COMP7035

Python for Data Analytics and Artificial Intelligence

Scikit-learn

Renjie Wan, Jun Qi

11/19/24

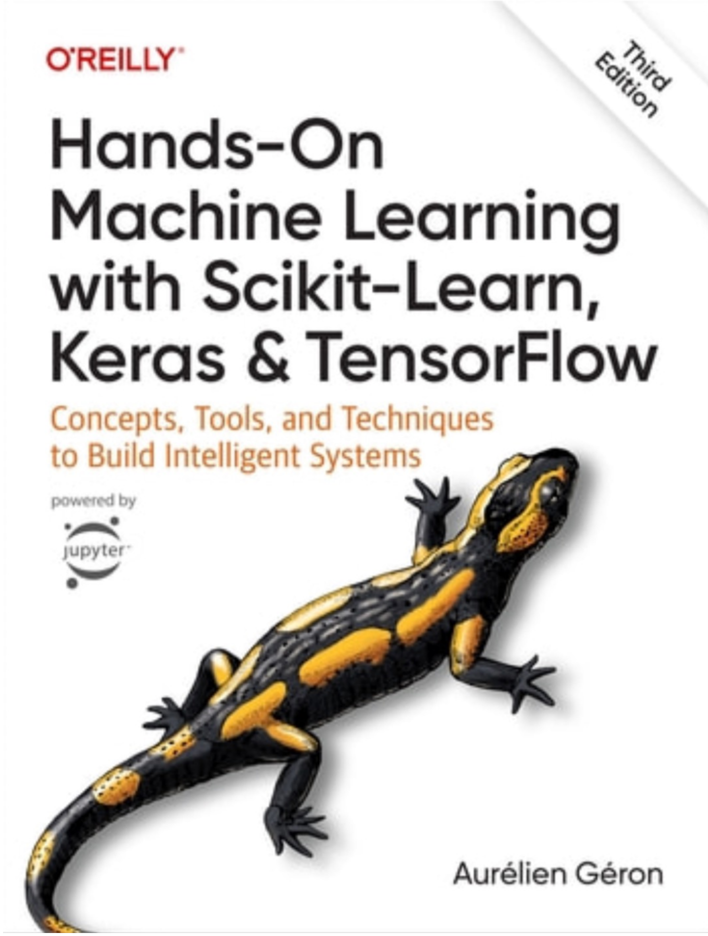
香港浸會大學
HONG KONG BAPTIST UNIVERSITY

DEPARTMENT OF
COMPUTER SCIENCE
計算機科學系

What Will We Learn?

Topic		**Hours**
I.	Python Fundamentals A. Program control and logic B. Data types and structures C. Function D. File I/O	12
II.	Numerical Computing and Data Visualization Tools and libraries such as A. NumPy B. Matplotlib C. Seaborn	9
III.	Exploratory Data Analysis (EDA) with Python Tools and libraries such as A. Pandas B. Sweetviz	9
IV.	Artificial Intelligence and Machine Learning with Python Tools and libraries such as **A. Scikit-learn** B. Keras	9

Hands-on Deep Learning and Machine Learning



Outline

1. **About Scikit-learn**
2. Data and Feature Processing
3. Unsupervised Learning
4. Supervised Learning

What is Scikit-learn?

- Open source data mining and analysis library in Python
- Classification, regression, and clustering algorithms
- First released in 2007
- Mostly written in Python:
- Built using NumPy and SciPy
- Integrates with matplotlib and plotly for plotting

What is Scikit-learn?

Popular

- Ranked as one of top machine learning projects on GitHub

Simple

- Fewer dependencies, makes for easier distribution

Using Scikit-learn

```
!pip install scikit-learn
```

```
import sklearn
```

Outline

1. About Scikit-learn
2. **Data and Feature Processing**
3. Unsupervised Learning
4. Supervised Learning

Dataset Splitting

- As we have introduced in the Keras part, we need to split the dataset into training, validation and testing datasets.
- Scikit-learn provides a simple API to achieve this goal.

```
sklearn.model_selection.train_test_split(*arrays, test_size=None, train_size=None, random_state=None, shuffle=True, stratify=None)
```

test_size should be between 0.0 and 1.0 and represent the proportion of the dataset to include in the test split.

Dataset Splitting

- As we have introduced in the Keras part, we need to split the dataset into training, validation and testing datasets.
- Scikit-learn provides a simple API to achieve this goal.

```
from sklearn.model_selection import train_test_split
import numpy as np
N = 1000
X = np.random.randn(N,128)
y = np.random.randn(N,1)
X_train, X_test, y_train, y_test = train_test_split(X,y,test_size=0.3)
for i in [X_train, X_test, y_train, y_test]:
  print(i.shape)
```

Standardization

- Standardization of datasets is a common requirement for many machine learning estimators
- Make each feature dimension of the dataset to be **zero mean and unit variance.**
- The [preprocessing](#) module provides the [StandardScaler](#) utility class

```
from sklearn import preprocessing
import numpy as np
X_train = np.array([[ 1., -1.,  2.],[ 2.,  0.,  0.],[ 0.,  1., -1.]])
print(X_train.mean(axis=0))
print(X_train.std(axis=0))
scaler = preprocessing.StandardScaler().fit(X_train)
X_scaled = scaler.transform(X_train)
print(X_scaled.mean(axis=0))
print(X_scaled.std(axis=0))
```

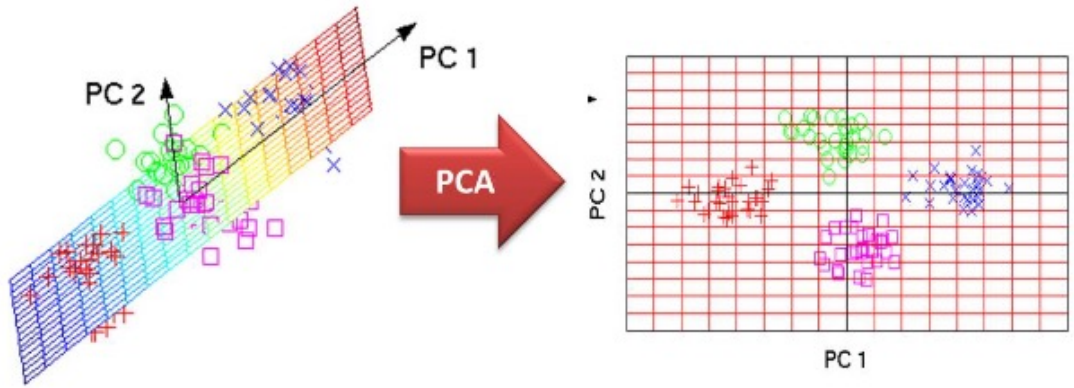
Normalization

- Normalization is the process of scaling individual samples to have unit norm.

```
from sklearn import preprocessing
import numpy as np
X_train = np.array([[ 1., -1.,  2.],[ 2.,  0.,  0.],[ 0.,  1., -1.]])
print(X_train.mean(axis=0))
print(X_train.std(axis=0))
X_normalized = preprocessing.normalize(X_train, norm='l2')
print(X_normalized)
print(np.sum(X_normalized*X_normalized,axis=1))
```

Principle Component Analysis (PCA)

- Linear dimensionality reduction using Singular Value Decomposition of the data to project it to a lower dimensional space.



