Exploring scikit-learn

Introducing the iris dataset

- 50 samples of 3 different species of iris (150 samples total)
- Measurements: sepal length, sepal width, petal length, petal width
- Predict the species of an iris using the measurements

Loading the iris dataset into scikit-learn

In [1]:
```
# import load_iris function from datasets module
from sklearn.datasets import load_iris
```

In [2]:
```
# save "bunch" object containing iris dataset and its attributes
iris = load_iris()
```

Machine Learning terminology

- Each row is an **observation** (also known as: sample, example, instance, record)
- Each column is a **feature** (also known as: predictor, attribute, independent variable, input, regressor, covariate)

In [6]:
```
# print the names of the four features
print(iris.feature_names)
```

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

In [7]:
```
# print integers representing the species of each observation
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 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]
```

```
In [8]: # print the encoding scheme for species: 0 = setosa, 1 = versicolor, 2 = virg
        inica
        print(iris.target_names)
```

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

- Each value we are predicting is the **response** (also known as: target, outcome, label, dependent variable)
- **Classification** is supervised learning in which the response is categorical
- **Regression** is supervised learning in which the response is ordered and continuous

Requirements for working with data in scikit-learn

1. Features and response are **separate objects**
2. Features should always be **numeric**, and response should be **numeric** for regression problems
3. Features and response should be **NumPy arrays**
4. Features and response should have **specific shapes**

```
In [9]: # check the types of the features and response
        print(type(iris.data))
        print(type(iris.target))
```

```
<class 'numpy.ndarray'>
<class 'numpy.ndarray'>
```

```
In [10]: # check the shape of the features (first dimension = number of observations,
         second dimensions = number of features)
         print(iris.data.shape)
```

```
(150, 4)
```

```
In [11]: # check the shape of the response (single dimension matching the number of ob
         servations)
         print(iris.target.shape)
```

```
(150,)
```

```
In [12]: # store feature matrix in "X"
         X = iris.data

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

Training a Machine Learning model with scikit-learn

- What is the **K-nearest neighbors** classification model?
- What are the four steps for **model training and prediction** in scikit-learn?
- How can I apply this pattern to **other Machine Learning models**?

K-nearest neighbors (KNN) classification

1. Pick a value for K.
2. Search for the K observations in the training data that are "nearest" to the measurements of the unknown iris.
3. Use the most popular response value from the K nearest neighbors as the predicted response value for the unknown iris.

Loading the data

In [3]:
```
# import load_iris function from datasets module
from sklearn.datasets import load_iris

# save "bunch" object containing iris dataset and its attributes
iris = load_iris()

# store feature matrix in "X"
X = iris.data

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

In [4]:
```
# print the shapes of X and y
print(X.shape)
print(y.shape)
```

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

scikit-learn 4-step modeling pattern

Step 1: Import the class you plan to use

In [5]:
```
from sklearn.neighbors import KNeighborsClassifier
```

Step 2: "Instantiate" the "estimator"

- "Estimator" is scikit-learn's term for model
- "Instantiate" means "make an instance of"

```
In [6]: knn = KNeighborsClassifier(n_neighbors=1)
```

- Name of the object does not matter
- Can specify tuning parameters (aka "hyperparameters") during this step
- All parameters not specified are set to their defaults

```
In [7]: print(knn)
```

```
KNeighborsClassifier(n_neighbors=1)
```

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

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

```
In [8]: knn.fit(X, y)
```

```
Out[8]: KNeighborsClassifier(n_neighbors=1)
```

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

```
Out[9]: array([2])
```

- Returns a NumPy array
- Can predict for multiple observations at once

```
In [10]: X_new = [[3, 5, 4, 2], [5, 4, 3, 2]]
         knn.predict(X_new)
```

```
Out[10]: array([2, 1])
```

Using a different value for K

In [11]:
```
# instantiate the model (using the value K=5)
knn = KNeighborsClassifier(n_neighbors=5)

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

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

Out[11]:
```
array([1, 1])
```

Using a different classification model

In [12]:
```
# import the class
from sklearn.linear_model import LogisticRegression

# instantiate the model
logreg = LogisticRegression(solver='liblinear')

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

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

Out[12]:
```
array([2, 0])
```

Comparing Machine Learning models in scikit-learn

- How do I choose **which model to use** for my supervised learning task?
- How do I choose the **best tuning parameters** for that model?
- How do I estimate the **likely performance of my model** on out-of-sample data?

Evaluation procedure #1: Train and test on the entire dataset

1. Train the model on the **entire dataset**.
2. Test the model on the **same dataset**, and evaluate how well we did by comparing the **predicted** response values with the **true** response values.

```
In [1]: # added empty cell so that the cell numbering matches the video
```

```
In [2]: # read in the iris data
        from sklearn.datasets import load_iris
        iris = load_iris()

        # create X (features) and y (response)
        X = iris.data
        y = iris.target
```

Logistic regression

```
In [3]: # import the class
        from sklearn.linear_model import LogisticRegression

        # instantiate the model
        logreg = LogisticRegression(solver='liblinear')

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

        # predict the response values for the observations in X
        logreg.predict(X)
```

```
Out[3]: array([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,
               2, 1, 1, 1, 2, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 2, 2, 2, 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, 1, 2, 2,
               2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2])
```

```
In [4]: # store the predicted response values
        y_pred = logreg.predict(X)

        # check how many predictions were generated
        len(y_pred)
```

```
Out[4]: 150
```

Classification accuracy:

- **Proportion** of correct predictions
- Common **evaluation metric** for classification problems

```
In [5]: # compute classification accuracy for the logistic regression model
        from sklearn import metrics
        print(metrics.accuracy_score(y, y_pred))
```

```
0.96
```

- Known as **training accuracy** when you train and test the model on the same data

KNN (K=5)

```
In [6]: from sklearn.neighbors import KNeighborsClassifier
        knn = KNeighborsClassifier(n_neighbors=5)
        knn.fit(X, y)
        y_pred = knn.predict(X)
        print(metrics.accuracy_score(y, y_pred))
```

```
0.9666666666666667
```

KNN (K=1)

```
In [7]: knn = KNeighborsClassifier(n_neighbors=1)
        knn.fit(X, y)
        y_pred = knn.predict(X)
        print(metrics.accuracy_score(y, y_pred))
```

```
1.0
```

Problems with training and testing on the same data

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

Overfitting

Evaluation procedure #2: Train/test split

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

In [8]:

```
# print the shapes of X and y
print(X.shape)
print(y.shape)
```

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

In [9]:

```
# 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, rand
om_state=4)
```

Train/test split

What did this accomplish?

- Model can be trained and tested on **different data**
- Response values are known for the testing set, and thus **predictions can be evaluated**
- **Testing accuracy** is a better estimate than training accuracy of out-of-sample performance

In [10]:

```
# added empty cell so that the cell numbering matches the video
```

In [11]:

```
# print the shapes of the new X objects
print(X_train.shape)
print(X_test.shape)
```

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

In [12]:

```
# print the shapes of the new y objects
print(y_train.shape)
print(y_test.shape)
```

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

In [13]:

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

Out[13]:

```
LogisticRegression(solver='liblinear')
```

In [14]:

```
# STEP 3: make predictions on the testing set
y_pred = logreg.predict(X_test)

# compare actual response values (y_test) with predicted response values (y_p
red)
print(metrics.accuracy_score(y_test, y_pred))
```

```
0.9333333333333333
```

Repeat for KNN with K=5:

```
In [15]: knn = KNeighborsClassifier(n_neighbors=5)
         knn.fit(X_train, y_train)
         y_pred = knn.predict(X_test)
         print(metrics.accuracy_score(y_test, y_pred))
```

```
0.9666666666666667
```

Repeat for KNN with K=1:

```
In [16]: knn = KNeighborsClassifier(n_neighbors=1)
         knn.fit(X_train, y_train)
         y_pred = knn.predict(X_test)
         print(metrics.accuracy_score(y_test, y_pred))
```

```
0.95
```

Can we locate an even better value for K?

