Artificial Intelligence and Machine Learning

Unit II

Linear Regression

My own latex definitions

In [1]:

```python
import matplotlib
import matplotlib.pyplot as plt
import numpy as np
%matplotlib inline
#plt.style.use('seaborn-whitegrid')

font = {'family' : 'Times',
        'weight' : 'bold',
        'size'   : 12}

matplotlib.rc('font', **font)


# Aux functions

def plot_grid(Xs, Ys, axs=None):
    ''' Aux function to plot a grid'''
    t = np.arange(Xs.size) # define progression of int for indexing colormap
    if axs:
        axs.plot(0, 0, marker='*', color='r', linestyle='none') #plot origin
        axs.scatter(Xs,Ys, c=t, cmap='jet', marker='.') # scatter x vs y
        axs.axis('scaled') # axis scaled
    else:
        plt.plot(0, 0, marker='*', color='r', linestyle='none') #plot origin
        plt.scatter(Xs,Ys, c=t, cmap='jet', marker='.') # scatter x vs y
        plt.axis('scaled') # axis scaled

def linear_map(A, Xs, Ys):
    '''Map src points with A'''
    # [NxN,NxN] -> NxNx2 # add 3-rd axis, like adding another layer
    src = np.stack((Xs,Ys), axis=Xs.ndim)
    # flatten first two dimension
    # (NN)x2
    src_r = src.reshape(-1,src.shape[-1]) #ask reshape to keep last dimension and adjust the rest
    # 2x2 @ 2x(NN)
    dst = A @ src_r.T # 2xNN
    #(NN)x2 and then reshape as NxNx2
    dst = (dst.T).reshape(src.shape)
    # Access X and Y
    return dst[...,0], dst[...,1]


def plot_points(ax, Xs, Ys, col='red', unit=None, linestyle='solid'):
    '''Plots points'''
    ax.set_aspect('equal')
    ax.grid(True, which='both')
    ax.axhline(y=0, color='gray', linestyle="--")
    ax.axvline(x=0, color='gray',  linestyle="--")
    ax.plot(Xs, Ys, color=col)
    if unit is None:
        plotVectors(ax, [[0,1],[1,0]], ['gray']*2, alpha=1, linestyle=linestyle)
    else:
        plotVectors(ax, unit, [col]*2, alpha=1, linestyle=linestyle)

def plotVectors(ax, vecs, cols, alpha=1, linestyle='solid'):
    '''Plot set of vectors.'''
    for i in range(len(vecs)):
        x = np.concatenate([[0,0], vecs[i]])
        ax.quiver([x[0]],
                   [x[1]],
                   [x[2]],
                   [x[3]],
                   angles='xy', scale_units='xy', scale=1, color=cols[i],
                   alpha=alpha, linestyle=linestyle, linewidth=2)
```

In [2]:

```python
import matplotlib
import matplotlib.pyplot as plt
import numpy as np
%matplotlib inline
# plt.style.use('seaborn-whitegrid')

font = {'family' : 'Times',
        'weight' : 'bold',
        'size'   : 12}

matplotlib.rc('font', **font)


# Aux functions

def plot_grid(Xs, Ys, axs=None):
    ''' Aux function to plot a grid'''
    t = np.arange(Xs.size) # define progression of int for indexing colormap
    if axs:
        axs.plot(0, 0, marker='*', color='r', linestyle='none') #plot origin
        axs.scatter(Xs,Ys, c=t, cmap='jet', marker='.') # scatter x vs y
        axs.axis('scaled') # axis scaled
    else:
        plt.plot(0, 0, marker='*', color='r', linestyle='none') #plot origin
        plt.scatter(Xs,Ys, c=t, cmap='jet', marker='.') # scatter x vs y
        plt.axis('scaled') # axis scaled

def linear_map(A, Xs, Ys):
    '''Map src points with A'''
    # [NxN,NxN] -> NxNx2 # add 3-rd axis, like adding another layer
    src = np.stack((Xs,Ys), axis=Xs.ndim)
    # flatten first two dimension
    # (NN)x2
    src_r = src.reshape(-1,src.shape[-1]) #ask reshape to keep last dimension and adjust the rest
    # 2x2 @ 2x(NN)
    dst = A @ src_r.T # 2xNN
    #(NN)x2 and then reshape as NxNx2
    dst = (dst.T).reshape(src.shape)
    # Access X and Y
    return dst[...,0], dst[...,1]


def plot_points(ax, Xs, Ys, col='red', unit=None, linestyle='solid'):
    '''Plots points'''
    ax.set_aspect('equal')
    ax.grid(True, which='both')
    ax.axhline(y=0, color='gray', linestyle="--")
    ax.axvline(x=0, color='gray',  linestyle="--")
    ax.plot(Xs, Ys, color=col)
    if unit is None:
        plotVectors(ax, [[0,1],[1,0]], ['gray']*2, alpha=1, linestyle=linestyle)
    else:
        plotVectors(ax, unit, [col]*2, alpha=1, linestyle=linestyle)

def plotVectors(ax, vecs, cols, alpha=1, linestyle='solid'):
    '''Plot set of vectors.'''
    for i in range(len(vecs)):
        x = np.concatenate([[0,0], vecs[i]])
        ax.quiver([x[0]],
                   [x[1]],
                   [x[2]],
                   [x[3]],
                  angles='xy', scale_units='xy', scale=1, color=cols[i],
                  alpha=alpha, linestyle=linestyle, linewidth=2)
```

Recap previous lecture

- Model Selection and Assessment
- Cross-validation
- Evaluation Metrics

Today's lecture

We go back to your loved 🥰Linear Algebra

Supervised, Parametric Models

1) Ordinary Linear Regression with Least Squares

2) Probabilistic Interpretation

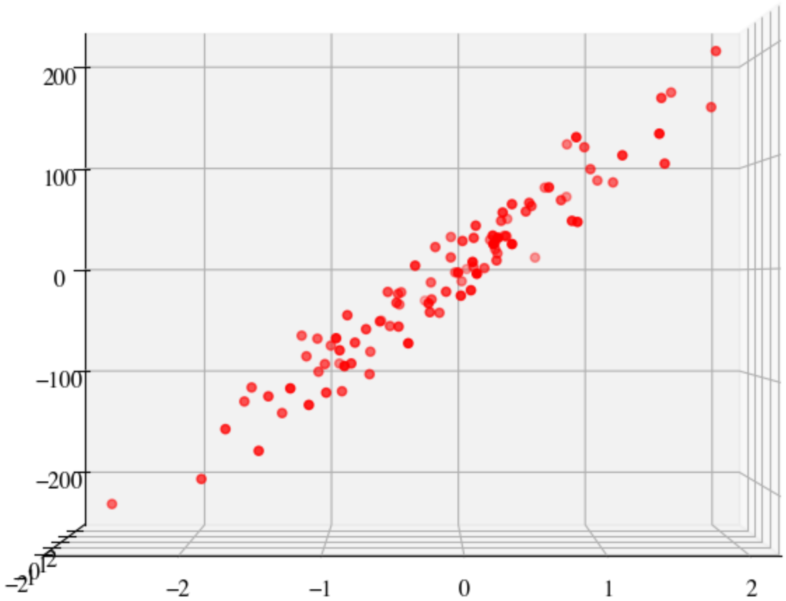
3) Gradient Descent "Family"

This lecture material is taken from

- Mostly from Stanford class
- Stanford notes
- Tibshirani - Chapter 4 page 43
- Sklearn model selection
- Bishop - Chapter 3 page 137

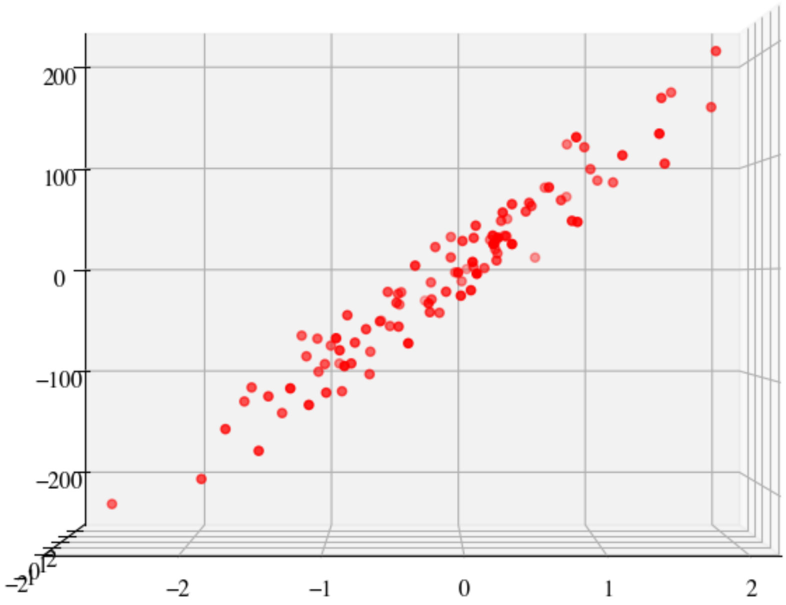
In [3]:

```
import numpy as np
from matplotlib import pyplot as plt

from sklearn import linear_model, datasets


n_samples = 100
size = 10

X, y, coef_gt = datasets.make_regression(
    n_samples=n_samples,
    n_features=2,
    n_informative=1,
    noise=20,
    coef=True,
    random_state=42,
)
fig = plt.figure(figsize=(size, size))
ax = fig.add_subplot(projection='3d')
ax.scatter(X[..., 0], X[..., 1], y, c='red', marker='o')
ax.view_init(0, -90)
```



In [4]:

```
import numpy as np
from matplotlib import pyplot as plt

from sklearn import linear_model, datasets


n_samples = 100
size = 10

X, y, coef_gt = datasets.make_regression(
    n_samples=n_samples,
    n_features=2,
    n_informative=1,
    noise=20,
    coef=True,
    random_state=42,
)
fig = plt.figure(figsize=(size, size))
ax = fig.add_subplot(projection='3d')
ax.scatter(X[..., 0], X[..., 1], y, c='red', marker='o')
ax.view_init(0, -90)
```



In [5]:
```python
table = "| x_1| x_2 | y| \n | --- | --- | --- \n"
for count, (ex, ey) in enumerate(zip(X,y)):
    table += f"| {str(ex[0])[:6]}| {str(ex[1])[:6]} | {str(ey)[:6]} \n"
    if count == 10: break
```

The data

{{print(table)}}

Living area vs Apartment Price

Living area (feet^2)	Price (1000$s)
2104	400
1600	330
2400	369
1416	232
3000	540
⋮	⋮

Linear Regression settings

We want to regress y from $\mathbf{x}$ that is we want to learn a function $f_{\boldsymbol{\theta}}$ parametrized by some parameters $\boldsymbol{\theta}$ so that $f_{\boldsymbol{\theta}}: \underbrace{\mathbf{x}}_{\text{input}} \mapsto \underbrace{y}_{\text{output}}$

- $\mathbf{x} \in \mathrm{R}^d$ (here d=2)
- y is a scalar that is continuous $y \in \mathrm{R}$
- We have a finite number of samples $D = \{\mathbf{x}_i, y_i\}_{i=1}^n \sim p(\mathbf{x}, y)$ that which labels are generated from a function f plus error so $y = f(\mathbf{x}) + \epsilon$

Linear Hypothesis

We assume relations $f \longleftrightarrow y$ is **linear**.

We know $D = \{\mathbf{x}_i, y_i\}_{i=1}^n$ and we want to find $\boldsymbol{\theta} \doteq (\theta_0, \ldots, \theta_d)$

$$f_{\boldsymbol{\theta}}(\mathbf{x}) = \theta_0 + \theta_1 \cdot x_1 + \theta_2 \cdot x_2 + \ldots + \theta_d \cdot x_d$$

So $\boldsymbol{\theta} \doteq (\theta_0, \ldots, \theta_d) \in \mathrm{R}^{d+1}$.

$$f_{\boldsymbol{\theta}}(\mathbf{x}) = \Big(\sum_{i=1}^{d} \theta_i \cdot x_i\Big) + \theta_0$$

Trick for Notation Compactness

We can augment each feature to have a **bias (intercept term)** set to 1 so that $\mathbf{x} \doteq [1, \mathbf{x}]$.

Doing so $\mathbf{x} \in \mathrm{R}^{d+1}$

$$f_{\boldsymbol{\theta}}(\mathbf{x}) = \theta_0 \cdot \underbrace{x_0}_{\text{always 1}} + \theta_1 \cdot x_1 + \theta_2 \cdot x_2 + \ldots + \theta_d \cdot x_d$$

So $\boldsymbol{\theta} \doteq (\theta_0, \ldots, \theta_d) \in \mathrm{R}^{d+1}$.

$$f_{\boldsymbol{\theta}}(\mathbf{x}) = \sum_{i=0}^{d} \theta_i \cdot x_i = \boldsymbol{\theta}^T \mathbf{x}$$

Parametric Nature

No matter how many training points N you have, the parameters are fixed in $\boldsymbol{\theta}$.

Note that $\boldsymbol{\theta} \in \mathrm{R}^{d+1}$.

$$f_{\boldsymbol{\theta}}(\mathbf{x}) = \sum_{i=0}^{d} \theta_i \cdot x_i = \boldsymbol{\theta}^T \mathbf{x}$$

Loss or Cost Function for Linear Regression

You see now that the loss is more explicit compared to non-parametric models (K-NN, Decision Trees).

$$\mathrm{J}(\theta; \mathbf{x}, y) = \frac{1}{2} \sum_{i=1}^{n} \mathrm{L}\big(y_i, f_{\boldsymbol{\theta}}(\mathbf{x}_i)\big)$$

where

$$\mathrm{L}\big(y, f_{\boldsymbol{\theta}}(\mathbf{x})\big) = \big(f_{\boldsymbol{\theta}}(\mathbf{x}) - y\big)^2$$

The loss is **the squared error.**

{{eps = np.arange(-100,100);plt.plot(eps,eps**2);plt.xlabel('Difference');_=plt.ylabel('Cost');}}

Minimize the Total Loss with a Closed Form Solution

We need to minimize

$$\mathrm{J}(\theta; \mathbf{x}, y) = \frac{1}{2}\sum_{i=1}^{n} \mathrm{L}\big(y_i, f_{\boldsymbol{\theta}}(\mathbf{x}_i)\big)$$

so to find:

$$\theta^{\star} = \arg\min_{\theta} \mathrm{J}(\theta; \mathbf{x}, y)$$

Explicit Cost

$$\mathrm{J}(\theta; \mathbf{x}, y) = \frac{1}{2}\sum_{i=1}^{n} (\underbrace{\boldsymbol{\theta}^T\mathbf{x}}_{f_{\boldsymbol{\theta}}}{}_i - y_i)^2$$

Vectorizing the Explicit Cost

$$\mathrm{J}(\theta; \mathbf{x}, y) = \frac{1}{2}\sum_{i=1}^{n} (\underbrace{\boldsymbol{\theta}^T\mathbf{x}}_{f_{\boldsymbol{\theta}}}{}_i - y_i)^2$$

We define the **design matrix** $\mathbf{X} \in \mathrm{R}^{n \times (d+1)}$ and **label matrix** $\mathbf{y} \in \mathrm{R}^{n \times 1}$

$$\mathbf{X} = \begin{bmatrix} — & \mathbf{x}_1^T & — \\ — & \mathbf{x}_2^T & — \\ & \vdots & \\ — & \mathbf{x}_n^T & — \end{bmatrix} \qquad \mathbf{y} = \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_n \end{bmatrix}$$

The parameters $\boldsymbol{\theta} \in \mathrm{R}^{d+1}$ are:

$$\boldsymbol{\theta} = \begin{bmatrix} \theta_0 \\ \theta_1 \\ \vdots \\ \theta_d \end{bmatrix}$$

Vectorizing the Explicit Cost

$$\underbrace{\mathbf{X}}_{\mathrm{R}^{n \times (d+1)}} \underbrace{\boldsymbol{\theta}}_{\mathrm{R}^{(d+1) \times 1}} - \underbrace{\mathbf{y}}_{\mathrm{R}^n}$$

$$\underbrace{\mathbf{X}\boldsymbol{\theta}}_{\mathrm{R}^n} - \underbrace{\mathbf{y}}_{\mathrm{R}^n}$$

$$\mathbf{X}\boldsymbol{\theta} - \mathbf{y} = \begin{bmatrix} \mathbf{x}_1^T\boldsymbol{\theta} - y_1 \\ \mathbf{x}_2^T\boldsymbol{\theta} - y_2 \\ \vdots \\ \mathbf{x}_n^T\boldsymbol{\theta} - y_n \end{bmatrix}$$

Vectorizing the Explicit Cost

$$\mathrm{J}(\theta; \mathbf{X}, \mathbf{y}) = \frac{1}{2}\left(\mathbf{X}\boldsymbol{\theta} - \mathbf{y}\right)^T\left(\mathbf{X}\boldsymbol{\theta} - \mathbf{y}\right) = \frac{1}{2}\sum_{i=1}^{n}(\underbrace{\boldsymbol{\theta}^T\mathbf{x}_i}_{f_\theta} - y_i)^2$$

Solve it

Set the gradient to zero to find **critical points**:

$$\nabla_\theta \mathrm{J}(\theta; \mathbf{X}, \mathbf{y}) = 0$$

$$\nabla_\theta \frac{1}{2}\left(\mathbf{X}\boldsymbol{\theta} - \mathbf{y}\right)^T\left(\mathbf{X}\boldsymbol{\theta} - \mathbf{y}\right) = 0$$

For now forget the "equal to zero":

$$\nabla_\theta \frac{1}{2}\left[(\mathbf{X}\boldsymbol{\theta})^T(\mathbf{X}\boldsymbol{\theta}) - (\underbrace{\mathbf{X}\boldsymbol{\theta})^T\mathbf{y}}_{\text{scalar}} - \mathbf{y}^T(\underbrace{\mathbf{X}\boldsymbol{\theta}}_{\text{scalar}}) + \mathbf{y}^T\mathbf{y})\right]$$

$$\nabla_\theta \frac{1}{2}\left[(\mathbf{X}\boldsymbol{\theta})^T(\mathbf{X}\boldsymbol{\theta}) - 2\boldsymbol{\theta}^T(\mathbf{X}^T\mathbf{y}) + \mathbf{y}^T\mathbf{y})\right]$$

$$\nabla_\theta \frac{1}{2}\left[\boldsymbol{\theta}^T(\mathbf{X}^T\mathbf{X})\boldsymbol{\theta} - 2\boldsymbol{\theta}^T(\mathbf{X}^T\mathbf{y}) + \mathbf{y}^T\mathbf{y})\right]$$

$$\frac{1}{2}\left[2\mathbf{X}^T\mathbf{X}\boldsymbol{\theta} - 2\mathbf{X}^T\mathbf{y}\right]$$

Set the gradient to zero

$$\frac{1}{2}\left[2\mathbf{X}^T\mathbf{X}\boldsymbol{\theta} - 2\mathbf{X}^T\mathbf{y}\right] = 0$$

To get the normal equation

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

Final Least Squares solution

Assumes $\mathbf{X}^T\mathbf{X}$ is **invertible**:

$$\boldsymbol{\theta} = \underbrace{(\mathbf{X}^T\mathbf{X})^{-1}\mathbf{X}^T}_{\text{pseudo inverse}}\mathbf{y} = \mathbf{X}^+\mathbf{y}$$

where:

$$\mathbf{X}^+ \doteq (\mathbf{X}^T\mathbf{X})^{-1}\mathbf{X}^T$$

In [6]:

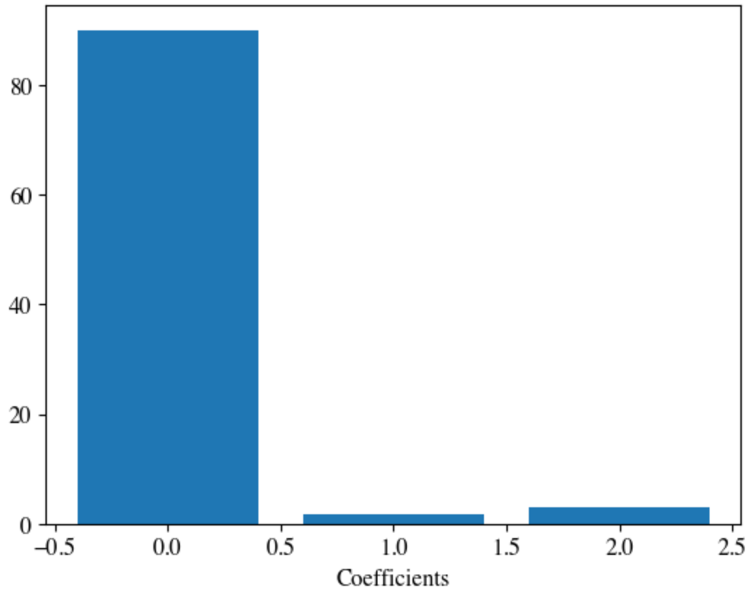
```python
%matplotlib notebook
from sklearn import linear_model, datasets
from matplotlib import pyplot as plt
import numpy as np

n_samples = 100
size = 8

X, y, coef_gt = datasets.make_regression(
    n_samples=n_samples,
    n_features=2,
    n_informative=1,
    noise=20,
    coef=True,
    random_state=42,
)
fig = plt.figure(figsize=(size, size))
ax = fig.add_subplot(projection='3d')
# Linear Regression
bias = np.ones((X.shape[0], 1))
X = np.hstack((X, bias))
theta = np.linalg.inv(X.T@X)@X.T@y
# Now MeshGrid
Xmin, Xmax = X.min(), X.max()
support = np.linspace(Xmin, Xmax, 10)
xx, yy = np.meshgrid(support, support)
data = np.stack((xx, yy), axis=2)
data = data.reshape(-1, 2)
data = np.hstack((data, np.ones((data.shape[0], 1))))
z = np.dot(theta, data.T)
z = z.reshape(xx.shape)
ax.plot_surface(xx, yy, z, alpha=0.2)
ax.scatter(X[..., 0], X[..., 1], y, c='red', marker='o')
ax.view_init(0, 90)
```

Debugging the Coefficients

$$f_{\theta}(\mathbf{x}) = \sum_{i=0}^{d} \theta_i \cdot x_i = \boldsymbol{\theta}^T \mathbf{x}$$

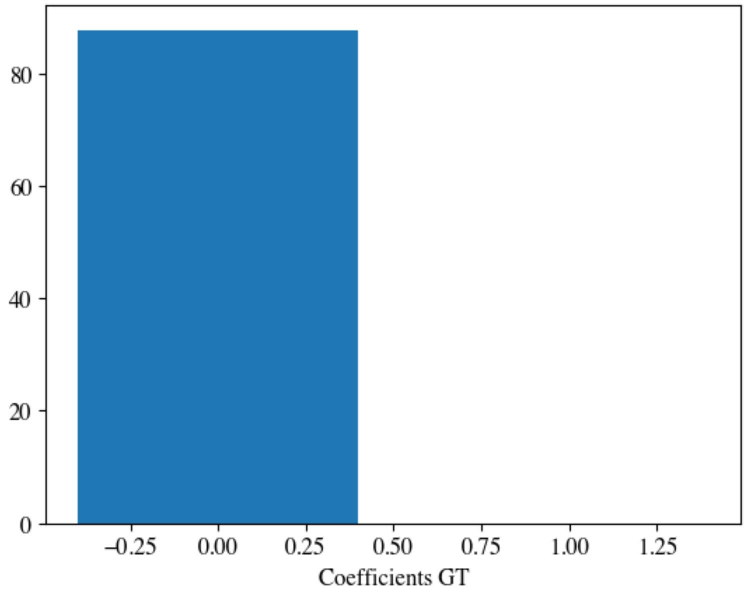
In [7]:

```
%matplotlib inline
plt.figure()
plt.bar(list(range(theta.size)),theta); plt.xlabel('Coefficients');
```

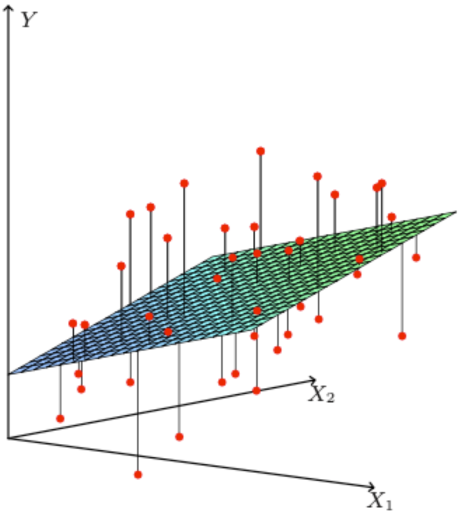


In [8]:

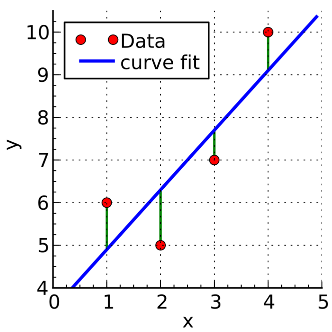
```
%matplotlib inline
plt.figure()
plt.bar(list(range(coef_gt.size)),coef_gt); plt.xlabel('Coefficients GT');
```



Important: The distance is NOT orthogonal



Important: The distance is NOT orthogonal (1D case)



Interpretation as solving a overdetermined Linear System ($n \gg d$)

Assumes $\mathbf{X}^T\mathbf{X}$ is **invertible** (full rank) and $n \gg d$:

$$\theta = (\underbrace{\mathbf{X}^T\mathbf{X}}_{d \times d})^{-1}\underbrace{\mathbf{X}^T}_{d \times n}\underbrace{\mathbf{y}}_{n \times 1}$$

The normal equation gives you a way to invert $\mathbf{X}^T\mathbf{X}$

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

it solves the linear system so that the plane best fit all the points with a trade-off given by the square of the residuals **(least squares)**.

What happens if $n = d + 1$?

$$\underbrace{\mathbf{X}}_{n \times n}\theta = \mathbf{y}$$

We can invert it "directly"

$$\theta = \mathbf{X}^{-1}\mathbf{y}$$

Why?

$$(\mathbf{AB})^{-1} = (\mathbf{B}^{-1}\mathbf{A}^{-1})$$

then:

$$\theta = (\mathbf{X}^T\mathbf{X})^{-1}\mathbf{X}^T\mathbf{y} = \mathbf{X}^{-1}(\mathbf{X}^T)^{-1}\mathbf{X}^T\mathbf{y} = \mathbf{X}^{-1}\mathbf{y}$$

In [9]:

```
%matplotlib notebook
from sklearn import linear_model, datasets
from matplotlib import pyplot as plt
import numpy as np

n_samples = 3
size = 12

X, y, coef_gt = datasets.make_regression(
    n_samples=n_samples,
    n_features=2,
    n_informative=1,
    noise=20,
    coef=True,
    random_state=42,
)
fig = plt.figure(figsize=(size, size))
ax = fig.add_subplot(projection='3d')
ax.scatter(X[..., 0], X[..., 1], y, c='blue', marker='o')
# Linear Regression
bias = np.ones((X.shape[0], 1))
X = np.hstack((X, bias))
theta = np.linalg.inv(X)@y
# Now MeshGrid
Xmin, Xmax = X.min(), X.max()
support = np.linspace(Xmin, Xmax, 10)
xx, yy = np.meshgrid(support, support)
data = np.stack((xx, yy), axis=2)
data = data.reshape(-1, 2)
data = np.hstack((data, np.ones((data.shape[0], 1))))
z = np.dot(theta, data.T)
z = z.reshape(xx.shape)
ax.plot_surface(xx, yy, z, alpha=0.2)
ax.scatter(X[..., 0], X[..., 1], y, c='red', marker='o')
ax.view_init(0, 90)
```

What happens if $n = d$?

We see the plane passes exactly through the "training points".

Probabilistic Interpretation

Probabilistic Interpretation for Linear Regression

To go probabilistic, we have to make an assumption that each y is generated linearly but with **additive Gaussian Noise.**

So

$$y_i = \boldsymbol{\theta}^T \mathbf{x}_i + \epsilon$$

where

$$\epsilon = \mathrm{N}(0, \sigma^2)$$

- We observe $(\mathbf{x}_i, y_i)$ but we do not know $\boldsymbol{\theta}$ and the noise ϵ.
- The noise changes from sample to sample but we know it is distributed as Gaussian.

Probabilistic Interpretation for Linear Regression

To go probabilistic, we have to make an assumption that each y is generated linearly but with **additive Gaussian Noise.**

So

$$\epsilon = y_i - \boldsymbol{\theta}^T \mathbf{x}_i \sim \mathrm{N}(0, \sigma^2)$$

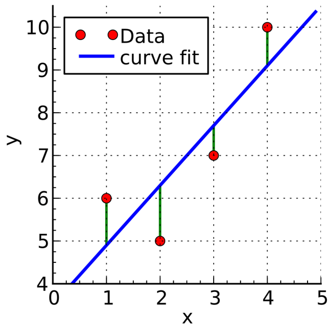
- We observe $(\mathbf{x}, y_i)$ but we do not know $\boldsymbol{\theta}$ and the noise ϵ.
- The noise changes from sample to sample but we know it is distributed as Gaussian.

What does the noise look like?

In [10]:

```
%matplotlib notebook
from sklearn import linear_model, datasets
from matplotlib import pyplot as plt
import numpy as np

n_samples = 100
size = 5

X, y, coef_gt = datasets.make_regression(
    n_samples=n_samples,
    n_features=2,
    n_informative=1,
    noise=0,
    coef=True,
    random_state=42,
)
fig = plt.figure(figsize=(size, size))
ax = fig.add_subplot(projection='3d')
# Linear Regression
bias = np.ones((X.shape[0], 1))
X = np.hstack((X, bias))
theta = np.linalg.inv(X.T@X)@X.T@y
# Now MeshGrid
Xmin, Xmax = X.min(), X.max()
support = np.linspace(Xmin, Xmax, 10)
xx, yy = np.meshgrid(support, support)
data = np.stack((xx, yy), axis=2)
data = data.reshape(-1, 2)
data = np.hstack((data, np.ones((data.shape[0], 1))))
z = np.dot(theta, data.T)
z = z.reshape(xx.shape)
ax.plot_surface(xx, yy, z, alpha=0.2)
ax.scatter(X[..., 0], X[..., 1], y, c='red', marker='.')
ax.view_init(0, 90)
ey = y[0]
yn = ey + np.random.randn(20)*10
for yns in yn:
    ax.scatter(X[0, 0], X[0, 1], yns, c='blue', marker='o')
ax.scatter(X[0, 0], X[0, 1], ey, c='green', marker='o');
```

What does the noise look like?



Probabilistic Interpretation for Linear Regression

To go probabilistic, we have to make an assumption that each y is generated linearly but with **additive Gaussian Noise.**

So

$$\epsilon_i = y_i - \boldsymbol{\theta}^T \mathbf{x}_i \sim \mathrm{N}(0, \sigma^2)$$

which measn the **errors behaves IID from a Normal Distribution.**

$$p\left(\epsilon_i\right) = \frac{1}{\sqrt{2\pi}\sigma} \exp\left(-\frac{\epsilon_i^2}{2\sigma^2}\right)$$

Probabilistic Interpretation for Linear Regression

We look at the conditional probability of y given $\mathbf{x}$ aka $p(y \mid \mathbf{x}; \boldsymbol{\theta})$:

$$p\left(y_i \mid x_i; \theta\right) = \frac{1}{\sqrt{2\pi}\sigma} \exp\left(-\frac{\left(y_i - \theta^T x_i\right)^2}{2\sigma^2}\right)$$

now is function of y yet centered on $\theta^T x_i$:

$$y_i \mid x_i; \theta \sim \mathrm{N}\left(\theta^T x_i, \sigma^2\right)$$

Estimate $\boldsymbol{\theta}$ by Maximum Likelihood (MLE)

For a single training point:

$$p\left(y_i \mid x_i; \theta\right) \doteq L(\boldsymbol{\theta}; \mathbf{x}_i, y_i) = \frac{1}{\sqrt{2\pi}\sigma} \exp\left(-\frac{\left(y_i - \theta^T x_i\right)^2}{2\sigma^2}\right)$$

Estimate $\boldsymbol{\theta}$ by Maximum Likelihood (MLE)

For multiple training point $\{\mathbf{x}_i, y_i\}$, given IID assumptions on ϵ and thus $y \mid x$

$$\begin{aligned} L(\theta) &= \prod_{i=1}^{n} p\left(y^{(i)} \mid x^{(i)}; \theta\right) \\ &= \prod_{i=1}^{n} \frac{1}{\sqrt{2\pi}\sigma} \exp\left(-\frac{\left(y^{(i)} - \theta^T x^{(i)}\right)^2}{2\sigma^2}\right) \end{aligned}$$

Maximizing the Log Likelihood (MLE)

$$\begin{aligned} \ell(\theta) &= \log L(\theta) \\ &= \log \prod_{i=1}^{n} \frac{1}{\sqrt{2\pi}\sigma} \exp\left(-\frac{\left(y^{(i)} - \theta^T x^{(i)}\right)^2}{2\sigma^2}\right) \\ &= \sum_{i=1}^{n} \log \frac{1}{\sqrt{2\pi}\sigma} \exp\left(-\frac{\left(y^{(i)} - \theta^T x^{(i)}\right)^2}{2\sigma^2}\right) \\ &= n \log \frac{1}{\sqrt{2\pi}\sigma} - \frac{1}{\sigma^2} \cdot \frac{1}{2} \sum^{n} \left(y^{(i)} - \theta^T x^{(i)}\right)^2 \end{aligned}$$

Maximizing the Log Likelihood (MLE) equals Minimizing the Squared Loss

(Under the assumption that the errors will distribution as Gaussians)

$$\arg\max_{\theta} \; n\log\frac{1}{\sqrt{2\pi}\sigma} - \frac{1}{\sigma^2}\cdot\frac{1}{2}\sum^{n}\left(y^{(i)} - \theta^T x^{(i)}\right)^2 \rightarrow \arg\min_{\theta} \; \frac{1}{2}\sum_{i=1}^{n}\left(y^{(i)} - \theta^T x^{(i)}\right)^2$$

`To summarize: Under the previous probabilistic assumptions on the data, least-squares regression corresponds to finding the maximum likelihood estimate of θ. This is thus one set of assumptions under which least-squares regression can be justified as a very natural method that’s just doing maximum likelihood estimation.`

(Note however that the probabilistic assumptions are by no means necessary for least-squares to be a perfectly good and rational procedure, and there may—and indeed there are—other natural assumptions that can also be used to justify it.)

Let's assume we could not find a closed form solution but we know how to program plus a bit of calculus, can we still solve Linear Regression?

We cannot derive a closed form solution...

We need to minimize

$$\mathrm{J}(\theta; \mathbf{x}, y) = \frac{1}{2}\sum_{i=1}^{n}\mathrm{L}\left(y_i, f_{\boldsymbol{\theta}}(\mathbf{x}_i)\right)$$

so to find:

$$\theta^{\star} = \arg\min_{\theta} \mathrm{J}(\theta; \mathbf{x}, y)$$

👋🏼Closed Form Solution; 🤗Iterative Methods

In general, if you can find a closed form solution, that is **the best you can do.**

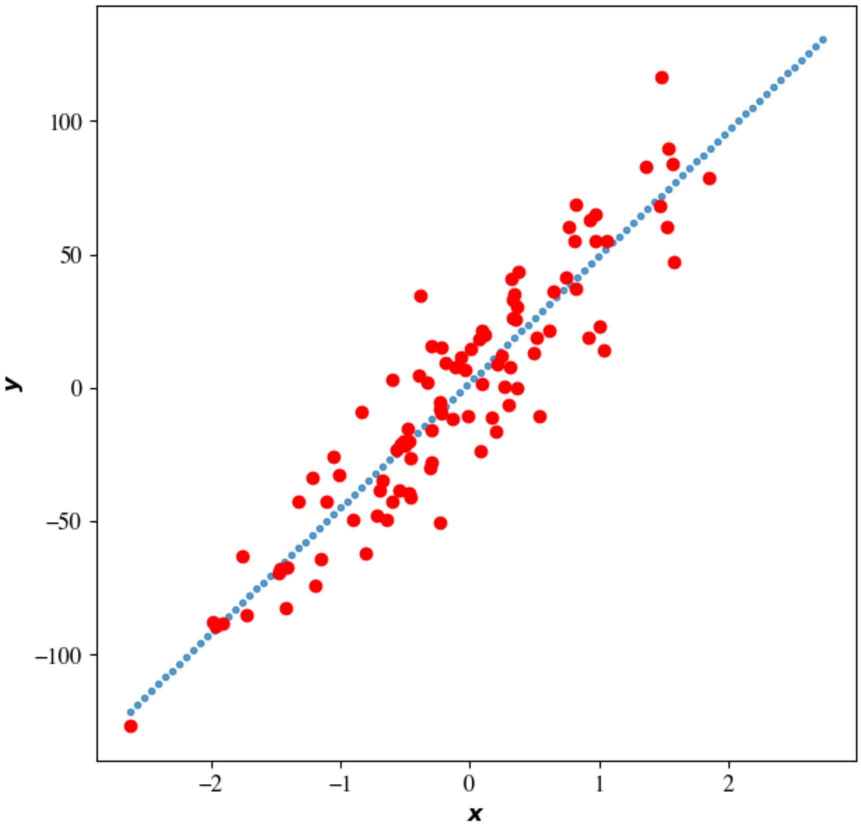
So if your problem is as simple as inverting a linear system, please **invert a linear system and use pseudo-inverse if you need to!**

In case you cannot derive, we can use **numerical, iterative methods**

A very simple yet effective Iterative method is Gradient Descent

- This part that we explain now starts to be propaedeutic for **Deep Learning**.