Principle Component Analysis (PCA)

- Linear dimensionality reduction using Singular Value Decomposition of the data to project it to a lower dimensional space.

```
import numpy as np
from sklearn.decomposition import PCA
X = np.random.randn(5,20)
pca = PCA(n_components=2)
pca.fit(X)
Y = pca.transform(X)
print(Y)
```

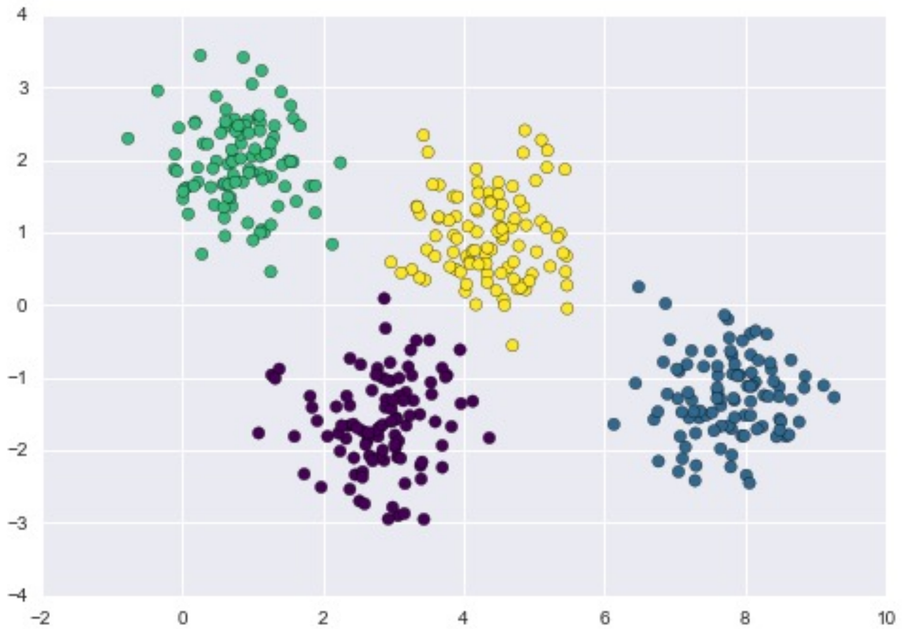
Define operator, fit, transform

Outline

1. About Scikit-learn
2. Data and Feature Processing
3. **Unsupervised Learning**
4. Supervised Learning

K-means Clustering

- The K-Means algorithm clusters data by trying to separate samples in n groups



K-means Clustering

- The K-Means algorithm clusters data by trying to separate samples in n groups

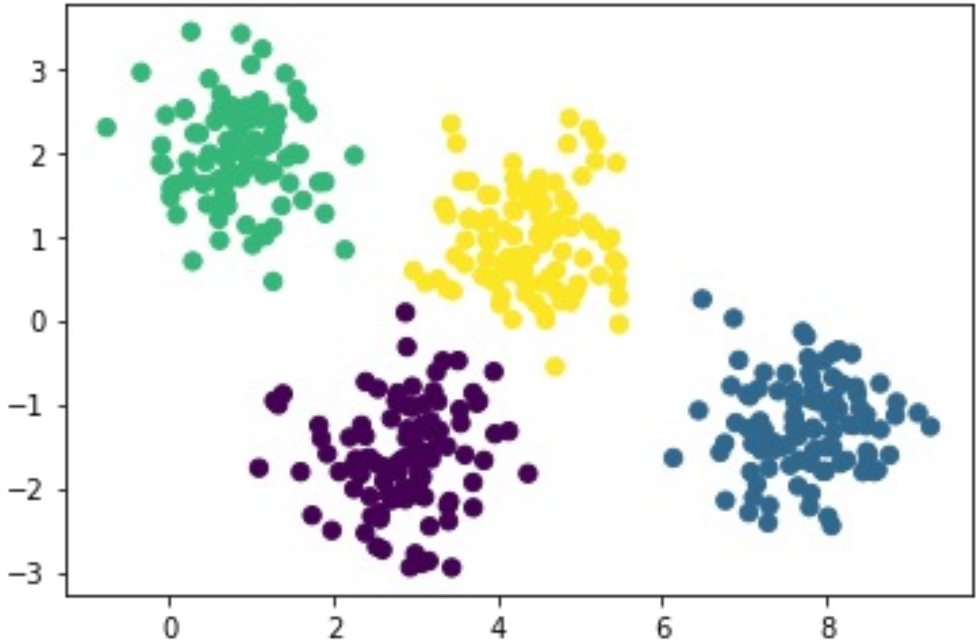
```
# Generate some data
from sklearn.datasets import make_blobs
import matplotlib.pyplot as plt
from sklearn.cluster import KMeans

X, y_true = make_blobs(n_samples=400, centers=4, cluster_std=0.60, random_state
=0)
print(X.shape)
X = X[:, ::-1] # flip axes for better plotting
# Plot the data with K Means Labels
kmeans = KMeans(4) # 4 is the number of clusters
kmeans.fit(X)
labels = kmeans.predict(X)
plt.scatter(X[:, 0], X[:, 1], c=labels, s=40, cmap='viridis')
```

