Scikit-Learn Practice Exercises

This notebook offers a set of excercises for different tasks with Scikit-Learn.

Notes:

- There may be more than one different way to answer a question or complete an exercise.
- Some skeleton code has been implemented for you.
- Exercises are based off (and directly taken from) the quick introduction to Scikit-Learn notebook.
- Different tasks will be detailed by comments or text. Places to put your own code are defined by `###` (don't remove anything other than `###`).

For further reference and resources, it's advised to check out the Scikit-Learn documnetation.

And if you get stuck, try searching for a question in the following format: "how to do XYZ with Scikit-Learn", where XYZ is the function you want to leverage from Scikit-Learn.

Since we'll be working with data, we'll import Scikit-Learn's counterparts, Matplotlib, NumPy and pandas.

Let's get started.

In [1]:

```
# Setup matplotlib to plot inline (within the notebook)
###
%matplotlib inline
# Import the pyplot module of Matplotlib as plt
###
import matplotlib.pyplot as plt
# Import pandas under the abbreviation 'pd'
###
import pandas as pd
# Import NumPy under the abbreviation 'np'
###
import numpy as np
```

End-to-end Scikit-Learn classification workflow

Let's start with an end to end Scikit-Learn workflow.

More specifically, we'll:

1. Get a dataset ready
2. Prepare a machine learning model to make predictions
3. Fit the model to the data and make a prediction
4. Evaluate the model's predictions

The data we'll be using is stored on GitHub. We'll start with `heart-disease.csv`, a dataset which contains anonymous patient data and whether or not they have heart disease.

Note: When viewing a `.csv` on GitHub, make sure it's in the raw format. For example, the URL should look like: https://raw.githubusercontent.com/mrdbourke/zero-to-mastery-ml/master/data/heart-disease.csv

1. Getting a dataset ready

In [2]:

```
# Import the heart disease dataset and save it to a variable
# using pandas and read_csv()
# Hint: You can directly pass the URL of a csv to read_csv()
heart_disease = pd.read_csv("heart-disease.csv")

# Check the first 5 rows of the data
heart_disease.head()
```

Out[2]:

	age	sex	cp	trestbps	chol	fbs	restecg	thalach	exang	oldpeak	slope	ca	thal	target
0	63	1	3	145	233	1	0	150	0	2.3	0	0	1	1
1	37	1	2	130	250	0	1	187	0	3.5	0	0	2	1
2	41	0	1	130	204	0	0	172	0	1.4	2	0	2	1
3	56	1	1	120	236	0	1	178	0	0.8	2	0	2	1
4	57	0	0	120	354	0	1	163	1	0.6	2	0	2	1

Our goal here is to build a machine learning model on all of the columns except `target` to predict `target`.

In essence, the `target` column is our **target variable** (also called `y` or `labels`) and the rest of the other columns are our independent variables (also called `data` or `X`).

And since our target variable is one thing or another (heart disease or not), we know our problem is a classification problem (classifying whether something is one thing or another).

Knowing this, let's create `X` and `y` by splitting our dataframe up.

In [5]:

```
# Create X (all columns except target)
X = heart_disease.drop("target", axis=1)

# Create y (only the target column)
y = heart_disease["target"]
```

Now we've split our data into `X` and `y`, we'll use Scikit-Learn to split it into training and test sets.

In [24]:

```
# Import train_test_split from sklearn's model_selection module
from sklearn.model_selection import train_test_split
np.random.seed(42)
# Use train_test_split to split X & y into training and test sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2)
```

In [8]:

```
# View the different shapes of the training and test datasets
X_train.shape, X_test.shape, y_train.shape, y_test.shape
```

Out[8]:

```
((242, 13), (61, 13), (242,), (61,))
```

What do you notice about the different shapes of the data?

Since our data is now in training and test sets, we'll build a machine learning model to fit patterns in the training data and then make predictions on the test data.

To figure out which machine learning model we should use, you can refer to Scikit-Learn's machine learning map.

After following the map, you decide to use the `RandomForestClassifier`.

2. Preparing a machine learning model

In [9]:

```
# Import the RandomForestClassifier from sklearn's ensemble module
from sklearn.ensemble import RandomForestClassifier

# Instantiate an instance of RandomForestClassifier as clf
clf = RandomForestClassifier()
```

Now you've got a `RandomForestClassifier` instance, let's fit it to the training data.

Once it's fit, we'll make predictions on the test data.

3. Fitting a model and making predictions

In [10]:

```
# Fit the RandomForestClassifier to the training data
clf.fit(X_train, y_train)
```

Out[10]:

```
RandomForestClassifier()
```

In [11]:

```
# Use the fitted model to make predictions on the test data and
# save the predictions to a variable called y_preds
y_preds = clf.predict(X_test)
```

4. Evaluating a model's predictions

Evaluating predictions is as important making them. Let's check how our model did by calling the `score()` method on it and passing it the training (`X_train, y_train`) and testing data (`X_test, y_test`).

In [12]:

```
# Evaluate the fitted model on the training set using the score() function
clf.score(X_train, y_train)
```

Out[12]:

```
1.0
```

In [13]:

```
# Evaluate the fitted model on the test set using the score() function
clf.score(X_test, y_test)
```

Out[13]:

```
0.8688524590163934
```

- How did you model go?
- What metric does `score()` return for classifiers?
- Did your model do better on the training dataset or test dataset?

Experimenting with different classification models

Now we've quickly covered an end-to-end Scikit-Learn workflow and since experimenting is a large part of machine learning, we'll now try a series of different machine learning models and see which gets the best results on our dataset.

Going through the Scikit-Learn machine learning map, we see there are a number of different classification models we can try (different models are in the green boxes).

