Supervised Learning With Scikit-Learn

Machine Learning from labelled data to make predictions

Black Raven (James Ng)

29 Jun 2020 · 35 min read

This is a tutorial to share what I have learnt in Supervised Learning with scikit-learn, capturing the learning objectives as well as my personal notes. The course is taught by Hugo Bowne-Anderson from DataCamp, and it includes 4 chapters:

Chapter 1. Classification

Chapter 2. Regression

Chapter 3. Fine-tuning your model

Chapter 4. Preprocessing and pipelines

Photo by Andy Kelly on Unsplash

Is a particular email spam?

Will a tumor be benign or malignant?

Which of your customers will take their business elsewhere?

These questions can be answered by Machine learning algorithms, where computers learn from existing data to make predictions on new data.

In this course, you'll learn how to use Python to perform supervised learning, an essential component of machine learning. You'll learn how to build predictive models, tune their parameters, and determine how well they will perform with unseen data — all while using real world datasets. You'll be using scikit-learn, one of the most popular and user-friendly machine learning libraries for Python.

Chapter 1. Classification

In this chapter, you will be introduced to classification problems and learn how to solve them using supervised learning techniques. And you'll apply what you learn to a political dataset, where you classify the party affiliation of United States congressmen based on their voting records.

Supervised learning

Supervised learning is giving computers the ability to learn to make decisions from *labelled* data without being explicitly programmed. Example, to predict whether an email is spam or not (**classification**), or to predict life expectancy (**regression**).

Features = predictor variables = independent variables
Target variable = dependent variable = response variable

Unsupervised learning is used to uncover hidden patterns by **clustering**, using *unlabeled* data. Example, to cluster wikipedia entries into different categories, or to group customers into distinct categories based on purchasing behavior.

Reinforcement learning is when machines or software agents interact with an environment. Reinforcement agents optimise their behaviour given a system of rewards and punishments.

Which of these is a classification problem?

Once you decide to leverage supervised machine learning to solve a new problem, you need to identify whether your problem is better suited to classification or regression.

Provided below are 4 example applications of machine learning. Which of them is a supervised classification problem?

○ Using labeled financial data to predict whether the value of a stock will go up or go down next week.

○ Using labeled housing price data to predict the price of a new house based on various features.

○ Using unlabeled data to cluster the students of an online education company into different categories based on their learning styles.

○ Using labeled financial data to predict what the value of a stock will be next week.

Answer: Using labeled financial data to predict whether the value of a stock will go up or go down next week. There are two discrete, qualitative outcomes: the stock market going up, and the stock market going down. This can be represented using a binary variable, and is an application perfectly suited for classification.

Exploratory data analysis (EDA)

Explore the Iris dataset, explore flower characteristics (4 features) in columns and the 3 target species types. EDA can be visually displayed using `pd.plotting.scatter_matrix()` function.
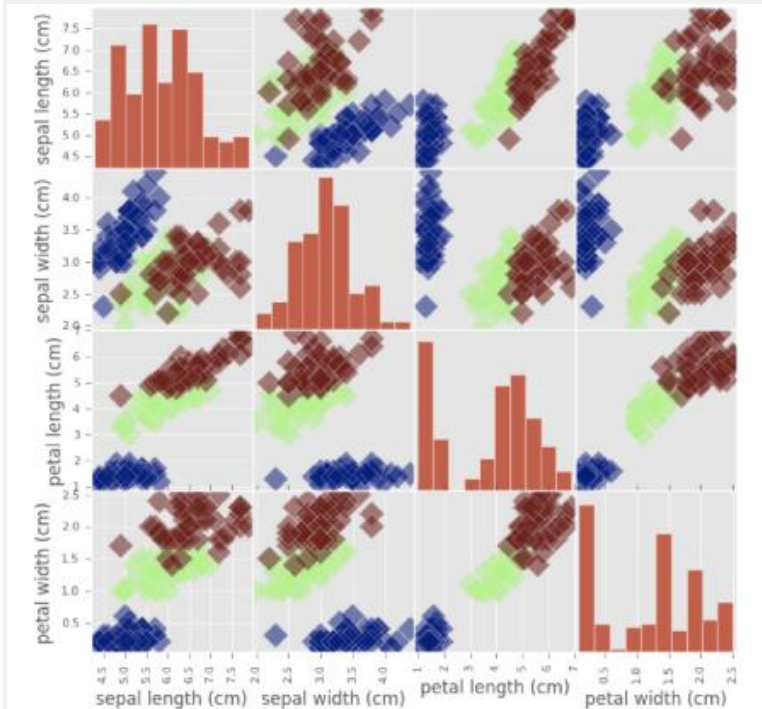
Numerical EDA

In this chapter, you'll be working with a dataset obtained from the **UCI Machine Learning Repository** consisting of votes made by US House of Representatives Congressmen. Your goal will be to predict their party affiliation ('Democrat' or 'Republican') based on how they voted on certain key issues. Here, it's worth noting that we have preprocessed this dataset to deal with missing values. This is so that your focus can be directed towards understanding how to train and evaluate supervised learning models. Once you

have mastered these fundamentals, you will be introduced to preprocessing techniques in Chapter 4 and have the chance to apply them there yourself — including on this very same dataset!

Before thinking about what supervised learning models you can apply to this, however, you need to perform Exploratory data analysis (EDA) in order to understand the structure of the data.

Get started with your EDA now by exploring this voting records dataset numerically. Use pandas' `.head()`, `.info()`, and `.describe()` methods on DataFrame `df`. Select the statement below that is **not** true.

○ The DataFrame has a total of `435` rows and `17` columns.

○ Except for `'party'`, all of the columns are of type `int64`.

○ The first two rows of the DataFrame consist of votes made by Republicans and the next three rows consist of votes made by Democrats.

○ There are 17 *predictor variables*, or *features*, in this DataFrame.

○ The target variable in this DataFrame is `'party'`.

Answer: There are 17 predictor variables, or features, in this DataFrame.

```
In [1]: df.info()
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 435 entries, 0 to 434
Data columns (total 17 columns):
party                435 non-null object
infants              435 non-null int64
water                435 non-null int64
budget               435 non-null int64
physician            435 non-null int64
salvador             435 non-null int64
religious            435 non-null int64
satellite            435 non-null int64
aid                  435 non-null int64
missile              435 non-null int64
immigration          435 non-null int64
synfuels             435 non-null int64
education            435 non-null int64
superfund            435 non-null int64
crime                435 non-null int64
duty_free_exports    435 non-null int64
eaa_rsa              435 non-null int64
dtypes: int64(16), object(1)
memory usage: 57.9+ KB

In [2]: df.head()
Out[2]:
        party  infants  water  budget  physician  salvador  religious  \
0  republican        0      1       0          1         1          1
1  republican        0      1       0          1         1          1
2    democrat        0      1       1          0         1          1
3    democrat        0      1       1          0         1          1
4    democrat        1      1       1          0         1          1
```

Observation: The number of columns in the DataFrame is not equal to the number of features. One of the columns `'party'` is the target variable.

Visual EDA

The Numerical EDA you did in the previous exercise gave you some very important information, such as the names and data types of the columns, and the dimensions of the DataFrame. Following this with some visual EDA will give you a better understanding of the data, using the `scatter_matrix()` function on the Iris data. However, you may have noticed in the previous exercise that all the features in this dataset are binary; that is, they are either 0 or 1. So a different type of plot would be more useful here, such as **Seaborn's** countplot.
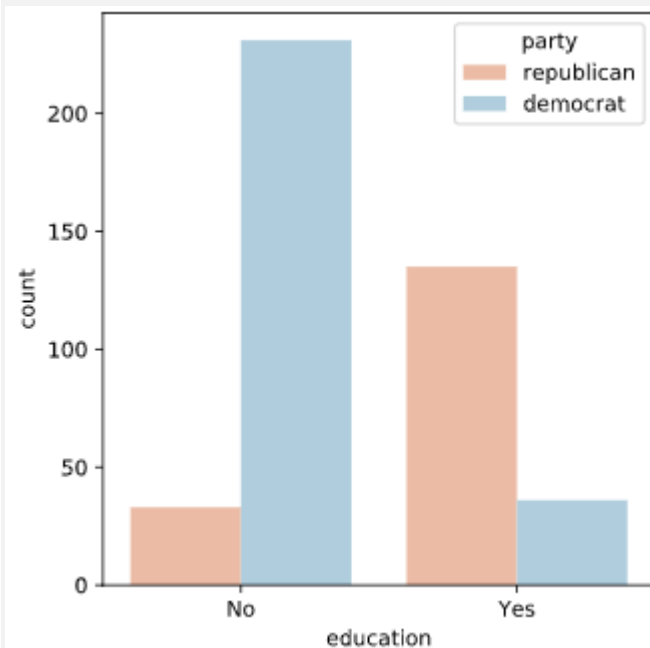
Given above is a `countplot` of the `'education'` bill, generated from the following code:

```
plt.figure()
sns.countplot(x='education', hue='party', data=df, palette='RdBu')
plt.xticks([0,1], ['No', 'Yes'])
plt.show()
```

In `sns.countplot()`, we specify the x-axis data to be `'education'`, and hue to be `'party'` (target variable). So the resulting plot shows the difference in voting behavior between the two parties for the `'education'` bill, with each party colored differently. We manually specified the color to be `'RdBu'`, as the Republican party has been traditionally associated with red, and the Democratic party with blue.

It seems like Democrats voted resoundingly *against* this bill, compared to Republicans. This is the kind of information that our machine learning model will seek to learn when we try to predict party affiliation solely based on voting behavior. An expert in U.S politics may be able to predict this without machine learning, but probably not instantaneously — and certainly not if we are dealing with hundreds of samples!

Explore the voting behavior further by generating countplots for the `'satellite'` and `'missile'` bills, and answer the following question: Of these two bills, for which ones do Democrats vote resoundingly in *favor* of, compared to Republicans?
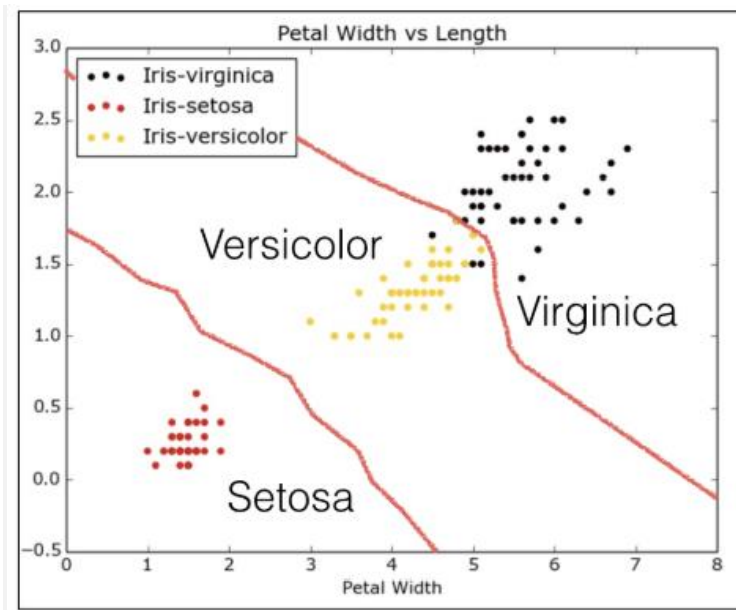
Answer: Both `'satellite'` and `'missile'`

```
plt.subplot(1,2,1)
sns.countplot(x='satellite', hue='party', data=df, palette='RdBu')
plt.xticks([0,1], ['No', 'Yes'])
plt.subplot(1,2,2)
sns.countplot(x='missile', hue='party', data=df, palette='RdBu')
plt.xticks([0,1], ['No', 'Yes'])
plt.show()
```
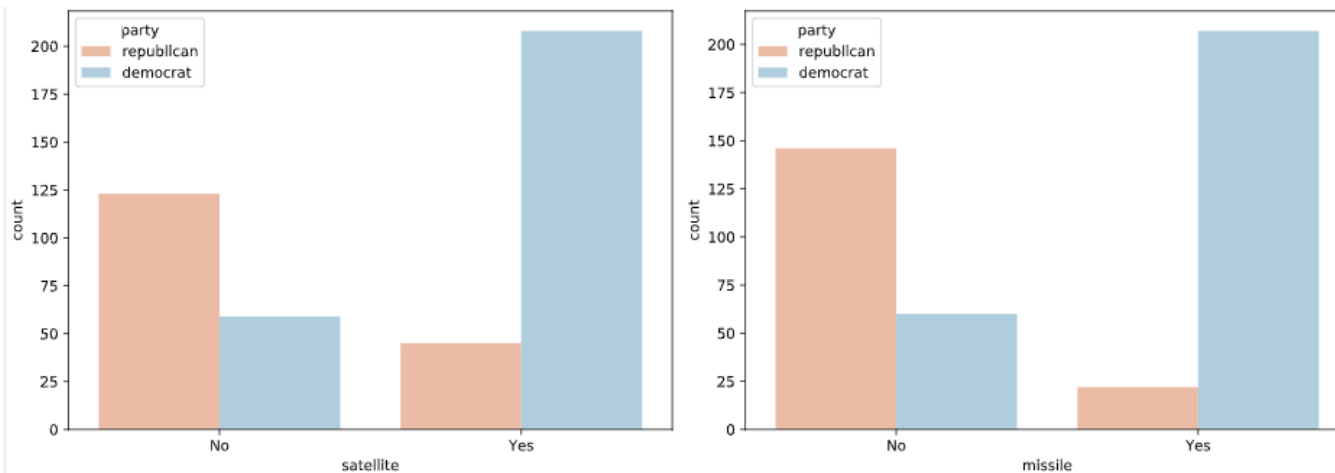
Observation: Democrats voted in favor of both `'satellite'` *and* `'missile'`

