Introduction to Machine Learning

Tan Bui-Thanh [*]

1st September 2023

[*]Department of Aerospace Engineering and Engineering Mechanics, The Oden Institute for Computational Engineering and Sciences, UT Austin, Austin, Texas (tanbui@oden.utexas.edu, https://users.oden.utexas.edu/~tanbui/).

Contents

CHAPTER 1

Lecture 1: Introduction

1.1 Welcome to Applied Machine Learning!

Machine learning is one of today's most exciting emerging technologies.

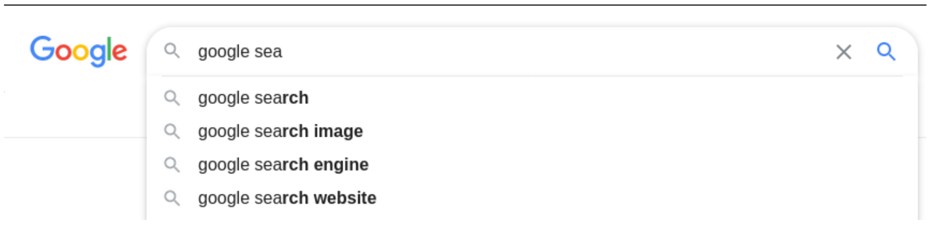
In this course, you will learn what machine learning is, what are the most important techniques in machine learning, and how to apply them to solve problems in the real world.

1.2 What is Machine Learning?

We hear a lot about machine learning (or ML for short) in the news. But what is it, really?

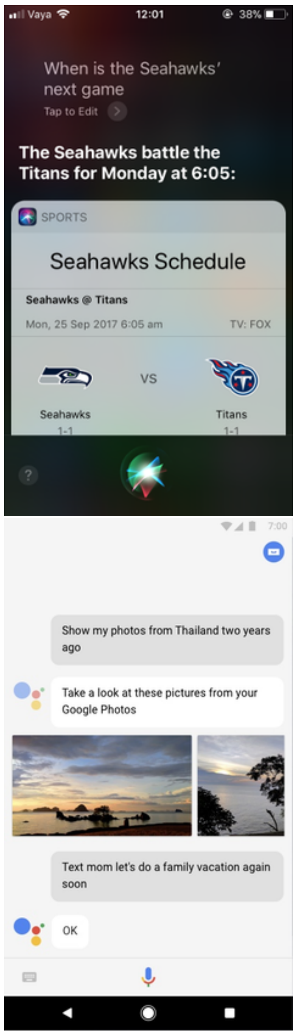
1.2.1 ML in Everyday Life: Search Engines

You use machine learninng every day when you run a search engine query.



1.2.2 ML in Everyday Life: Personal Assistants

Machine learning also powers the speech recognition, question answering and other intelligent capabilities of smartphone assistants like Apple Siri.



When is the Seahawks' next game

Tap to Edit

The Seahawks battle the Titans for Monday at 6:05:

Show my photos from Thailand two years ago

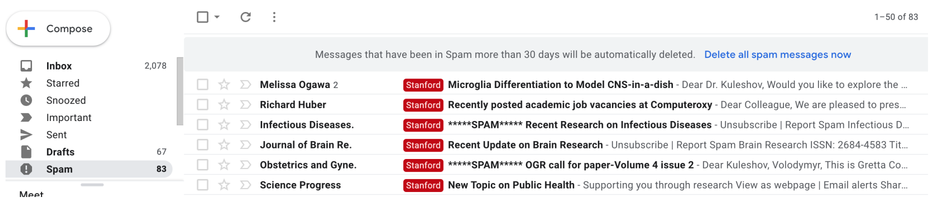
Take a look at these pictures from your Google Photos

Text mom let's do a family vacation again soon

OK

1.2.3 ML in Everyday Life: Spam/Fraud Detection

Machine learning is used in every spam filter, such as in Gmail.



Messages that have been in Spam more than 30 days will be automatically deleted. Delete all spam messages now

Melissa Ogawa 2 — Stanford **Microglia Differentiation to Model CNS-in-a-dish** - Dear Dr. Kuleshov, Would you like to explore the ...

Richard Huber — Stanford **Recently posted academic job vacancies at Computeroxy** - Dear Colleague, We are pleased to pres...

Infectious Diseases. — Stanford *******SPAM***** Recent Research on Infectious Diseases** - Unsubscribe | Report Spam Infectious D...

Journal of Brain Re. — Stanford **Recent Update on Brain Research** - Unsubscribe | Report Spam Brain Research ISSN: 2684-4583 Tit...

Obstetrics and Gyne. — Stanford *******SPAM***** OGR call for paper-Volume 4 issue 2** - Dear Kuleshov, Volodymyr, This is Gretta Co...

Science Progress — Stanford **New Topic on Public Health** - Supporting you through research View as webpage | Email alerts Shar...

ML systems are also used by credit card companies and banks to automatically detect fraudulent behavior.

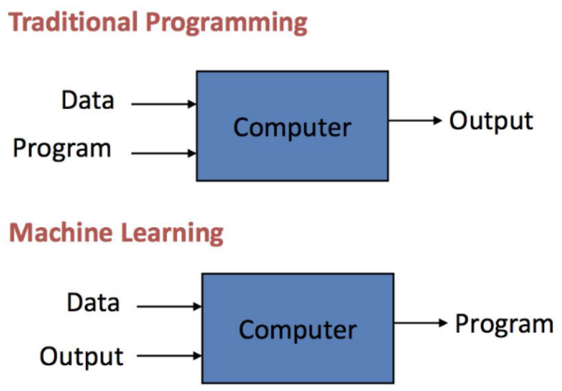
1.2.4 ML in Everyday Life: Self-Driving Cars

One of the most exciting and cutting-edge uses of machine learning algorithms are in autonomous vehicles.

1.3 A Definition of Machine Learning

In 1959, Arthur Samuel defined machine learning as follows.

Machine learning is a field of study that gives computers the ability to learn without being explicitly programmed.



For example, in traditional programming we would like to calculate the addition rule

$$y = \textbf{sum}\,(x_1, x_2)$$

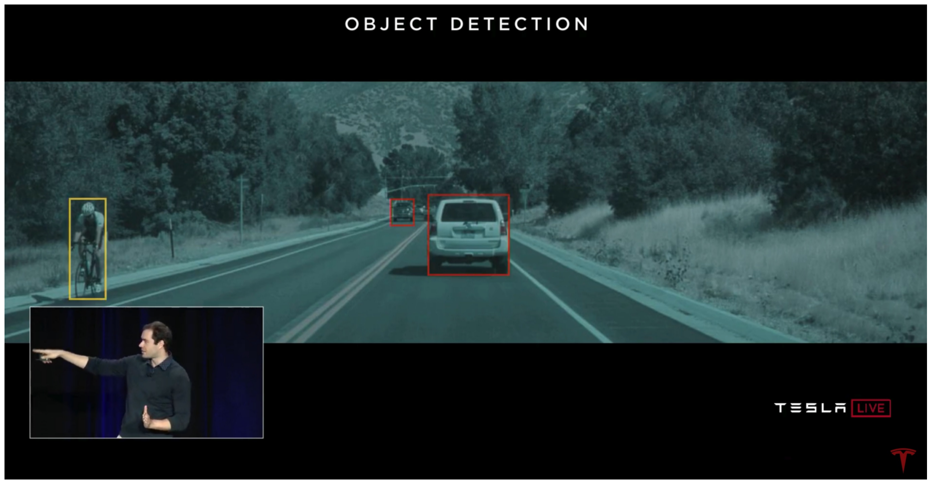
we simply use a calculator or call the functions to implement

$$y = x_1 + x_2$$
$$\textbf{return}\ y$$

On the other hand, assume that we do not know the addition rule, however, we have some pairs of $\left\{x_1^i, x_2^i, y^i\right\}_{i=1}^N$, we can use machine learning to learn/approximate the underlying rule (addition rule).

What does "learn" and "explicitly programmed" mean here? Let's look at an example of Self Driving Cars

A self-driving car system uses dozens of components that include detection of cars, pedestrians, and other objects.



1.3.1 Self Driving Cars: A Rule-Based Algorithm

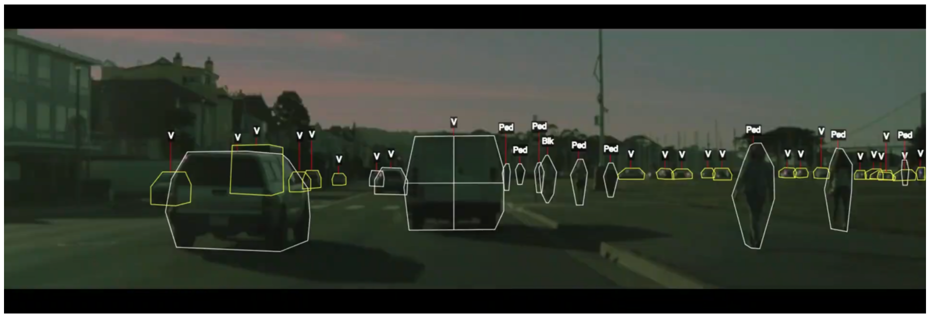
One way to build a detection system is to write down rules.

```
[ ]: # pseudocode example for a rule-based classification system
object = camera.get_object()
if object.has_wheels(): # does the object have wheels?
    if len(object.wheels) == 4: return "Car" # four wheels => car
    elif len(object.wheels) == 2:,
        if object.seen_from_back():
            return "Car" # viewed from back, car has 2 wheels
        else:
            return "Bicycle" # normally, 2 wheels => bicycle
return "Unknown" # no wheels? we don't know what it is
```

In practice it takes too much time to account for all rules. Even when it is possible, the program is too complicated. Furthermore, it's almost impossible for a human to specify all the edge cases. So what are we going to do?

1.3.2 Self Driving Cars: An ML Approach

The machine learning approach is to teach a computer how to do detection by showing it many examples of different objects, just like how we learn.



No manual programming is needed: the computer learns to identify a pedestrian or a car on its own! Just like a human does. This principle can be applied to countless domains: medical diagnosis, factory automation, machine translation, and many more!

Note that the output of a machine learning algorithm is a program that try to learns the rule/law in the data.

1.3.3 Why Machine Learning?

Why is this approach to building software interesting?

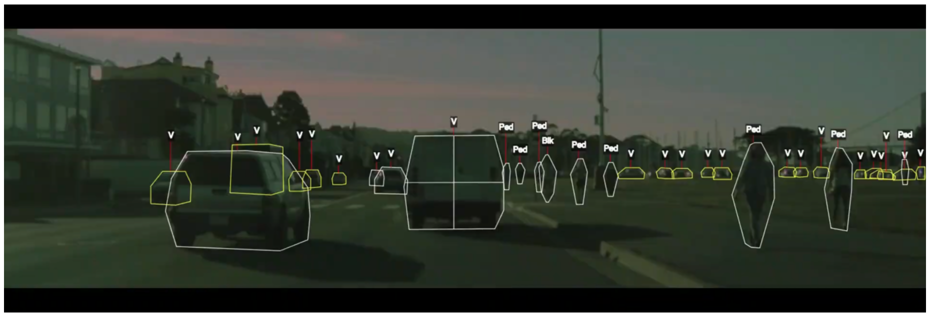
- It allows building practical systems for real-world applications that couldn't be solved otherwise. For example, for application in which we do not know its governing physics/equations: all we have is data!
- Learning is wildly regarded as a key approach towards building general-purpose artificial intelligence systems.
- As we have discussed, machine learning mimics human learning. Thus, the science and engineering of machine learning offers insights into human intelligence.

1.4 Three Main Approaches to Machine Learning

We have distinguished traditional programming and machine learning. W would like to emphasize that the output of a machine learning algorithm is a program that tries to learning the underlying rule in the data. We now discuss three main approaches for machine learning. This provides us the first peek into how the learning process is carried out within a particular machine learning approach.

1.4.1 Supervised Learning

The most common approach to machine learning is supervised learning. The self-driving car example, shown again here, is an example of supervised learning.



