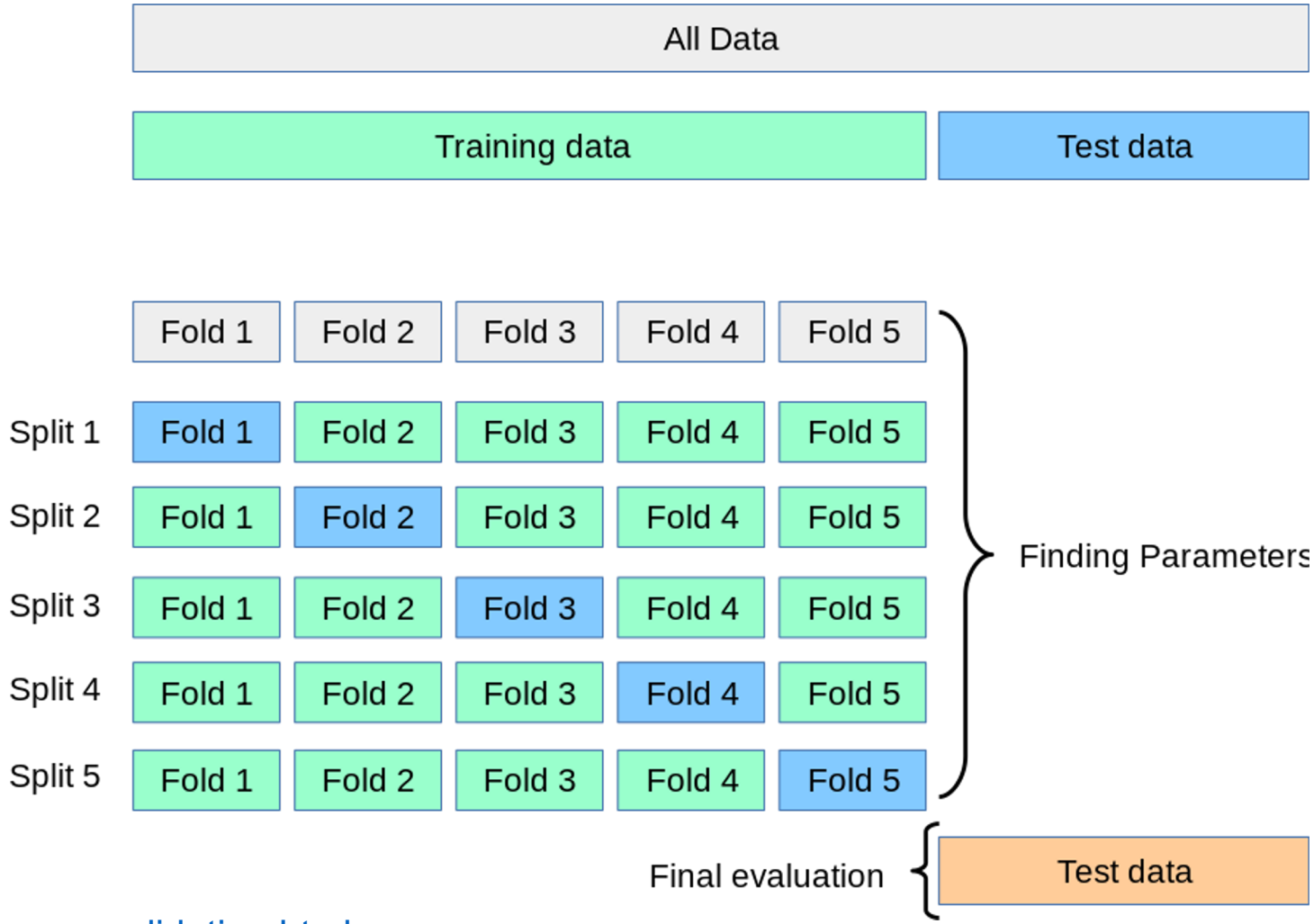


Image credit: https://scikit-learn.org/stable/modules/cross_validation.html

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.