The classification challenge

Training a model on the data = 'fitting' a model to the data using `.fit()` method

To predict the labels of new data, use `.predict()` method

k–Nearest Neighbors: Fit

Having explored the Congressional voting records dataset, it is time now to build your first classifier. In this exercise, you will fit a k-Nearest Neighbors classifier to the voting dataset, loaded as DataFrame `df`.

Ensure your data adheres to the format required by the scikit-learn API. The features need to be in an array where each column is a feature, and each row a different observation or data point — in this case, a Congressman's voting record. The target needs to be a single column with the same number of observations as the feature data. We named the feature array `X` and target `y`, in accordance with the common scikit-learn practice.

Your job is to create an instance of a k-NN classifier with 6 neighbors (specifying parameter `n_neighbors=6`) and then fit it to the data.

```
# Import KNeighborsClassifier from sklearn.neighbors
from sklearn.neighbors import KNeighborsClassifier

# Create arrays for the features and the response variable
y = df['party'].values
X = df.drop('party', axis=1).values

# Create a k-NN classifier with 6 neighbors
knn = KNeighborsClassifier(n_neighbors=6)

# Fit the classifier to the data
knn.fit(X,y)
```

Now that your k-NN classifier with 6 neighbors has been fit to the data, it can be used to predict the labels of new data points.

k-Nearest Neighbors: Predict

Having fit a k-NN classifier, you can now use it to predict the label of a new data point. However, there is no unlabeled data available since all of it was used to fit the model! You can still use the `.predict()` method on the `x` that was used to fit the model, but it is not a good indicator of the model's ability to generalize to new, unseen data. A solution to this problem is the next exercises.

To simulate unseen data for this exercise, a data point without label has been randomly generated as `x_new`. You will use your classifier to predict the label for this 1 new data point, as well as on the training data `x` that the model has already seen. Using `.predict()` on `X_new` will generate 1 prediction, while using it on `x` will generate 435 predictions (1 for each sample).

```python
# Import KNeighborsClassifier from sklearn.neighbors
from sklearn.neighbors import KNeighborsClassifier

# Create arrays for the features and the response variable
y = df['party'].values
X = df.drop('party', axis=1).values

# Create a k-NN classifier with 6 neighbors: knn
knn = KNeighborsClassifier(n_neighbors=6)

# Fit the classifier to the data
knn.fit(X,y)

# Predict the labels for the training data X
y_pred = knn.predict(X)

# Predict and print the label for the new data point X_new
new_prediction = knn.predict(X_new)
print("Prediction: {}".format(new_prediction))<script.py> output:
    Prediction: ['democrat']
```
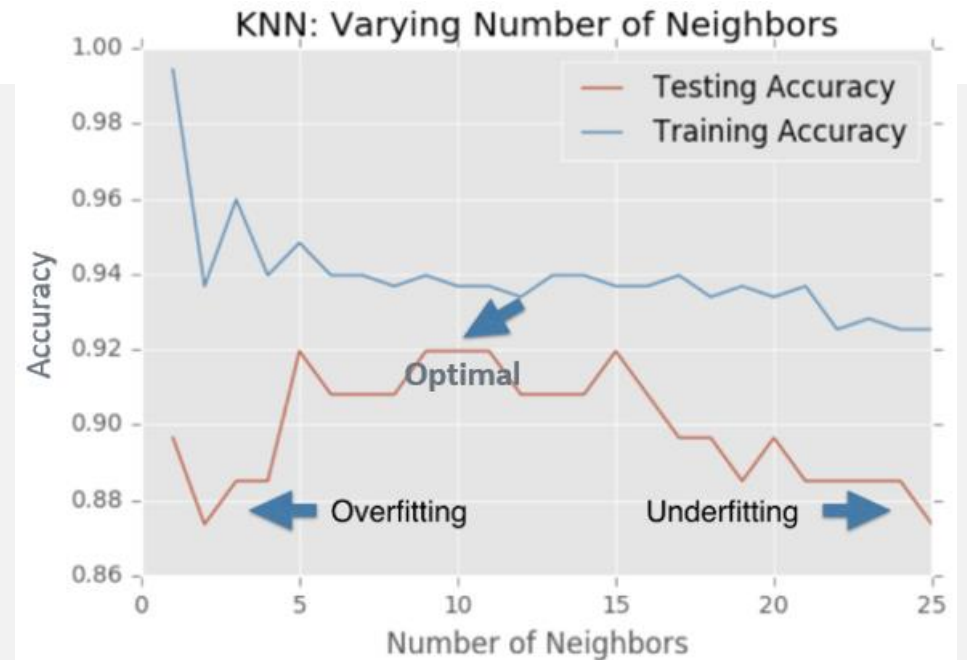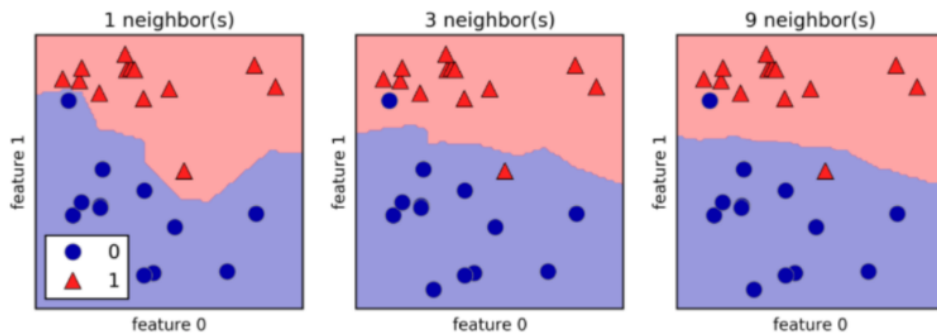
Observation: The model predicts 'democrat', but there is no indication of its accuracy. The next section, we will learn how to measure the model's prediction performance.

Measuring model performance

Accuracy = fraction of correct predictions.

The digits recognition dataset

Up until now, you have been performing binary classification, since the target variable had two possible outcomes. In the following exercises, you'll be working with the **MNIST** digits recognition dataset, which has 10 classes, the digits 0 through 9! A reduced version of the MNIST dataset is one of scikit-learn's included datasets, to be use in this exercise.

Each sample in this scikit-learn dataset is an 8x8 image representing a handwritten digit. Each pixel is represented by an integer in the range 0 to 16, indicating varying levels of black. Scikit-learn's built-in datasets are of type `Bunch`, which are dictionary-like objects. Helpfully for the MNIST dataset, scikit-learn provides an `'images'` key in addition to the `'data'` and `'target'` keys that you have seen with the Iris data. Because it is a 2D array of the images corresponding to each sample, this `'images'` key is useful for visualizing the images, as you'll see in this exercise. On the other hand, the `'data'` key contains the feature array - that is, the images as a flattened array of 64 pixels.

You can access the keys of these `Bunch` objects in two different ways: By using the `.` notation, as in `digits.images`, or the `[]` notation, as in `digits['images']`.

```
# Import necessary modules
from sklearn import datasets
import matplotlib.pyplot as plt

# Load the digits dataset: digits
digits = datasets.load_digits()

# Print the keys and DESCR of the dataset
print(digits.keys())
print(digits.DESCR)

# Print the shape of the images and data keys
print(digits.images.shape)
print(digits.data.shape)

# Display 1011th image, digit index 1010
plt.imshow(digits.images[1010], cmap=plt.cm.gray_r, interpolation='nearest')
plt.show()<script.py> output:
    dict_keys(['data', 'target', 'target_names', 'images', 'DESCR'])

#truncated
    (1797, 8, 8)
    (1797, 64)
```
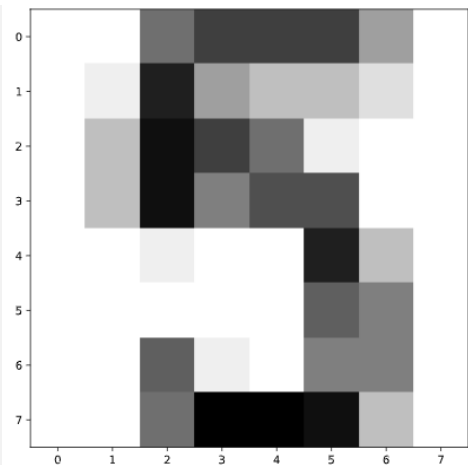
It looks like the image in question corresponds to the digit '5'. In the next exercise, you can build a classifier that can make this prediction not only for this image, but for all the other ones in the dataset.

Train/Test Split + Fit/Predict/Accuracy

Now that you have learned about the importance of splitting your data into training and test sets, it's time to practice doing this on the digits dataset! After creating arrays for the features and target variable, you will split them into training and test sets, fit a k-NN classifier to the training data, and then compute its accuracy using the `.score()` method.

```
# Import necessary modules
from sklearn.neighbors import KNeighborsClassifier
from sklearn.model_selection import train_test_split

# Create feature and target arrays
X = digits.data
y = digits.target

# Split into training and test set
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size = 0.2, random_state=42, stratify=y)
```

```
# Create a k-NN classifier with 7 neighbors: knn
knn = KNeighborsClassifier(n_neighbors=7)

# Fit the classifier to the training data
knn.fit(X_train, y_train)

# Print the accuracy
print(knn.score(X_test, y_test))<script.py> output:
    0.983333333333333
```

Observation: The k-NN classifier with 7 neighbors has learned from the training data and predicted the labels of the images in the test set with 98% accuracy.

Overfitting and underfitting

Construct the model complexity curve for the digits dataset. In this exercise, you will compute and plot the training and testing accuracy scores for a variety of different neighbor values. By observing how the accuracy scores differ for the training and testing sets with different values of k, you will develop your intuition for overfitting and underfitting.