For this exercise, the models we're going to try and compare are:

- LinearSVC
- KNeighborsClassifier (also known as K-Nearest Neighbors or KNN)
- SVC (also known as support vector classifier, a form of support vector machine)
- LogisticRegression (despite the name, this is actually a classifier)
- RandomForestClassifier (an ensemble method and what we used above)

We'll follow the same workflow we used above (except this time for multiple models):

1. Import a machine learning model
2. Get it ready
3. Fit it to the data and make predictions
4. Evaluate the fitted model

Note: Since we've already got the data ready, we can reuse it in this section.

In [14]:

```python
# Import LinearSVC from sklearn's svm module
from sklearn.svm import LinearSVC

# Import KNeighborsClassifier from sklearn's neighbors module
from sklearn.neighbors import KNeighborsClassifier

# Import SVC from sklearn's svm module
from sklearn.svm import SVC

# Import LogisticRegression from sklearn's linear_model module
from sklearn.linear_model import LogisticRegression

# Note: we don't have to import RandomForestClassifier, since we already have
```

Thanks to the consistency of Scikit-Learn's API design, we can use virtually the same code to fit, score and make predictions with each of our models.

To see which model performs best, we'll do the following:

1. Instantiate each model in a dictionary
2. Create an empty results dictionary
3. Fit each model on the training data
4. Score each model on the test data
5. Check the results

If you're wondering what it means to instantiate each model in a dictionary, see the example below.

In [18]:

```python
# EXAMPLE: Instantiating a RandomForestClassifier() in a dictionary
example_dict = {"RandomForestClassifier": RandomForestClassifier()}
```

```
# Create a dictionary called models which contains all of the classification models we've imported
# Make sure the dictionary is in the same format as example_dict
# The models dictionary should contain 5 models
models = {"LinearSVC": LinearSVC(),
          "KNN": KNeighborsClassifier(),
          "SVC": SVC(),
          "LogisticRegression": LogisticRegression(),
          "RandomForestClassifier": RandomForestClassifier()}

# Create an empty dictionary called results
results = {}
```

Since each model we're using has the same `fit()` and `score()` functions, we can loop through our models dictionary and, call `fit()` on the training data and then call `score()` with the test data.

In [19]:

```
# EXAMPLE: Looping through example_dict fitting and scoring the model
example_results = {}
for model_name, model in example_dict.items():
    model.fit(X_train, y_train)
    example_results[model_name] = model.score(X_test, y_test)

# EXAMPLE: View the results
example_results
```

Out[19]:

```
{'RandomForestClassifier': 0.8688524590163934}
```

In [21]:

```
# Loop through the models dictionary items, fitting the model on the training data
# and appending the model name and model score on the test data to the results dictionary
for model_name, model in models.items():
    model.fit(X_train, y_train)
    results[model_name] = model.score(X_test, y_test)

# View the results
results
```

```
D:\ds_and_ml_projects\sample_project\env\lib\site-packages\sklearn\svm\_base.py:985: ConvergenceWa
rning: Liblinear failed to converge, increase the number of iterations.
  warnings.warn("Liblinear failed to converge, increase "
D:\ds_and_ml_projects\sample_project\env\lib\site-packages\sklearn\linear_model\_logistic.py:763:
ConvergenceWarning: lbfgs failed to converge (status=1):
STOP: TOTAL NO. of ITERATIONS REACHED LIMIT.

Increase the number of iterations (max_iter) or scale the data as shown in:
    https://scikit-learn.org/stable/modules/preprocessing.html
Please also refer to the documentation for alternative solver options:
    https://scikit-learn.org/stable/modules/linear_model.html#logistic-regression
  n_iter_i = _check_optimize_result(
```

Out[21]:

```
{'LinearSVC': 0.7049180327868853,
 'KNN': 0.639344262295082,
 'SVC': 0.6229508196721312,
 'LogisticRegression': 0.8524590163934426,
 'RandomForestClassifier': 0.8032786885245902}
```

- Which model performed the best?
- Do the results change each time you run the cell?
- Why do you think this is?

Due to the randomness of how each model finds patterns in the data, you might notice different results each time.

Without manually setting the random state using the `random_state` parameter of some models or using a NumPy random seed, every time you run the cell, you'll get slightly different results.

Let's see this in effect by running the same code as the cell above, except this time setting a NumPy random seed equal to 42.

In [27]:

```python
# Run the same code as the cell above, except this time set a NumPy random seed
# equal to 42
np.random.seed(42)

for model_name, model in models.items():
    model.fit(X_train, y_train)
    results[model_name] = model.score(X_test, y_test)

results
```

```
D:\ds_and_ml_projects\sample_project\env\lib\site-packages\sklearn\svm\_base.py:985: ConvergenceWa
rning: Liblinear failed to converge, increase the number of iterations.
  warnings.warn("Liblinear failed to converge, increase "
D:\ds_and_ml_projects\sample_project\env\lib\site-packages\sklearn\linear_model\_logistic.py:763:
ConvergenceWarning: lbfgs failed to converge (status=1):
STOP: TOTAL NO. of ITERATIONS REACHED LIMIT.

Increase the number of iterations (max_iter) or scale the data as shown in:
    https://scikit-learn.org/stable/modules/preprocessing.html
Please also refer to the documentation for alternative solver options:
    https://scikit-learn.org/stable/modules/linear_model.html#logistic-regression
  n_iter_i = _check_optimize_result(
```

Out[27]:

```
{'LinearSVC': 0.8688524590163934,
 'KNN': 0.6885245901639344,
 'SVC': 0.7049180327868853,
 'LogisticRegression': 0.8852459016393442,
 'RandomForestClassifier': 0.8360655737704918}
```

- Run the cell above a few times, what do you notice about the results?
- Which model performs the best this time?
- What happens if you add a NumPy random seed to the cell where you called `train_test_split()` (towards the top of the notebook) and then rerun the cell above?

