Lecture 3: Optimization and Linear Regression

Adapted from Applied Machine Learning Lecture Notes of Volodymyr Kuleshov, Cornel Tech

Instructor Tan Bui

Part 1: Optimization and Calculus Background

In the previous lecture, we learned what is a supervised machine learning problem.
Before we turn our attention to Linear Regression, we will first dive deeper into the question of optimization.

Review: Components of A Supervised Machine Learning Problem

At a high level, a supervised machine learning problem has the following structure:

$$\text{Dataset} + \underbrace{\text{Learning Algorithm}}\_\text{Model Class} + \text{Objective} + \text{Optimizer}$$
$$\rightarrow \text{Predictive Model}$$

The predictive model is chosen to model the relationship between inputs and targets. For instance, it can predict future targets.

Optimizer: Notation

At a high-level an optimizer takes

- an objective J (also called a loss function) and
- a model class $\mathcal{M}$ and finds a model $f \in \mathcal{M}$ with the smallest value of the objective J.

$$\min_{f \in \mathcal{M}} J(f)$$

Intuitively, this is the function that bests "fits" the data on the training dataset $\mathcal{D} = \{(x^{(i)}, y^{(i)}) \mid i = 1, 2, \ldots, n\}$.

We will use the a quadratic function as our running example for an objective J.

In [1]:

```
import numpy as np
import matplotlib.pyplot as plt
plt.rcParams['figure.figsize'] = [8, 4]
```

In [2]:

```
def quadratic_function(theta):
    """The cost function, J(theta)."""
    return 0.5*(2*theta-1)**2
```

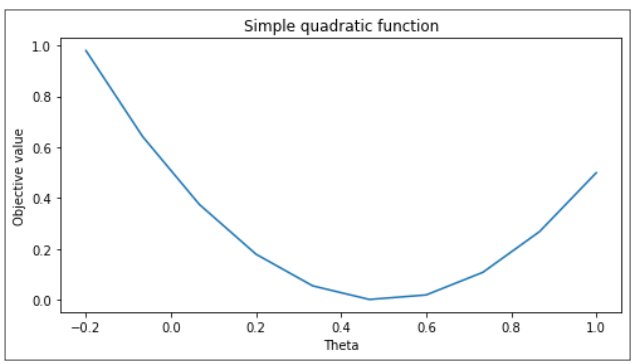
We can visualize it.

In [3]:

```
# First construct a grid of theta1 parameter pairs and their corresponding
# cost function values.
thetas = np.linspace(-0.2,1,10)
f_vals = quadratic_function(thetas[:,np.newaxis])

plt.plot(thetas, f_vals)
plt.xlabel('Theta')
plt.ylabel('Objective value')
plt.title('Simple quadratic function')
```

Out[3]:

```
Text(0.5, 1.0, 'Simple quadratic function')
```



Calculus Review: Derivatives

Recall that the derivative

$$\frac{df(\theta^*)}{d\theta}$$

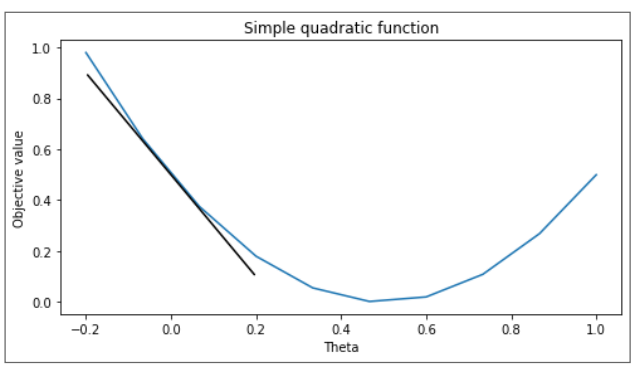
of a univariate function $f : \mathbb{R} \to \mathbb{R}$ is the instantaneous rate of change of the function $f(\theta)$ with respect to its parameter θ at the point θ^*.

In [4]:

```
def quadratic_derivative(theta):
    return (2*theta-1)*2

df0 = quadratic_derivative(np.array([[0]])) # derivative at zero
f0 = quadratic_function(np.array([[0]]))
line_length = 0.2

plt.plot(thetas, f_vals)
plt.annotate('', xytext=(0-line_length, f0-line_length*df0), xy=(0+line_length, f0+line_length*df0),
             arrowprops={'arrowstyle': '-', 'lw': 1.5}, va='center', ha='center')
plt.xlabel('Theta')
plt.ylabel('Objective value')
plt.title('Simple quadratic function')
```

Out[4]:

```
Text(0.5, 1.0, 'Simple quadratic function')
```

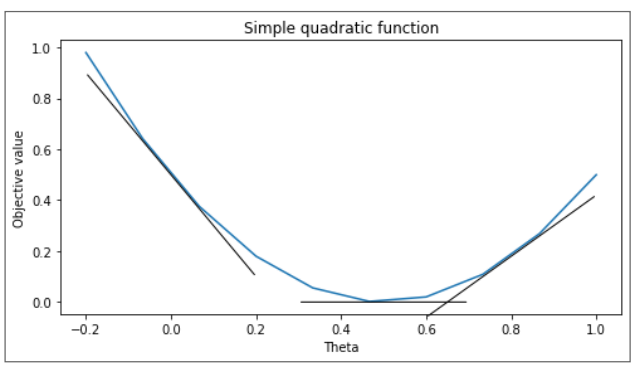


In [5]:

```
pts = np.array([[0, 0.5, 0.8]]).reshape((3,1))
df0s = quadratic_derivative(pts)
f0s = quadratic_function(pts)

plt.plot(thetas, f_vals)
for pt, f0, df0 in zip(pts.flatten(), f0s.flatten(), df0s.flatten()):
    plt.annotate('', xytext=(pt-line_length, f0-line_length*df0), xy=(pt+line_length, f0+line_length*df0),
             arrowprops={'arrowstyle': '-', 'lw': 1}, va='center', ha='center')
plt.xlabel('Theta')
plt.ylabel('Objective value')
plt.title('Simple quadratic function')
```

Out[5]:

```
Text(0.5, 1.0, 'Simple quadratic function')
```



Calculus Review: Partial Derivatives

The partial derivative

$$\frac{\partial f(\theta)}{\partial \theta_j}$$

of a multivariate function $f : \mathbb{R}^d \to \mathbb{R}$ is the derivative of f with respect to θ_j while all other inputs θ_k for $k \neq j$ are fixed.