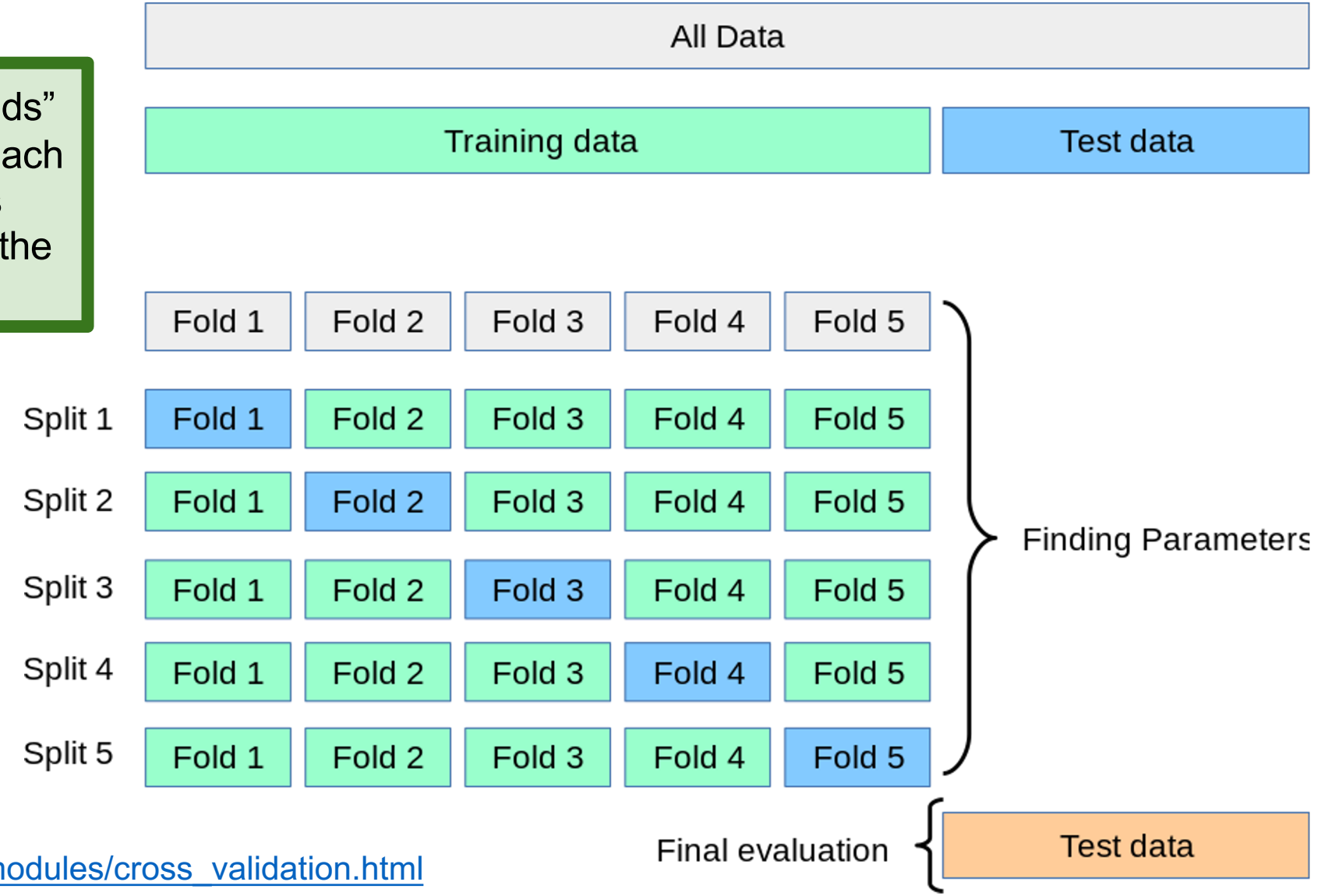


Image credit: https://scikit-learn.org/stable/modules/cross_validation.html

Cross-validation in `sklearn`

```
1  from sklearn.model_selection import cross_val_score
2  (n_samp, dim, k) = (200, 500, 10)
3  beta = np.zeros(dim)
4  inds = np.random.choice(np.arange(dim), size=k, replace=False)
5  beta[inds] = 5*np.random.randn(k)
6  X = np.random.randn(n_samp, dim)
7  y = np.dot(X, beta) + 0.1*np.random.randn(n_samp)
8
9  lasso = Lasso(alpha=5)
10 scores = cross_val_score(lasso, X, y, cv=10)
11 scores
```

Generating sparse data just like before.

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

Cross-validation in `sklearn`

```
1  from sklearn.model_selection import cross_val_score
2  (n_samp, dim, k) = (200, 500, 10)
3  beta = np.zeros(dim)
4  inds = np.random.choice(np.arange(dim), size=k, replace=False)
5  beta[inds] = 5*np.random.randn(k)
6  X = np.random.randn(n_samp, dim)
7  y = np.dot(X, beta) + 0.1*np.random.randn(n_samp)
8
9  lasso = Lasso(alpha=5)
10 scores = cross_val_score(lasso, X, y, cv=10)
11 scores
```

