11_regress

March 9, 2023

1 Polynomial Regression

variationalform https://variationalform.github.io/

Just Enough: progress at pace https://variationalform.github.io/

https://github.com/variationalform

Simon Shaw https://www.brunel.ac.uk/people/simon-shaw.

This document uses python

and also makes use of LaTeX

in Markdown

1.1 What this is about:

- Polynomial regression.
- The best straight line (or polynomial) through data.
- Cost and Loss: MSE and TSE (Mean/Total Squared Error)
- The standard approach, and also ridge and LASSO regularization.
- $p > n$
- Logistic regression for classification.
- How these work, the maths, and the code.

As usual our emphasis will be on *doing* rather than *proving*: *just enough: progress at pace*

1.2 Assigned Reading

For this worksheet you are recommended Chapters 9 of [MML], Chapter 3 of [MLFCES], Chapter 7 of [IPDS],

- MML: Mathematics for Machine Learning, by Marc Peter Deisenroth, A. Aldo Faisal, and Cheng Soon Ong. Cambridge University Press. https://mml-book.github.io.
- MLFCES: Machine Learning: A First Course for Engineers and Scientists, by Andreas Lindholm, Niklas Wahlström, Fredrik Lindsten, Thomas B. Schön. Cambridge University Press. http://smlbook.org.

- IPDS: Introduction to Probability for Data Science, by Stanley H. Chan, https://probability4datascience.com

These can be accessed legally and without cost.

There are also these useful references for coding:

- PT: `python`: https://docs.python.org/3/tutorial
- NP: `numpy`: https://numpy.org/doc/stable/user/quickstart.html
- MPL: `matplotlib`: https://matplotlib.org

1.3 Context

So far we have been mainly concerned with **Classification**. This is a big part of Machine Learning but not the only part.

Classification is where we seek to categorise among a set of possible labels.

Regression is where we seek to determine a value from a continuous set of possible values.

In the limit of lots of labels, or a coarsely discretized set, the distinction between these becomes blurred and some models can then be used for both tasks.

1.4 The Basic Idea

The aim is to build a model that given x (a feature), predicts $y(x)$ (a label). We begin with this example data…

$$\begin{pmatrix} x_i \\ y_i \end{pmatrix} = \begin{pmatrix} 0 \\ 1 \end{pmatrix}, \begin{pmatrix} 1 \\ 3 \end{pmatrix}, \begin{pmatrix} 2 \\ 1 \end{pmatrix}, \begin{pmatrix} 4 \\ 3 \end{pmatrix}, \begin{pmatrix} 6 \\ 7 \end{pmatrix}, \begin{pmatrix} 7 \\ 5 \end{pmatrix}$$

We start the indexing at zero, for `python`, so, for example, $(x_3, y_3)^T = (4, 3)^T$ (again, we only use column vectors).

There are $N_p = 6$ training points.

By now we should be familiar with the words **training** and **testing**. In Machine Learning we talk about **training** when we build a model using a set of training data that has a particular set of features. In supervised learning these data points are also labelled. The label is what we want the model to predict from the features. After the model is built, **testing** refers to its use on a hold-out set in order to report on its success on unseen data.

Let's bring in our standard imports - with a new one: `linear_model` from `sklearn`

We then set these data points up in `numpy`.

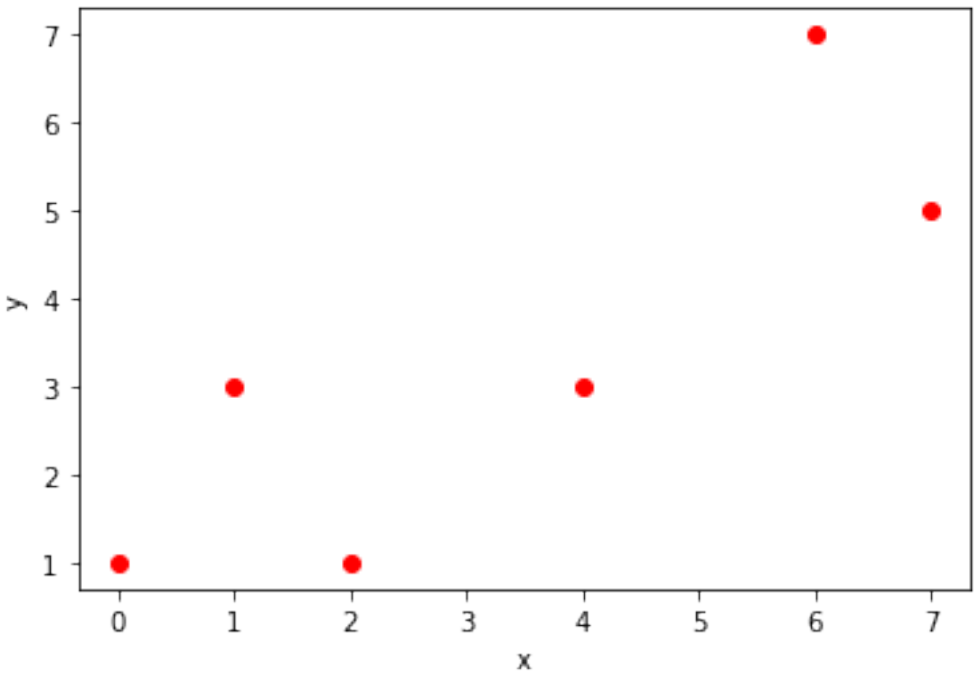
[1]:
```
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn import linear_model
```

[2]:
```
X_vals = np.array([[0,1,2,4,6,7]]).T
y_vals = np.array([[1,3,1,3,7,5]]).T
```

```
# this is the number of data points
Np = X_vals.shape[0]
```

We know these points represent a function, which will have a graph, but it isn't useful to join the points up as a graph yet because they may well contain noise. So we just plot the scatter…

[3]:
```
plt.plot(X_vals,y_vals,',r', marker='o')
plt.xlabel('x'); plt.ylabel('y');
```



Suppose we believe that $y = mx + c$ - a straight line. How do we find the **best** straight line?

This means, how do we find the **best** m and c? What do we mean by **best**?

We define $\boldsymbol{X}$, the **Feature Matrix** (recall that `numpy` indices start at zero), and the training input vector $\boldsymbol{y}$ as follows:

$$\boldsymbol{X} = \begin{pmatrix} 1 & x_0 \\ 1 & x_1 \\ 1 & x_2 \\ 1 & x_3 \\ \vdots & \vdots \\ 1 & x_{N_p-1} \end{pmatrix} \quad \text{and} \quad \boldsymbol{y} = \begin{pmatrix} y_0 \\ y_1 \\ y_2 \\ y_3 \\ \vdots \\ y_{N_p-1} \end{pmatrix}$$

Note that $\boldsymbol{X}$ is sometimes called the **Design Matrix**, or the **Matrix of Regressors**.

We now want to use this training data to construct a predictor $\hat{y}(x_i)$ of y_i where $\hat{y} = mx + c$ is the **best straight line** through the data. We write $\boldsymbol{\theta} = (c, m)^T$ and note that

$$\hat{\boldsymbol{y}} = \begin{pmatrix} \hat{y}(x_0) \\ \hat{y}(x_1) \\ \hat{y}(x_2) \\ \hat{y}(x_3) \\ \vdots \\ \hat{y}(x_{N_p-1}) \end{pmatrix} = \begin{pmatrix} mx_0 + c \\ mx_1 + c \\ mx_2 + c \\ mx_3 + c \\ \vdots \\ mx_{N_p-1} + c \end{pmatrix} = \begin{pmatrix} 1 & x_0 \\ 1 & x_1 \\ 1 & x_2 \\ 1 & x_3 \\ \vdots & \vdots \\ 1 & x_{N_p-1} \end{pmatrix} \begin{pmatrix} c \\ m \end{pmatrix} = \boldsymbol{X\theta}.$$

Notice that we can write $y = \boldsymbol{\theta}^T \boldsymbol{x}$ for $\boldsymbol{x} = (1, x)^T$ an **augmented (with unity) data point** for which we want to predict y.

2 Loss and Cost

To define **best** straight line we need the notions of **cost** and **loss**. The *best* will then be the *cheapest* as measured by this *cost*.

These terms are used in quite specific ways.

We have set up a straight line as $\hat{y}(x) = mx + c$

For each i we then measure the difference between y_i (what we want x_i to give), and $\hat{y}_i = mx_i + c$ (what x_i actually gives from the model).

We actually look at the squared difference because we want to eliminate the sign. This quantity, $\big(y_i - \hat{y}(x_i)\big)^2$ is what we call the **loss** - in this case, the *squared error loss*.

The loss depends on the choice of data point. To get the overall picture of how well the model is doing we add all these squared error losses together and divide by their number to get the average. This **mean-squared-error** (MSE) is called the **cost**. It is given by,

$$\mathcal{E} = \frac{1}{N_p} \sum_i |y_i - \hat{y}_i|^2 = \frac{1}{N_p} \sum_i |y_i - (mx_i + c)|^2 = \frac{1}{N_p} \|\boldsymbol{y} - \boldsymbol{X\theta}\|_2^2.$$

This is because the vector 2-norm sums up all the squares of the vector elements.

In [MLFCES, Chapter 3] they talk about the cost, as above, being the average of the squared losses, $y_i - \hat{y}$.

In [MML, Chapter 8] the cost is called the **empirical risk**.

This terminology is not always used consistently though. In [IPDS, Chapter 7] the term *loss* refers to $\|\boldsymbol{y} - \boldsymbol{X\theta}\|_2^2$. Although this isn't our cost because the average isn't taken.

Don't get too hung up on this. The MSE Cost s what we are going to work with.

And the process we are going to follow is essentially the same in all sources.

We will choose m and c, as captured in $\boldsymbol{\theta} = (c, m)^T$, such that this MSE cost (or empirical risk) is as small as it can be given this training set of data.

This is called **Ordinary Least Squares** (OLS) regression: we are minimizing the average of the squared error.

We have this *cost*, $\mathcal{E}(\boldsymbol{\theta}) = N_p^{-1}\|\boldsymbol{y} - \boldsymbol{X}\boldsymbol{\theta}\|_2^2$, and we want to find its minimum value. The data are fixed - the only variable we can control is $\boldsymbol{\theta}$.

Let's think about this: ideally we would like to determine the regression polynomial, the parameters in $\boldsymbol{\theta} = (c, m)^T$, so that they bring the cost to zero: $\mathcal{E}(\boldsymbol{\theta}) = 0$.

This may not be possible (why?), but we can at least ask for it to be as small as possible. (It can't be negative - so zero is the greatest lower bound.) Before we think about minimization let's play with this cost expression and see what it reveals.