A supervised learning approach consists of three steps (will be made precise later when we get to supervised learning):

1. First, we collect a dataset of labeled training examples $\{x_i, y_i\}_{i=1}^N$ with N being the total number of data, where x_i is the image and y_i is the label (e.g. vehicle, pedestrian, etc).
2. We train a model to output accurate predictions on this dataset.
3. When the model sees new, but similar data, it will also be accurate.

A Supervised Learning Dataset

Consider a simple dataset for supervised learning: house prices in Boston.

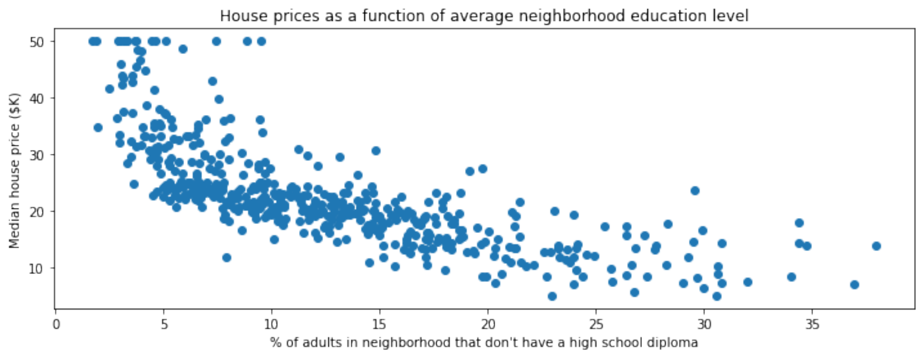
- Each datapoint is a house.
- We know its price, neighborhood, size, etc.

[ ]:
```python
# We will load the dataset from the sklearn ML library
from sklearn import datasets
boston = datasets.load_boston()
```

We will visualize two variables in this dataset: house price and the education level in the neighborhood.

[ ]:
```python
import matplotlib.pyplot as plt
plt.rcParams['figure.figsize'] = [12, 4]
plt.scatter(boston.data[:,12], boston.target)
plt.ylabel("Median house price ($K)")
plt.xlabel("% of adults in neighborhood that don't have a high school diploma")
plt.title("House prices as a function of average neighborhood education level")
```

[ ]:
```
Text(0.5, 1.0, 'House prices as a function of average neighborhood education
level')
```



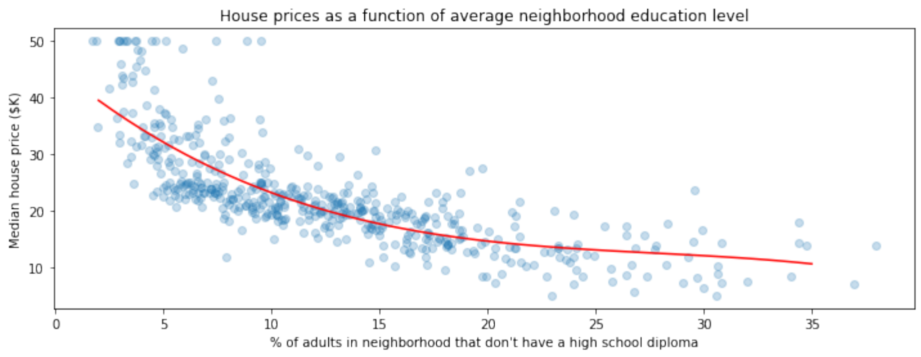
A Supervised Learning Algorithm

We can use this dataset of examples to fit a supervised learning model.

- The model maps input *x* (the education level) to output a *y* (the house price).
- It learns the mapping from our dataset of examples (x, y).

[ ]:
```
import numpy as np
from sklearn.kernel_ridge import KernelRidge

# Apply a supervised learning algorithm
model = KernelRidge(alpha=1, kernel='poly')
model.fit(boston.data[:,[12]], boston.target.flatten())
predictions = model.predict(np.linspace(2, 35)[:, np.newaxis])

# Visualize the results
plt.scatter(boston.data[:,[12]], boston.target, alpha=0.25)
plt.plot(np.linspace(2, 35), predictions, c='red')
plt.ylabel("Median house price ($K)")
plt.xlabel("% of adults in neighborhood that don't have a high school diploma")
plt.title("House prices as a function of average neighborhood education level")
```

[ ]:
```
Text(0.5, 1.0, 'House prices as a function of average neighborhood education
level')
```



Applications of Supervised Learning

Many of the most important applications of machine learning are supervised:

- Classifying medical images.
- Translating between pairs of languages.
- Detecting objects in a self-driving car.

1.4.2 Unsupervised Learning

Here, we have a dataset *without* labels, that is, we only have $\{x_i\}_{i=1}^N$. Our goal is to learn/identify/detect something interesting about the structure of the data:

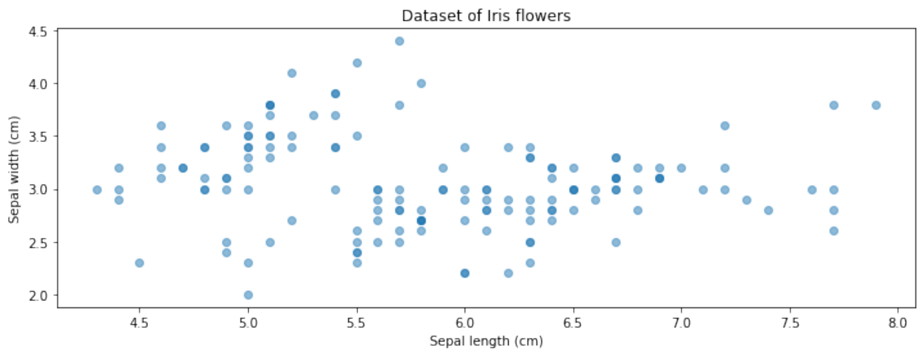
- Clusters hidden in the dataset.
- Outliers: particularly unusual and/or interesting datapoints.
- Useful signal hidden in noise, e.g. human speech over a noisy phone.

An Unsupervised Learning Dataset

Here is a simple example of an unsupervised learning dataset: Iris flowers.

[ ]:
```
# Load and visualize the Iris flower dataset
iris = datasets.load_iris()
plt.scatter(iris.data[:,0], iris.data[:,1], alpha=0.5)
plt.ylabel("Sepal width (cm)")
plt.xlabel("Sepal length (cm)")
plt.title("Dataset of Iris flowers")
```

[ ]:
```
Text(0.5, 1.0, 'Dataset of Iris flowers')
```



An Unsupervised Learning Algorithm

We can use this dataset of examples to fit an unsupervised learning model.

- The model defines a probability distribution over the inputs.
- The probability distribution identifies multiple components (multiple peaks).

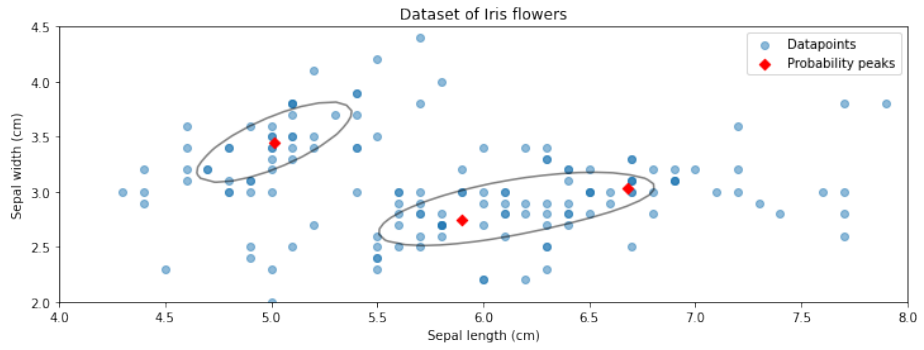
- The components indicate structure in the data.

```
[ ]: # fit a Gaussian Mixture Model with three components
     from sklearn import mixture
     model = mixture.GaussianMixture(n_components=3, covariance_type='full')
     model.fit(iris.data[:,[0,1]])
```

```
[ ]: GaussianMixture(n_components=3)
```

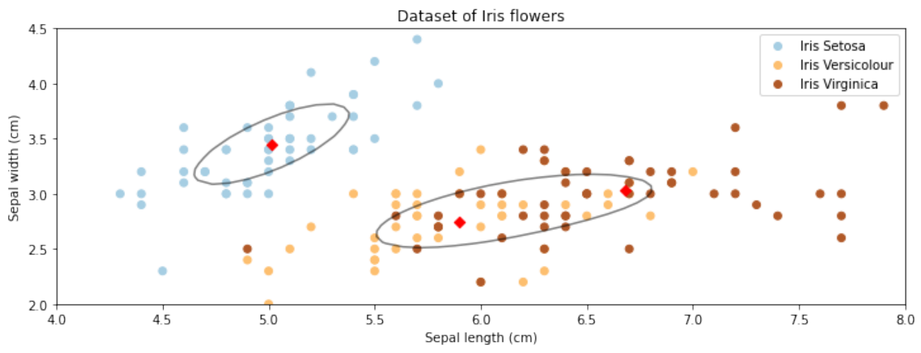
```
[ ]: # display learned probabilities as a contour plot
     x, y = np.linspace(4.0, 8.0), np.linspace(2.0, 4.5)
     X, Y = np.meshgrid(x, y)
     Z = -model.score_samples(np.array([X.ravel(), Y.ravel()]).T).reshape(X.shape)
     plt.contour(X, Y, Z, levels=np.logspace(0, 10, 1), cmap="gray", alpha=0.5)
     plt.scatter(iris.data[:,0], iris.data[:,1], alpha=0.5)
     plt.scatter(model.means_[:,0], model.means_[:,1], marker='D', c='r')
     plt.ylabel("Sepal width (cm)")
     plt.xlabel("Sepal length (cm)")
     plt.title("Dataset of Iris flowers")
     plt.legend(['Datapoints', 'Probability peaks'])
```

```
[ ]: <matplotlib.legend.Legend at 0x12293ad68>
```



```
[ ]: CS = plt.contour(X, Y, Z, levels=np.logspace(0, 30, 1), cmap='gray', alpha=0.5)
     p1 = plt.scatter(iris.data[:,0], iris.data[:,1], alpha=1, c=iris.target,␣
       ↪cmap='Paired')
     plt.scatter(model.means_[:,0], model.means_[:,1], marker='D', c='r')
     plt.ylabel("Sepal width (cm)")
     plt.xlabel("Sepal length (cm)")
     plt.title("Dataset of Iris flowers")
     plt.legend(handles=p1.legend_elements()[0], labels=['Iris Setosa', 'Iris␣
       ↪Versicolour', 'Iris Virginica'])
```

```
[ ]: <matplotlib.legend.Legend at 0x1229d3668>
```



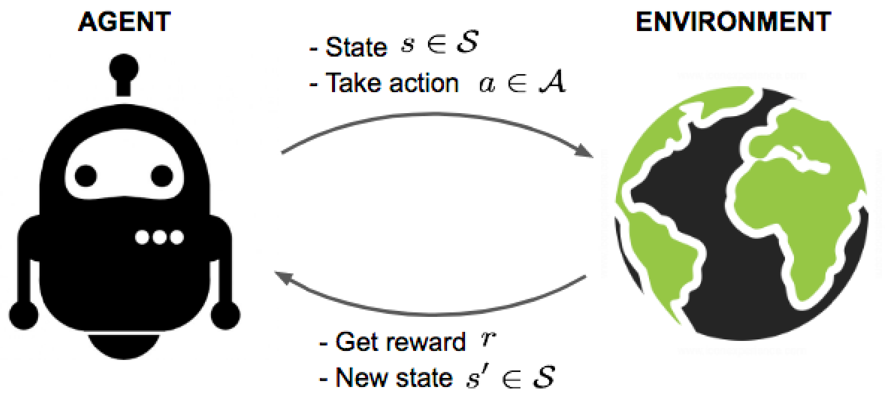
Applications of Unsupervised Learning

Unsupervised learning also has numerous applications:

- Recommendation systems: suggesting movies on Netflix.
- Anomaly detection: identifying factory components that are likely to break soon.
- Signal denoising: extracting human speech from a noisy recording.

1.4.3 Reinforcement Learning

In reinforcement learning, an agent is interacting with the world over time. We teach it good behavior by providing it with rewards.