Define algorithm, fit, predict

K-means Clustering

- The K-Means algorithm clusters data by trying to separate samples in n groups



K-means Clustering

- The K-Means algorithm clusters data by trying to separate samples in n groups

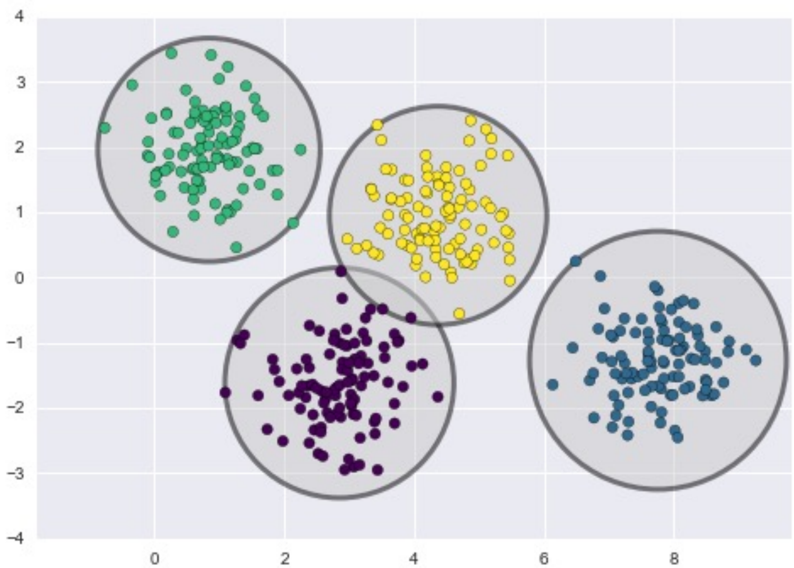
```
# Generate some data
from sklearn.datasets import make_blobs
import matplotlib.pyplot as plt
from sklearn.cluster import KMeans

X, y_true = make_blobs(n_samples=400, centers=4, cluster_std=0.60, random_state
=0)
print(X.shape)
X = X[:, ::-1] # flip axes for better plotting
plt.scatter(X[:, 0], X[:, 1], c=y_true, s=40, cmap='viridis')
```

Now we compare the prediction with the ground truth

Gaussian Mixture Model (GMM)

- An important observation for k-means is that these cluster models must be circular



Gaussian Mixture Model (GMM)

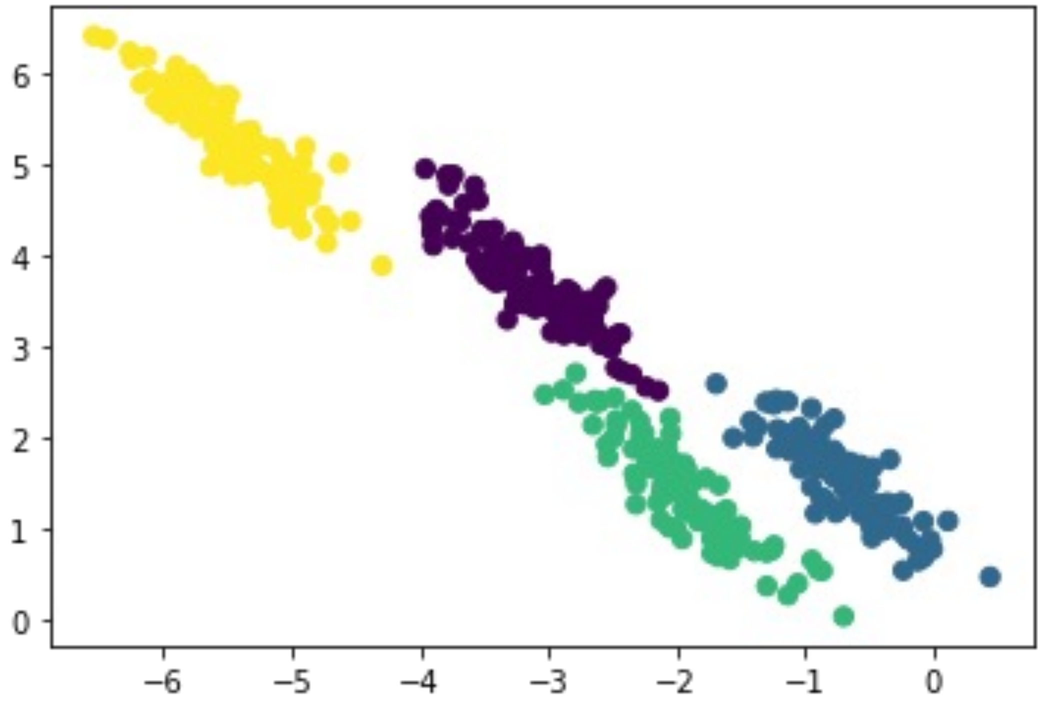
- Now we generate some "stretched" data by applying a linear transformation

```
from sklearn.datasets import make_blobs
import matplotlib.pyplot as plt
from sklearn.cluster import KMeans

X, y_true = make_blobs(n_samples=400, centers=4, cluster_std=0.60, random_s
tate=0)
X = X[:, ::-1]
rng = np.random.RandomState(13)
X_stretched = np.dot(X, rng.randn(2, 2))
plt.scatter(X_stretched[:, 0], X_stretched[:, 1], c=y_true, s=40, cmap='vir
idis')
```

Gaussian Mixture Model (GMM)

- Now we generate some "stretched" data by applying a linear transformation



Gaussian Mixture Model (GMM)

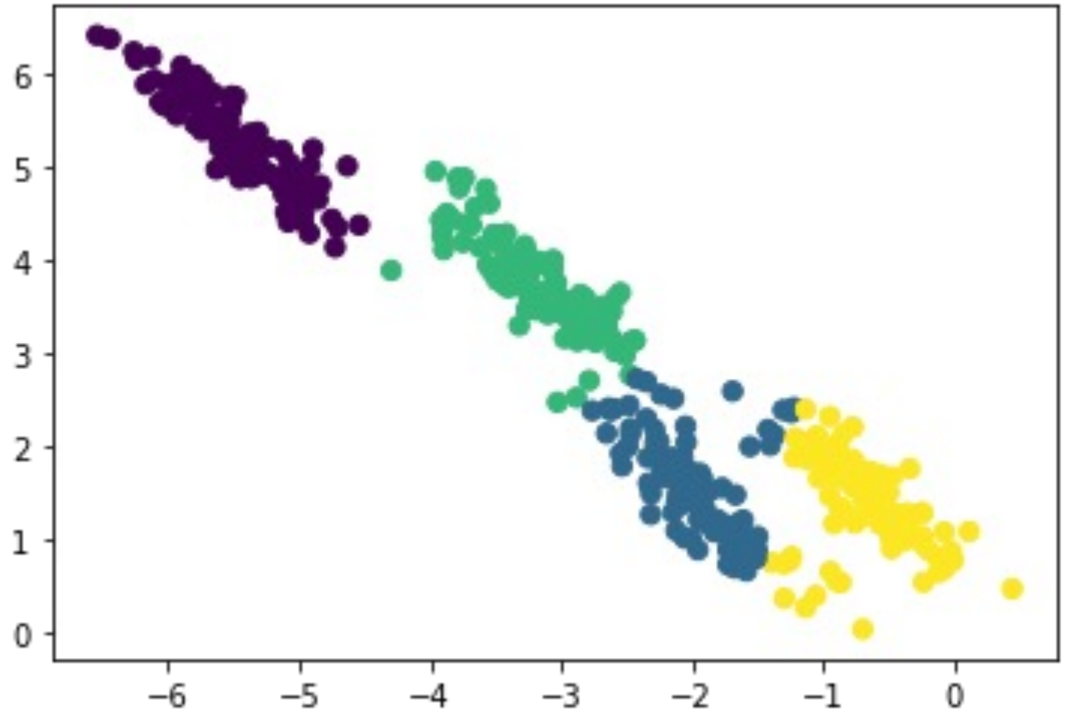
- Let us predict the labels using K-Means

```
from sklearn.datasets import make_blobs
import matplotlib.pyplot as plt
from sklearn.cluster import KMeans
import numpy as np

X, y_true = make_blobs(n_samples=400, centers=4, cluster_std=0.60, rando
m_state=0)
X = X[:, ::-1]
rng = np.random.RandomState(13)
X_stretched = np.dot(X, rng.randn(2, 2))
print(X_stretched.shape)
kmeans = KMeans(4)
kmeans.fit(X_stretched)
labels = kmeans.predict(X_stretched)
plt.scatter(X_stretched[:, 0], X_stretched[:, 1], c=labels, s=40, cmap='
viridis')
```