Calculus Review: The Gradient

The gradient vector $\nabla_\theta f$ is the generalization of the derivative to multivariate functions $f : \mathbb{R}^d \to \mathbb{R}$, and is defined at an arbitrary parameter vector θ^* as

$$\nabla_\theta f(\theta^*) = \begin{bmatrix} \frac{\partial f(\theta^*)}{\partial \theta_1} \\ \frac{\partial f(\theta^*)}{\partial \theta_2} \\ \vdots \\ \frac{\partial f(\theta^*)}{\partial \theta_d} \end{bmatrix}.$$

The j-th entry of the vector $\nabla_\theta f(\theta^*)$ is the partial derivative $\frac{\partial f(\theta^*)}{\partial \theta_j}$ of f with respect to the j-th component of θ.

We will use a quadratic function as a running example.

In [6]:

```
def quadratic_function2d(theta0, theta1):
    """Quadratic objective function, J(theta0, theta1).

    The inputs theta0, theta1 are 2d arrays and we evaluate
    the objective at each value theta0[i,j], theta1[i,j].
    We implement it this way so it's easier to plot the
    level curves of the function in 2d.

    Parameters:
    theta0 (np.array): 2d array of first parameter theta0
    theta1 (np.array): 2d array of second parameter theta1

    Returns:
    fvals (np.array): 2d array of objective function values
        fvals is the same dimension as theta0 and theta1.
        fvals[i,j] is the value at theta0[i,j] and theta1[i,j].
    """
    theta0 = np.atleast_2d(np.asarray(theta0))
    theta1 = np.atleast_2d(np.asarray(theta1))
    return 0.5*((2*theta1-2)**2 + (theta0-3)**2)
```

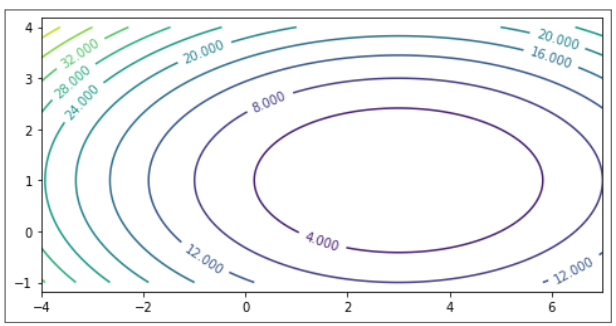
Let's visualize this function.

In [7]:

```
theta0_grid = np.linspace(-4,7,101)
theta1_grid = np.linspace(-1,4,101)
theta_grid = theta0_grid[np.newaxis,:], theta1_grid[:,np.newaxis]
J_grid = quadratic_function2d(theta0_grid[np.newaxis,:], theta1_grid[:,np.newaxis])

X, Y = np.meshgrid(theta0_grid, theta1_grid)
contours = plt.contour(X, Y, J_grid, 10)
plt.clabel(contours)
plt.axis('equal')
```

Out[7]:

```
(-4.0, 7.0, -1.0, 4.0)
```



Let's write down the derivative of the quadratic function.

In [8]:

```
def quadratic_derivative2d(theta0, theta1):
    """Derivative of quadratic objective function.

    The inputs theta0, theta1 are 1d arrays and we evaluate
    the derivative at each value theta0[i], theta1[i].

    Parameters:
    theta0 (np.array): 1d array of first parameter theta0
    theta1 (np.array): 1d array of second parameter theta1

    Returns:
    grads (np.array): 2d array of partial derivatives
        grads is of the same size as theta0 and theta1
        along first dimension and of size
        two along the second dimension.
        grads[i,j] is the j-th partial derivative
        at input theta0[i], theta1[i].
    """
    # this is the gradient of 0.5*((2*theta1-2)**2 + (theta0-3)**2)
    grads = np.stack([theta0-3, (2*theta1-2)*2], axis=1)
    grads = grads.reshape([len(theta0), 2])
    return grads
```

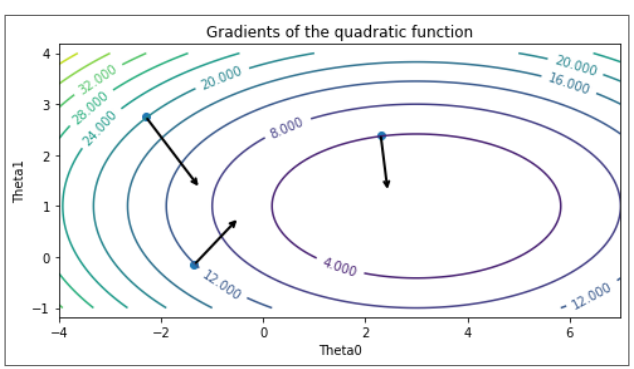
We can visualize the derivative.

In [9]:

```
theta0_pts, theta1_pts = np.array(
    [2.3, -1.35, -2.3]), np.array([2.4, -0.15, 2.75])
dfs = quadratic_derivative2d(theta0_pts, theta1_pts)
line_length = 0.2

contours = plt.contour(X, Y, J_grid, 10)
for theta0_pt, theta1_pt, df0 in zip(theta0_pts, theta1_pts, dfs):
    plt.annotate('', xytext=(theta0_pt, theta1_pt),
                 xy=(theta0_pt-line_length*df0[0],
                     theta1_pt-line_length*df0[1]),
                 arrowprops={'arrowstyle': '->', 'lw': 2}, va='center', ha='center')
plt.scatter(theta0_pts, theta1_pts)
plt.clabel(contours)
plt.xlabel('Theta0')
plt.ylabel('Theta1')
plt.title('Gradients of the quadratic function')
plt.axis('equal')
```

Out[9]:

```
(-4.0, 7.0, -1.0, 4.0)
```



Part 1b: Gradient Descent

Next, we will use gradients to define an important algorithm called *gradient descent*.