AGENT
ENVIRONMENT
- State $s \in \mathcal{S}$
- Take action $a \in \mathcal{A}$
- Get reward r
- New state $s' \in \mathcal{S}$

We will not cover Reinforcement learning in this class.

Applications of Reinforcement Learning

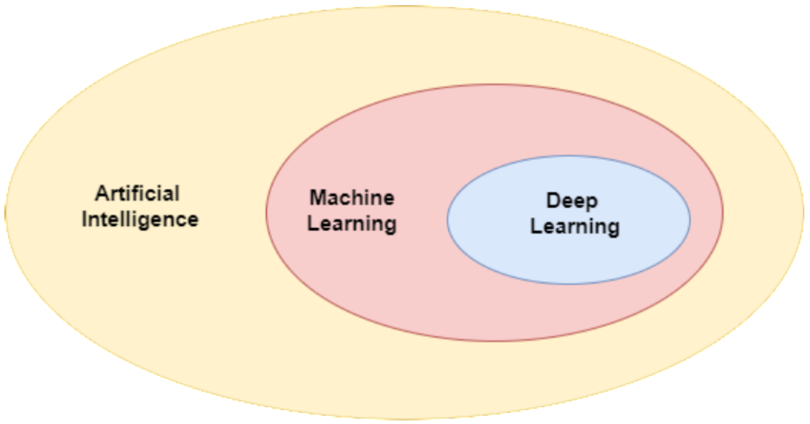
Applications of reinforcement learning include:

- Creating agents that play games such as Chess or Go.
- Controling the cooling systems of datacenters to use energy more efficiently.
- Designing new drug compounds.

1.5 Artificial Intelligence and Deep Learning

Machine learning is often discussed in the context of these two fields.

- AI is about building machines that exhibit intelligence.
- ML enables machines to learn from experience, a useful tool for AI.
- Deep learning focuses on a family of learning algorithms loosely inspired by the brain.



1.6 About this class

Next, let's look at the machine learning topics that we will cover.

1.6.1 Teaching Approach

The focus of this course is on applied machine learning.

- We will cover a broad toolset of core algorithms from many different subfields of ML.
- We will emphasize both theories (the math behind) and applications and show how to implement and apply algorithms via examples and exercises.

Why are we following this approach?

- Applying machine learning is among the most in demand industry skills right now.
- There can be a gap between theory and practice, especially in modern machine learning. We try to bridge that gap in this class. We will make effort to understand how an algorithm works. The implementation aspect of the class is a mix of using already-implemented algorithms and self-implementation.

1.6.2 What will you Learn?

- What are the core algorithms of ML and their mathematics.
- How to implement algorithms from scratch as well as using ML libraries and apply them to problems in computer vision, language processing, medical analysis, and more.

- Why machine learning algorithms work and how to use that knowledge to debug and improve them.

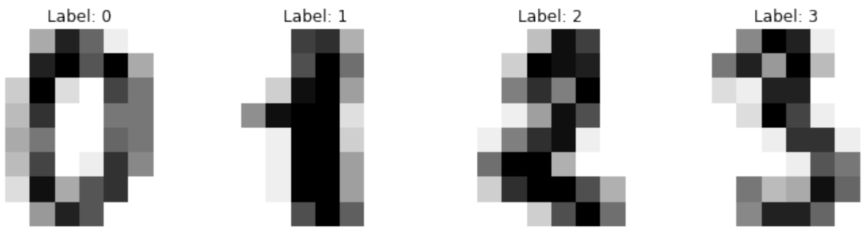
1.6.3 Software You Will Use

You will use `scikit-learn` for our class. It implements most classical machine learning algorithms.

```
[ ]: import numpy as np
     import matplotlib.pyplot as plt
     from sklearn import datasets, neural_network
     plt.rcParams['figure.figsize'] = [12, 4]
```

We can use these libraries to load a simple datasets of handwritten digits.

```
[ ]: # https://scikit-learn.org/stable/auto_examples/classification/
       ↪plot_digits_classification.html
     # load the digits dataset
     digits = datasets.load_digits()

     # The data that we are interested in is made of 8x8 images of digits, let's
     # have a look at the first 4 images.
     _, axes = plt.subplots(1, 4)
     images_and_labels = list(zip(digits.images, digits.target))
     for ax, (image, label) in zip(axes, images_and_labels[:4]):
         ax.set_axis_off()
         ax.imshow(image, cmap=plt.cm.gray_r, interpolation='nearest')
         ax.set_title('Label: %i' % label)
```



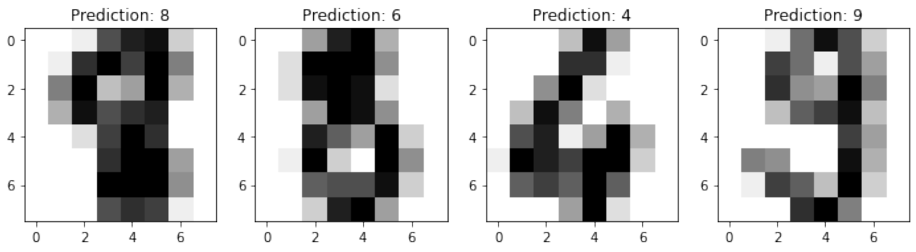
We can now load and train this algorithm inside the slides.

```
[ ]: np.random.seed(0)
     # To apply a classifier on this data, we need to flatten the image, to
     # turn the data in a (samples, feature) matrix:
     data = digits.images.reshape((len(digits.images), -1))

     # create a small neural network classifier
     from sklearn.neural_network import MLPClassifier
     classifier = MLPClassifier(alpha=1e-3)

     # Split data into train and test subsets
     X_train, X_test, y_train, y_test = sk.model_selection.train_test_split(data,␣
       ↪digits.target, test_size=0.5, shuffle=False)
```

```
# We learn the digits on the first half of the digits
classifier.fit(X_train, y_train)

# Now predict the value of the digit on the second half:
predicted = classifier.predict(X_test)
```

We can now visualize the results.

[ ]:
```
_, axes = plt.subplots(1, 4)
images_and_predictions = list(zip(digits.images[n_samples // 2:], predicted))
for ax, (image, prediction) in zip(axes, images_and_predictions[:4]):
    ax.imshow(image, cmap=plt.cm.gray_r, interpolation='nearest')
    ax.set_title('Prediction: %i' % prediction)
```



1.7 Class Content

The course spans about 26-28 lectures approximately divided up into a set of blocks:

- Supervised and unsupervised algorithms.
- Foundations of machine learning.
- Applying machine learning in practice.
- Advanced topics and **guest lectures (?)**.

1.7.1 Machine Learning Algorithms

- Supervised learning algorithms: linear models and extensions, kernel machines, tree-based algorithms.
- Unsupervised learning algorithms: density estimation, clustering, dimensionality reduction
- Introduction to learning with deep neural networks.

1.7.2 Foundations of Machine Learning

- The basic language of machine learning: datasets, features, models, objective functions.
- Tools for machine learning: optimization, probability, linear algebra.
- Why do algorithms work in practice? Probabilistic foundations.

1.7.3 Applying Machine Learning

- Evaluating machine learning algorithms.
- Diagnosing and debugging performance.

- Analyzing errors and improving models.
- Deploying and debugging pipelines.

1.7.4 Advanced Machine Learning Topics

- Convolutional Neural Networks, Recurrent Neural networks, Graph Neural networks (if time permits)

1.8 Course Assignments

Homework will include both derivations (either expanding some details that lectures do not cover or new related approaches/derivations) and programings in python within the Scikit-learn environment. There will be approximately 7-10 homework assignments in the semester. Students will be given a minimum of one or two weeks to complete each assignment. Assignments must be submitted electronically via canvas. There will be a group project due on the last day of the class when each group will present their project.

1.9 Prerequisites. Is this class For You?

This course is designed to aimed at a very general technical audience. **Boldface** topics will be reviewed. Main requirements are:

- Programming experience (at least 1 year), preferably in Python.
- College-level linear algebra, Matrix operations, and the curiosity and the desire to learn the math behind machine learning algorithms/approaches. This is the key different from this class and the other machine learning classes.
- **College-level probability. Probability distributions, random variables, Bayes' rule, etc.** We will review most of them in class.

Other Logistics

- The majority of course materials will be accessible online.
- Grading will be based on a combination of homework assignments, exames and the final project. See the syllabus for more details.
- There is no required textbook, but we recommend Elements of Statistical Learning by Hastie, Tibshirani, and Friedman.

CHAPTER 2

Lecture 2: Supervised Machine Learning

Recall that a supervised learning approach consists of three steps:

1. **First, we collect a dataset of labeled training examples**
2. **We train a model to output accurate predictions on this dataset.**
3. **When the model sees new, but similar data, it will also be accurate.**

2.1 A First Supervised Machine Learning Problem

Let's start with a simple example of a supervised learning problem: predicting diabetes risk.

Suppose we have a dataset of diabetes patients.

- For each patient we have a access to measurements from their medical record and an estimate of diabetes risk.
- We are interested in predicting how the measurements imply an individual's diabetes risk.

2.1.1 Three Components of A Supervised Machine Learning Problem

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

$$\text{Dataset} + \text{Algorithm} \rightarrow \text{Predictive Model}$$

Note that the predictive model (e.g. a function) is the result of a supervised machine learning approach. It belongs to a family of models (e.g. a family of functions) that is capable of presenting the relationship between inputs and targets. After being trained by the algorithm, the predictive model (typically the best among all in the family of models) can predict labels for unseen data.

2.1.2 A Supervised Learning Dataset

Let's return to our example: predicting diabates risk. What would a dataset look like?

We will use the UCI Diabetes Dataset; it's a toy dataset that's often used to demonstrate machine learning algorithms.

- For each patient we have a access to a measurement of their body mass index (BMI) and a quantiative diabetes risk score (from 0-400).
- We are interested in predicting an individual's diabetes risk (y) from his/her BMI x.

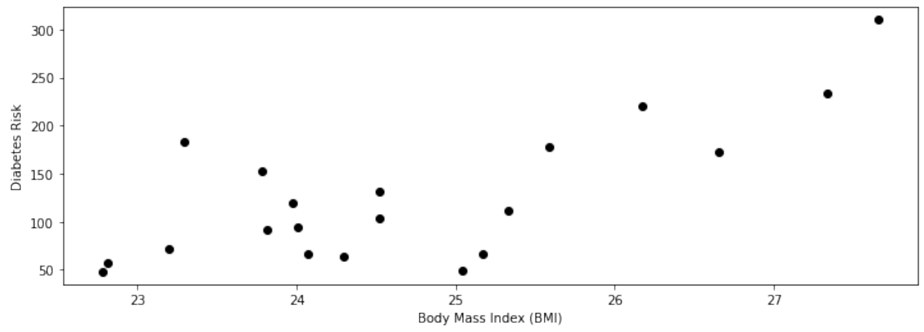
[1]:
```
import numpy as np
import pandas as pd
from sklearn import datasets

# Load the diabetes dataset
diabetes_X, diabetes_y = datasets.load_diabetes(return_X_y=True, as_frame=True)

# Use only the BMI feature
diabetes_X = diabetes_X.loc[:, ['bmi']]

# The BMI is zero-centered and normalized; we recenter it for ease of␣
 ↪presentation
diabetes_X = diabetes_X * 30 + 25

# Collect 20 data points
diabetes_X_train = diabetes_X.iloc[-20:]
diabetes_y_train = diabetes_y.iloc[-20:]

# Display some of the data points
pd.concat([diabetes_X_train, diabetes_y_train], axis=1).head()
```

[1]:
```
            bmi  target
422  27.335902   233.0
423  23.811456    91.0
424  25.331171   111.0
425  23.779122   152.0
426  23.973128   120.0
```

We can also visualize this two-dimensional dataset.

[2]:
```
%matplotlib inline
import matplotlib.pyplot as plt
plt.rcParams['figure.figsize'] = [12, 4]

plt.scatter(diabetes_X_train, diabetes_y_train,  color='black')
plt.xlabel('Body Mass Index (BMI)')
plt.ylabel('Diabetes Risk')
```

[2]:
```
Text(0, 0.5, 'Diabetes Risk')
```



2.1.3 A Supervised Learning Algorithm (Part 1)

What is the relationship between BMI and diabetes risk?