The training and testing sets are X_train, X_test, y_train, y_test.

```
# Setup arrays to store train and test accuracies
neighbors = np.arange(1, 9)
train_accuracy = np.empty(len(neighbors))
test_accuracy = np.empty(len(neighbors))

# Loop over different values of k
for i, k in enumerate(neighbors):
    # Setup a k-NN Classifier with k neighbors: knn
    knn = KNeighborsClassifier(n_neighbors=k)
    # Fit the classifier to the training data
    knn.fit(X_train, y_train)
```

```
    #Compute accuracy on the training set
    train_accuracy[i] = knn.score(X_train, y_train)
    #Compute accuracy on the testing set
    test_accuracy[i] = knn.score(X_test, y_test)

# Generate plot
plt.title('k-NN: Varying Number of Neighbors')
plt.plot(neighbors, test_accuracy, label = 'Testing Accuracy')
plt.plot(neighbors, train_accuracy, label = 'Training Accuracy')
plt.legend()
plt.xlabel('Number of Neighbors')
plt.ylabel('Accuracy')
plt.show()
```
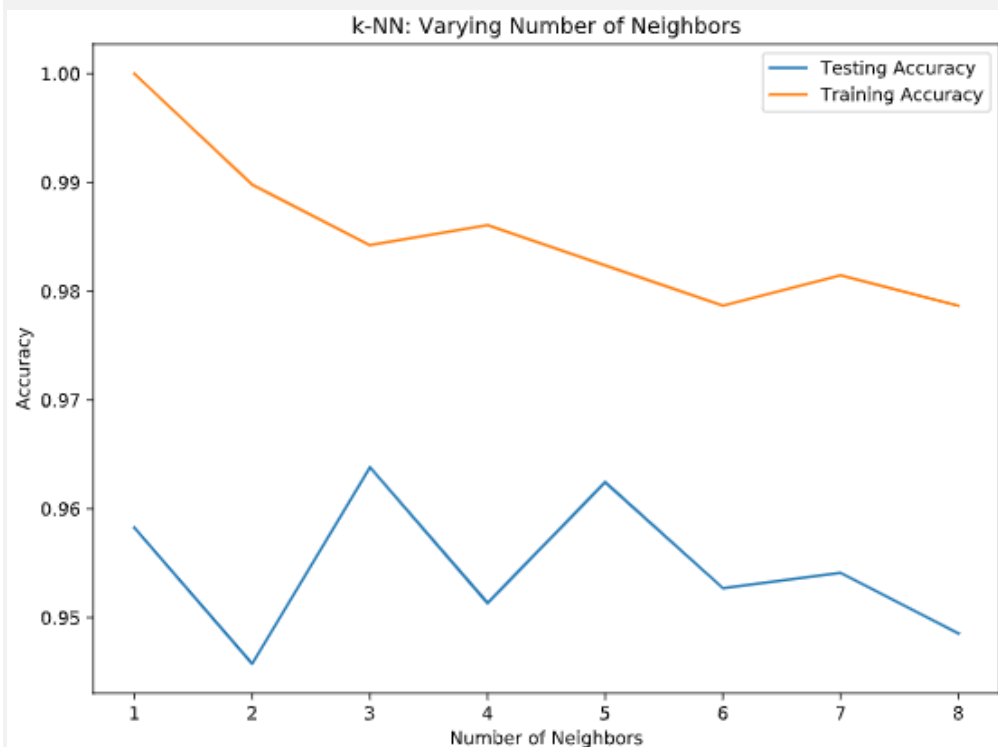

Observation: The test accuracy is highest when using 3 and 5 neighbors. Using 8 neighbors or more seems to result in a simple model that underfits the data.

Chapter 2. Regression

In the previous chapter, you used image and political datasets to predict binary and multiclass outcomes. But what if your problem requires a continuous outcome? Regression is best suited to solving such problems. You will learn about fundamental concepts in regression and apply them to predict the life expectancy in a given country using Gapminder data.

Introduction to regression

```
lr = LinearRegression()
lr.fit(X_rooms, y)
pred_space = np.linspace(min(X_rooms),max(X_rooms)).reshape(-1,1)
plt.scatter(X_rooms, y, color='blue')
plt.plot(pred_space, lr.predict(pred_space), color='black')
plt.show()
```
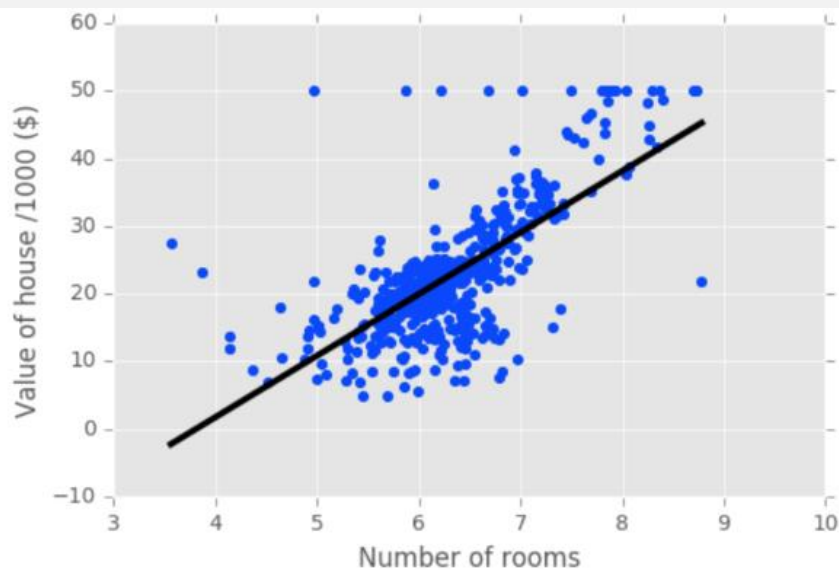
Which of the following is a regression problem?

Regression models can be used in a variety of contexts to solve a variety of different problems. Given below are four example applications of machine learning. Your job is to pick the one that is *best* framed as a **regression** problem.

○ An e-commerce company using labeled customer data to predict whether or not a customer will purchase a particular item.

○ A healthcare company using data about cancer tumors (such as their geometric measurements) to predict whether a new tumor is benign or malignant.

○ A restaurant using review data to ascribe positive or negative sentiment to a given review.

○ A bike share company using time and weather data to predict the number of bikes being rented at any given hour.

Answer: A bike share company using time and weather data to predict the number of bikes being rented at any given hour. The target variable (number of bike rentals at any given hour) is quantitative, so this is best framed as a regression problem.

Importing data for supervised learning

In this chapter, you will work with **Gapminder** data that has been consolidated as `'gapminder.csv'`. Your goal will be to use this data to predict the life expectancy (target) in a given country based on features such as the country's GDP, fertility rate, and population. Since the target variable here is quantitative, this is a regression problem.

To begin, you will fit a linear regression with just *one* feature: `'fertility'`, which is the average number of children a woman in a given country gives birth to. In later exercises, you will use *all* the features to build regression models.

Before that, you need to import the data and get it into the form needed by scikit-learn. This involves creating feature and target variable arrays. Furthermore, since you are going to use only one feature to begin with, you need to do some reshaping using NumPy's `.reshape()` method. This reshaping is something you will have to do occasionally when working with scikit-learn, so it is useful to practice.

```python
# Import numpy and pandas
import numpy as np
import pandas as pd

# Read the CSV file into a DataFrame: df
df = pd.read_csv('gapminder.csv')

# Create arrays for features and target variable
y = df['life'].values
X = df['fertility'].values

# Print the dimensions of X and y before reshaping
print("Dimensions of y before reshaping: {}".format(y.shape))
print("Dimensions of X before reshaping: {}".format(X.shape))

# Reshape X and y
y = y.reshape(-1,1)
X = X.reshape(-1,1)

# Print the dimensions of X and y after reshaping
print("Dimensions of y after reshaping: {}".format(y.shape))
print("Dimensions of X after reshaping: {}".format(X.shape))<script.py> output:
    Dimensions of y before reshaping: (139,)
    Dimensions of X before reshaping: (139,)
    Dimensions of y after reshaping: (139, 1)
    Dimensions of X after reshaping: (139, 1)
```

Observation: Note the differences in shape before and after applying the `.reshape()` method. Getting the feature and target arrays into the right format for scikit-learn is an important precursor to model building.

Exploring the Gapminder data

It is important to explore your data before building models. Compute the pairwise correlation between columns and display in a heatmap using `sns.heatmap(df.corr(), square=True, cmap='RdYlGn')`
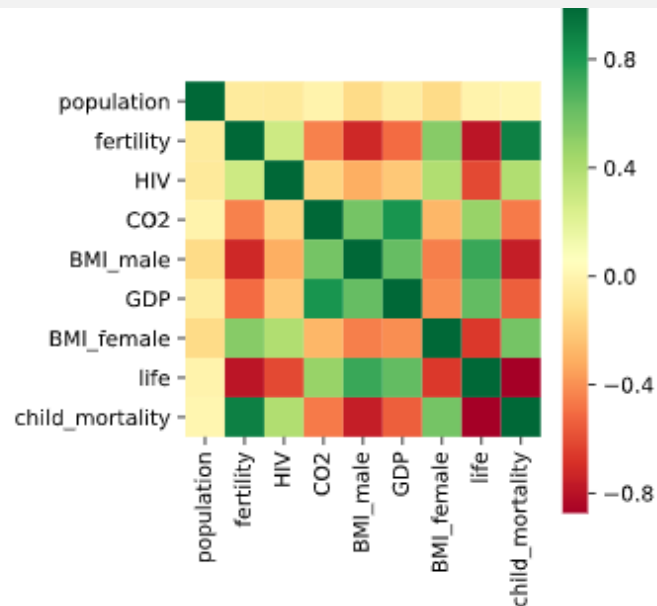
The heatmap shows the correlation between the different features of the Gapminder dataset, loaded as DataFrame `df`. Cells that are in green show positive correlation, while cells that are in red show negative correlation. Take a moment to explore this: Which features are positively correlated with `life`, and which ones are negatively correlated? Does this match your intuition?

Explore the DataFrame using `.info()`, `.describe()`, `.head()`.

Once you have a feel for the data, consider the statements below and select the one that is **not** true.

- ○ The DataFrame has `139` samples (or rows) and `9` columns.

- ○ `life` and `fertility` are negatively correlated.

- ○ The mean of `life` is `69.602878` .

- ○ `fertility` is of type `int64` .

- ○ `GDP` and `life` are positively correlated.

Answer: `fertility` is of type `int64` is not true.

