Lecture 6 - Curse of Dimensionality (continued) & Regularization

The Curse of Dimensionality

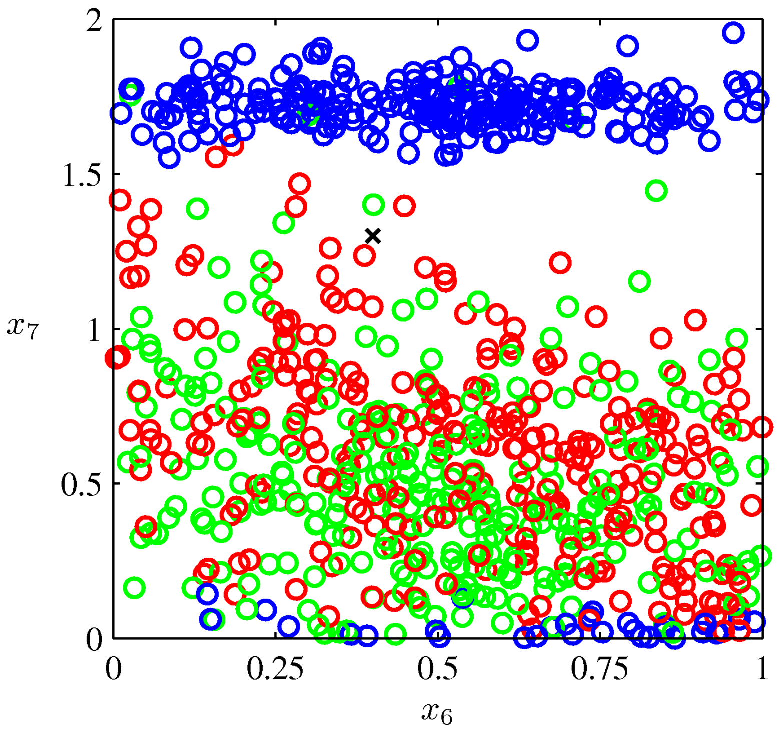
- The Curse of Dimensionality illustrates various phenomena that arise when we work with high-dimensional data spaces that would not otherwise occur in lower-dimensional settings (such as the 3-dimensional space).

Let's consider the following data set representing measurements taken from a pipeline containing a mixture of oil, water, and gas (Bishop, 2006).

In [1]:

```
from IPython.display import Image
Image('figures/Figure1.19.png', width=400)
```

Out[1]:



Each data point comprises a 12-dimensional input vector consisting of measurements taken with gamma ray densitometers that measure the attenuation of gamma rays passing along narrow beams through the pipe.

- We observe that the cross is surrounded by numerous red points, and so we might suppose that it belongs to the red class.

- However, there are also plenty of green points nearby, so we might think that it could instead belong to the green class. It seems unlikely that it belongs to the blue class.

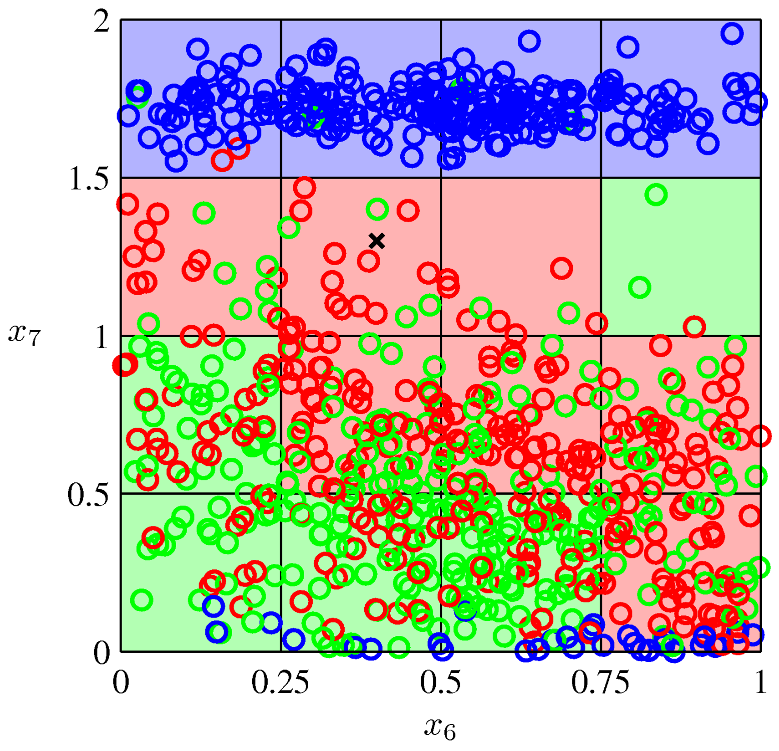
- The intuition here is that the identity of the cross should be determined more strongly by nearby points from the training set and less strongly by more distant points.