Calculus Review: The Gradient

The gradient vector $\nabla_\theta f$ is the generalization of the derivative to multivariate functions $f : \mathbb{R}^d \to \mathbb{R}$, and is defined at an arbitrary parameter vector θ^* as

$$\nabla_\theta f(\theta^*) = \begin{bmatrix} \frac{\partial f(\theta^*)}{\partial \theta_1} \\ \frac{\partial f(\theta^*)}{\partial \theta_2} \\ \vdots \\ \frac{\partial f(\theta^*)}{\partial \theta_d} \end{bmatrix}.$$

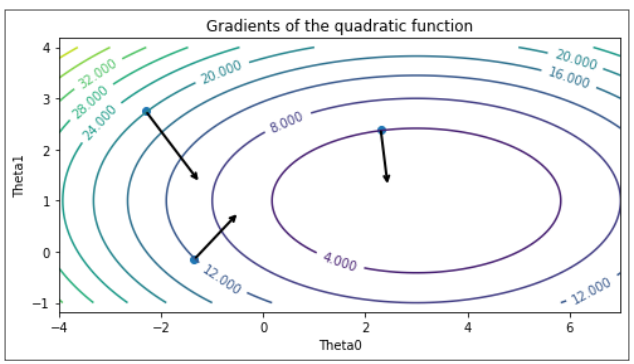
The j-th entry of the vector $\nabla_\theta f(\theta^*)$ is the partial derivative $\frac{\partial f(\theta^*)}{\partial \theta_j}$ of f with respect to the j-th component of θ.

In [10]:

```
theta0_pts, theta1_pts = np.array([2.3, -1.35, -2.3]), np.array([2.4, -0.15, 2.75])
dfs = quadratic_derivative2d(theta0_pts, theta1_pts)
line_length = 0.2

contours = plt.contour(X, Y, J_grid, 10)
for theta0_pt, theta1_pt, df0 in zip(theta0_pts, theta1_pts, dfs):
    plt.annotate('', xytext=(theta0_pt, theta1_pt),
                     xy=(theta0_pt-line_length*df0[0], theta1_pt-line_length*df0[1]),
                     arrowprops={'arrowstyle': '->', 'lw': 2}, va='center', ha='center')
plt.scatter(theta0_pts, theta1_pts)
plt.clabel(contours)
plt.xlabel('Theta0')
plt.ylabel('Theta1')
plt.title('Gradients of the quadratic function')
plt.axis('equal')
```

Out[10]:

```
(-4.0, 7.0, -1.0, 4.0)
```



Gradient Descent Method: Intuition

Gradient descent method is perhaps the most important optimization algorithm used in machine learning. The intuition behind gradient descent is the following. We know that the gradient pointing to the steepest ascent direction along which the function increases (locally). Thus to reach a minimum we simply move along the negative gradient direction. However, the gradient is a local information and thus a descent towards a minimum is guaranteed for a small step along the negative gradient direction. The gradient descent algorithm is therefore an iterative approach that continuously computes the gradient at the current point, follows the gradient descent a bit to find a new point, and then repeats the process until the gradient at the new point is sufficiently small.

Gradient Descent: Notation

More formally, if we want to optimize $J(\theta)$, we start with an initial guess θ^0 for the parameters and repeat the following update

$$\theta^i := \theta^{i-1} - \alpha \cdot \nabla_\theta J(\theta^{i-1}).$$

until θ^i and θ^{i-1} is not much different from each other or the gradient norm is sufficiently small.
As code, this method may look as follows:

```
theta, theta_prev = random_initialization()
while norm(theta - theta_prev) > convergence_threshold:
    theta_prev = theta
    theta = theta_prev - step_size * gradient(theta_prev)
```

In the above algorithm, we stop when $||\theta^i - \theta^{i-1}||$ or $||\nabla_\theta(\theta^i)||$ is small. Here the Euclidean norm $||\theta||$ is defined as

$$||\theta|| = (\sum_{i=0}^{d} \theta_i^2)^{1/2}.$$

Algorithm: Gradient descent

Inputs Tolerance τ, initial guess $\vec{\theta}^i, i = 0$
while $||\nabla_\theta(\theta^i)|| \geq \tau$ do

$$\theta^i := \theta^{i-1} - \alpha \cdot \nabla_\theta J(\theta^{i-1}).$$
$$i = i + 1$$

end $\vec{\theta}^* = \vec{\theta}^i$

Output The approximate minimizer $\vec{\theta}^*$

It's easy to implement this function in numpy.

In [11]:

```
convergence_threshold = 2e-1
step_size = 2e-1
theta, theta_prev = np.array([[-2], [3]]), np.array([[0], [0]])
opt_pts = [theta.flatten()]
opt_grads = []

while np.linalg.norm(theta - theta_prev) > convergence_threshold:
    # we repeat this while the value of the function is decreasing
    theta_prev = theta
    gradient = quadratic_derivative2d(*theta).reshape([2,1])
    theta = theta_prev - step_size * gradient
    opt_pts += [theta.flatten()]
    opt_grads += [gradient.flatten()]
```

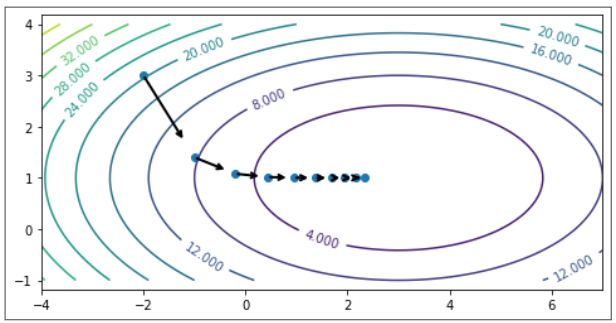
We can now visualize gradient descent.