In [11]:
```
%matplotlib inline
from sklearn import linear_model, datasets
from matplotlib import pyplot as plt
import numpy as np

n_samples = 100
size = 7

X, y, coef_gt = datasets.make_regression(
    n_samples=n_samples,
    n_features=1,
    n_informative=1,
    noise=20,
    coef=True,
    random_state=42,
)
fig = plt.figure(figsize=(size, size))
ax = fig.add_subplot()
# Linear Regression
bias = np.ones((X.shape[0], 1))
X = np.hstack((X, bias))
theta = np.linalg.inv(X.T@X)@X.T@y
# Now MeshGrid
x_interp = np.linspace(Xmin, Xmax, 100)
x_interp = x_interp.reshape(-1,1)
x_interp = np.c_[x_interp,np.ones_like(x_interp)]
y_interp = np.dot(theta, x_interp.T)
ax.scatter(x_interp[:,0], y_interp, alpha=0.7, marker='.')
ax.scatter(X[..., 0], y, c='red', marker='o')
ax.set_xlabel('$x$');
ax.set_ylabel('$y$');
```

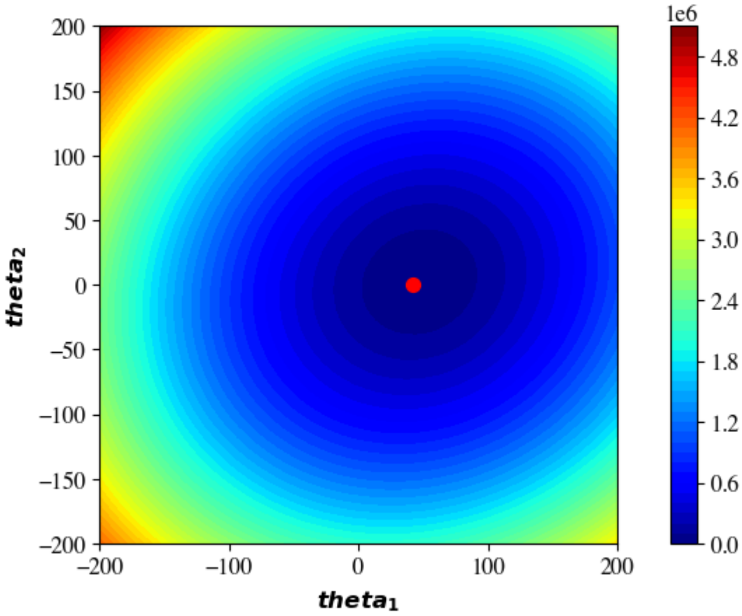


In [12]:
```
# do mesh grid on possible theta, evaluate the loss and plot the countf
theta_sampling = 50
theta_space = np.linspace(-200, 200, theta_sampling)
xxt, yyt = np.meshgrid(theta_space, theta_space)
xxt = xxt.flatten()
yyt = yyt.flatten()
all_coeff = np.stack((xxt, yyt), axis=1)
loss = 0.5*1
```

In [13]:
```
losses = []
for coeff in all_coeff:
    diff = np.dot(X, coeff.T) - y
    losses.append(0.5*np.dot(diff.T, diff))
losses = np.array(losses)
losses = losses.reshape(theta_sampling, theta_sampling)
xxt = xxt.reshape(theta_sampling, theta_sampling)
yyt = yyt.reshape(theta_sampling, theta_sampling)
```

In [14]:

```
plt.rcParams['axes.grid'] = False
plt.contourf(xxt, yyt, losses, levels=50, cmap='jet')
plt.colorbar()
plt.scatter(coef_gt, 0, color='red', marker='o', s=50)
plt.axis('scaled')
plt.xlabel('$theta_1$')
plt.ylabel('$theta_2$');
```

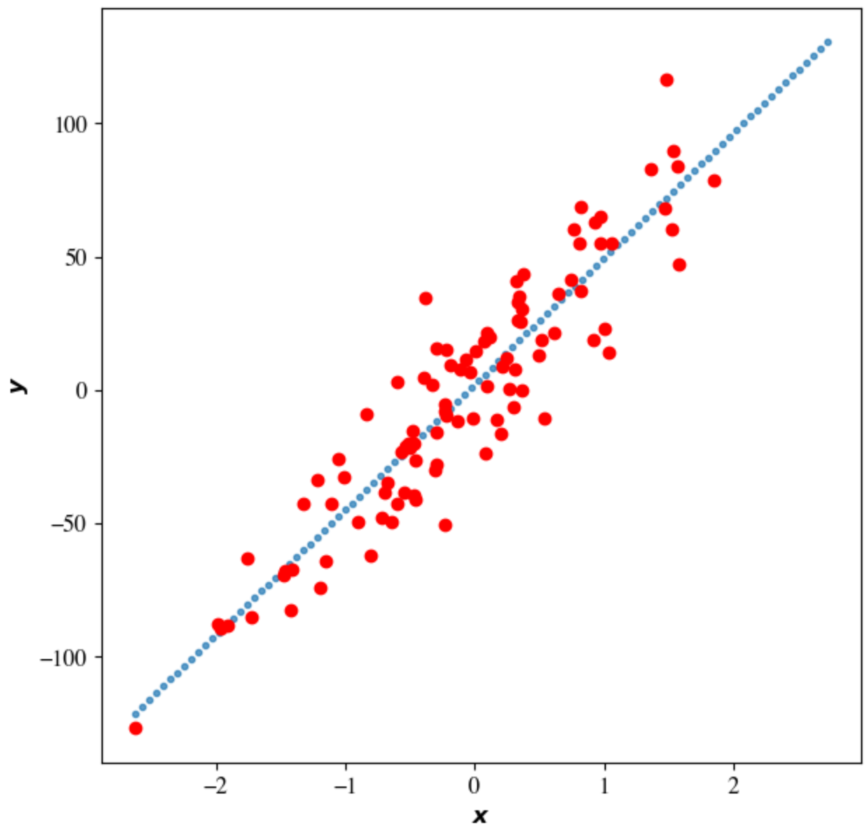


In [15]:

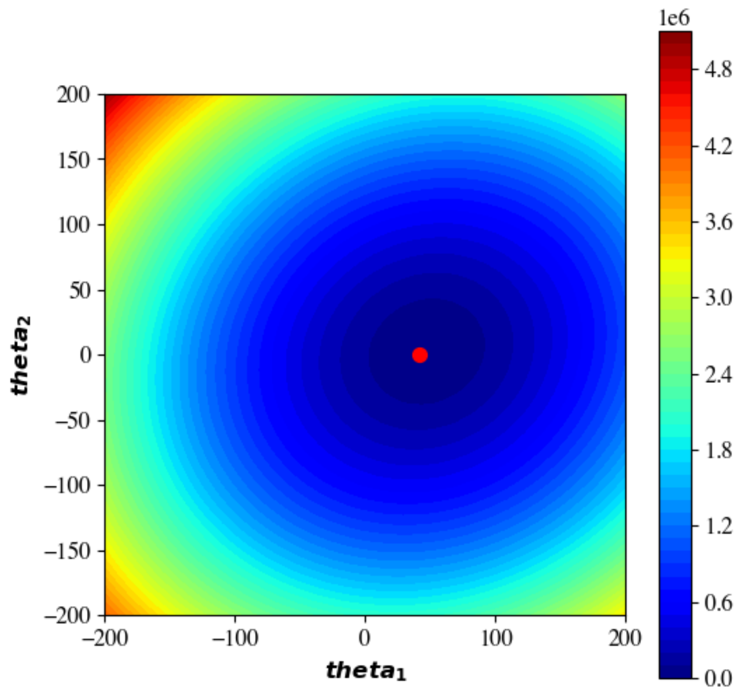
```
%matplotlib notebook
fig = plt.figure(figsize=(7,7))
ax = fig.add_subplot(111, projection='3d')
ax.plot_surface(xxt, yyt, losses)
ax.set_xlabel('$theta_1$')
ax.set_ylabel('$theta_2$')
ax.set_zlabel('loss')
plt.show()
```

In [16]:

```
%matplotlib inline
from sklearn import linear_model, datasets
from matplotlib import pyplot as plt
import numpy as np

n_samples = 100
size = 7

X, y, coef_gt = datasets.make_regression(
    n_samples=n_samples,
    n_features=1,
    n_informative=1,
    noise=20,
    coef=True,
    random_state=42,
)
fig = plt.figure(figsize=(size, size))
ax = fig.add_subplot()
# Linear Regression
bias = np.ones((X.shape[0], 1))
X = np.hstack((X, bias))
theta = np.linalg.inv(X.T@X)@X.T@y
# Now MeshGrid
x_interp = np.linspace(Xmin, Xmax, 100)
x_interp = x_interp.reshape(-1,1)
x_interp = np.c_[x_interp,np.ones_like(x_interp)]
y_interp = np.dot(theta, x_interp.T)
ax.scatter(x_interp[:,0], y_interp, alpha=0.7, marker='.')
ax.scatter(X[..., 0], y, c='red', marker='o')
ax.set_xlabel('$x$');
ax.set_ylabel('$y$');
```



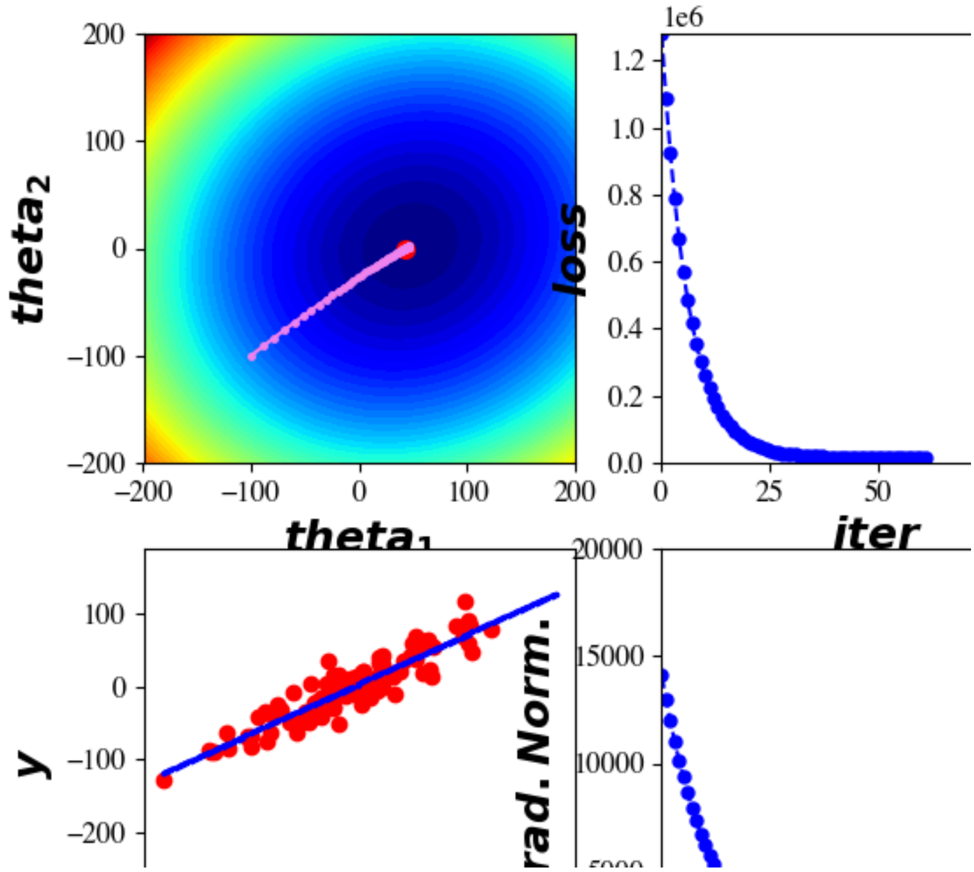
In [17]:
```
fig = plt.figure(figsize=(size-1, size-1))
plt.rcParams['axes.grid'] = False
plt.contourf(xxt, yyt, losses, levels=50, cmap='jet');
plt.colorbar()
plt.scatter(coef_gt,0,color='red',marker='o',s=50)
plt.axis('scaled')
plt.xlabel('$theta_1$')
plt.ylabel('$theta_2$');
```



Ready for an awesome demo?

In [18]:

```python
## Implementation of Gradient Descent for Logistic Regression

import time
%matplotlib notebook


def get_diff(X, theta, y):
    return X@theta - y[..., np.newaxis]


def get_loss(diff):
    return 0.5*np.dot(diff.T, diff)


def plot_line(plot3, theta):
    x_interp = np.linspace(Xmin, Xmax, 100)
    x_interp = x_interp.reshape(-1, 1)
    x_interp = np.c_[x_interp, np.ones_like(x_interp)]
    y_interp = np.dot(x_interp, theta)
    if plot3:
        plot3.set_xdata(x_interp[:, 0])
        plot3.set_ydata(y_interp)
    else:
        return x_interp, y_interp


plt.ion()
figure, (axes_1, axes_2) = plt.subplots(2, 2, figsize=(7, 7))
plt.rcParams['axes.grid'] = False
ax0, ax1 = axes_1
ax2, ax3 = axes_2
ax0.contourf(xxt, yyt, losses, levels=50, cmap='jet')
ax0.scatter(coef_gt, 0, color='red', marker='o', s=50)
ax0.set_xlabel('$theta_1$',fontsize=18)
ax0.set_ylabel('$theta_2$',fontsize=18)
ax1.set_ylabel('$loss$',fontsize=18)
ax1.set_xlabel('$iter$',fontsize=18)
ax1.set(xlim=(0, 100), ylim=(0, 1.28e6))
ax2.scatter(X[..., 0], y, c='red', marker='o')
ax2.set_xlabel('$x$',fontsize=18)
ax2.set_ylabel('$y$',fontsize=18)
ax3.set_ylabel('$Grad. Norm.$',fontsize=18)
ax3.set_xlabel('$iter$',fontsize=18)
ax3.set(xlim=(0, 100), ylim=(0, 20000))

theta_curr = np.array([[-100, -100]]).T
losses_track = [get_loss(get_diff(X, theta_curr, y))]
grad_norm_track = [1000]

theta_track = np.array(theta_curr)
lr = 1e-3
loss_tol = 10

plot1, = ax0.plot(*theta_curr, color='violet',
                  marker='.', markersize=5, linestyle='-')
plot2, = ax1.plot(*losses_track, color='blue',
                  marker='.', markersize=10, linestyle='--')
xi, yi = plot_line(None, theta_curr)
plot3a,plot3b = ax2.plot(xi, yi, color='blue', marker='.',
                  markersize=3, linestyle='--')
plot4, = ax3.plot(1000, color='blue',
                  marker='.', markersize=10, linestyle='--')
while True:
    diff = get_diff(X, theta_curr, y)
    grad = (diff * X).sum(axis=0, keepdims=True).T
    theta_curr = theta_curr - lr*grad
    theta_track = np.append(theta_track, theta_curr, axis=1)
    diff = get_diff(X, theta_curr, y)
    losses_track.append(get_loss(diff))
    grad_norm_track.append(np.linalg.norm(grad,2))
    if abs(losses_track[-2]-losses_track[-1]) < loss_tol:
        break
    plot1.set_xdata(theta_track[0, :])
    plot1.set_ydata(theta_track[1, :])
    plot2.set_xdata(range(len(losses_track)))
    plot2.set_ydata(losses_track)
    plot4.set_xdata(range(len(grad_norm_track[1:])))
    plot4.set_ydata(grad_norm_track[1:])
    plot_line(plot3a, theta_curr)
    plot_line(plot3b, theta_curr)
    figure.canvas.draw()
    figure.canvas.flush_events()
    time.sleep(0.1)
print(*theta_curr)
plt.show()
```




```
[46.09610615] [1.80913084]
```

Gradient Descent (GD)

$$J(\theta; \mathbf{x}, y) = \frac{1}{2}\sum_{i=1}^{n} L\big(y_i, f_\theta(\mathbf{x}_i)\big)$$

where

$$L\big(y, f_\theta(\mathbf{x})\big) = \big(f_\theta(\mathbf{x}) - y\big)^2$$

$$\hat{\theta} = \arg\min_\theta \frac{1}{2}\sum_{i=1}^{n}\big(f_\theta(\mathbf{x}_i) - y_i\big)^2$$

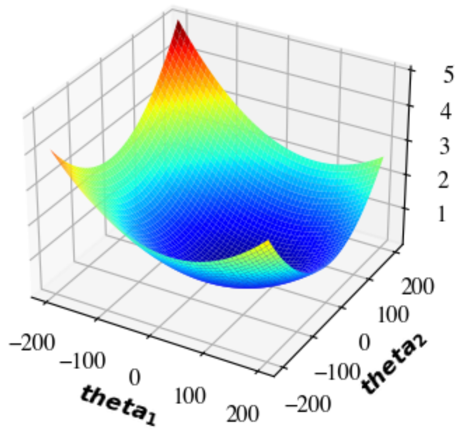
Analysis

The function $J(\theta; \mathbf{x}, y)$ is a **convex quadratic function**. The Hessian of $J(\theta; \mathbf{x}, y)$ at any vector θ is the positive definite matrix $\mathbf{X}^T\mathbf{X}$. Since J is lower bounded and grows at infinity, there is a minimum.

- if $\mathrm{rank}(\mathbf{X}) = \min\{d, n\}$ then $\mathbf{X}^T\mathbf{X}$ is strictly positive definite. In this case the error function J is strictly convex, so the **minimum is unique (Ball Shape)**
- if $\mathrm{rank}(\mathbf{X}) < \min\{d, n\}$ then then J is not strictly convex and the minimum is not unique

In [37]:

```
plt.rcParams['axes.grid'] = False
fig = plt.figure(figsize=(4, 4))
ax = fig.add_subplot(111, projection='3d')
ax.plot_surface(xxt, yyt, losses, cmap='jet')
ax.set_xlabel('$theta_1$')
ax.set_zlabel('$J$')
ax.set_ylabel('$theta_2$');
```



Gradient Descent Algorithm as an Iterative Method

Idea: make a little step so that locally after each step the cost is lower than before
Input: Training set $\{\mathbf{x}_i, y_i\}$, learning rate γ, a small value in $\{0.1, \ldots, 1\text{e-}6\}$.

1. **Initialization - Very Important if the function is not strictly convex**

$$\boldsymbol{\theta} \doteq \mathbf{0}^T$$

Set it to all zeros or random initialization from a distribution. |

1. Repeat until **convergence**:
    - Compute the gradient of the loss wrt the parameters $\boldsymbol{\theta}$ given **all the training set**
    - Take a small step in the opposite direction of steepest ascent **(so steepest descent).**

$$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \gamma \nabla_{\boldsymbol{\theta}} \mathrm{J}(\theta; \mathbf{x}, y)$$

2. When convergence is reached, you final estimate is in $\boldsymbol{\theta}$

Convergence

$$\{\boldsymbol{\theta}_{t=0}, \boldsymbol{\theta}_{t=1}, \ldots, \boldsymbol{\theta}_{t=100}\}$$

0) Always: validation loss/metric *(early stopping)* (required)

1) No significant decrease in the loss function (preferred)

$$|\mathrm{J}(\theta; \mathbf{x}, y)_t - \mathrm{J}(\theta; \mathbf{x}, y)_{t-1}|$$

1) No variations in the parameters

$$||\, \theta_t - \theta_{t-1} \,||$$

2) Gradient Norm goes to zero

$$||\, \nabla_{\boldsymbol{\theta}} \mathrm{J}(\theta; \mathbf{x}, y) \,|| \rightarrow 0$$

Gradient Descent on Linear Regression

1. Initialization

$$\boldsymbol{\theta} \sim \text{random} \quad \text{or zero}$$

2.

$$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \gamma \nabla_{\boldsymbol{\theta}} \mathrm{J}(\theta; \mathbf{x}, y)$$

$$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \gamma \nabla_{\boldsymbol{\theta}} \frac{1}{2} \sum_{i=1}^{n} (\boldsymbol{\theta}^T \mathbf{x} - y)^2$$

$$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \gamma \frac{1}{2} \sum_{i=1}^{n} (2\boldsymbol{\theta}^T \mathbf{x}\mathbf{x} - 2y\mathbf{x})$$

$$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \gamma \sum_{i=1}^{n} (\boldsymbol{\theta}^T \mathbf{x}_i - y_i)\mathbf{x}_i$$

Dimension Check

Summing up over $\mathbf{x}_i \in \mathrm{R}^d$ scaled by the difference between the prediction and ground-truth

$$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \gamma \sum_{i=1}^{n} \underbrace{(\boldsymbol{\theta}^T \mathbf{x}_i - y_i)}_{\text{scalar}} \underbrace{\mathbf{x}_i}_{\mathrm{R}^d}$$

Stochastic Gradient Descent (SGD)

Problem of GD: What happens if $n \mapsto \infty$.

To make a small step we have to go through **ALL the training samples**. Optimization could be slow for large n.