How can we turn this intuition into a learning algorithm?

- One very simple approach would be to divide the input space into regular cells, as indicated in the figure below. When we are given a test point and we wish to predict its class, we first decide which cell it belongs to, and we then find all of the training data points that fall in the same cell.
- The identity of the test point is predicted as being the same as the class having the largest number of training points in the same cell as the test point (with ties being broken at random).

In [2]:
```
Image('figures/Figure1.20.png', width=400)
```

Out[2]:



What is the biggest problem with this approach?

- If we divide a region of a space into regular cells, then the number of such cells grows exponentially with the dimensionality of the space.
- The problem with an exponentially large number of cells is that we would need an exponentially large quantity of training data in order to ensure that the cells are not empty.
- Clearly, we have no hope of applying such a technique in a space of more than a few variables, and so we need to find a more sophisticated approach.

We can gain further insight into the problems of high-dimensional spaces by returning to the example of polynomial curve fitting and considering how we would extend this approach to deal with input spaces having several variables. If we have D-dimensional input variables, then a general polynomial with coefficients up to order 3 would take the form:

$$y(\mathbf{x}, \mathbf{w}) = w_0 + \sum_{i=1}^{D} w_i x_i + \sum_{i=1}^{D} \sum_{j=1}^{D} w_{ij} x_i x_j + \sum_{i=1}^{D} \sum_{j=1}^{D} \sum_{k=1}^{D} w_{ijk} x_i x_j x_k$$

- As D increases, so the number of independent coefficients grows proportionally to D^3.
- In practice, to capture complex dependencies in the data, we may need to use a higher-order polynomial. For a polynomial of order M, the growth in the number of coefficients is D^M.
- Although this is now a power law growth, rather than an exponential growth, it still points to the method becoming rapidly unwieldy and of limited practical utility.

> ## Intuitions do not hold in Higher Dimensional Spaces

Our geometrical intuitions, formed through a life spent in a 3-dimensional space, will not scale to spaces of much higher dimensionality.

Let's illustrate this with two examples.

Example 1: Volume of a Crust

Let's take the example of the **volume of a crust**.

- Consider two spheres. One sphere S_1 with radius r, and another sphere S_2 with radious $r - \epsilon$.
- The ratio of the crust to the S_1 sphere:

$$ratio = \frac{V_{crust}}{V_{S_1}} = \frac{V_{S_1} - V_{S_2}}{V_{S_1}}$$

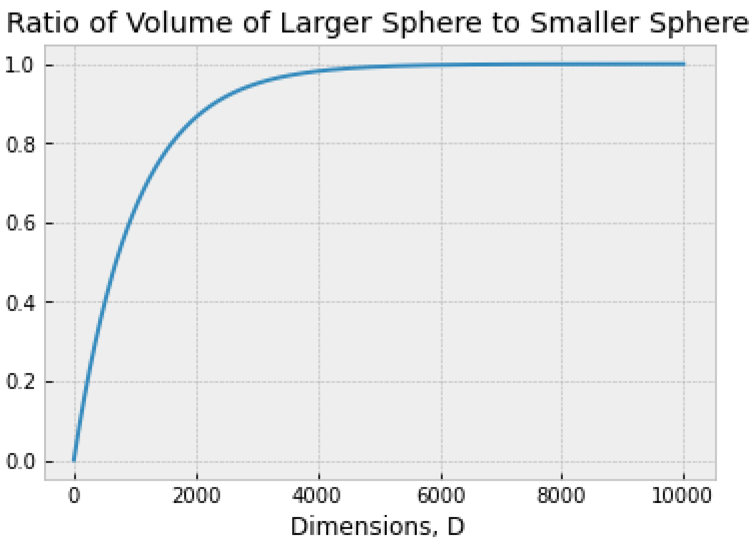
- The D-dimensional volume of a ball of radius r in D-dimensional space is: $V = \frac{r^D \pi^{\frac{D}{2}}}{\Gamma\left(\frac{D}{2}+1\right)}$, then