Let's make our results a little more visual.

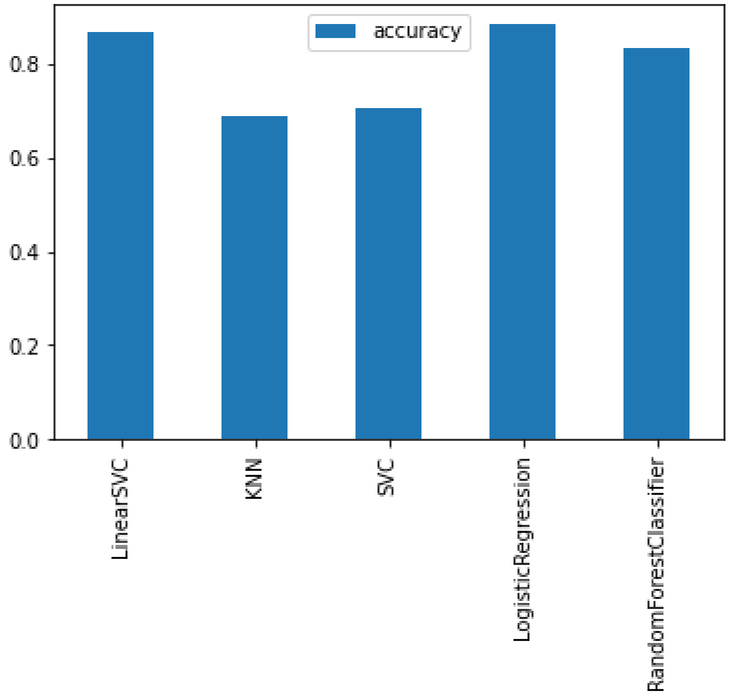
In [28]:

```python
# Create a pandas dataframe with the data as the values of the results dictionary,
# the index as the keys of the results dictionary and a single column called accuracy.
# Be sure to save the dataframe to a variable.
results_df = pd.DataFrame(results.values(),
                          results.keys(),
                          columns=["accuracy"])

# Create a bar plot of the results dataframe using plot.bar()
results_df.plot.bar()
```

Out[28]:

```
<AxesSubplot:>
```



Using `np.random.seed(42)` results in the `LogisticRegression` model perfoming the best (at least on my computer).

Let's tune its hyperparameters and see if we can improve it.

Hyperparameter Tuning

Remember, if you're ever trying to tune a machine learning models hyperparameters and you're not sure where to start, you can always search something like "MODEL_NAME hyperparameter tuning".

In the case of LogisticRegression, you might come across articles, such as Hyperparameter Tuning Using Grid Search by Chris Albon.

The article uses `GridSearchCV` but we're going to be using `RandomizedSearchCV`.

The different hyperparameters to search over have been setup for you in `log_reg_grid` but feel free to change them.

In [29]:

```
# Different LogisticRegression hyperparameters
log_reg_grid = {"C": np.logspace(-4, 4, 20),
                "solver": ["liblinear"]}
```

Since we've got a set of hyperparameters we can import `RandomizedSearchCV`, pass it our dictionary of hyperparameters and let it search for the best combination.

In [30]:

```
# Setup np random seed of 42
np.random.seed(42)

# Import RandomizedSearchCV from sklearn's model_selection module
from sklearn.model_selection import RandomizedSearchCV

# Setup an instance of RandomizedSearchCV with a LogisticRegression() estimator,
# our log_reg_grid as the param_distributions, a cv of 5 and n_iter of 5.
rs_log_reg = RandomizedSearchCV(estimator=LogisticRegression(),
                                param_distributions=log_reg_grid,
                                cv=5,
                                n_iter=5,
                                verbose=2)

# Fit the instance of RandomizedSearchCV
rs_log_reg.fit(X_train, y_train)
```

```
Fitting 5 folds for each of 5 candidates, totalling 25 fits
[CV] END .........................C=0.0001, solver=liblinear; total time=   0.0s
[CV] END .........................C=0.0001, solver=liblinear; total time=   0.0s
[CV] END .........................C=0.0001, solver=liblinear; total time=   0.0s
[CV] END .........................C=0.0001, solver=liblinear; total time=   0.0s
[CV] END .........................C=0.0001, solver=liblinear; total time=   0.0s
[CV] END ...............C=1438.44988828766, solver=liblinear; total time=   0.0s
[CV] END ...............C=1438.44988828766, solver=liblinear; total time=   0.0s
[CV] END ...............C=1438.44988828766, solver=liblinear; total time=   0.0s
[CV] END ...............C=1438.44988828766, solver=liblinear; total time=   0.0s
[CV] END ...............C=1438.44988828766, solver=liblinear; total time=   0.0s
[CV] END ...............C=206.913808111479, solver=liblinear; total time=   0.0s
[CV] END ...............C=206.913808111479, solver=liblinear; total time=   0.0s
[CV] END ...............C=206.913808111479, solver=liblinear; total time=   0.0s
[CV] END ...............C=206.913808111479, solver=liblinear; total time=   0.0s
[CV] END ...............C=206.913808111479, solver=liblinear; total time=   0.0s
[CV] END .........C=0.00026366508987303583, solver=liblinear; total time=   0.0s
[CV] END .........C=0.00026366508987303583, solver=liblinear; total time=   0.0s
[CV] END .........C=0.00026366508987303583, solver=liblinear; total time=   0.0s
[CV] END .........C=0.00026366508987303583, solver=liblinear; total time=   0.0s
[CV] END .........C=0.00026366508987303583, solver=liblinear; total time=   0.0s
[CV] END ............C=0.23357214690901212, solver=liblinear; total time=   0.0s
[CV] END ............C=0.23357214690901212, solver=liblinear; total time=   0.0s
[CV] END ............C=0.23357214690901212, solver=liblinear; total time=   0.0s
[CV] END ............C=0.23357214690901212, solver=liblinear; total time=   0.0s
[CV] END ............C=0.23357214690901212, solver=liblinear; total time=   0.0s
```