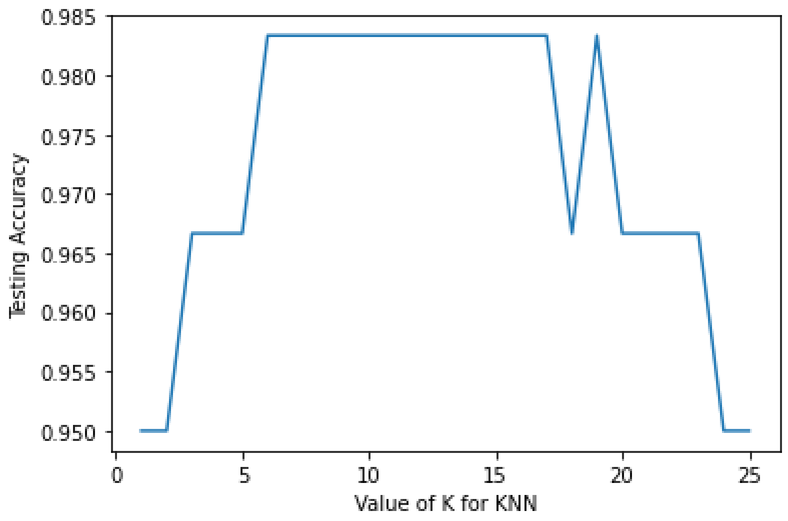
```
In [17]: # try K=1 through K=25 and record testing accuracy
         k_range = list(range(1, 26))
         scores = []
         for k in k_range:
             knn = KNeighborsClassifier(n_neighbors=k)
             knn.fit(X_train, y_train)
             y_pred = knn.predict(X_test)
             scores.append(metrics.accuracy_score(y_test, y_pred))
```

In [18]:
```
# import Matplotlib (scientific plotting library)
import matplotlib.pyplot as plt

# allow plots to appear within the notebook
%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 Accuracy')
```

Out[18]:
```
Text(0, 0.5, 'Testing Accuracy')
```



- **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 [19]:
```
# instantiate 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]])
```

Out[19]:
```
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**

Cross-validation for parameter tuning, model selection, and feature selection

- What is the drawback of using the **train/test split** procedure for model evaluation?
- How does **K-fold cross-validation** overcome this limitation?
- How can cross-validation be used for selecting **tuning parameters**, choosing between **models**, and selecting **features**?
- What are some possible **improvements** to cross-validation?

Review of model evaluation procedures

Motivation: Need a way to choose between Machine Learning models

- Goal is to estimate likely performance of a model on **out-of-sample data**

Initial idea: Train and test on the same data

- But, maximizing **training accuracy** rewards overly complex models which **overfit** the training data

Alternative idea: Train/test split

- Split the dataset into two pieces, so that the model can be trained and tested on **different data**
- **Testing accuracy** is a better estimate than training accuracy of out-of-sample performance
- But, it provides a **high variance** estimate since changing which observations happen to be in the testing set can significantly change testing accuracy

In [2]:
```
from sklearn.datasets import load_iris
from sklearn.model_selection import train_test_split
from sklearn.neighbors import KNeighborsClassifier
from sklearn import metrics
```

In [3]:
```python
# read in the iris data
iris = load_iris()

# create X (features) and y (response)
X = iris.data
y = iris.target
```

In [4]:
```python
# use train/test split with different random_state values
X_train, X_test, y_train, y_test = train_test_split(X, y, random_state=4)

# check classification accuracy of KNN with K=5
knn = KNeighborsClassifier(n_neighbors=5)
knn.fit(X_train, y_train)
y_pred = knn.predict(X_test)
print(metrics.accuracy_score(y_test, y_pred))
```

```
0.9736842105263158
```

Question: What if we created a bunch of train/test splits, calculated the testing accuracy for each, and averaged the results together?

Answer: That's the essense of cross-validation!

Steps for K-fold cross-validation

1. Split the dataset into K **equal** partitions (or "folds").
2. Use fold 1 as the **testing set** and the union of the other folds as the **training set**.
3. Calculate **testing accuracy**.
4. Repeat steps 2 and 3 K times, using a **different fold** as the testing set each time.
5. Use the **average testing accuracy** as the estimate of out-of-sample accuracy.

Diagram of **5-fold cross-validation:**

5-fold cross-validation

```
In [7]: # simulate splitting a dataset of 25 observations into 5 folds
        from sklearn.model_selection import KFold
        kf = KFold(n_splits=5, shuffle=False).split(range(25))

        # print the contents of each training and testing set
        print('{} {:^61} {}'.format('Iteration', 'Training set observations', 'Testin
        g set observations'))
        for iteration, data in enumerate(kf, start=1):
            print('{:^9} {} {:^25}'.format(iteration, data[0], str(data[1])))
```

```
Iteration                   Training set observations                   Testi
ng set observations
    1     [ 5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24]
[0 1 2 3 4]
    2     [ 0  1  2  3  4 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24]
[5 6 7 8 9]
    3     [ 0  1  2  3  4  5  6  7  8  9 15 16 17 18 19 20 21 22 23 24]
[10 11 12 13 14]
    4     [ 0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 20 21 22 23 24]
[15 16 17 18 19]
    5     [ 0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19]
[20 21 22 23 24]
```

- Dataset contains **25 observations** (numbered 0 through 24)
- 5-fold cross-validation, thus it runs for **5 iterations**
- For each iteration, every observation is either in the training set or the testing set, **but not both**
- Every observation is in the testing set **exactly once**

Comparing cross-validation to train/test split

Advantages of **cross-validation:**

- More accurate estimate of out-of-sample accuracy
- More "efficient" use of data (every observation is used for both training and testing)

Advantages of **train/test split:**

- Runs K times faster than K-fold cross-validation
- Simpler to examine the detailed results of the testing process

Cross-validation recommendations

1. K can be any number, but **K=10** is generally recommended
2. For classification problems, **stratified sampling** is recommended for creating the folds
    - Each response class should be represented with equal proportions in each of the K folds
    - scikit-learn's `cross_val_score` function does this by default

Cross-validation example: parameter tuning

Goal: Select the best tuning parameters (aka "hyperparameters") for KNN on the iris dataset

In [8]:
```
from sklearn.model_selection import cross_val_score
```

In [9]:
```
# 10-fold cross-validation with K=5 for KNN (the n_neighbors parameter)
knn = KNeighborsClassifier(n_neighbors=5)
scores = cross_val_score(knn, X, y, cv=10, scoring='accuracy')
print(scores)
```

```
[1.         0.93333333 1.         1.         0.86666667 0.93333333
 0.93333333 1.         1.         1.        ]
```

In [10]:
```
# use average accuracy as an estimate of out-of-sample accuracy
print(scores.mean())
```

```
0.9666666666666668
```

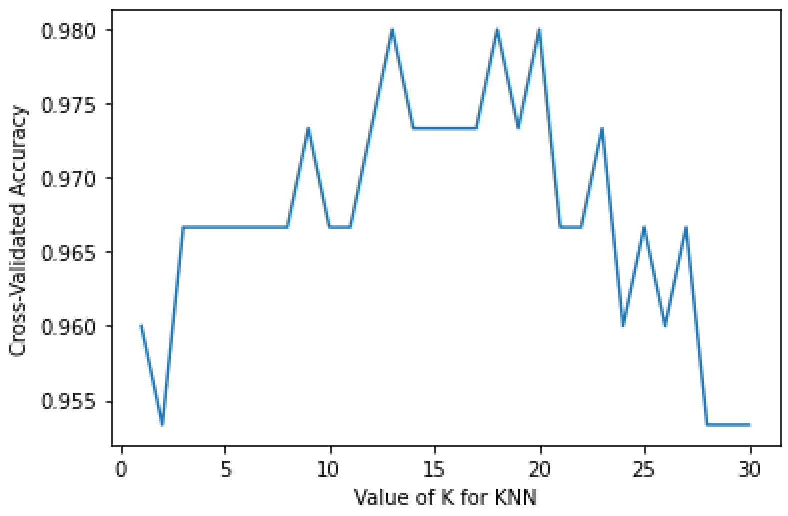
In [11]:
```
# search for an optimal value of K for KNN
k_range = list(range(1, 31))
k_scores = []
for k in k_range:
    knn = KNeighborsClassifier(n_neighbors=k)
    scores = cross_val_score(knn, X, y, cv=10, scoring='accuracy')
    k_scores.append(scores.mean())
print(k_scores)
```