We could assume that risk is a linear function of BMI. In other words, for some unknown $\theta_0, \theta_1 \in \mathbb{R}$, we have

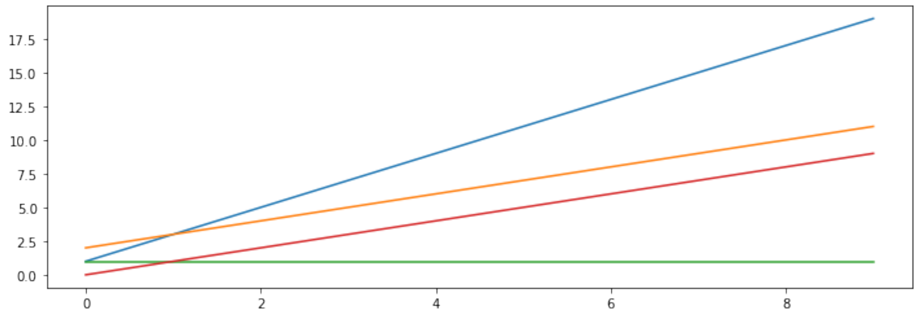
$$y = \theta_1 \cdot x + \theta_0,$$

where x is the BMI (also called the dependent variable), and y is the diabetes risk score (the independent variable).

Note that θ_1, θ_0 are the slope and the intercept of the line relates x to y. We call them *parameters*.

We can visualize this for a few values of θ_1, θ_0.

[3]:
```python
theta_list = [(1, 2), (2,1), (1,0), (0,1)]
for theta0, theta1 in theta_list:
    x = np.arange(10)
    y = theta1 * x + theta0
    plt.plot(x,y)
```



2.1.4 A Supervised Learning Algorithm (Part 2)

Assuming that x, y follow the above linear relationship, the goal of the **supervised learning algorithm** is to find a good set of parameters consistent with the data.

We will see many algorithms for this task. For now, let's call the `sklearn.linear_model` library to find a θ_1, θ_0 that fit the data well.

[4]:
```
from sklearn import linear_model
from sklearn.metrics import mean_squared_error

# Create linear regression object
regr = linear_model.LinearRegression()

# Train the model using the training sets
regr.fit(diabetes_X_train, diabetes_y_train.values)

# Make predictions on the training set
diabetes_y_train_pred = regr.predict(diabetes_X_train)

# The coefficients
print('Slope (theta1): \t', regr.coef_[0])
print('Intercept (theta0): \t', regr.intercept_)
```

```
Slope (theta1):          37.378842160517664
Intercept (theta0):      -797.0817390342369
```

2.1.5 A Supervised Learning Model

The supervised learning algorithm gave us a pair of parameters θ_1^*, θ_0^* after training. We will learn what it means by training shortly. These define the *predictive model* f^*, defined as

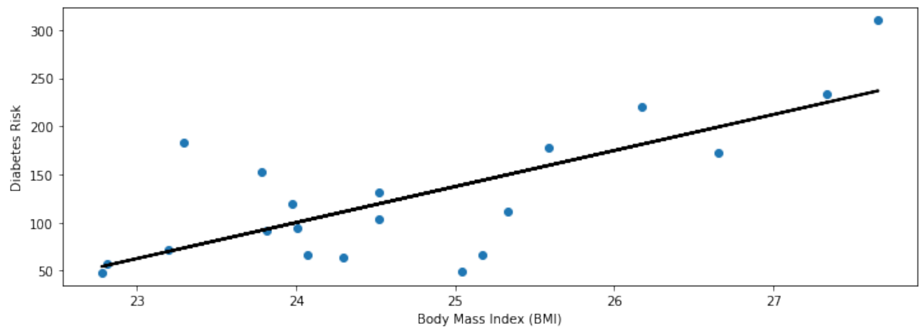
$$f(x) = \theta_1^* \cdot x + \theta_0^*,$$

where again x is the BMI, and y is the diabetes risk score.

We can visualize the linear model that fits our data.

[5]:
```
plt.xlabel('Body Mass Index (BMI)')
plt.ylabel('Diabetes Risk')
plt.scatter(diabetes_X_train, diabetes_y_train)
plt.plot(diabetes_X_train, diabetes_y_train_pred, color='black', linewidth=2)
```

[5]:
```
[<matplotlib.lines.Line2D at 0x7f5f0057d5f8>]
```



2.1.6 Predictions using the predictive model obtained from Supervised Learning

Given a new dataset of patients with a known BMI, we can use this model to estimate their diabetes risk.

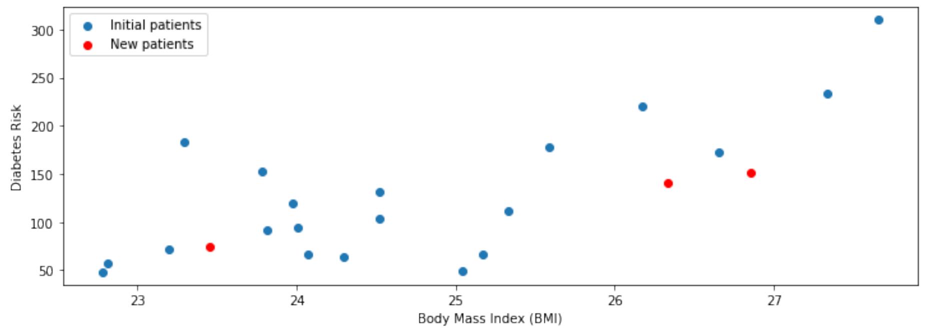
Given a new x', we can output a predicted y' as

$$y' = f(x') = \theta_1^* \cdot x' + \theta_0.$$

Let's start by loading more data. We will load three new patients (shown in red below) that we haven't seen before.

[6]:
```
# Collect 3 data points
diabetes_X_test = diabetes_X.iloc[:3]
diabetes_y_test = diabetes_y.iloc[:3]

plt.scatter(diabetes_X_train, diabetes_y_train)
plt.scatter(diabetes_X_test, diabetes_y_test,  color='red')
plt.xlabel('Body Mass Index (BMI)')
plt.ylabel('Diabetes Risk')
plt.legend(['Initial patients', 'New patients'])
```

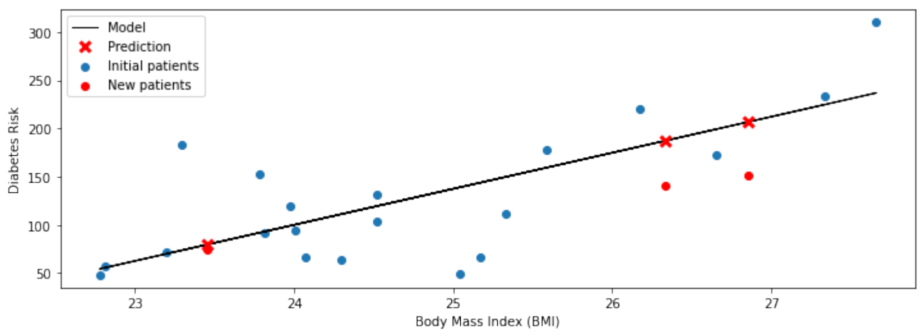
[6]:
```
<matplotlib.legend.Legend at 0x7f5efe84c710>
```



Our linear model provides an estimate of the diabetes risk for these patients.

[7]:
```
# generate predictions on the new patients
diabetes_y_test_pred = regr.predict(diabetes_X_test)

# visualize the results
plt.xlabel('Body Mass Index (BMI)')
plt.ylabel('Diabetes Risk')
plt.scatter(diabetes_X_train, diabetes_y_train)
plt.scatter(diabetes_X_test, diabetes_y_test, color='red', marker='o')
plt.plot(diabetes_X_train, diabetes_y_train_pred, color='black', linewidth=1)
plt.plot(diabetes_X_test, diabetes_y_test_pred, 'x', color='red', mew=3,␣
  ↪markersize=8)
plt.legend(['Model', 'Prediction', 'Initial patients', 'New patients'])
```

```
[7]: <matplotlib.legend.Legend at 0x7f5efe7c7550>
```



2.2 Why Supervised Learning?

Supervised learning can be useful in many ways.

- Making predictions on new data.
- Understanding the mechanisms through which input variables affect targets.

2.3 Applications of Supervised Learning

Many of the most important applications of machine learning are supervised:

- Classifying medical images.
- Translating between pairs of languages.
- Detecting objects in a self-driving car.

2.4 Anatomy of a Supervised Learning Problem: Datasets

We have seen a simple example of a supervised machine learning problem and an algorithm for solving this problem.

Let's now look at what a general supervised learning problem looks like.

Recall: Three Components of A Supervised Machine Learning Problem

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

$$\text{Dataset} + \text{Algorithm} \to \text{Predictive Model}$$

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

A Supervised Learning Dataset

We are going to dive deeper into what's a supervised learning dataset. As an example, consider the full version of the UCI Diabetes Dataset seen earlier.

Previsouly, we only looked at the patients' BMI, but this dataset actually records many additional measurements.

The UCI dataset contains many additional data columns besides `bmi`, including age, sex, and blood pressure. We can ask `sklearn` to give us more information about this dataset.

[8]:
```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
plt.rcParams['figure.figsize'] = [12, 4]
from sklearn import datasets

# Load the diabetes dataset
diabetes = datasets.load_diabetes(as_frame=True)
print(diabetes.DESCR)
```

```
.. _diabetes_dataset:

Diabetes dataset
----------------

Ten baseline variables, age, sex, body mass index, average blood
pressure, and six blood serum measurements were obtained for each of n =
442 diabetes patients, as well as the response of interest, a
quantitative measure of disease progression one year after baseline.

**Data Set Characteristics:**

  :Number of Instances: 442

  :Number of Attributes: First 10 columns are numeric predictive values

  :Target: Column 11 is a quantitative measure of disease progression one year
after baseline

  :Attribute Information:
      - age     age in years
      - sex
      - bmi     body mass index
      - bp      average blood pressure
      - s1      tc, total serum cholesterol
      - s2      ldl, low-density lipoproteins
      - s3      hdl, high-density lipoproteins
      - s4      tch, total cholesterol / HDL
      - s5      ltg, possibly log of serum triglycerides level
      - s6      glu, blood sugar level
```

```
Note: Each of these 10 feature variables have been mean centered and scaled by
the standard deviation times `n_samples` (i.e. the sum of squares of each column
totals 1).

Source URL:
https://www4.stat.ncsu.edu/~boos/var.select/diabetes.html

For more information see:
Bradley Efron, Trevor Hastie, Iain Johnstone and Robert Tibshirani (2004) "Least
Angle Regression," Annals of Statistics (with discussion), 407-499.
(https://web.stanford.edu/~hastie/Papers/LARS/LeastAngle_2002.pdf)
```

A Supervised Learning Dataset: Notation

We say that a training dataset of size n (e.g., n patients) is a set

$$\mathcal{D} = \{(x^{(i)}, y^{(i)}) \mid i = 1, 2, ..., n\}$$

Each $x^{(i)}$ denotes an input (e.g., the measurements for patient i), and each $y^{(i)} \in \mathcal{Y}$ is a target (e.g., the diabetes risk).

Together, $(x^{(i)}, y^{(i)})$ form a *training example*.

We can look at the diabetes dataset in this form.