Out[30]:

```
RandomizedSearchCV(cv=5, estimator=LogisticRegression(), n_iter=5,
                   param_distributions={'C': array([1.00000000e-04, 2.63665090e-04, 6.95192796e-0
4, 1.83298071e-03,
       4.83293024e-03, 1.27427499e-02, 3.35981829e-02, 8.85866790e-02,
       2.33572147e-01, 6.15848211e-01, 1.62377674e+00, 4.28133240e+00,
       1.12883789e+01, 2.97635144e+01, 7.84759970e+01, 2.06913808e+02,
       5.45559478e+02, 1.43844989e+03, 3.79269019e+03, 1.00000000e+04]),
                                        'solver': ['liblinear']},
                   verbose=2)
```

Once `RandomizedSearchCV` has finished, we can find the best hyperparmeters it found using the `best_params_` attributes.

In [31]:

```
# Find the best parameters of the RandomizedSearchCV instance using the best_params_ attribute
rs_log_reg.best_params_
```

Out[31]:

```
{'solver': 'liblinear', 'C': 0.23357214690901212}
```

In [32]:

```
# Score the instance of RandomizedSearchCV using the test data
rs_log_reg.score(X_test, y_test)
```

Out[32]:

```
0.8852459016393442
```

After hyperparameter tuning, did the models score improve? What else could you try to improve it? Are there any other methods of hyperparameter tuning you can find for `LogisticRegression` ?

Classifier Model Evaluation

We've tried to find the best hyperparameters on our model using `RandomizedSearchCV` and so far we've only been evaluating our model using the `score()` function which returns accuracy.

But when it comes to classification, you'll likely want to use a few more evaluation metrics, including:

- **Confusion matrix** - Compares the predicted values with the true values in a tabular way, if 100% correct, all values in the matrix will be top left to bottom right (diagnol line).
- **Cross-validation** - Splits your dataset into multiple parts and train and tests your model on each part and evaluates performance as an average.
- **Precision** - Proportion of true positives over total number of samples. Higher precision leads to less false positives.
- **Recall** - Proportion of true positives over total number of true positives and false positives. Higher recall leads to less false negatives.
- **F1 score** - Combines precision and recall into one metric. 1 is best, 0 is worst.
- **Classification report** - Sklearn has a built-in function called `classification_report()` which returns some of the main classification metrics such as precision, recall and f1-score.
- **ROC Curve** - Receiver Operating Characterisitc is a plot of true positive rate versus false positive rate.
- **Area Under Curve (AUC)** - The area underneath the ROC curve. A perfect model achieves a score of 1.0.

Before we get to these, we'll instantiate a new instance of our model using the best hyerparameters found by `RandomizedSearchCV`.

In [33]:

```
# Instantiate a LogisticRegression classifier using the best hyperparameters from RandomizedSearch
clf = LogisticRegression(C=0.23357214690901212, solver='liblinear')

# Fit the new instance of LogisticRegression with the best hyperparameters on the training data
clf.fit(X_train, y_train)
```

Out[33]:

```
LogisticRegression(C=0.23357214690901212, solver='liblinear')
```

Now it's to import the relative Scikit-Learn methods for each of the classification evaluation metrics we're after.

In [35]:

```
# Import confusion_matrix and classification_report from sklearn's metrics module
from sklearn.metrics import confusion_matrix, classification_report

# Import precision_score, recall_score and f1_score from sklearn's metrics module
from sklearn.metrics import precision_score, recall_score, f1_score

# Import plot_roc_curve from sklearn's metrics module
from sklearn.metrics import plot_roc_curve
```

Evaluation metrics are very often comparing a model's predictions to some ground truth labels.

Let's make some predictions on the test data using our latest model and save them to `y_preds`.

In [36]:

```
# Make predictions on test data and save them
y_preds = clf.predict(X_test)
```

Time to use the predictions our model has made to evaluate it beyond accuracy.

In [37]:

```
# Create a confusion matrix using the confusion_matrix function
con_mat = confusion_matrix(y_test, y_preds)
```

Challenge: The in-built `confusion_matrix` function in Scikit-Learn produces something not too visual, how could you make your confusion matrix more visual?

You might want to search something like "how to plot a confusion matrix". Note: There may be more than one way to do this.

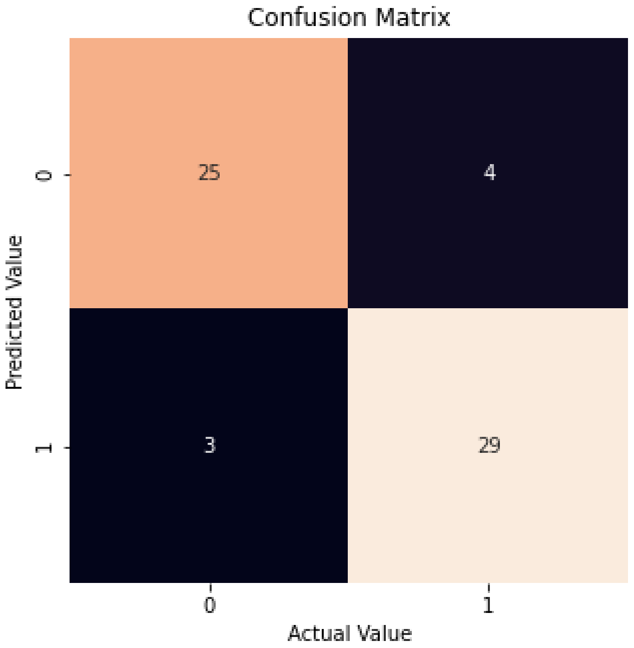
In [43]:

```
# Create a more visual confusion matrix
import seaborn as sbn

fig, ax = plt.subplots(figsize=(5,5))
ax = sbn.heatmap(con_mat, annot=True, cbar=False)
ax.set(title="Confusion Matrix",
      xlabel="Actual Value",
      ylabel="Predicted Value")
```

Out[43]:

```
[Text(0.5, 1.0, 'Confusion Matrix'),
 Text(0.5, 24.0, 'Actual Value'),
 Text(24.0, 0.5, 'Predicted Value')]
```



How about a classification report?