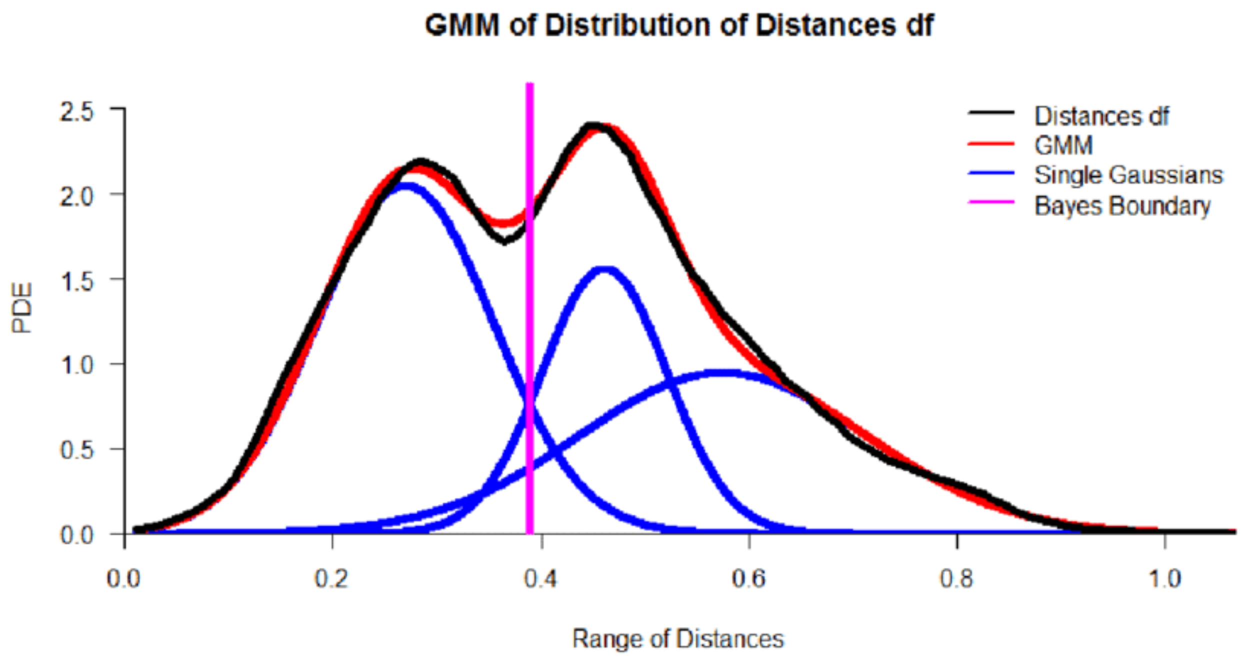
Gaussian Mixture Model (GMM)

- Let us predict the labels using K-Means



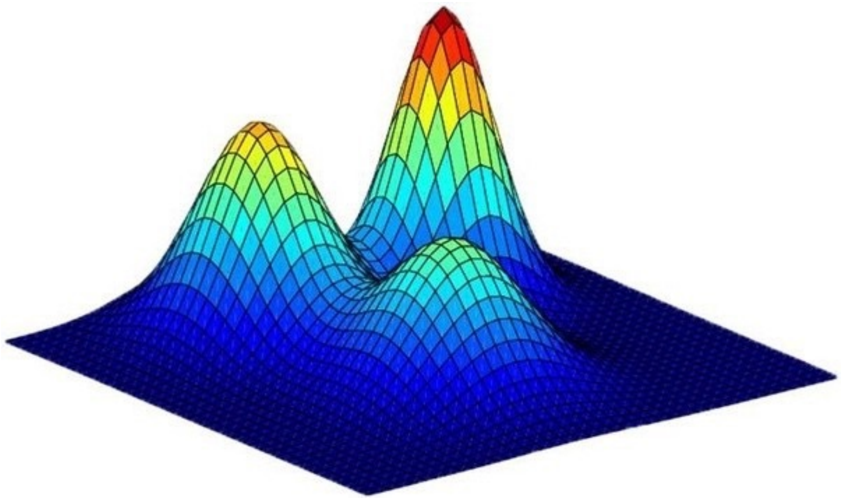
Gaussian Mixture Model (GMM)

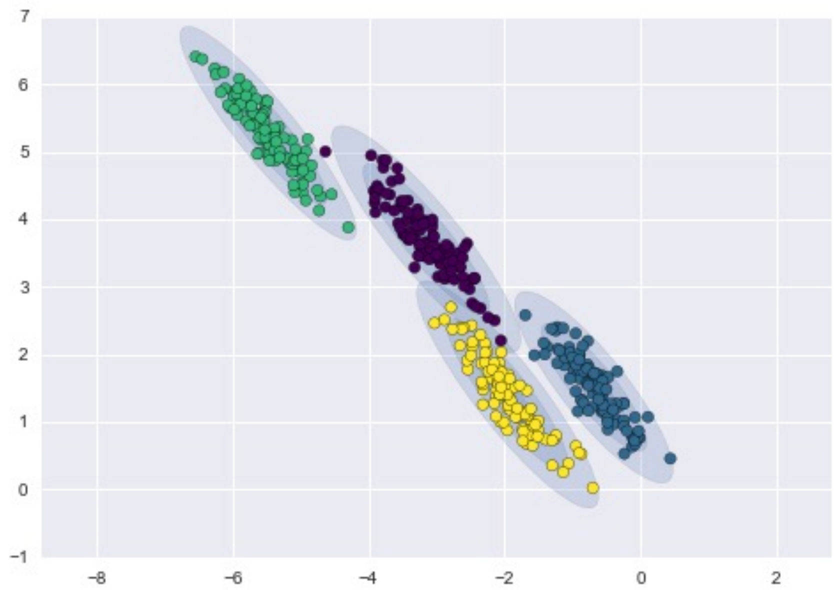
- A GMM attempts to find a mixture of multi-dimensional Gaussian probability distributions that best model any input dataset.