$$\begin{aligned}
ratio &= \frac{V_{S_1} - V_{S_2}}{V_{S_1}} && (1) \\
&= 1 - \frac{V_{S_2}}{V_{S_1}} && (2) \\
&= 1 - \frac{\frac{(r-\epsilon)^D \pi^{\frac{D}{2}}}{\Gamma\left(\frac{D}{2}+1\right)}}{\frac{r^D \pi^{\frac{D}{2}}}{\Gamma\left(\frac{D}{2}+1\right)}} && (3) \\
&= 1 - \frac{(r-\epsilon)^D}{r^D} && (4) \\
&= 1 - \frac{r^D\left(1 - \frac{\epsilon}{r}\right)^D}{r^D} && (5) \\
&= 1 - \left(1 - \frac{\epsilon}{r}\right)^D && (6)
\end{aligned}$$

- For a fixed value for ϵ, a fixed radius r and $\epsilon << r$, what happens as D increases?

In [3]:
```
import numpy as np
import numpy.random as npr
import matplotlib.pyplot as plt
%matplotlib inline
plt.style.use('bmh')
```

In [4]:
```
# Crust volume between spheres with epsilon different radii and increasing dimensionali

r = 1
eps = 0.001
D = range(1,10000)
RatioVol = [1-(1-eps/r)**d for d in D]
plt.plot(D, RatioVol)
plt.title('Ratio of Volume of Larger Sphere to Smaller Sphere');
plt.xlabel('Dimensions, D');
```



Example 2: Unit Porcupine

Consider the unit porcupine with is represented as the unit hypersphere inscribed within the unit hypercube.

Recall that the D-dimensional volume of a D-dimensional cube with radius r is $(2r)^D$.

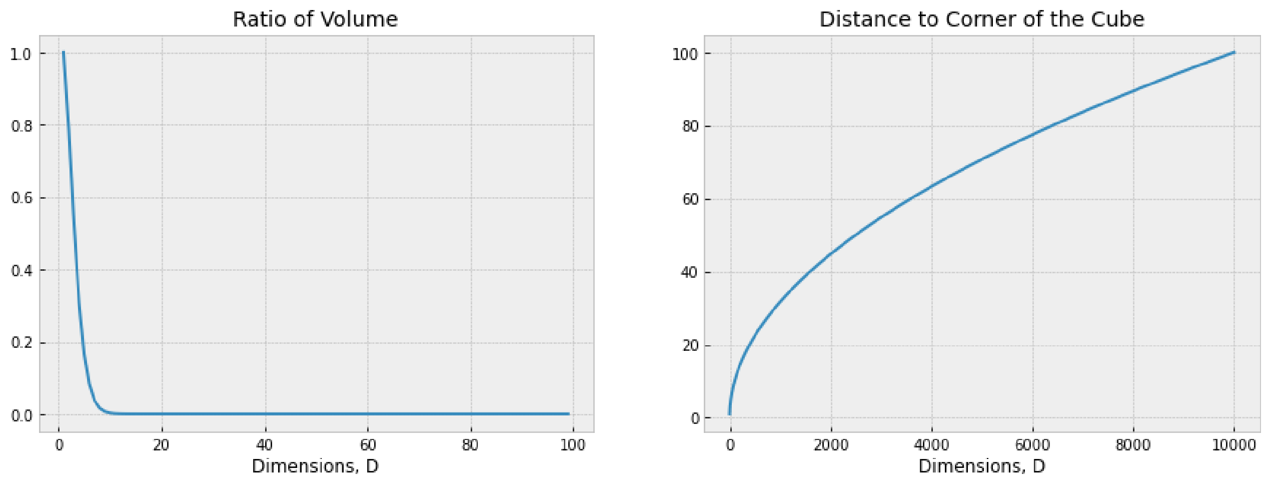
- Take the case where the hypersphere is inscribed within the unit hypercube. What happens to the relative volume of the sphere and cube as D increases?

$$\frac{V(sphere)}{V(cube)} = \frac{\frac{r^D \pi^{\frac{D}{2}}}{\Gamma\left(\frac{D}{2}+1\right)}}{(2r)^D} \tag{7}$$

$$= \frac{r^D \pi^{\frac{D}{2}}}{(2r)^D \Gamma\left(\frac{D}{2}+1\right)} \tag{8}$$