In [45]:

```
# Create a classification report using the classification_report function
clf_rep = classification_report(y_test, y_preds)
clf_rep
```

Out[45]:

```
'              precision    recall  f1-score   support\n\n           0       0.89      0.86
0.88        29\n           1       0.88      0.91      0.89        32\n\n    accuracy
0.89        61\n   macro avg       0.89      0.88      0.88        61\nweighted avg       0.89
0.89      0.89        61\n'
```

Challenge: Write down what each of the columns in this classification report are.

- **Precision** - Indicates the proportion of positive identifications (model predicted class 1) which were actually correct. A model which produces no false positives has a precision of 1.0.
- **Recall** - Indicates the proportion of actual positives which were correctly classified. A model which produces no false negatives has a recall of 1.0.
- **F1 score** - A combination of precision and recall. A perfect model achieves an F1 score of 1.0.

- **Support** - The number of samples each metric was calculated on.
- **Accuracy** - The accuracy of the model in decimal form. Perfect accuracy is equal to 1.0.
- **Macro avg** - Short for macro average, the average precision, recall and F1 score between classes. Macro avg doesn't class imbalance into effort, so if you do have class imbalances, pay attention to this metric.
- **Weighted avg** - Short for weighted average, the weighted average precision, recall and F1 score between classes. Weighted means each metric is calculated with respect to how many samples there are in each class. This metric will favour the majority class (e.g. will give a high value when one class out performs another due to having more samples).

The classification report gives us a range of values for precision, recall and F1 score, time to find these metrics using Scikit-Learn functions.

In [46]:

```
# Find the precision score of the model using precision_score()
precision_score(y_test, y_preds)
```

Out[46]:

```
0.8787878787878788
```

In [47]:

```
# Find the recall score
recall_score(y_test, y_preds)
```

Out[47]:

```
0.90625
```

In [48]:

```
# Find the F1 score
f1_score(y_test, y_preds)
```

Out[48]:

```
0.8923076923076922
```

Confusion matrix: done. Classification report: done. ROC (receiver operator characteristic) curve & AUC (area under curve) score: not done.

Let's fix this.

If you're unfamiliar with what a ROC curve, that's your first challenge, to read up on what one is.

In a sentence, a ROC curve is a plot of the true positive rate versus the false positive rate.

And the AUC score is the area behind the ROC curve.

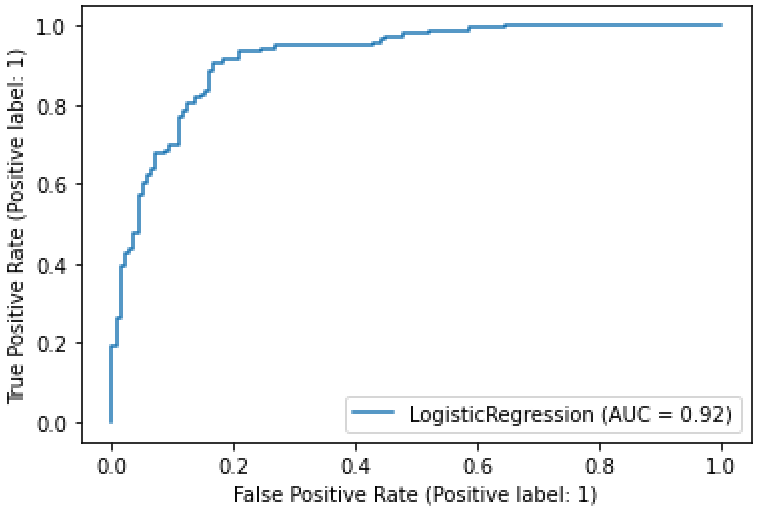
Scikit-Learn provides a handy function for creating both of these called `plot_roc_curve()`.

In [49]:

```
# Plot a ROC curve using our current machine learning model using plot_roc_curve
plot_roc_curve(clf, X, y)
```

Out[49]:

```
<sklearn.metrics._plot.roc_curve.RocCurveDisplay at 0x17a38665d00>
```



Beautiful! We've gone far beyond accuracy with a plethora extra classification evaluation metrics.

If you're not sure about any of these, don't worry, they can take a while to understand. That could be an optional extension, reading up on a classification metric you're not sure of.

The thing to note here is all of these metrics have been calculated using a single training set and a single test set. Whilst this is okay, a more robust way is to calculate them using cross-validation.

We can calculate various evaluation metrics using cross-validation using Scikit-Learn's `cross_val_score()` function along with the `scoring` parameter.

In [50]:

```
# Import cross_val_score from sklearn's model_selection module
from sklearn.model_selection import cross_val_score
```

In [51]:

```
# EXAMPLE: By default cross_val_score returns 5 values (cv=5).
cross_val_score(clf,
                X,
                y,
                scoring="accuracy",
                cv=5)
```

Out[51]:

```
array([0.81967213, 0.90163934, 0.8852459 , 0.88333333, 0.75      ])
```

In [52]:

```
# EXAMPLE: Taking the mean of the returned values from cross_val_score
# gives a cross-validated version of the scoring metric.
cross_val_acc = np.mean(cross_val_score(clf,
                                        X,
                                        y,
                                        scoring="accuracy",
                                        cv=5))

cross_val_acc
```

Out[52]:

```
0.8479781420765027
```

In the examples, the cross-validated accuracy is found by taking the mean of the array returned by `cross_val_score()`.