Gaussian Mixture Model (GMM)

- A GMM attempts to find a mixture of multi-dimensional Gaussian probability distributions that best model any input dataset.





We aim to express the whole data with 4 Gaussian distributions

Gaussian Mixture Model (GMM)

- Let us predict the labels using GMM

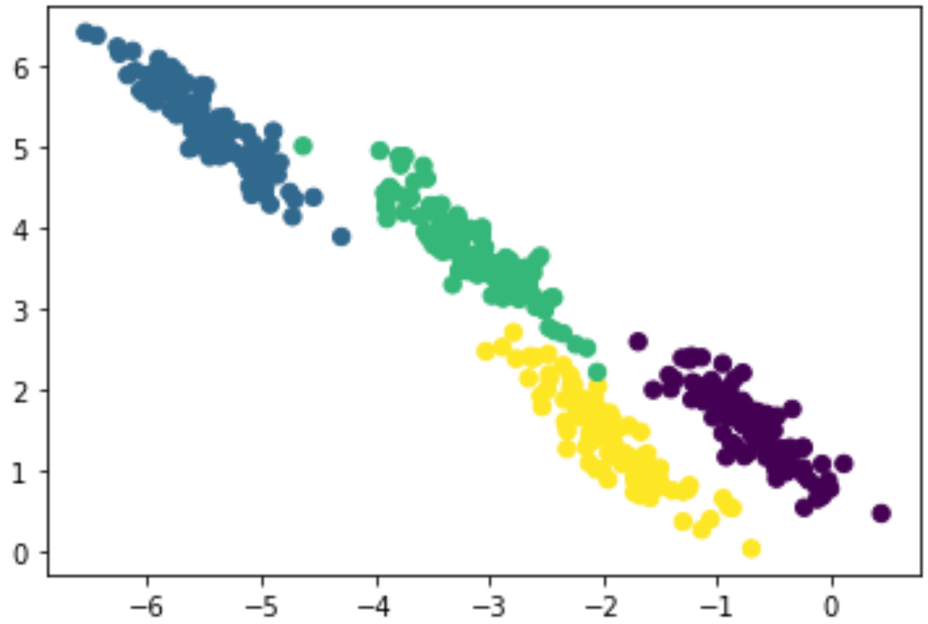
```
from sklearn.datasets import make_blobs
import matplotlib.pyplot as plt
import numpy as np
from sklearn.mixture import GaussianMixture

X, y_true = make_blobs(n_samples=400, centers=4, cluster_std=0.60, random_state=0)
X = X[:, ::-1]
rng = np.random.RandomState(13)
X_stretched = np.dot(X, rng.randn(2, 2))
gmm = GaussianMixture(n_components=4).fit(X_stretched)
labels = gmm.predict(X_stretched)
plt.scatter(X_stretched[:, 0], X_stretched[:, 1], c=label
```

Define algorithm, fit, predict

Gaussian Mixture Model (GMM)