Consider how the value of $\mathcal{E}$ changes between the point $\boldsymbol{\theta} \in \mathbb{R}^2$ and another point $\boldsymbol{\theta} + \boldsymbol{v} \in \mathbb{R}^2$. We need some algebra…

With $N_p\mathcal{E}(\boldsymbol{\theta}) = \|\boldsymbol{y}-\boldsymbol{X}\boldsymbol{\theta}\|_2^2$, we express $N_p\mathcal{E}(\boldsymbol{\theta}+\boldsymbol{v}) = \|\boldsymbol{y}-\boldsymbol{X}(\boldsymbol{\theta}+\boldsymbol{v})\|_2^2$ in terms of $N_p\mathcal{E}(\boldsymbol{\theta})+$'extras'. Recall that the 2-norm arises from the scalar product…

$$
\begin{aligned}
N_p\mathcal{E}(\boldsymbol{\theta} + \boldsymbol{v}) &= \|\boldsymbol{y} - \boldsymbol{X}\boldsymbol{\theta} - \boldsymbol{X}\boldsymbol{v}\|_2^2, && (1)\\
&= (\boldsymbol{y} - \boldsymbol{X}\boldsymbol{\theta} - \boldsymbol{X}\boldsymbol{v}, \boldsymbol{y} - \boldsymbol{X}\boldsymbol{\theta} - \boldsymbol{X}\boldsymbol{v}), && (2)\\
&= \|\boldsymbol{y} - \boldsymbol{X}\boldsymbol{\theta}\|_2^2 - 2(\boldsymbol{y} - \boldsymbol{X}\boldsymbol{\theta}, \boldsymbol{X}\boldsymbol{v}) + \|\boldsymbol{X}\boldsymbol{v}\|_2^2, && (3)\\
&= N_p\mathcal{E}(\boldsymbol{\theta}) + 2(\boldsymbol{X}\boldsymbol{\theta} - \boldsymbol{y}, \boldsymbol{X}\boldsymbol{v}) + \|\boldsymbol{X}\boldsymbol{v}\|_2^2. && (4)
\end{aligned}
$$

Next, in general, by **taking the transpose through**,

$$(\boldsymbol{v}, \boldsymbol{K}\boldsymbol{u}) = \boldsymbol{v}^T\boldsymbol{K}\boldsymbol{u} = (\boldsymbol{K}^T\boldsymbol{v})^T\boldsymbol{u} = (\boldsymbol{K}^T\boldsymbol{v}, \boldsymbol{u})$$

$$\text{We then have:} \qquad 2(\boldsymbol{X}\boldsymbol{\theta} - \boldsymbol{y}, \boldsymbol{X}\boldsymbol{v}) = 2(\boldsymbol{X}^T(\boldsymbol{X}\boldsymbol{\theta} - \boldsymbol{y}), \boldsymbol{v}).$$

We have therefore shown that

$$N_p\mathcal{E}(\boldsymbol{\theta} + \boldsymbol{v}) = N_p\mathcal{E}(\boldsymbol{\theta}) + 2(\boldsymbol{X}^T(\boldsymbol{X}\boldsymbol{\theta} - \boldsymbol{y}), \boldsymbol{v}) + \|\boldsymbol{X}\boldsymbol{v}\|_2^2.$$

Suppose now that we consider the particular case where $\boldsymbol{\theta}$ satisfies $\boldsymbol{X}^T(\boldsymbol{X}\boldsymbol{\theta} - \boldsymbol{y})$.

Then, any departure from $\boldsymbol{\theta}$ to a different point $\boldsymbol{\theta} + \boldsymbol{v}$ produces this cost,

$$
\begin{aligned}
\mathcal{E}(\boldsymbol{\theta} + \boldsymbol{v}) &= \mathcal{E}(\boldsymbol{\theta}) + \frac{2}{N_p}(\underbrace{\boldsymbol{X}^T(\boldsymbol{X}\boldsymbol{\theta} - \boldsymbol{y})}_{=0}, \boldsymbol{v}) + \frac{1}{N_p}\|\boldsymbol{X}\boldsymbol{v}\|_2^2 && (5)\\
&= \mathcal{E}(\boldsymbol{\theta}) + \frac{1}{N_p}\|\boldsymbol{X}\boldsymbol{v}\|_2^2 \geq \mathcal{E}(\boldsymbol{\theta}) \text{ for all non-zero } \boldsymbol{v}. && (6)
\end{aligned}
$$

HENCE: cost is minimized when $\boldsymbol{\theta}$ satisfies $\boldsymbol{X}^T\boldsymbol{X}\boldsymbol{\theta} = \boldsymbol{X}^T\boldsymbol{y}$.

These are called the **Normal Equations**.

2.0.1 The Normal Equations

We refer to the system

$$\boldsymbol{X}^T\boldsymbol{X}\boldsymbol{\theta} = \boldsymbol{X}^T\boldsymbol{y}$$

as *the normal equations*. The matrix $\boldsymbol{X}^T\boldsymbol{X}$ in our example above is

$$\boldsymbol{X}^T\boldsymbol{X} = \begin{pmatrix} 1 & 1 & 1 & 1 & \cdots & 1 \\ x_0 & x_1 & x_2 & x_3 & \cdots & x_{N_p-1} \end{pmatrix} \begin{pmatrix} 1 & x_0 \\ 1 & x_1 \\ 1 & x_2 \\ 1 & x_3 \\ \vdots & \vdots \\ 1 & x_{N_p-1} \end{pmatrix} = \begin{pmatrix} N_p & \sum_i x_i \\ \sum_i x_i & \boldsymbol{x}^T\boldsymbol{x} \end{pmatrix}$$

where $\boldsymbol{x} = (x_0, x_1, x_2, \ldots)^T$.

2.0.2 Linear Least Squares Regression Solved

Given the data set $\boldsymbol{x} = (x_0, x_1, \ldots)^T$ and $\boldsymbol{y} = (y_0, y_1, \ldots)^T$, where for example earlier we had,

$$\begin{pmatrix} x_i \\ y_i \end{pmatrix} = \begin{pmatrix} 0 \\ 1 \end{pmatrix}, \begin{pmatrix} 1 \\ 3 \end{pmatrix}, \begin{pmatrix} 2 \\ 1 \end{pmatrix}, \begin{pmatrix} 4 \\ 3 \end{pmatrix}, \begin{pmatrix} 6 \\ 7 \end{pmatrix}, \begin{pmatrix} 7 \\ 5 \end{pmatrix},$$

we form the design matrix $\boldsymbol{X} = (\boldsymbol{1}, \boldsymbol{x})$ (here $\boldsymbol{1}$ denotes a column of 1's).

The solution $\boldsymbol{\theta} = (c, m)^T$ to the normal equations $\boldsymbol{X}^T\boldsymbol{X}\boldsymbol{\theta} = \boldsymbol{X}^T\boldsymbol{y}$ then produces a straight line $\hat{y}(x) = mx + c$ through these data points.

This line is the **best straight line through these data** in the sense of minimum (MSE) cost $N_p^{-1}\mathcal{E}(\boldsymbol{\theta}) = \|\boldsymbol{X}\boldsymbol{\theta} - \boldsymbol{y}\|_2^2$.

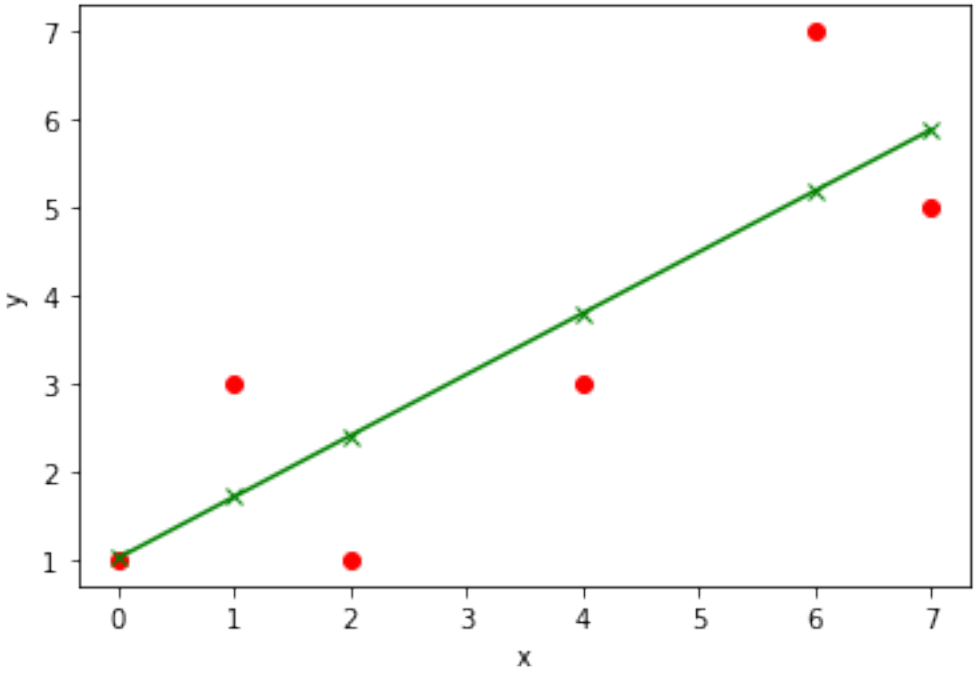
Let's do this in `numpy`. we already have `X_vals` and `y_vals` set up.

[4]:
```
print(f'X_vals = {X_vals.T} and y_vals = {y_vals.T}')
```

```
X_vals = [[0 1 2 4 6 7]] and y_vals = [[1 3 1 3 7 5]]
```

[5]:
```
# linear polynomial regression: yhat1 = mx+c
# set up the design matrix
X = np.c_[np.ones([Np,1]), X_vals]
# solve the normal equations
theta1 = theta = np.linalg.solve(X.T @ X, X.T @ y_vals)
print(f'c = {theta[0]} and m = {theta[1]}')
plt.plot(X_vals, y_vals,'.r', marker='o')
y_hat1 = theta[0] + theta[1]*X_vals
plt.plot(X_vals, y_hat1,'g', marker='x')
plt.xlabel('x'); plt.ylabel('y');
```

```
c = [1.01694915] and m = [0.69491525]
```



2.0.3 Higher Degree Polynomial Regression

Again, we are given data $\boldsymbol{x} = (x_0, x_1, \ldots)^T$ and $\boldsymbol{y} = (y_0, y_1, \ldots)^T$, but this time we want to fit the **best degree-n** polynomial through the points.