Idea: make update for each single training sample selected randomly

$$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \gamma \underbrace{(\boldsymbol{\theta}^T \mathbf{x}_i - y_i)}_{\text{scalar}} \underbrace{\mathbf{x}_i}_{\mathrm{R}^d} \qquad \text{where} \qquad i \sim \mathrm{U}(0, n)$$

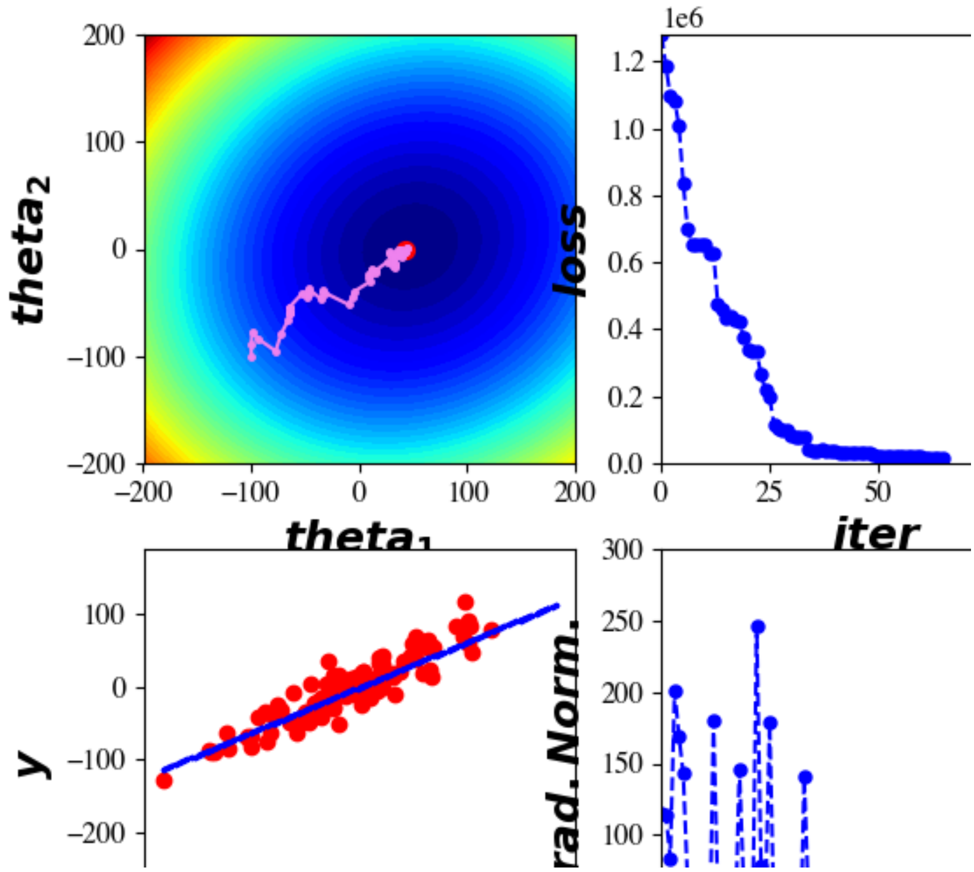
Many Variations of SGD

Let's see the dynamic of SGD!

Another demo

In [20]:

```python
# Implementation of Stochastic Gradient Descent for Logistic Regression

import time
%matplotlib notebook


def get_diff(X, theta, y):
    return X@theta - y[..., np.newaxis]


def get_loss(diff):
    return 0.5*np.dot(diff.T, diff)


def plot_line(plot3, theta):
    x_interp = np.linspace(Xmin, Xmax, 100)
    x_interp = x_interp.reshape(-1, 1)
    x_interp = np.c_[x_interp, np.ones_like(x_interp)]
    y_interp = np.dot(x_interp, theta)
    if plot3:
        plot3.set_xdata(x_interp[:, 0])
        plot3.set_ydata(y_interp)
    else:
        return x_interp, y_interp


plt.ion()
figure, (axes_1, axes_2) = plt.subplots(2, 2, figsize=(7, 7))
plt.rcParams['axes.grid'] = False
ax0, ax1 = axes_1
ax2, ax3 = axes_2
ax0.contourf(xxt, yyt, losses, levels=50, cmap='jet')
ax0.scatter(coef_gt, 0, color='red', marker='o', s=50)
ax0.set_xlabel('$theta_1$', fontsize=18)
ax0.set_ylabel('$theta_2$', fontsize=18)
ax1.set_ylabel('$loss$', fontsize=18)
ax1.set_xlabel('$iter$', fontsize=18)
ax1.set(xlim=(0, 100), ylim=(0, 1.28e6))
ax2.scatter(X[..., 0], y, c='red', marker='o')
ax2.set_xlabel('$x$', fontsize=18)
ax2.set_ylabel('$y$', fontsize=18)
ax3.set_ylabel('$Grad. Norm.$', fontsize=18)
ax3.set_xlabel('$iter$', fontsize=18)
ax3.set(xlim=(0, 100), ylim=(0, 300))

theta_curr = np.array([[-100, -100]]).T
losses_track = [get_loss(get_diff(X, theta_curr, y))]
grad_norm_track = [1000]

theta_track = np.array(theta_curr)
lr = 1e-1
loss_tol = 10
np.random.seed(42)

plot1, = ax0.plot(*theta_curr, color='violet',
                  marker='.', markersize=5, linestyle='-')
plot2, = ax1.plot(*losses_track, color='blue',
                  marker='.', markersize=10, linestyle='--')
xi, yi = plot_line(None, theta_curr)
plot3a, plot3b = ax2.plot(xi, yi, color='blue', marker='.',
                          markersize=3, linestyle='--')
plot4, = ax3.plot(1000, color='blue',
                  marker='.', markersize=10, linestyle='--')
while True:
    diff = get_diff(X, theta_curr, y)
    # STOCHASTIC PART ########################
    idx_sampled = np.random.randint(n_samples)
    grad = (diff * X)[idx_sampled, :].T.reshape(-1, 1)
    ##############################
    theta_curr = theta_curr - lr*grad
    theta_track = np.append(theta_track, theta_curr, axis=1)
    diff = get_diff(X, theta_curr, y)
    losses_track.append(get_loss(diff))
    grad_norm_track.append(np.linalg.norm(grad, 2))
    if abs(losses_track[-2]-losses_track[-1]) < loss_tol:
        break
    plot1.set_xdata(theta_track[0, :])
    plot1.set_ydata(theta_track[1, :])
    plot2.set_xdata(range(len(losses_track)))
    plot2.set_ydata(losses_track)
    plot4.set_xdata(range(len(grad_norm_track[1:])))
    plot4.set_ydata(grad_norm_track[1:])
    plot_line(plot3a, theta_curr)
    plot_line(plot3b, theta_curr)
    figure.canvas.draw()
    figure.canvas.flush_events()
    time.sleep(0.1)
print(*theta_curr)
plt.show()
```




```
[42.27801342] [-3.01691851]
```

Stochastic Gradient Descent (SGD): lots of Variations

- Zig-Zag "Noisy" Trajectory of SGD
- Converge to a γ-ball of the solution of GD
- Increases the iterations wrt GD
- Each iteration is so fast that speed of SGD much higher than GD for large training set n.

Notable Variations for Deep Learning

- SGD **on mini-batches** (a trade-off between SGD and GD)
- SGD **with momentum** (memory of previous state)

Many Variations of SGD

Artificial Intelligence and Machine Learning

Unit II

Polynomial Regression, Feature Maps, Ridge Regression

Recap

We go back to your loved 🥰Linear Algebra

Supervised, Parametric Models

1) Ordinary Linear Regression with Least Squares

2) Probabilistic Interpretation

3) Gradient Descent "Family"

Gradient Descent and [Stochastic] GD

1. **Initialization - Very Important if the function is not strictly convex**

$$\boldsymbol{\theta} \doteq \mathbf{0}^T$$

   Set it to all zeros or random initialization from a distribution.

2. Repeat until **convergence**:
   - Compute the gradient of the loss wrt the parameters $\boldsymbol{\theta}$ given **all the training set**
   - Take a small step in the opposite direction of steepest ascent **(so steepest descent).**

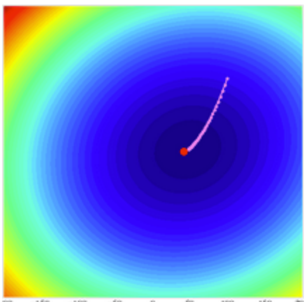
$$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \gamma \nabla_{\boldsymbol{\theta}} \mathrm{J}(\theta; \mathbf{x}, y)$$

3. When convergence is reached, you final estimate is in $\boldsymbol{\theta}$

Many Variations of SGD

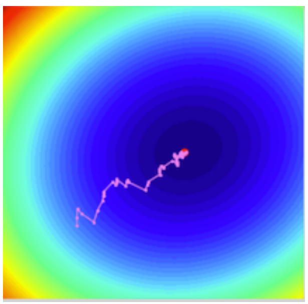
GD

$$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \gamma \sum_{i=1}^{n} (\underbrace{\boldsymbol{\theta}^T \mathbf{x}_i - y_i}_{\text{scalar}}) \underbrace{\mathbf{x}_i}_{\mathrm{R}^d}$$



SGD

$$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \gamma (\underbrace{\boldsymbol{\theta}^T \mathbf{x}_i - y_i}_{\text{scalar}}) \underbrace{\mathbf{x}_i}_{\mathrm{R}^d} \qquad \text{where } i \sim \mathrm{U}(0, n)$$



Maximizing the Log Likelihood (MLE) equals Minimizing the Squared Loss

(Under the assumption that the errors will distribution as Gaussians)

$$\arg\max_{\theta}\, n\log\frac{1}{\sqrt{2\pi}\sigma} - \frac{1}{\sigma^2}\cdot\frac{1}{2}\sum^{n}\left(y^{(i)} - \theta^T x^{(i)}\right)^2 \rightarrow \arg\min_{\theta}\, \frac{1}{2}\sum_{i=1}^{n}\left(y^{(i)} - \theta^T x^{(i)}\right)^2$$

```
To summarize: Under the previous probabilistic assumptions on the data,
least-squares regression corresponds to finding the maximum likelihood estimate of θ. This is thus
one set of assumptions under which least-squares regression can be justified as a very natural
method that's just doing maximum likelihood estimation.
```

(Note however that the probabilistic assumptions are by no means necessary for least-squares to be a perfectly good and rational procedure, and there may—and indeed there are—other natural assumptions that can also be used to justify it.)

Today

Make Linear Regression... Non-Linear

Polynomial Regression with Basis Functions (Feature Map)

From Feature Maps to Kernel Methods

This lecture material is taken from

- Mostly from Bishop - Chapter 3 page 137
- Stanford notes
- Tibshirani - Chapter 4 page 43
- Sklearn Polynomial Regression

Linear Hypothesis

We assume relations $f \longleftrightarrow y$ is **linear** We know $D = \{\mathbf{x}_i, y_i\}_{i=1}^{n}$ and we want to find $\boldsymbol{\theta} \doteq (\theta_0, \ldots, \theta_d)$

$$f_{\boldsymbol{\theta}}(\mathbf{x}) = \theta_0 + \theta_1 \cdot x_1 + \theta_2 \cdot x_2 + \ldots + \theta_d \cdot x_d$$

So $\boldsymbol{\theta} \doteq (\theta_0, \ldots, \theta_d) \in \mathrm{R}^{d+1}$.

$$f_{\boldsymbol{\theta}}(\mathbf{x}) = \left(\sum_{i=1}^{d}\theta_i \cdot x_i\right) + \theta_0$$

Trick for Notation Compactness

We can augment each feature to have a **bias (intercept term)** set to 1 so that $\mathbf{x} \doteq [1, \mathbf{x}]$.

Doing so $\mathbf{x} \in \mathrm{R}^{d+1}$

$$f_{\boldsymbol{\theta}}(\mathbf{x}) = \theta_0 \cdot \underbrace{x_0}_{\text{always 1}} + \theta_1 \cdot x_1 + \theta_2 \cdot x_2 + \ldots + \theta_d \cdot x_d$$

So $\boldsymbol{\theta} \doteq (\theta_0, \ldots, \theta_d) \in \mathrm{R}^{d+1}$.

$$f_{\boldsymbol{\theta}}(\mathbf{x}) = \sum_{i=0}^{d}\theta_i \cdot x_i = \boldsymbol{\theta}^T\mathbf{x}$$

Linear Function of parameters θ *and* input $\mathbf{x}$

With $\mathbf{x} = [1, x_1, \ldots, x_d]$ and $\theta = [\theta_0, \theta_1, \ldots, \theta_d]$, we have:

$$f_{\boldsymbol{\theta}}(\mathbf{x}) = \sum_{i=0}^{d} \theta_i \cdot x_i = \boldsymbol{\theta}^T \mathbf{x}$$

Does not capture non-linear relationships between y and $\mathbf{x}$

Interpreting Linear Regression with Basis Functions

We can have another dimensionality m instead of d by using **Basis Functions $\boldsymbol{\phi}(\mathbf{x})$**.

With $\boldsymbol{\phi}(\mathbf{x} = [1, \phi(x_1), \ldots, \phi(x_m)]$ and $\theta = [\theta_0, \theta_1, \ldots, \theta_m]$, we have:

$$f_{\boldsymbol{\theta}}(\mathbf{x}) = \sum_{i=0}^{m} \theta_i \cdot x_i = \boldsymbol{\theta}^T \boldsymbol{\phi}(\mathbf{x})$$

For Linear Regression:

- $m = d$, and
- the **Basis Functions** is : $\phi_m(\mathbf{x}) = x_m$

What if..

- $m \neq d$, and
- the **Basis Functions** is : $\boldsymbol{\phi}_m(\mathbf{x}) \neq x_m$

then we do not have Linear Regression, and the settings impact the model

Think Basis Function as (Non-Linear) Transform or Feature Map

Let's consider the one dimensional case. We have already seen that we could change the feature x to add the bias term $\mathbf{x} \doteq [x, 1]$

$$\mathbf{x} = \begin{bmatrix} x \\ 1 \end{bmatrix} \rightarrow \quad \boldsymbol{\phi}(\mathbf{x}) \quad = \begin{bmatrix} x^2 \\ x \\ 1 \end{bmatrix}$$

So input dimension is $d = 2$ then output dimension after $\boldsymbol{\phi}(\cdot)$ is $m = 3$.

In this case we used a second order polynomial to lift up the features

$$\boldsymbol{\phi}_m(\mathbf{x}) = x^m$$

Basis Function as Non-Linear Transform

$$f_{\boldsymbol{\theta}}(\mathbf{x}) = \sum_{i=0}^{m} \theta_i \cdot \boldsymbol{\phi}_i(x) = \boldsymbol{\theta}^T \boldsymbol{\phi}(\mathbf{x}) = \theta_0 + \theta_1 \cdot x + \theta_2 \cdot x^2$$

Important observation:

- This is still **linear function** of the parameters θ, in fact we still **take the dot product**
- Though it is **NON linear function** of the features x

Two Observations

- This is still **linear function** of the parameters θ
  - **Good** we can solve it with Linear Regression
- Though it is **NON linear function** of the features x
  - **Even better**, we capture non-linearity in the data

Polynomial Regression (Basis Function $\phi_m(\mathbf{x}) = x^m$)

$$f_\theta(\mathbf{x}) = \sum_{i=0}^{m} \theta_i \cdot \phi_i(x) = \boldsymbol{\theta}^T \boldsymbol{\phi}(\mathbf{x}) = \theta_0 + \theta_1 \cdot x + \theta_2 \cdot x^2$$

Important observation:

- This is still **linear function** of the parameters θ, in fact we still **take the dot product**
- Though it is **NON linear function** of the features x
- m is the degree of the **Polynomial** we consider
- m the higher the degree, the more expressive is the model

Poly (Multi) Nomial (Names or Terms)

In [21]:
```
sigma_noise = 0.5
support_X = 10
offset_valid = 7
np.random.seed(0)

def gen_data(x, sigma):
    n_samples  = x.shape[0]
    return x*np.sin(x)+ sigma*np.random.randn(n_samples)

XX = np.random.uniform(-support_X, support_X, size=n_samples)
x = XX[:80]
x_valid = np.random.uniform(-support_X-offset_valid, support_X+offset_valid, size=20)#XX[80:]
y = gen_data(x, sigma=sigma_noise).reshape(-1,1)
y_valid = gen_data(x_valid, sigma=sigma_noise).reshape(-1,1)
x = x.reshape(-1,1)
x_valid = x_valid.reshape(-1,1)
```

In [22]:
```
plt.figure(figsize=(7, 7))
_ = plt.scatter(x, y, c='red', marker='.')
_ = plt.scatter(x_valid, y_valid, c='blue', marker='.')
plt.legend(['Train', 'Valid']);
```

In [23]:
```
%matplotlib inline
```

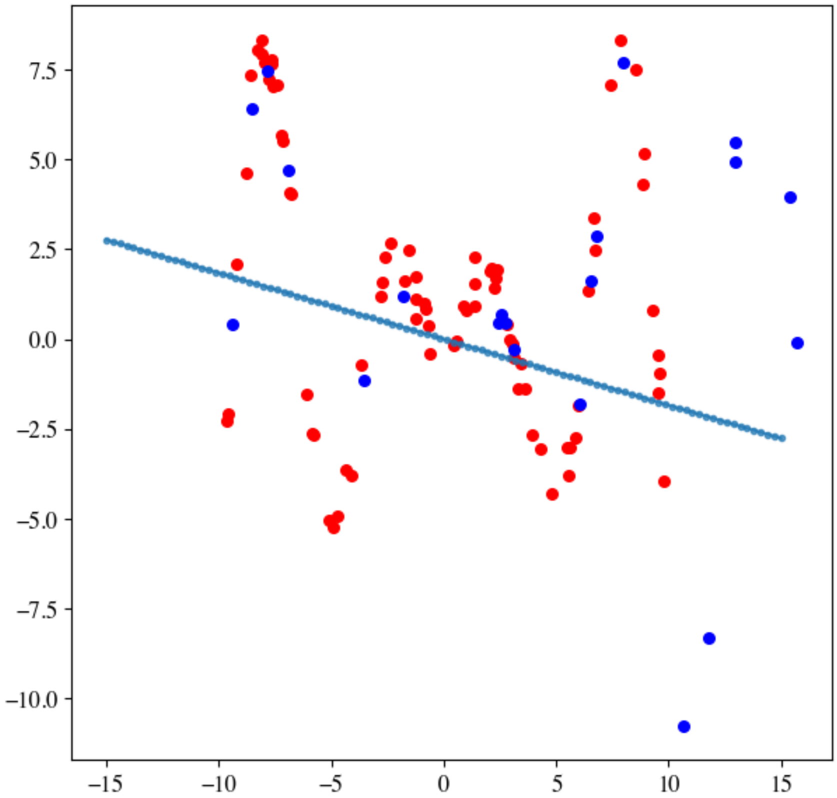
Non-Linear Data

Linear Hypothesis? 😬

{{plt.figure(figsize=(7,7)); plt.scatter(x,y, c='red', marker='o', s=30);_=plt.scatter(x_valid,y_valid, c='blue', marker='o', s=30);}}

Now let's solve it with Linear Regression (we assume no bias)

In [24]:
```
def solve_lstq(x, y):
    return np.linalg.inv(x.T@x)@x.T@y


plt.figure(figsize=(7, 7))
plt.scatter(x, y, c='red', marker='o', s=30)
plt.scatter(x_valid, y_valid, c='blue', marker='o', s=30)
theta = solve_lstq(x, y)
x_interp = np.linspace(-support_X*1.5, support_X*1.5, 100).reshape(-1, 1)
y_interp = np.dot(theta, x_interp.T)
plt.plot(x_interp[:, 0], y_interp.T, alpha=0.7, marker='.');
```



Let's check the training error (or fitting error)

```
err = np.power(y - np.dot(theta, x.T), 2).mean()
```

Numerically it seems pretty high!