```
[0.96, 0.9533333333333334, 0.9666666666666666, 0.9666666666666666, 0.96666666
66666668, 0.9666666666666668, 0.9666666666666668, 0.9666666666666668, 0.97333
33333333334, 0.9666666666666668, 0.9666666666666668, 0.9733333333333334, 0.98
00000000000001, 0.9733333333333334, 0.9733333333333334, 0.9733333333333334,
0.9733333333333334, 0.9800000000000001, 0.9733333333333334, 0.980000000000000
1, 0.9666666666666666, 0.9666666666666666, 0.9733333333333334, 0.96, 0.966666
6666666666, 0.96, 0.9666666666666666, 0.9533333333333334, 0.9533333333333334,
0.9533333333333334]
```

In [12]:
```python
import matplotlib.pyplot as plt
%matplotlib inline

# plot the value of K for KNN (x-axis) versus the cross-validated accuracy (y
-axis)
plt.plot(k_range, k_scores)
plt.xlabel('Value of K for KNN')
plt.ylabel('Cross-Validated Accuracy')
```

Out[12]:
```
Text(0, 0.5, 'Cross-Validated Accuracy')
```



Cross-validation example: model selection

Goal: Compare the best KNN model with logistic regression on the iris dataset

In [13]:
```python
# 10-fold cross-validation with the best KNN model
knn = KNeighborsClassifier(n_neighbors=20)
print(cross_val_score(knn, X, y, cv=10, scoring='accuracy').mean())
```

```
0.9800000000000001
```

In [14]:
```python
# 10-fold cross-validation with logistic regression
from sklearn.linear_model import LogisticRegression
logreg = LogisticRegression(solver='liblinear')
print(cross_val_score(logreg, X, y, cv=10, scoring='accuracy').mean())
```

```
0.9533333333333334
```

Improvements to cross-validation

Repeated cross-validation

- Repeat cross-validation multiple times (with **different random splits** of the data) and average the results
- More reliable estimate of out-of-sample performance by **reducing the variance** associated with a single trial of cross-validation

Creating a hold-out set

- "Hold out" a portion of the data **before** beginning the model building process
- Locate the best model using cross-validation on the remaining data, and test it **using the hold-out set**
- More reliable estimate of out-of-sample performance since hold-out set is **truly out-of-sample**

Feature engineering and selection within cross-validation iterations

- Normally, feature engineering and selection occurs **before** cross-validation
- Instead, perform all feature engineering and selection **within each cross-validation iteration**
- More reliable estimate of out-of-sample performance since it **better mimics** the application of the model to out-of-sample data

Efficiently searching for optimal tuning parameters

- How can K-fold cross-validation be used to search for an **optimal tuning parameter**?
- How can this process be made **more efficient**?
- How do you search for **multiple tuning parameters** at once?
- What do you do with those tuning parameters before making **real predictions**?
- How can the **computational expense** of this process be reduced?

Review of K-fold cross-validation

Steps for cross-validation:

- Dataset is split into K "folds" of **equal size**
- Each fold acts as the **testing set** 1 time, and acts as the **training set** K-1 times
- **Average testing performance** is used as the estimate of out-of-sample performance

Benefits of cross-validation:

- More **reliable** estimate of out-of-sample performance than train/test split
- Can be used for selecting **tuning parameters**, choosing between **models**, and selecting **features**

Drawbacks of cross-validation:

- Can be computationally **expensive**

Review of parameter tuning using `cross_val_score`

Goal: Select the best tuning parameters (aka "hyperparameters") for KNN on the iris dataset

In [2]:
```
from sklearn.datasets import load_iris
from sklearn.neighbors import KNeighborsClassifier
from sklearn.model_selection import cross_val_score
import matplotlib.pyplot as plt
%matplotlib inline
```

In [3]:
```
# read in the iris data
iris = load_iris()

# create X (features) and y (response)
X = iris.data
y = iris.target
```

In [4]:
```
# 10-fold cross-validation with K=5 for KNN (the n_neighbors parameter)
knn = KNeighborsClassifier(n_neighbors=5)
scores = cross_val_score(knn, X, y, cv=10, scoring='accuracy')
print(scores)
```

```
[1.         0.93333333 1.         1.         0.86666667 0.93333333
 0.93333333 1.         1.         1.        ]
```

In [5]:
```
# use average accuracy as an estimate of out-of-sample accuracy
print(scores.mean())
```

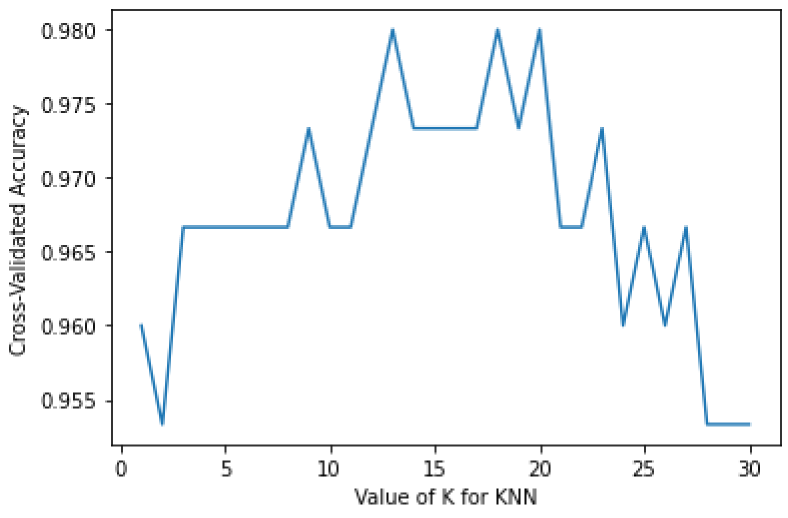
```
0.9666666666666668
```

In [6]:
```python
# search for an optimal value of K for KNN
k_range = list(range(1, 31))
k_scores = []
for k in k_range:
    knn = KNeighborsClassifier(n_neighbors=k)
    scores = cross_val_score(knn, X, y, cv=10, scoring='accuracy')
    k_scores.append(scores.mean())
print(k_scores)
```

```
[0.96, 0.9533333333333334, 0.9666666666666666, 0.9666666666666666, 0.96666666
66666668, 0.9666666666666668, 0.9666666666666668, 0.9666666666666668, 0.97333
33333333334, 0.9666666666666668, 0.9666666666666668, 0.9733333333333334, 0.98
00000000000001, 0.9733333333333334, 0.9733333333333334, 0.9733333333333334,
0.9733333333333334, 0.9800000000000001, 0.9733333333333334, 0.980000000000000
1, 0.9666666666666666, 0.9666666666666666, 0.9733333333333334, 0.96, 0.966666
6666666666, 0.96, 0.9666666666666666, 0.9533333333333334, 0.9533333333333334,
0.9533333333333334]
```

In [7]:
```python
# plot the value of K for KNN (x-axis) versus the cross-validated accuracy (y
-axis)
plt.plot(k_range, k_scores)
plt.xlabel('Value of K for KNN')
plt.ylabel('Cross-Validated Accuracy')
```

Out[7]:
```
Text(0, 0.5, 'Cross-Validated Accuracy')
```



More efficient parameter tuning using `GridSearchCV`

Allows you to define a **grid of parameters** that will be **searched** using K-fold cross-validation

In [8]:
```python
from sklearn.model_selection import GridSearchCV
```

In [9]:
```
# define the parameter values that should be searched
k_range = list(range(1, 31))
print(k_range)
```

```
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 2
2, 23, 24, 25, 26, 27, 28, 29, 30]
```

In [10]:
```
# create a parameter grid: map the parameter names to the values that should
be searched
param_grid = dict(n_neighbors=k_range)
print(param_grid)
```