In [12]:

```
opt_pts = np.array(opt_pts)
opt_grads = np.array(opt_grads)

contours = plt.contour(X, Y, J_grid, 10)
plt.clabel(contours)
plt.scatter(opt_pts[:,0], opt_pts[:,1])

for opt_pt, opt_grad in zip(opt_pts, opt_grads):
    plt.annotate('', xytext=(opt_pt[0], opt_pt[1]),
                 xy=(opt_pt[0]-0.8*step_size*opt_grad[0], opt_pt[1]-0.8*step_size*opt_grad[1]),
                 arrowprops={'arrowstyle': '->', 'lw': 2}, va='center', ha='center')

plt.axis('equal')
```

Out[12]:

```
(-4.0, 7.0, -1.0, 4.0)
```



Part 2: Gradient Descent in Linear Models

Let's now use gradient descent to derive a supervised learning algorithm for linear models.

Review: Linear Model Family

Recall that a linear model has the form

$$y = \theta_0 + \theta_1 \cdot x_1 + \theta_2 \cdot x_2 + \ldots + \theta_d \cdot x_d$$

where $x \in \mathbb{R}^d$ is a vector of features and y is the target. The θ_j are the *parameters* of the model.
By using the notation $x_0 = 1$, and thus the vector x as $x = (x_0, x_1, \ldots, x_d)^\top$ together with $\theta = (\theta_0, \theta_1, \ldots, \theta_d)^\top$ we can represent the model in a vectorized form

$$f_\theta(x) = \sum_{j=0}^{d} \theta_j x_j = \theta^\top x.$$

Let's define our model in Python.

In [13]:

```
def f(X, theta):
    """The linear model we are trying to fit.

    Parameters:
    theta (np.array): d-dimensional vector of parameters
    X (np.array): (n,d)-dimensional data matrix

    Returns:
    y_pred (np.array): n-dimensional vector of predicted targets
    """
    return X.dot(theta)
```

An Objective: Mean Squared Error

We pick θ to minimize the mean squared error (MSE). Slight variants of this objective are also known as the residual sum of squares (RSS) or the sum of squared residuals (SSR) or the L^2 loss.

$$J(\theta) = \frac{1}{2n} \sum_{i=1}^{n} (y^{(i)} - \theta^\top x^{(i)})^2$$

In other words, we are looking for the best compromise in θ over all the data points.

$$\Rightarrow \frac{\partial J(\theta)}{\partial \theta_j} = \frac{\partial}{\partial \theta_j} \frac{1}{2n} \sum_{i=1}^{n} (y^i - \theta^T \vec{x}^i)^2$$

$$= \frac{1}{2n} \sum_{i=1}^{n} \frac{\partial}{\partial \theta_j} (y^i - \theta^T \vec{x}^i)^2$$

Let's implement mean squared error.

In [14]:

```
def mean_squared_error(theta, X, y):
    """The cost function, J, describing the goodness of fit.

    Parameters:
    theta (np.array): d-dimensional vector of parameters
    X (np.array): (n,d)-dimensional design matrix
    y (np.array): n-dimensional vector of targets
    """
    return 0.5*np.mean((y-f(X, theta))**2)
```

Mean Squared Error: Partial Derivatives

Let's work out what a partial derivative is for the MSE error loss for a linear model with a generic data pair (x, y).

$$\begin{aligned}
\frac{\partial J(\theta)}{\partial \theta_j} &= \frac{\partial}{\partial \theta_j} \frac{1}{2} (f_\theta(x) - y)^2 \\
&= (f_\theta(x) - y) \cdot \frac{\partial}{\partial \theta_j} (f_\theta(x) - y) \\
&= (f_\theta(x) - y) \cdot \frac{\partial}{\partial \theta_j} \left(\sum_{k=0}^{d} \theta_k \cdot x_k - y \right) \\
&= (f_\theta(x) - y) \cdot x_j
\end{aligned}$$

$f_\theta(\vec{x}) = \theta^T \vec{x} = \sum_{k=0}^{d} x_k \theta_k$

Mean Squared Error: The Gradient

We can use this derivation to obtain an expression for the gradient of the MSE for a linear model

$$\nabla_\theta J(\theta) = \begin{bmatrix} \frac{\partial f(\theta)}{\partial \theta_0} \\ \frac{\partial f(\theta)}{\partial \theta_1} \\ \vdots \\ \frac{\partial f(\theta)}{\partial \theta_d} \end{bmatrix} = \begin{bmatrix} (f_\theta(x) - y) \cdot x_0 \\ (f_\theta(x) - y) \cdot x_1 \\ \vdots \\ (f_\theta(x) - y) \cdot x_d \end{bmatrix} = (f_\theta(\vec{x}) - y) \cdot \vec{x}.$$

Let's implement the gradient.

In [15]:

```
def mse_gradient(theta, X, y):
    """The gradient of the cost function.

    Parameters:
    theta (np.array): d-dimensional vector of parameters
    X (np.array): (n,d)-dimensional design matrix
    y (np.array): n-dimensional vector of targets

    Returns:
    grad (np.array): d-dimensional gradient of the MSE
    """
    return np.mean((f(X, theta) - y) * X.T, axis=1)
```

The UCI Diabetes Dataset

In this section, we are going to again use the UCI Diabetes Dataset.

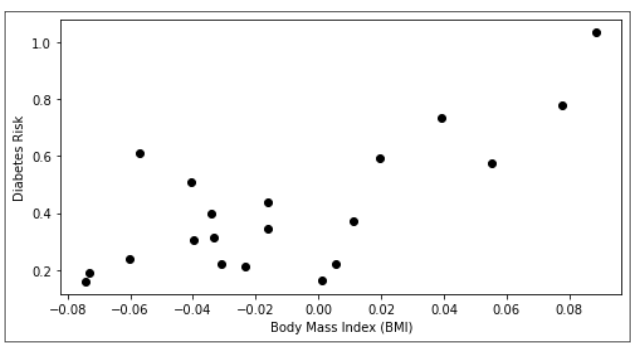
- For each patient we have a access to a measurement of their body mass index (BMI) and a quantiative diabetes risk score (from 0-300).
- We are interested in understanding how BMI affects an individual's diabetes risk.