This means that we want $\boldsymbol{\theta} = (\theta_0, \theta_1, \ldots, \theta_n)^T$ such that

$$\hat{y}(x) = \theta_0 + \theta_1 x + \theta_2 x^2 + \cdots + \theta_n x^n$$

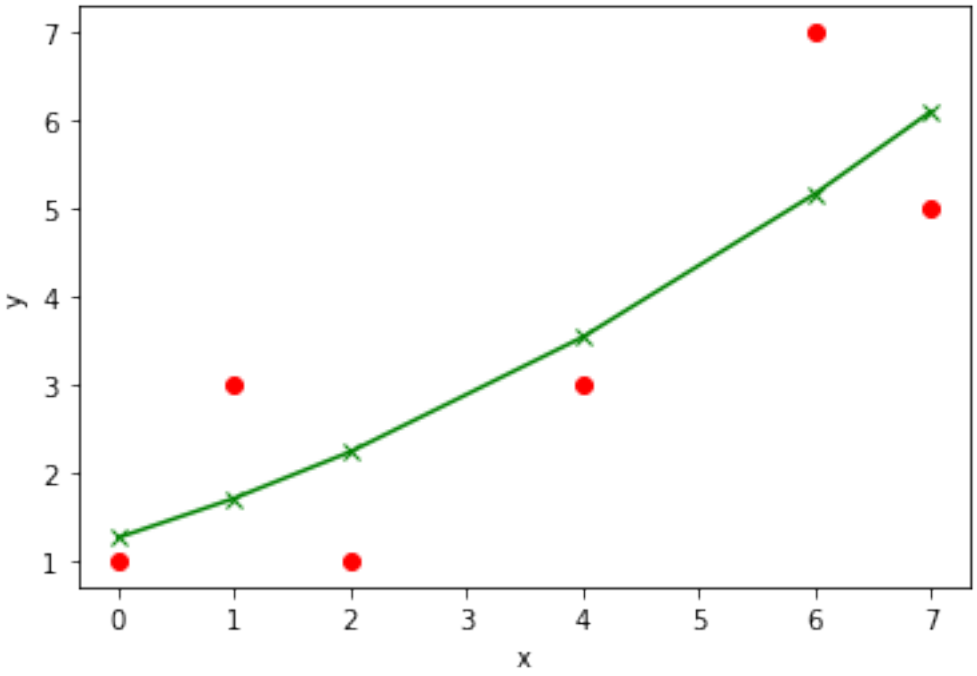
predicts y with minimum MSE cost $\mathcal{E}(\boldsymbol{\theta}) = N_p^{-1} \|\boldsymbol{X}\boldsymbol{\theta} - \boldsymbol{y}\|_2^2$. This time, the design matrix looks like this,

$$\boldsymbol{X} = \begin{pmatrix} 1 & x_0 & x_0^2 & \cdots & x_0^n \\ 1 & x_1 & x_1^2 & \cdots & x_1^n \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & x_{N_p-1} & x_{N_p-1}^2 & \cdots & x_{N_p-1}^n \end{pmatrix}.$$

Exactly the same reasoning as above tells us that $\boldsymbol{\theta}$ solves the normal equations: $\boldsymbol{X}^T\boldsymbol{X}\boldsymbol{\theta} = \boldsymbol{X}^T\boldsymbol{y}$.

[6]:
```
# quadratic polynomial regression: stack up the design matrix
X = np.hstack( (np.ones([Np,1]), X_vals, X_vals**2) )
# solve the normal equations
theta = np.linalg.solve(X.T @ X, X.T @ y_vals)
```

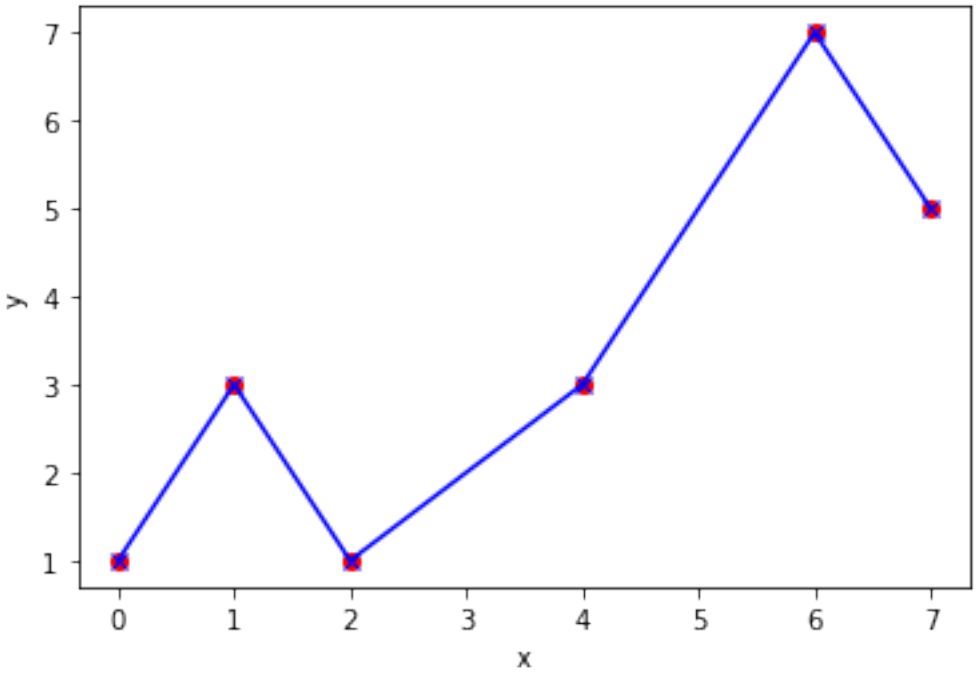
```
plt.plot(X_vals,y_vals,'.r',marker='o')
#y_hat2 = theta[0] + theta[1]*X_vals + theta[2]*X_vals*X_vals
# the line above is not wrong, but this is more elegant...
y_hat2 = X @ theta
plt.plot(X_vals, y_hat2,'g', marker='x')
plt.xlabel('x'); plt.ylabel('y');
```



It doesn't take much though to see that we can go to arbitrary polynomial degree. **BUT SHOULD WE?**

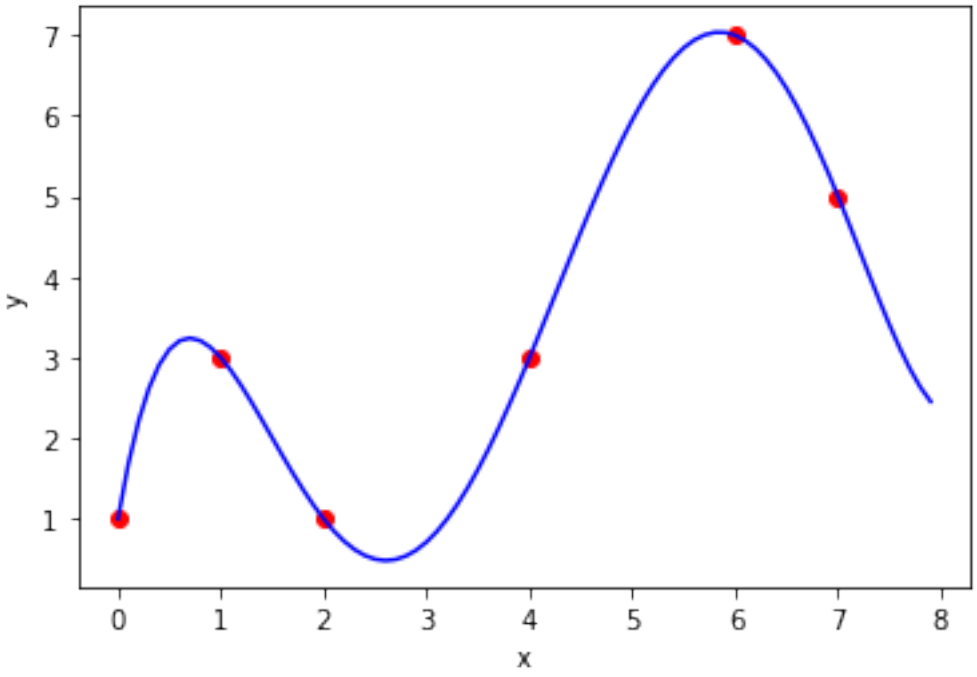
There are six data points - and a quintic can fit to these exactly. (Why?)

[7]:
```
# quintic polynomial regression: stack up the design matrix
X = np.ones([Np,1])
for k in range(1,6): X = np.hstack( (X, X_vals**k) )
theta = np.linalg.solve(X.T @ X, X.T @ y_vals)
y_hat5 = X @ theta
plt.plot(X_vals, y_vals,'.r',marker='o')
plt.plot(X_vals, y_hat5,'b',marker='x')
plt.xlabel('x'); plt.ylabel('y');
```



This is no good - we need many more x values to see the smooth curve.

[8]:
```
plt.plot(X_vals,y_vals,'.r',marker='o')
# a fine grid of x-values, and a re-built design matrix with them
X_grid = np.arange(0,1+X_vals[Np-1],0.1).reshape(-1,1)
X = np.c_[np.ones([X_grid.shape[0],1]), X_grid]
for k in range(2,Np): X = np.c_[X, np.power(X_grid,k)]
y_hat5 = X @ theta
plt.plot(X_grid, y_hat5,'b')
plt.xlabel('x'); plt.ylabel('y');
```



This seems perfect. Are you happy with this model for predicting y?

What about new data arriving? Will this be a good predictor? There is a danger that it is severly **over-fitted** to the training data.

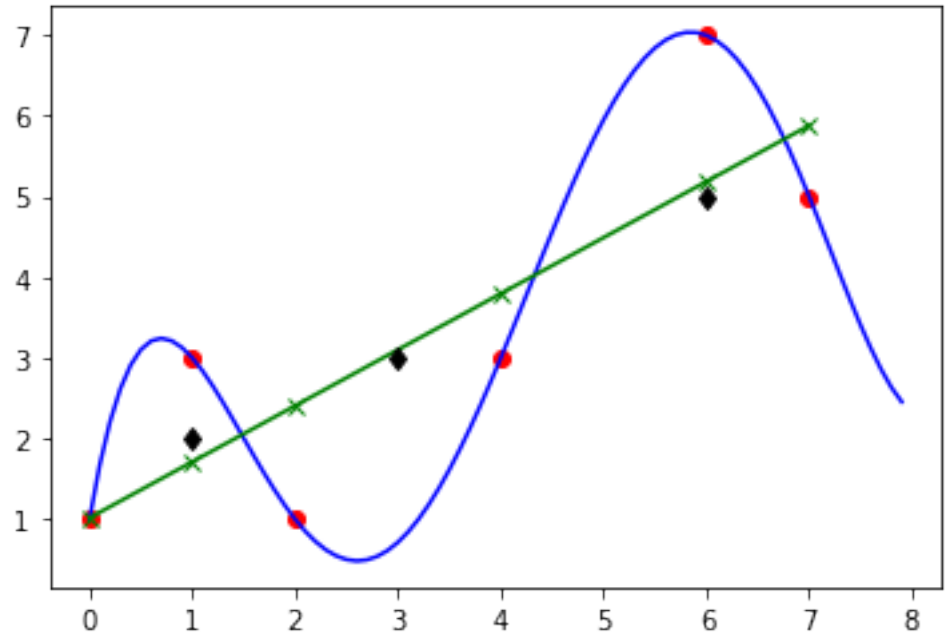
Suppose in testing the model these new unseen data points arrive

$$\begin{pmatrix} x_i \\ y_i \end{pmatrix} = \begin{pmatrix} 1 \\ 2 \end{pmatrix}, \begin{pmatrix} 3 \\ 3 \end{pmatrix}, \begin{pmatrix} 6 \\ 5 \end{pmatrix}.$$

Let's plot them with black diamonds. The linear regression model is in green and the quintic one in blue. The original training data is red.