Now it's time to find the same for precision, recall and F1 score.

In [55]:

```
# Find the cross-validated precision
cross_val_prec = np.mean(cross_val_score(clf, X, y, scoring="precision"))
cross_val_prec
```

Out[55]:

```
0.8215873015873015
```

In [56]:

```
# Find the cross-validated recall
cross_val_rec = np.mean(cross_val_score(clf, X, y, scoring="recall"))
cross_val_rec
```

Out[56]:

```
0.9272727272727274
```

In [57]:

```
# Find the cross-validated F1 score
cross_val_f1 = np.mean(cross_val_score(clf, X, y, scoring="f1"))
cross_val_f1
```

Out[57]:

```
0.8705403543192143
```

Exporting and importing a trained model

Once you've trained a model, you may want to export it and save it to file so you can share it or use it elsewhere.

One method of exporting and importing models is using the joblib library.

In Scikit-Learn, exporting and importing a trained model is known as model persistence.

In [58]:

```
# Import the dump and load functions from the joblib library
from joblib import dump, load
```

In [59]:

```
# Use the dump function to export the trained model to file
dump(clf, filename="logistic-regression-classifier.joblib")
```

Out[59]:

```
['logistic-regression-classifier.joblib']
```

In [61]:

```
# Use the load function to import the trained model you just exported
# Save it to a different variable name to the origial trained model
loaded_logistic = load("logistic-regression-classifier.joblib")

# Evaluate the loaded trained model on the test data
y_preds_clf = loaded_logistic.predict(X_test)
classification_report(y_test, y_preds_clf)
```

Out[61]:

```
'              precision    recall  f1-score   support\n\n           0       0.89      0.86
0.88        29\n           1       0.88      0.91      0.89        32\n\n    accuracy
0.89        61\n   macro avg       0.89      0.88      0.88        61\nweighted avg       0.89
0.89      0.89        61\n'
```

What do you notice about the loaded trained model results versus the original (pre-exported) model results?

Scikit-Learn Regression Practice

For the next few exercises, we're going to be working on a regression problem, in other words, using some data to predict a number.

Our dataset is a table of car sales, containing different car characteristics as well as a sale price.

We'll use Scikit-Learn's built-in regression machine learning models to try and learn the patterns in the car characteristics and their prices on a certain group of the dataset before trying to predict the sale price of a group of cars the model has never seen before.

To begin, we'll import the data from GitHub into a pandas DataFrame, check out some details about it and try to build a model as soon as possible.

In [62]:

```
# Read in the car sales data
car_sales = pd.read_csv("https://raw.githubusercontent.com/mrdbourke/zero-to-mastery-ml/master/dat

# View the first 5 rows of the car sales data
car_sales.head()
```

Out[62]:

	Make	Colour	Odometer (KM)	Doors	Price
0	Honda	White	35431.0	4.0	15323.0
1	BMW	Blue	192714.0	5.0	19943.0
2	Honda	White	84714.0	4.0	28343.0
3	Toyota	White	154365.0	4.0	13434.0
4	Nissan	Blue	181577.0	3.0	14043.0

In [63]:

```
# Get information about the car sales DataFrame
car_sales.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 1000 entries, 0 to 999
Data columns (total 5 columns):
 #   Column         Non-Null Count  Dtype  
---  ------         --------------  -----  
 0   Make           951 non-null    object 
 1   Colour         950 non-null    object 
 2   Odometer (KM)  950 non-null    float64
 3   Doors          950 non-null    float64
 4   Price          950 non-null    float64
dtypes: float64(3), object(2)
memory usage: 39.2+ KB
```

Looking at the output of `info()`,

- How many rows are there total?
- What datatypes are in each column?
- How many missing values are there in each column?

In [64]:

```
# Find number of missing values in each column
car_sales.isna().sum()
```

Out[64]:

```
Make             49
Colour           50
Odometer (KM)    50
```

```
Doors            50
Price            50
dtype: int64
```

In [65]:

```python
# Find the datatypes of each column of car_sales
car_sales.dtypes
```

Out[65]:

```
Make              object
Colour            object
Odometer (KM)    float64
Doors            float64
Price            float64
dtype: object
```

Knowing this information, what would happen if we tried to model our data as it is?

Let's see.

In [67]:

```python
# EXAMPLE: This doesn't work because our car_sales data isn't all numerical
from sklearn.ensemble import RandomForestRegressor
car_sales_X, car_sales_y = car_sales.drop("Price", axis=1), car_sales.Price
# rf_regressor = RandomForestRegressor().fit(car_sales_X, car_sales_y)
```

As we see, the cell above breaks because our data contains non-numerical values as well as missing data.

To take care of some of the missing data, we'll remove the rows which have no labels (all the rows with missing values in the `Price` column).

In [69]:

```python
# Remove rows with no labels (NaN's in the Price column)
car_sales.dropna(subset=["Price"], inplace=True)
```

Building a pipeline

Since our `car_sales` data has missing numerical values as well as the data isn't all numerical, we'll have to fix these things before we can fit a machine learning model on it.

There are ways we could do this with pandas but since we're practicing Scikit-Learn, we'll see how we might do it with the `Pipeline` class.

Because we're modifying columns in our dataframe (filling missing values, converting non-numerical data to numbers) we'll need the `ColumnTransformer`, `SimpleImputer` and `OneHotEncoder` classes as well.

Finally, because we'll need to split our data into training and test sets, we'll import `train_test_split` as well.