(**fertility** is of type `float64`)

```
In [1]: df.describe()
Out[1]:
          population    fertility        HIV         CO2    BMI_male  \
count  1.390000e+02   139.000000  139.000000  139.000000  139.000000
mean   3.549977e+07     3.005108    1.915612    4.459874   24.623054
std    1.095121e+08     1.615354    4.408974    6.268349    2.209368
min    2.773150e+05     1.280000    0.060000    0.008618   20.397420
25%    3.752776e+06     1.810000    0.100000    0.496190   22.448135
50%    9.705130e+06     2.410000    0.400000    2.223796   25.156990
75%    2.791973e+07     4.095000    1.300000    6.589156   26.497575
max    1.197070e+09     7.590000   25.900000   48.702062   28.456980

                GDP   BMI_female         life  child_mortality
count    139.000000   139.000000   139.000000       139.000000
mean   16638.784173   126.701914    69.602878        45.097122
std    19207.299083     4.471997     9.122189        45.724667
min      588.000000   117.375500    45.200000         2.700000
25%     2899.000000   123.232200    62.200000         8.100000
50%     9938.000000   126.519600    72.000000        24.000000
75%    23278.500000   130.275900    76.850000        74.200000
max   126076.000000   135.492000    82.600000       192.000000
```
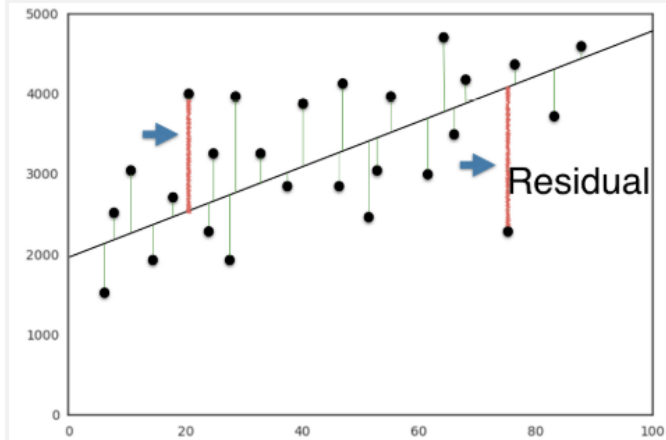
The basics of linear regression

The best fit line is the line that minimises the error/cost/loss function.
Ordinary least squares (OLS) minimises the mean squared error (MSE), ie, sum of squares of residuals.
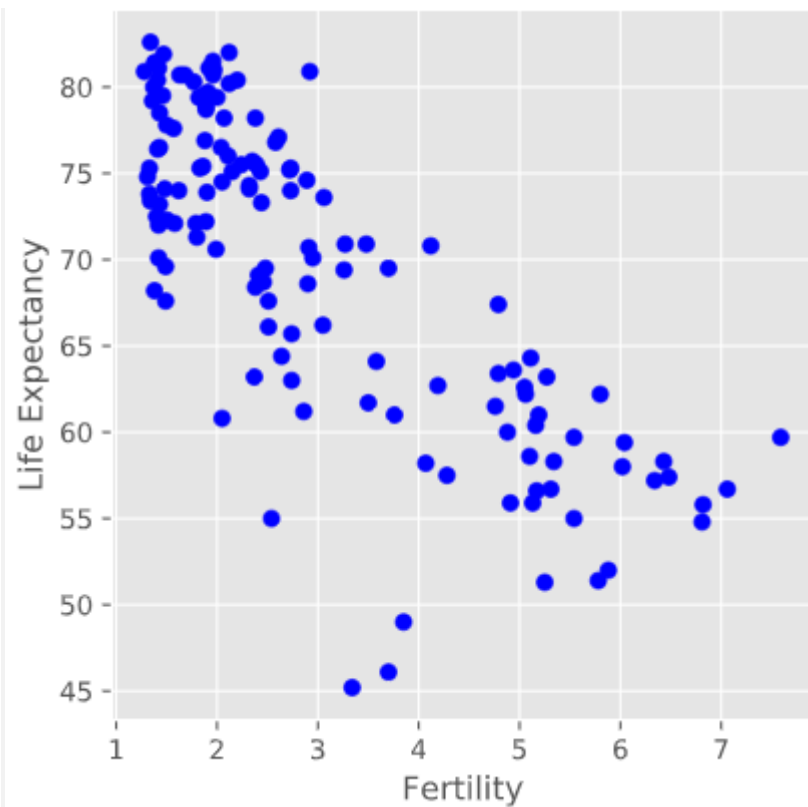

Fit & predict for regression

Fit a linear regression and predict life expectancy using just one feature. In this exercise, you will use the `'fertility'` feature of the Gapminder dataset. Since the goal is to predict life expectancy, the target variable here is `'life'`.
The array for the target variable `'life'` is `y`.
The array for `'fertility'` is `X_fertility`.

A scatter plot (with `'fertility'` on the x-axis and `'life'` on the y-axis) shows a strongly negative correlation, so a linear regression should be able to capture this trend. Your job is to fit a linear regression and then predict the life expectancy, overlaying these predicted values on the plot to generate a regression line. You will also compute and print the R² score using sckit-learn's `.score()` method.

```
# Import LinearRegression
from sklearn.linear_model import LinearRegression

# Create the regressor: reg
reg = LinearRegression()

# Create the prediction space
prediction_space = np.linspace(min(X_fertility), max(X_fertility)).reshape(-1,1)
```
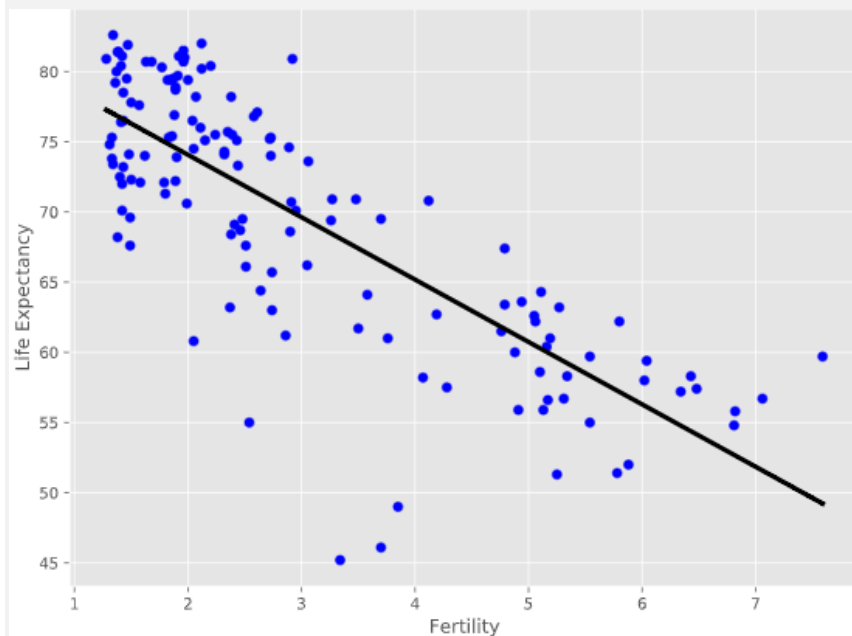
```
# Fit the model to the data
reg.fit(X_fertility, y)

# Compute predictions over the prediction space: y_pred
y_pred = reg.predict(prediction_space)

# Print R^2
print(reg.score(X_fertility, y))

# Plot regression line
plt.plot(prediction_space, y_pred, color='black', linewidth=3)
plt.show()

<script.py> output:
    0.6192442167740035
```


Observation: The line captures the underlying trend in the data. And the performance (R^2=0.619) is quite decent for this basic regression model with only one feature.

Train/test split for regression

Train and test sets are vital to ensure that your supervised learning model is able to generalize well to new data. This was true for classification models, and is equally true for linear regression models.

In this exercise, you will split the Gapminder dataset into training and testing sets, and then fit and predict a linear regression over **all** features. In addition to computing the R^2 score, you will also compute the Root Mean Squared Error (RMSE), which is another commonly used metric to evaluate regression models.

The feature array x and target variable array y have been extracted from the DataFrame df.

```python
# Import necessary modules
from sklearn.linear_model import LinearRegression
from sklearn.metrics import mean_squared_error
from sklearn.model_selection import train_test_split

# Create training and test sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size = 0.3, random_state=42)

# Create the regressor: reg_all
reg_all = LinearRegression()

# Fit the regressor to the training data
reg_all.fit(X_train, y_train)

# Predict on the test data: y_pred
y_pred = reg_all.predict(X_test)

# Compute and print R^2 and RMSE
print("R^2: {}".format(reg_all.score(X_test, y_test)))
rmse = np.sqrt(mean_squared_error(y_test, y_pred))
print("Root Mean Squared Error: {}".format(rmse))<script.py> output:
```

```
R^2: 0.838046873142936
Root Mean Squared Error: 3.2476010800377213
```

Observation: Using all features has improved the model score (R²=0.838). This makes sense, as the model has more information to learn from. However, model performance is dependent on the way the data is split, and it is not representative of the model's ability to generalise. A better way to validate the model is to use cross-validation.

Cross–validation (CV)

Illustration of a 5-fold Cross Validation

5–fold cross–validation

Cross-validation is a vital step in evaluating a model. It maximizes the amount of data that is used to train the model, as during the course of training, the model is not only trained, but also tested on all of the available data.

In this exercise, you will practice 5-fold cross validation on the Gapminder data. By default, scikit-learn's `cross_val_score()` function uses R² as the metric of choice for regression. Since you are performing 5-fold cross-validation, the function will return 5 scores. Your job is to compute these 5 scores and then take their average.

The DataFrame has been loaded as `df` and split into the feature/target variable arrays `X` and `y`.

```
# Import the necessary modules
from sklearn.linear_model import LinearRegression
from sklearn.model_selection import cross_val_score

# Create a linear regression object: reg
reg = LinearRegression()

# Compute 5-fold cross-validation scores: cv_scores
cv_scores = cross_val_score(reg, X, y, cv=5)

# Print the 5-fold cross-validation scores
print(cv_scores)print("Average 5-Fold CV Score: {}".format(np.mean(cv_scores)))

<script.py> output:
    [0.81720569 0.82917058 0.90214134 0.80633989 0.94495637]
    Average 5-Fold CV Score: 0.8599627722793232
```

After cross-validated your model, you can more confidently evaluate its predictions.

K–Fold CV comparison

Cross validation is essential but do not forget that the more folds you use, the more computationally expensive cross-validation becomes. In this exercise, you will explore this for yourself. Your job is to perform 3-fold cross-validation and then 10-fold cross-validation on the Gapminder dataset.

You can use `%timeit` to see how long each 3-fold CV takes compared to 10-fold CV by executing the following `cv=3` and `cv=10`: `%timeit cross_val_score(reg, X, y, cv = ____)`

The feature/target variable arrays `x` and `y` have been created.

```python
# Import necessary modules
from sklearn.linear_model import LinearRegression
from sklearn.model_selection import cross_val_score

# Create a linear regression object: reg
reg = LinearRegression()

# Perform 3-fold CV
cvscores_3 = cross_val_score(reg, X, y, cv=3)
print(np.mean(cvscores_3))

# Perform 10-fold CV
cvscores_10 = cross_val_score(reg, X, y, cv=10)
print(np.mean(cvscores_10))
```

```
<script.py> output:
    0.8718712782622108
    0.8436128620131201

In [1]: %timeit cross_val_score(reg, X, y, cv = 3)
100 loops, best of 3: 8.01 ms per loop

In [2]: %timeit cross_val_score(reg, X, y, cv = 10)
10 loops, best of 3: 21.8 ms per loop
```

Observation: 10-fold CV took almost 3 times longer than 3-fold CV.

Regularized regression

Large coefficients generated by regression models can lead to overfitting; regularisation can prevent that.

Alpha = hyperparameter to be tuned.

Low Alpha (Alpha=0): original OLS, which can lead to overfitting.

High Alpha: can lead to underfitting.

Regularization I: Lasso

Lasso model is able to select the `'RM'` feature as being the most important for predicting Boston house prices, while shrinking the coefficients of certain other features to 0. Its ability to perform feature selection in this way becomes even more useful when you are dealing with data involving thousands of features.

In this exercise, you will fit a lasso regression to the Gapminder data you have been working with and plot the coefficients. The coefficients of some features will be shrunk to 0, with only the most important ones remaining.