- Let us predict the labels using GMM

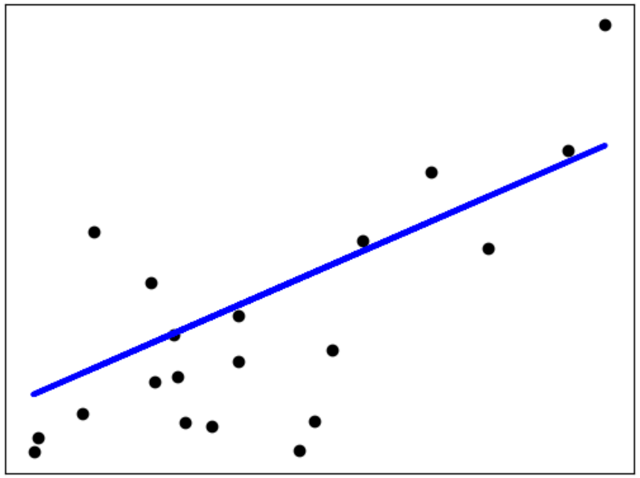


Outline

1. About Scikit-learn
2. Data and Feature Processing
3. Unsupervised Learning
4. **Supervised Learning**

Linear Models

- Ordinary Least Squares



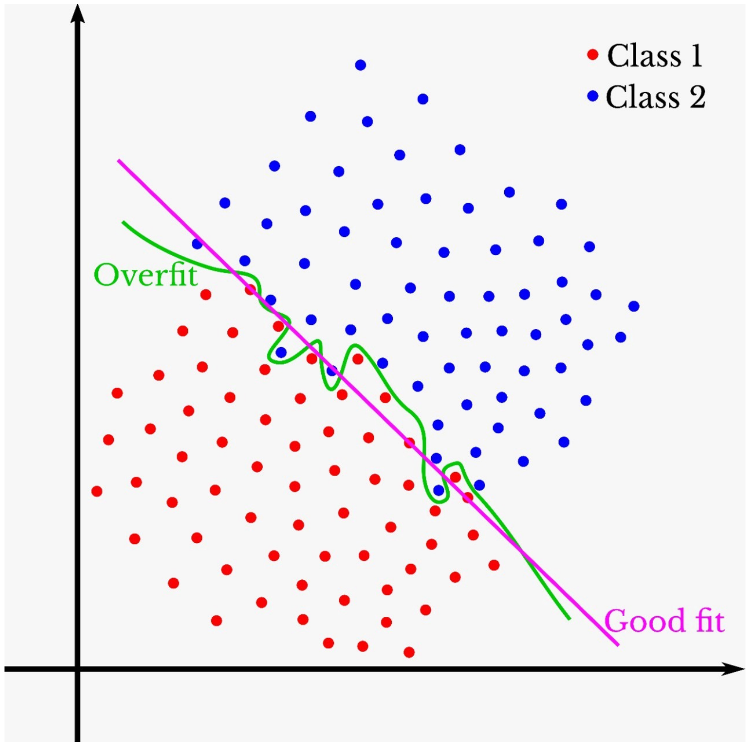
$$\min_{w} ||Xw - y||_2^2$$

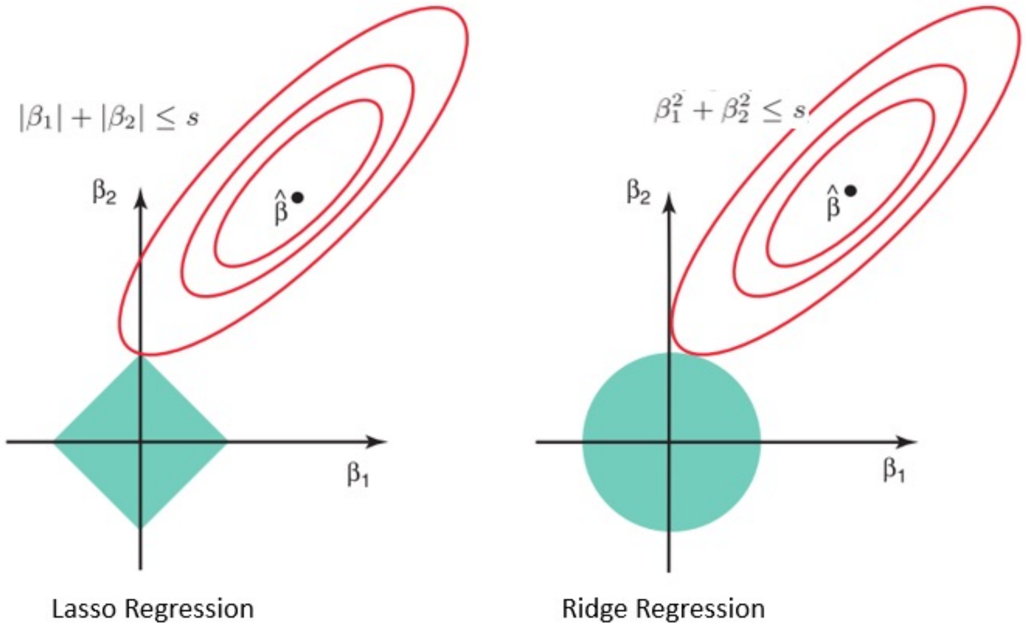
```
from sklearn import linear_model
reg = linear_model.LinearRegression()
x= [[0, 0], [1, 1], [2, 2]]
y = [0, 1, 2]
reg.fit(x,y)
y_pred = reg.predict(x)
print(y_pred)
```

```
[1.11022302e-16 1.00000000e+00 2.00000000e+00]
```

Linear Models

- To avoid overfitting, regularization is used.

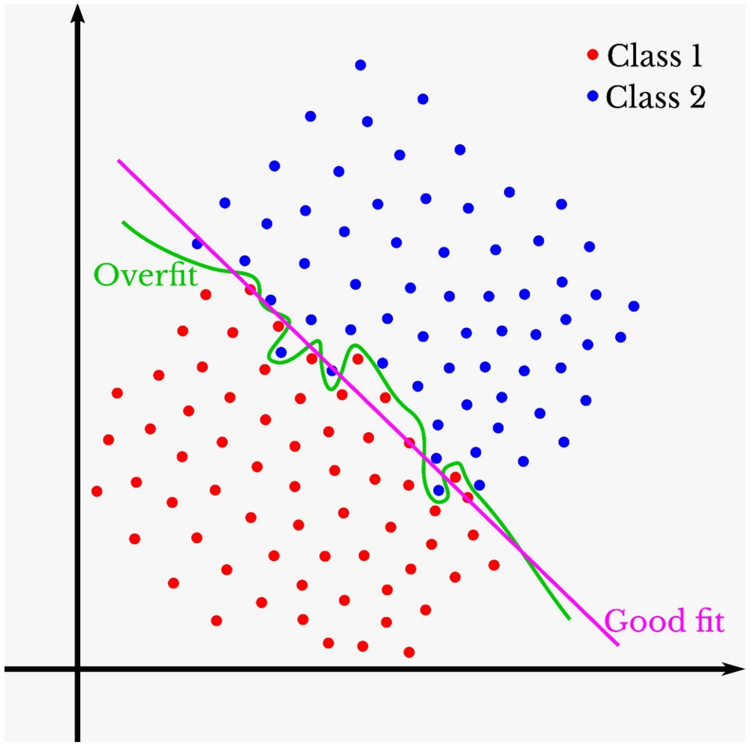


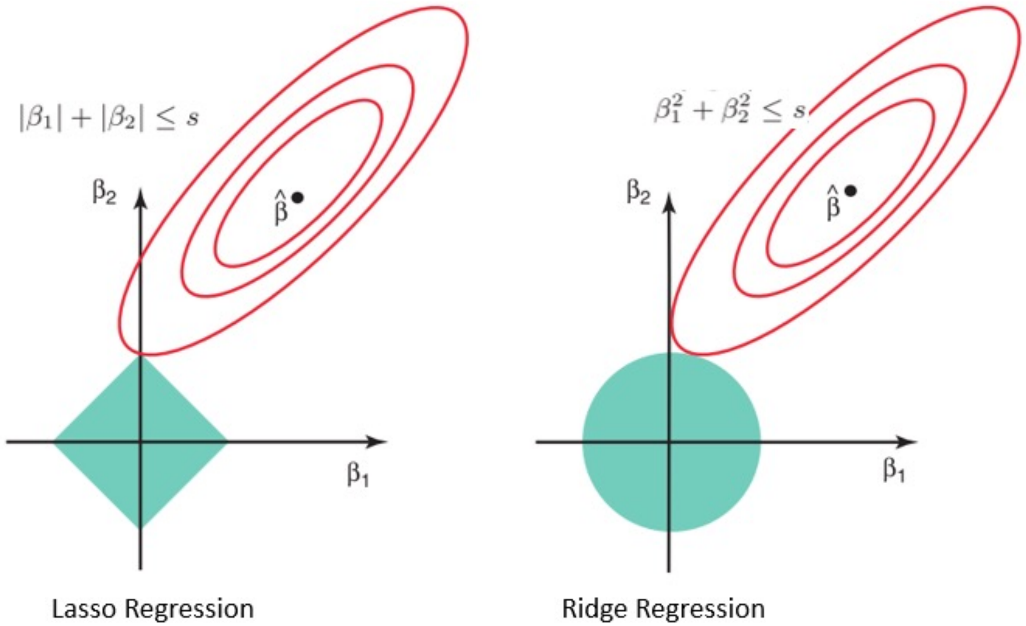


The overfitting happens when too much model parameters are used to fit simple data

Linear Models

- To avoid overfitting, regularization is used.





