In [1]:

```
%matplotlib inline
import matplotlib.pyplot as plt
import numpy as np
```

Supervised Learning Part 2 -- Regression Analysis

In regression we are trying to predict a continuous output variable -- in contrast to the nominal variables we were predicting in the previous classification examples.

Let's start with a simple toy example with one feature dimension (explanatory variable) and one target variable. We will create a dataset out of a sine curve with some noise:

In [2]:

```
x = np.linspace(-3, 3, 100)
print(x)
```

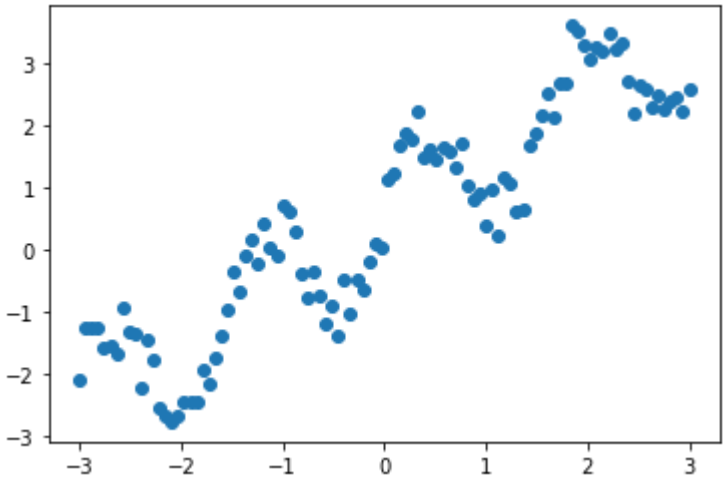
```
[-3.         -2.93939394 -2.87878788 -2.81818182 -2.75757576 -2.6969
697
 -2.63636364 -2.57575758 -2.51515152 -2.45454545 -2.39393939 -2.3333
3333
 -2.27272727 -2.21212121 -2.15151515 -2.09090909 -2.03030303 -1.9696
9697
 -1.90909091 -1.84848485 -1.78787879 -1.72727273 -1.66666667 -1.6060
6061
 -1.54545455 -1.48484848 -1.42424242 -1.36363636 -1.3030303  -1.2424
2424
 -1.18181818 -1.12121212 -1.06060606 -1.         -0.93939394 -0.8787
8788
 -0.81818182 -0.75757576 -0.6969697  -0.63636364 -0.57575758 -0.5151
5152
 -0.45454545 -0.39393939 -0.33333333 -0.27272727 -0.21212121 -0.1515
1515
 -0.09090909 -0.03030303  0.03030303  0.09090909  0.15151515  0.2121
2121
  0.27272727  0.33333333  0.39393939  0.45454545  0.51515152  0.5757
5758
  0.63636364  0.6969697   0.75757576  0.81818182  0.87878788  0.9393
9394
  1.          1.06060606  1.12121212  1.18181818  1.24242424  1.3030
303
  1.36363636  1.42424242  1.48484848  1.54545455  1.60606061  1.6666
6667
  1.72727273  1.78787879  1.84848485  1.90909091  1.96969697  2.0303
0303
  2.09090909  2.15151515  2.21212121  2.27272727  2.33333333  2.3939
3939
  2.45454545  2.51515152  2.57575758  2.63636364  2.6969697   2.7575
7576
  2.81818182  2.87878788  2.93939394  3.        ]
```

In [3]:

```
rng = np.random.RandomState(42)
y = np.sin(4 * x) + x + rng.uniform(size=len(x))
```

In [4]:

```
plt.plot(x, y, 'o');
```



Linear Regression

The first model that we will introduce is the so-called simple linear regression. Here, we want to fit a line to the data, which

One of the simplest models again is a linear one, that simply tries to predict the data as lying on a line. One way to find such a line is `LinearRegression` (also known as *Ordinary Least Squares (OLS)* (https://en.wikipedia.org/wiki/Ordinary_least_squares) regression). The interface for LinearRegression is exactly the same as for the classifiers before, only that `y` now contains float values, instead of classes.