The feature and target variable arrays are `X` and `y`.

```python
# Import Lasso
from sklearn.linear_model import Lasso

# Instantiate a lasso regressor: lasso
lasso = Lasso(alpha=0.4, normalize=True)

# Fit the regressor to the data
lasso.fit(X,y)
```

```
# Compute and print the coefficients
lasso_coef = lasso.coef_
print(lasso_coef)

# Plot the coefficients
plt.plot(range(len(df_columns)), lasso_coef)
plt.xticks(range(len(df_columns)), df_columns.values, rotation=60)
plt.margins(0.02)
plt.show()
```

<script.py> output: [-0.   -0.   -0.   0.   0.   0.   -0.   -0.07087587]
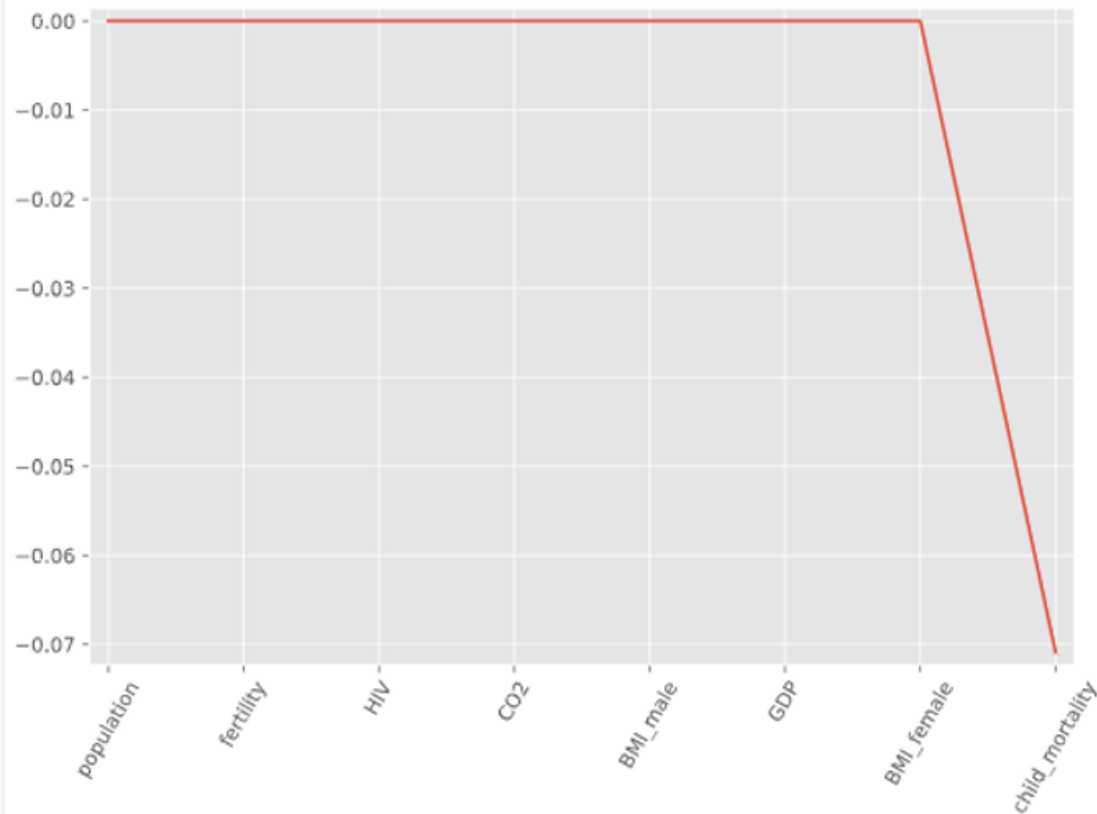
Observation: According to the lasso algorithm, `'child_mortality'` is the most important feature when predicting life expectancy.

Lasso performs regularization by adding to the loss function a penalty term of the *absolute* value of each coefficient multiplied by some alpha. This is also known as **L1** regularization because the regularization term is the **L1** norm of the coefficients.

Loss function = OLS loss function +

$$\alpha * \sum_{i=1}^{n} |a_i|$$

Lasso is great for feature selection, but when building regression models, Ridge regression should be your first choice.

Regularization II: Ridge

If the sum of the *squared* values of the coefficients is multiplied by some alpha — like in Ridge regression — you would be computing the **L2** norm.

Loss function = OLS loss function +

$$\alpha * \sum_{i=1}^{n} a_i^2$$

In this exercise, you will practice fitting ridge regression models over a range of different alphas, and plot cross-validated R^2 scores for each, using this function that we have defined for you, which plots the **R2** score as well as standard error for each alpha:

```python
def display_plot(cv_scores, cv_scores_std):
    fig = plt.figure()
    ax = fig.add_subplot(1,1,1)
    ax.plot(alpha_space, cv_scores)
    std_error = cv_scores_std / np.sqrt(10)
    ax.fill_between(alpha_space, cv_scores + std_error, cv_scores - std_error, alpha=0.2)
    ax.set_ylabel('CV Score +/- Std Error')
    ax.set_xlabel('Alpha')
    ax.axhline(np.max(cv_scores), linestyle='--', color='.5')
    ax.set_xlim([alpha_space[0], alpha_space[-1]])
    ax.set_xscale('log')
    plt.show()
```

Don't worry about the specifics of the above function works. The motivation behind this exercise is for you to see how the R^2 score varies with different alphas, and to understand the importance of selecting the right value for alpha. You'll learn how to tune alpha in the next chapter.

```python
# Import necessary modules
from sklearn.linear_model import Ridge
from sklearn.model_selection import cross_val_score

# Setup the array of alphas and lists to store scores
alpha_space = np.logspace(-4, 0, 50)
ridge_scores = []
ridge_scores_std = []

# Create a ridge regressor: ridge
ridge = Ridge(normalize=True)

# Compute scores over range of alphas
for alpha in alpha_space:
    # Specify the alpha value to use: ridge.alpha
    ridge.alpha = alpha

    # Perform 10-fold CV: ridge_cv_scores
    ridge_cv_scores = cross_val_score(ridge, X, y, cv=10)

    # Append the mean of ridge_cv_scores to ridge_scores
    ridge_scores.append(np.mean(ridge_cv_scores))
```
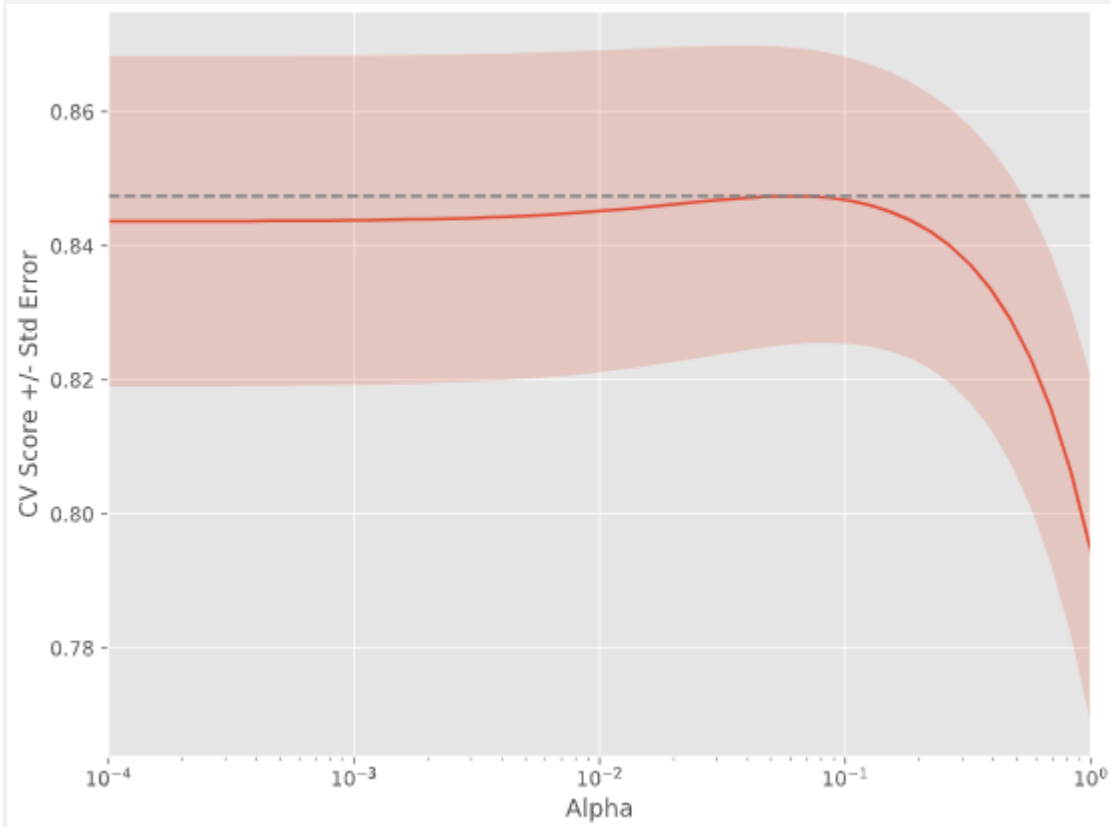
```
    # Append the std of ridge_cv_scores to ridge_scores_std
    ridge_scores_std.append(np.std(ridge_cv_scores))

# Display the plot
display_plot(ridge_scores, ridge_scores_std)
```


The cross-validation scores change with different alphas. Which alpha should you pick? How can you fine-tune your model? You'll learn all about this in the next chapter!

Chapter 3. Fine–tuning your model

Having trained your model, your next task is to evaluate its performance. In this chapter, you will learn about some of the other metrics available in scikit-learn that will allow you to assess your model's performance in a more nuanced manner. Next, learn to optimize your classification and regression models using hyperparameter tuning.

How good is your model?

Accuracy is not a good performance for class imbalance data sets

	Predicted 0	Predicted 1
Actual 0	TN	FP
Actual 1	FN	TP

$$precision = \frac{TP}{TP + FP}$$

$$recall = \frac{TP}{TP + FN}$$

$$F1 = \frac{2 \times precision \times recall}{precision + recall}$$

$$accuracy = \frac{TP + TN}{TP + FN + TN + FP}$$

$$specificity = \frac{TN}{TN + FP}$$

Metrics for classification

In Chapter 1, you evaluated the performance of your k-NN classifier based on its accuracy. However, accuracy is not always an informative metric. In this exercise, you will dive more deeply into evaluating the performance of binary classifiers by computing a confusion matrix and generating a classification report.

Here, you'll work with the **PIMA Indians** dataset obtained from the UCI Machine Learning Repository. The goal is to predict whether or not a given female patient will contract diabetes based on features such as BMI, age, and number of pregnancies. Therefore, it is a binary classification problem. A target value of `0` indicates that the patient does *not* have diabetes, while a value of `1` indicates that the patient *does* have diabetes.

The dataset has been loaded as DataFrame `df`, and the feature and target variable arrays are `X` and `y` respectively.

Your job is to train a k-NN classifier to the data and evaluate its performance by generating a confusion matrix and classification report.

```
# Import necessary modules
from sklearn.metrics import classification_report
from sklearn.metrics import confusion_matrix

# Create training and test set
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.4, random_state=42)

# Instantiate a k-NN classifier: knn
knn = KNeighborsClassifier(n_neighbors=6)

# Fit the classifier to the training data
knn.fit(X_train, y_train)
```

```
# Predict the labels of the test data: y_pred
y_pred = knn.predict(X_test)

# Generate the confusion matrix and classification report
print(confusion_matrix(y_test, y_pred))
print(classification_report(y_test, y_pred))
```

```
<script.py> output:
    [[176  30]
     [ 52  50]]
              precision    recall  f1-score   support

           0       0.77      0.85      0.81       206
           1       0.62      0.49      0.55       102

   avg / total       0.72      0.73      0.72       308
```

By analyzing the confusion matrix and classification report, you can get a much better understanding of your classifier's performance.