In [16]:

```
%matplotlib inline
import matplotlib.pyplot as plt
plt.rcParams['figure.figsize'] = [8, 4]

import numpy as np
import pandas as pd
from sklearn import datasets

# Load the diabetes dataset
X, y = datasets.load_diabetes(return_X_y=True, as_frame=True)

# add an extra column of onens
X['one'] = 1

# Collect 20 data points and only use bmi dimension
X_train = X.iloc[-20:].loc[:, ['bmi', 'one']]
y_train = y.iloc[-20:] / 300

plt.scatter(X_train.loc[:,['bmi']], y_train,  color='black')
plt.xlabel('Body Mass Index (BMI)')
plt.ylabel('Diabetes Risk')
```

Out[16]:

```
Text(0, 0.5, 'Diabetes Risk')
```



Gradient Descent for Linear Regression

Putting this together with the gradient descent algorithm, we obtain a learning method for training linear models.

```
theta, theta_prev = random_initialization()
while abs(J(theta) - J(theta_prev)) > conv_threshold:
    theta_prev = theta
    theta = theta_prev - step_size * (f(x, theta)-y) * x
```

This update rule is also known as the Least Mean Squares (LMS) or Widrow-Hoff learning rule.

In [17]:

```
threshold = 1e-3
step_size = 4e-1
theta, theta_prev = np.array([2,1]), np.ones(2,)
opt_pts = [theta]
opt_grads = []
iter = 0

while np.linalg.norm(theta - theta_prev) > threshold:
    if iter % 100 == 0:
        print('Iteration %d. MSE: %.6f' % (iter, mean_squared_error(theta, X_train, y_train)))
    theta_prev = theta
    gradient = mse_gradient(theta, X_train, y_train)
    theta = theta_prev - step_size * gradient
    opt_pts += [theta]
    opt_grads += [gradient]
    iter += 1
```

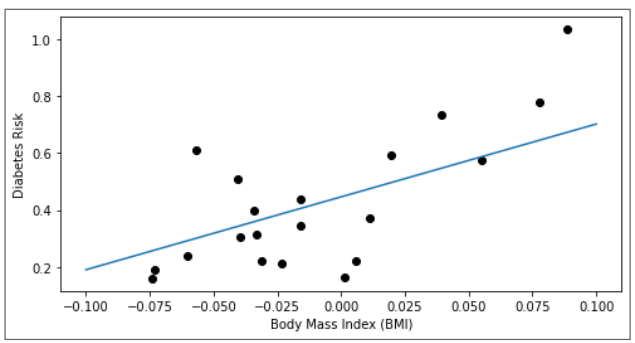
```
Iteration 0. MSE: 0.171729
Iteration 100. MSE: 0.014765
Iteration 200. MSE: 0.014349
Iteration 300. MSE: 0.013997
Iteration 400. MSE: 0.013701
```

In [18]:

```
x_line = np.stack([np.linspace(-0.1, 0.1, 10), np.ones(10,)])
y_line = opt_pts[-1].dot(x_line)

plt.scatter(X_train.loc[:,['bmi']], y_train,  color='black')
plt.plot(x_line[0], y_line)
plt.xlabel('Body Mass Index (BMI)')
plt.ylabel('Diabetes Risk')
```

Out[18]:

```
Text(0, 0.5, 'Diabetes Risk')
```



Part 3: Ordinary Least Squares

In practice, there is a more effective way than gradient descent to find linear model parameters.
We will see this method here, which will lead to our first non-toy algorithm: Ordinary Least Squares.

Review: The Gradient

Recall the gradient vector $\nabla_\theta f$ is the generalization of the derivative to multivariate functions $f : \mathbb{R}^d \to \mathbb{R}$, and is defined at an arbitrary parameter vector θ^* as

$$\nabla_\theta f(\theta^*) = \begin{bmatrix} \frac{\partial f(\theta^*)}{\partial \theta_1} \\ \frac{\partial f(\theta^*)}{\partial \theta_2} \\ \vdots \\ \frac{\partial f(\theta^*)}{\partial \theta_d} \end{bmatrix}.$$

The j-th entry of the vector $\nabla_\theta f(\theta^*)$ is the partial derivative $\frac{\partial f(\theta^*)}{\partial \theta_j}$ of f with respect to the j-th component of θ.

The UCI Diabetes Dataset

In this section, we are going to again use the UCI Diabetes Dataset.

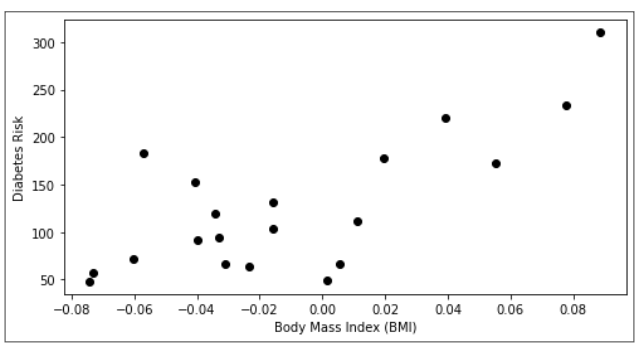
- For each patient we have a access to a measurement of their body mass index (BMI) and a quantiative diabetes risk score (from 0-300).
- We are interested in understanding how BMI affects an individual's diabetes risk.

In [19]:

```
%matplotlib inline
import matplotlib.pyplot as plt
plt.rcParams['figure.figsize'] = [8, 4]

import numpy as np
import pandas as pd
from sklearn import datasets

# Load the diabetes dataset
X, y = datasets.load_diabetes(return_X_y=True, as_frame=True)

# add an extra column of onens
X['one'] = 1

# Collect 20 data points
X_train = X.iloc[-20:]
y_train = y.iloc[-20:]

plt.scatter(X_train.loc[:,['bmi']], y_train,  color='black')
plt.xlabel('Body Mass Index (BMI)')
plt.ylabel('Diabetes Risk')
```

Out[19]:

```
Text(0, 0.5, 'Diabetes Risk')
```



Notation: Design Matrix

Machine learning algorithms are most easily defined in the language of linear algebra. Therefore, it will be useful to represent the entire dataset as one matrix $X \in \mathbb{R}^{n \times d}$, of the form:

$$X = \begin{bmatrix} x_1^{(1)} & x_2^{(1)} & \dots & x_d^{(1)} \\ x_1^{(2)} & x_2^{(2)} & \dots & x_d^{(2)} \\ \vdots & & & \\ x_1^{(n)} & x_2^{(n)} & \dots & x_d^{(n)} \end{bmatrix} = \begin{bmatrix} - & (\vec{x}^{(1)})^\top & - \\ - & (\vec{x}^{(2)})^\top & - \\ & \vdots & \\ - & (\vec{x}^{(n)})^\top & - \end{bmatrix}.$$

We can view the design matrix for the diabetes dataset.