The regularization is used to constrain the number of active parameters

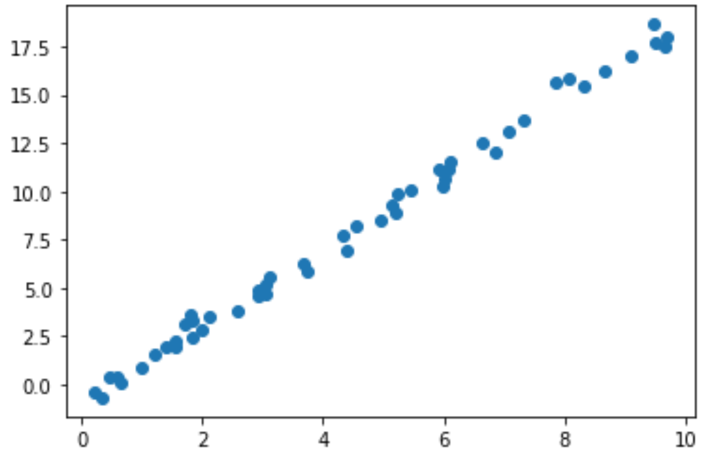
Linear Models

$$\min_{w} ||Xw - y||_2^2 + \boxed{\alpha ||w||_2^2}$$

- Ridge regression
- Now let us first try using `LinearRegression` to fit the noisy data using the over-complicated model (20 parameters)

```
import numpy as np
from sklearn import linear_model
import matplotlib.pyplot as plt

reg = linear_model.LinearRegression()
rng = np.random.RandomState(42)
x = 10 * rng.rand(50,1)
y = 2 * x - 1 + rng.randn(50,1)
x_train = x
for ii in range(19):# add unnecessary dimensions
  z = rng.rand(50,1)
  x_train = np.concatenate((x_train,z),axis=1)
reg.fit(x_train,y) #using 20 parameters
y_pred = reg.predict(x_train)
plt.scatter(x, y_pred);
print(reg.coef_.shape)
```



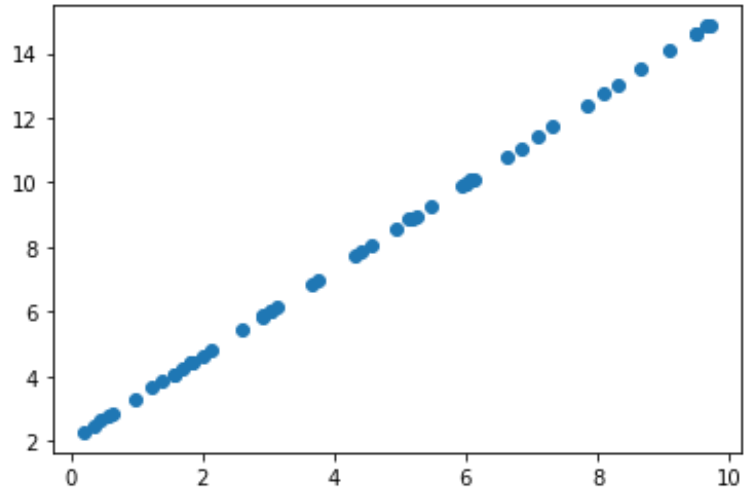
Linear Models

$$\min_{w} ||Xw - y||_2^2 + \boxed{\alpha||w||_2^2}$$

- Ridge regression
- Now let us try using `Ridge` to fit the noisy data using the over-complicated model (20 parameters), but with regularizations

```
import numpy as np
from sklearn import linear_model
import matplotlib.pyplot as plt

reg = linear_model.Ridge(alpha=200)
rng = np.random.RandomState(42)
x = 10 * rng.rand(50,1)
y = 2 * x - 1 + rng.randn(50,1)
x_train = x
for ii in range(19):# add unnecessary dimensions
  z = rng.rand(50,1)
  x_train = np.concatenate((x_train,z),axis=1)
reg.fit(x_train,y) #using 20 parameters
y_pred = reg.predict(x_train)
plt.scatter(x, y_pred);
print(reg.coef_.shape)
```



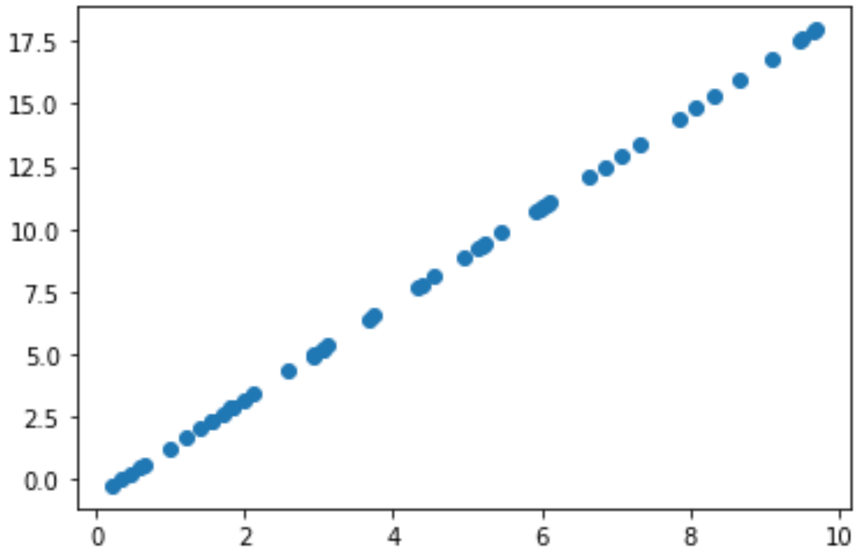
Linear Models

$$\min_{w} \frac{1}{2n_{\text{samples}}} ||Xw - y||_2^2 + \boxed{\alpha ||w||_1}$$

- Lasso regression
- Now let us try using Lasso to fit the noisy data using the over-complicated model (20 parameters), but with regularizations