[9]:
```
# Load the diabetes dataset
diabetes_X, diabetes_y = diabetes.data, diabetes.target

# Print part of the dataset
diabetes_X.head()
```

[9]:
```
        age       sex       bmi        bp        s1        s2        s3  \
0  0.038076  0.050680  0.061696  0.021872 -0.044223 -0.034821 -0.043401
1 -0.001882 -0.044642 -0.051474 -0.026328 -0.008449 -0.019163  0.074412
2  0.085299  0.050680  0.044451 -0.005671 -0.045599 -0.034194 -0.032356
3 -0.089063 -0.044642 -0.011595 -0.036656  0.012191  0.024991 -0.036038
4  0.005383 -0.044642 -0.036385  0.021872  0.003935  0.015596  0.008142

         s4        s5        s6
0 -0.002592  0.019908 -0.017646
1 -0.039493 -0.068330 -0.092204
2 -0.002592  0.002864 -0.025930
3  0.034309  0.022692 -0.009362
4 -0.002592 -0.031991 -0.046641
```

Training Dataset: Inputs

More precisely, an input $\boldsymbol{x}^{(i)} \in \mathcal{X}$ is a d-dimensional vector of the form

$$\boldsymbol{x}^{(i)} = \begin{bmatrix} x_1^{(i)} \\ x_2^{(i)} \\ \vdots \\ x_d^{(i)} \end{bmatrix}$$

For example, it could be the measurements the values of the d features for patient i.

The set $\mathcal{X}$ is called the feature space. Often, we have, $\mathcal{X} = \mathbb{R}^d$.

Let's look at data for one patient.

[10]:
```
diabetes_X.iloc[0]
```

[10]:
```
age    0.038076
sex    0.050680
bmi    0.061696
bp     0.021872
s1    -0.044223
s2    -0.034821
s3    -0.043401
s4    -0.002592
s5     0.019908
s6    -0.017646
Name: 0, dtype: float64
```

Training Dataset: Attributes

We refer to the numerical variables describing the patient as *attributes*. Examples of attributes include:

- The age of a patient.
- The patient's gender.
- The patient's BMI.

Note that thes attributes in the above example have been mean-centered at zero and re-scaled to have a variance of one.

Training Dataset: Features

Often, an input object has many attributes, and we want to use these attributes to define more complex descriptions of the input.

- Is the patient old and a man? (Useful if old men are at risk).
- Is the BMI above the obesity threshold?

We call these custom attributes *features*.

Let's create an "old man" feature.

```
[11]: diabetes_X['old_man'] = (diabetes_X['sex'] > 0) & (diabetes_X['age'] > 0.05)
      diabetes_X.head()
```

```
[11]:        age       sex       bmi        bp        s1        s2        s3  \
      0  0.038076  0.050680  0.061696  0.021872 -0.044223 -0.034821 -0.043401
      1 -0.001882 -0.044642 -0.051474 -0.026328 -0.008449 -0.019163  0.074412
      2  0.085299  0.050680  0.044451 -0.005671 -0.045599 -0.034194 -0.032356
      3 -0.089063 -0.044642 -0.011595 -0.036656  0.012191  0.024991 -0.036038
      4  0.005383 -0.044642 -0.036385  0.021872  0.003935  0.015596  0.008142

               s4        s5        s6  old_man
      0 -0.002592  0.019908 -0.017646    False
      1 -0.039493 -0.068330 -0.092204    False
      2 -0.002592  0.002864 -0.025930     True
      3  0.034309  0.022692 -0.009362    False
      4 -0.002592 -0.031991 -0.046641    False
```

Training Dataset: Features

More formally, we can define a function $\boldsymbol{\phi} : \mathcal{X} \to \mathbb{R}^p$ that takes an input $\boldsymbol{x}^{(i)} \in \mathcal{X}$ and outputs a p-dimensional vector

$$\boldsymbol{\phi}(x^{(i)}) = \begin{bmatrix} \phi_1(x^{(i)}) \\ \phi_2(x^{(i)}) \\ \vdots \\ \phi_p(x^{(i)}) \end{bmatrix}$$

We say that $\boldsymbol{\phi}(x^{(i)})$ is a *featurized* input, and each $\phi_j(x^{(i)})$ is a *feature*.

Features vs Attributes

In practice, the terms attribute and features are often used interchangeably. Most authors refer to $x^{(i)}$ as a vector of features (i.e., they've been precomputed).

We will follow this convention and use attribute only when there is ambiguity between features and attributes.

Features: Discrete vs. Continuous

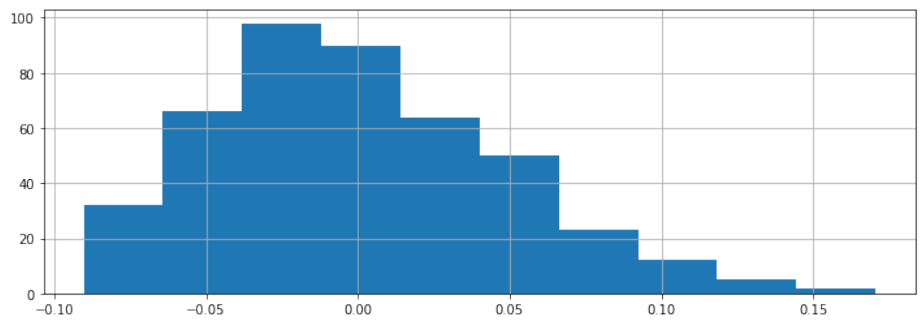
Features can be either discrete or continuous. We will see later that they may be handled differently by ML algorthims.

The BMI feature that we have seen earlier is an example of a continuous feature.

We can visualize its distribution.

```
[12]: diabetes_X.loc[:, 'bmi'].hist()
```

```
[12]: <AxesSubplot:>
```



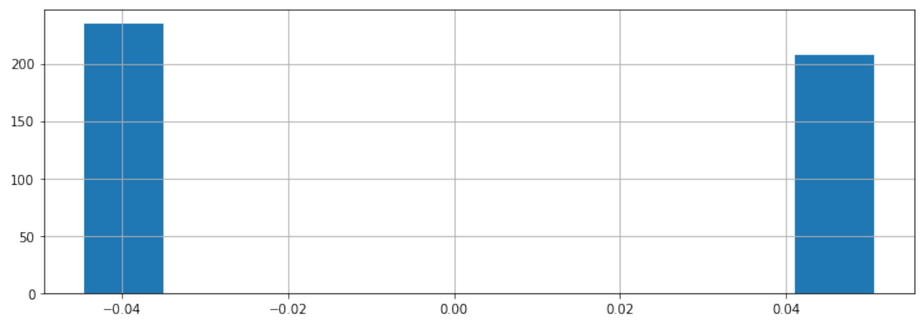
Other features take on one of a finite number of discrete values. The `sex` column is an example of a categorical feature.

In this example, the dataset has been pre-processed such that the two values happen to be `0.05068012` and `-0.04464164`.

```
[13]: print(diabetes_X.loc[:, 'sex'].unique())
      diabetes_X.loc[:, 'sex'].hist()
```

```
[ 0.05068012 -0.04464164]
```

```
[13]: <AxesSubplot:>
```



Training Dataset: Targets

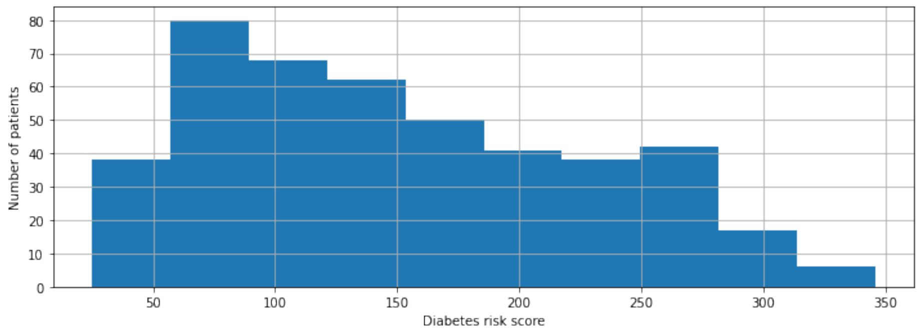
For each patient, we are interested in predicting a quantity of interest, the *target*. In our example, this is the patient's diabetes risk.

Formally, when $(x^{(i)}, y^{(i)})$ form a *training example*, each $y^{(i)} \in \mathcal{Y}$ is a target. We call $\mathcal{Y}$ the target space.

We plot the distirbution of risk scores below.

```
[14]: plt.xlabel('Diabetes risk score')
      plt.ylabel('Number of patients')
      diabetes_y.hist()
```

```
[14]: <AxesSubplot:xlabel='Diabetes risk score', ylabel='Number of patients'>
```



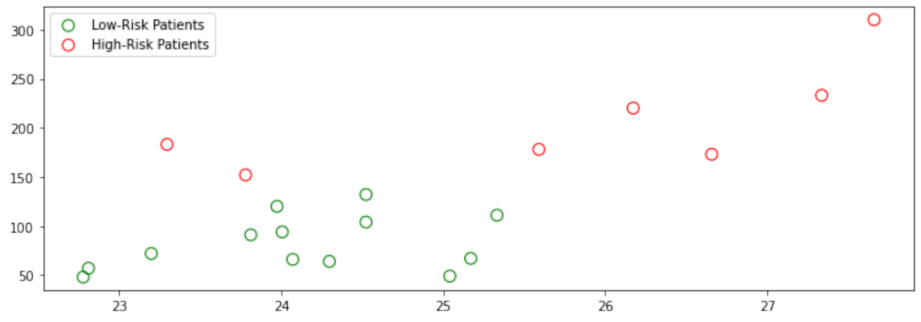
Targets: Regression vs. Classification

We distinguish between two broad types of supervised learning problems that differ in the form of the target variable.

1. **Regression**: The target variable y is continuous. We are fitting a curve in a high-dimensional feature space that approximates the shape of the dataset.
2. **Classification**: The target variable y is discrete. Each discrete value corresponds to a *class* and we are looking for a hyperplane that separates the different classes.

We can easily turn our earlier regression example into classification by discretizing the diabetes risk scores into high or low.

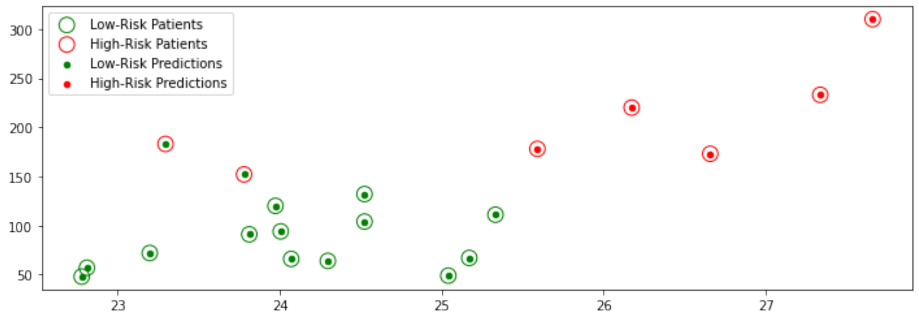
[15]:
```
# Discretize the targets
diabetes_y_train_discr = np.digitize(diabetes_y_train, bins=[150])

# Visualize it
plt.scatter(diabetes_X_train[diabetes_y_train_discr==0],␣
  ↪diabetes_y_train[diabetes_y_train_discr==0], marker='o', s=80,␣
  ↪facecolors='none', edgecolors='g')
plt.scatter(diabetes_X_train[diabetes_y_train_discr==1],␣
  ↪diabetes_y_train[diabetes_y_train_discr==1], marker='o', s=80,␣
  ↪facecolors='none', edgecolors='r')
plt.legend(['Low-Risk Patients', 'High-Risk Patients'])
```



Let's try to generate predictions for this dataset.

[16]:
```
# Create logistic regression object (note: this is actually a classification␣
 ↪algorithm!)
clf = linear_model.LogisticRegression()

# Train the model using the training sets
clf.fit(diabetes_X_train, diabetes_y_train_discr)

# Make predictions on the training set
diabetes_y_train_pred = clf.predict(diabetes_X_train)

# Visualize it
plt.scatter(diabetes_X_train[diabetes_y_train_discr==0],␣
 ↪diabetes_y_train[diabetes_y_train_discr==0], marker='o', s=140,␣
 ↪facecolors='none', edgecolors='g')
plt.scatter(diabetes_X_train[diabetes_y_train_discr==1],␣
 ↪diabetes_y_train[diabetes_y_train_discr==1], marker='o', s=140,␣
 ↪facecolors='none', edgecolors='r')
plt.scatter(diabetes_X_train[diabetes_y_train_pred==0],␣
 ↪diabetes_y_train[diabetes_y_train_pred==0], color='g', s=20)
plt.scatter(diabetes_X_train[diabetes_y_train_pred==1],␣
 ↪diabetes_y_train[diabetes_y_train_pred==1], color='r', s=20)
plt.legend(['Low-Risk Patients', 'High-Risk Patients', 'Low-Risk Predictions',␣
 ↪'High-Risk Predictions'])
```

[16]:
```
<matplotlib.legend.Legend at 0x7f5efe5986d8>
```



Part 3: Anatomy of a Supervised Learning Problem: Learning Algorithm

Let's now look at what a general supervised learning algorithm looks like.