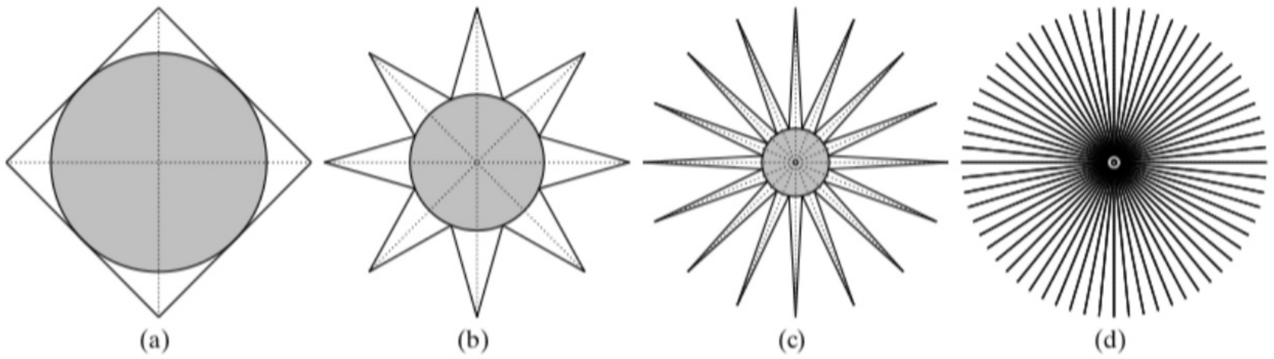
$$= \frac{\pi^{\frac{D}{2}}}{2^D \Gamma\left(\frac{D}{2}+1\right)} \tag{9}$$

In [5]:
```
# The Unit Porcupine (the unit hyper-sphere inscribed within the unit hyper-cube)

import math

D = range(1,100)
V = [np.pi**(i/2)/(2**i*math.gamma(i/2 + 1)) for i in D]

fig = plt.figure(figsize=(15,5))
ax = fig.add_subplot(1,2,1)
ax.plot(D, V)
ax.set_title('Ratio of Volume')
plt.xlabel('Dimensions, D')

dist_to_Corner = [math.sqrt(d) for d in range(1,10000)]
ax = fig.add_subplot(1,2,2)
ax.plot(range(1,10000), dist_to_Corner)
ax.set_title('Distance to Corner of the Cube')
plt.xlabel('Dimensions, D');
```



In [6]:
```
Image('figures/Unit Porcupine.jpg',width=800)
```

Out[6]:



In my mind, this is what the Curse of Dimensionality sounds like:

In [7]:
```python
from IPython.display import Audio
Audio("EvilLaugh.mp4", autoplay=False)
```

Out[7]:

0:00 / 0:08

Curse of Dimensionality Takeaway

- All of the volume is in the corners
- We need exponentially more data as we go to higher and higher dimensions.
- We need to be careful choosing a model as that choice "injects" what we want the data to look like or follow a specific behavior.
- Always employ the **Occam's Razor** principle: the simplest model that works for our data is usually the most appropriate and sufficient. Model simplicity can mean different things, but we can consider model order.
- When we are in a high-dimensional input space (such as images), much of that space is empty. The input data can be represented in only a few *degrees of freedom* of variability. We say that the data in *embedded* in a **manifold** of equal dimensionaly as the degrees of freedom (which is drastically smaller than the input space dimensionality).
- And, again, intuition or assumptions don't always hold in higher dimensions.
    - Try to think of the problem: When we draw from a univariate Gaussian, where are the data samples most likely coming from? Where are data samples most likely coming from if we draw them from a high-dimensional multivariate Gaussian?

