Machine Learning on the Iris dataset (classification model)

K-nearest Neighbors (KNN)

Logistic regression

In [1]:

```
# Import and load Iris dataset directly from sklearn.datasets
from sklearn.datasets import load_iris
```

In [2]:

```
# Instance iris dataset as an object (bunch)
iris = load_iris()
```

In [3]:

```
type(iris)
```

Out[3]:

```
sklearn.utils.Bunch
```

In [4]:

```
# Print the iris data
print (iris.data)
```

```
[[5.1 3.5 1.4 0.2]
 [4.9 3.  1.4 0.2]
 [4.7 3.2 1.3 0.2]
 [4.6 3.1 1.5 0.2]
 [5.  3.6 1.4 0.2]
 [5.4 3.9 1.7 0.4]
 [4.6 3.4 1.4 0.3]
 [5.  3.4 1.5 0.2]
 [4.4 2.9 1.4 0.2]
 [4.9 3.1 1.5 0.1]
 [5.4 3.7 1.5 0.2]
 [4.8 3.4 1.6 0.2]
 [4.8 3.  1.4 0.1]
 [4.3 3.  1.1 0.1]
 [5.8 4.  1.2 0.2]
 [5.7 4.4 1.5 0.4]
 [5.4 3.9 1.3 0.4]
 [5.1 3.5 1.4 0.3]
 [5.7 3.8 1.7 0.3]
 [5.1 3.8 1.5 0.3]
 [5.4 3.4 1.7 0.2]
 [5.1 3.7 1.5 0.4]
 [4.6 3.6 1.  0.2]
 [5.1 3.3 1.7 0.5]
 [4.8 3.4 1.9 0.2]
 [5.  3.  1.6 0.2]
 [5.  3.4 1.6 0.4]
 [5.2 3.5 1.5 0.2]
 [5.2 3.4 1.4 0.2]
 [4.7 3.2 1.6 0.2]
 [4.8 3.1 1.6 0.2]
 [5.4 3.4 1.5 0.4]
 [5.2 4.1 1.5 0.1]
 [5.5 4.2 1.4 0.2]
 [4.9 3.1 1.5 0.1]
 [5.  3.2 1.2 0.2]
 [5.5 3.5 1.3 0.2]
 [4.9 3.1 1.5 0.1]
 [4.4 3.  1.3 0.2]
 [5.1 3.4 1.5 0.2]
 [5.  3.5 1.3 0.3]
 [4.5 2.3 1.3 0.3]
 [4.4 3.2 1.3 0.2]
 [5.  3.5 1.6 0.6]
 [5.1 3.8 1.9 0.4]
 [4.8 3.  1.4 0.3]
 [5.1 3.8 1.6 0.2]
 [4.6 3.2 1.4 0.2]
 [5.3 3.7 1.5 0.2]
 [5.  3.3 1.4 0.2]
 [7.  3.2 4.7 1.4]
 [6.4 3.2 4.5 1.5]
 [6.9 3.1 4.9 1.5]
 [5.5 2.3 4.  1.3]
 [6.5 2.8 4.6 1.5]
 [5.7 2.8 4.5 1.3]
 [6.3 3.3 4.7 1.6]
 [4.9 2.4 3.3 1. ]
 [6.6 2.9 4.6 1.3]
 [5.2 2.7 3.9 1.4]
 [5.  2.  3.5 1. ]
```

```
 [5.9 3.  4.2 1.5]
 [6.  2.2 4.  1. ]
 [6.1 2.9 4.7 1.4]