Logistic regression and the ROC curve

Logistic regression is used for classification, with a linear decision boundary.
When you vary the threshold **p**, you vary the true positive rate and false positive rate, resulting in the Receiver Operating Characteristic (ROC) curve.
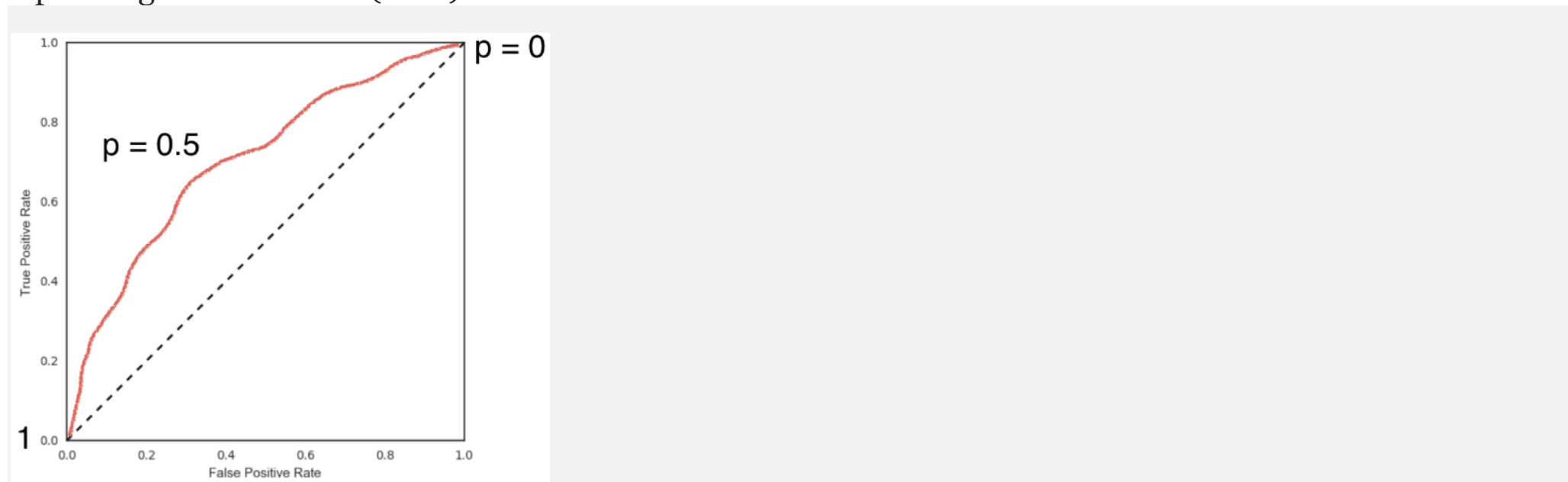
Building a logistic regression model

Time to build your first logistic regression model. Scikit-learn makes it very easy to try different models, since the Train-Test-Split/Instantiate/Fit/Predict paradigm applies to all classifiers and regressors — which are known in scikit-learn as 'estimators'. You will train a logistic regression model on exactly the same data as in the previous exercise. Will it outperform k-NN?

The feature and target variable arrays x and y have been loaded.

```python
# Import the necessary modules
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import confusion_matrix, classification_report

# Create training and test sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size = 0.4, random_state=42)

# Create the classifier: logreg
logreg = LogisticRegression()

# Fit the classifier to the training data
logreg.fit(X_train, y_train)

# Predict the labels of the test set: y_pred
y_pred = logreg.predict(X_test)

# Compute and print the confusion matrix and classification report
print(confusion_matrix(y_test, y_pred))
print(classification_report(y_test, y_pred))
```

```
<script.py> output:
    [[176  30]
     [ 35  67]]
                 precision    recall  f1-score   support

              0       0.83      0.85      0.84       206
              1       0.69      0.66      0.67       102

    avg / total       0.79      0.79      0.79       308
```

Logistic regression works well for binary classification and is used in a variety of machine learning applications.

Plotting an ROC curve

Classification reports and confusion matrices are great methods to quantitatively evaluate model performance, while ROC curves provide a way to visually evaluate models. Most classifiers in scikit-learn have a `.predict_proba()` method which returns the probability of a given sample being in a particular class. Having built a logistic regression model, you'll now evaluate its performance by plotting an ROC curve. In doing so, you'll make use of the `.predict_proba()` method.

Here, you'll continue working with the PIMA Indians diabetes dataset, with the trained classifier as `logreg`.

```
# Import necessary modules
from sklearn.metrics import roc_curve

# Compute predicted probabilities: y_pred_prob
y_pred_prob = logreg.predict_proba(X_test)[:,1]

# Generate ROC curve values: fpr, tpr, thresholds
fpr, tpr, thresholds = roc_curve(y_test, y_pred_prob)

# Plot ROC curve
plt.plot([0, 1], [0, 1], 'k--')
plt.plot(fpr, tpr)
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')
plt.title('ROC Curve')
plt.show()
```
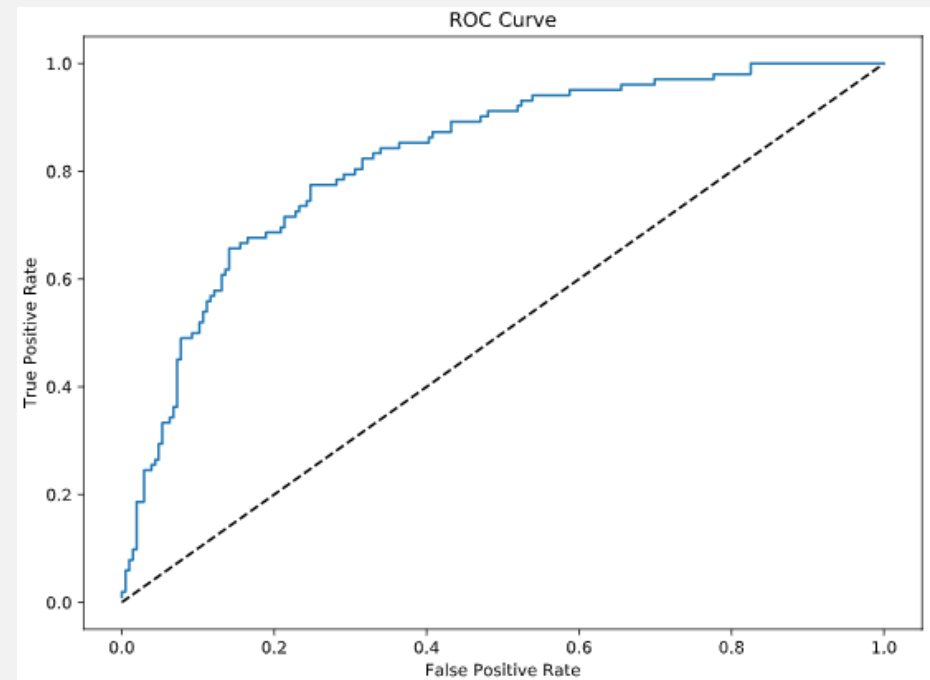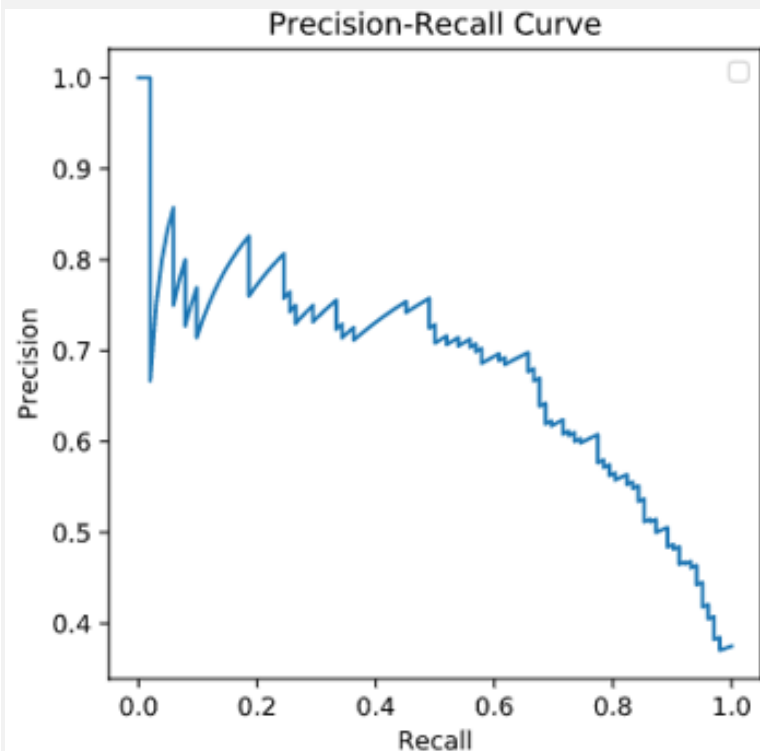
Precision–recall Curve

When looking at your ROC curve, you may have noticed that the y-axis (True positive rate) is also known as recall. Another way to visually evaluate model performance is the precision-recall curve, which is generated by plotting the precision and recall for different thresholds.

Precision-Recall Curve

classification report

	precision	recall	f1-score	support
0	0.83	0.85	0.84	206
1	0.69	0.66	0.67	102
avg / total	0.79	0.79	0.79	308

confusion matrix

```
[[176  30]
 [ 35  67]]
```

Study the above precision-recall curve and then consider the statements given below. Choose the one statement that is **not** true. Note that here, the class is positive (1) if the individual *has* diabetes.

Answer: Precision and recall take *true negatives* into consideration. True negatives do not appear at all in the definitions of precision and recall.

Area under the ROC curve

This ROC curve provides a visual way to assess the classifier's performance. When the area under the ROC curve (AUC) is big, the model is a good classifier.

AUC computation

Say you have a binary classifier that in fact is just randomly making guesses. It would be correct approximately 50% of the time, and the resulting ROC curve would be a diagonal line in which the True Positive Rate and False Positive Rate are always equal. The Area under this ROC curve would be 0.5. This is one way in which the AUC, is an informative metric to evaluate a model. If the AUC is greater than 0.5, the model is better than random guessing.

In this exercise, you'll calculate AUC scores using the `roc_auc_score()` function from `sklearn.metrics` as well as by performing cross-validation on the diabetes dataset.

```
# Import necessary modules
from sklearn.metrics import roc_auc_score
from sklearn.model_selection import cross_val_score

# Compute predicted probabilities: y_pred_prob
y_pred_prob = logreg.predict_proba(X_test)[:,1]

# Compute and print AUC score
print("AUC: {}".format(roc_auc_score(y_test, y_pred_prob)))

# Compute cross-validated AUC scores: cv_auc
cv_auc = cross_val_score(logreg, X, y, cv=5,
scoring='roc_auc')

# Print list of AUC scores
print("AUC scores computed using 5-fold cross-validation: {}".format(cv_auc))
```

```
<script.py> output:
    AUC: 0.8254806777079764
    AUC scores computed using 5-fold cross-validation:
    [0.80148148 0.8062963  0.81481481 0.86245283 0.8554717 ]
```

Hyperparameter tuning

Grid search cross-validation (`GridSearchCV`) can be used to choose the best set of hyperparameters.

		0.1	0.2	0.3	0.4
	0.5	0.701	0.703	0.697	0.696
	0.4	0.699	0.702	0.698	0.702
	0.3	0.721	0.726	0.713	0.703
	0.2	0.706	0.705	0.704	0.701
C	0.1	0.698	0.692	0.688	0.675

Alpha

Hyperparameter tuning with GridSearchCV

You will now practise how to tune the logistic regression using GridSearchCV on the diabetes dataset.

Like the alpha parameter of lasso and ridge regularization, logistic regression also has a regularization parameter **C**, which controls the *inverse* of the regularization strength. A large **C** can lead to an *overfit* model, while a small **C** can lead to an *underfit* model.

The hyperparameter space for **C** has been setup for you. Your job is to use GridSearchCV and logistic regression to find the optimal **C** in this hyperparameter space. The feature array is x and target variable array is y.

```python
# Import necessary modules
from sklearn.linear_model import LogisticRegression
from sklearn.model_selection import GridSearchCV

# Setup the hyperparameter grid
c_space = np.logspace(-5, 8, 15)
param_grid = {'C': c_space}

# Instantiate a logistic regression classifier: logreg
logreg = LogisticRegression()

# Instantiate the GridSearchCV object: logreg_cv
logreg_cv = GridSearchCV(logreg, param_grid, cv=5)

# Fit it to the data
logreg_cv.fit(X, y)

# Print the tuned parameters and score
print("Tuned Logistic Regression Parameters: {}".format(logreg_cv.best_params_))
print("Best score is {}".format(logreg_cv.best_score_))
```

```
<script.py> output:
    Tuned Logistic Regression Parameters: {'C': 3.727593720314938}
    Best score is 0.7708333333333334
```

The hyperparameter `'C'` of `3.727` results in the best performance.