{{print(np.power(y - np.dot(theta, x.T), 2).mean())}}

In [25]:
```
errors = []
errors.append(np.power(y - np.dot(theta, x.T), 2).mean())
```

Let's try a quadratic basis function

Let's consider the one dimensional case. We have already seen that we could change the feature x to add the bias term $\mathbf{x} \doteq [x, 1]$

$$\mathbf{x} = \begin{bmatrix} x \\ 1 \end{bmatrix} \rightarrow \quad \phi(\mathbf{x}) \qquad = \begin{bmatrix} x^2 \\ x \\ 1 \end{bmatrix}$$

So input dimension is $d = 1$ then output dimension after $\phi(\cdot)$ is $m = 2$.

In this case we used a second order polynomial to lift up the features

$$\phi_m(\mathbf{x}) = x^m$$

The blessing of dimensionality

Let's consider the one dimensional case. We have already seen that we could change the feature x to add the bias term $\mathbf{x} \doteq [x, 1]$

$$\mathbf{x} = \begin{bmatrix} x \\ 1 \end{bmatrix} \rightarrow \quad \phi(\mathbf{x}) \qquad = \begin{bmatrix} x^2 \\ x \\ 1 \end{bmatrix}$$

- In some sense this can be seen as opposite to the **curse of dimensionality**
- Though if you increase m too much you may face **curse of dimensionality** again

In [26]:
```
%matplotlib notebook
xq = np.c_[x, x**2]  # make quadratic features
fig = plt.figure(figsize=(size, size))
ax = fig.add_subplot(projection='3d')
ax.scatter(xq[..., 0], xq[..., 1], y, c='red', marker='o', s=30)
ax.view_init(0, -90)
```

Now we can still solve it with LS but $m = 2$

We can have another dimensionality m instead of d by using **Basis Functions** $\boldsymbol{\phi}(\mathbf{x})$.

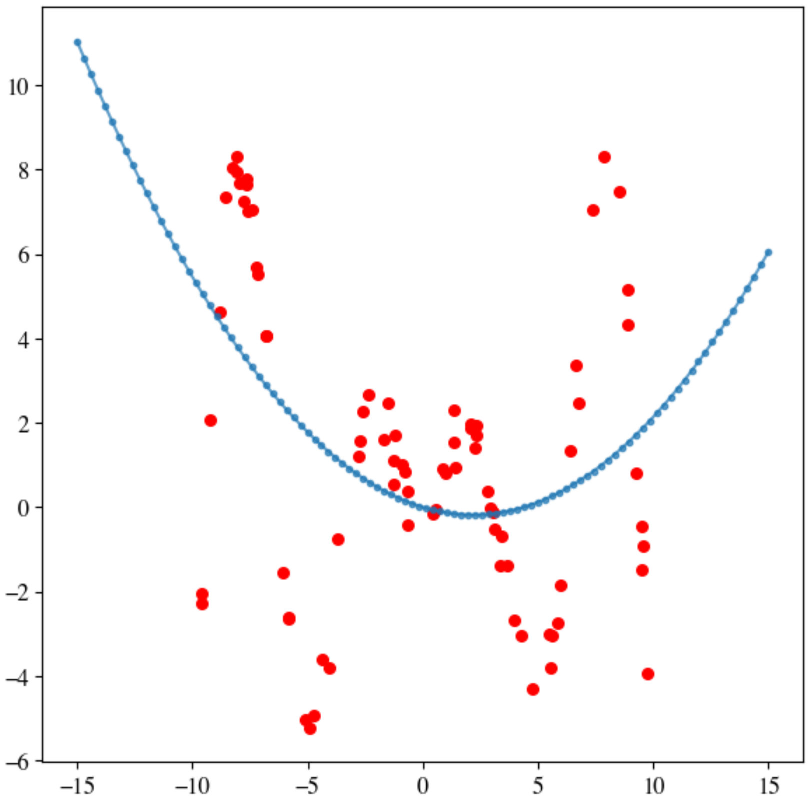
With $\boldsymbol{\phi}(\mathbf{x} = [1, \phi(x_1), \ldots, \phi(x_m)]$ and $\theta = [\theta_0, \theta_1, \ldots, \theta_m]$, we have:

$$f_{\theta}(\mathbf{x}) = \sum_{i=0}^{m} \theta_i \cdot \boldsymbol{\phi}_i(x) = \boldsymbol{\theta}^T \boldsymbol{\phi}(\mathbf{x})$$

For Linear Regression:

- $m > d$, and
- the **Basis Functions** is : $\phi_m(\mathbf{x}) = x^m$

In [27]:
```
%matplotlib inline
plt.figure(figsize=(7, 7))
plt.scatter(x, y, c='red',marker='o', s=30)
theta_q = solve_lstq(xq, y)
x_interp = np.linspace(-support_X*1.5, support_X*1.5, 100).reshape(-1, 1)
x_interp_q = np.c_[x_interp, x_interp**2]
y_interp_q = np.dot(theta_q.T, x_interp_q.T)
plt.plot(x_interp_q[:, 0], y_interp_q.T, alpha=0.7, marker='.');
```



Let's check the training error (or fitting error) again

```
err = np.power(y - np.dot(theta_q.T, xq.T), 2).mean()
```

Numerically it seems pretty high!

{{print(np.power(y - np.dot(theta_q.T, xq.T), 2).mean())}}

```
In [28]: errors.append(np.power(y - np.dot(theta_q.T, xq.T), 2).mean())
```

Now we can still solve it with LS but $m = 3$

We can have another dimensionality m instead of d by using **Basis Functions** $\phi(\mathbf{x})$.

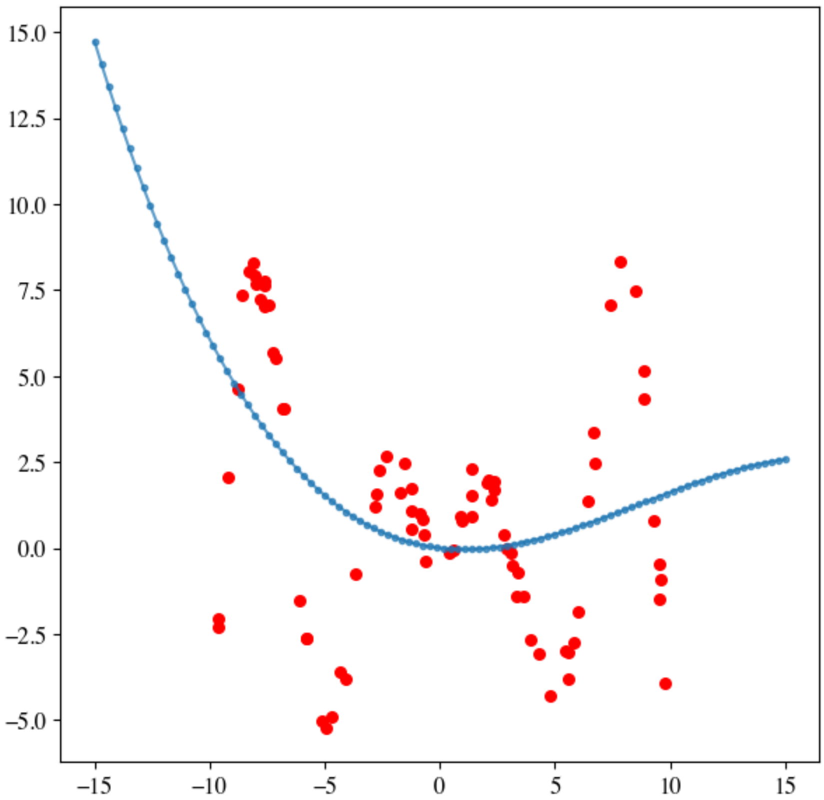
With $\phi(\mathbf{x} = [1, \phi(x_1), \ldots, \phi(x_m)]$ and $\theta = [\theta_0, \theta_1, \ldots, \theta_m]$, we have:

$$f_\theta(\mathbf{x}) = \sum_{i=0}^{m} \theta_i \cdot \phi_i(x) = \theta^T \phi(\mathbf{x})$$

For Linear Regression:

- $m > d$, and
- the **Basis Functions** is : $\phi_m(\mathbf{x}) = x^m$

```
In [29]: %matplotlib inline
xc = np.c_[x, x**2, x**3]  # make cubic features
plt.figure(figsize=(7, 7))
plt.scatter(x, y, c='red', marker='o', s=30);
theta_c = solve_lstq(xc,y)
x_interp_c = np.c_[x_interp, x_interp**2, x_interp**3]
y_interp_c = np.dot(theta_c.T, x_interp_c.T)
plt.plot(x_interp_c[:, 0], y_interp_c.T, alpha=0.7, marker='.');
```



Let's check the training error (or fitting error) again

```
err = np.power(y - np.dot(theta_c.T, xc.T), 2).mean()
```

Numerically it seems pretty high!

{{print(np.power(y - np.dot(theta_c.T, xc.T), 2).mean())}}

```
In [30]: errors.append(np.power(y - np.dot(theta_c.T, xc.T), 2).mean())
```

We can analyze what happens in function of m

We can have another dimensionality m instead of d by using **Basis Functions** $\phi(\mathbf{x})$.

With $\phi(\mathbf{x} = [1, \phi(x_1), \ldots, \phi(x_m)]$ and $\theta = [\theta_0, \theta_1, \ldots, \theta_m]$, we have:

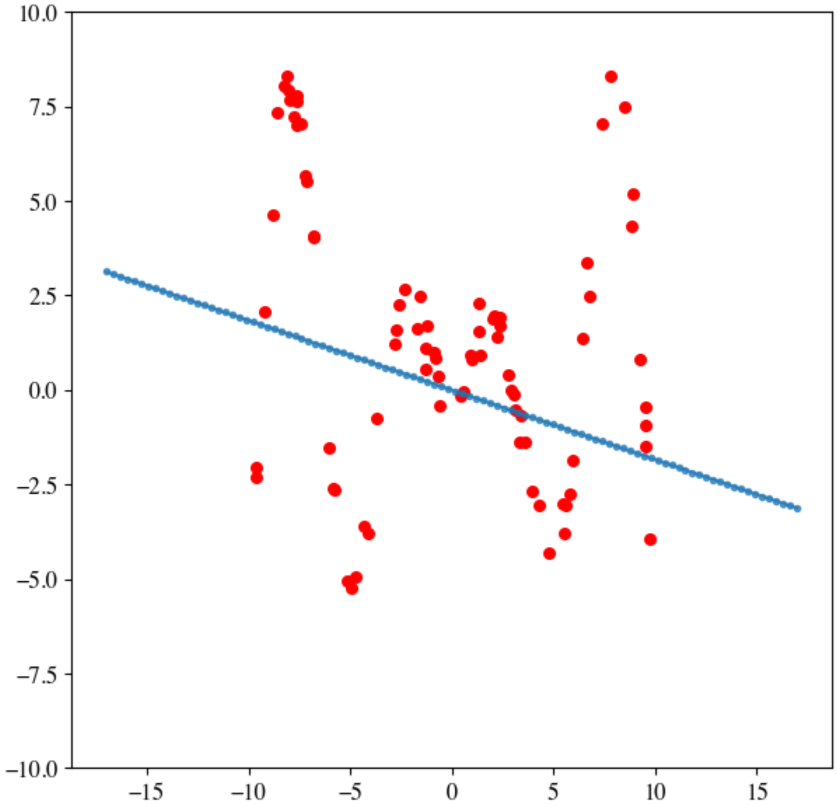
$$f_\theta(\mathbf{x}) = \sum_{i=0}^{m} \theta_i \cdot x_i = \theta^T \phi(\mathbf{x})$$

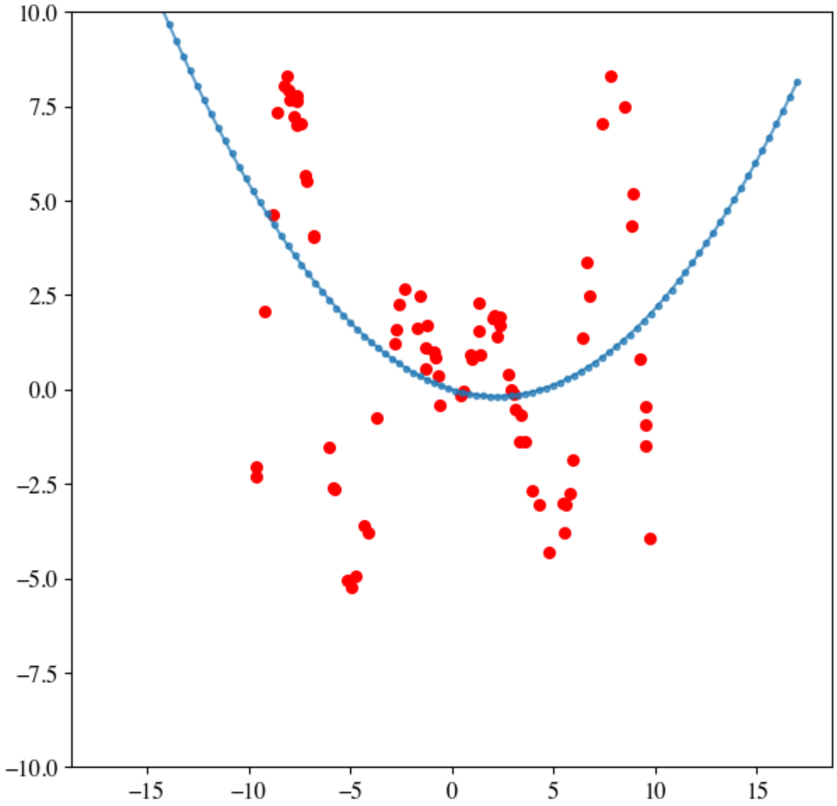
For Linear Regression:

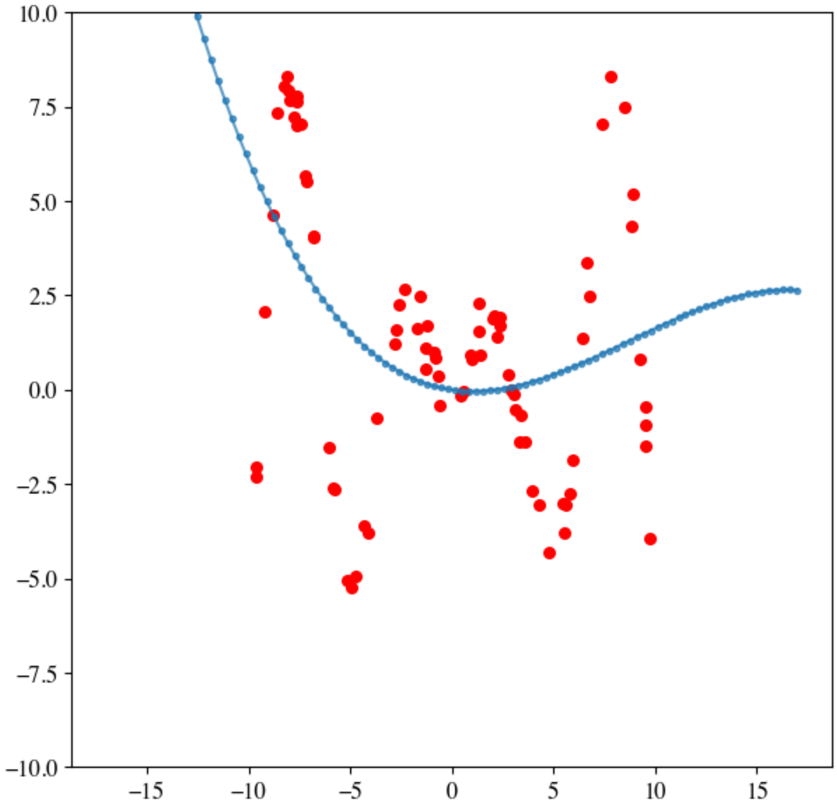
- $m > d$, and
- the **Basis Functions** is : $\phi_m(\mathbf{x}) = x^m$

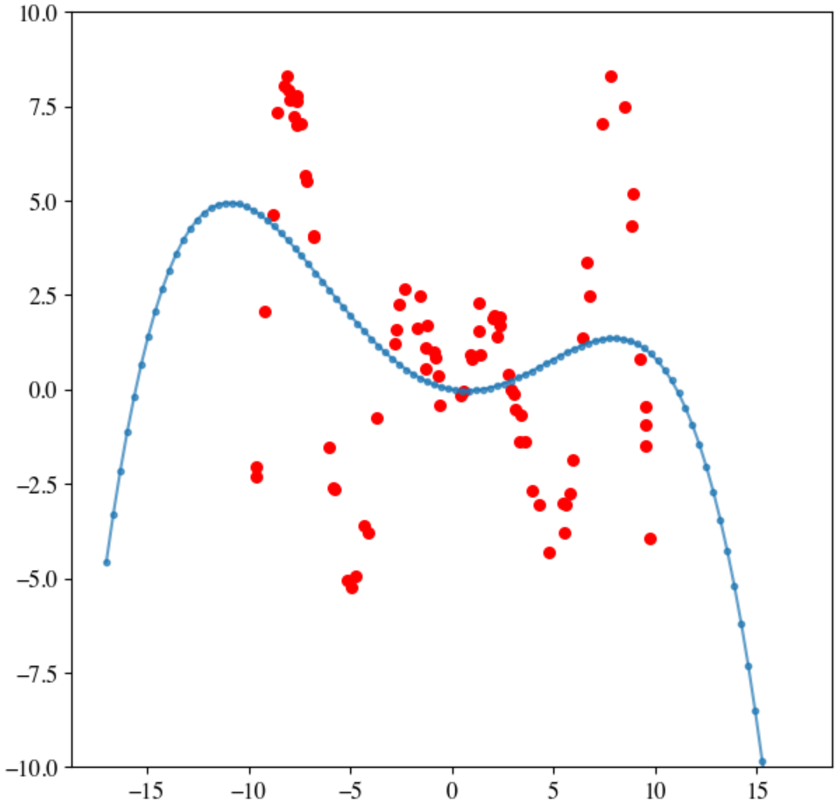
In [31]:

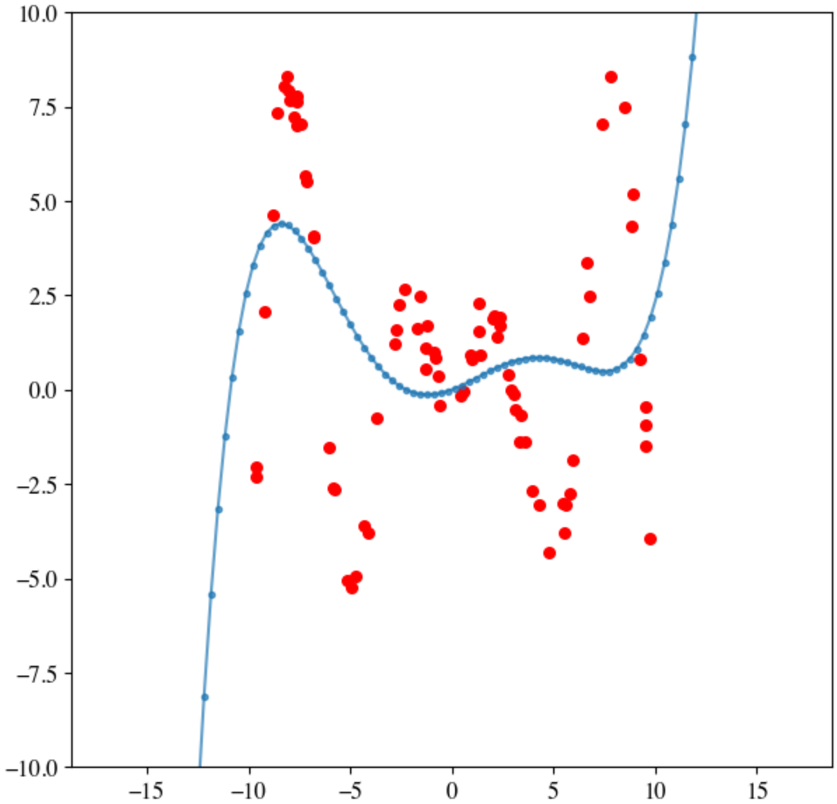
```
from sklearn.preprocessing import PolynomialFeatures
from sklearn.linear_model import LinearRegression
from sklearn.pipeline import Pipeline
import numpy as np

errors = []
errors_valid = []
models = []
for m in range(1, 20):
    model = Pipeline([('poly', PolynomialFeatures(degree=m, include_bias=False, interaction_only=False)),
                      ('linear', LinearRegression(fit_intercept=False))])
    models.append(model)
    model = model.fit(x, y)
    x_interp = np.linspace(-support_X-offset_valid,
                           support_X+offset_valid, 100).reshape(-1, 1)
    y_interp = model.predict(x_interp)
    y_est = model.predict(x)
    errors.append(np.power(y - y_est, 2).mean())
    errors_valid.append(np.power(y_valid - model.predict(x_valid), 2).mean())
    # Draw
    plt.figure(figsize=(7, 7))
    plt.scatter(x, y, c='red', marker='o', s=30)
    plt.plot(x_interp, y_interp, alpha=0.7, marker='.')
    plt.ylim([-10, 10])
```