[9]:
```
# Overfitting - generalization
X_new = np.array([[1,3,6]]).T
y_new = np.array([[2,3,5]]).T
plt.plot(X_vals, y_vals,'.r', marker='o')
plt.plot(X_grid, y_hat5,'b')
plt.plot(X_vals, y_hat1,'g', marker='x')
plt.plot(X_new,  y_new, 'dk')
```

[9]:
```
[<matplotlib.lines.Line2D at 0x7ffbd28ad0b8>]
```



COMMENTS?

What we are seeing here is an example of **overfitting**. The training set is modelled perfectly but at the expense of the model being able to **generalise** to unseen data.

For this reason a *validation* hold out set is often introduced into a ML workflow in order to capture such undesirable properties in models.

2.0.4 Linear Regression in Scikit-Learn

Let's see how to implement the standard OLS **linear** regression model in the `sklearn` library.

The details can be found here https://scikit-learn.org/stable/modules/linear_model.html#ordinary-least-squares

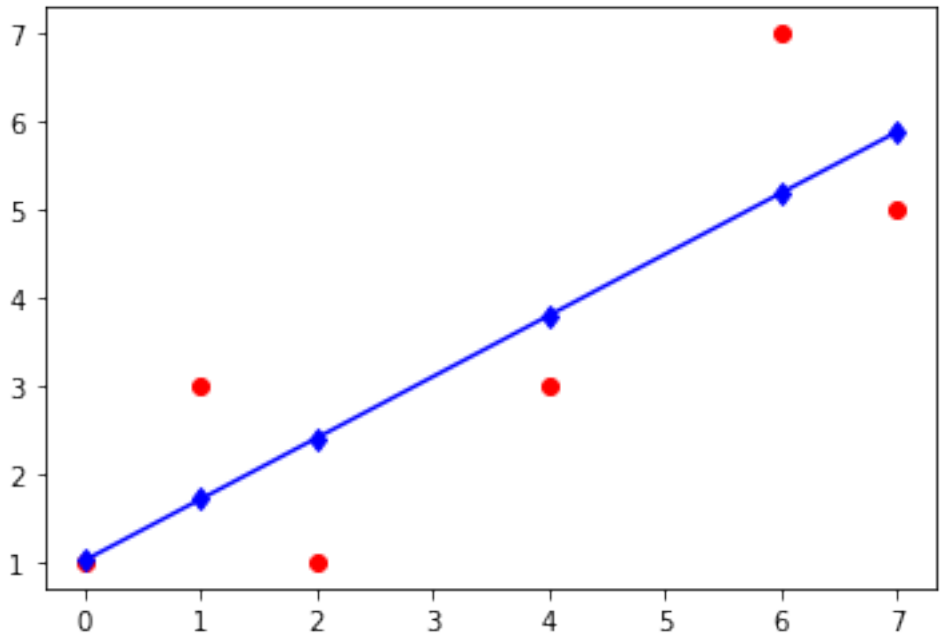
[10]:
```
# standard regression
reg_linear = linear_model.LinearRegression()
reg_linear.fit(X_vals, y_vals)
print('reg_coef_ = ', reg_linear.coef_)
print('reg_intercept_ = ', reg_linear.intercept_)
print(f'Our earlier values were: c = {theta1[0]} and m = {theta1[1]}')
```

```
reg_coef_ =  [[0.69491525]]
reg_intercept_ =  [1.01694915]
Our earlier values were: c = [1.01694915] and m = [0.69491525]
```

We can plot using the model's predicted values...

[11]:
```
# Make predictions from the fitted model
y_pred = reg_linear.predict(X_vals)
plt.plot(X_vals,y_vals,'.r',marker='o')
plt.plot(X_vals,y_pred,'b',marker='d')
#plt.plot(X_vals,y_hat, 'g',marker='x', markersize=20)
```

[11]:
```
[<matplotlib.lines.Line2D at 0x7ffb80a17748>]
```



3 Regularization

We're now going to explore the topic of *regularization*. This is a useful concept when the problem we are trying to solve is in some sense *ill posed*, or likely to suggest spurious solutions. This can happen when for example we get a lot of data points clustered around a straight line and a few way off. The ones way off of the line - the *outliers* - may well result from measurement errors, or other erroneous data and yet may result in the straight line regressor that we compute being altered to one *nearby* but different to the one we actually want.

The following discussion is distilled from

> Chapter 9, Section 1, **Theoretical Numerical Analysis: an introduction to advanced techniques**, by Peter Linz, Dover 2001.

You don't need to track that down and read it - the main ideas follow.

To eliminate unwanted spurious, or *nearby*, solutions, Andrey Nikolayevich Tikhonov - see e.g. https://en.wikipedia.org/wiki/Andrey_Nikolayevich_Tikhonov - put forward an idea that is now known as *Tikhonov regularization* - see e.g. https://en.wikipedia.org/wiki/Ridge_

regression#Tikhonov_regularization.

The idea is to add an extra term to the cost. In regression we typically a term proportional to the ℓ_2 or ℓ_1 norm of the parameters. So, with $p = 2$ or $p = 1$ we write our cost as

$$\mathcal{E}_\alpha(\boldsymbol{\theta}) = \|\boldsymbol{y} - \boldsymbol{X}\boldsymbol{\theta}\|_2^2 + \alpha\|\boldsymbol{\theta}\|_p^p.$$

Here we are working with **Total Squared Error** (TSE) just to keep the formulae simpler. It's easy to divide by the number of data points to get the MSE cost as above.

In this TSE cost, $\alpha \geq 0$, is the **regularization parameter**. Note that $\alpha = 0$ gets us back to where we were, and it makes no sense to allow $\alpha < 0$.

There three cases of interest:

- $p = 2$: Ridge regression
- $p = 1$: LASSO (*Least Absolute Shrinkage and Selection Operator*)
- $p = 1$ and $p = 2$ Elastic net - see e.g. https://en.wikipedia.org/wiki/Elastic_net_regularization

It takes a lot of work to properly understand why this regularization can be useful, but let's try and develop some intuition.

The best case scenario is where we can find $\boldsymbol{\theta}$ such that $\boldsymbol{y} = \boldsymbol{X}\boldsymbol{\theta}$. If in this case $\boldsymbol{\psi}$ minimizes $\mathcal{E}_\lambda$, then

$$\|\boldsymbol{y} - \boldsymbol{X}\boldsymbol{\psi}\|_2^2 \leq \mathcal{E}_\alpha(\boldsymbol{\psi}) \leq \mathcal{E}_\alpha(\boldsymbol{\theta}) = \alpha\|\boldsymbol{\theta}\|_p^p$$

which means we have some control over $\boldsymbol{\psi}$. In particular, though it is not necessarily the case that $\boldsymbol{y} = \boldsymbol{X}\boldsymbol{\psi}$, this suggests that by taking α small we can make $\|\boldsymbol{y} - \boldsymbol{X}\boldsymbol{\psi}\|_2$ as small as we please.

On the other hand though, if we do make α small then we get back to the original least squares regression - and we presumably don't want that because if we did we wouldn't have introduced α in the first place.

If $\boldsymbol{\psi}$ minimizes $\mathcal{E}_\alpha$, then, when $p = 2$, we have already seen that $\boldsymbol{X}^T\boldsymbol{X}\boldsymbol{\psi} + \alpha\boldsymbol{\psi} = \boldsymbol{X}^T\boldsymbol{y}$.

Using this, as we will have seen before,

$$\begin{aligned}
\mathcal{E}_\alpha(\boldsymbol{\psi} + \boldsymbol{v}) &= \|\boldsymbol{y} - \boldsymbol{X}\boldsymbol{\psi} - \boldsymbol{X}\boldsymbol{v}\|_2^2 + \alpha\|\boldsymbol{\psi} + \boldsymbol{v}\|_2^2, && (7)\\
&= \|\boldsymbol{y} - \boldsymbol{X}\boldsymbol{\psi}\|_2^2 - 2(\boldsymbol{y} - \boldsymbol{X}\boldsymbol{\psi}, \boldsymbol{X}\boldsymbol{v}) + \|\boldsymbol{X}\boldsymbol{v}\|_2^2 + \alpha\|\boldsymbol{\psi}\|_2^2 + 2\alpha(\boldsymbol{\psi}, \boldsymbol{v}) + \alpha\|\boldsymbol{v}\|_2^2, && (8)\\
&= \mathcal{E}_\alpha(\boldsymbol{\psi}) + 2(\boldsymbol{X}\boldsymbol{\psi} - \boldsymbol{y}, \boldsymbol{X}\boldsymbol{v}) + 2\alpha(\boldsymbol{\psi}, \boldsymbol{v}) + \|\boldsymbol{X}\boldsymbol{v}\|_2^2 + \alpha\|\boldsymbol{v}\|_2^2, && (9)\\
&= \mathcal{E}_\alpha(\boldsymbol{\psi}) + 2(\boldsymbol{X}^T(\boldsymbol{X}\boldsymbol{\psi} - \boldsymbol{y}), \boldsymbol{v}) + 2\alpha(\boldsymbol{\psi}, \boldsymbol{v}) + \|\boldsymbol{X}\boldsymbol{v}\|_2^2 + \alpha\|\boldsymbol{v}\|_2^2, && (10)\\
&= \mathcal{E}_\alpha(\boldsymbol{\psi}) + 2(\underbrace{\boldsymbol{X}^T(\boldsymbol{X}\boldsymbol{\psi} - \boldsymbol{y}) + \alpha\boldsymbol{\psi}}_{=0}, \boldsymbol{v}) + \|\boldsymbol{X}\boldsymbol{v}\|_2^2 + \alpha\|\boldsymbol{v}\|_2^2, && (11)\\
&= \mathcal{E}_\alpha(\boldsymbol{\psi}) + \|\boldsymbol{X}\boldsymbol{v}\|_2^2 + \alpha\|\boldsymbol{v}\|_2^2. && (12)
\end{aligned}$$

So: $\mathcal{E}_\alpha(\boldsymbol{\psi} + \boldsymbol{v}) = \|\boldsymbol{y} - \boldsymbol{X}\boldsymbol{\psi} - \boldsymbol{X}\boldsymbol{v}\|_2^2 = \mathcal{E}_\alpha(\boldsymbol{\psi}) + \|\boldsymbol{X}\boldsymbol{v}\|_2^2 + \lambda\|\boldsymbol{v}\|_2^2$.

This is telling us that if we have spurious solutions, and if they are somehow *nearby* $\boldsymbol{\psi}$ in the sense that we can reach them with $\boldsymbol{v}$ being quite small, then the cost can be made significantly different, even for a nearby solution, by choosing α large.

Here, then, is the dilemma:

> α small keeps us near to the original problem, but this problem might be hard to solve because of nearby spurious solutions.
>
> α large makes it easier to find a well defined minimum cost, but this cost is may be some distance from the one we actually want minimized.

In the end, we often have to proceed by trial and error.

> **NOTE:** there is no simple manipulation like that above for $p = 1$. This is because the 2 norm arises from an inner product, whereas the 1 norm has no useful alternative representation.

Let's see how to implement **Ridge** and **LASSO** in `sklearn`.