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

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

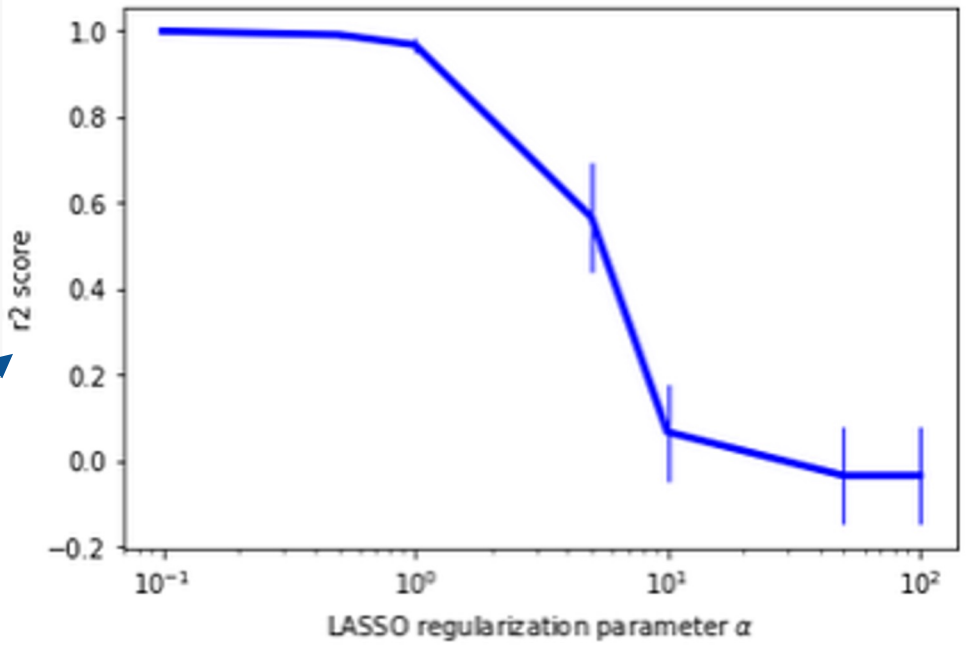
`cross_val_score` performs `cv` splits. On each split, we hold out one fold, train on the rest, and evaluate on the held-out fold. `cross_val_score` returns an array of the scores obtained in this way.

Cross-validation in `sklearn`

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

```
1  from sklearn.model_selection import cross_val_score
2  alphavals = np.array([0.1, 0.5, 1.0, 5, 10.0, 50, 100])
3  mean_scores = np.zeros(alphavals.shape)
4  sd_scores = np.zeros(alphavals.shape)
5
6  for i in range(len(alphavals)):
7      lasso = Lasso(alpha=alphavals[i])
8      scores = cross_val_score(lasso, X, y, cv=10)
9      mean_scores[i] = np.mean(scores)
10     sd_scores[i] = np.std(scores)
11
12 plt.errorbar(alphavals, mean_scores, yerr=2*sd_scores,
13                color='blue', linewidth=3, elinewidth=1)
14 plt.xscale('log'); plt.ylabel('r2 score')
15 _=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.

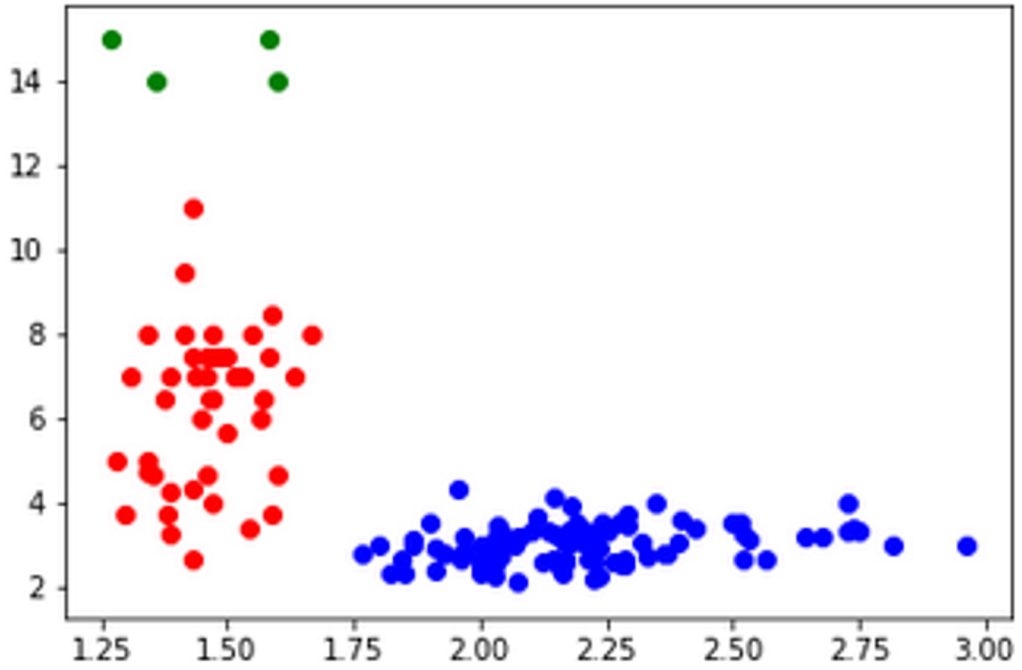


The `score` method of the `Lasso` model is the `r2score`, which we saw a few slides ago.

Assessing Models: `sklearn.metrics`

```
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.

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

```
0.507523474732037
```

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

Model persistence: pickling model objects

```
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)
```

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.).

```
import pickle
filehandler = open("model.pkl", 'wb')
pickle.dump(lasso, filehandler)
```

```
filehandler2 = open('model.pkl', 'rb')
lasso2 = pickle.load(filehandler2)
```

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

Other things

HW5 out

Coming next:

Matplotlib/ Seaborn / Intro to Pandas.