In [70]:

```python
# Import Pipeline from sklearn's pipeline module
from sklearn.pipeline import Pipeline

# Import ColumnTransformer from sklearn's compose module
from sklearn.compose import ColumnTransformer

# Import SimpleImputer from sklearn's impute module
from sklearn.impute import SimpleImputer

# Import OneHotEncoder from sklearn's preprocessing module
from sklearn.preprocessing import OneHotEncoder
```

```
# Import train_test_split from sklearn's model_selection module
from sklearn.model_selection import train_test_split
```

Now we've got the necessary tools we need to create our preprocessing `Pipeline` which fills missing values along with turning all non-numerical data into numbers.

Let's start with the categorical features.

In [71]:

```
# Define different categorical features
categorical_features = ["Make", "Colour"]

# Create categorical transformer Pipeline
categorical_transformer = Pipeline(steps=[
    # Set SimpleImputer strategy to "constant" and fill value to "missing"
    ("imputer", SimpleImputer(strategy="constant", fill_value="missing")),
    # Set OneHotEncoder to ignore the unknowns
    ("onehot", OneHotEncoder(handle_unknown="ignore"))])
```

It would be safe to treat `Doors` as a categorical feature as well, however since we know the vast majority of cars have 4 doors, we'll impute the missing `Doors` values as 4.

In [72]:

```
# Define Doors features
door_feature = ["Doors"]

# Create Doors transformer Pipeline
door_transformer = Pipeline(steps=[
    # Set SimpleImputer strategy to "constant" and fill value to 4
    ("imputer", SimpleImputer(strategy="constant", fill_value=4))])
```

Now onto the numeric features. In this case, the only numeric feature is the `Odometer (KM)` column. Let's fill its missing values with the median.

In [73]:

```
# Define numeric features (only the Odometer (KM) column)
numeric_features = ["Odometer (KM)"]

# Crearte numeric transformer Pipeline
numeric_transformer = Pipeline(steps=[
    # Set SimpleImputer strategy to fill missing values with the "Median"
    ("imputer", SimpleImputer(strategy="median"))])
```

Time to put all of our individual transformer `Pipeline`'s into a single `ColumnTransformer` instance.

In [74]:

```
# Setup preprocessing steps (fill missing values, then convert to numbers)
preprocessor = ColumnTransformer(
    transformers=[
        # Use the categorical_transformer to transform the categorical_features
        ("cat", categorical_transformer, categorical_features),
        # Use the door_transformer to transform the door_feature
        ("door", door_transformer, door_feature),
        # Use the numeric_transformer to transform the numeric_features
        ("num", numeric_transformer, numeric_features)])
```

Boom! Now our `preprocessor` is ready, time to import some regression models to try out.

Comparing our data to the Scikit-Learn machine learning map, we can see there's a handful of different regression models we can try.

- RidgeRegression
- SVR(kernel="linear") - short for Support Vector Regressor, a form form of support vector machine.
- SVR(kernel="rbf") - short for Support Vector Regressor, a form of support vector machine.
- RandomForestRegressor - the regression version of RandomForestClassifier.

In [75]:

```
# Import Ridge from sklearn's linear_model module
from sklearn.linear_model import Ridge

# Import SVR from sklearn's svm module
from sklearn.svm import SVR

# Import RandomForestRegressor from sklearn's ensemble module
from sklearn.ensemble import RandomForestRegressor
```

Again, thanks to the design of the Scikit-Learn library, we're able to use very similar code for each of these models.

To test them all, we'll create a dictionary of regression models and an empty dictionary for regression model results.

In [78]:

```
# Create dictionary of model instances, there should be 4 total key, value pairs
# in the form {"model_name": model_instance}.
# Don't forget there's two versions of SVR, one with a "linear" kernel and the
# other with kernel set to "rbf".
regression_models = {"Ridge": Ridge(),
                     "SVR_linear": SVR(kernel="linear"),
                     "SVR_rbf": SVR(),
                     "RandomForestRegressor": RandomForestRegressor()}

# Create an empty dictionary for the regression results
regression_results = {}
```

Our regression model dictionary is prepared as well as an empty dictionary to append results to, time to get the data split into `X` (feature variables) and `y` (target variable) as well as training and test sets.

In our car sales problem, we're trying to use the different characteristics of a car (`X`) to predict its sale price (`y`).

In [79]:

```
# Create car sales X data (every column of car_sales except Price)
car_sales_X = car_sales.drop("Price", axis=1)

# Create car sales y data (the Price column of car_sales)
car_sales_y = car_sales["Price"]
```

In [80]:

```
# Use train_test_split to split the car_sales_X and car_sales_y data into
# training and test sets.
# Give the test set 20% of the data using the test_size parameter.
# For reproducibility set the random_state parameter to 42.
car_X_train, car_X_test, car_y_train, car_y_test = train_test_split(car_sales_X,
                                                                    car_sales_y,
                                                                    test_size=0.2,
                                                                    random_state=42)

# Check the shapes of the training and test datasets
car_X_train.shape, car_X_test.shape, car_y_train.shape, car_y_test.shape
```

Out[80]:

```
((760, 4), (190, 4), (760,), (190,))
```

- How many rows are in each set?
- How many columns are in each set?

Alright, our data is split into training and test sets, time to build a small loop which is going to:

1. Go through our `regression_models` dictionary
2. Create a `Pipeline` which contains our `preprocessor` as well as one of the models in the dictionary
3. Fits the `Pipeline` to the car sales training data
4. Evaluates the target model on the car sales test data and appends the results to our `regression_results` dictionary

In [81]:

```python
# Loop through the items in the regression_models dictionary
for model_name, model in regression_models.items():

    # Create a model Pipeline with a preprocessor step and model step
    model_pipeline = Pipeline(steps=[("preprocessor", preprocessor),
                                      ("model", model)])

    # Fit the model Pipeline to the car sales training data
    print(f"Fitting {model_name}...")
    model_pipeline.fit(car_X_train, car_y_train)

    # Score the model Pipeline on the test data appending the model_name to the
    # results dictionary
    print(f"Scoring {model_name}...")
    regression_results[model_name] = model_pipeline.score(car_X_test,
                                                          car_y_test)
```

```
Fitting Ridge...
Scoring Ridge...
Fitting SVR_linear...
Scoring SVR_linear...
Fitting SVR_rbf...
Scoring SVR_rbf...
Fitting RandomForestRegressor...
Scoring RandomForestRegressor...
```

Our regression models have been fit, let's see how they did!