[12]:
```
print('previous reg_coef_ = ', reg_linear.coef_, end=', ')
print('previous reg_intercept_ = ', reg_linear.intercept_)
# ridge regression with
reg_ridge = linear_model.Ridge(alpha=0.5)
reg_ridge.fit(X_vals, y_vals)
# Make predictions using the testing set
y_hat_ridge = reg_linear.predict(X_vals)
print('reg_coef_ = ', reg_ridge.coef_, end=', ')
print('reg_intercept_ = ', reg_ridge.intercept_)
```

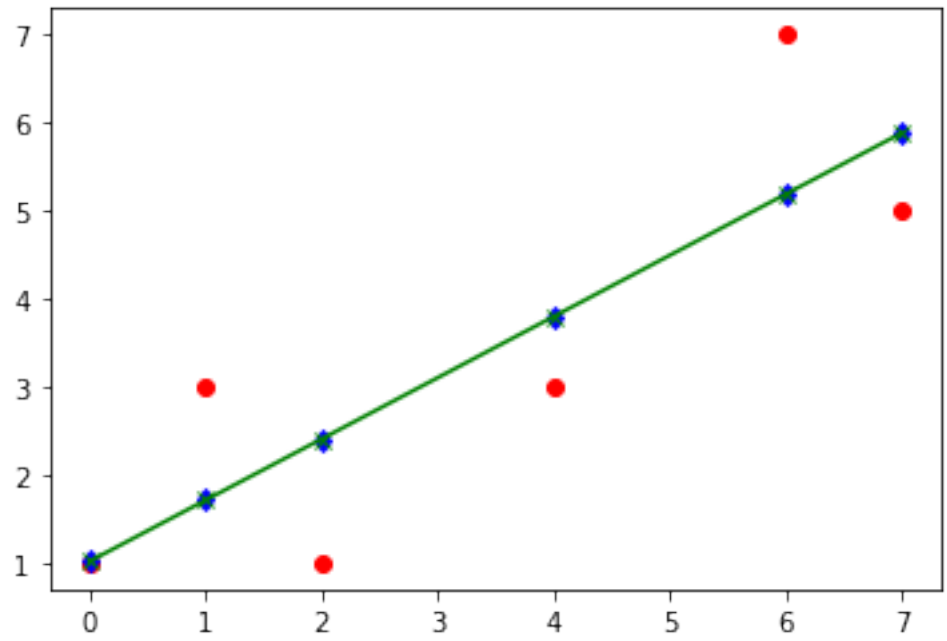
```
previous reg_coef_ =  [[0.69491525]], previous reg_intercept_ =  [1.01694915]
reg_coef_ =  [[0.68619247]], reg_intercept_ =  [1.0460251]
```

The coefficients differ, but only slightly.

Here are the training data in red, original OLS linear predictor in blue, and the ridge predictor in green.

[13]:
```
plt.plot(X_vals, y_vals,'.r',marker='o')
plt.plot(X_vals, y_hat1,'.b',marker='d')
plt.plot(X_vals, y_hat_ridge,'g',marker='x')
```

[13]:
```
[<matplotlib.lines.Line2D at 0x7ffbc02ff438>]
```



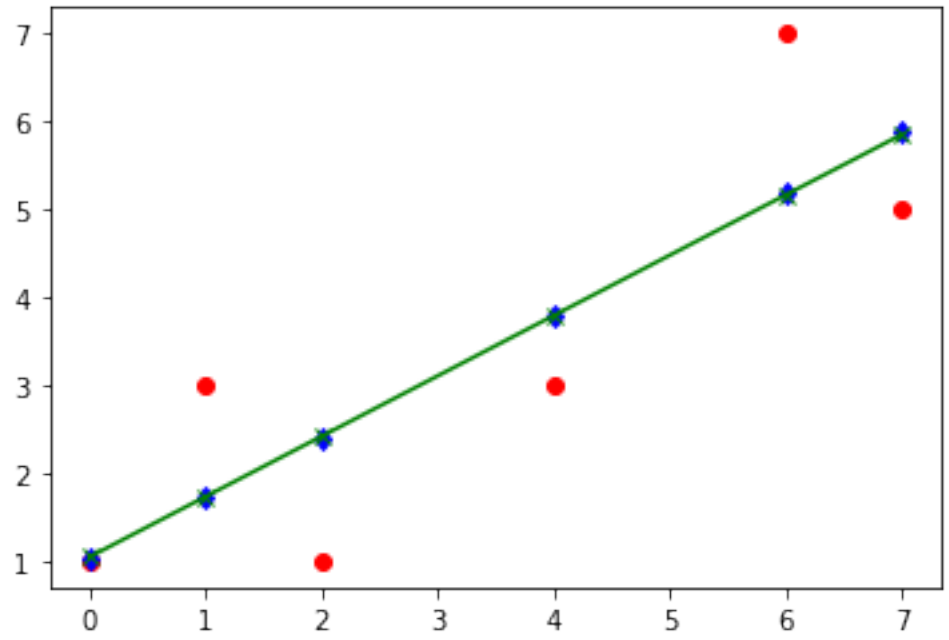
We can do the comparisons all in one cell.

[14]:
```
# standard regression
reg_linear = linear_model.LinearRegression()
reg_linear.fit(X_vals, y_vals)
# Make predictions
y_hatOL = reg_linear.predict(X_vals)
# ridge regression
reg_ridge = linear_model.Ridge(alpha=0.5)
reg_ridge.fit(X_vals, y_vals)
# Make predictions
y_hatR = reg_ridge.predict(X_vals)
print('linear reg_coef_ = ', reg_linear.coef_)
print('linear reg_intercept_ = ', reg_linear.intercept_)
print('ridge reg_coef_ = ', reg_ridge.coef_)
print('ridge reg_intercept_ = ', reg_ridge.intercept_)
```

```
linear reg_coef_ =  [[0.69491525]]
linear reg_intercept_ =  [1.01694915]
ridge reg_coef_ =  [[0.68619247]]
ridge reg_intercept_ =  [1.0460251]
```

[15]:
```
plt.plot(X_vals,y_vals,'.r',marker='o')
plt.plot(X_vals,y_hatOL,'.b',marker='d')
plt.plot(X_vals,y_hatR,'g',marker='x')
```

[15]: [<matplotlib.lines.Line2D at 0x7ffba0cf8860>]



For LASSO we can just alter a few bits of code like this...

[16]:
```
print('linear reg_coef_ = ', reg_linear.coef_)
print('linear reg_intercept_ = ', reg_linear.intercept_)
print('ridge reg_coef_ = ', reg_ridge.coef_)
print('ridge reg_intercept_ = ', reg_ridge.intercept_)
# standard regression
reg_linear = linear_model.LinearRegression()
reg_linear.fit(X_vals, y_vals)
# Make predictions
y_hatOL = reg_linear.predict(X_vals)
# LASSO regression
reg_lasso = linear_model.Lasso(alpha=.5)
reg_lasso.fit(X_vals, y_vals)
# Make predictions
y_hatL = reg_lasso.predict(X_vals)
print('LASSO reg_coef_ = ', reg_lasso.coef_)
print('LASSO reg_intercept_ = ', reg_lasso.intercept_)
```

```
linear reg_coef_ =  [[0.69491525]]
linear reg_intercept_ =  [1.01694915]
ridge reg_coef_ =  [[0.68619247]]
ridge reg_intercept_ =  [1.0460251]
LASSO reg_coef_ =  [0.61864407]
```

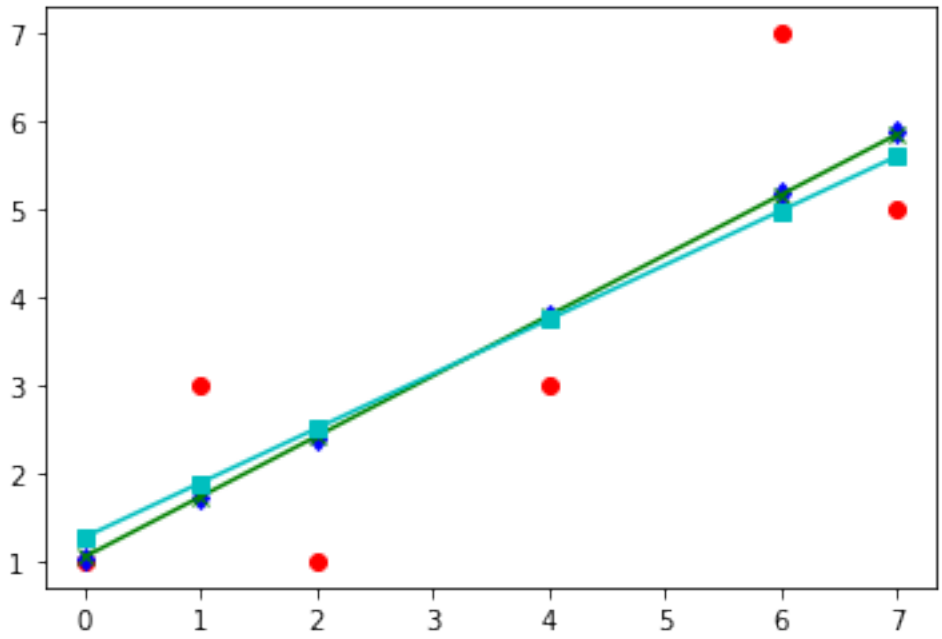
```
LASSO reg_intercept_ =  [1.27118644]
```

This time the coefficient differences are more noticeable.

Here are the training data in red, original OLS linear predictor in blue, the ridge predictor in green, and the LASSO predictor in cyan.

[17]:
```
plt.plot(X_vals,y_vals,'.r',marker='o')
plt.plot(X_vals,y_hatOL,'.b',marker='d')
plt.plot(X_vals,y_hatR,'g',marker='x')
plt.plot(X_vals,y_hatL,'c',marker='s')
```

[17]:
```
[<matplotlib.lines.Line2D at 0x7ffb80b29278>]
```



3.0.1 Summary

Standard OLS linear regression is ubiquitous. A regularized version is used when there are particular needs. We'll return to this point below when we have discussed multi-variate linear regression.

3.0.2 Multivariate Linear Regression

It is common that there is more than independent variable in play and that we want a linear predictor for y as a function of these many independent variables. In this case we could think of,

$$\hat{y}(x) = \theta_0 + \theta_1 x_1 + \theta_2 x_2 + \cdots + \theta_p x_p.$$

Note that this is quite different to the expression above for a high-degree polynomial. Don't get these notions confused.

In this case we would still expect to have training data, y_i and $\boldsymbol{x}_i = (x_{1,i}, x_{2,i}, \ldots, x_{p,i})^T$ (for $i = 1, \ldots, n$ say), that can be placed in a design matrix, and so we assume that we have,

$$\boldsymbol{X} = \begin{pmatrix} 1 & x_{1,1} & x_{2,1} & \cdots \\ 1 & x_{1,2} & x_{2,2} & \cdots \\ 1 & x_{1,3} & x_{2,3} & \cdots \\ \vdots & \vdots & \vdots & \ddots \end{pmatrix}, \quad \boldsymbol{\theta} = \begin{pmatrix} \theta_0 \\ \theta_1 \\ \theta_2 \\ \theta_3 \\ \vdots \end{pmatrix} \quad \text{and} \quad \boldsymbol{y} = \begin{pmatrix} y_1 \\ y_2 \\ y_3 \\ \vdots \end{pmatrix}$$

We can still set up the MSE cost, $\mathcal{E} = N_p^{-1} \|\boldsymbol{y} - \boldsymbol{X}\boldsymbol{\theta}\|_2^2$ and show as before that we can mimimize it if we can solve the normal equations.