The No Free Lunch Theorem

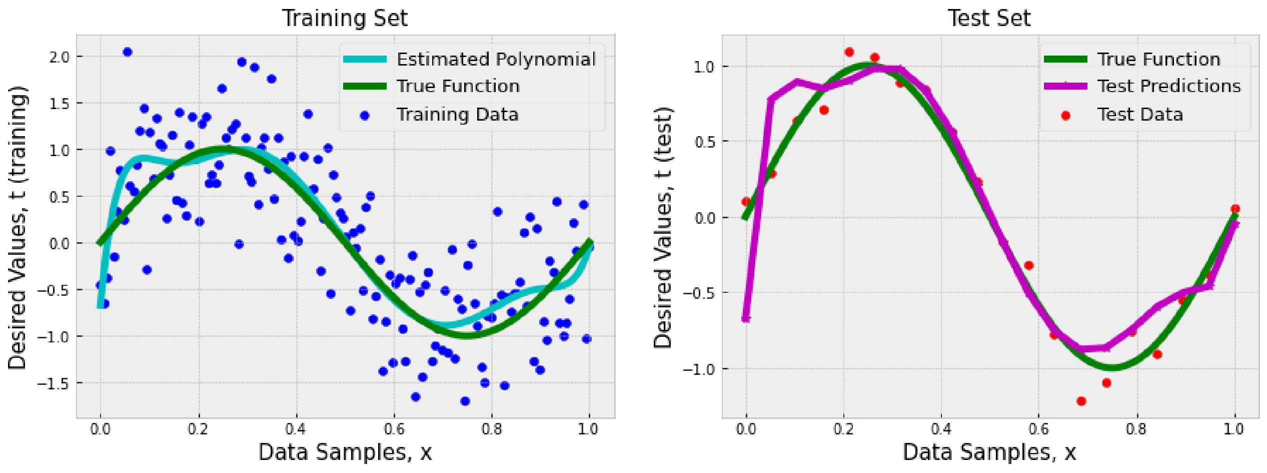
In [8]:
```python
def NoisySinusoidalData(N, a, b, gVar):
    x = np.linspace(a,b,N)
    noise = npr.normal(0,gVar,N)
    t = np.sin(2*np.pi*x) + noise
    return x, t

def PolynomialRegression(x,t,M):
    X = np.array([x**m for m in range(M)]).T
    w = np.linalg.inv(X.T@X)@X.T@t
    y = X@w
    error = t-y
    return w, y, error

def PolynomialRegression_test(x,M,w):
    X = np.array([x**m for m in range(M)]).T
    y = X@w
    return y
```

In [21]:

```
# Generate input samples and desired values

Ntrain = 150
Ntest = 20
a, b = [0,1]
sigma_train = 0.5
sigma_test = 0.1
x_train, t_train = NoisySinusoidalData(Ntrain, a, b, sigma_train)
x_true, t_true = NoisySinusoidalData(Ntrain, a, b, 0)
x_test, t_test = NoisySinusoidalData(Ntest, a, b, sigma_test)
```

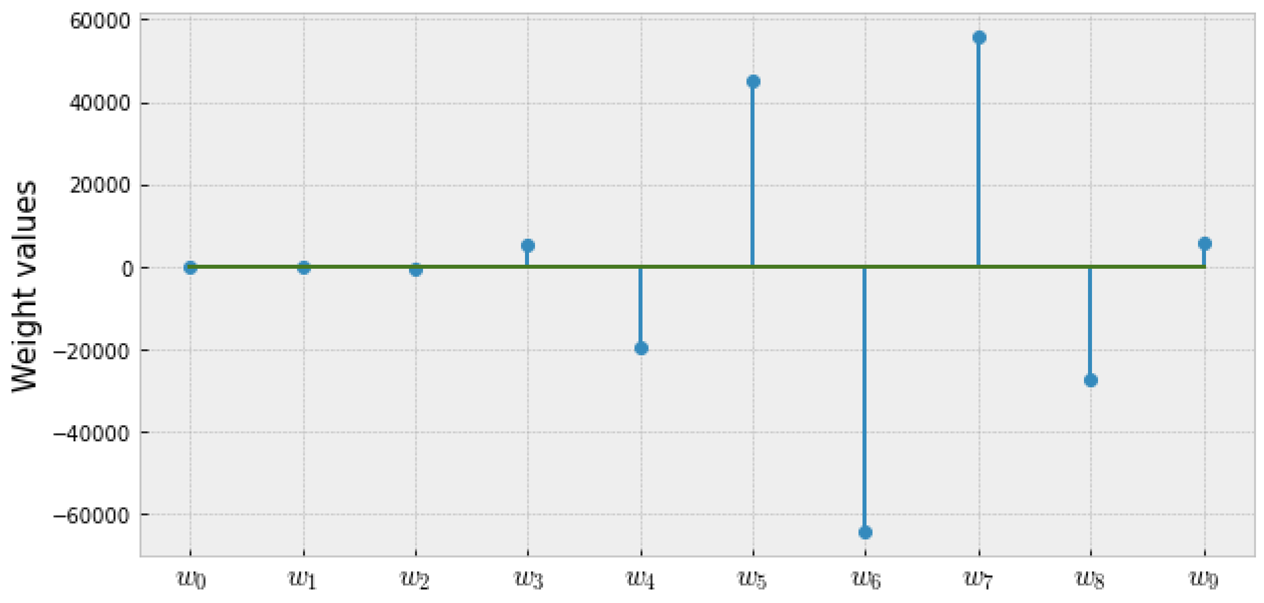
In [22]:

```
M = 10

w, y_train, error = PolynomialRegression(x_train,t_train,M)
y_test = PolynomialRegression_test(x_test, M, w)

fig = plt.figure(figsize=(15,5))
fig.add_subplot(1,2,1)
plt.plot(x_train,y_train,'c', label = 'Estimated Polynomial', linewidth=5)
plt.plot(x_true,t_true,'g', label = 'True Function', linewidth=5)
plt.scatter(x_train,t_train,c='b', label='Training Data')
plt.legend(fontsize=13); plt.title('Training Set',size=15)
plt.xlabel('Data Samples, x', size=15)
plt.ylabel('Desired Values, t (training)', size=15);

fig.add_subplot(1,2,2)
plt.plot(x_true,t_true,'g', label = 'True Function', linewidth=5)
plt.scatter(x_test,t_test, c='r', label='Test Data')
plt.plot(x_test,y_test,'-m*', label = 'Test Predictions', linewidth=5)
plt.legend(fontsize=13); plt.title('Test Set',size=15)
plt.xlabel('Data Samples, x', fontsize=15)
plt.ylabel('Desired Values, t (test)', fontsize=15);
```