In [20]:

```
X_train.head()
```

Out[20]:

	age	sex	bmi	bp	s1	s2	s3	s4	s5
422	-0.078165	0.050680	0.077863	0.052858	0.078236	0.064447	0.026550	-0.002592	0.040672
423	0.009016	0.050680	-0.039618	0.028758	0.038334	0.073529	-0.072854	0.108111	0.015567
424	0.001751	0.050680	0.011039	-0.019442	-0.016704	-0.003819	-0.047082	0.034309	0.024053
425	-0.078165	-0.044642	-0.040696	-0.081414	-0.100638	-0.112795	0.022869	-0.076395	-0.020289
426	0.030811	0.050680	-0.034229	0.043677	0.057597	0.068831	-0.032356	0.057557	0.035462

Notation: Design Matrix

Similarly, we can vectorize the target variables into a vector $y \in \mathbb{R}^n$ of the form

recall the training data

$\{\vec{x}^i, y^i\}$

$$\vec{y} = \begin{bmatrix} y^{(1)} \\ y^{(2)} \\ \vdots \\ y^{(n)} \end{bmatrix}.$$

$$X = \begin{bmatrix} - & \vec{x}^{1\,T} & - \\ & \vdots & \\ - & x^{n\,T} & - \end{bmatrix}$$

Squared Error in Matrix Form

Recall that we may fit a linear model by choosing θ that minimizes the squared error:

$$J(\theta) = \frac{1}{2}\sum_{i=1}^{n}(y^{(i)} - \vec{\theta}^{\top}\vec{x}^{(i)})^2$$

In other words, we are looking for the best compromise in θ over all the data points.

We can write this sum in matrix-vector form as:

$$J(\theta) = \frac{1}{2}(\vec{y} - X\vec{\theta})^{\top}(\vec{y} - X\vec{\theta}) = \frac{1}{2}\|\vec{y} - X\vec{\theta}\|^2,$$

where X is the design matrix and $\|\cdot\|$ denotes the Euclidean norm, again, defined as

$$||\vec{y}|| = (\sum_{i=1}^{n} y_i^2)^{1/2}.$$

The Gradient of the Squared Error

We can a gradient for the mean squared error as follows.

$$
\begin{aligned}
\nabla_\theta J(\theta) &= \nabla_\theta \frac{1}{2}(X\theta - y)^\top (X\theta - y) \\
&= \frac{1}{2}\nabla_\theta \left((X\theta)^\top (X\theta) - (X\theta)^\top y - y^\top (X\theta) + y^\top y\right) \\
&= \frac{1}{2}\nabla_\theta \left(\theta^\top (X^\top X)\theta - 2(X\theta)^\top y\right) \\
&= \frac{1}{2}\left(2(X^\top X)\theta - 2X^\top y\right) \\
&= (X^\top X)\theta - X^\top y
\end{aligned}
$$

We used the facts that $a^\top b = b^\top a$ (line 3), that $\nabla_x b^\top x = b$ (line 4), and that $\nabla_x x^\top A x = 2Ax$ for a symmetric matrix A (line 4).

Normal Equations

Setting the above derivative to zero, we obtain the *normal equations*:

$$\nabla_{\vec{\theta}} J(\vec{\theta}) = X^T X \vec{\theta} - X^T \vec{y} = 0 \rightarrow (X^\top X)\vec{\theta} = X^\top y.$$

Hence, the value θ^* that minimizes this objective is given by:

$$\vec{\theta^*} = (X^\top X)^{-1} X^\top y.$$

Note that we assumed that the matrix $(X^\top X)$ is invertible; if this is not the case, there are easy ways of addressing this issue.

because $\nabla^2_{\vec{\theta}} J(\vec{\theta}) \stackrel{??}{=} X^T X = H$

H is symmetric

HW } H is positive definite if H is invertible

$X^T X$ is invertible : X has full column rank

Let's apply the normal equations.

In [21]:

```
import numpy as np

theta_best = np.linalg.inv(X_train.T.dot(X_train)).dot(X_train.T).dot(y_train)
theta_best_df = pd.DataFrame(data=theta_best[np.newaxis, :], columns=X.columns)
theta_best_df
```

Out[21]:

	age	sex	bmi	bp	s1	s2	s3	
0	-3.888868	204.648785	-64.289163	-262.796691	14003.726808	-11798.307781	-5892.15807	-1136.9476

We can now use our estimate of theta to compute predictions for 3 new data points.

In [22]:

```
# Collect 3 data points for testing
X_test = X.iloc[:3]
y_test = y.iloc[:3]

# generate predictions on the new patients
y_test_pred = X_test.dot(theta_best)
```

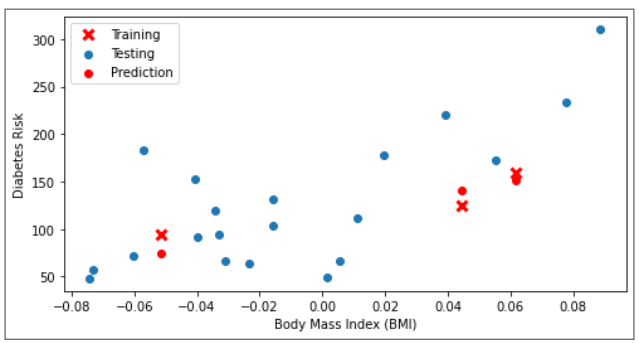
Let's visualize these predictions.