```
{'n_neighbors': [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 1
8, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30]}
```

In [11]:
```
# instantiate the grid
grid = GridSearchCV(knn, param_grid, cv=10, scoring='accuracy')
```

- You can set **`n_jobs = -1`** to run computations in parallel (if supported by your computer and OS)

In [12]:
```
# fit the grid with data
grid.fit(X, y)
```

Out[12]:
```
GridSearchCV(cv=10, estimator=KNeighborsClassifier(n_neighbors=30),
             param_grid={'n_neighbors': [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 1
2,
                                         13, 14, 15, 16, 17, 18, 19, 20, 21,
22,
                                         23, 24, 25, 26, 27, 28, 29, 30]},
             scoring='accuracy')
```

In [13]:
```
# view the results as a pandas DataFrame
import pandas as pd
pd.DataFrame(grid.cv_results_)[['mean_test_score', 'std_test_score', 'param
s']]
```

Out[13]:

	mean_test_score	std_test_score	params
0	0.960000	0.053333	{'n_neighbors': 1}
1	0.953333	0.052068	{'n_neighbors': 2}
2	0.966667	0.044721	{'n_neighbors': 3}
3	0.966667	0.044721	{'n_neighbors': 4}
4	0.966667	0.044721	{'n_neighbors': 5}
5	0.966667	0.044721	{'n_neighbors': 6}
6	0.966667	0.044721	{'n_neighbors': 7}
7	0.966667	0.044721	{'n_neighbors': 8}
8	0.973333	0.032660	{'n_neighbors': 9}
9	0.966667	0.044721	{'n_neighbors': 10}
10	0.966667	0.044721	{'n_neighbors': 11}
11	0.973333	0.032660	{'n_neighbors': 12}
12	0.980000	0.030551	{'n_neighbors': 13}
13	0.973333	0.044222	{'n_neighbors': 14}
14	0.973333	0.032660	{'n_neighbors': 15}
15	0.973333	0.032660	{'n_neighbors': 16}
16	0.973333	0.032660	{'n_neighbors': 17}
17	0.980000	0.030551	{'n_neighbors': 18}
18	0.973333	0.032660	{'n_neighbors': 19}
19	0.980000	0.030551	{'n_neighbors': 20}
20	0.966667	0.033333	{'n_neighbors': 21}
21	0.966667	0.033333	{'n_neighbors': 22}
22	0.973333	0.032660	{'n_neighbors': 23}
23	0.960000	0.044222	{'n_neighbors': 24}
24	0.966667	0.033333	{'n_neighbors': 25}
25	0.960000	0.044222	{'n_neighbors': 26}
26	0.966667	0.044721	{'n_neighbors': 27}
27	0.953333	0.042687	{'n_neighbors': 28}
28	0.953333	0.042687	{'n_neighbors': 29}
29	0.953333	0.042687	{'n_neighbors': 30}

In [14]:
```
# examine the first result
print(grid.cv_results_['params'][0])
print(grid.cv_results_['mean_test_score'][0])
```

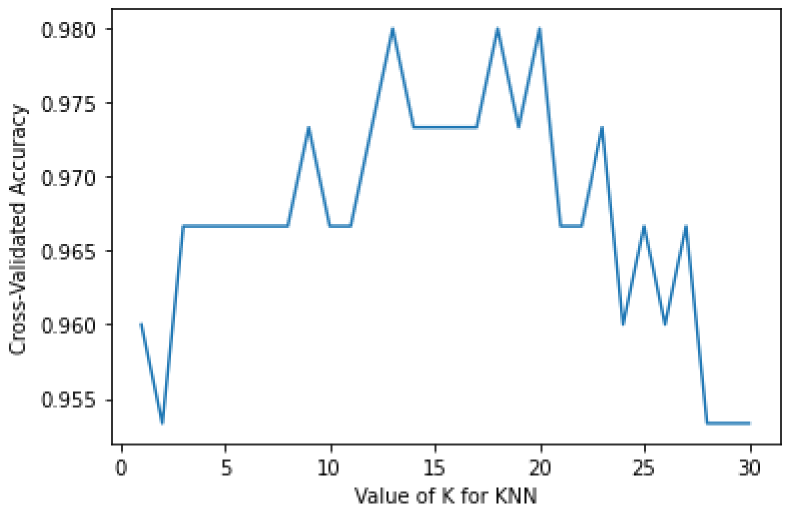
```
{'n_neighbors': 1}
0.96
```

In [15]:
```
# print the array of mean scores only
grid_mean_scores = grid.cv_results_['mean_test_score']
print(grid_mean_scores)
```

```
[0.96       0.95333333 0.96666667 0.96666667 0.96666667 0.96666667
 0.96666667 0.96666667 0.97333333 0.96666667 0.96666667 0.97333333
 0.98       0.97333333 0.97333333 0.97333333 0.97333333 0.98
 0.97333333 0.98       0.96666667 0.96666667 0.97333333 0.96
 0.96666667 0.96       0.96666667 0.95333333 0.95333333 0.95333333]
```

In [16]:
```
# plot the results
plt.plot(k_range, grid_mean_scores)
plt.xlabel('Value of K for KNN')
plt.ylabel('Cross-Validated Accuracy')
```

Out[16]:
```
Text(0, 0.5, 'Cross-Validated Accuracy')
```



In [17]:
```
# examine the best model
print(grid.best_score_)
print(grid.best_params_)
print(grid.best_estimator_)
```

```
0.9800000000000001
{'n_neighbors': 13}
KNeighborsClassifier(n_neighbors=13)
```

Searching multiple parameters simultaneously

- **Example:** tuning `max_depth` and `min_samples_leaf` for a `DecisionTreeClassifier`
- Could tune parameters **independently**: change `max_depth` while leaving `min_samples_leaf` at its default value, and vice versa
- But, best performance might be achieved when **neither parameter** is at its default value

In [18]:
```
# define the parameter values that should be searched
k_range = list(range(1, 31))
weight_options = ['uniform', 'distance']
```

In [19]:
```
# create a parameter grid: map the parameter names to the values that should
be searched
param_grid = dict(n_neighbors=k_range, weights=weight_options)
print(param_grid)
```