In [23]:

```python
plt.figure(figsize=(10,5))
plt.stem(w)
plt.ylabel('Weight values', size=15)
plt.xticks(np.arange(len(w)), ['$w_{'+str(i)+'}$' for i in range(len(w))],rotation=0, s
```



Motivation Question: How do you select which model setting should you use?

- A model setting could simply include different parameter settings, such as different values for M using the same model and features.
- But in general, a "model setting" refers to the collection of choices for all steps in the training stage.

Experimental Design - How to use your data without cheating In experimental design we need data to train (learn) models, and to test how good the models are. The training data needs to be different (disjoint) from the test data. Otherwise we would be testing the learned model on data it had previously seen, and we would get a biased estimate of the model's generalized performance. Most machine learning algorithms require choosing parameter values; very often this is done by setting aside some of the training data to evaluate the quality of different parameter settings.

Typically we split the **training data** into three disjoint sets:

- **Training set**, 80\% - set of samples (and its labels) used to estimate the parameter values of the model (*learning the model*)
- **Validation set** - set of samples (and its labels) used for exploring and picking best parameter values
- **Test set**, typically 20\% - set of samples (and its labels) used for testing the model generalization performance, and testing hypotheses

1. The key thing to remember when planning experiments is that the test data is used to form conclusions but not to make decisions during model building. Basing decisions on test data results is frequently called *cheating* in the machine learning community, and often results in wrong conclusions.

2. Our generalization performance is only as good as our test set is representative of the true test data in application.

3. After all parameter value decisions have been made, we often use **ALL** training data for the final training of the system and deployment.