The details can be found in the recommended reading.

3.0.3 Overview

Standard OLS linear regression is ubiquitous. The regularized versions are used when there are particular needs.

- Ridge: this is often used to prevent *overfitting* - the trap that we discussed above where we lose the the model's ability to generalise to unseen data.
- LASSO: the ℓ_1 norm encourages *sparsity*. This means that the vector of regression coefficents, $\boldsymbol{\theta}$, may contain several zeros. This propensity of the LASSO regularizer is a useful **feature selection** technique. This is useful for multi-variable regression (as above).

For more discussion see [MLFCES, Chapter 3.3], [MML, Chap 9.2.4] and, in particular, [IPDS, Chapter 7.4].

3.0.4 The $p > n$ Issue

Suppose that $\hat{y}(x) = \theta_0 + \theta_1 x_1 + \theta_2 x_2 + \cdots + \theta_p x_p$. with n observations. Then…

For $i = 1, \ldots, n$ we have training data, y_i and $\boldsymbol{x}_i = (x_{1,i}, x_{2,i}, \ldots, x_{p,i})^T$…

$$\boldsymbol{X} = \begin{pmatrix} 1 & x_{1,1} & x_{2,1} & \cdots & \cdots x_{p,1} \\ 1 & x_{1,2} & x_{2,2} & \cdots & \cdots x_{p,2} \\ 1 & x_{1,3} & x_{2,3} & \cdots & \cdots x_{p,3} \\ \vdots & \vdots & \vdots & \ddots & \\ 1 & x_{1,n} & x_{2,n} & \cdots & \cdots x_{p,n} \end{pmatrix}, \quad \boldsymbol{\theta} = \begin{pmatrix} \theta_0 \\ \theta_1 \\ \theta_2 \\ \vdots \\ \theta_p \end{pmatrix} \quad \text{and} \quad \boldsymbol{y} = \begin{pmatrix} y_1 \\ y_2 \\ \vdots \\ y_n \end{pmatrix}$$

The normal equations are $\boldsymbol{X}^T\boldsymbol{X}\boldsymbol{\theta} = \boldsymbol{X}^T\boldsymbol{y}$ with $\boldsymbol{X}^T\boldsymbol{X}$ is a p by p square symmetric matrix, and $\boldsymbol{X}^T\boldsymbol{y}$ an p by 1 column vector.

THEOREM: $\boldsymbol{X}^T\boldsymbol{X}$ is invertible if and only if $\boldsymbol{X}$ has full column rank.

In our earlier terms the column rank is the number of independent equations that the columns can describe.

The row rank is the number of independent equations that the rows can describe.

THEOREM: The row and column ranks of a matrix are identical.

Why? Use the SVD: $\boldsymbol{K} = \boldsymbol{U}\boldsymbol{\Sigma}\boldsymbol{V}^T$ and so $\boldsymbol{K}^T = \boldsymbol{V}\boldsymbol{\Sigma}^T\boldsymbol{U}^T$. In both cases number of non-zero singular values (the rank) is the same. The rank is equal to the row rank because (for us) they both mean the same thing (the number of independent equations).

WARNING: (1) if $p > n$ then the column rank of $\boldsymbol{X}$ cannot exceed n.

WARNING: (2) if $p > n$ then then $\boldsymbol{X}$ does not have the full column rank of p.

THEREFORE: if $p > n$ then $\boldsymbol{X}^T\boldsymbol{X}$ is **not invertible** and so the Normal Equations cannot be solved in the sense that we have meant above where we took $\boldsymbol{\theta} = \left(\boldsymbol{X}^T\boldsymbol{X}\right)^{-1}\boldsymbol{X}^T\boldsymbol{y}$.

See [IPDS, Chapter 7.1.4] for a deeper treatment.

Fortunately there are other ways that we can seek to minimize the cost. The `sklearn` routines that we introduced will be able to do this.

This is another good reason to use reputable software libraries rather than always write your own code.

3.1 Logistic Regression

We can use these regression ideas for binary classification. The idea is to use a linear regression to create a decision boundary between the two classes.

The logistic, or sigmoid, function can then be used to create a switch from **off** to **on** as we cross the decision boundary.

Here, for $a \in \mathbb{R}$, is the logistic function, often called the **sigmoid**:

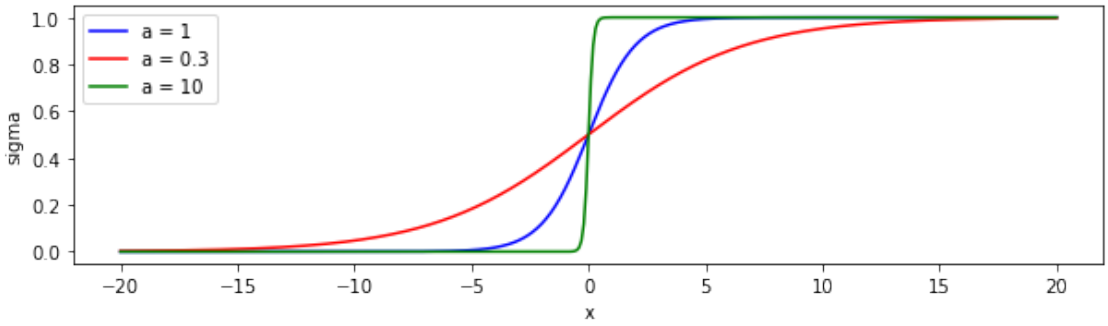
$$\sigma(x \mid a) = \frac{1}{1 + \exp(-ax)}$$

In Particular: the sigmoid function can model a **probability**.

[18]:
```
def sigma(x, a):
    return (1+np.exp(-a*x))**(-1)

x_vals = np.arange(-20, 20.1, 0.1)
y_vals_1 = sigma(x_vals, 1)
y_vals_03 = sigma(x_vals, 0.3)
y_vals_10 = sigma(x_vals, 10)

plt.figure(figsize=(10,4)); plt.gca().set_aspect(10)
plt.plot(x_vals, y_vals_1, color='blue', label='a = 1')
plt.plot(x_vals, y_vals_03, color='red', label='a = 0.3')
plt.plot(x_vals, y_vals_10, color='green', label='a = 10')
plt.xlabel('x'); plt.ylabel('sigma');
plt.legend()
plt.show()
```



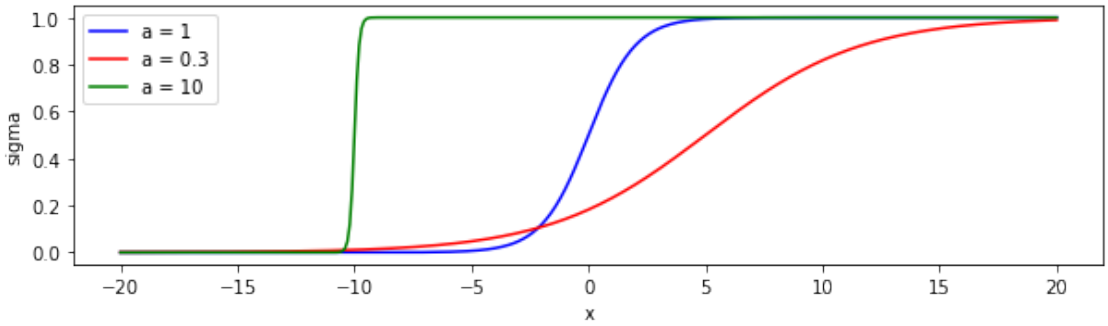
It can be scaled by the choice of a, and also translated by x_0:

$$\sigma(x \mid a, x_0) = \frac{1}{1 + \exp\big(-a(x - x_0)\big)}$$

[19]:
```
def sigma(x, a, x0):
    return (1+np.exp(-a*(x-x0)))**(-1)

x_vals = np.arange(-20, 20.1, 0.1)
y_vals_0 = sigma(x_vals, 1, 0)
y_vals_5 = sigma(x_vals, 0.3, 5)
y_vals_m10 = sigma(x_vals, 10, -10)

plt.figure(figsize=(10,4)); plt.gca().set_aspect(10)
plt.plot(x_vals, y_vals_0, color='blue', label='a = 1')
plt.plot(x_vals, y_vals_5, color='red', label='a = 0.3')
plt.plot(x_vals, y_vals_m10, color='green', label='a = 10')
plt.xlabel('x'); plt.ylabel('sigma');
plt.legend()
plt.show()
```



In 2D we can write this kind of thing

$$\sigma(x_1, x_2 \mid a, b, c) = \frac{1}{1 + \exp\big(-(ax_1 + bx_2 + c)\big)}$$

For example, with $a = -1$, $b = 1$ and $c = 0$ we have,

$$\sigma(x_1, x_2 \mid -1, 1, 0) = \frac{1}{1 + \exp\big(-(x_2 - x_1)\big)}$$

Now, along the line $x - 1 = x_2$ we have $\sigma(x_1, x_2 \mid -1, 1, 0) = 0.5$.

To one side of the line $\sigma \to 0$ and to the other side of the line $\sigma \to 1$. This means we have a switch, or a signal, that we can think of as being off on one side of the line and switched on if we are on the other side.

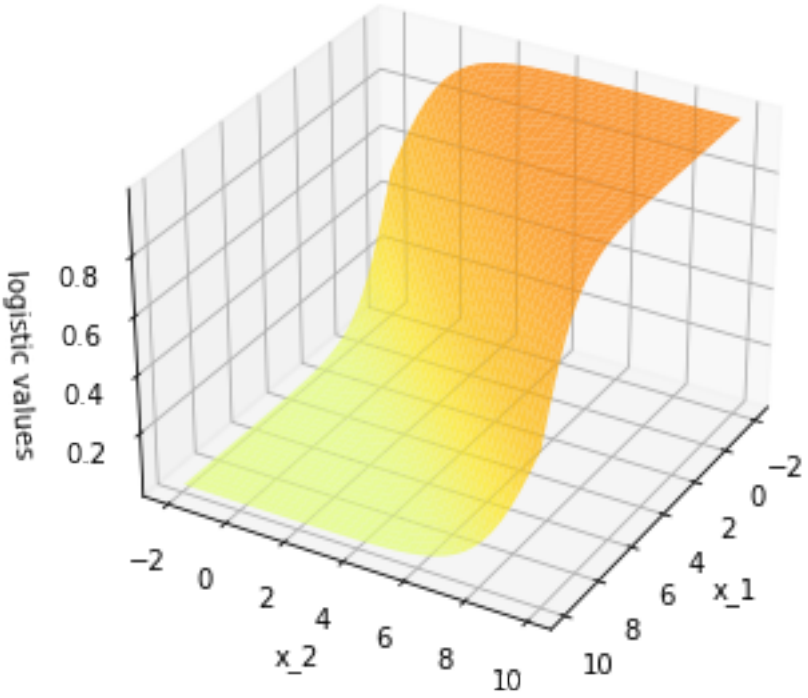
This is **binary classification**.