 [5.6 2.9 3.6 1.3]
 [6.7 3.1 4.4 1.4]
 [5.6 3.  4.5 1.5]
 [5.8 2.7 4.1 1. ]
 [6.2 2.2 4.5 1.5]
 [5.6 2.5 3.9 1.1]
 [5.9 3.2 4.8 1.8]
 [6.1 2.8 4.  1.3]
 [6.3 2.5 4.9 1.5]
 [6.1 2.8 4.7 1.2]
 [6.4 2.9 4.3 1.3]
 [6.6 3.  4.4 1.4]
 [6.8 2.8 4.8 1.4]
 [6.7 3.  5.  1.7]
 [6.  2.9 4.5 1.5]
 [5.7 2.6 3.5 1. ]
 [5.5 2.4 3.8 1.1]
 [5.5 2.4 3.7 1. ]
 [5.8 2.7 3.9 1.2]
 [6.  2.7 5.1 1.6]
 [5.4 3.  4.5 1.5]
 [6.  3.4 4.5 1.6]
 [6.7 3.1 4.7 1.5]
 [6.3 2.3 4.4 1.3]
 [5.6 3.  4.1 1.3]
 [5.5 2.5 4.  1.3]
 [5.5 2.6 4.4 1.2]
 [6.1 3.  4.6 1.4]
 [5.8 2.6 4.  1.2]
 [5.  2.3 3.3 1. ]
 [5.6 2.7 4.2 1.3]
 [5.7 3.  4.2 1.2]
 [5.7 2.9 4.2 1.3]
 [6.2 2.9 4.3 1.3]
 [5.1 2.5 3.  1.1]
 [5.7 2.8 4.1 1.3]
 [6.3 3.3 6.  2.5]
 [5.8 2.7 5.1 1.9]
 [7.1 3.  5.9 2.1]
 [6.3 2.9 5.6 1.8]
 [6.5 3.  5.8 2.2]
 [7.6 3.  6.6 2.1]
 [4.9 2.5 4.5 1.7]
 [7.3 2.9 6.3 1.8]
 [6.7 2.5 5.8 1.8]
 [7.2 3.6 6.1 2.5]
 [6.5 3.2 5.1 2. ]
 [6.4 2.7 5.3 1.9]
 [6.8 3.  5.5 2.1]
 [5.7 2.5 5.  2. ]
 [5.8 2.8 5.1 2.4]
 [6.4 3.2 5.3 2.3]
 [6.5 3.  5.5 1.8]
 [7.7 3.8 6.7 2.2]
 [7.7 2.6 6.9 2.3]
 [6.  2.2 5.  1.5]
 [6.9 3.2 5.7 2.3]
 [5.6 2.8 4.9 2. ]
```

```
 [7.7 2.8 6.7 2. ]
 [6.3 2.7 4.9 1.8]
 [6.7 3.3 5.7 2.1]
 [7.2 3.2 6.  1.8]
 [6.2 2.8 4.8 1.8]
 [6.1 3.  4.9 1.8]
 [6.4 2.8 5.6 2.1]
 [7.2 3.  5.8 1.6]
 [7.4 2.8 6.1 1.9]
 [7.9 3.8 6.4 2. ]
 [6.4 2.8 5.6 2.2]
 [6.3 2.8 5.1 1.5]
 [6.1 2.6 5.6 1.4]
 [7.7 3.  6.1 2.3]
 [6.3 3.4 5.6 2.4]
 [6.4 3.1 5.5 1.8]
 [6.  3.  4.8 1.8]
 [6.9 3.1 5.4 2.1]
 [6.7 3.1 5.6 2.4]
 [6.9 3.1 5.1 2.3]
 [5.8 2.7 5.1 1.9]
 [6.8 3.2 5.9 2.3]
 [6.7 3.3 5.7 2.5]
 [6.7 3.  5.2 2.3]
 [6.3 2.5 5.  1.9]
 [6.5 3.  5.2 2. ]
 [6.2 3.4 5.4 2.3]
 [5.9 3.  5.1 1.8]]
```