As we remember, the scikit-learn API requires us to provide the target variable (`y`) as a 1-dimensional array; scikit-learn's API expects the samples (`X`) in form a 2-dimensional array -- even though it may only consist of 1 feature. Thus, let us convert the 1-dimensional `x` NumPy array into an `X` array with 2 axes:

In [5]:

```
print('Before: ', x.shape)
X = x[:, np.newaxis]
print('After: ', X.shape)
```

```
Before:  (100,)
After:  (100, 1)
```

Again, we start by splitting our dataset into a training (75%) and a test set (25%):

In [6]:

```
from sklearn.model_selection import train_test_split

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.25, random
_state=42)
```

Next, we use the learning algorithm implemented in `LinearRegression` to **fit a regression model to the training data**:

In [7]:

```
from sklearn.linear_model import LinearRegression

regressor = LinearRegression()
regressor.fit(X_train, y_train)
```

Out[7]:

```
LinearRegression(copy_X=True, fit_intercept=True, n_jobs=None, norma
lize=False)
```

After fitting to the training data, we paramerterized a linear regression model with the following values.

In [8]:

```
print('Weight coefficients: ', regressor.coef_)
print('y-axis intercept: ', regressor.intercept_)
```

```
Weight coefficients:  [0.90211711]
y-axis intercept:  0.44840974988268
```

Since our regression model is a linear one, the relationship between the target variable (y) and the feature variable (x) is defined as

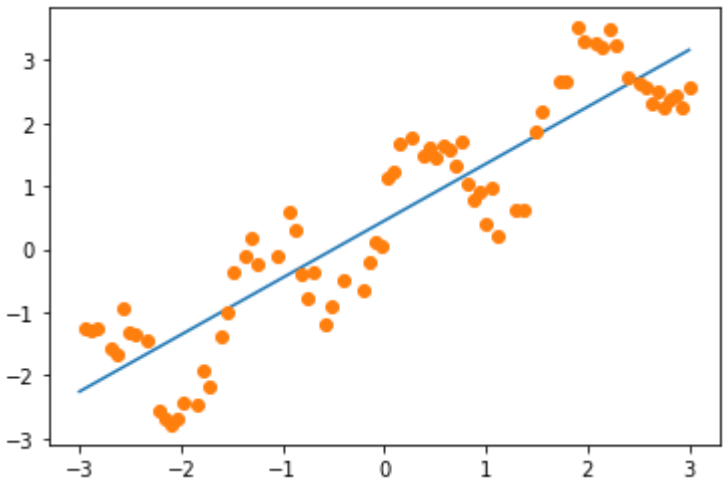
$$y = \text{weight} \times x + \text{intercept}\ .$$

Plugging in the min and max values into thos equation, we can plot the regression fit to our training data:

In [9]:

```
min_pt = X.min() * regressor.coef_[0] + regressor.intercept_
max_pt = X.max() * regressor.coef_[0] + regressor.intercept_

plt.plot([X.min(), X.max()], [min_pt, max_pt])
plt.plot(X_train, y_train, 'o');
```



Similar to the estimators for classification in the previous notebook, we use the `predict` method to predict the target variable. And we expect these predicted values to fall onto the line that we plotted previously:

In [10]:

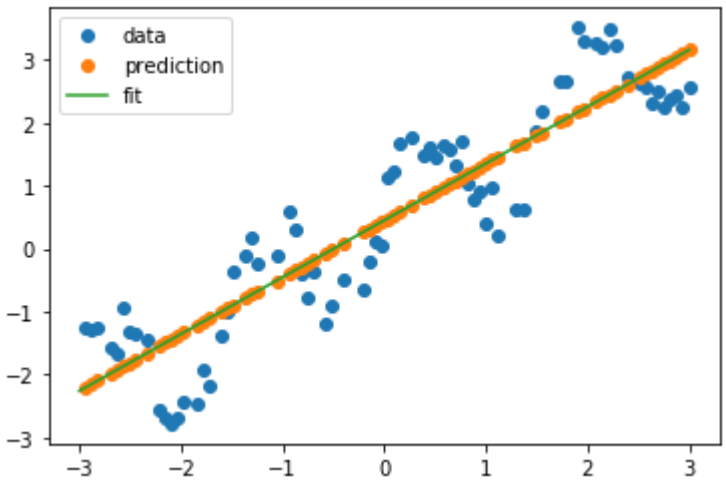
```
y_pred_train = regressor.predict(X_train)
```

In [11]:

```
plt.plot(X_train, y_train, 'o', label="data")
plt.plot(X_train, y_pred_train, 'o', label="prediction")
plt.plot([X.min(), X.max()], [min_pt, max_pt], label='fit')
plt.legend(loc='best')
```

Out[11]:

```
<matplotlib.legend.Legend at 0x1a14291320>
```



As we can see in the plot above, the line is able to capture the general slope of the data, but not many details.

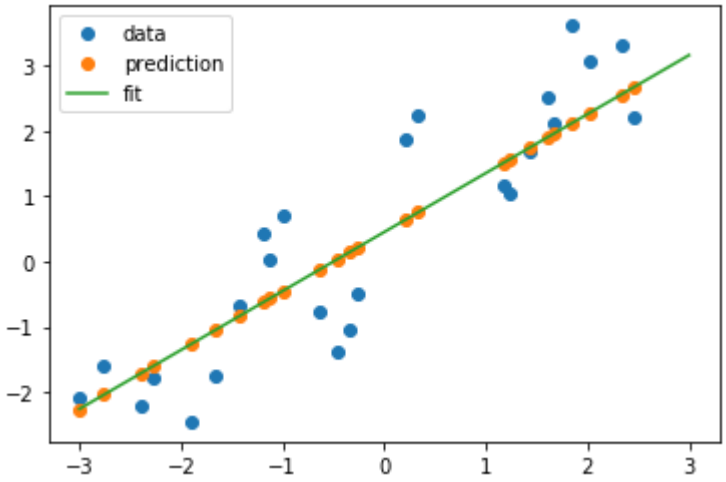
Next, let's try the test set:

In [12]:

```
y_pred_test = regressor.predict(X_test)
```

In [13]:

```
plt.plot(X_test, y_test, 'o', label="data")
plt.plot(X_test, y_pred_test, 'o', label="prediction")
plt.plot([X.min(), X.max()], [min_pt, max_pt], label='fit')
plt.legend(loc='best');
```



Again, scikit-learn provides an easy way to evaluate the prediction quantitatively using the `score` method. For regression tasks, this is the R^2 score. Another popular way would be the Mean Squared Error (MSE). As its name implies, the MSE is simply the average squared difference over the predicted and actual target values

$$MSE = \frac{1}{n}\sum_{i=1}^{n}(\text{predicted}_i - \text{true}_i)^2$$

In [14]:

```
regressor.score(X_test, y_test)
```

Out[14]:

```
0.7994321405079685
```

EXERCISE:

- Add a (non-linear) feature containing `sin(4x)` to `X` and redo the fit as a new column to X_train (and X_test). Visualize the predictions with this new richer, yet linear, model.
- Hint: you can use `np.concatenate(A, B, axis=1)` to concatenate two matrices A and B horizontal (to combine the columns).

In [15]:

```
# %load solutions/06B_lin_with_sine.py
```

KNeighborsRegression

As for classification, we can also use a neighbor based method for regression. We can simply take the output of the nearest point, or we could average several nearest points. This method is less popular for regression than for classification, but still a good baseline.