Hyperparameter tuning with RandomizedSearchCV

GridSearchCV can be computationally expensive, especially if you are searching over a large hyperparameter space and dealing with multiple hyperparameters. A solution to this is to use `RandomizedSearchCV`, in which not all hyperparameter values are tried out. Instead, a fixed number of hyperparameter settings is sampled from specified probability distributions.

Here, you'll also be introduced to a new model: the Decision Tree. Just like k-NN, linear regression, and logistic regression, decision trees in scikit-learn have `.fit()` and `.predict()` methods that you can use in exactly the same way as before. Decision trees have many parameters that can be tuned, such as `max_features`, `max_depth`, and `min_samples_leaf`: This makes it an ideal use case for `RandomizedSearchCV`.

The feature array is `X` and target variable array is `y`. The hyperparameter settings have been specified for you. Use `RandomizedSearchCV` to find the optimal hyperparameters.

```
# Import necessary modules
from scipy.stats import randint
from sklearn.tree import DecisionTreeClassifier
from sklearn.model_selection import RandomizedSearchCV
```

```
# Setup the parameters and distributions to sample from: param_dist
param_dist = {"max_depth": [3, None],
              "max_features": randint(1, 9),
              "min_samples_leaf": randint(1, 9),
              "criterion": ["gini", "entropy"]}

# Instantiate a Decision Tree classifier: tree
tree = DecisionTreeClassifier()

# Instantiate the RandomizedSearchCV object: tree_cv
tree_cv = RandomizedSearchCV(tree, param_dist, cv=5)

# Fit it to the data
tree_cv.fit(X, y)

# Print the tuned parameters and score
print("Tuned Decision Tree Parameters: {}".format(tree_cv.best_params_))
print("Best score is {}".format(tree_cv.best_score_))
```

```
<script.py> output:
    Tuned Decision Tree Parameters: {'criterion': 'gini', 'max_depth': 3,
        'max_features': 5, 'min_samples_leaf': 2}
    Best score is 0.7395833333333334
```

Note that `RandomizedSearchCV` will never outperform `GridSearchCV`. Instead, it is valuable because it saves on computation time.

Hold-out set for final evaluation

We will cover how to split the data into training and test sets, so as to hold out a portion of your data for evaluation purposes.

Hold-out set reasoning

For which of the following reasons would you want to use a hold-out set for the very end?

○ You want to maximize the amount of training data used.

○ You want to be absolutely certain about your model's ability to generalize to unseen data.

○ You want to tune the hyperparameters of your model.

Answer: You want to be absolutely certain about your model's ability to generalize to unseen data. The idea is to tune the model's hyperparameters on the training set, and then evaluate its performance on the hold-out set which it has never seen before.

Hold-out set in practice I: Classification

You will now practice evaluating a model with tuned hyperparameters on a hold-out set. The feature array and target variable array from the diabetes dataset are `x` and `y`.

In addition to **C**, logistic regression has a `'penalty'` hyperparameter which specifies whether to use `'l1'` or `'l2'` regularization. Your job in this exercise is to create a hold-out set, tune the `'C'` and `'penalty'` hyperparameters of a logistic regression classifier using `GridSearchCV` on the training set.

```
# Import necessary modules
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LogisticRegression
from sklearn.model_selection import GridSearchCV

# Create the hyperparameter grid
c_space = np.logspace(-5, 8, 15)
param_grid = {'C': c_space, 'penalty': ['l1', 'l2']}
```

```
# Instantiate the logistic regression classifier: logreg
logreg = LogisticRegression()

# Create train and test sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size = 0.4, random_state=42)

# Instantiate the GridSearchCV object: logreg_cv
logreg_cv = GridSearchCV(logreg, param_grid, cv=5)

# Fit it to the training data
logreg_cv.fit(X_train, y_train)

# Print the optimal parameters and best score
print("Tuned Logistic Regression Parameter: {}".format(logreg_cv.best_params_))
print("Tuned Logistic Regression Accuracy: {}".format(logreg_cv.best_score_))
```

```
<script.py> output:
    Tuned Logistic Regression Parameter: {'C': 0.4393970560760795,
        'penalty': 'l1'}
    Tuned Logistic Regression Accuracy: 0.7652173913043478
```

Hold−out set in practice II: Regression

Remember lasso and ridge regression from the previous chapter? Lasso used the L1 penalty to regularize, while ridge used the L2 penalty. There is another type of regularized regression known as the **elastic net regularization**, and its penalty term is a linear combination of the L1 and L2 penalties: a*L1 + b*L2

In scikit-learn, this term is represented by the `'l1_ratio'` parameter: An `'l1_ratio'` of 1 corresponds to an L1 penalty, and anything lower is a combination of L1 and L2.

In this exercise, you will `GridSearchCV` to tune the `'l1_ratio'` of an elastic net model trained on the Gapminder data. As in the previous exercise, use a hold-out set to evaluate your model's performance.

```python
# Import necessary modules
from sklearn.linear_model import ElasticNet
from sklearn.metrics import mean_squared_error
from sklearn.model_selection import GridSearchCV
from sklearn.model_selection import train_test_split

# Create train and test sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size = 0.40, random_state = 42)

# Create the hyperparameter grid
l1_space = np.linspace(0, 1, 30)
param_grid = {'l1_ratio': l1_space}

# Instantiate the ElasticNet regressor: elastic_net
elastic_net = ElasticNet()

# Setup the GridSearchCV object: gm_cv
gm_cv = GridSearchCV(elastic_net, param_grid, cv=5)

# Fit it to the training data
gm_cv.fit(X_train, y_train)

# Predict on the test set and compute metrics
y_pred = gm_cv.predict(X_test)
r2 = gm_cv.score(X_test, y_test)
mse = mean_squared_error(y_test, y_pred)
print("Tuned ElasticNet l1 ratio: {}".format(gm_cv.best_params_))
print("Tuned ElasticNet R squared: {}".format(r2))
print("Tuned ElasticNet MSE: {}".format(mse))
```

```
<script.py> output:
    Tuned ElasticNet l1 ratio: {'l1_ratio': 0.20689655172413793}
    Tuned ElasticNet R squared: 0.8668305372460283
    Tuned ElasticNet MSE: 10.05791413339844
```

After learning how to fine-tune your models, it's time to learn about preprocessing techniques and how to piece together all the different stages of the machine learning process into a pipeline!

Chapter 4. Preprocessing and pipelines

This chapter introduces pipelines, and how scikit-learn allows for transformers and estimators to be chained together and used as a single unit. Preprocessing techniques will be introduced as a way to enhance model performance, and pipelines will tie together concepts from previous chapters.

Preprocessing data

Scikit-learn only works with numerical features. Need to encode categorical features using `OneHotEncoder()` or `get_dummies()`.

Origin
US
Europe
Asia

origin_Asia	origin_Europe	origin_US
0	0	1
0	1	0
1	0	0

Exploring categorical features

The Gapminder dataset that you worked with in previous chapters also contained a categorical `'Region'` feature (which has been skipped in previous exercises).
Your job in this exercise is to explore this feature. Boxplots are particularly useful for visualizing categorical features such as this.

```
# Import pandas
import pandas as pd

# Read 'gapminder.csv' into a DataFrame: df
df = pd.read_csv('gapminder.csv')

# Create a boxplot of life expectancy per region
df.boxplot('life', 'Region', rot=60)

# Show the plot
plt.show()
```


Boxplot grouped by Region
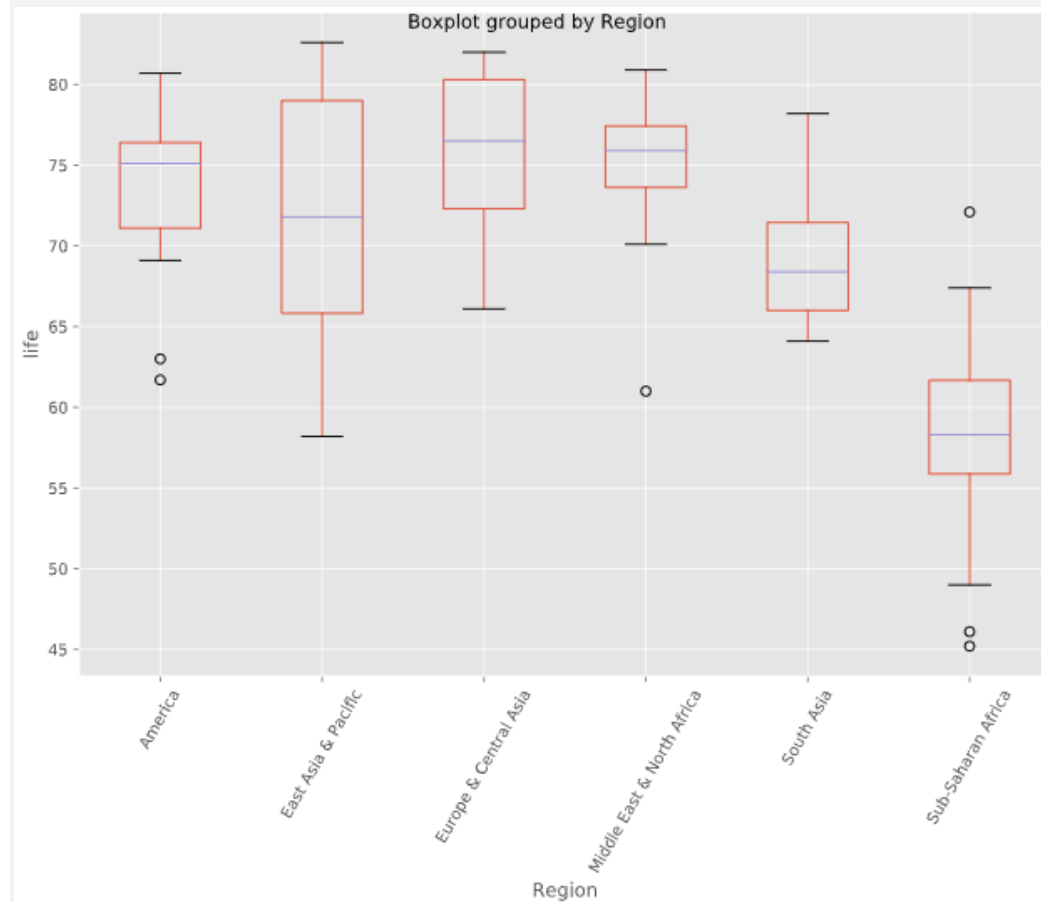
Note: Exploratory data analysis should always be the precursor to model building.

Creating dummy variables

Scikit-learn does not accept non-numerical features. The `'Region'` feature contains very useful information that can predict life expectancy. For example, Sub-Saharan Africa has a lower life expectancy compared to Europe and Central Asia. Therefore, if you are trying to predict life expectancy, it would be preferable to retain the `'Region'` feature. To do this, you need to binarize it by creating dummy variables.

```
# Create dummy variables: df_region
df_region = pd.get_dummies(df)

# Print the columns of df_region
print(df_region.columns)

# Create dummy variables with drop_first=True: df_region
df_region = pd.get_dummies(df, drop_first=True)

# Print the new columns of df_region
print(df_region.columns)
```

```
<script.py> output:
    Index(['population', 'fertility', 'HIV', 'CO2', 'BMI_male', 'GDP',
           'BMI_female', 'life', 'child_mortality', 'Region_America',
           'Region_East Asia & Pacific', 'Region_Europe & Central Asia',
           'Region_Middle East & North Africa', 'Region_South Asia',
           'Region_Sub-Saharan Africa'],
          dtype='object')
    Index(['population', 'fertility', 'HIV', 'CO2', 'BMI_male', 'GDP',
           'BMI_female', 'life', 'child_mortality', 'Region_East Asia & Pacific',
           'Region_Europe & Central Asia', 'Region_Middle East & North Africa',
           'Region_South Asia', 'Region_Sub-Saharan Africa'],
          dtype='object')
```

The `'Region'` can now be used as an additional feature to predict life expectancy!