In [82]:

```python
# Check the results of each regression model by printing the regression_results
# dictionary
regression_results
```

Out[82]:

```
{'Ridge': 0.254026110579439,
 'SVR_linear': -0.489452821008145,
 'SVR_rbf': 0.0018546241516633755,
 'RandomForestRegressor': 0.22466337002302106}
```

- Which model did the best?
- How could you improve its results?
- What metric does the `score()` method of a regression model return by default?

Since we've fitted some models but only compared them via the default metric contained in the `score()` method (R^2 score or coefficient of determination), let's take the `RidgeRegression` model and evaluate it with a few other regression metrics.

Specifically, let's find:

1. **R^2 (pronounced r-squared) or coefficient of determination** - Compares your models predictions to the mean of the targets. Values can range from negative infinity (a very poor model) to 1. For example, if all your model does is predict the mean of the targets, its R^2 value would be 0. And if your model perfectly predicts a range of numbers it's R^2 value would be 1.
2. **Mean absolute error (MAE)** - The average of the absolute differences between predictions and actual values. It gives you an idea of how wrong your predictions were.
3. **Mean squared error (MSE)** - The average squared differences between predictions and actual values. Squaring the errors removes negative errors. It also amplifies outliers (samples which have larger errors).

Scikit-Learn has a few classes built-in which are going to help us with these, namely, `mean_absolute_error`, `mean_squared_error` and `r2_score`.

In [83]:

```
# Import mean_absolute_error from sklearn's metrics module
from sklearn.metrics import mean_absolute_error

# Import mean_squared_error from sklearn's metrics module
from sklearn.metrics import mean_squared_error

# Import r2_score from sklearn's metrics module
from sklearn.metrics import r2_score
```

All the evaluation metrics we're concerned with compare a model's predictions with the ground truth labels. Knowing this, we'll have to make some predictions.

Let's create a `Pipeline` with the `preprocessor` and a `Ridge()` model, fit it on the car sales training data and then make predictions on the car sales test data.

In [85]:

```
# Create RidgeRegression Pipeline with preprocessor as the "preprocessor" and
# Ridge() as the "model".
ridge_pipeline = Pipeline(steps=[("preprocessor", preprocessor),
                                 ("model", Ridge())])

# Fit the RidgeRegression Pipeline to the car sales training data
ridge_pipeline.fit(car_X_train, car_y_train)

# Make predictions on the car sales test data using the RidgeRegression Pipeline
car_y_preds = ridge_pipeline.predict(car_X_test)

# View the first 50 predictions
car_y_preds[:50]
```

Out[85]:

```
array([18514.480873  , 22204.86580267, 11045.72092225,  6891.87687957,
        8793.93970278, 10926.96786577, 15267.76229871, 13834.80312146,
       20207.44377898, 14412.59968701,  6216.01228306, 16543.06707068,
       11783.69374936, 13492.13284255, 14321.17899086, 16425.45131776,
       15999.88230172,  9925.04252351, 11576.64865889, 11580.55130633,
       10647.22402588, 13113.54464064, 17874.90033257, 23447.29246732,
       11797.33019118, 14481.85336505, 18430.527126  , 14680.33044208,
       20585.63901269, 19953.91395769, 18163.1170151 , 22366.43390174,
       12435.69707467, 14265.25495748, 18351.62656383, 19564.58267503,
       12213.05747919, 12479.20772529, 18677.93478691, 11284.89072638,
       15278.54001605, 17378.81425165, 19247.77623181, 17343.90692672,
       15048.12797569, 12724.44477165, 12389.86021752,  8475.63778401,
       15255.49324105, 18602.79970861])
```

Nice! Now we've got some predictions, time to evaluate them. We'll find the mean squared error (MSE), mean absolute error (MAE) and R^2 score (coefficient of determination) of our model.

In [86]:

```
# EXAMPLE: Find the MSE by comparing the car sales test labels to the car sales predictions
mse = mean_squared_error(car_y_test, car_y_preds)
# Return the MSE
mse
```

Out[86]:

```
49950182.63337458
```

In [87]:

```
# Find the MAE by comparing the car sales test labels to the car sales predictions
mae = mean_absolute_error(car_y_test, car_y_preds)
# Return the MAE
mae
```

Out[87]:

```
5713.8215208551555
```

In [88]:

```
# Find the R^2 score by comparing the car sales test labels to the car sales predictions
r2 = r2_score(car_y_test, car_y_preds)
# Return the R^2 score
r2
```

Out[88]:

```
0.254026110579439
```

Boom! Our model could potentially do with some hyperparameter tuning (this would be a great extension). And we could probably do with finding some more data on our problem, 1000 rows doesn't seem to be sufficient.

- How would you export the trained regression model?

Extensions

You should be proud. Getting this far means you've worked through a classification problem and regression problem using pure (mostly) Scikit-Learn (no easy feat!).

For more exercises, check out the Scikit-Learn getting started documentation. A good practice would be to read through it and for the parts you find interesting, add them into the end of this notebook.

Finally, as always, remember, the best way to learn something new is to try it. And try it relentlessly. If you're unsure of how to do something, never be afraid to ask a question or search for something such as, "how to tune the hyperparmaters of a scikit-learn ridge regression model".