```
{'n_neighbors': [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 1
8, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30], 'weights': ['uniform', 'd
istance']}
```

In [20]:
```
# instantiate and fit the grid
grid = GridSearchCV(knn, param_grid, cv=10, scoring='accuracy')
grid.fit(X, y)
```

Out[20]:
```
GridSearchCV(cv=10, estimator=KNeighborsClassifier(n_neighbors=30),
             param_grid={'n_neighbors': [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 1
2,
                                         13, 14, 15, 16, 17, 18, 19, 20, 21,
22,
                                         23, 24, 25, 26, 27, 28, 29, 30],
                         'weights': ['uniform', 'distance']},
             scoring='accuracy')
```

In [21]:
```
# view the results
pd.DataFrame(grid.cv_results_)[['mean_test_score', 'std_test_score', 'param
s']]
```

Out[21]:

	mean_test_score	std_test_score	params
0	0.960000	0.053333	{'n_neighbors': 1, 'weights': 'uniform'}
1	0.960000	0.053333	{'n_neighbors': 1, 'weights': 'distance'}
2	0.953333	0.052068	{'n_neighbors': 2, 'weights': 'uniform'}
3	0.960000	0.053333	{'n_neighbors': 2, 'weights': 'distance'}
4	0.966667	0.044721	{'n_neighbors': 3, 'weights': 'uniform'}
5	0.966667	0.044721	{'n_neighbors': 3, 'weights': 'distance'}
6	0.966667	0.044721	{'n_neighbors': 4, 'weights': 'uniform'}
7	0.966667	0.044721	{'n_neighbors': 4, 'weights': 'distance'}
8	0.966667	0.044721	{'n_neighbors': 5, 'weights': 'uniform'}
9	0.966667	0.044721	{'n_neighbors': 5, 'weights': 'distance'}
10	0.966667	0.044721	{'n_neighbors': 6, 'weights': 'uniform'}
11	0.966667	0.044721	{'n_neighbors': 6, 'weights': 'distance'}
12	0.966667	0.044721	{'n_neighbors': 7, 'weights': 'uniform'}
13	0.966667	0.044721	{'n_neighbors': 7, 'weights': 'distance'}
14	0.966667	0.044721	{'n_neighbors': 8, 'weights': 'uniform'}
15	0.966667	0.044721	{'n_neighbors': 8, 'weights': 'distance'}
16	0.973333	0.032660	{'n_neighbors': 9, 'weights': 'uniform'}
17	0.973333	0.032660	{'n_neighbors': 9, 'weights': 'distance'}
18	0.966667	0.044721	{'n_neighbors': 10, 'weights': 'uniform'}
19	0.973333	0.032660	{'n_neighbors': 10, 'weights': 'distance'}
20	0.966667	0.044721	{'n_neighbors': 11, 'weights': 'uniform'}
21	0.973333	0.032660	{'n_neighbors': 11, 'weights': 'distance'}
22	0.973333	0.032660	{'n_neighbors': 12, 'weights': 'uniform'}
23	0.973333	0.044222	{'n_neighbors': 12, 'weights': 'distance'}
24	0.980000	0.030551	{'n_neighbors': 13, 'weights': 'uniform'}
25	0.973333	0.032660	{'n_neighbors': 13, 'weights': 'distance'}
26	0.973333	0.044222	{'n_neighbors': 14, 'weights': 'uniform'}
27	0.973333	0.032660	{'n_neighbors': 14, 'weights': 'distance'}
28	0.973333	0.032660	{'n_neighbors': 15, 'weights': 'uniform'}
29	0.980000	0.030551	{'n_neighbors': 15, 'weights': 'distance'}
30	0.973333	0.032660	{'n_neighbors': 16, 'weights': 'uniform'}
31	0.973333	0.032660	{'n_neighbors': 16, 'weights': 'distance'}
32	0.973333	0.032660	{'n_neighbors': 17, 'weights': 'uniform'}
33	0.980000	0.030551	{'n_neighbors': 17, 'weights': 'distance'}
34	0.980000	0.030551	{'n_neighbors': 18, 'weights': 'uniform'}

	mean_test_score	std_test_score	params
35	0.973333	0.032660	{'n_neighbors': 18, 'weights': 'distance'}
36	0.973333	0.032660	{'n_neighbors': 19, 'weights': 'uniform'}
37	0.980000	0.030551	{'n_neighbors': 19, 'weights': 'distance'}
38	0.980000	0.030551	{'n_neighbors': 20, 'weights': 'uniform'}
39	0.966667	0.044721	{'n_neighbors': 20, 'weights': 'distance'}
40	0.966667	0.033333	{'n_neighbors': 21, 'weights': 'uniform'}
41	0.966667	0.044721	{'n_neighbors': 21, 'weights': 'distance'}
42	0.966667	0.033333	{'n_neighbors': 22, 'weights': 'uniform'}
43	0.966667	0.044721	{'n_neighbors': 22, 'weights': 'distance'}
44	0.973333	0.032660	{'n_neighbors': 23, 'weights': 'uniform'}
45	0.973333	0.032660	{'n_neighbors': 23, 'weights': 'distance'}
46	0.960000	0.044222	{'n_neighbors': 24, 'weights': 'uniform'}
47	0.973333	0.032660	{'n_neighbors': 24, 'weights': 'distance'}
48	0.966667	0.033333	{'n_neighbors': 25, 'weights': 'uniform'}
49	0.973333	0.032660	{'n_neighbors': 25, 'weights': 'distance'}
50	0.960000	0.044222	{'n_neighbors': 26, 'weights': 'uniform'}
51	0.966667	0.044721	{'n_neighbors': 26, 'weights': 'distance'}
52	0.966667	0.044721	{'n_neighbors': 27, 'weights': 'uniform'}
53	0.980000	0.030551	{'n_neighbors': 27, 'weights': 'distance'}
54	0.953333	0.042687	{'n_neighbors': 28, 'weights': 'uniform'}
55	0.973333	0.032660	{'n_neighbors': 28, 'weights': 'distance'}
56	0.953333	0.042687	{'n_neighbors': 29, 'weights': 'uniform'}
57	0.973333	0.032660	{'n_neighbors': 29, 'weights': 'distance'}
58	0.953333	0.042687	{'n_neighbors': 30, 'weights': 'uniform'}
59	0.966667	0.033333	{'n_neighbors': 30, 'weights': 'distance'}

In [22]:
```
# examine the best model
print(grid.best_score_)
print(grid.best_params_)
```

```
0.9800000000000001
{'n_neighbors': 13, 'weights': 'uniform'}
```

Using the best parameters to make predictions

In [23]:
```
# train your model using all data and the best known parameters
knn = KNeighborsClassifier(n_neighbors=13, weights='uniform')
knn.fit(X, y)

# make a prediction on out-of-sample data
knn.predict([[3, 5, 4, 2]])
```

Out[23]:
```
array([1])
```

In [24]:
```
# shortcut: GridSearchCV automatically refits the best model using all of the
data
grid.predict([[3, 5, 4, 2]])
```

Out[24]:
```
array([1])
```

Reducing computational expense using `RandomizedSearchCV`

- Searching many different parameters at once may be computationally infeasible
- `RandomizedSearchCV` searches a subset of the parameters, and you control the computational "budget"

In [25]:
```
from sklearn.model_selection import RandomizedSearchCV
```

In [26]:
```
# specify "parameter distributions" rather than a "parameter grid"
param_dist = dict(n_neighbors=k_range, weights=weight_options)
```

- **Important:** Specify a continuous distribution (rather than a list of values) for any continous parameters

In [27]:
```python
# n_iter controls the number of searches
rand = RandomizedSearchCV(knn, param_dist, cv=10, scoring='accuracy', n_iter=
10, random_state=5)
rand.fit(X, y)
pd.DataFrame(rand.cv_results_)[['mean_test_score', 'std_test_score', 'param
s']]
```

Out[27]:

	mean_test_score	std_test_score	params
0	0.973333	0.032660	{'weights': 'distance', 'n_neighbors': 16}
1	0.966667	0.033333	{'weights': 'uniform', 'n_neighbors': 22}
2	0.980000	0.030551	{'weights': 'uniform', 'n_neighbors': 18}
3	0.966667	0.044721	{'weights': 'uniform', 'n_neighbors': 27}
4	0.953333	0.042687	{'weights': 'uniform', 'n_neighbors': 29}
5	0.973333	0.032660	{'weights': 'distance', 'n_neighbors': 10}
6	0.966667	0.044721	{'weights': 'distance', 'n_neighbors': 22}
7	0.973333	0.044222	{'weights': 'uniform', 'n_neighbors': 14}
8	0.973333	0.044222	{'weights': 'distance', 'n_neighbors': 12}
9	0.973333	0.032660	{'weights': 'uniform', 'n_neighbors': 15}

In [28]:
```python
# examine the best model
print(rand.best_score_)
print(rand.best_params_)
```

```
0.9800000000000001
{'weights': 'uniform', 'n_neighbors': 18}
```

In [29]:
```python
# run RandomizedSearchCV 20 times (with n_iter=10) and record the best score
best_scores = []
for _ in range(20):
    rand = RandomizedSearchCV(knn, param_dist, cv=10, scoring='accuracy', n_i
ter=10)
    rand.fit(X, y)
    best_scores.append(round(rand.best_score_, 3))
print(best_scores)
```