[20]:
```
from matplotlib import cm

fig = plt.figure(figsize=(5,5))
ax = fig.add_subplot(projection='3d')

# Define dimensions
Nx, Ny, Nz = 10, 10, 1
X, Y, Z = np.meshgrid(np.arange(-2,Nx,0.1), np.arange(-2,Ny,0.1), np.arange(Nz))

# Create sigmoid data
sigmoid = (1+np.exp(-(Y-X)))**(-1)
# plot surface
ax.plot_surface(X[:, :, 0], Y[:, :, 0], sigmoid[:, :, 0], cmap=cm.Wistia,␣
 ↪alpha=0.75)
ax.set_xlabel('x_1')
ax.set_ylabel('x_2')
ax.set_zlabel('logistic values')
ax.view_init(30, 30)
plt.show()
```



3.1.1 The Iris Data Set

We are going to illustrate the idea behind logistic regression using the Iris Data Set. See, for example, https://en.wikipedia.org/wiki/Iris_flower_data_set for the details.

This is very well known. Something that any aspiring Data Scientist ought to be aware of.

It is not without controversy though. It was used by Fisher, and Fisher was associated with **eugenics**. You can read more about that here:

https://www.nature.com/articles/s41437-020-00394-6

If this offends you then you can replace the use of the Iris data below with the penguins data we have been using.

[21]:
```
# load it in and take a look
sns.get_dataset_names()
dfi = sns.load_dataset('iris')
dfi.head()
```

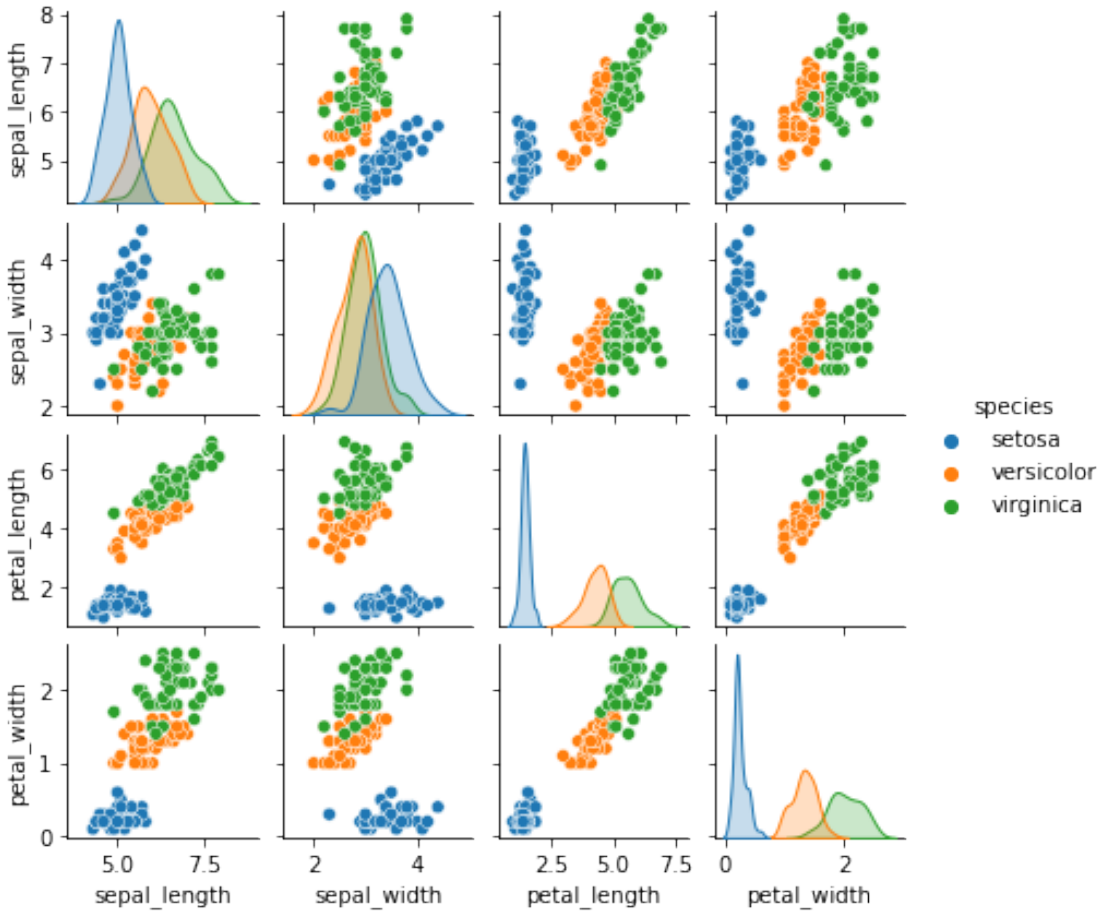
[21]:

	sepal_length	sepal_width	petal_length	petal_width	species
0	5.1	3.5	1.4	0.2	setosa
1	4.9	3.0	1.4	0.2	setosa
2	4.7	3.2	1.3	0.2	setosa
3	4.6	3.1	1.5	0.2	setosa

```
4            5.0           3.6            1.4           0.2  setosa
```

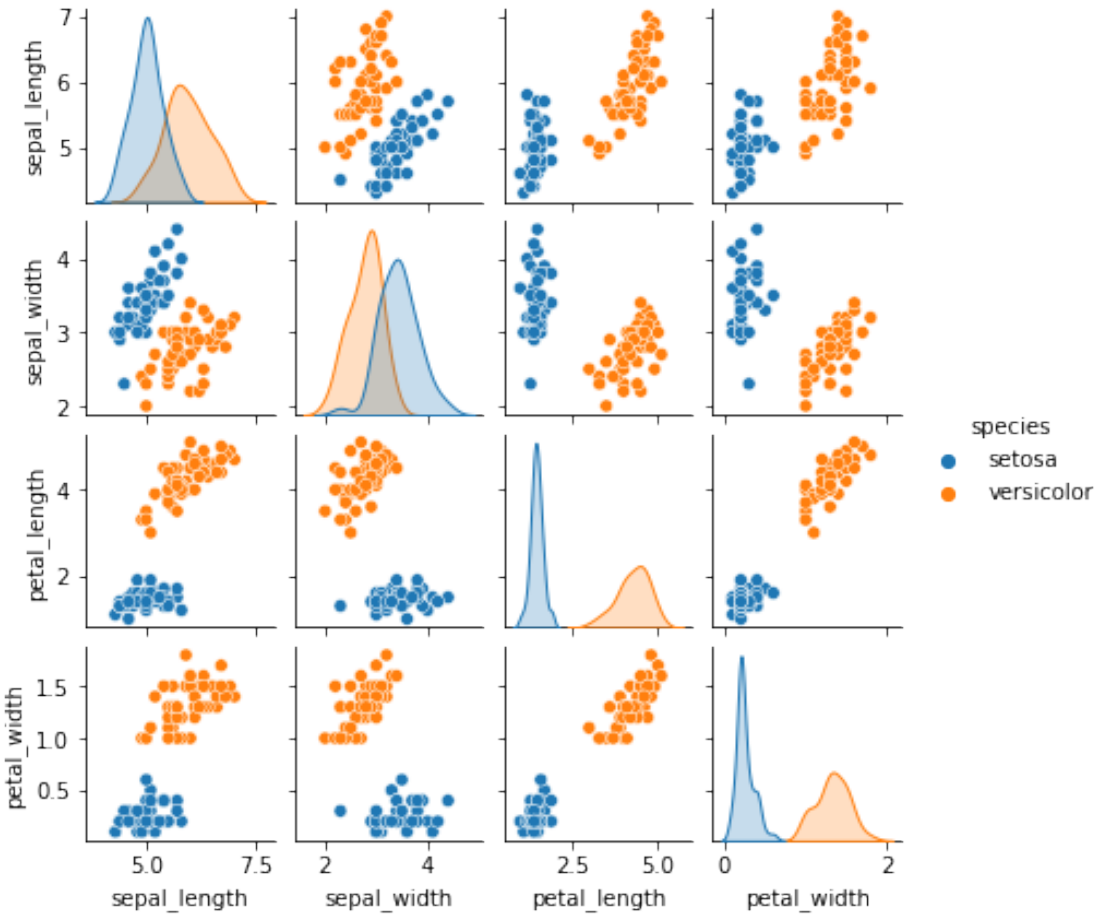
[22]:
```
sns.pairplot(dfi, hue='species', height = 1.5)
```

[22]:
```
<seaborn.axisgrid.PairGrid at 0x7ffba0d14ba8>
```



[23]:
```
# we want a binary classifier so we drop the virginica data
dfid = dfi[ (dfi['species'] != 'virginica') == True ]
sns.pairplot(dfid, hue='species', height = 1.5)
```

[23]:
```
<seaborn.axisgrid.PairGrid at 0x7ffbc032e710>
```



[24]:
```
dfid.head()
```

[24]:

	sepal_length	sepal_width	petal_length	petal_width	species
0	5.1	3.5	1.4	0.2	setosa
1	4.9	3.0	1.4	0.2	setosa
2	4.7	3.2	1.3	0.2	setosa
3	4.6	3.1	1.5	0.2	setosa
4	5.0	3.6	1.4	0.2	setosa

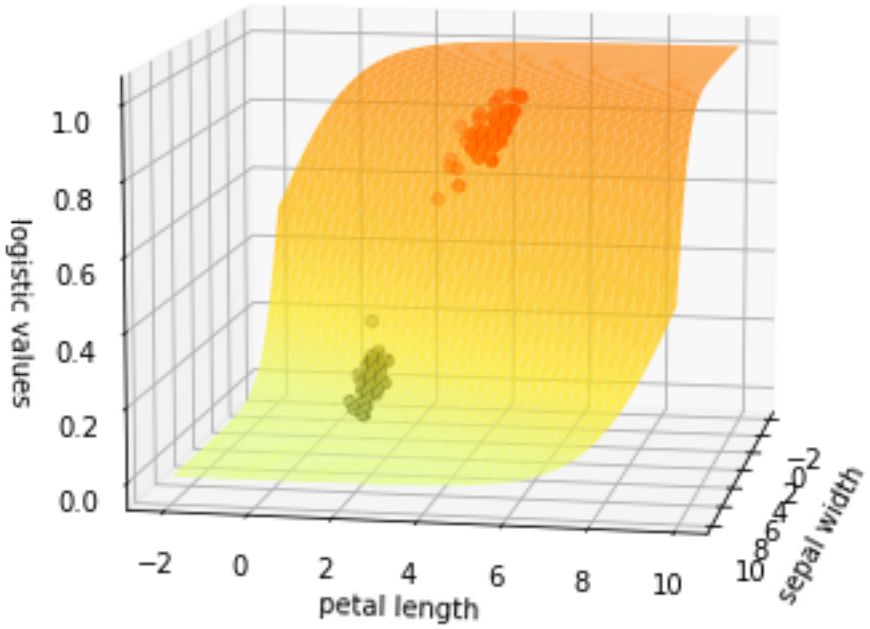
[25]:
```
# let's use petal length and sepal width as our features
xall = dfid.iloc[:,[1,2]].values
# and species as our label
yall = dfid.iloc[:, 4].values
print(xall[0:5,:], yall[0:5])
```

```
[[3.5 1.4]
 [3.  1.4]
 [3.2 1.3]
 [3.1 1.5]]
```

```
[3.6 1.4]] ['setosa' 'setosa' 'setosa' 'setosa' 'setosa']
```