Recall: Three Components of A Supervised Machine Learning Problem

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

$$\text{Dataset} + \text{Algorithm} \rightarrow \text{Predictive Model}$$

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

The Components of A Supervised Machine Learning Algorithm

We can also define the high-level structure of a supervised learning algorithm as consisting of three components:

- A **model class**: the set of possible models we consider.
- An **objective**: function, which defines how good a model is.
- An **optimizer**: which finds the best predictive model in the model class according to the objective function.

Example: given the data set as x = BMI, and y = risk. We assume the model $y^i = f^*\left(x^i\right)$ which is unknown, and we want to approximate f^*. For example, in section 2.1.6., we look for an approximation in the model class { all straight lines } = $\{f : f = \theta_1 x + \theta_0\}$, where θ_0, θ_1 are unknown.

Let's look again at our diabetes dataset for an example.

[17]:
```
import numpy as np
import pandas as pd
from sklearn import datasets
import matplotlib.pyplot as plt
plt.rcParams['figure.figsize'] = [12, 4]

# Load the diabetes dataset
diabetes = datasets.load_diabetes(as_frame=True)
diabetes_X, diabetes_y = diabetes.data, diabetes.target

# Print part of the dataset
diabetes_X.head()
```

[17]:
```
        age       sex       bmi        bp        s1        s2        s3  \
0  0.038076  0.050680  0.061696  0.021872 -0.044223 -0.034821 -0.043401
1 -0.001882 -0.044642 -0.051474 -0.026328 -0.008449 -0.019163  0.074412
2  0.085299  0.050680  0.044451 -0.005671 -0.045599 -0.034194 -0.032356
3 -0.089063 -0.044642 -0.011595 -0.036656  0.012191  0.024991 -0.036038
4  0.005383 -0.044642 -0.036385  0.021872  0.003935  0.015596  0.008142

         s4        s5        s6
0 -0.002592  0.019908 -0.017646
1 -0.039493 -0.068330 -0.092204
2 -0.002592  0.002864 -0.025930
3  0.034309  0.022692 -0.009362
4 -0.002592 -0.031991 -0.046641
```

For this example, we can see

- Objective: measurement of the loss/gain/reward/penalty for each member in the model class.
- Optimizer: algorithm (typically optimization) to determine the optimal member.

Model: Notation

We'll say that a model is a function

$$f : \mathcal{X} \to \mathcal{Y}$$

that maps inputs $\boldsymbol{x} \in \mathcal{X}$ to targets $\boldsymbol{y} \in \mathcal{Y}$.

Often, models have *parameters* $\boldsymbol{\theta} \in \Theta$ living in a set Θ. We will then write the model as

$$f_{\boldsymbol{\theta}} : \mathcal{X} \to \mathcal{Y}$$

to denote that it's parametrized by $\boldsymbol{\theta}$.

Model Class: Notation

Formally, the model class is a set

$$\mathcal{M} \subseteq \{f \mid f : \mathcal{X} \to \mathcal{Y}\}$$

of possible models that map input features to targets.

When the models $f_{\boldsymbol{\theta}}$ are paremetrized by *parameters* $\boldsymbol{\theta} \in \Theta$ living in some set Θ. Thus we can also write

$$\mathcal{M} = \{f_{\boldsymbol{\theta}} \mid f : \mathcal{X} \to \mathcal{Y};\ \boldsymbol{\theta} \in \Theta\}.$$

Model Class: Example

One simple approach is to assume that x and y are related by a linear model of the form

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

where $\boldsymbol{x} = (x_1, ..., x_d)$ is a featurized output and y is the target.

The θ_j are the *parameters* of the model.

Objectives: Notation

To capture this intuition, we define an *objective function* (also called a *loss function*)

$$J(f) : \mathcal{M} \to [0, \infty),$$

which describes the extent to which f "fits" the data $\mathcal{D} = \{(\boldsymbol{x}^{(i)}, \boldsymbol{y}^{(i)}) \mid i = 1, 2, ..., n\}$.

When f is parametrized by $\boldsymbol{\theta} \in \Theta$, the objective becomes a function $J(\boldsymbol{\theta}) : \Theta \to [0, \infty)$. where

$$J(\Theta) := J(f_\theta) = J \circ f(\theta)$$

Objective: Examples

What would are some possible objective functions? We will see many, but here are a few examples: * Mean squared error (L2 loss):

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

* Absolute error (L1 loss):

$$J(\theta) = \frac{1}{n} \sum_{i=1}^{n} \left| f_\theta(x^{(i)}) - y^{(i)} \right|$$

These are defined for a dataset $\mathcal{D} = \{(x^{(i)}, y^{(i)}) \mid i = 1, 2, ..., n\}$.

[18]:
```
from sklearn.metrics import mean_squared_error, mean_absolute_error

y1 = np.array([1, 2, 3, 4])
y2 = np.array([-1, 1, 3, 5])

print('Mean squared error: %.2f' % mean_squared_error(y1, y2))
print('Mean absolute error: %.2f' % mean_absolute_error(y1, y2))
```

```
Mean squared error: 1.50
Mean absolute error: 1.00
```

Optimizer: Notation

At a high-level an optimizer takes an objective J 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.

When f is parametrized by $\theta \in \Theta$, the optimizer minimizes a function $J(\theta)$ over all $\theta \in \Theta$.

Optimizer: Example

We will see that behind the scenes, the `sklearn.linear_models.LinearRegression` algorithm optimizes the MSE loss. Here we consider input x^i with one feature/attribute.

$$\min_{\theta \in \mathbb{R}} \frac{1}{2n} \sum_{i=1}^{n} \left(f_\theta(x^{(i)}) - y^{(i)} \right)^2$$

We can easily measure the quality of the fit on the training set and the test set.

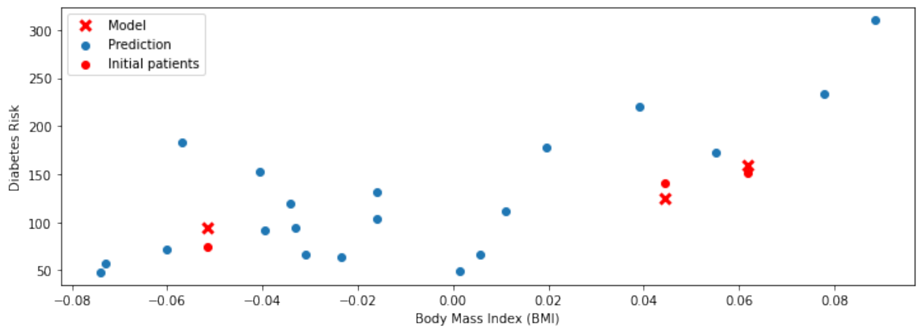
Let's run the above algorithm on our diabetes dataset.

[19]:
```
# Collect 20 data points for training
diabetes_X_train = diabetes_X.iloc[-20:]
diabetes_y_train = diabetes_y.iloc[-20:]

# Create linear regression object
regr = linear_model.LinearRegression()

# Train the model using the training sets
regr.fit(diabetes_X_train, diabetes_y_train.values)

# Make predictions on the training set
diabetes_y_train_pred = regr.predict(diabetes_X_train)

# Collect 3 data points for testing
diabetes_X_test = diabetes_X.iloc[:3]
diabetes_y_test = diabetes_y.iloc[:3]

# generate predictions on the new patients
diabetes_y_test_pred = regr.predict(diabetes_X_test)
```

The algorithm returns a predictive model. We can visualize its predictions below.

[20]:
```
# visualize the results
plt.xlabel('Body Mass Index (BMI)')
plt.ylabel('Diabetes Risk')
plt.scatter(diabetes_X_train.loc[:, ['bmi']], diabetes_y_train)
plt.scatter(diabetes_X_test.loc[:, ['bmi']], diabetes_y_test, color='red',␣
  ↪marker='o')
# plt.scatter(diabetes_X_train.loc[:, ['bmi']], diabetes_y_train_pred,␣
  ↪color='black', linewidth=1)
plt.plot(diabetes_X_test.loc[:, ['bmi']], diabetes_y_test_pred, 'x',␣
  ↪color='red', mew=3, markersize=8)
plt.legend(['Model', 'Prediction', 'Initial patients', 'New patients'])
```

[20]:
```
<matplotlib.legend.Legend at 0x7f5efc41efd0>
```



```
[21]: from sklearn.metrics import mean_squared_error

      print('Training set mean squared error: %.2f'
            % mean_squared_error(diabetes_y_train, diabetes_y_train_pred))
      print('Test set mean squared error: %.2f'
            % mean_squared_error(diabetes_y_test, diabetes_y_test_pred))
      print('Test set mean squared error on random inputs: %.2f'
            % mean_squared_error(diabetes_y_test, np.random.
        ↪randn(*diabetes_y_test_pred.shape)))
```

```
Training set mean squared error: 1144.28
Test set mean squared error: 228.50
Test set mean squared error on random inputs: 15983.78
```

Summary: Components of A Supervised Machine Learning Problem

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

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

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

Notation: Feature Matrix

Suppose that we have a dataset of size n (e.g., n patients), indexed by $i = 1, 2, ..., n$. Each $x^{(i)}$ is a vector of d features.

Feature 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 $X \in \mathbb{R}^{d\times n}$, of the form:

$$X = \begin{bmatrix} x_1^{(1)} & x_1^{(2)} & \ldots & x_1^{(n)} \\ x_2^{(1)} & x_2^{(2)} & \ldots & x_2^{(n)} \\ \vdots \\ x_d^{(1)} & x_d^{(2)} & \ldots & x_d^{(n)} \end{bmatrix}.$$

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

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

CHAPTER 3

Lecture 3: Optimization and Linear Regression

Lecture 3: Optimization and Linear Regression

Adapted from Applied Machine Learning Lecture Notes of Volodymyr Kuleshov, Cornell 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:

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

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 $\mathcal{D} = \{(x^{(i)}, y^{(i)}) \mid i = 1, 2, ..., n\}$.

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

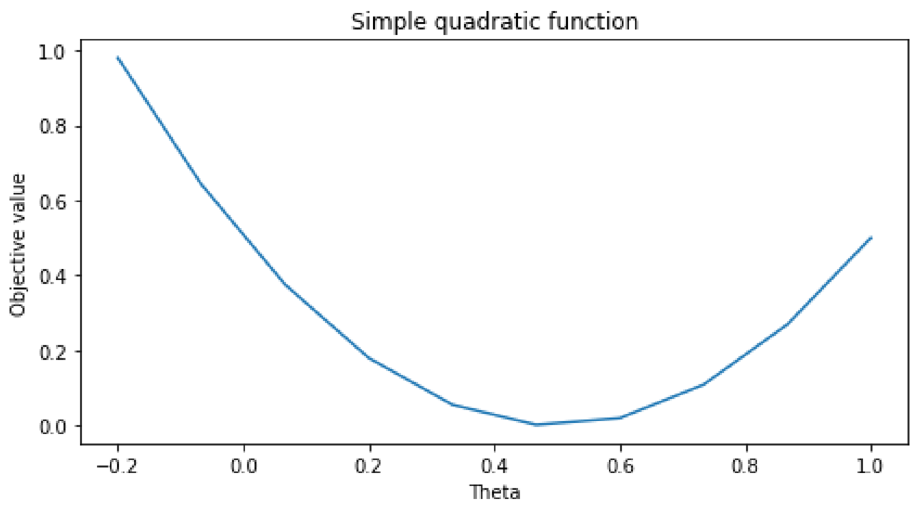
[1]:
```
import numpy as np
import matplotlib.pyplot as plt
plt.rcParams['figure.figsize'] = [8, 4]
```

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

We can visualize it.