```
[0.98, 0.98, 0.98, 0.98, 0.973, 0.98, 0.973, 0.98, 0.98, 0.98, 0.973, 0.98,
0.98, 0.973, 0.973, 0.98, 0.98, 0.973, 0.973, 0.98]
```

Evaluating a classification model

- What is the purpose of **model evaluation**, and what are some common evaluation procedures?
- What is the usage of **classification accuracy**, and what are its limitations?
- How does a **confusion matrix** describe the performance of a classifier?
- What **metrics** can be computed from a confusion matrix?
- How can you adjust classifier performance by **changing the classification threshold**?
- What is the purpose of an **ROC curve**?
- How does **Area Under the Curve (AUC)** differ from classification accuracy?

Review of model evaluation

- Need a way to choose between models: different model types, tuning parameters, and features
- Use a **model evaluation procedure** to estimate how well a model will generalize to out-of-sample data
- Requires a **model evaluation metric** to quantify the model performance

Model evaluation procedures

1. **Training and testing on the same data**
    - Rewards overly complex models that "overfit" the training data and won't necessarily generalize
2. **Train/test split**
    - Split the dataset into two pieces, so that the model can be trained and tested on different data
    - Better estimate of out-of-sample performance, but still a "high variance" estimate
    - Useful due to its speed, simplicity, and flexibility
3. **K-fold cross-validation**
    - Systematically create "K" train/test splits and average the results together
    - Even better estimate of out-of-sample performance
    - Runs "K" times slower than train/test split

Model evaluation metrics

- **Regression problems:** Mean Absolute Error, Mean Squared Error, Root Mean Squared Error
- **Classification problems:** Classification accuracy

Classification accuracy

Pima Indians Diabetes dataset (https://www.kaggle.com/uciml/pima-indians-diabetes-database)

In [1]:
```
# read the data into a pandas DataFrame
import pandas as pd
path = 'data/pima-indians-diabetes.data'
col_names = ['pregnant', 'glucose', 'bp', 'skin', 'insulin', 'bmi', 'pedigree', 'ag
e', 'label']
pima = pd.read_csv(path, header=None, names=col_names)
```

In [2]:
```
# print the first 5 rows of data
pima.head()
```

Out[2]:

	pregnant	glucose	bp	skin	insulin	bmi	pedigree	age	label
0	6	148	72	35	0	33.6	0.627	50	1
1	1	85	66	29	0	26.6	0.351	31	0
2	8	183	64	0	0	23.3	0.672	32	1
3	1	89	66	23	94	28.1	0.167	21	0
4	0	137	40	35	168	43.1	2.288	33	1

Question: Can we predict the diabetes status of a patient given their health measurements?

In [3]:
```
# define X and y
feature_cols = ['pregnant', 'insulin', 'bmi', 'age']
X = pima[feature_cols]
y = pima.label
```

In [4]:
```
# 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, random_state=0)
```

In [5]:
```
# train a logistic regression model on the training set
from sklearn.linear_model import LogisticRegression
logreg = LogisticRegression(solver='liblinear')
logreg.fit(X_train, y_train)
```

Out[5]:
```
LogisticRegression(C=1.0, class_weight=None, dual=False, fit_intercept=True,
                   intercept_scaling=1, l1_ratio=None, max_iter=100,
                   multi_class='auto', n_jobs=None, penalty='l2',
                   random_state=None, solver='liblinear', tol=0.0001, verbose=0,
                   warm_start=False)
```

In [7]:
```
# make class predictions for the testing set
y_pred_class = logreg.predict(X_test)
```

Classification accuracy: percentage of correct predictions

```
In [8]: # calculate accuracy
        from sklearn import metrics
        print(metrics.accuracy_score(y_test, y_pred_class))

        0.6927083333333334
```

Null accuracy: accuracy that could be achieved by always predicting the most frequent class

```
In [9]: # examine the class distribution of the testing set (using a Pandas Series method)
        y_test.value_counts()

Out[9]: 0    130
        1     62
        Name: label, dtype: int64
```

```
In [10]: # calculate the percentage of ones
         y_test.mean()

Out[10]: 0.3229166666666667
```

```
In [11]: # calculate the percentage of zeros
         1 - y_test.mean()

Out[11]: 0.6770833333333333
```

```
In [12]: # calculate null accuracy (for binary classification problems coded as 0/1)
         max(y_test.mean(), 1 - y_test.mean())

Out[12]: 0.6770833333333333
```

```
In [13]: # calculate null accuracy (for multi-class classification problems)
         y_test.value_counts().head(1) / len(y_test)

Out[13]: 0    0.677083
         Name: label, dtype: float64
```

Comparing the **true** and **predicted** response values

```
In [14]: # print the first 25 true and predicted responses
         print('True:', y_test.values[0:25])
         print('Pred:', y_pred_class[0:25])

         True: [1 0 0 1 0 0 1 1 0 0 1 1 0 0 0 0 1 0 0 0 1 1 0 0 0]
         Pred: [0 0 0 0 0 0 0 1 0 1 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0]
```

Conclusion:

- Classification accuracy is the **easiest classification metric to understand**
- But, it does not tell you the **underlying distribution** of response values
- And, it does not tell you what **"types" of errors** your classifier is making

Confusion matrix

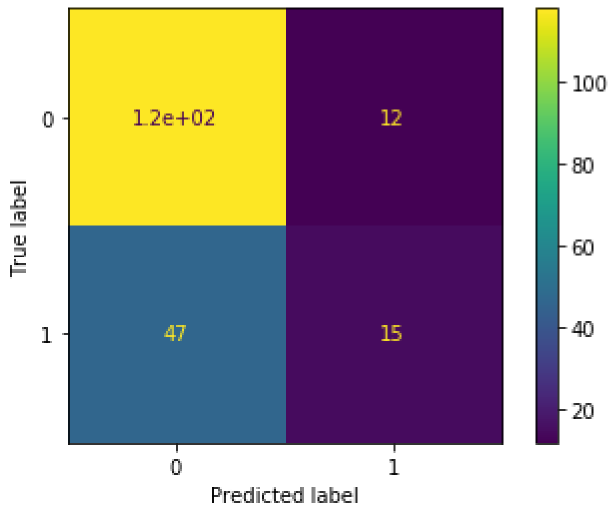
Table that describes the performance of a classification model

In [15]:
```
# IMPORTANT: first argument is true values, second argument is predicted values
print(metrics.confusion_matrix(y_test, y_pred_class))
```

```
[[118  12]
 [ 47  15]]
```

In [21]:
```
from sklearn.metrics import ConfusionMatrixDisplay
cm=metrics.confusion_matrix(y_test, y_pred_class)
disp = ConfusionMatrixDisplay(cm,display_labels=logreg.classes_)
disp.plot()
```

Out[21]:
```
<sklearn.metrics._plot.confusion_matrix.ConfusionMatrixDisplay at 0x256c5e4d048>
```



- Every observation in the testing set is represented in **exactly one box**
- It's a 2x2 matrix because there are **2 response classes**

Basic terminology

- **True Positives (TP):** we *correctly* predicted that they *do* have diabetes
- **True Negatives (TN):** we *correctly* predicted that they *don't* have diabetes
- **False Positives (FP):** we *incorrectly* predicted that they *do* have diabetes (a "Type I error")
- **False Negatives (FN):** we *incorrectly* predicted that they *don't* have diabetes (a "Type II error")

In [16]:
```
# print the first 25 true and predicted responses
print('True:', y_test.values[0:25])
print('Pred:', y_pred_class[0:25])
```

```
True: [1 0 0 1 0 0 1 1 0 0 1 1 0 0 0 0 1 0 0 0 1 1 0 0 0]
Pred: [0 0 0 0 0 0 0 1 0 1 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0]
```

```
In [17]: # save confusion matrix and slice into four pieces
         confusion = metrics.confusion_matrix(y_test, y_pred_class)
         TP = confusion[1, 1]
         TN = confusion[0, 0]
         FP = confusion[0, 1]
         FN = confusion[1, 0]
```

Large confusion matrix

Metrics computed from a confusion matrix

Classification Accuracy: Overall, how often is the classifier correct?

```
In [18]: print((TP + TN) / (TP + TN + FP + FN))
         print(metrics.accuracy_score(y_test, y_pred_class))
```

```
0.6927083333333334
0.6927083333333334
```

Classification Error: Overall, how often is the classifier incorrect?

- Also known as "Misclassification Rate"

```
In [19]: print((FP + FN) / (TP + TN + FP + FN))
         print(1 - metrics.accuracy_score(y_test, y_pred_class))
```

```
0.3072916666666667
0.30729166666666663
```

Sensitivity: When the actual value is positive, how often is the prediction correct?

- How "sensitive" is the classifier to detecting positive instances?
- Also known as "True Positive Rate" or "Recall"

```
In [20]: print(TP / (TP + FN))
         print(metrics.recall_score(y_test, y_pred_class))
```