In [23]:

```
# visualize the results
plt.xlabel('Body Mass Index (BMI)')
plt.ylabel('Diabetes Risk')
plt.scatter(X_train.loc[:, ['bmi']], y_train)
plt.scatter(X_test.loc[:, ['bmi']], y_test, color='red', marker='o')
plt.plot(X_test.loc[:, ['bmi']], y_test_pred, 'x', color='red', mew=3, markersize=8)
plt.legend(['Training', 'Testing','Prediction'])
```

Out[23]:

```
<matplotlib.legend.Legend at 0x7f44fd9d3278>
```



Algorithm: Ordinary Least Squares

- **Type**: Supervised learning (regression)
- **Model family**: Linear models
- **Objective function**: Mean squared error
- **Optimizer**: Normal equations

Part 4: Non-Linear Least Squares

So far, we have learned about a very simple linear model. These can capture only simple linear relationships in the data. How can we use what we learned so far to model more complex relationships?
We will now see a simple approach to model complex non-linear relationships called *least squares*.

Review: Polynomial Functions

Recall that a polynomial of degree p is a function of the form

$$a_p x^p + a_{p-1} x^{p-1} + \ldots + a_1 x + a_0.$$

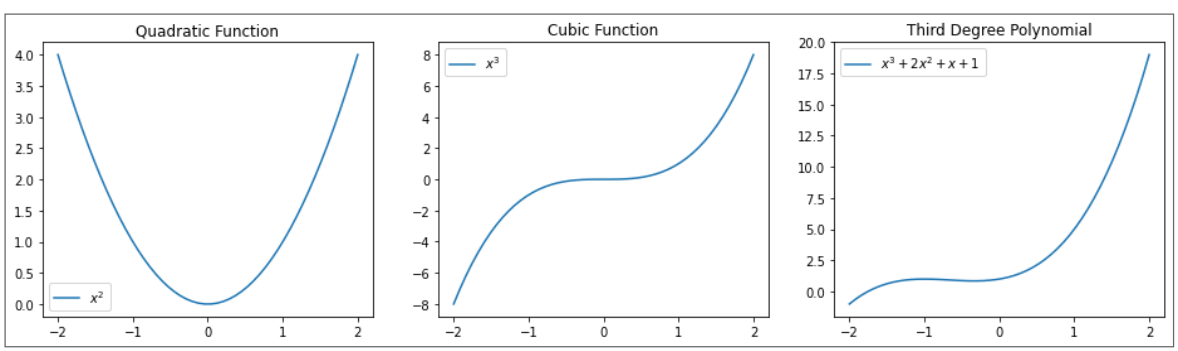
Below are some examples of polynomial functions.

In [24]:

```
import warnings
warnings.filterwarnings("ignore")

plt.figure(figsize=(16,4))
x_vars = np.linspace(-2, 2)

plt.subplot('131')
plt.title('Quadratic Function')
plt.plot(x_vars, x_vars**2)
plt.legend(["$x^2$"])

plt.subplot('132')
plt.title('Cubic Function')
plt.plot(x_vars, x_vars**3)
plt.legend(["$x^3$"])

plt.subplot('133')
plt.title('Third Degree Polynomial')
plt.plot(x_vars, x_vars**3 + 2*x_vars**2 + x_vars + 1)
plt.legend(["$x^3 + 2 x^2 + x + 1$"])
```

Out[24]:

```
<matplotlib.legend.Legend at 0x7f44fd932c50>
```



Modeling Non-Linear Relationships With Polynomial Regression

Specifically, given a one-dimensional continuous variable x, we can defining a feature function $\phi : \mathbb{R} \to \mathbb{R}^{p+1}$ as

$$\phi(x) = \begin{bmatrix} 1 \\ x \\ x^2 \\ \vdots \\ x^p \end{bmatrix}.$$

The class of models of the form

$$f_\theta(x) := \sum_{j=0}^{p} \theta_j \phi_j = \theta^\top \phi(x)$$

with parameters θ and polynomial features ϕ is the set of p-degree polynomials.

- This model is non-linear in the input variable x, meaning that we can model complex data relationships.
- It is a linear model as a function of the parameters θ, meaning that we can use our familiar ordinary least squares algorithm to learn these features.

The UCI Diabetes Dataset

In this section, we are going to again use the UCI Diabetes Dataset.

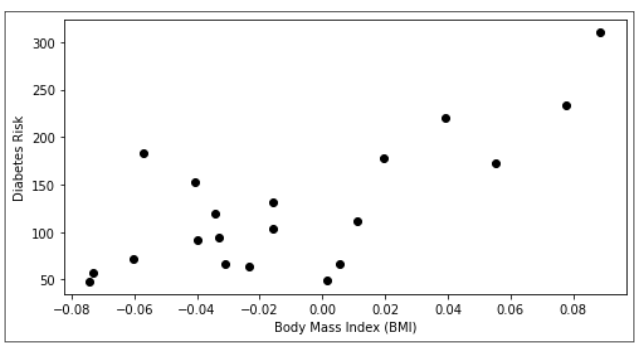
- For each patient we have a access to a measurement of their body mass index (BMI) and a quantiative diabetes risk score (from 0-300).
- We are interested in understanding how BMI affects an individual's diabetes risk.

In [25]:

```
%matplotlib inline
import matplotlib.pyplot as plt
plt.rcParams['figure.figsize'] = [8, 4]

import numpy as np
import pandas as pd
from sklearn import datasets

# Load the diabetes dataset
X, y = datasets.load_diabetes(return_X_y=True, as_frame=True)

# add an extra column of onens
X['one'] = 1

# Collect 20 data points
X_train = X.iloc[-20:]
y_train = y.iloc[-20:]

plt.scatter(X_train.loc[:,['bmi']], y_train,  color='black')
plt.xlabel('Body Mass Index (BMI)')
plt.ylabel('Diabetes Risk')
```

Out[25]:

```
Text(0, 0.5, 'Diabetes Risk')
```



Diabetes Dataset: A Non-Linear Featurization

Let's now obtain linear features for this dataset.