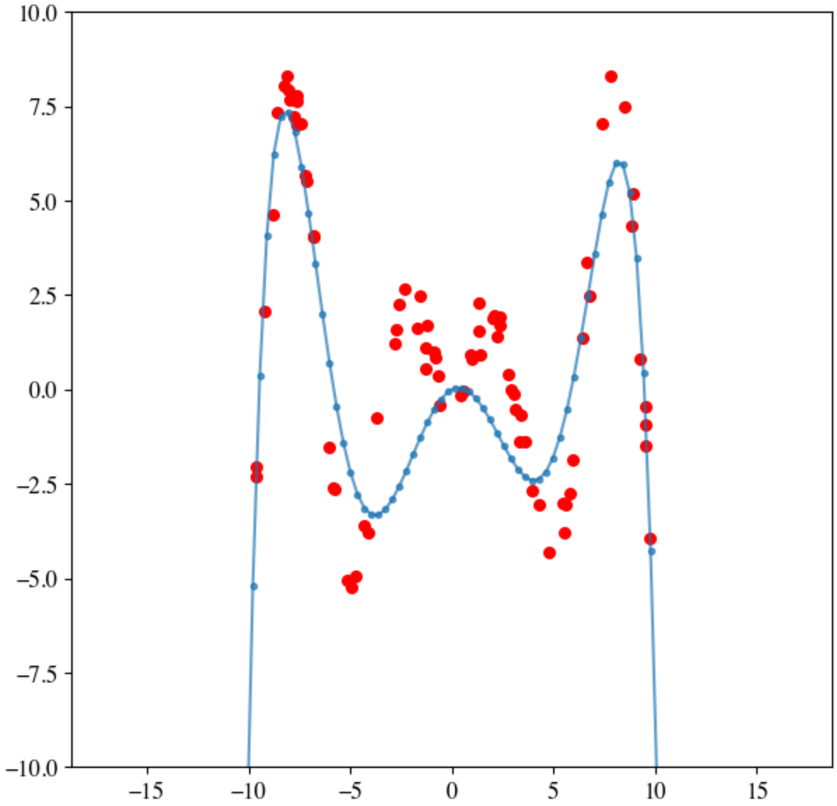


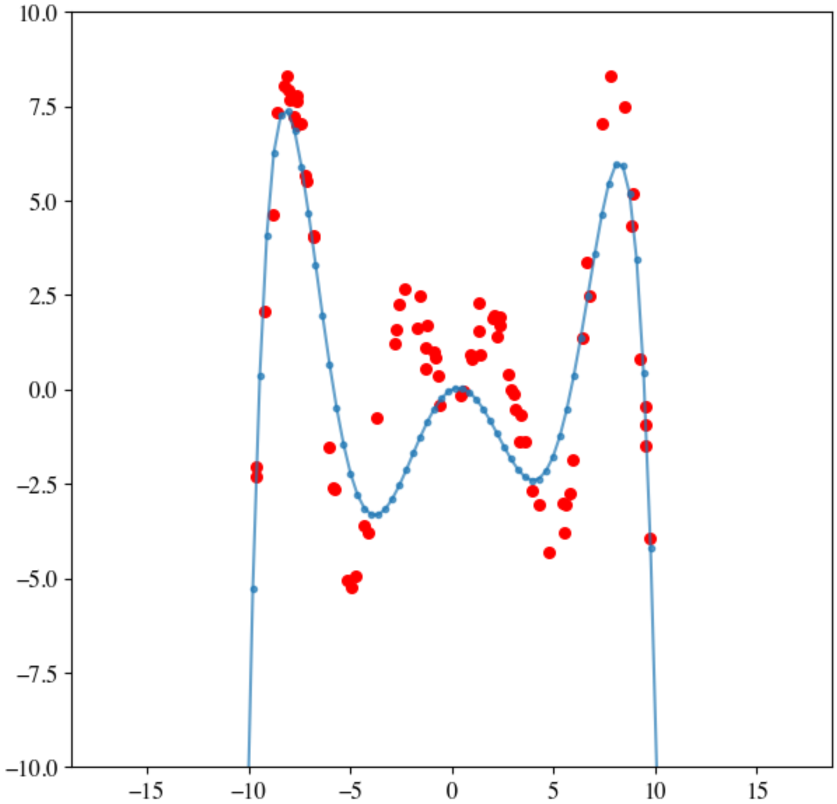

10.0
7.5
5.0
2.5
0.0
−2.5
−5.0
−7.5
−10.0
−15
−10
−5
0
5
10
15

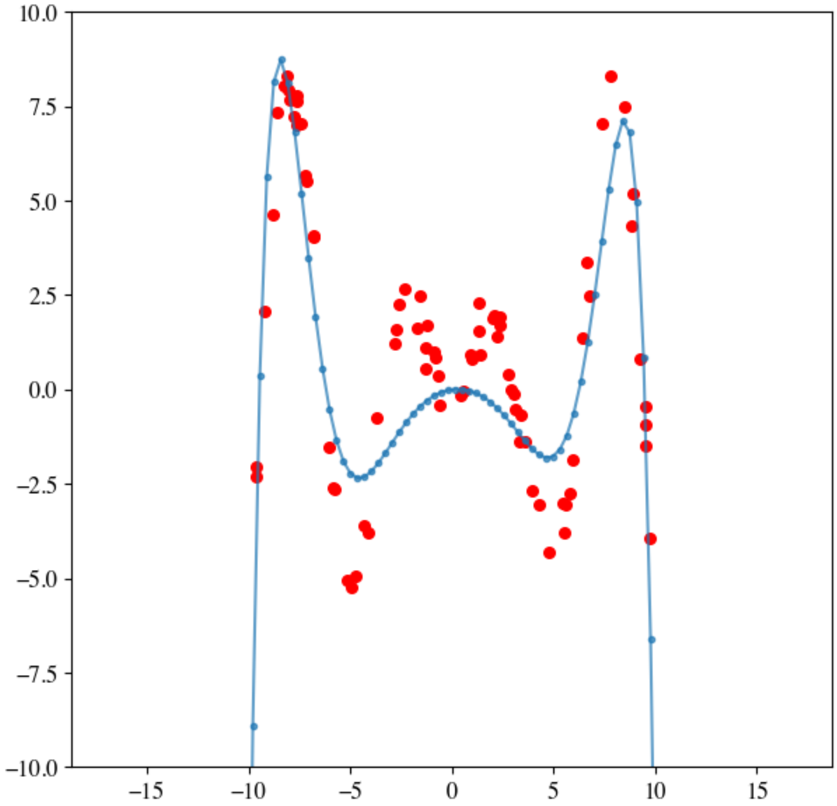

10.0
7.5
5.0
2.5
0.0
−2.5
−5.0
−7.5
−10.0
−15
−10
−5
0
5
10
15

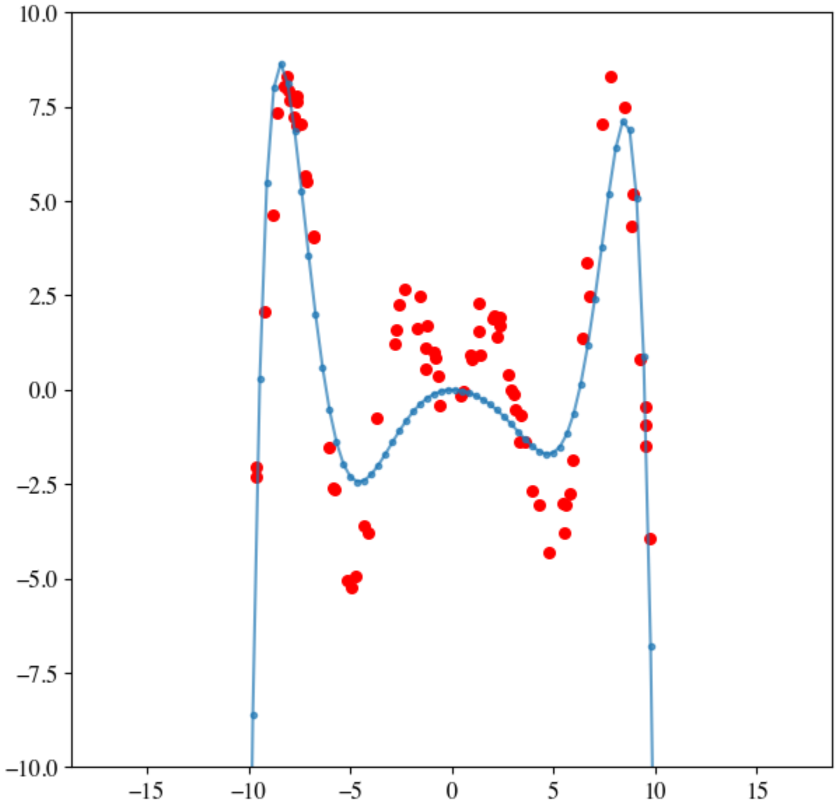

10.0
7.5
5.0
2.5
0.0
−2.5
−5.0
−7.5
−10.0
−15
−10
−5
0
5
10
15

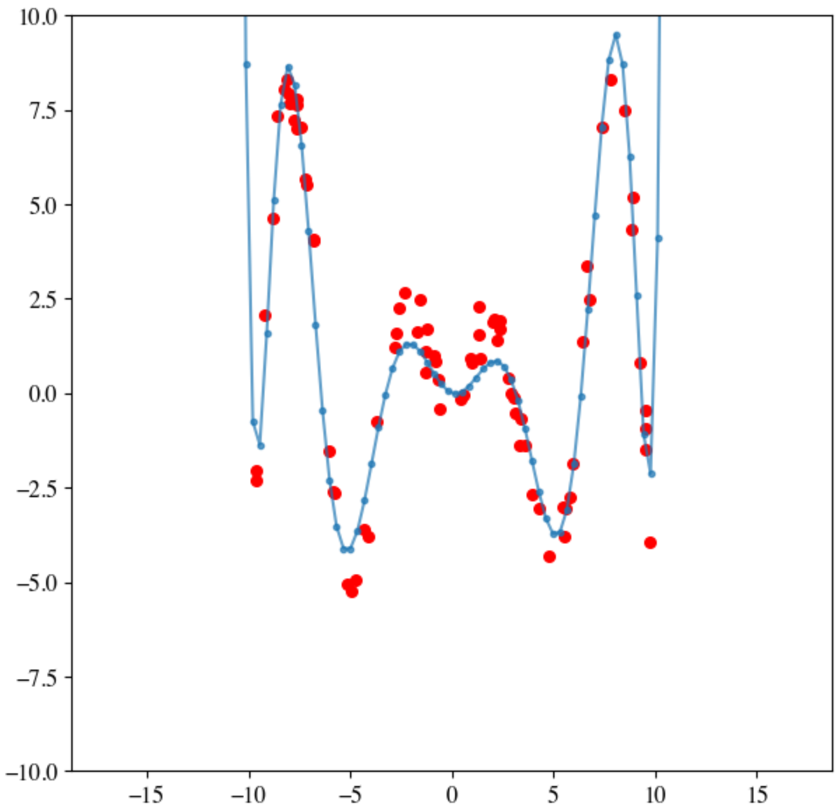

10.0
7.5
5.0
2.5
0.0
−2.5
−5.0
−7.5
−10.0
−15
−10
−5
0
5
10
15

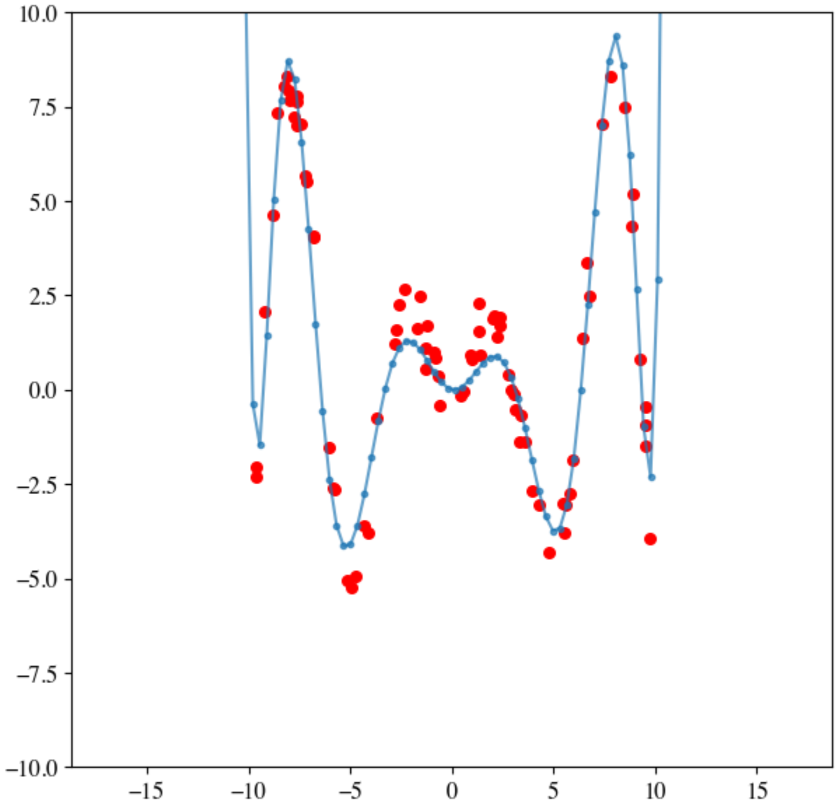

10.0
7.5
5.0
2.5
0.0
−2.5
−5.0
−7.5
−10.0
−15
−10
−5
0
5
10
15

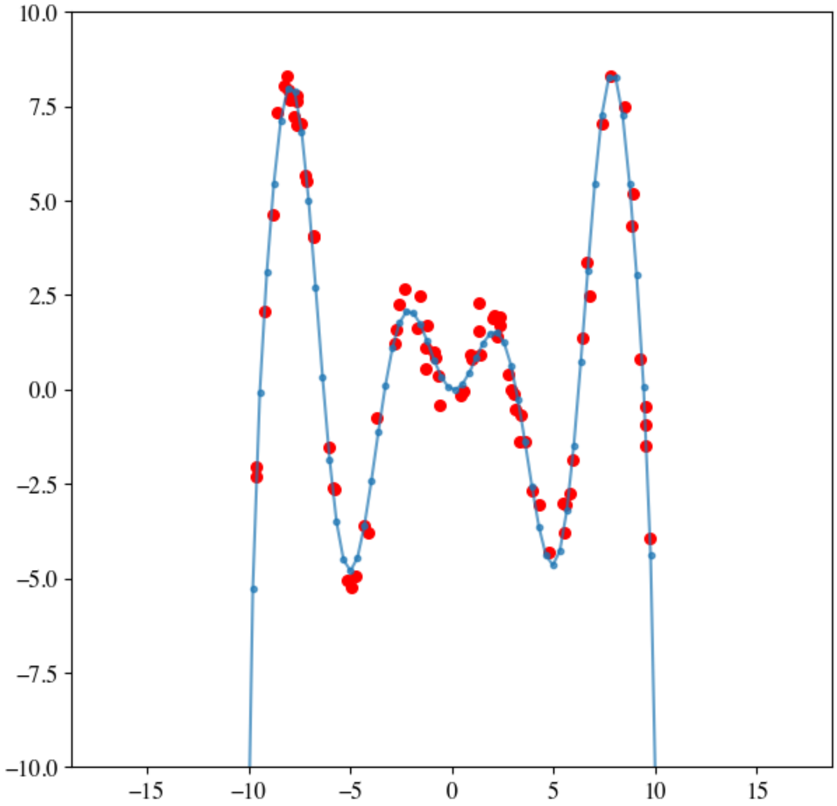

10.0
7.5
5.0
2.5
0.0
−2.5
−5.0
−7.5
−10.0
−15
−10
−5
0
5
10
15

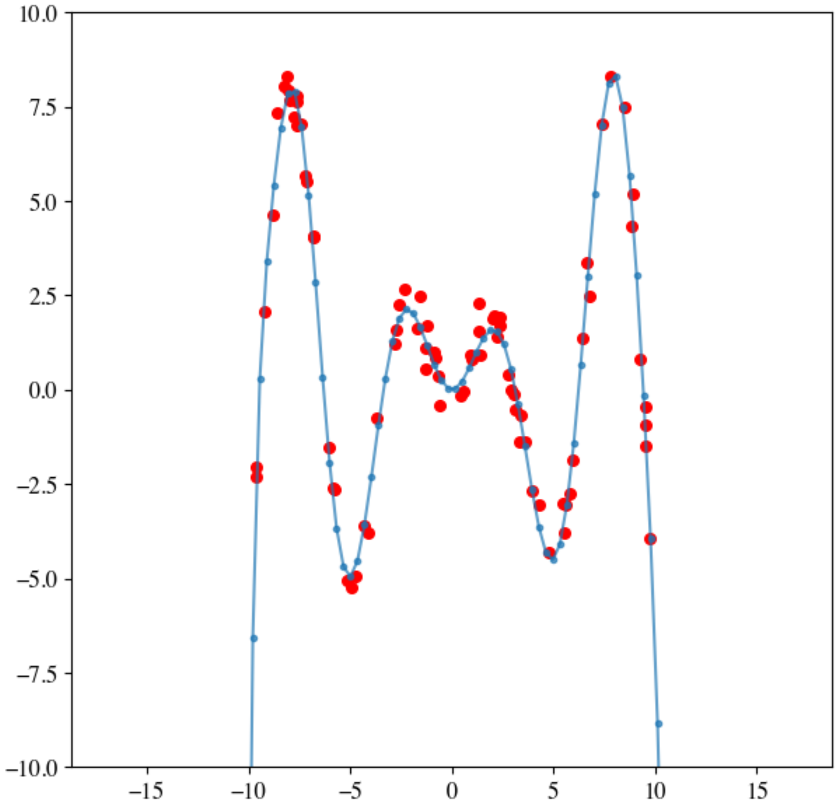

10.0
7.5
5.0
2.5
0.0
−2.5
−5.0
−7.5
−10.0
−15
−10
−5
0
5
10
15

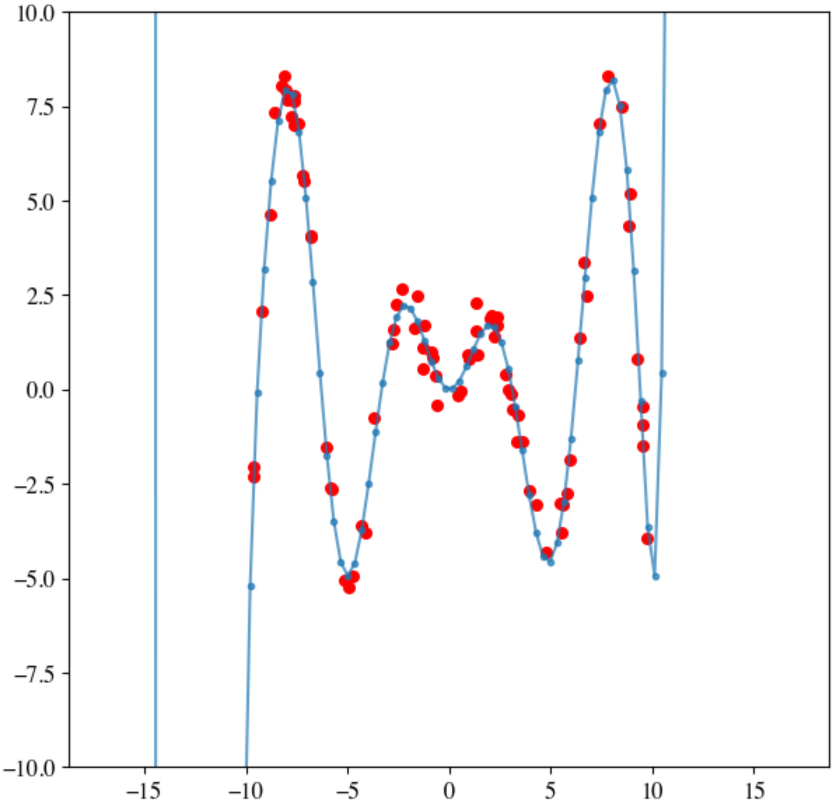

10.0
7.5
5.0
2.5
0.0
−2.5
−5.0
−7.5
−10.0
−15
−10
−5
0
5
10
15


10.0
7.5
5.0
2.5
0.0
−2.5
−5.0
−7.5
−10.0
−15
−10
−5
0
5
10
15

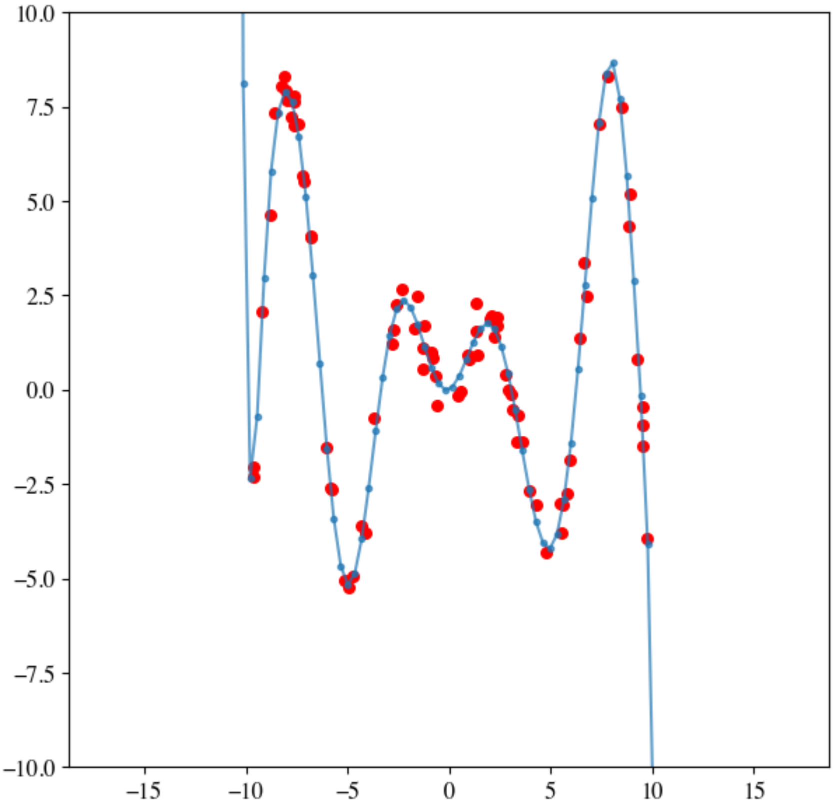

10.0
7.5
5.0
2.5
0.0
−2.5
−5.0
−7.5
−10.0
−15
−10
−5
0
5
10
15


10.0
7.5
5.0
2.5
0.0
−2.5
−5.0
−7.5
−10.0
−15
−10
−5
0
5
10
15

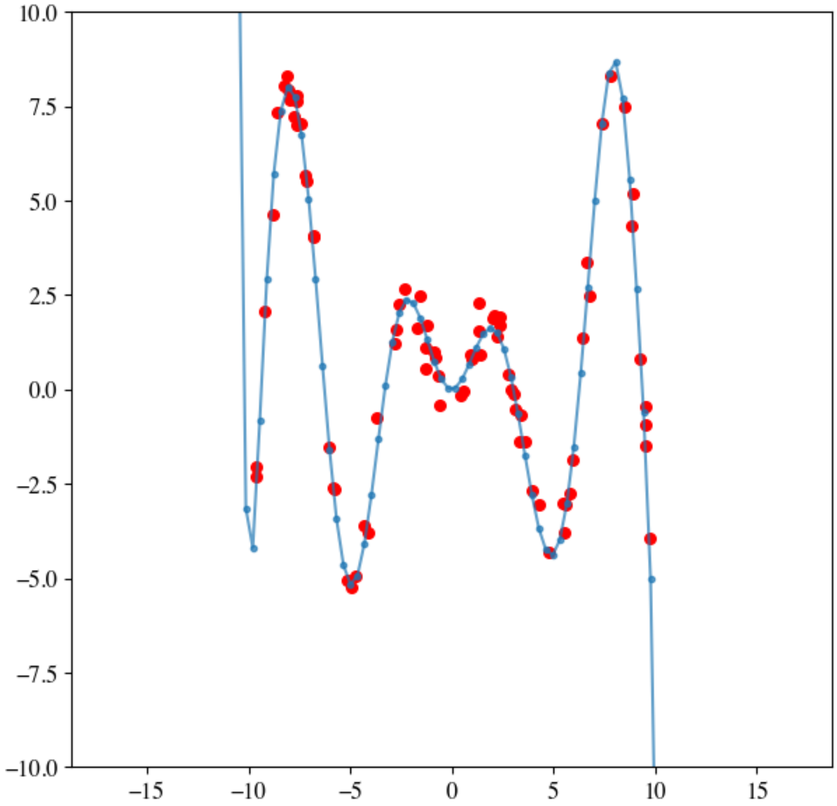

10.0
7.5
5.0
2.5
0.0
−2.5
−5.0
−7.5
−10.0
−15