[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')
```

[3]:
```
Text(0.5, 1.0, 'Simple quadratic function')
```



Calculus Review: Derivatives

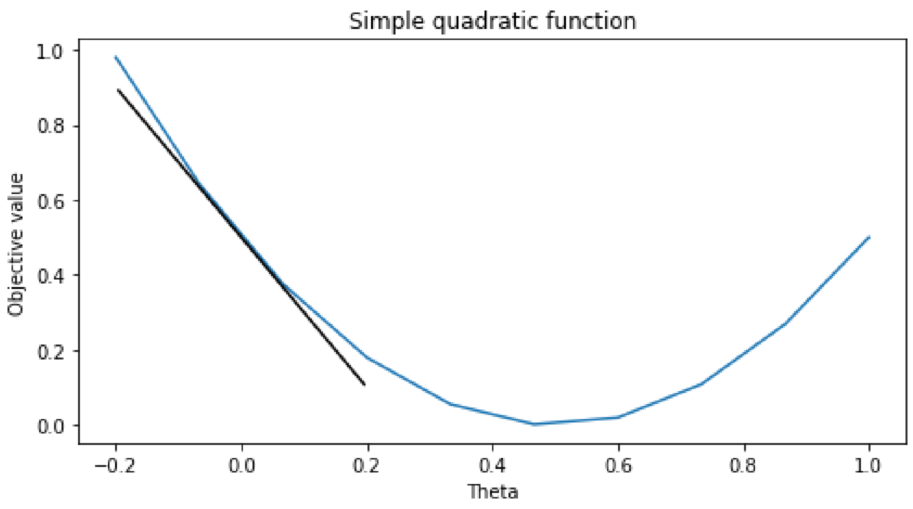
Recall that the derivative

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

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^*.

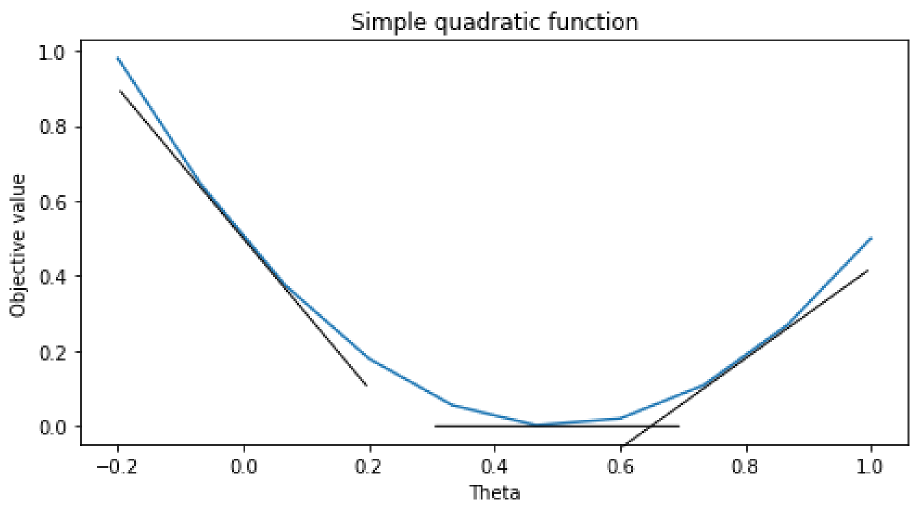
[4]:
```
def quadratic_derivative(theta):
    return (2*theta-1)*2

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')
```

[4]:
```
Text(0.5, 1.0, 'Simple quadratic function')
```



[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')
```

[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.

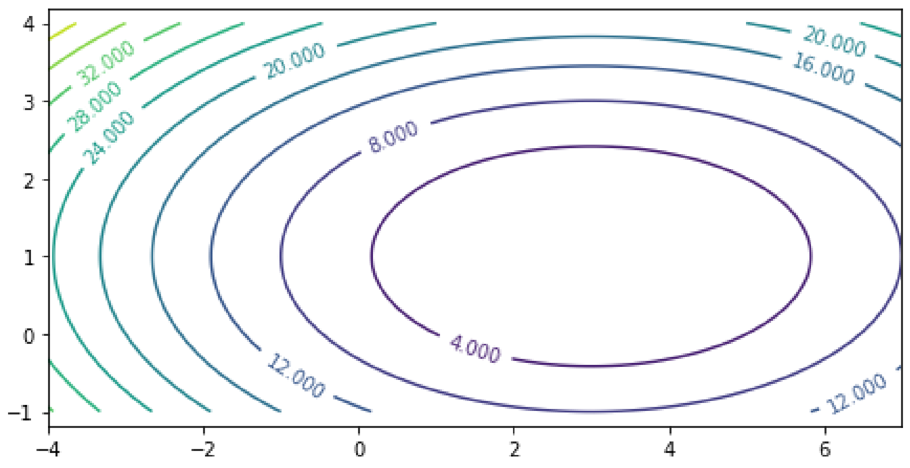
We will use a quadratic function as a running example.

```
[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.

```
[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')
```

```
[7]: (-4.0, 7.0, -1.0, 4.0)
```



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

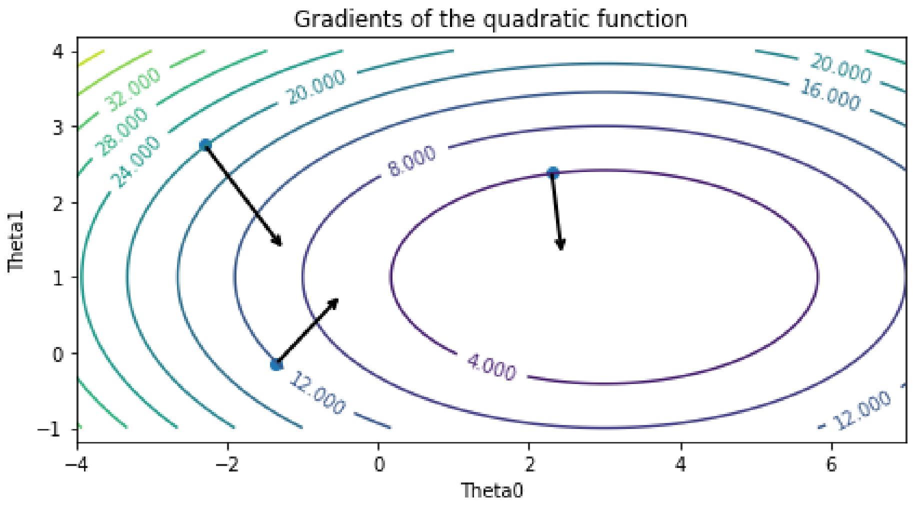
[8]:
```
def quadratic_derivative2d(theta0, theta1):
    """Derivative of quadratic objective function.

    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.

[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')
```

[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

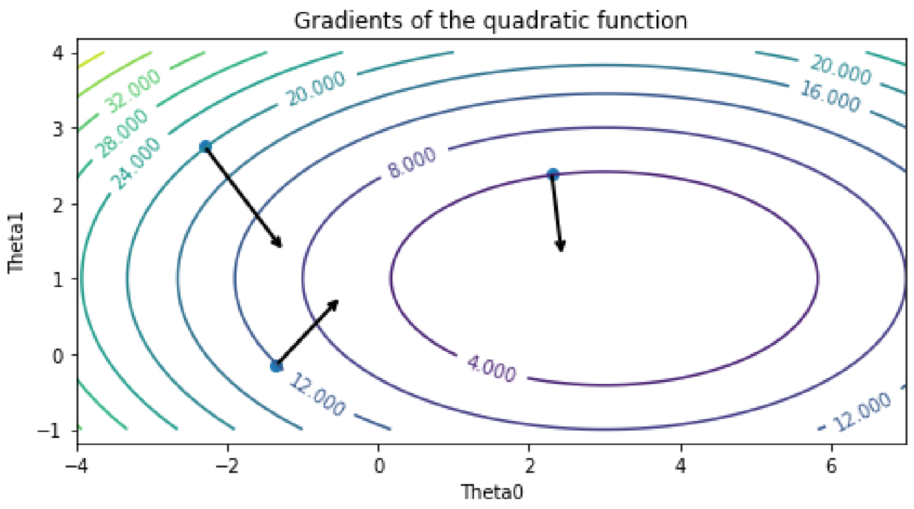
The gradient vector $\nabla_\theta f$ is the generalization of the derivative to multivariate functions $f : \mathbb{R}^d \to \mathbb{R}$, and is defined at an arbitrary parameter vector θ^* as

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

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.

```
[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')
```

```
[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

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

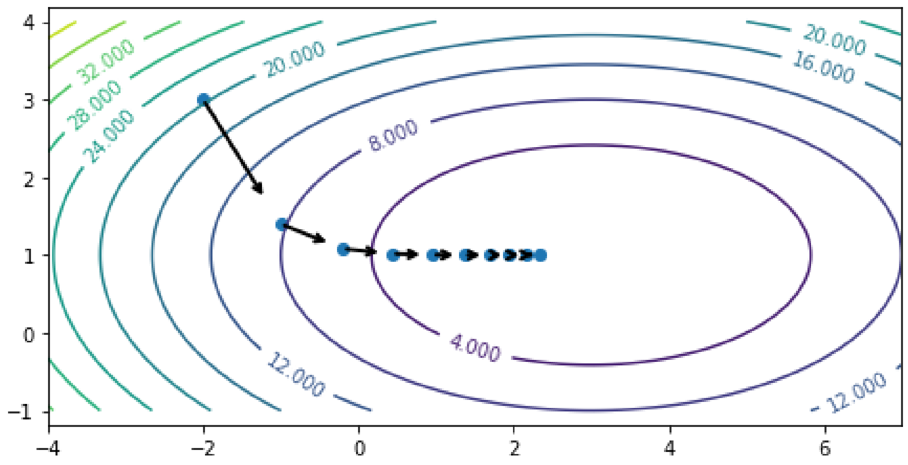
It's easy to implement this function in numpy.

[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.

[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')
```

[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

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

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 $x = (x_0, x_1, ..., x_d)^\top$ together with $\theta = (\theta_0, \theta_1, ..., \theta_d)^\top$ 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.

[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.

Let's implement mean squared error.

```
[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).

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

Mean Squared Error: The Gradient

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

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

Let's implement the gradient.

[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

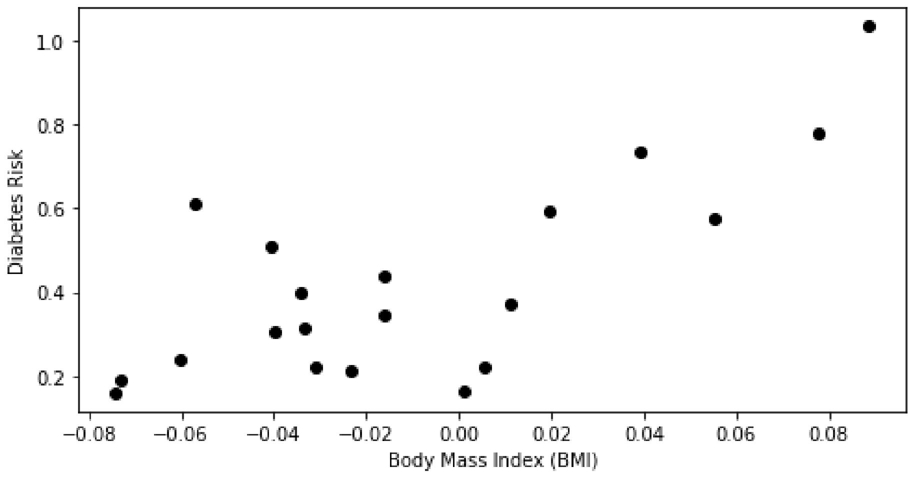
In this section, we are going to again use the UCI Diabetes Dataset.

- 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.