In [16]:

```
from sklearn.neighbors import KNeighborsRegressor
kneighbor_regression = KNeighborsRegressor(n_neighbors=1)
kneighbor_regression.fit(X_train, y_train)
```

Out[16]:

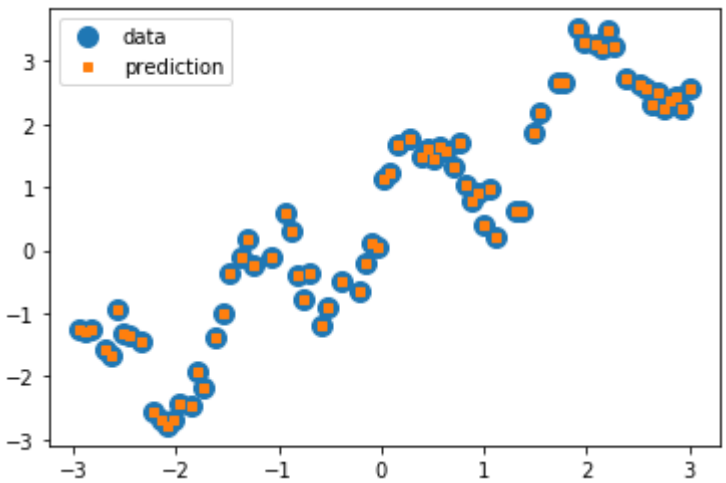
```
KNeighborsRegressor(algorithm='auto', leaf_size=30, metric='minkowsk
i',
                    metric_params=None, n_jobs=None, n_neighbors=1,
p=2,
                    weights='uniform')
```

Again, let us look at the behavior on training and test set:

In [17]:

```
y_pred_train = kneighbor_regression.predict(X_train)

plt.plot(X_train, y_train, 'o', label="data", markersize=10)
plt.plot(X_train, y_pred_train, 's', label="prediction", markersize=4)
plt.legend(loc='best');
```

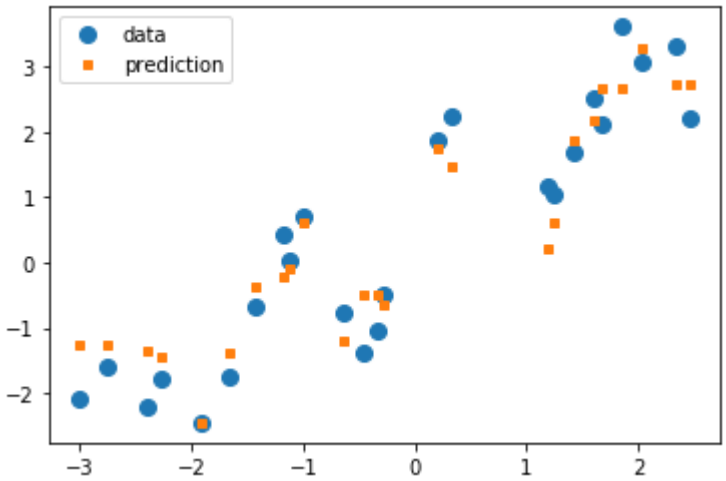


On the training set, we do a perfect job: each point is its own nearest neighbor!

In [18]:

```
y_pred_test = kneighbor_regression.predict(X_test)

plt.plot(X_test, y_test, 'o', label="data", markersize=8)
plt.plot(X_test, y_pred_test, 's', label="prediction", markersize=4)
plt.legend(loc='best');
```



On the test set, we also do a better job of capturing the variation, but our estimates look much messier than before. Let us look at the R^2 score:

In [19]:

```
kneighbor_regression.score(X_test, y_test)
```

Out[19]:

```
0.9166293022467948
```

Much better than before! Here, the linear model was not a good fit for our problem; it was lacking in complexity and thus under-fit our data.

EXERCISE:

- Compare the KNeighborsRegressor and LinearRegression on the boston housing dataset. You can load the dataset using ``sklearn.datasets.load_boston``. You can learn about the dataset by reading the ``DESCR`` attribute.

In [ ]:

```

```

In [20]:

```
# %load solutions/06A_knn_vs_linreg.py
```