[26]:
```
# select the setosa and versicolor feature rows
x_set = xall[yall == 'setosa',:]
x_ver = xall[yall == 'versicolor',:]
# set the vertical coordinate for the 3D surface plot
z_set = (1+np.exp(-(x_set[:,1]-x_set[:,0]) ))**(-1)
z_ver = (1+np.exp(-(x_ver[:,1]-x_ver[:,0]) ))**(-1)
```

[27]:
```
fig = plt.figure(figsize=(6,6)) # some of this set up was done above
ax = fig.add_subplot(projection='3d')
ax.plot_surface(X[:, :, 0], Y[:, :, 0], sigmoid[:, :, 0], cmap=cm.Wistia,␣
↪alpha=0.75)
ax.scatter(x_set[:,0], x_set[:,1], z_set, c='black', marker='o')
ax.scatter(x_ver[:,0], x_ver[:,1], z_ver, c='red', marker='o')
ax.set_xlabel('sepal width'); ax.set_ylabel('petal length')
ax.set_zlabel('logistic values'); ax.view_init(10, 10); plt.show()
```



3.1.2 Discussion

What you see here is that the data are being separated by the sigmoid, or logistic, function's **ramp** from 0 to 1.

The idea behind **logistic regression** is to use training data to determine a line $ax_1 + bx_2 + c = 0$ so that

$$\sigma(x_1, x_2 \mid a, b, c) = \frac{1}{1 + \exp\big(-(ax_1 + bx_2 + c)\big)}$$

can be used as a classifier. The line is then a **decision boundary**.

For example, for input feature values x_1 and x_2, and for a, b and c determined by the regression, we would classify as:

$$\begin{cases} \sigma(x_1, x_2 \mid a, b, c) \geq 0.5 & (x_1, x_2) \text{ indicate Class 1 (e.g. versicolor);} \\ \sigma(x_1, x_2 \mid a, b, c) < 0.5 & (x_1, x_2) \text{ indicate Class 2 (e.g. setosa);} \end{cases}$$

3.1.3 Technical Details

Suppose we use $\boldsymbol{\theta} = (\theta_0, \theta_1, \theta_2)^T$ with $\boldsymbol{x} = (1, x_1, x_2)^T$ to represent the decision boundary with $\boldsymbol{\theta}^T\boldsymbol{x} = 0$. Then, with

$$p = \frac{1}{1 + \exp(-\boldsymbol{\theta}^T\boldsymbol{x})}$$

we get

$$1 - p = 1 - \frac{1}{1 + \exp(-\boldsymbol{\theta}^T\boldsymbol{x})} = \frac{1 + \exp(-\boldsymbol{\theta}^T\boldsymbol{x})}{1 + \exp(-\boldsymbol{\theta}^T\boldsymbol{x})} - \frac{1}{1 + \exp(-\boldsymbol{\theta}^T\boldsymbol{x})} = \frac{\exp(-\boldsymbol{\theta}^T\boldsymbol{x})}{1 + \exp(-\boldsymbol{\theta}^T\boldsymbol{x})}.$$

Hence,

$$\frac{p}{1-p} = \frac{1 + \exp(-\boldsymbol{\theta}^T\boldsymbol{x})}{\exp(-\boldsymbol{\theta}^T\boldsymbol{x})} \frac{1}{1 + \exp(-\boldsymbol{\theta}^T\boldsymbol{x})} = \exp(\boldsymbol{\theta}^T\boldsymbol{x})$$

and $0 < p < 1$ - so p is a probability and $p/(p-1)$ the **odds**.

3.1.4 The log-odds

Taking the log of both sides

$$\ln\left(\frac{p}{1-p}\right) = \ln\exp(\boldsymbol{\theta}^T\boldsymbol{x})$$

Therefore, simplifying,

$$\ln\left(\frac{p}{1-p}\right) = \boldsymbol{\theta}^T \boldsymbol{x}$$

and there is the connection to linear regression. The decision boundary models the log-odds.

3.1.5 How Do We Do Logistic Regression?

Suppose we have a training set of features $\boldsymbol{x}_1$, $\boldsymbol{x}_2$, $\boldsymbol{x}_3$, ..., with corresponding labels y_1, y_2, y_3, ... We suppose that the labels are either $y = 0$ or $y = 1$ to represent the **binary classification**.

Then, for a given pair $(\boldsymbol{x}_k, y_k)$ with $\boldsymbol{x}_k = (x_{1,k}, x_{2,k})^T$, we can set up the **squared loss**,

$$\left|y_k - \sigma(x_{1,k}, x_{2,k} \mid a, b, c)\right|^2$$

and use it to form a TSE or MSE cost over the training set. We then choose a, b and c to minimise this cost.

There is no closed-form solution to this (no equivalent of the Normal Equations), but we can use numerical techniques.

It can (of course) be done in `sklearn`. The details are here:

https://scikit-learn.org/stable/modules/generated/sklearn.linear_model.LogisticRegression.html

along with an actual example on the Iris data set.

3.1.6 Closing Remarks

We just covered a great deal. We finish with a couple of remarks.

The $p > n$ Issue

We discussed the fact that $\boldsymbol{X}^T\boldsymbol{X}$ is invertible if $\boldsymbol{X}$ has full column rank

For a discussion and proof see Page 149 of *Linear Algebra for Everyone*, by Gilbert Strang, Wellesley Cambridge Press, 2020.

This doesn't mean that we can't minimize our cost $\mathcal{E}(\boldsymbol{\theta}) = \|\boldsymbol{y} - \boldsymbol{X}\boldsymbol{\theta}\|_2^2$ though.

But it may mean that the minimizer is not unique.

The situation is very technical - and beyond our scope.

[IPDS, Chapter 7.1.4] has an excellent discussion of this and you are referred there if you want to see the mathematical details.

3.1.7 Bias-Variance Decomposition and Trade-Off

Consider a model that produces a random variable z as an estimate of an unknown z_0. We would like $\bar{z}$ - the mean of z - to be z_0.

- We call $\bar{z} - z_0$ the **bias** in the model.

Further, the random variable will have a variance,

- $\text{Var}(z) = \mathbb{E}\Big((z - \bar{z})^2\Big)$

We have been using the squared error as a cost function. What is its expected value?

$$\begin{aligned}\mathbb{E}\Big((z - z_0)^2\Big) &= \mathbb{E}\Big(\big((z - \bar{z}) + (\bar{z} - z_0)\big)^2\Big), \\ &= \mathbb{E}\Big((z - \bar{z})^2 + 2(z - \bar{z})(\bar{z} - z_0) + (\bar{z} - z_0)^2\Big).\end{aligned}$$

Using **linearity of expectation**

$$\begin{aligned}\mathbb{E}\Big((z - z_0)^2\Big) &= \mathbb{E}\Big((z - \bar{z})^2\Big) + 2(\bar{z} - z_0)\mathbb{E}(z - \bar{z}) + \mathbb{E}\Big((\bar{z} - z_0)^2\Big), \\ &= \underbrace{\mathbb{E}\Big((z - \bar{z})^2\Big)}_{\text{variance}} + 2(\bar{z} - z_0)\underbrace{\mathbb{E}(z - \bar{z})}_{\text{zero}} + \underbrace{(\bar{z} - z_0)^2}_{\text{bias}^2}.\end{aligned}$$

Hence $\mathbb{E}\Big((z - z_0)^2\Big) = (\bar{z} - z_0)^2 + \mathbb{E}\Big((z - \bar{z})^2\Big) = \text{bias}^2 + \text{variance}$.

What this means is that for small MSE cost we need to control both the bias and the variance.

Typically, driving down the bias will mean increasing the variance and *vice versa*: there is a **trade off**.

See [IPDS, Theorem 7.5] and, in particular, [MLFCES, Chapter 4.4] for much higher quality discussions.

3.1.8 Review

We covered *just enough*, to make *progress at pace*. We looked at

- Polynomial Regression using a least squares cost function.
- Ordinary Least Squares as well as regularized.
- Multivariable Regression.
- Logistic Regression.

Now we can start putting all of this material to work.

3.1.9 Homework

In the next session we will briefly discuss **Support Vector Machines** (SVM's) and quickly move to to discuss the **perceptron**.

Our SVM treatment will be very shallow - to prepare yourself for this your homework is to read https://www.syncfusion.com/succinctly-free-ebooks/support-vector-machines-succinctly up page 43 prior to the next session. This isn't as much as it seems: pages 1-20 are a revision of the vector material we have already covered.

3.2 Technical Notes, Production and Archiving

Ignore the material below. What follows is not relevant to the material being taught.

Production Workflow

- Finalise the notebook material above
- Clear and fresh run of entire notebook
- Create html slide show:
  - `jupyter nbconvert --to slides 11_regress.ipynb`
- Set `OUTPUTTING=1` below
- Comment out the display of web-sourced diagrams
- Clear and fresh run of entire notebook
- Comment back in the display of web-sourced diagrams
- Clear all cell output
- Set `OUTPUTTING=0` below
- Save
- git add, commit and push to FML
- copy PDF, HTML etc to web site
  - git add, commit and push
- rebuild binder

Some of this originated from

https://stackoverflow.com/questions/38540326/save-html-of-a-jupyter-notebook-from-within-the-n

These lines create a back up of the notebook. They can be ignored.

At some point this is better as a bash script outside of the notebook

[28]:
```
%%bash
NBROOTNAME=11_regress
OUTPUTTING=1

if [ $OUTPUTTING -eq 1 ]; then
  jupyter nbconvert --to html $NBROOTNAME.ipynb
  cp $NBROOTNAME.html ../backups/$(date +"%m_%d_%Y-%H%M%S")_$NBROOTNAME.html
  mv -f $NBROOTNAME.html ./formats/html/

  jupyter nbconvert --to pdf $NBROOTNAME.ipynb
  cp $NBROOTNAME.pdf ../backups/$(date +"%m_%d_%Y-%H%M%S")_$NBROOTNAME.pdf
  mv -f $NBROOTNAME.pdf ./formats/pdf/

  jupyter nbconvert --to script $NBROOTNAME.ipynb
  cp $NBROOTNAME.py ../backups/$(date +"%m_%d_%Y-%H%M%S")_$NBROOTNAME.py
  mv -f $NBROOTNAME.py ./formats/py/
else
  echo 'Not Generating html, pdf and py output versions'
fi
```

```
Not Generating html, pdf and py output versions
```