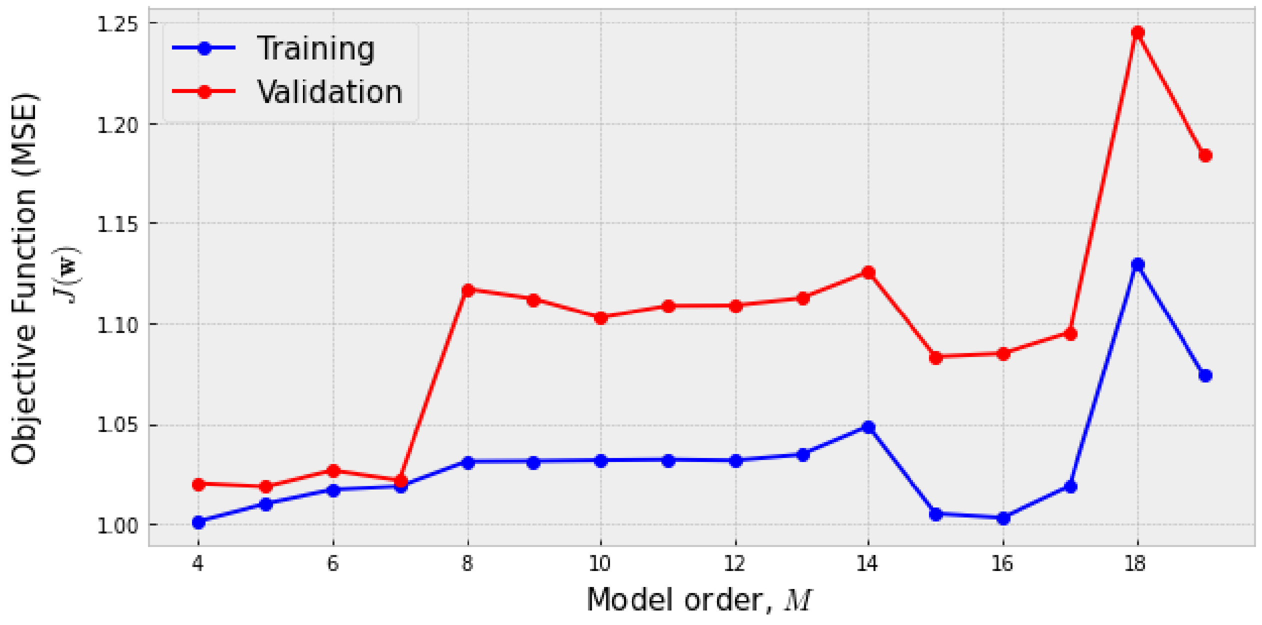
4. The training and validation sets may be rotated by using **cross-validation**.

Consider the synthetic example from earlier. Let's split it into training and validation sets:

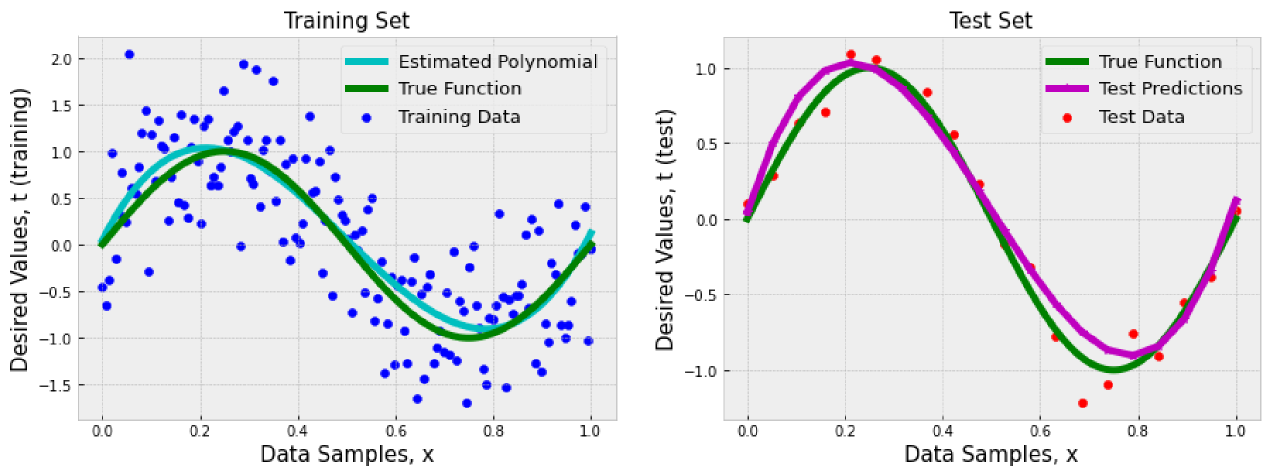
In [24]:
```
from sklearn.model_selection import train_test_split

train_test_split?
```

We can use the validation set to *check* the performance of polynomial regression model for different model orders:

In [41]:
```
train_data, validation_data, train_labels, validation_labels = train_test_split(x_train

J_train=[]
J_val=[]
Mvalues=range(4,20)

for M in Mvalues:

    w, y_train, error = PolynomialRegression(train_data, train_labels, M)
    y_validation = PolynomialRegression_test(validation_data, M,w)
    J_train+=[np.mean((train_data-y_train)**2)]
    J_val+=[np.mean((validation_data-y_validation)**2)]

plt.figure(figsize=(10,5))
plt.plot(Mvalues, J_train, '-ob',label='Training')
plt.plot(Mvalues, J_val, '-or', label='Validation')
plt.legend(fontsize=15); plt.xlabel('Model order, $M$',size=15)
plt.ylabel('Objective Function (MSE)\n $J(\mathbf{w})$',size=15);
```