−10
−5
0
5
10
15

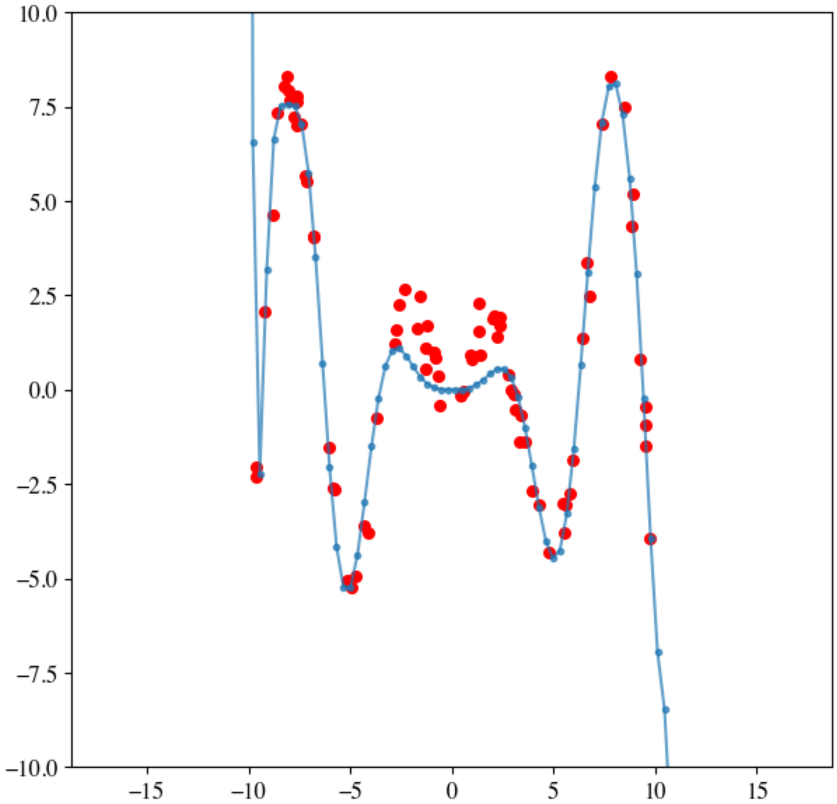


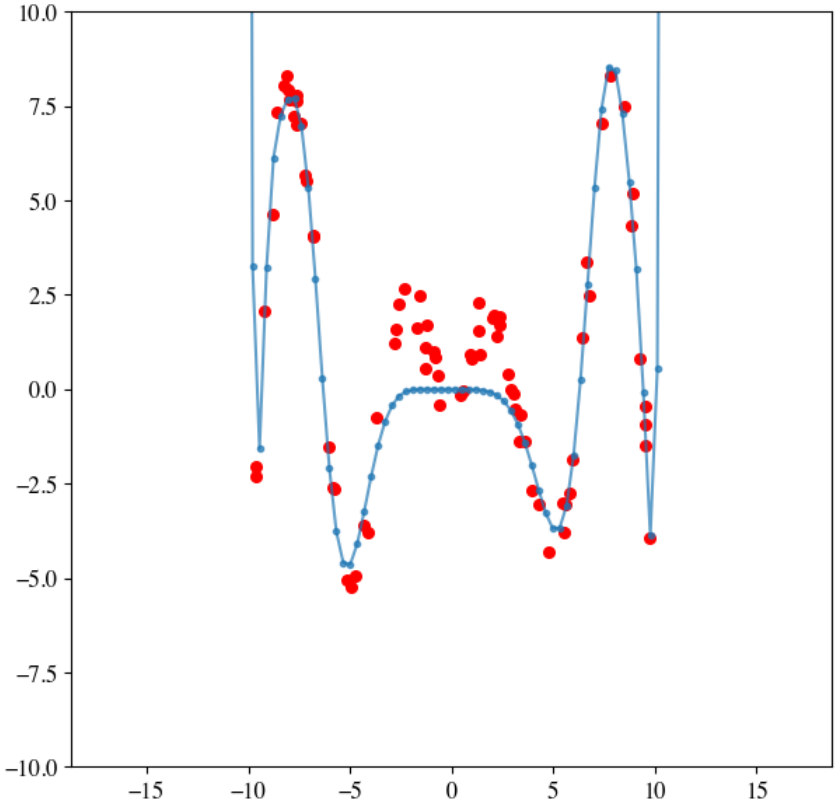
10.0
7.5
5.0
2.5
0.0
−2.5
−5.0
−7.5
−10.0
−15
−10
−5
0
5
10
15

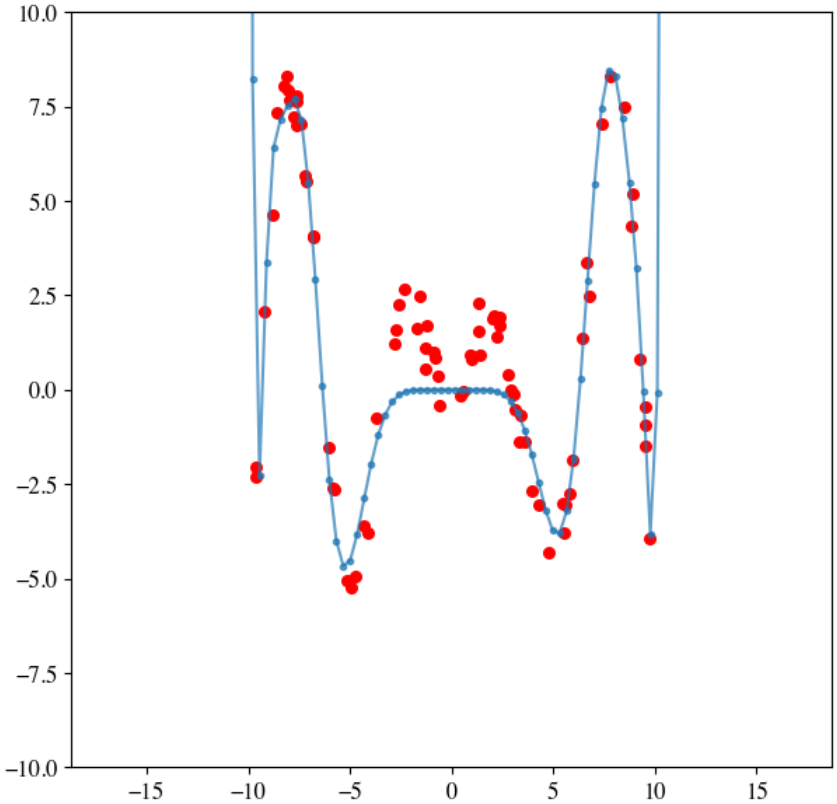


10.0
7.5
5.0
2.5
0.0
−2.5
−5.0
−7.5
−10.0
−15
−10
−5
0
5
10
15


10.0
7.5
5.0
2.5
0.0
−2.5
−5.0
−7.5
−10.0
−15
−10
−5
0
5
10
15


10.0
7.5
5.0
2.5
0.0
−2.5
−5.0
−7.5
−10.0
−15
−10
−5
0
5
10
15

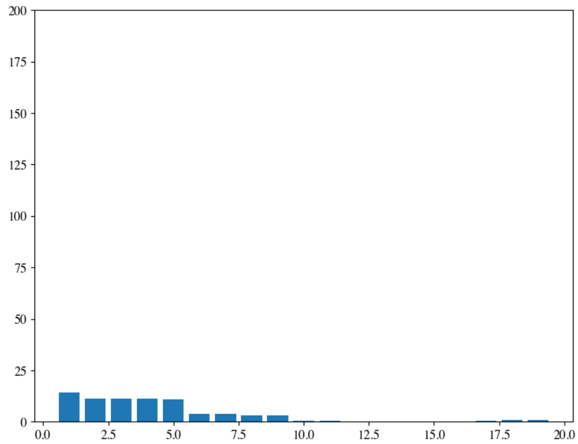


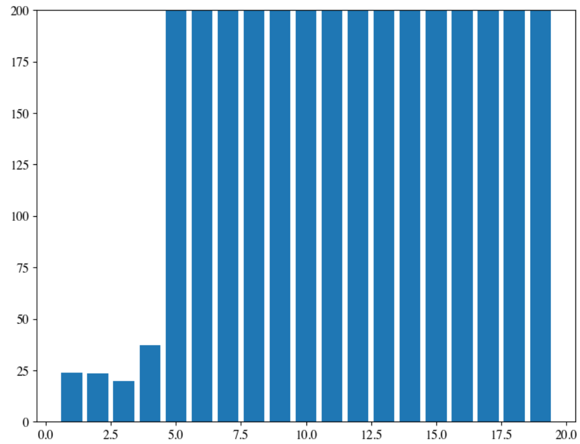


Now let's check both the errors in train and validation

In [32]:
```
fig, axes = plt.subplots(1,2, figsize=(20, 7))
axes[0].bar(range(1, 20), errors)
axes[1].bar(range(1, 20), errors_valid);
axes[1].set_ylim([0,200])
axes[0].set_ylim([0,200])
m_best = np.argmin(errors_valid)
print(f'M best (polynomial degree is) {m_best+1}')
```

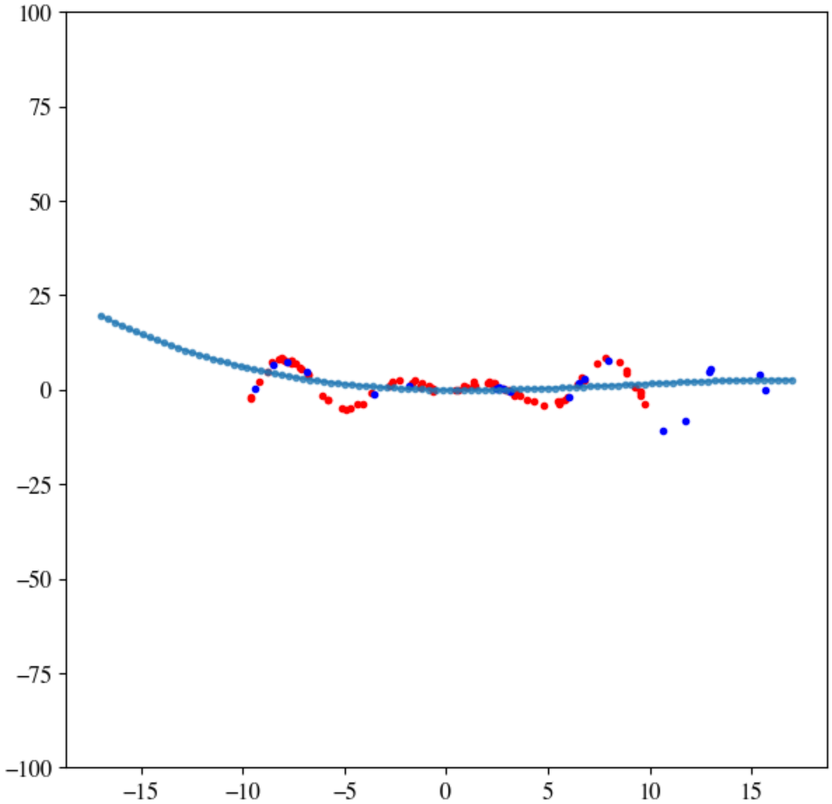
```
M best (polynomial degree is) 3
```





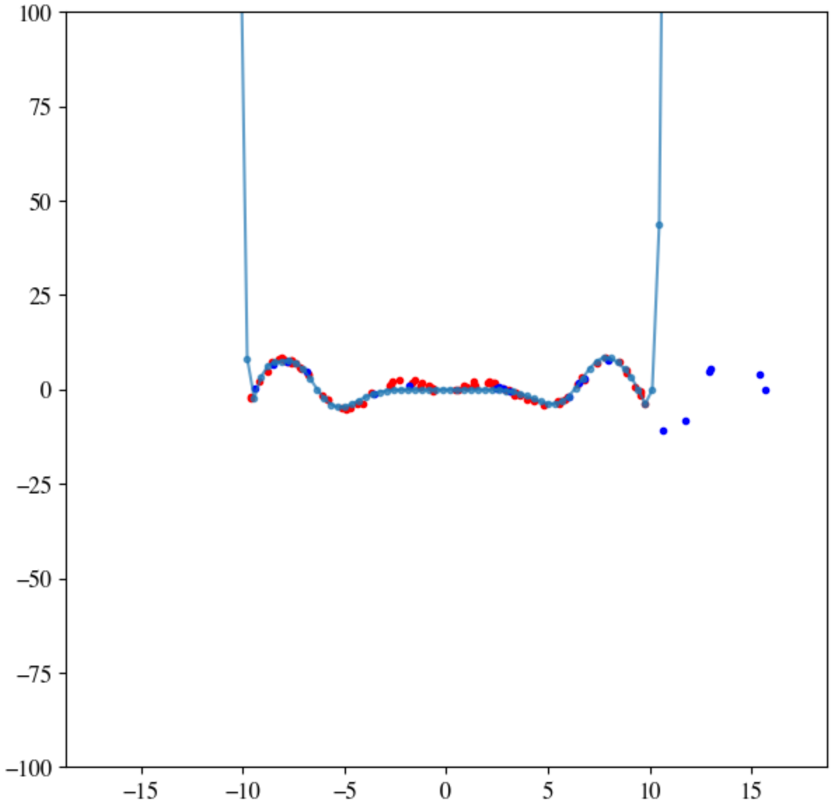
In [33]:

```
# Draw
plt.figure(figsize=(7, 7))
plt.scatter(x, y, c='red', marker='.')
plt.scatter(x_valid, y_valid, c='blue', marker='.')
y_interp = models[m_best].predict(x_interp)
plt.plot(x_interp, y_interp, alpha=0.7, marker='.');
plt.ylim([-100,100]);
```