[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')
```

[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
```

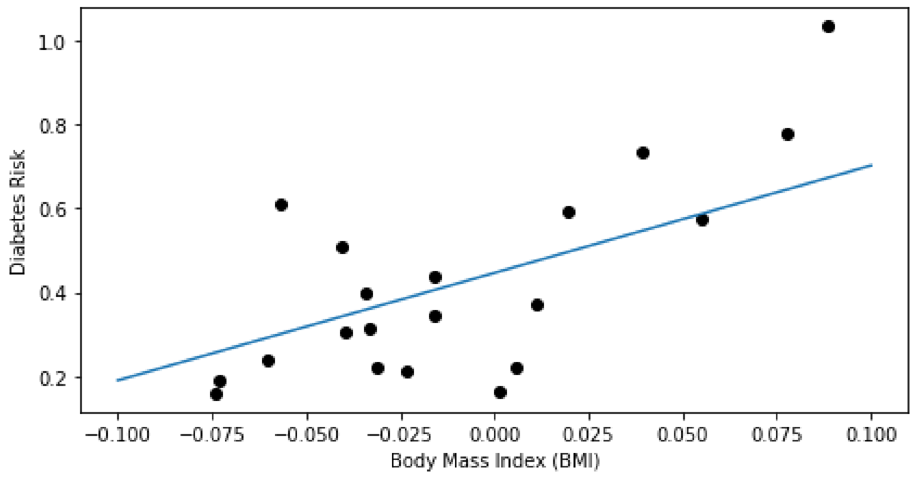
This update rule is also known as the Least Mean Squares (LMS) or Widrow-Hoff learning rule.

[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
```

[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')
```

[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.

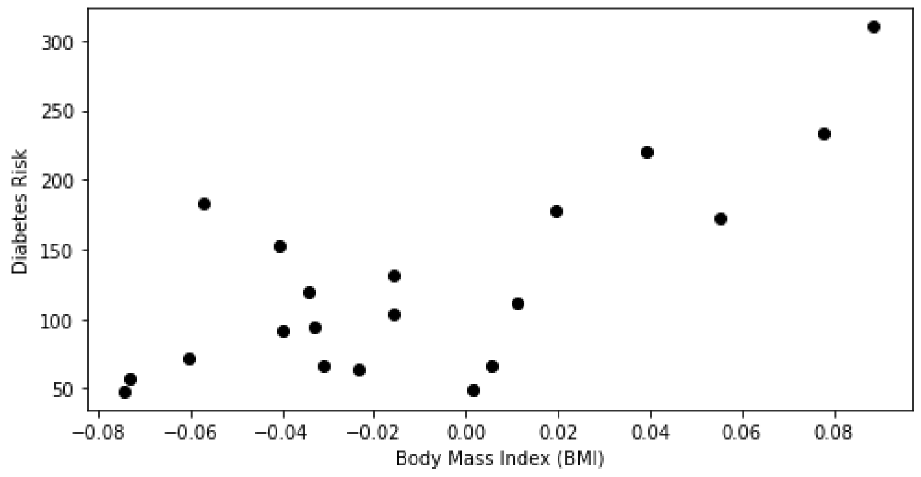
The UCI Diabetes Dataset

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.

[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')
```

[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:

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

We can view the design matrix for the diabetes dataset.

[20]:
```
X_train.head()
```

[20]:
```
          age       sex       bmi        bp        s1        s2        s3  \
422 -0.078165  0.050680  0.077863  0.052858  0.078236  0.064447  0.026550
423  0.009016  0.050680 -0.039618  0.028758  0.038334  0.073529 -0.072854
424  0.001751  0.050680  0.011039 -0.019442 -0.016704 -0.003819 -0.047082
425 -0.078165 -0.044642 -0.040696 -0.081414 -0.100638 -0.112795  0.022869
426  0.030811  0.050680 -0.034229  0.043677  0.057597  0.068831 -0.032356

           s4        s5        s6  one
422 -0.002592  0.040672 -0.009362    1
423  0.108111  0.015567 -0.046641    1
424  0.034309  0.024053  0.023775    1
425 -0.076395 -0.020289 -0.050783    1
426  0.057557  0.035462  0.085907    1
```

Notation: Design Matrix

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

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

Squared Error in Matrix Form

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

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

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:

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

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

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

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 $\nabla_x x^\top Ax = 2Ax$ for a symmetric matrix A (line 4).

Normal Equations

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

$$(X^\top X)\theta = X^\top y.$$

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

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

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.

Let's apply the normal equations.

```
[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
```

```
[21]:        age         sex        bmi          bp            s1            s2  \
      0 -3.888868  204.648785 -64.289163 -262.796691  14003.726808 -11798.307781

                 s3           s4           s5          s6         one
      0 -5892.15807 -1136.947646 -2736.597108 -393.879743  155.698998
```

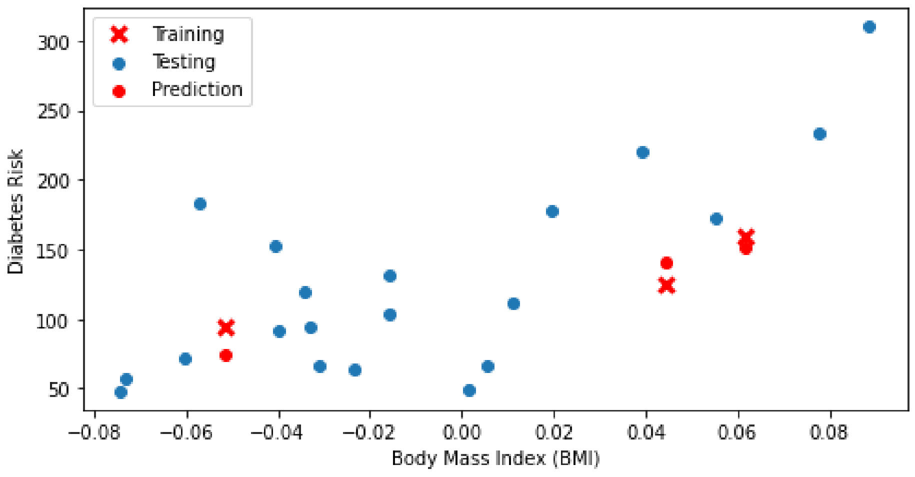
We can now use our estimate of theta to compute predictions for 3 new data points.

```
[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.

```
[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'])
```

```
[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

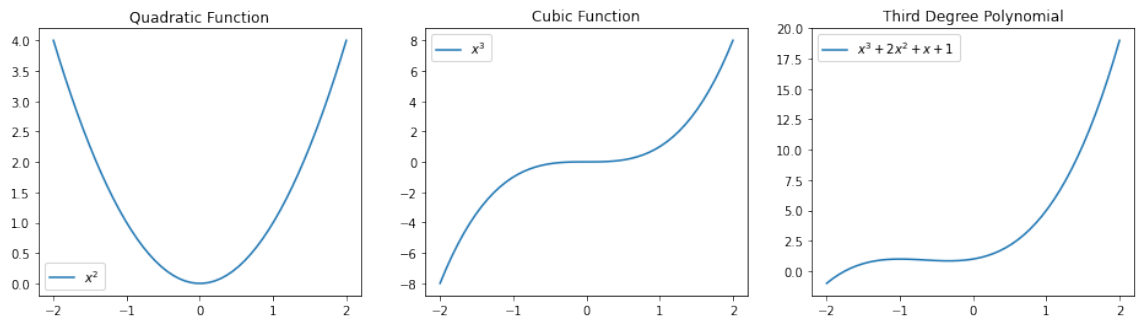
Recall that a polynomial of degree p is a function of the form

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

Below are some examples of polynomial functions.

[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(["$x^3 + 2 x^2 + x + 1$"])
```

```
[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.

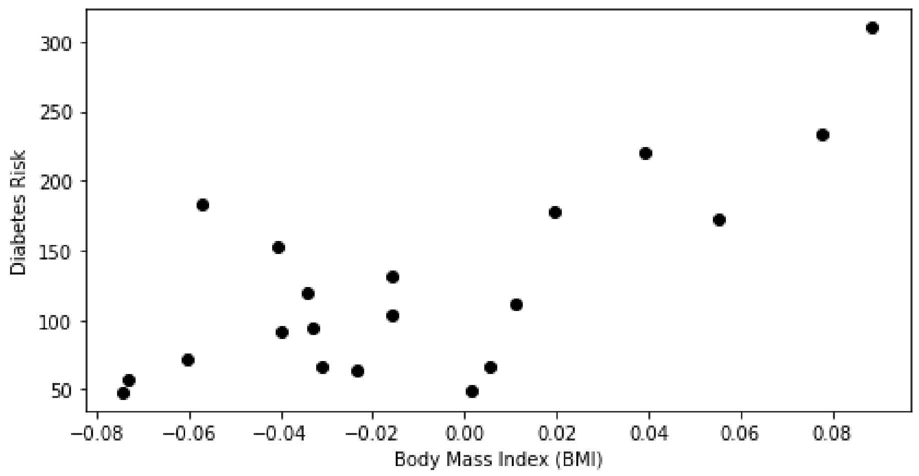
The UCI Diabetes Dataset

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.

[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')
```

[25]:
```
Text(0, 0.5, 'Diabetes Risk')
```



Diabetes Dataset: A Non-Linear Featurization

Let's now obtain linear features for this dataset.

[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()
```

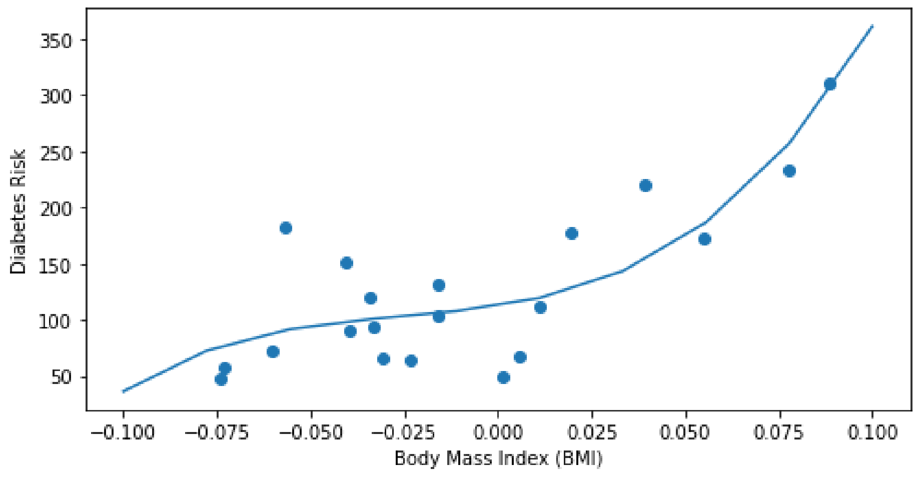
[26]:
```
          bmi      bmi2      bmi3  one
422  0.077863  0.006063  0.000472    1
423 -0.039618  0.001570 -0.000062    1
424  0.011039  0.000122  0.000001    1
425 -0.040696  0.001656 -0.000067    1
426 -0.034229  0.001172 -0.000040    1
```

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.

[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)
```

[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

$$\phi(x) = \begin{bmatrix} 1 \\ x_1 \\ x_1^2 \\ x_2 \\ x_2^2 \\ x_1x_2 \end{bmatrix}.$$

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

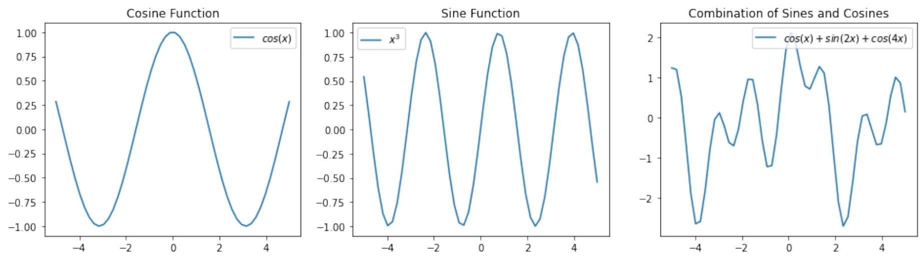
$$f_\theta(x) := \theta^\top \phi(x)$$

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.

[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(["$cos(x) + sin(2x) + cos(4x)$"])
```

[28]:
```
<matplotlib.legend.Legend at 0x7f44fd4bd6d8>
```



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