In [5]:

```
print ( iris.feature_names)
```

```
['sepal length (cm)', 'sepal width (cm)', 'petal length (cm)', 'petal wi
dth (cm)']
```

In [6]:

```
# Setosa (0), Versicolor (1), Virginica (2)
print (iris.target)
```

```
[0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0
 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
1
 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 2 2 2 2 2 2 2 2 2 2 2 2
2
 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2
2
 2 2]
```

In [7]:

```
# Setosa (0), Versicolor (1), Virginica (2)
print (iris.target_names)
```

```
['setosa' 'versicolor' 'virginica']
```

Requirements for working with data in scikit-learn

1. Features and responses are **separeted objects**
2. Features and response should be **numeric**
3. Features and response should be **NumPy arrays**
4. Features and response should have **specific shapes**

In [8]:

```
# Check the types of the features and response
print(("Type of the features: %s") % type(iris.data))
print(("Type of the response: %s") % type(iris.target))
```

```
Type of the features: <class 'numpy.ndarray'>
Type of the response: <class 'numpy.ndarray'>
```

In [9]:

```
# Check the shape of the features and response
print(iris.data.shape)
print(iris.target.shape)
```

```
(150, 4)
(150,)
```

In [10]:

```
# Store feature matrix in "X"
X = iris.data

# Store response vector in "y"
y = iris.target
```

In [11]:

```
print(X.shape)
print(y.shape)
```

```
(150, 4)
(150,)
```

Scikit-Learn 4-step Modelling Pattern

Step 1: Import the class you plan to use

In [12]:

```
from sklearn.neighbors import KNeighborsClassifier
```

Step 2: Make an Instance of the "Estimator"

In [13]:

```
knn = KNeighborsClassifier(n_neighbors=1)
# You can understand all parameters using the print command
print(knn)
```

```
KNeighborsClassifier(algorithm='auto', leaf_size=30, metric='minkowski',
           metric_params=None, n_jobs=1, n_neighbors=1, p=2,
           weights='uniform')
```

Step 3: Fit the model with data (aka "model training")

- Model is learning the relationship between X and Y
- Occurs in-place

In [14]:

```
knn.fit(X,y)
```

Out[14]:

```
KNeighborsClassifier(algorithm='auto', leaf_size=30, metric='minkowski',
           metric_params=None, n_jobs=1, n_neighbors=1, p=2,
           weights='uniform')
```

Step 4: Predict the response for a new observation

- New observations are called "out-of-sample" data
- Uses the information it learned during the model training process

In [15]:

```
knn.predict([[3, 5, 4, 2]])
```

Out[15]:

```
array([2])
```

- Can predict for multiple observations at once

In [16]:

```
X_new = [[3,5,4,2],[5,4,3,2]]
knn.predict(X_new)
```

Out[16]:

```
array([2, 1])
```

Using a different value for K

In [17]:

```
# Make an instance for the model using the value k = 5
knn = KNeighborsClassifier(n_neighbors=5)

# Fit the model with data
knn.fit(X,y)
```

Out[17]:

```
KNeighborsClassifier(algorithm='auto', leaf_size=30, metric='minkowski',
           metric_params=None, n_jobs=1, n_neighbors=5, p=2,
           weights='uniform')
```

In [18]:

```
# prodict the response for new observation
knn.predict(X_new)
```

Out[18]:

```
array([1, 1])
```

Using a different classification model

In [19]:

```
# import the class
from sklearn.linear_model import LogisticRegression

# Make an instance for the model (default parameters)
logreg = LogisticRegression()

# Fit the model with data
logreg.fit(X,y)

# Predict the response for new observations
logreg.predict(X_new)
```

Out[19]:

```
array([2, 0])
```

Problems with training and testing on the same data

- Goal is to estimate likely performance at a model on **out-of-sample data**
- But, maximizing training accuracy rewards **overly complex models** that won't necessarilly generalize
- Unnecessarilly complex models **overfit** the training data

Train / Test (split dataset)

1. Split the dataset into two pieces: **a training set** and **a test set**
2. Train the model on **the training set**
3. Test the model on **the testing set,** and evaluate how well we did.