```
0.24193548387096775
0.24193548387096775
```

Specificity: When the actual value is negative, how often is the prediction correct?

- How "specific" (or "selective") is the classifier in predicting positive instances?

In [21]:
```
print(TN / (TN + FP))
```
```
0.9076923076923077
```

False Positive Rate: When the actual value is negative, how often is the prediction incorrect?

In [22]:
```
print(FP / (TN + FP))
```
```
0.09230769230769231
```

Precision: When a positive value is predicted, how often is the prediction correct?

- How "precise" is the classifier when predicting positive instances?

In [23]:
```
print(TP / (TP + FP))
print(metrics.precision_score(y_test, y_pred_class))
```
```
0.5555555555555556
0.5555555555555556
```

Many other metrics can be computed: F1 score, Matthews correlation coefficient, etc.

Conclusion:

- Confusion matrix gives you a **more complete picture** of how your classifier is performing
- Also allows you to compute various **classification metrics**, and these metrics can guide your model selection

Which metrics should you focus on?

- Choice of metric depends on your **business objective**
- **Spam filter** (positive class is "spam"): Optimize for **precision or specificity** because false negatives (spam goes to the inbox) are more acceptable than false positives (non-spam is caught by the spam filter)
- **Fraudulent transaction detector** (positive class is "fraud"): Optimize for **sensitivity** because false positives (normal transactions that are flagged as possible fraud) are more acceptable than false negatives (fraudulent transactions that are not detected)

Adjusting the classification threshold

In [24]:
```
# print the first 10 predicted responses
logreg.predict(X_test)[0:10]
```
Out[24]:
```
array([0, 0, 0, 0, 0, 0, 0, 1, 0, 1])
```

```
In [25]: # print the first 10 predicted probabilities of class membership
         logreg.predict_proba(X_test)[0:10, :]
```

```
Out[25]: array([[0.63247571, 0.36752429],
                [0.71643656, 0.28356344],
                [0.71104114, 0.28895886],
                [0.5858938 , 0.4141062 ],
                [0.84103973, 0.15896027],
                [0.82934844, 0.17065156],
                [0.50110974, 0.49889026],
                [0.48658459, 0.51341541],
                [0.72321388, 0.27678612],
                [0.32810562, 0.67189438]])
```

```
In [26]: # print the first 10 predicted probabilities for class 1
         logreg.predict_proba(X_test)[0:10, 1]
```

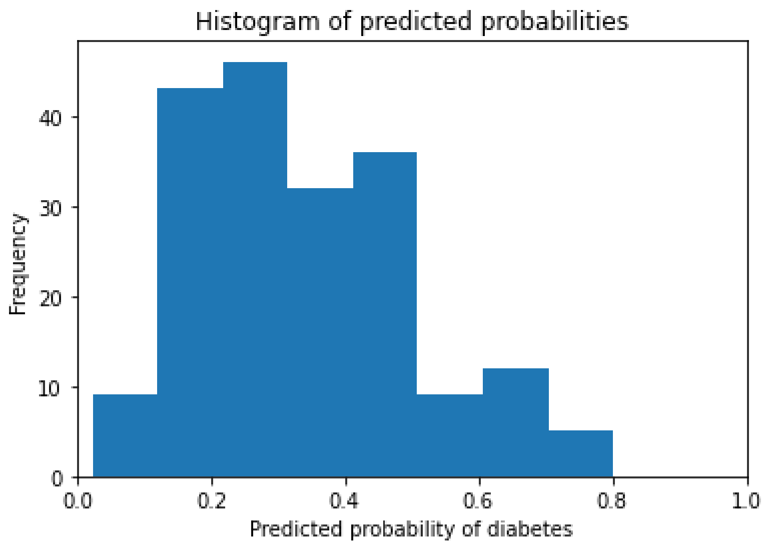
```
Out[26]: array([0.36752429, 0.28356344, 0.28895886, 0.4141062 , 0.15896027,
                0.17065156, 0.49889026, 0.51341541, 0.27678612, 0.67189438])
```

```
In [27]: # store the predicted probabilities for class 1
         y_pred_prob = logreg.predict_proba(X_test)[:, 1]
```

```
In [28]: # allow plots to appear in the notebook
         %matplotlib inline
         import matplotlib.pyplot as plt
```

```
In [29]: # histogram of predicted probabilities
         plt.hist(y_pred_prob, bins=8)
         plt.xlim(0, 1)
         plt.title('Histogram of predicted probabilities')
         plt.xlabel('Predicted probability of diabetes')
         plt.ylabel('Frequency')
```

```
Out[29]: Text(0, 0.5, 'Frequency')
```



Decrease the threshold for predicting diabetes in order to **increase the sensitivity** of the classifier

In [30]:
```
# predict diabetes if the predicted probability is greater than 0.3
from sklearn.preprocessing import binarize
y_pred_class = binarize([y_pred_prob], threshold=0.3)[0]
```

In [31]:
```
# print the first 10 predicted probabilities
y_pred_prob[0:10]
```

Out[31]:
```
array([0.36752429, 0.28356344, 0.28895886, 0.4141062 , 0.15896027,
       0.17065156, 0.49889026, 0.51341541, 0.27678612, 0.67189438])
```

In [32]:
```
# print the first 10 predicted classes with the lower threshold
y_pred_class[0:10]
```

Out[32]:
```
array([1., 0., 0., 1., 0., 0., 1., 1., 0., 1.])
```

In [33]:
```
# previous confusion matrix (default threshold of 0.5)
print(confusion)
```

```
[[118  12]
 [ 47  15]]
```

In [34]:
```
# new confusion matrix (threshold of 0.3)
print(metrics.confusion_matrix(y_test, y_pred_class))
```

```
[[80 50]
 [16 46]]
```

In [35]:
```
# sensitivity has increased (used to be 0.24)
print(46 / (46 + 16))
```

```
0.7419354838709677
```

In [36]:
```
# specificity has decreased (used to be 0.91)
print(80 / (80 + 50))
```

```
0.6153846153846154
```

Conclusion:

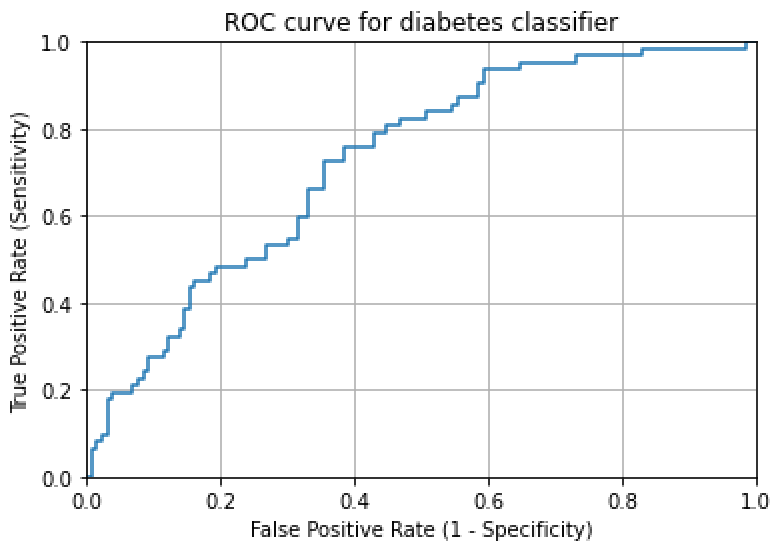
- **Threshold of 0.5** is used by default (for binary problems) to convert predicted probabilities into class predictions
- Threshold can be **adjusted** to increase sensitivity or specificity
- Sensitivity and specificity have an **inverse relationship**

ROC Curves and Area Under the Curve (AUC)

Question: Wouldn't it be nice if we could see how sensitivity and specificity are affected by various thresholds, without actually changing the threshold?

Answer: Plot the ROC curve!

In [37]:
```
# IMPORTANT: first argument is true values, second argument is predicted probabilitie
s
fpr, tpr, thresholds = metrics.roc_curve(y_test, y_pred_prob)
plt.plot(fpr, tpr)
plt.xlim([0.0, 1.0])
plt.ylim([0.0, 1.0])
plt.title('ROC curve for diabetes classifier')
plt.xlabel('False Positive Rate (1 - Specificity)')
plt.ylabel('True Positive Rate (Sensitivity)')
plt.grid(True)
```



- ROC curve can help you to **choose a threshold** that balances sensitivity and specificity in a way that makes sense for your particular context
- You can't actually **see the thresholds** used to generate the curve on the ROC curve itself

In [38]:
```
# define a function that accepts a threshold and prints sensitivity and specificity
def evaluate_threshold(threshold):
    print('Sensitivity:', tpr[thresholds > threshold][-1])
    print('Specificity:', 1 - fpr[thresholds > threshold][-1])
```

In [39]:
```
evaluate_threshold(0.5)
```

```
Sensitivity: 0.24193548387096775
Specificity: 0.9076923076923077
```

In [40]:
```
evaluate_threshold(0.3)
```