Regression with categorical features

Having created the dummy variables from the `'Region'` feature, you can build regression models as you did before. Here, you'll use ridge regression to perform 5-fold cross-validation.

```python
# Import necessary modules
from sklearn.linear_model import Ridge
from sklearn.model_selection import cross_val_score

# Instantiate a ridge regressor: ridge
ridge = Ridge(normalize=True, alpha=0.5)

# Perform 5-fold cross-validation: ridge_cv
ridge_cv = cross_val_score(ridge, X, y, cv=5)

# Print the cross-validated scores
print(ridge_cv)
```

```
<script.py> output:
    [0.86808336 0.80623545 0.84004203 0.7754344  0.87503712]
```

Handling missing data

Drop missing data (delete rows or delete columns)

Impute missing data, make an educated guess, eg. replace with mean

Use pipeline to impute missing data, add the model, split data into training and test sets, fit the train data, make predictions and score the model. There is no need to transform the data, as imputers are also known as transformers.

Dropping missing data

The voting dataset from Chapter 1 contained a bunch of missing values that had been processed by steps in this exercise.

The unprocessed dataset is in a DataFrame `df`. Explore with the `.head()` method, and notice certain data points labeled with a `'?'` denoting missing values. Different datasets encode missing values in different ways. Sometimes it may be a `'9999'`, other times a `0`, so real-world data can be very messy! If you're lucky, the missing values will already be encoded as `NaN`. We use `NaN` because it is an efficient and simplified way of internally representing missing data, and it lets us take advantage of pandas methods such as `.dropna()` and `.fillna()`, as well as scikit-learn's Imputation transformer `Imputer()`.

In this exercise, your job is to convert the `'?'`s to `NaN`s, and then drop the rows that contain them from the DataFrame.

```
# Convert '?' to NaN
df[df == '?'] = np.nan

# Print the number of NaNs
print(df.isnull().sum())

# Print shape of original DataFrame
print("Shape of Original DataFrame: {}".format(df.shape))

# Drop missing values and print shape of new DataFrame
df = df.dropna()

# Print shape of new DataFrame
print("Shape of DataFrame After Dropping All Rows with Missing Values: {}".format(df.shape))
```

```
<script.py> output:
    party                0
    infants             12
    water               48
    budget              11
    physician           11
    salvador            15
    religious           11
    satellite           14
    aid                 15
    missile             22
    immigration          7
    synfuels            21
    education           31
    superfund           25
    crime               17
    duty_free_exports   28
    eaa_rsa            104
    dtype: int64
    Shape of Original DataFrame: (435, 17)
    Shape of DataFrame After Dropping All Rows with Missing Values: (232, 17)
```

Observation: When many values in your dataset are missing, if you drop them, you may end up throwing away valuable information along with the missing data. It's better instead to develop an imputation strategy. This is where domain knowledge is useful, but in the absence of it, you can impute missing values with the mean or the median of the row or column that the missing value is in.

Imputing missing data in a ML Pipeline I

As you've come to appreciate, there are many steps to building a model, from creating training and test sets, to fitting a classifier or regressor, to tuning its parameters, to evaluating its performance on new data. Imputation can be seen as the first step of this machine learning process, the entirety of which can be viewed within the context of a pipeline. Scikit-learn provides a pipeline constructor that allows you to piece together these steps into one process and thereby simplify your workflow.

You'll now practice setting up a pipeline with two steps: the imputation step, followed by the instantiation of a classifier. You've seen three classifiers in this course so far: k-NN, logistic regression, and the decision tree. You will now be introduced to a fourth one — the Support Vector Machine, or **SVM**. It works the same as those scikit-learn estimators used previously, in that it has the same `.fit()` and `.predict()` methods as before.

```
# Import the Imputer module
from sklearn.preprocessing import Imputer
from sklearn.svm import SVC

# Setup the Imputation transformer: imp
imp = Imputer(missing_values='NaN', strategy='most_frequent', axis=0)

# Instantiate the SVC classifier: clf
clf = SVC()

# Setup the pipeline with the required steps: steps
steps = [('imputation', imp),
        ('SVM', clf)]
```

Having set up the pipeline steps, you can now use it for classification.

Imputing missing data in a ML Pipeline II

Having setup the steps of the pipeline in the previous exercise, you will now use it on the voting dataset to classify a Congressman's party affiliation. What makes pipelines so incredibly useful is the simple interface that they provide. You can use the `.fit()` and `.predict()` methods on pipelines just as you did with your classifiers and regressors!

Generate a classification report of your predictions. The steps of the pipeline have been set up for you, and the feature array is X and target variable array is y.

```
# Import necessary modules
from sklearn.preprocessing import Imputer
from sklearn.pipeline import Pipeline
from sklearn.svm import SVC

# Setup the pipeline steps: steps
steps = [('imputation', Imputer(missing_values='NaN', strategy='most_frequent', axis=0)),
        ('SVM', SVC())]

# Create the pipeline: pipeline
pipeline = Pipeline(steps)

# Create training and test sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3, random_state=42)# Fit the pipeline to
the train set
pipeline.fit(X_train, y_train)

# Predict the labels of the test set
y_pred = pipeline.predict(X_test)

# Compute metrics
print(classification_report(y_test, y_pred))
```

```
<script.py> output:
                precision    recall  f1-score   support

       democrat      0.99      0.96      0.98        85
     republican      0.94      0.98      0.96        46

    avg / total      0.97      0.97      0.97       131
```

The pipeline has performed imputation as well as classification!

Centering and scaling

Feature scaling ranges widely, and can unduly influence the model, especially for explicit distance models eg. k-NN.
Normalising = scaling and centering, and can be used in a pipeline.

Centering and scaling your data

the performance of a model can improve significantly if the features are scaled. Note that this is not always the case: In the Congressional voting records dataset, for example, all of the features are binary. In such a situation, scaling will have minimal impact.

You will now explore scaling for yourself on a new dataset — **White Wine Quality**! We shall use the `'quality'` feature of the wine to create a binary target variable: If `'quality'` is less than `5`, the target variable is `1`, and otherwise, it is `0`.

The DataFrame, feature and target variable are `df`, `x` and `y` respectively. Some features have different units of measurement. `'density'`, for instance, takes values between 0.98 and 1.04, while `'total sulfur dioxide'` ranges from 9 to 440. As a result, it may be worth scaling the features here. Your job in this exercise is to scale the features and compute the mean and standard deviation of the unscaled features compared to the scaled features.

```
# Import scale
from sklearn.preprocessing import scale

# Scale the features: X_scaled
X_scaled = scale(X)

# Print the mean and standard deviation of the unscaled features
print("Mean of Unscaled Features: {}".format(np.mean(X)))
print("Standard Deviation of Unscaled Features: {}".format(np.std(X)))

# Print the mean and standard deviation of the scaled features
print("Mean of Scaled Features: {}".format(np.mean(X_scaled)))
print("Standard Deviation of Scaled Features: {}".format(np.std(X_scaled)))
```

```
<script.py> output:
    Mean of Unscaled Features: 18.432687072460002
    Standard Deviation of Unscaled Features: 41.54494764094571
    Mean of Scaled Features: 2.73149729816682206e-15
    Standard Deviation of Scaled Features: 0.9999999999999999
```

Observation: The mean and standard deviation of the features have changed after scaling.

Centering and scaling in a pipeline

Check whether or not scaling the features of the White Wine Quality dataset has any impact on its performance. You will use a k-NN classifier as part of a pipeline that includes scaling, and for the purposes of comparison, a k-NN classifier trained on the unscaled data has been provided.

The feature array and target variable array are x and y.

```
# Import the necessary modules
from sklearn.preprocessing import StandardScaler
from sklearn.pipeline import Pipeline

# Setup the pipeline steps: steps
steps = [('scaler', StandardScaler()),
```

```
                ('knn', KNeighborsClassifier())]

# Create the pipeline: pipeline
pipeline = Pipeline(steps)

# Create train and test sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3, random_state=42)

# Fit the pipeline to the training set: knn_scaled
knn_scaled = pipeline.fit(X_train, y_train)

# Instantiate and fit a k-NN classifier to the unscaled data
knn_unscaled = KNeighborsClassifier().fit(X_train, y_train)

# Compute and print metrics
print('Accuracy with Scaling: {}'.format(knn_scaled.score(X_test, y_test)))
print('Accuracy without Scaling: {}'.format(knn_unscaled.score(X_test, y_test)))
```

```
<script.py> output:
    Accuracy with Scaling: 0.7700680272108843
    Accuracy without Scaling: 0.6979591836734694
```

Observation: Scaling has significantly improved model performance!

Bringing it all together I: Pipeline for classification

It is time now to piece together everything you have learned so far into a pipeline for classification! Your job in this exercise is to build a pipeline that includes scaling and hyperparameter tuning to classify wine quality.

Use the SVM classifier. The hyperparameters you will tune are **C** and **gamma**. **C** controls the regularization strength (similar to logistic regression), while **gamma** controls the kernel coefficient.

```python
# Setup the pipeline
steps = [('scaler', StandardScaler()),
         ('SVM', SVC())]pipeline = Pipeline(steps)

# Specify the hyperparameter space
parameters = {'SVM__C':[1, 10, 100],
              'SVM__gamma':[0.1, 0.01]}

# Create train and test sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.20, random_state=21)

# Instantiate the GridSearchCV object: cv
cv = GridSearchCV(pipeline, param_grid=parameters)

# Fit to the training set
cv.fit(X_train, y_train)

# Predict the labels of the test set: y_pred
y_pred = cv.predict(X_test)

# Compute and print metrics
print("Accuracy: {}".format(cv.score(X_test, y_test)))
print(classification_report(y_test, y_pred))
print("Tuned Model Parameters: {}".format(cv.best_params_))
```

```
<script.py> output:
    Accuracy: 0.7795918367346939
               precision    recall  f1-score   support

        False       0.83      0.85      0.84       662
         True       0.67      0.63      0.65       318

    avg / total       0.78      0.78      0.78       980

    Tuned Model Parameters: {'SVM__C': 10, 'SVM__gamma': 0.1}
```

Bringing it all together II: Pipeline for regression

For this final exercise, you will return to the Gapminder dataset (including missing values).

Your job is to build a pipeline that imputes the missing data, scales the features, and fits an ElasticNet to the Gapminder data. You will then tune the `l1_ratio` of your ElasticNet using GridSearchCV.

```python
# Setup the pipeline steps: steps
steps = [('imputation', Imputer(missing_values='NaN', strategy='mean', axis=0)),
         ('scaler', StandardScaler()),
         ('elasticnet', ElasticNet())]

# Create the pipeline: pipeline
pipeline = Pipeline(steps)

# Specify the hyperparameter space
parameters = {'elasticnet__l1_ratio':np.linspace(0,1,30)}

# Create train and test sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.4, random_state=42)

# Create the GridSearchCV object: gm_cv
gm_cv = GridSearchCV(pipeline, param_grid=parameters)

# Fit to the training set
gm_cv.fit(X_train, y_train)

# Compute and print the metrics
r2 = gm_cv.score(X_test, y_test)
print("Tuned ElasticNet Alpha: {}".format(gm_cv.best_params_))
print("Tuned ElasticNet R squared: {}".format(r2))
```

You have now mastered the fundamentals of supervised learning with scikit-learn!

Final thoughts

These are what you have learnt:

- Using machine learning techniques to build predictive models

- for both regression and classification problems

- using real-world data

- Concept of underfitting and overfitting

- Train-Test split

- Cross-validation

- Grid search to fine tune models and report performance

- Regularisation: lasso, ridge, elasticnet

- Data preprocessing

- Pipeline