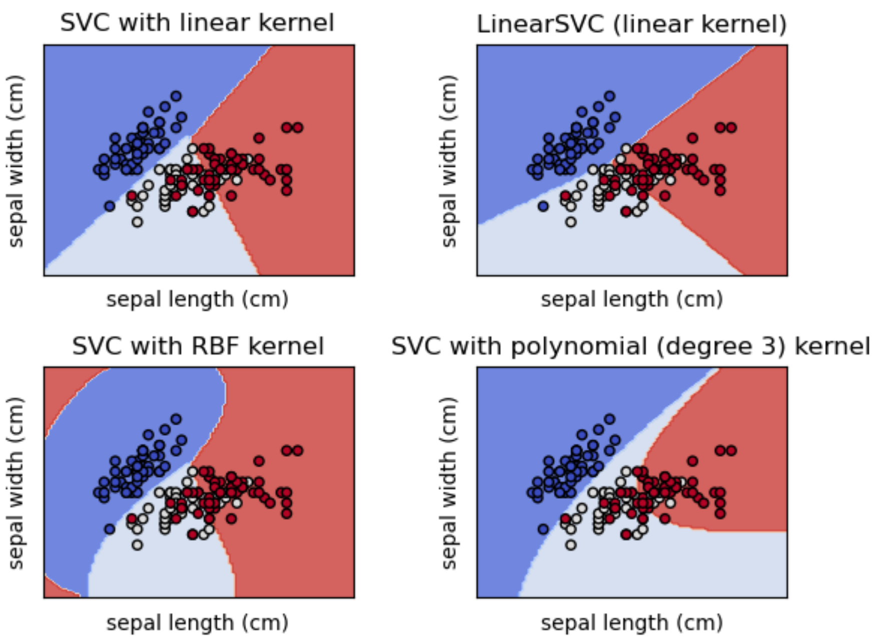
```
import numpy as np
from sklearn import linear_model
import matplotlib.pyplot as plt

reg = linear_model.Lasso(alpha=0.5)
rng = np.random.RandomState(42)
x = 10 * rng.rand(50,1)
y = 2 * x - 1 + rng.randn(50,1)
x_train = x
for ii in range(19):# add unnecessary dimensions
  z = rng.rand(50,1)
  x_train = np.concatenate((x_train,z),axis=1)
reg.fit(x_train,y) #using 20 parameters
y_pred = reg.predict(x_train)
plt.scatter(x, y_pred);
print(reg.coef_.shape)
```



Support Vector Machine (SVM)

- **Support vector machine (SVM)** is one of the most popular classifiers besides DNN.
- It maps the data into high dimensions using kernels, and designs classifiers in the high-dimension space



Support Vector Machine (SVM)

- **Support vector machine (SVM)** is one of the most popular classifiers besides DNN.

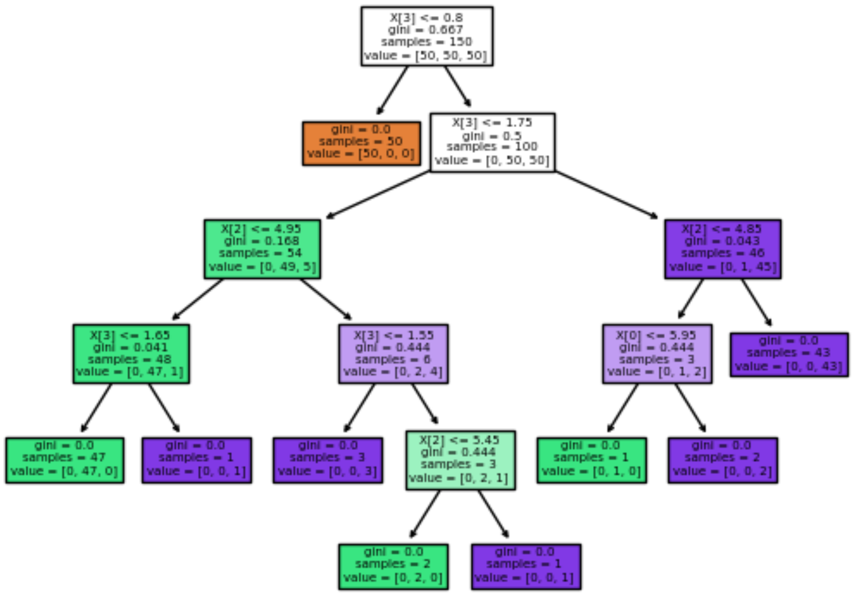
```
from sklearn import svm, datasets
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score

# import some data to play with
iris = datasets.load_iris()
X_iris = iris.data
y_iris = iris.target
Xtrain, Xtest, ytrain, ytest = train_test_split(X_iris, y_iris,test_si
ze=0.1)
model = svm.SVC(kernel="rbf")
model.fit(Xtrain, ytrain)
y_pred = model.predict(Xtest)
print(y_pred[:10])
print(ytest[:10])
print(accuracy_score(y_pred,ytest))
```

Decision Tree

- The **decision tree** predicts the value of a target variable by learning simple decision rules inferred from the data features.

Decision tree trained on all the iris features



Decision Tree

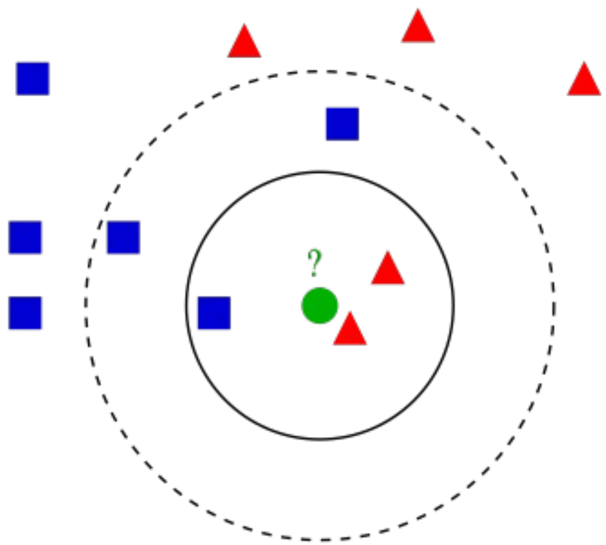
- The **decision tree** predicts the value of a target variable by learning simple decision rules inferred from the data features.

```
from sklearn import datasets
from sklearn.tree import DecisionTreeClassifier
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score

# import some data to play with
iris = datasets.load_iris()
X_iris = iris.data
y_iris = iris.target
Xtrain, Xtest, ytrain, ytest = train_test_split(X_iris, y_iris,test_size=0.1)
model = DecisionTreeClassifier()
model.fit(Xtrain, ytrain)
y_pred = model.predict(Xtest)
print(y_pred[:10])
print(ytest[:10])
print(accuracy_score(y_pred,ytest))
```

KNeighborsClassifier

- Majority voting based on the nearest K neighbors



KNeighborsClassifier

- Majority voting based on the nearest K neighbors

```
from sklearn import datasets
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score
from sklearn.neighbors import KNeighborsClassifier

# import some data to play with
iris = datasets.load_iris()
X_iris = iris.data
y_iris = iris.target
Xtrain, Xtest, ytrain, ytest = train_test_split(X_iris, y_iris,test_size
=0.1)
model = KNeighborsClassifier()
model.fit(Xtrain, ytrain)
y_pred = model.predict(Xtest)
print(y_pred[:10])
print(ytest[:10])
print(accuracy_score(y_pred,ytest))
```

Exercise

1. Split the Digits Dataset (load_digits) into training and testing sets with ratio 9:1
2. Standardize the data for each dimension
3. Reduce the dimension to 32 using PCA
4. Train a SVM classifier