In [20]:

```
# Step 1: split X and y into training and testing sets
from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.4,random_state
=4)
```

In [21]:

```
# Only to check the shape of the new objects
print(X_train.shape)
print(X_test.shape)
print(y_train.shape)
print(y_test.shape)
```

```
(90, 4)
(60, 4)
(90,)
(60,)
```

Using LogisticRegression (classification model)

In [22]:

```
# Step 2: train the model on the training set
logreg = LogisticRegression()
logreg.fit(X_train, y_train)
```

Out[22]:

```
LogisticRegression(C=1.0, class_weight=None, dual=False, fit_intercept=T
rue,
          intercept_scaling=1, max_iter=100, multi_class='ovr', n_jobs=
1,
          penalty='l2', random_state=None, solver='liblinear', tol=0.000
1,
          verbose=0, warm_start=False)
```

In [23]:

```
# Step 3: make predictions on the testing set

from sklearn.metrics import accuracy_score

y_pred = logreg.predict(X_test)

# Compare actual response values (y_test) with predicted response values (y_pred)
print(accuracy_score(y_test, y_pred))
```

```
0.95
```

Using KNN (with K = 1)

In [24]:

```
knn = KNeighborsClassifier(n_neighbors=1)

knn.fit(X_train,y_train)
y_pred = knn.predict(X_test)
print(accuracy_score(y_test, y_pred))
```

```
0.95
```

Using KNN (with K = 5)

In [25]:

```
knn = KNeighborsClassifier(n_neighbors=5)

knn.fit(X_train,y_train)
y_pred = knn.predict(X_test)
print(accuracy_score(y_test, y_pred))
```

```
0.9666666666666667
```

Can we locate an even better value for K

In [26]:

```
# Try k=1 through k=25 and record testing accuracy
k_range = range(1, 26)
scores = []
for k in k_range:
    knn = KNeighborsClassifier(n_neighbors=k)
    knn.fit(X_train, y_train)
    y_pred = knn.predict(X_test)
    scores.append(accuracy_score(y_test, y_pred))
```

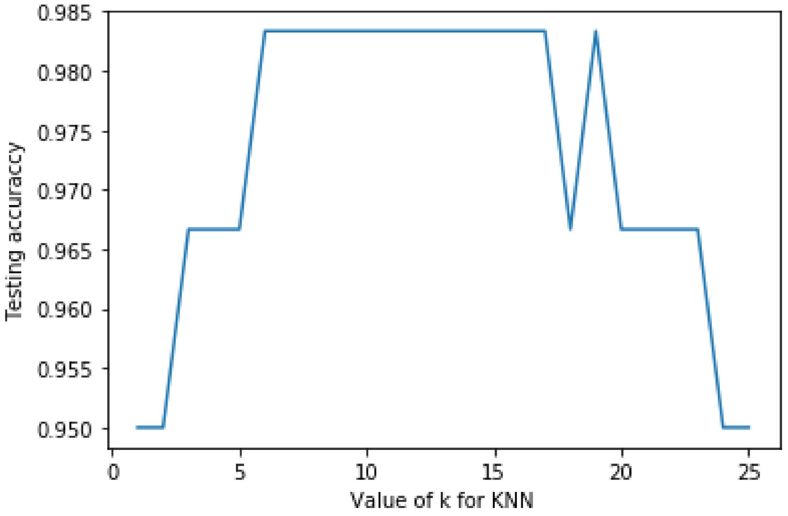
In [27]:

```
# import Matplotlib
import matplotlib.pyplot as plt
%matplotlib inline

# plot the relationship between K and testing accuracy
plt.plot(k_range, scores)
plt.xlabel('Value of k for KNN')
plt.ylabel('Testing accuraccy')
#plt.show()
```

Out[27]:

```
Text(0,0.5,'Testing accuraccy')
```



- Training accuracy rises as model complexity increases.
- Testing accuracy penalizes models that are too complex or not complex enough
- For KNN models, complexity is determined by the value of K (lower value = more complex)

Making predictions on out-of-sample data

In [30]:

```
# Make an instance for the model with the best known parameters
knn = KNeighborsClassifier(n_neighbors=11)

# Train the model with x and y (not X_train and y_train)
knn.fit(X, y)

# Make a prediction for an out-of-sample observation
knn.predict([[3, 5, 4, 2]])


# When we made this first prection, the result we get was array([2])
```

Out[30]:

```
array([1])
```

Downsides of train/test split?

- Provides a high-variance estimate of out-of-sample accuracy
- K-fold cross-validation overcomes this limitation
- But, train/test split is still useful because of its **flexibility and speed.**

In [ ]:

```

```