```
Sensitivity: 0.7258064516129032
Specificity: 0.6153846153846154
```

AUC is the **percentage** of the ROC plot that is **underneath the curve**:

In [41]:
```
# IMPORTANT: first argument is true values, second argument is predicted probabilitie
s
print(metrics.roc_auc_score(y_test, y_pred_prob))
```

```
0.7245657568238213
```

- AUC is useful as a **single number summary** of classifier performance.
- If you randomly chose one positive and one negative observation, AUC represents the likelihood that your classifier will assign a **higher predicted probability** to the positive observation.
- AUC is useful even when there is **high class imbalance** (unlike classification accuracy).

In [42]:
```
# calculate cross-validated AUC
from sklearn.model_selection import cross_val_score
cross_val_score(logreg, X, y, cv=10, scoring='roc_auc').mean()
```

Out[42]:
```
0.7378233618233618
```

Confusion matrix advantages:

- Allows you to calculate a **variety of metrics**
- Useful for **multi-class problems** (more than two response classes)

ROC/AUC advantages:

- Does not require you to **set a classification threshold**
- Still useful when there is **high class imbalance**

Building a Machine Learning workflow

- Why should you use a Pipeline?
- How do you encode categorical features with OneHotEncoder?
- How do you apply OneHotEncoder to selected columns with ColumnTransformer?
- How do you build and cross-validate a Pipeline?
- How do you make predictions on new data using a Pipeline?
- Why should you use scikit-learn (rather than pandas) for preprocessing?

Step 1: Load the dataset

In [1]:
```
import pandas as pd
```

In [2]:
```
df = pd.read_csv('http://bit.ly/kaggletrain')
```

In [3]:
```
df.shape
```
Out[3]:
```
(891, 12)
```

Step 2: Select features

In [4]:
```
df.columns
```
Out[4]:
```
Index(['PassengerId', 'Survived', 'Pclass', 'Name', 'Sex', 'Age', 'SibSp',
       'Parch', 'Ticket', 'Fare', 'Cabin', 'Embarked'],
      dtype='object')
```

In [5]:
```
df.isna().sum()
```
Out[5]:
```
PassengerId      0
Survived         0
Pclass           0
Name             0
Sex              0
Age            177
SibSp            0
Parch            0
Ticket           0
Fare             0
Cabin          687
Embarked         2
dtype: int64
```

In [6]:
```
df = df.loc[df.Embarked.notna(), ['Survived', 'Pclass', 'Sex', 'Embarked']]
```

In [7]:
```
df.shape
```
Out[7]:
```
(889, 4)
```

In [8]:
```
df.isna().sum()
```
Out[8]:
```
Survived    0
Pclass      0
Sex         0
Embarked    0
dtype: int64
```

In [9]:
```
df.head()
```
Out[9]:

	Survived	Pclass	Sex	Embarked
0	0	3	male	S
1	1	1	female	C
2	1	3	female	S
3	1	1	female	S
4	0	3	male	S

Step 3: Cross-validate a model with one feature

```
In [10]: X = df.loc[:, ['Pclass']]
         y = df.Survived
```

```
In [11]: X.shape
```

```
Out[11]: (889, 1)
```

```
In [12]: y.shape
```

```
Out[12]: (889,)
```

```
In [13]: from sklearn.linear_model import LogisticRegression
```

```
In [14]: logreg = LogisticRegression()
```

```
In [15]: from sklearn.model_selection import cross_val_score
```

```
In [16]: cross_val_score(logreg, X, y, cv=5, scoring='accuracy').mean()
```

```
Out[16]: 0.6783406335301212
```

```
In [17]: y.value_counts(normalize=True)
```

```
Out[17]: 0    0.617548
         1    0.382452
         Name: Survived, dtype: float64
```

Step 4: Encode categorical features

```
In [18]: df.head()
```

Out[18]:

	Survived	Pclass	Sex	Embarked
0	0	3	male	S
1	1	1	female	C
2	1	3	female	S
3	1	1	female	S
4	0	3	male	S

```
In [19]: # dummy encoding of categorical features
         from sklearn.preprocessing import OneHotEncoder
         ohe = OneHotEncoder(sparse=False)
```

```
In [20]: ohe.fit_transform(df[['Sex']])
```

```
Out[20]: array([[0., 1.],
                [1., 0.],
                [1., 0.],
                ...,
                [1., 0.],
                [0., 1.],
                [0., 1.]])
```

```
In [21]: ohe.categories_
```

```
Out[21]: [array(['female', 'male'], dtype=object)]
```

```
In [22]: ohe.fit_transform(df[['Embarked']])
```

```
Out[22]: array([[0., 0., 1.],
                [1., 0., 0.],
                [0., 0., 1.],
                ...,
                [0., 0., 1.],
                [1., 0., 0.],
                [0., 1., 0.]])
```

```
In [23]: ohe.categories_
```

```
Out[23]: [array(['C', 'Q', 'S'], dtype=object)]
```

Step 5: Cross-validate a Pipeline with all features

```
In [24]: X = df.drop('Survived', axis='columns')
```

```
In [25]: X.head()
```

Out[25]:

	Pclass	Sex	Embarked
0	3	male	S
1	1	female	C
2	3	female	S
3	1	female	S
4	3	male	S

```
In [26]: # use when different features need different preprocessing
         from sklearn.compose import make_column_transformer
```

```
In [27]: column_trans = make_column_transformer(
             (OneHotEncoder(), ['Sex', 'Embarked']),
             remainder='passthrough')
```

In [28]:
```
column_trans.fit_transform(X)
```

Out[28]:
```
array([[0., 1., 0., 0., 1., 3.],
       [1., 0., 1., 0., 0., 1.],
       [1., 0., 0., 0., 1., 3.],
       ...,
       [1., 0., 0., 0., 1., 3.],
       [0., 1., 1., 0., 0., 1.],
       [0., 1., 0., 1., 0., 3.]])
```

In [29]:
```
# chain sequential steps together
from sklearn.pipeline import make_pipeline
```

In [30]:
```
pipe = make_pipeline(column_trans, logreg)
```

In [31]:
```
# cross-validate the entire process
# thus, preprocessing occurs within each fold of cross-validation
cross_val_score(pipe, X, y, cv=5, scoring='accuracy').mean()
```

Out[31]:
```
0.7727924839713071
```

Step 6: Make predictions on "new" data

In [32]:
```
# added empty cell so that the cell numbering matches the video
```

In [33]:
```
X_new = X.sample(5, random_state=99)
X_new
```

Out[33]:

	Pclass	Sex	Embarked
599	1	male	C
512	1	male	S
273	1	male	C
215	1	female	C
790	3	male	Q

In [34]:
```
pipe.fit(X, y)
```

Out[34]:
```
Pipeline(steps=[('columntransformer',
                 ColumnTransformer(remainder='passthrough',
                                   transformers=[('onehotencoder',
                                                  OneHotEncoder(),
                                                  ['Sex', 'Embarked'])])),
                ('logisticregression', LogisticRegression())])
```

In [35]:
```
pipe.predict(X_new)
```

Out[35]:
```
array([1, 0, 1, 1, 0])
```

Recap

```
In [36]: import pandas as pd
         from sklearn.compose import make_column_transformer
         from sklearn.preprocessing import OneHotEncoder
         from sklearn.linear_model import LogisticRegression
         from sklearn.pipeline import make_pipeline
         from sklearn.model_selection import cross_val_score
```

```
In [37]: df = pd.read_csv('http://bit.ly/kaggletrain')
         df = df.loc[df.Embarked.notna(), ['Survived', 'Pclass', 'Sex', 'Embarked']]
         X = df.drop('Survived', axis='columns')
         y = df.Survived
```

```
In [38]: column_trans = make_column_transformer(
             (OneHotEncoder(), ['Sex', 'Embarked']),
             remainder='passthrough')
         logreg = LogisticRegression(solver='lbfgs')
```

```
In [39]: pipe = make_pipeline(column_trans, logreg)
```

```
In [40]: cross_val_score(pipe, X, y, cv=5, scoring='accuracy').mean()

Out[40]: 0.7727924839713071
```

```
In [41]: X_new = X.sample(5, random_state=99)
```

```
In [42]: pipe.fit(X, y)
         pipe.predict(X_new)

Out[42]: array([1, 0, 1, 1, 0])
```