In [43]:
```
M = 4

w, y_train, error = PolynomialRegression(x_train,t_train,M)
y_test = PolynomialRegression_test(x_test, M, w)

fig = plt.figure(figsize=(15,5))
fig.add_subplot(1,2,1)
plt.plot(x_train,y_train,'c', label = 'Estimated Polynomial', linewidth=5)
plt.plot(x_true,t_true,'g', label = 'True Function', linewidth=5)
plt.scatter(x_train,t_train,c='b', label='Training Data')
plt.legend(fontsize=13); plt.title('Training Set',size=15)
plt.xlabel('Data Samples, x', size=15)
plt.ylabel('Desired Values, t (training)', size=15);

fig.add_subplot(1,2,2)
plt.plot(x_true,t_true,'g', label = 'True Function', linewidth=5)
plt.scatter(x_test,t_test, c='r', label='Test Data')
plt.plot(x_test,y_test,'-m*', label = 'Test Predictions', linewidth=5)
plt.legend(fontsize=13); plt.title('Test Set',size=15)
plt.xlabel('Data Samples, x', fontsize=15)
plt.ylabel('Desired Values, t (test)', fontsize=15);
```



What happens if we sample a new training set?

We cannot rely on one training run of the algorithm:

- Variations in training/validation sets
- random factors during training (e.g., random initialization, local optima, etc.)

No Free Lunch Theorem The No Free Lunch Theorem states that there is no single learning algorithm that in any domain always induces the most accurate learner. The usual approach is to try many and choose the one that performs the best on a separate validation set. For any learning algorithm, there is a dataset where it is very accurate and another dataset where it is very poor. When we say that a learning algorithm is good, we only quantify how well its inductive bias matches the properties of the data.

Performance of an algorithm can be determined using a variety of statistical goodness-of-fit measures. Some examples are: error rate, accuracy, ROC curves, performance-recall curves, etc.. But it can also be in terms of:

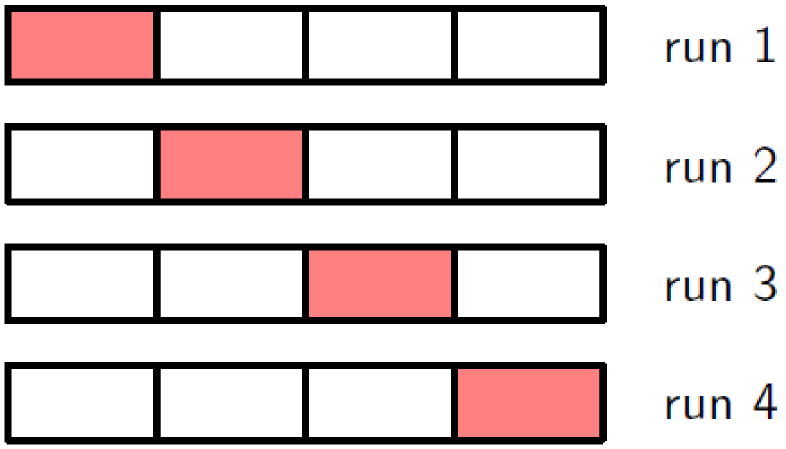
- Risk,
- Running time,
- Training time and storage/memory,
- Testing time and storage/memory,
- Interpretability, namely, whether the method allows knowledge extraction which can be checked and validated by experts, and
- computational complexity.

k-fold Cross-Validation

k-fold Cross-Validation The goal of **cross-validation** is to test the model's ability to predict new data that was not used in estimating the model, in order to flag problems like *overfitting* or *selection bias* and to give an insight on how the model will generalize to an independent dataset (i.e., an unknown dataset). The technique of **k-fold cross-validation**, illustrated below for the case of $k = 4$, involves taking the available data and partitioning it into k groups (in the simplest case these are of equal size). Then $k - 1$ of the groups are used to train a set of models that are then evaluated on the remaining group. This procedure is then repeated for all k possible choices for the held-out group, indicated in the picture below by the red blocks, and the performance scores from the S runs are then averaged.

In [29]:
```
Image('figures/Kfold CV.png',width=400)
```

Out[29]:


run 1
run 2
run 3
run 4