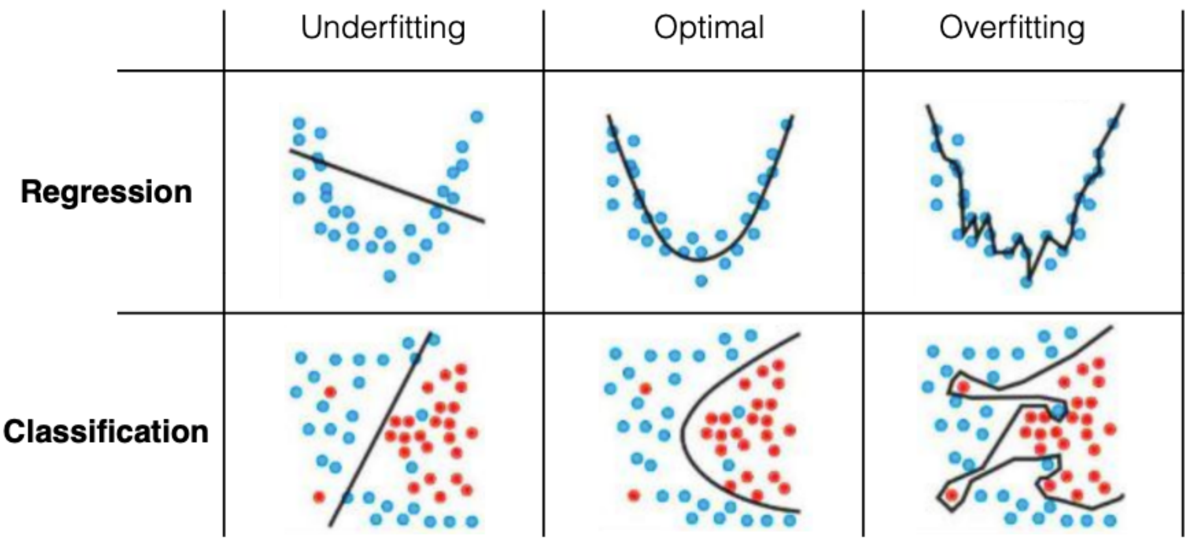


In [34]:

```
# Draw
plt.figure(figsize=(7, 7))
plt.scatter(x, y, c='red', marker='.')
plt.scatter(x_valid, y_valid, c='blue', marker='.')
y_interp = models[-1].predict(x_interp)
plt.plot(x_interp, y_interp, alpha=0.7, marker='.');
plt.ylim([-100,100]);
```



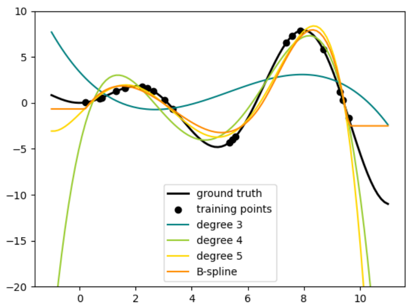
Over or Under Fitting



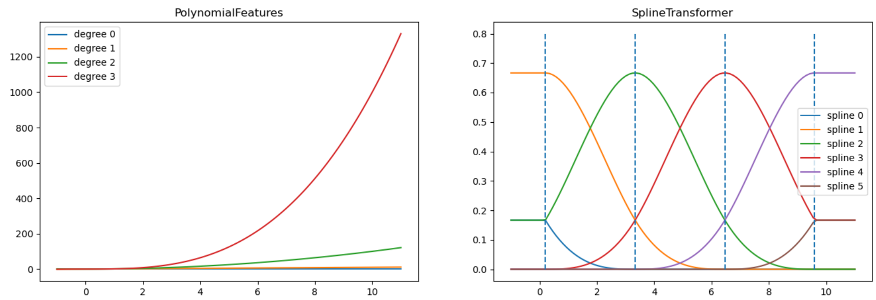
Problem of Polynomial Regression

The basis function $\phi_m(\mathbf{x}) = x^m$ is **global** wrt the domain of the feature x.

- The big limitation of polynomial basis functions is that they are **global functions of the input variable**, so that **changes in one region of input space affect all other regions.**
- This can be resolved by dividing the input space up into regions and fit a different polynomial in each region, leading to **spline functions** (that we do not cover).



Hint on Spline Functions



Other Basis Functions

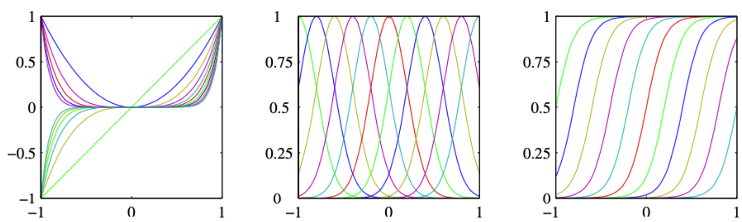


Figure 3.1 Examples of basis functions, showing polynomials on the left, Gaussians of the form (3.4) in the centre, and sigmoidal of the form (3.5) on the right.

Limitations of Basis Functions

Advantage: It is simple, and your problem stays convex and well behaved. (i.e. you can still use your original gradient descent code, just with the higher dimensional representation)

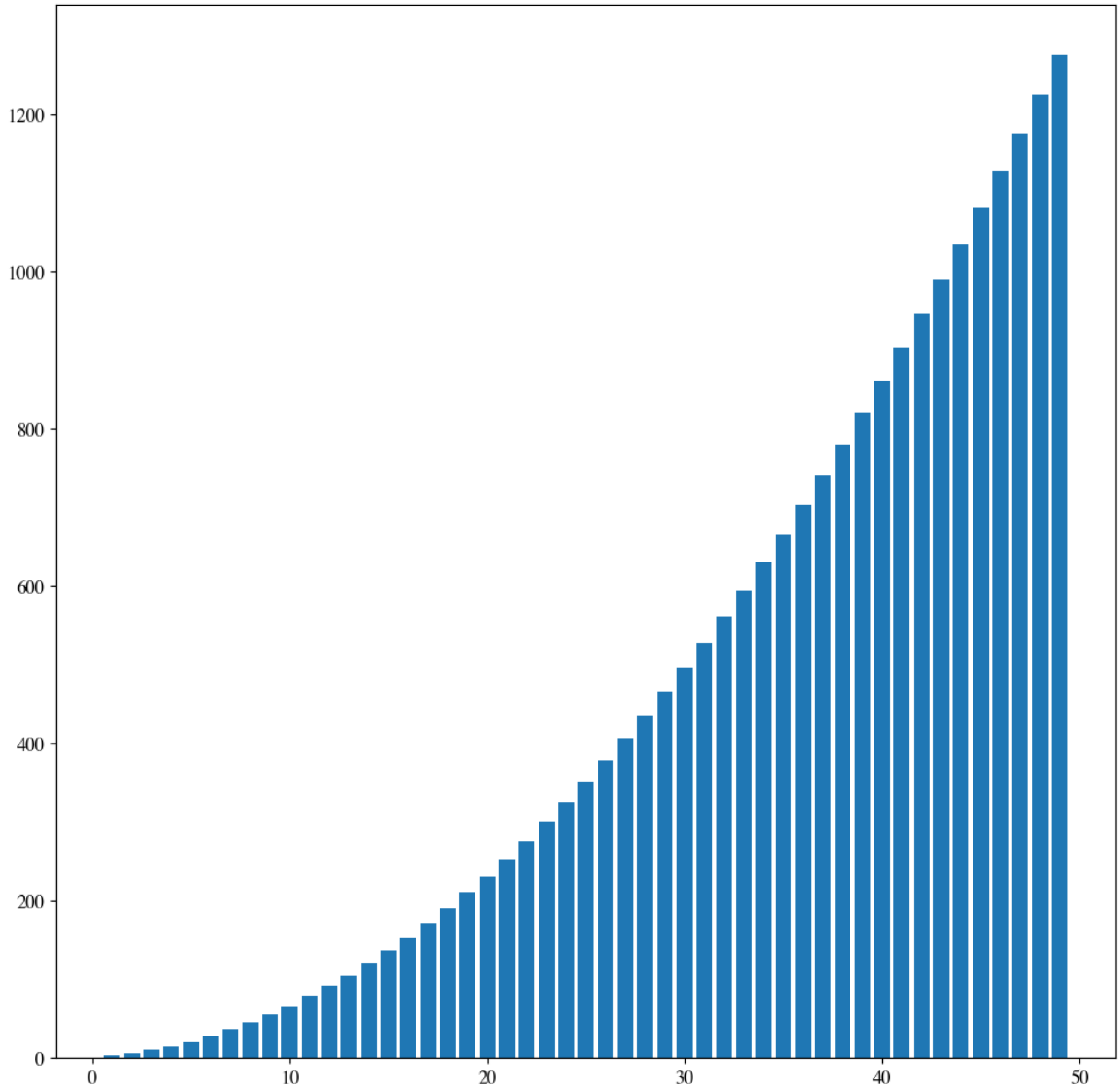
Disadvantage: $\phi(x)$ might be **Very** high dimensional.

```
X = PolynomialFeatures(interaction_only=True).fit_transform(X)
```

In [35]:
```
X_rand = np.array([[3, 7]], dtype=float)
print(f'Input Dimension {X_rand.shape}')
X_rand_poly = PolynomialFeatures(
    degree=2, interaction_only=False).fit_transform(X_rand)
print(f'Output Dimension {X_rand_poly.shape}')
print(X_rand[0, :], X_rand_poly[0, :], sep='\n')
```

```
Input Dimension (1, 2)
Output Dimension (1, 6)
[3. 7.]
[ 1.  3.  7.  9. 21. 49.]
```

In [36]:
```
%matplotlib inline
new_size = [PolynomialFeatures(degree=2, interaction_only=False).fit_transform(
    np.random.rand(1, d)).shape[1] for d in range(1, 50)]
plt.figure(figsize=(12,12))
plt.bar(range(1, 50), new_size);
```



Debug the Coefficients

1.1. Example: Polynomial Curve Fitting 7

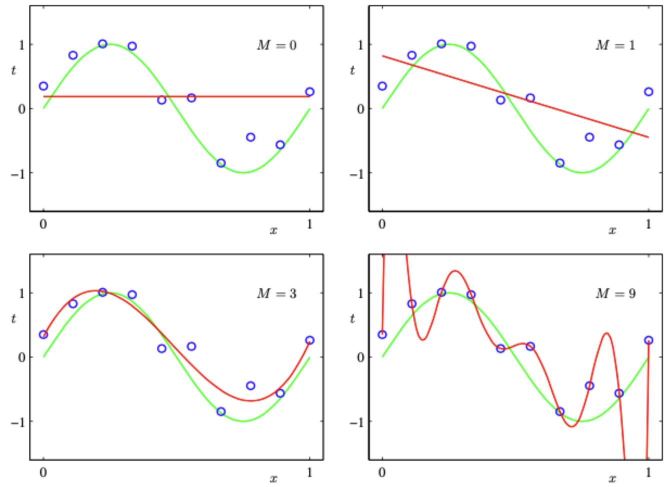


Figure 1.4 Plots of polynomials having various orders M, shown as red curves, fitted to the data set shown in Figure 1.2.

Debug the Coefficients $\longrightarrow$ Large Coefficients lead to overfit

Table 1.1 Table of the coefficients $\mathbf{w}^\star$ for polynomials of various order. Observe how the typical magnitude of the coefficients increases dramatically as the order of the polynomial increases.

	$M=0$	$M=1$	$M=6$	$M=9$
$w_0^\star$	0.19	0.82	0.31	0.35
$w_1^\star$		-1.27	7.99	232.37
$w_2^\star$			-25.43	-5321.83
$w_3^\star$			17.37	48568.31
$w_4^\star$				-231639.30
$w_5^\star$				640042.26
$w_6^\star$				-1061800.52
$w_7^\star$				1042400.18
$w_8^\star$				-557682.99
$w_9^\star$				125201.43

Remedy for Large Coefficients: *Weight Decay* (ℓ_2 Regularization)

We minimize the cost plus **penalty term**

$$\mathrm{J}(\theta; \mathbf{x}, y) = \frac{1}{2}\sum_{i=1}^{n} \mathrm{L}\big(y_i, f_\theta(\mathbf{x}_i)\big) + \frac{\lambda}{2}\boldsymbol{\theta}^T\boldsymbol{\theta} = \frac{1}{2}\sum_{i=1}^{n} \mathrm{L}\big(y_i, f_\theta(\mathbf{x}_i)\big) + \frac{\lambda}{2}\,||\boldsymbol{\theta}||_2^2$$

so to find:

$$\theta^\star = \arg\min_\theta \mathrm{J}_{\text{data}}(\theta; \mathbf{x}, y) + \lambda \mathrm{J}_{\text{reg.}}(\theta)$$

Still has a Closed Form Solution (Regularized Least Squares)

$$\boldsymbol{\theta} = (\lambda \mathbf{I} + \mathbf{X}^T\mathbf{X})^{-1}\mathbf{X}^T\mathbf{y}$$

Regularization allows complex models to be trained on data sets of limited size without severe over-fitting, essentially by limiting the effective model complexity

Linear Regression with *Weight Decay* (ℓ_2 Regularization)

Consider the eigendecomposition of the symmetric **Positive Semi Definite (PSD)** matrix $\mathbf{X}\mathbf{X}^T$:

$$X^T X = U \Sigma U^T = U \underbrace{\begin{bmatrix} \sigma_1^2 & \dots & 0 \\ \vdots & \ddots & \vdots \\ 0 & \dots & \sigma_d^2 \end{bmatrix}}_{\text{diag}\left(\sigma_1^2, \dots, \sigma_d^2\right)} U^T$$

$\sigma_1^2, \dots, \sigma_d^2$ are the eigenvalues and $UU^T = Id$ (since Σ is symmetric and square).

If $\mathbf{X}$ and $\mathbf{X}\mathbf{X}^T$ are not full rank then some of σ maybe zeros.

Linear Regression with *Weight Decay* (ℓ_2 Regularization)

This implies that if we regularized it, then the pseudo inverse is **always invertible and has a unique solution** since $\lambda > 0$:

$$X^T X + \lambda I = U \begin{bmatrix} \sigma_1^2 + \lambda & \dots & 0 \\ \vdots & \ddots & \vdots \\ 0 & \dots & \sigma_d^2 + \lambda \end{bmatrix} U^T$$

Linear Regression with *Weight Decay* (ℓ_2 Regularization)

This implies that if we regularized it then the pseudo inverse is **always invertible and has a unique solution** since $\lambda > 0$:

$$\left(X^T X + \lambda I\right)^{-1} = U \begin{bmatrix} \frac{1}{\sigma_1^2 + \lambda} & \cdots & 0 \\ \vdots & \ddots & \vdots \\ 0 & \cdots & \frac{1}{\sigma_d^2 + \lambda} \end{bmatrix} U^T$$

Bias-Variance for Linear Reg. with *Weight Decay* (ℓ_2 Regularization)

$$\begin{aligned} \hat{\theta}_n &= \left(X^T X + \lambda I\right)^{-1} X^T \vec{y} \\ &= \left(X^T X + \lambda I\right)^{-1} X^T \left(X\theta^* + \vec{\epsilon}\right) \\ &= \left[\left(X^T X + \lambda I\right)^{-1} X^T X\right] \theta^* + \left[\left(X^T X + \lambda I\right)^{-1} X^T\right] \vec{\epsilon} \end{aligned}$$

$$
\begin{aligned}
\mathrm{E}\left[\hat{\theta}_n\right] &= \mathrm{E}\left[\left[\left(X^TX+\lambda I\right)^{-1}X^TX\right]\theta^* + \left[\left(X^TX+\lambda I\right)^{-1}X^T\right]\vec{\epsilon}\right] \\
&= \left[\left(X^TX+\lambda I\right)^{-1}X^TX\right]\theta^* + \left[\left(X^TX+\lambda I\right)^{-1}X^T\right]\mathrm{E}[\vec{\epsilon}] \\
&= \left[\left(X^TX+\lambda I\right)^{-1}X^TX\right]\theta^* + \left[\left(X^TX+\lambda I\right)^{-1}X^T\right]\vec{0} \\
&= \left[\left(X^TX+\lambda I\right)^{-1}X^TX\right]\theta^*
\end{aligned}
$$

$$
= U\begin{bmatrix} \frac{1}{\sigma_1^2+\lambda} & \cdots & 0 \\ \vdots & \ddots & \vdots \\ 0 & \cdots & \frac{1}{\sigma_d^2+\lambda} \end{bmatrix} U^TX^TX\theta^*
$$

$$
= U\begin{bmatrix} \frac{1}{\sigma_1^2+\lambda} & \cdots & 0 \\ \vdots & \ddots & \vdots \\ 0 & \cdots & \frac{1}{\sigma_d^2+\lambda} \end{bmatrix} \underbrace{U^TU}_{Id} \begin{bmatrix} \sigma_1^2 & \dots & 0 \\ \vdots & \ddots & \vdots \\ 0 & \dots & \sigma_d^2 \end{bmatrix} U^T\theta^*
$$

$$
= U\begin{bmatrix} \frac{1}{\sigma_1^2+\lambda} & \cdots & 0 \\ \vdots & \ddots & \vdots \\ 0 & \dots & \frac{1}{\sigma_d^2+\lambda} \end{bmatrix} \begin{bmatrix} \sigma_1^2 & \dots & 0 \\ \vdots & \ddots & \vdots \\ 0 & \dots & \sigma_d^2 \end{bmatrix} U^T\theta^*
$$

$$
= U\begin{bmatrix} \frac{\sigma_1^2}{\sigma_1^2+\lambda} & \cdots & 0 \\ \vdots & \ddots & \vdots \\ 0 & \cdots & \frac{\sigma_d^2}{\sigma_d^2+\lambda} \end{bmatrix} U^T\theta^*.
$$

Bias-Variance for Linear Reg. with *Weight Decay* (ℓ_2 Regularization)

- From the above, we can make a few observations. First, when $\lambda = 0$, we see that $\mathrm{E}[\theta_n] = \theta$. This implies that **standard linear regression estimator (without regularization) is Unbiased.**
- The more regularization we add (i.e. larger λ), the smaller the eigenvalues will be, and hence the stronger the "shrinkage" towards 0. Thus it is biased towards zero.

Weight Decay arises from Bayesian Linear Regression

$$
\epsilon = y_i - \boldsymbol{\theta}^T\mathbf{x}_i \sim \mathrm{N}(0, \sigma^2)
$$

$$
\theta \sim \mathrm{N}(0, \boldsymbol{\Theta}^2) \quad \text{prior on weights}
$$

Unlike MLE, now there is a Gaussian prior on the weights; We solve it by doing Maximum A Posteriori (MAP)

Hint on Bayesian Linear Regression

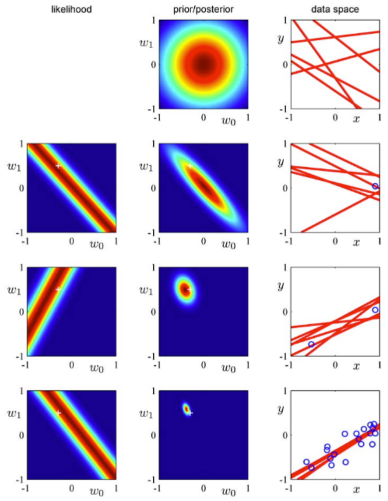


Figure 3.7 Illustration of sequential Bayesian learning for a simple linear model of the form $y(x, \mathbf{w}) = w_0 + w_1 x$. A detailed description of this figure is given in the text.