In [26]:

```
X_bmi = X_train.loc[:, ['bmi']]

X_bmi_p3 = pd.concat([X_bmi, X_bmi**2, X_bmi**3], axis=1)
X_bmi_p3.columns = ['bmi', 'bmi2', 'bmi3']
X_bmi_p3['one'] = 1
X_bmi_p3.head()
```

Out[26]:

	bmi	bmi2	bmi3	one
422	0.077863	0.006063	0.000472	1
423	-0.039618	0.001570	-0.000062	1
424	0.011039	0.000122	0.000001	1
425	-0.040696	0.001656	-0.000067	1
426	-0.034229	0.001172	-0.000040	1

Diabetes Dataset: A Polynomial Model

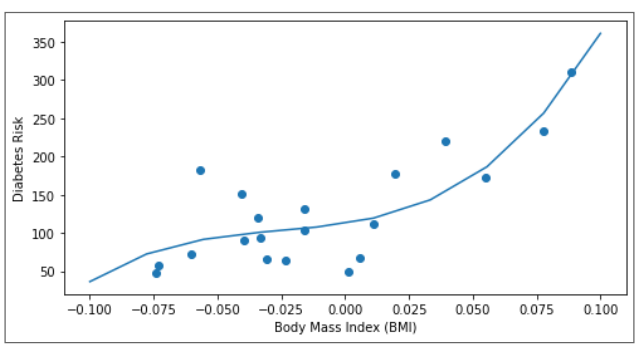
By training a linear model on this featurization of the diabetes set, we can obtain a polynomial model of diabetest risk as a function of BMI.

In [27]:

```
# Fit a linear regression
theta = np.linalg.inv(X_bmi_p3.T.dot(X_bmi_p3)).dot(X_bmi_p3.T).dot(y_train)

# Show the learned polynomial curve
x_line = np.linspace(-0.1, 0.1, 10)
x_line_p3 = np.stack([x_line, x_line**2, x_line**3, np.ones(10,)], axis=1)
y_train_pred = x_line_p3.dot(theta)

plt.xlabel('Body Mass Index (BMI)')
plt.ylabel('Diabetes Risk')
plt.scatter(X_bmi, y_train)
plt.plot(x_line, y_train_pred)
```

Out[27]:

```
[<matplotlib.lines.Line2D at 0x7f44fd5e7e48>]
```



Multivariate Polynomial Regression

We can also take this approach to construct non-linear function of multiples variable by using multivariate polynomials.
For example, a polynomial of degree 2 over two variables x_1, x_2 is a function of the form

$$a_{20}x_1^2 + a_{10}x_1 + a_{02}x_2^2 + a_{01}x_2 + a_{11}x_1x_2 + a_{00}.$$

In general, a polynomial of degree p over two variables x_1, x_2 is a function of the form

$$f(x_1, x_2) = \sum_{i,j \geq 0 : i+j \leq p} a_{ij} x_1^i x_2^j.$$

In our two-dimensional example, this corresponds to a feature function $\phi : \mathbb{R}^2 \to \mathbb{R}^6$ of the form

$$\phi(x) = \begin{bmatrix} 1 \\ x_1 \\ x_1^2 \\ x_2 \\ x_2^2 \\ x_1 x_2 \end{bmatrix}.$$

The same approach holds for polynomials of an degree and any number of variables.

Towards General Non-Linear Features

Any non-linear feature map $\phi(x) : \mathbb{R}^d \to \mathbb{R}^p$ can be used in this way to obtain general models of the form

$$f_\theta(x) := \theta^\top \phi(x)$$

that are highly non-linear in x but linear in θ.

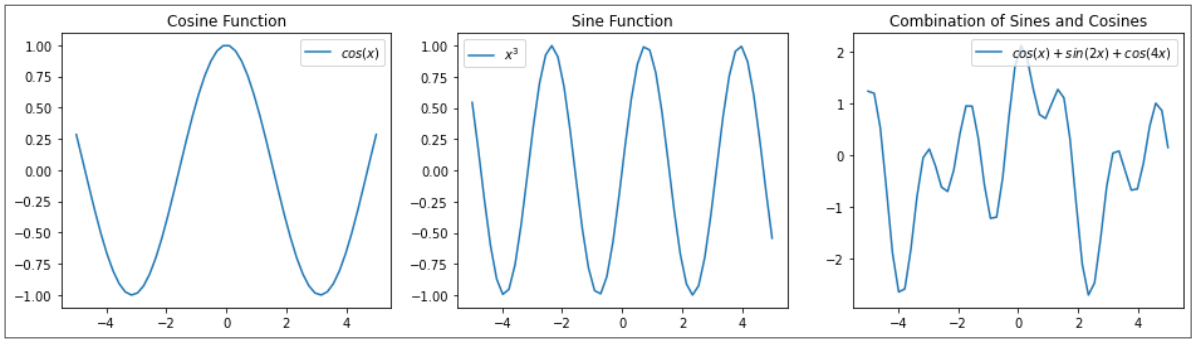
For example, here is a way of modeling complex periodic functions via a sum of sines and cosines.

In [28]:

```
import warnings
warnings.filterwarnings("ignore")

plt.figure(figsize=(16,4))
x_vars = np.linspace(-5, 5)

plt.subplot('131')
plt.title('Cosine Function')
plt.plot(x_vars, np.cos(x_vars))
plt.legend(["$cos(x)$"])

plt.subplot('132')
plt.title('Sine Function')
plt.plot(x_vars, np.sin(2*x_vars))
plt.legend(["$x^3$"])

plt.subplot('133')
plt.title('Combination of Sines and Cosines')
plt.plot(x_vars, np.cos(x_vars) + np.sin(2*x_vars) + np.cos(4*x_vars))
plt.legend(["$cos(x) + sin(2x) + cos(4x)$"])
```

Out[28]:

```
<matplotlib.legend.Legend at 0x7f44fd4bd6d8>
```


Cosine Function
cos(x)
Sine Function
x³
Combination of Sines and Cosines
cos(x) + sin(2x) + cos(4x)

Algorithm: Non-Linear Least Squares

- **Type**: Supervised learning (regression)
- **Model family**: Linear in the parameters; non-linear with respect to raw inputs.
- **Features**: Non-linear functions of the attributes
- **Objective function**: Mean squared error
- **Optimizer**: Normal equations

In [ ]: