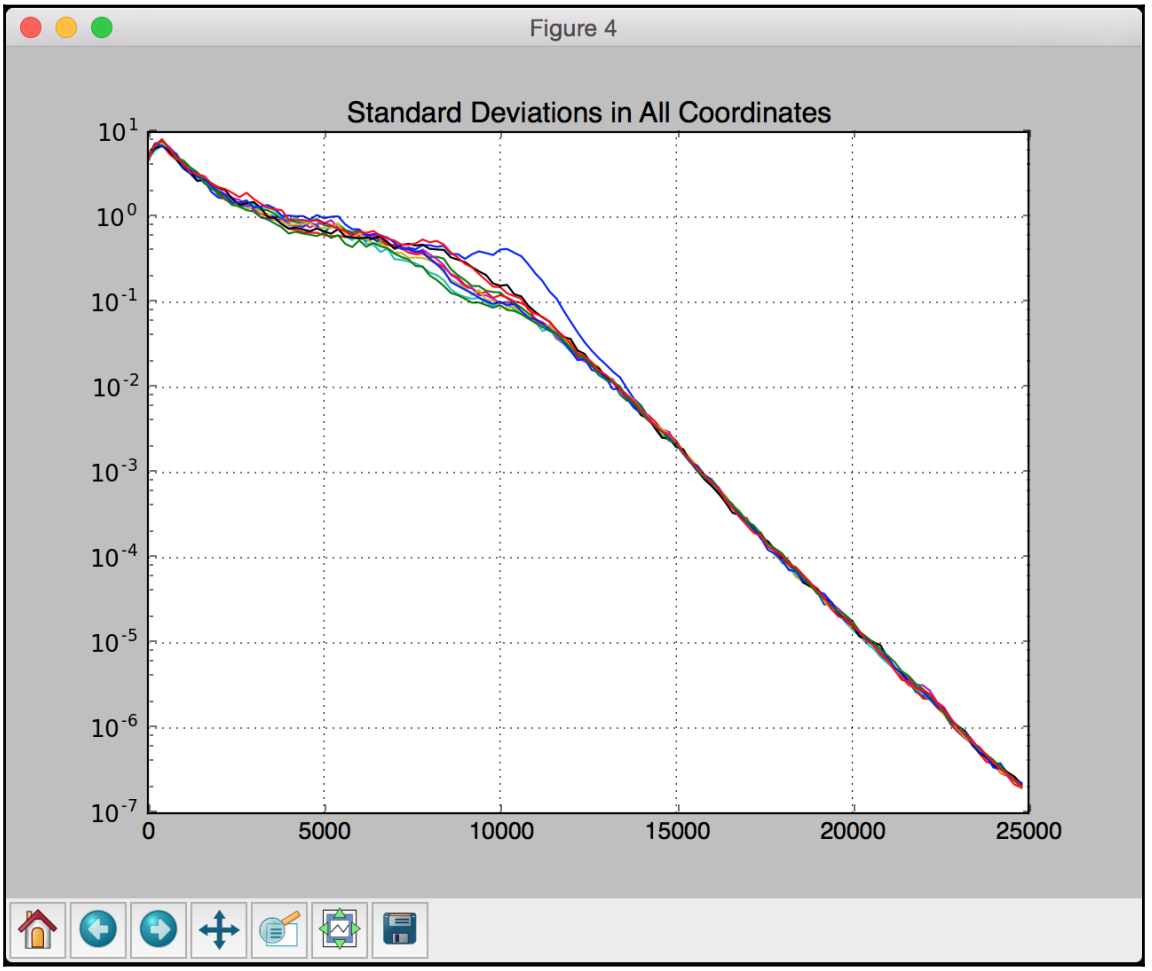
The fourth screenshot shows standard deviations:



You will see the progress printed on the Terminal. At the start, you will see something like the following:

```
gen     evals   std     min     avg     max
0       200     188.36  217.082 576.281 1199.71
1       200     250.543 196.583 659.389 1869.02
2       200     273.081 199.455 683.641 1770.65
3       200     215.326 111.298 503.933 1579.3
4       200     133.046 149.47  373.124 790.899
5       200     75.4405 131.117 274.092 585.433
6       200     61.2622 91.7121 232.624 426.666
7       200     49.8303 88.8185 201.117 373.543
8       200     39.9533 85.0531 178.645 326.209
9       200     31.3781 87.4824 159.211 261.132
10      200     31.3488 54.0743 144.561 274.877
11      200     30.8796 63.6032 136.791 240.739
12      200     24.1975 70.4913 125.691 190.684
13      200     21.2274 50.6409 122.293 177.483
14      200     25.4931 67.9873 124.132 199.296
15      200     26.9804 46.3411 119.295 205.331
16      200     24.8993 56.0033 115.614 176.702
17      200     21.9789 61.4999 113.417 170.156
18      200     21.2823 50.2455 112.419 190.677
19      200     22.5016 48.153  111.543 166.2
20      200     21.1602 32.1864 106.044 171.899
21      200     23.3864 52.8601 107.301 163.617
22      200     23.1008 51.1226 109.628 185.777
23      200     22.0836 51.3058 106.402 179.673
```

At the end, you will see the following:

```
100     200     2.38865e-07     1.12678e-07     5.18814e-07     1.23527e-06
101     200     1.49444e-07     5.56979e-08     3.3199e-07      7.98774e-07
102     200     1.11635e-07     2.07109e-08     2.41361e-07     7.96738e-07
103     200     9.50257e-08     3.69117e-08     1.94641e-07     5.75896e-07
104     200     5.63849e-08     2.09827e-08     1.26148e-07     2.887e-07
105     200     4.42488e-08     1.64212e-08     8.6972e-08      2.58639e-07
106     200     2.34933e-08     1.28302e-08     5.47789e-08     1.54658e-07
107     200     1.74434e-08     7.13185e-09     3.64705e-08     9.88235e-08
108     200     1.17157e-08     6.32208e-09     2.54673e-08     7.13075e-08
109     200     8.73027e-09     4.60369e-09     1.79681e-08     5.88066e-08
110     200     6.39874e-09     1.92573e-09     1.43229e-08     4.00087e-08
111     200     5.31196e-09     2.05551e-09     1.13736e-08     3.16793e-08
112     200     3.15607e-09     1.72427e-09     7.28548e-09     1.67727e-08
113     200     2.3789e-09      1.01164e-09     5.01177e-09     1.24541e-08
114     200     1.38424e-09     6.43112e-10     2.94696e-09     9.25819e-09
115     200     1.04172e-09     2.87571e-10     2.06068e-09     7.90436e-09
116     200     6.08685e-10     4.32905e-10     1.4704e-09      3.80221e-09
117     200     4.51515e-10     2.1538e-10      9.23627e-10     2.2759e-09
118     200     2.77204e-10     1.46869e-10     6.3507e-10      1.44637e-09
119     200     2.06475e-10     7.54881e-11     4.41427e-10     1.33167e-09
120     200     1.3138e-10      5.97282e-11     2.98116e-10     8.60453e-10
121     200     9.52385e-11     6.753e-11       2.32358e-10     5.45441e-10
122     200     7.55001e-11     4.1851e-11      1.72688e-10     5.05054e-10
123     200     5.52125e-11     3.2216e-11      1.23505e-10     3.10081e-10
124     200     4.38068e-11     1.32871e-11     8.94929e-11     2.57202e-10
```

As seen from the preceding figure, the values keep decreasing as we progress. This indicates that it's converging.

Solving the symbol regression problem

Let's see how to use genetic programming to solve the symbol regression problem. It is important to understand that genetic programming is not the same as genetic algorithms. Genetic programming is a type of evolutionary algorithm in which the solutions occur in the form of computer programs. Basically, the individuals in each generation would be computer programs and their fitness level corresponds to their ability to solve problems. These programs are modified, at each iteration, using a genetic algorithm. In essence, genetic programming is the application of a genetic algorithm.

Coming to the symbol regression problem, we have a polynomial expression that needs to be approximated here. It's a classic regression problem where we try to estimate the underlying function. In this example, we will use the expression: $f(x) = 2x^3 - 3x^2 + 4x - 1$

The code discussed here is a variant of the symbol regression problem given in the `DEAP` library. Create a new Python file and import the following:

```
import operator
import math
import random

import numpy as np
from deap import algorithms, base, creator, tools, gp
```

Create a division operator that can handle divide-by-zero error gracefully:

```
# Define new functions
def division_operator(numerator, denominator):
    if denominator == 0:
        return 1

    return numerator / denominator
```

Define the evaluation function that will be used for fitness calculation. We need to define a callable function to run computations on the input individual:

```
# Define the evaluation function
def eval_func(individual, points):
    # Transform the tree expression in a callable function
    func = toolbox.compile(expr=individual)
```

Compute the **mean squared error** (**MSE**) between the function defined earlier and the original expression:

```
    # Evaluate the mean squared error
    mse = ((func(x) - (2 * x**3 - 3 * x**2 - 4 * x + 1))**2 for x in
points)

    return math.fsum(mse) / len(points),
```

Define a function to create the toolbox. In order to create the toolbox here, we need to create a set of primitives. These primitives are basically operators that will be used during the evolution. They serve as building blocks for the individuals. We are going to use basic arithmetic functions as our primitives here:

```
# Function to create the toolbox
def create_toolbox():
    pset = gp.PrimitiveSet("MAIN", 1)
    pset.addPrimitive(operator.add, 2)
    pset.addPrimitive(operator.sub, 2)
    pset.addPrimitive(operator.mul, 2)
    pset.addPrimitive(division_operator, 2)
    pset.addPrimitive(operator.neg, 1)
    pset.addPrimitive(math.cos, 1)
    pset.addPrimitive(math.sin, 1)
```

We now need to declare an ephemeral constant. It is a special terminal type that does not have a fixed value. When a given program appends such an ephemeral constant to the tree, the function gets executed. The result is then inserted into the tree as a constant terminal. These constant terminals can take the values `-1`, `0` or `1`:

```
    pset.addEphemeralConstant("rand101", lambda: random.randint(-1,1))
```

The default name for the arguments is `ARGx`. Let's rename it `x`. It's not exactly necessary, but it's a useful feature that comes in handy:

```
    pset.renameArguments(ARG0='x')
```

We need to define two object types – fitness and an individual. Let's do it using the `creator`:

```
    creator.create("FitnessMin", base.Fitness, weights=(-1.0,))
    creator.create("Individual", gp.PrimitiveTree,
fitness=creator.FitnessMin)
```

Create the `toolbox` and `register` the functions. The registration process is similar to previous sections:

```
    toolbox = base.Toolbox()

    toolbox.register("expr", gp.genHalfAndHalf, pset=pset, min_=1, max_=2)
    toolbox.register("individual", tools.initIterate, creator.Individual,
toolbox.expr)
    toolbox.register("population", tools.initRepeat, list,
toolbox.individual)
    toolbox.register("compile", gp.compile, pset=pset)
    toolbox.register("evaluate", eval_func, points=[x/10. for x in
```

```
range(-10,10)])
    toolbox.register("select", tools.selTournament, tournsize=3)
    toolbox.register("mate", gp.cxOnePoint)
    toolbox.register("expr_mut", gp.genFull, min_=0, max_=2)
    toolbox.register("mutate", gp.mutUniform, expr=toolbox.expr_mut,
pset=pset)

    toolbox.decorate("mate",
gp.staticLimit(key=operator.attrgetter("height"), max_value=17))
    toolbox.decorate("mutate",
gp.staticLimit(key=operator.attrgetter("height"), max_value=17))

    return toolbox
```

Define the `main` function and start by seeding the random number generator:

```
if __name__ == "__main__":
    random.seed(7)
```

Create the `toolbox` object:

```
    toolbox = create_toolbox()
```

Define the initial population using the method available in the `toolbox` object. We will use `450` individuals. The user defines this number, so you should feel free to experiment with it. Also define the `hall_of_fame` object:

```
    population = toolbox.population(n=450)
    hall_of_fame = tools.HallOfFame(1)
```

Statistics are useful when we build genetic algorithms. Define the stats objects:

```
    stats_fit = tools.Statistics(lambda x: x.fitness.values)
    stats_size = tools.Statistics(len)
```

Register the stats using the objects defined previously:

```
    mstats = tools.MultiStatistics(fitness=stats_fit, size=stats_size)
    mstats.register("avg", np.mean)
    mstats.register("std", np.std)
    mstats.register("min", np.min)
    mstats.register("max", np.max)
```

Define the crossover probability, mutation probability, and the number of generations:

```
    probab_crossover = 0.4
    probab_mutate = 0.2
    num_generations = 60
```

Run the evolutionary algorithm using the above parameters:

```
population, log = algorithms.eaSimple(population, toolbox,
        probab_crossover, probab_mutate, num_generations,
        stats=mstats, halloffame=hall_of_fame, verbose=True)
```

The full code is given in the file `symbol_regression.py`. If you run the code, you will see the following on your Terminal at the start of the evolution:

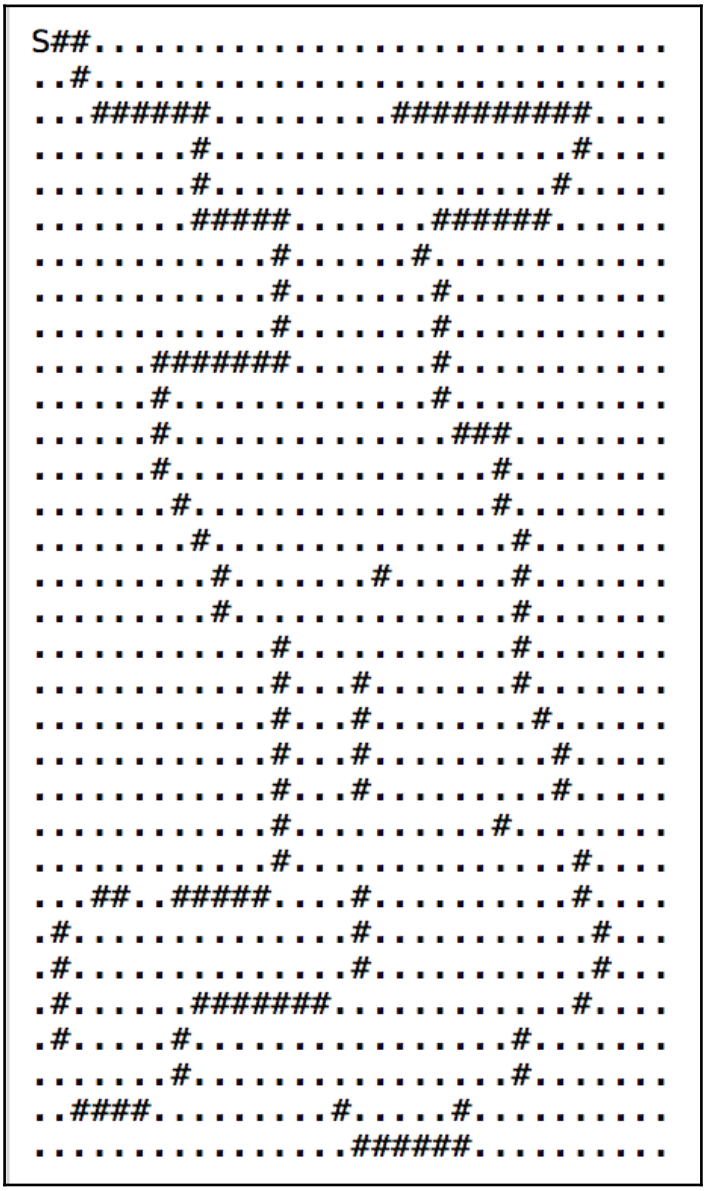
		fitness				size			
gen	nevals	avg	max	min	std	avg	max	min	std
0	450	18.6918	47.1923	7.39087	6.27543	3.73556	7	2	1.62449
1	251	15.4572	41.3823	4.46965	4.54993	3.80222	12	1	1.81316
2	236	13.2545	37.7223	4.46965	4.06145	3.96889	12	1	1.98861
3	251	12.2299	60.828	4.46965	4.70055	4.19556	12	1	1.9971
4	235	11.001	47.1923	4.46965	4.48841	4.84222	13	1	2.17245
5	229	9.44483	31.478	4.46965	3.8796	5.56	19	1	2.43168
6	225	8.35975	22.0546	3.02133	3.40547	6.38889	15	1	2.40875
7	237	7.99309	31.1356	1.81133	4.08463	7.14667	16	1	2.57782
8	224	7.42611	359.418	1.17558	17.0167	8.33333	19	1	3.11127
9	237	5.70308	24.1921	1.17558	3.71991	9.64444	23	1	3.31365
10	254	5.27991	30.4315	1.13301	4.13556	10.5089	25	1	3.51898

At the end, you will see the following:

36	209	1.10464	22.0546	0.0474957	2.71898	26.4867	46	1	5.23289
37	258	1.61958	86.0936	0.0382386	6.1839	27.2111	45	3	4.75557
38	257	2.03651	70.4768	0.0342642	5.15243	26.5311	49	1	6.22327
39	235	1.95531	185.328	0.0472693	9.32516	26.9711	48	1	6.00345
40	234	1.51403	28.5529	0.0472693	3.24513	26.6867	52	1	5.39811
41	230	1.4753	70.4768	0.0472693	5.4607	27.1	46	3	4.7433
42	233	12.3648	4880.09	0.0396503	229.754	26.88	53	1	5.18192
43	251	1.807	86.0936	0.0396503	5.85281	26.4889	50	1	5.43741
44	236	9.30096	3481.25	0.0277886	163.888	26.9622	55	1	6.27169
45	231	1.73196	86.7372	0.0342642	6.8119	27.4711	51	2	5.27807
46	227	1.86086	185.328	0.0342642	10.1143	28.0644	56	1	6.10812
47	216	12.5214	4923.66	0.0342642	231.837	29.1022	54	1	6.45898
48	232	14.3469	5830.89	0.0322462	274.536	29.8244	58	3	6.24093
49	242	2.56984	272.833	0.0322462	18.2752	29.9267	51	1	6.31446
50	227	2.80136	356.613	0.0322462	21.0416	29.7978	56	4	6.50275
51	243	1.75099	86.0936	0.0322462	5.70833	29.8089	56	1	6.62379
52	253	10.9184	3435.84	0.0227048	163.602	29.9911	55	1	6.66833
53	243	1.80265	48.0418	0.0227048	4.73856	29.88	55	1	7.33084
54	234	1.74487	86.0936	0.0227048	6.0249	30.6067	55	1	6.85782
55	220	1.58888	31.094	0.0132398	3.82809	30.5644	54	1	6.96669
56	234	1.46711	103.287	0.00766444	6.81157	30.6689	55	3	6.6806
57	250	17.0896	6544.17	0.00424267	308.689	31.1267	60	4	7.25837
58	231	1.66757	141.584	0.00144401	7.35306	32	52	1	7.23295
59	229	2.22325	265.224	0.00144401	13.388	33.5489	64	1	8.38351
60	248	2.60303	521.804	0.00144401	24.7018	35.2533	58	1	7.61506

Building an intelligent robot controller

Let's see how to build a robot controller using a genetic algorithm. We are given a map with the targets sprinkled all over it. The map looks like this:



There are 124 targets in the preceding map. The goal of the robot controller is to automatically traverse the map and consume all those targets. This program is a variant of the artificial ant program given in the `deap` library.

Create a new Python file and import the following:

```
import copy
import random
from functools import partial

import numpy as np
from deap import algorithms, base, creator, tools, gp
```

Create the class to control the robot:

```
class RobotController(object):
    def __init__(self, max_moves):
        self.max_moves = max_moves
        self.moves = 0
        self.consumed = 0
        self.routine = None
```

Define the directions and movements:

```
        self.direction = ["north", "east", "south", "west"]
        self.direction_row = [1, 0, -1, 0]
        self.direction_col = [0, 1, 0, -1]
```

Define the reset functionality:

```
    def _reset(self):
        self.row = self.row_start
        self.col = self.col_start
        self.direction = 1
        self.moves = 0
        self.consumed = 0
        self.matrix_exc = copy.deepcopy(self.matrix)
```

Define the conditional operator:

```
    def _conditional(self, condition, out1, out2):
        out1() if condition() else out2()
```

Define the left turning operator:

```
    def turn_left(self):
        if self.moves < self.max_moves:
            self.moves += 1
            self.direction = (self.direction - 1) % 4
```

Define the right turning operator:

```
    def turn_right(self):
        if self.moves < self.max_moves:
            self.moves += 1
            self.direction = (self.direction + 1) % 4
```

Define the method to control how the robot moves forward:

```
    def move_forward(self):
        if self.moves < self.max_moves:
            self.moves += 1
            self.row = (self.row + self.direction_row[self.direction]) %
                   self.matrix_row
            self.col = (self.col + self.direction_col[self.direction]) %
                   self.matrix_col

            if self.matrix_exc[self.row][self.col] == "target":
                self.consumed += 1

            self.matrix_exc[self.row][self.col] = "passed"
```

Define a method to sense the target. If you see the target ahead, then update the matrix accordingly:

```
    def sense_target(self):
        ahead_row = (self.row + self.direction_row[self.direction]) %
self.matrix_row
        ahead_col = (self.col + self.direction_col[self.direction]) %
self.matrix_col
        return self.matrix_exc[ahead_row][ahead_col] == "target"
```

If you see the target ahead, then create the relevant function and return it:

```
    def if_target_ahead(self, out1, out2):
        return partial(self._conditional, self.sense_target, out1, out2)
```

Define the method to run it:

```
    def run(self,routine):
        self._reset()
        while self.moves < self.max_moves:
            routine()
```

Define a function to traverse the input map. The symbol `#` indicates all the targets on the map and the symbol `S` indicates the starting point. The symbol `.` denotes empty cells:

```
    def traverse_map(self, matrix):
        self.matrix = list()
        for i, line in enumerate(matrix):
            self.matrix.append(list())

            for j, col in enumerate(line):
                if col == "#":
                    self.matrix[-1].append("target")

                elif col == ".":
                    self.matrix[-1].append("empty")

                elif col == "S":
                    self.matrix[-1].append("empty")
                    self.row_start = self.row = i
                    self.col_start = self.col = j
                    self.direction = 1

        self.matrix_row = len(self.matrix)
        self.matrix_col = len(self.matrix[0])
        self.matrix_exc = copy.deepcopy(self.matrix)
```

Define a class to generate functions depending on the number of input arguments:

```
class Prog(object):
    def _progn(self, *args):
        for arg in args:
            arg()

    def prog2(self, out1, out2):
        return partial(self._progn, out1, out2)

    def prog3(self, out1, out2, out3):
        return partial(self._progn, out1, out2, out3)
```

Define an evaluation function for each individual:

```
def eval_func(individual):
    global robot, pset

    # Transform the tree expression to functional Python code
    routine = gp.compile(individual, pset)
```

Run the current program:

```
    # Run the generated routine
    robot.run(routine)
    return robot.consumed,
```

Define a function to create the toolbox and add primitives:

```
def create_toolbox():
    global robot, pset

    pset = gp.PrimitiveSet("MAIN", 0)
    pset.addPrimitive(robot.if_target_ahead, 2)
    pset.addPrimitive(Prog().prog2, 2)
    pset.addPrimitive(Prog().prog3, 3)
    pset.addTerminal(robot.move_forward)
    pset.addTerminal(robot.turn_left)
    pset.addTerminal(robot.turn_right)
```

Create the object types using the fitness function:

```
    creator.create("FitnessMax", base.Fitness, weights=(1.0,))
    creator.create("Individual", gp.PrimitiveTree,
fitness=creator.FitnessMax)
```

Create the `toolbox` and `register` all the operators:

```
    toolbox = base.Toolbox()

    # Attribute generator
    toolbox.register("expr_init", gp.genFull, pset=pset, min_=1, max_=2)

    # Structure initializers
    toolbox.register("individual", tools.initIterate, creator.Individual,
toolbox.expr_init)
    toolbox.register("population", tools.initRepeat, list,
toolbox.individual)

    toolbox.register("evaluate", eval_func)
    toolbox.register("select", tools.selTournament, tournsize=7)
    toolbox.register("mate", gp.cxOnePoint)
```

```
    toolbox.register("expr_mut", gp.genFull, min_=0, max_=2)
    toolbox.register("mutate", gp.mutUniform, expr=toolbox.expr_mut,
pset=pset)

    return toolbox
```

Define the `main` function and start by seeding the random number generator:

```
if __name__ == "__main__":
    global robot

    # Seed the random number generator
    random.seed(7)
```

Create the robot controller object using the initialization parameter:

```
    # Define the maximum number of moves
    max_moves = 750

    # Create the robot object
    robot = RobotController(max_moves)
```

Create the `toolbox` using the function we defined earlier:

```
    # Create the toolbox
    toolbox = create_toolbox()
```

Read the map data from the input file:

```
    # Read the map data
    with open('target_map.txt', 'r') as f:
      robot.traverse_map(f)
```

Define the population with `400` individuals and define the `hall_of_fame` object:

```
    # Define population and hall of fame variables
    population = toolbox.population(n=400)
    hall_of_fame = tools.HallOfFame(1)
```

Register the stats:

```
    # Register the stats
    stats = tools.Statistics(lambda x: x.fitness.values)
    stats.register("avg", np.mean)
    stats.register("std", np.std)
    stats.register("min", np.min)
    stats.register("max", np.max)
```

Define the crossover probability, mutation probability, and the number of generations:

```
# Define parameters
probab_crossover = 0.4
probab_mutate = 0.3
num_generations = 50
```

Run the evolutionary algorithm using the parameters defined earlier:

```
# Run the algorithm to solve the problem
algorithms.eaSimple(population, toolbox, probab_crossover,
        probab_mutate, num_generations, stats,
        halloffame=hall_of_fame)
```

The full code is given in the file `robot.py`. If you run the code, you will get the following on your Terminal:

```
gen     nevals  avg     std     min     max
0       400     1.4875  4.37491 0       62
1       231     4.285   7.56993 0       73
2       235     10.8925 14.8493 0       73
3       231     21.72   22.1239 0       73
4       238     29.9775 27.7861 0       76
5       224     37.6275 31.8698 0       76
6       231     42.845  33.0541 0       80
7       223     43.55   33.9369 0       83
8       234     44.0675 34.5201 0       83
9       231     49.2975 34.3065 0       83
10      249     47.075  36.4106 0       93
11      222     52.7925 36.2826 0       97
12      248     51.0725 37.2598 0       97
13      234     54.01   37.4614 0       97
14      229     59.615  37.7894 0       97
15      228     63.3    39.8205 0       97
16      220     64.605  40.3962 0       97
17      236     62.545  40.5607 0       97
18      233     67.99   38.9033 0       97
19      236     66.4025 39.6574 0       97
20      221     69.785  38.7117 0       97
21      244     65.705  39.0957 0       97
22      230     70.32   37.1206 0       97
23      241     67.3825 39.4028 0       97
```

Towards the end, you will see the following:

```
26      214     71.505  36.964  0       97
27      246     72.72   37.1637 0       97
28      238     73.5975 36.5385 0       97
29      239     76.405  35.5696 0       97
30      246     78.6025 33.4281 0       97
31      240     74.83   36.5157 0       97
32      216     80.2625 32.6659 0       97
33      220     80.6425 33.0933 0       97
34      247     78.245  34.6022 0       97
35      241     81.22   32.1885 0       97
36      234     83.6375 29.0002 0       97
37      228     82.485  31.7354 0       97
38      219     83.4625 30.0592 0       97
39      212     88.64   24.2702 0       97
40      231     86.7275 27.0879 0       97
41      229     89.1825 23.8773 0       97
42      216     87.96   25.1649 0       97
43      218     86.85   27.1116 0       97
44      236     88.78   23.7278 0       97
45      225     89.115  23.4212 0       97
46      232     88.5425 24.187  0       97
47      245     87.7775 25.3909 0       97
48      231     87.78   26.3786 0       97
49      238     88.8525 24.5115 0       97
50      233     87.82   25.4164 1       97
```

Summary

In this chapter, we learned about genetic algorithms and their underlying concepts. We discussed evolutionary algorithms and genetic programming. We understood how they are related to genetic algorithms. We discussed the fundamental building blocks of genetic algorithms including the concepts of population, crossover, mutation, selection, and fitness function. We learned how to generate a bit pattern with predefined parameters. We discussed how to visualize the evolution process using CMA-ES. We learnt how to solve the symbol regression problem in this paradigm. We then used these concepts to build a robot controller to traverse a map and consume all the targets. In the next chapter, we will learn about reinforcement learning and see how to build a smart agent.

9
Building Games With Artificial Intelligence

In this chapter, we are going to learn how to build games with Artificial Intelligence. We will learn how to use search algorithms to effectively come up with strategies to win the games. We will then use these algorithms to build intelligent bots for different games.

By the end of this chapter, you will understand the following concepts:

- Using search algorithms in games
- Combinatorial search
- Minimax algorithm
- Alpha-Beta pruning
- Negamax algorithm
- Building a bot to play Last Coin Standing
- Building a bot to play Tic Tac Toe
- Building two bots to play Connect Four against each other
- Building two bots to play Hexapawn against each other

Using search algorithms in games

Search algorithms are used in games to figure out a strategy. The algorithms search through the possibilities and pick the best move. There are various parameters to think about – speed, accuracy, complexity, and so on. These algorithms consider all possible actions available at this time and then evaluate their future moves based on these options. The goal of these algorithms is to find the optimal set of moves that will help them arrive at the final condition. Every game has a different set of winning conditions. These algorithms use those conditions to find the set of moves.

The description given in the previous paragraph is ideal if there is no opposing player. Things are not as straightforward with games that have multiple players. Let's consider a two-player game. For every move made by a player, the opposing player will make a move to prevent the player from achieving the goal. So when a search algorithm finds the optimal set of moves from the current state, it cannot just go ahead and make those moves because the opposing player will stop it. This basically means that search algorithms need to constantly re-evaluate after each move.

Let's discuss how a computer perceives any given game. We can think of a game as a search tree. Each node in this tree represents a future state. For example, if you are playing **Tic–Tac–Toe** (Noughts and Crosses), you can construct this tree to represent all possible moves. We start from the root of the tree, which is the starting point of the game. This node will have several children that represent various possible moves. Those children, in turn, will have more children that represent game states after more moves by the opponent. The terminal nodes of the tree represent the final results of the game after making various moves. The game would either end in a draw or one of the players would win it. The search algorithms search through this tree to make decisions at each step of the game.

Combinatorial search

Search algorithms appear to solve the problem of adding intelligence to games, but there's a drawback. These algorithms employ a type of search called exhaustive search, which is also known as brute force search. It basically explores the entire search space and tests every possible solution. It means that, in the worst case, we will have to explore all the possible solutions before we get the right solution.

As the games get more complex, we cannot rely on brute force search because the number of possibilities gets enormous. This quickly becomes computationally intractable. In order to solve this problem, we use combinatorial search to solve problems. It refers to a field of study where search algorithms efficiently explore the solution space using heuristics or by reducing the size of the search space. This is very useful in games like Chess or Go. Combinatorial search works efficiently by using pruning strategies. These strategies help it avoid testing all possible solutions by eliminating the ones that are obviously wrong. This helps save time and effort.

Minimax algorithm

Now that we have briefly discussed combinatorial search, let's talk about the heuristics that are employed by combinatorial search algorithms. These heuristics are used to speed up the search strategy and the Minimax algorithm is one such strategy used by combinatorial search. When two players are playing against each other, they are basically working towards opposite goals. So each side needs to predict what the opposing player is going to do in order to win the game. Keeping this in mind, Minimax tries to achieve this through strategy. It will try to minimize the function that the opponent is trying to maximize.

As we know, brute forcing the solution is not an option. The computer cannot go through all the possible states and then get the best possible set of moves to win the game. The computer can only optimize the moves based on the current state using a heuristic. The computer constructs a tree and it starts from the bottom. It evaluates which moves would benefit its opponent. Basically, it knows which moves the opponent is going to make based on the premise that the opponent will make the moves that would benefit them the most, and thereby be of the least benefit to the computer. This outcome is one of the terminal nodes of the tree and the computer uses this position to work backwards. Each option that's available to the computer can be assigned a value and it can then pick the highest value to take an action.

Alpha-Beta pruning

Minimax search is an efficient strategy, but it still ends up exploring parts of the tree that are irrelevant. Let's consider a tree where we are supposed to search for solutions. Once we find an indicator on a node that tells us that the solution does not exist in that sub-tree, there is no need to evaluate that sub-tree. But Minimax search is a bit too conservative, so it ends up exploring that sub-tree.

We need to be smart about it and avoid searching a part of a tree that is not necessary. This process is called **pruning** and Alpha-Beta pruning is a type of avoidance strategy that is used to avoid searching parts of the tree that do not contain the solution.

The Alpha and Beta parameters in alpha-beta pruning refer to the two bounds that are used during the calculation. These parameters refer to the values that restrict the set of possible solutions. This is based on the section of the tree that has already been explored. Alpha is the maximum lower bound of the number of possible solutions and Beta is the minimum upper bound on the number of possible solutions.

As we discussed earlier, each node can be assigned a value based on the current state. When the algorithm considers any new node as a potential path to the solution, it can work out if the current estimate of the value of the node lies between alpha and beta. This is how it prunes the search.

Negamax algorithm

The **Negamax** algorithm is a variant of Minimax that's frequently used in real world implementations. A two-player game is usually a zero-sum game, which means that one player's loss is equal to another player's gain and vice versa. Negamax uses this property extensively to come up with a strategy to increases its chances of winning the game.

In terms of the game, the value of a given position to the first player is the negation of the value to the second player. Each player looks for a move that will maximize the damage to the opponent. The value resulting from the move should be such that the opponent gets the least value. This works both ways seamlessly, which means that a single method can be used to value the positions. This is where it has an advantage over Minimax in terms of simplicity. Minimax requires that the first player select the move with the maximum value, whereas the second player must select a move with the minimum value. Alpha-beta pruning is used here as well.

Installing easyAI library

We will be using a library called `easyAI` in this chapter. It is an artificial intelligence framework and it provides all the functionality necessary to build two-player games. You can learn about it at `http://zulko.github.io/easyAI`.

Install it by running the following command on your Terminal:

```
$ pip3 install easyAI
```

We need some of the files to be accessible in order to use some of the pre-built routines. For ease of use, the code provided with this book contains a folder called `easyAI`. Make sure you place this folder in the same folder as your code files. This folder is basically a subset of the `easyAI` GitHub repository available at `https://github.com/Zulko/easyAI`. You can go through the source code to make yourself more familiar with it.

Building a bot to play Last Coin Standing

This is a game where we have a pile of coins and each player takes turns to take a number of coins from the pile. There is a lower and an upper bound on the number of coins that can be taken from the pile. The goal of the game is to avoid taking the last coin in the pile. This recipe is a variant of the Game of Bones recipe given in the `easyAI` library. Let's see how to build a game where the computer can play against the user.

Create a new Python file and import the following packages:

```
from easyAI import TwoPlayersGame, id_solve, Human_Player, AI_Player
from easyAI.AI import TT
```

Create a class to handle all the operations of the game. We will be inheriting from the base class `TwoPlayersGame` available in the `easyAI` library. There are a couple of parameters that have been to defined in order for it to function properly. The first one is the `players` variable. We will talk about the `player` object later. Create the class using the following code:

```
class LastCoinStanding(TwoPlayersGame):
    def __init__(self, players):
        # Define the players. Necessary parameter.
        self.players = players
```

Define who is going to start the game. The players are numbered from one. So in this case, player one starts the game:

```
        # Define who starts the game. Necessary parameter.
        self.nplayer = 1
```

Define the number of coins in the pile. You are free to choose any number here. In this case, let's choose `25`:

```
        # Overall number of coins in the pile
        self.num_coins = 25
```

Define the maximum number of coins that can be taken out in any move. You are free to choose any number for this parameter as well. Let's choose 4 in our case:

```
        # Define max number of coins per move
        self.max_coins = 4
```

Define all the possible movies. In this case, players can take either 1, 2, 3, or 4 coins in each move:

```
    # Define possible moves
    def possible_moves(self):
        return [str(x) for x in range(1, self.max_coins + 1)]
```

Define a method to remove the coins and keep track of the number of coins remaining in the pile:

```
    # Remove coins
    def make_move(self, move):
        self.num_coins -= int(move)
```

Check if somebody won the game by checking the number of coins remaining:

```
    # Did the opponent take the last coin?
    def win(self):
        return self.num_coins <= 0
```

Stop the game after somebody wins it:

```
    # Stop the game when somebody wins
    def is_over(self):
        return self.win()
```

Compute the score based on the `win` method. It's necessary to define this method:

```
    # Compute score
    def scoring(self):
        return 100 if self.win() else 0
```

Define a method to show the current status of the pile:

```
    # Show number of coins remaining in the pile
    def show(self):
        print(self.num_coins, 'coins left in the pile')
```

Define the `main` function and start by defining the transposition table. Transposition tables are used in games to store the positions and movements so as to speed up the algorithm. Type in the following code:

```
if __name__ == "__main__":
    # Define the transposition table
    tt = TT()
```

Define the method `ttentry` to get the number of coins. It's an optional method that's used to create a string to describe the game:

```
    # Define the method
    LastCoinStanding.ttentry = lambda self: self.num_coins
```

Let's solve the game using AI. The function `id_solve` is used to solve a given game using iterative deepening. It basically determines who can win a game using all the paths. It looks to answer questions such as, Can the first player force a win by playing perfectly? Will the computer always lose against a perfect opponent?

The method `id_solve` explores various options in the game's Negamax algorithm several times. It always starts at the initial state of the game and takes increasing depth to keep going. It will do it until the score indicates that somebody will win or lose. The second argument in the method takes a list of depths that it will try out. In this case, it will try all the values from 2 to 20:

```
    # Solve the game
    result, depth, move = id_solve(LastCoinStanding,
            range(2, 20), win_score=100, tt=tt)
    print(result, depth, move)
```

Start the game against the computer:

```
    # Start the game
    game = LastCoinStanding([AI_Player(tt), Human_Player()])
    game.play()
```

The full code is given in the file `coins.py`. It's an interactive program, so it will expect input from the user. If you run the code, you will basically be playing against the computer. Your goal is to force the computer to take the last coin, so that you win the game. If you run the code, you will get the following output on your Terminal at the beginning:

```
d:2, a:0, m:1
d:3, a:0, m:1
d:4, a:0, m:1
d:5, a:0, m:1
d:6, a:0, m:1
d:7, a:0, m:1
d:8, a:0, m:1
d:9, a:0, m:1
d:10, a:100, m:4
1 10 4
25 coins left in the pile

Move #1: player 1 plays 4 :
21 coins left in the pile

Player 2 what do you play ? 1

Move #2: player 2 plays 1 :
20 coins left in the pile

Move #3: player 1 plays 4 :
16 coins left in the pile
```

If you scroll down, you will see the following towards the end:

```
Move #5: player 1 plays 2 :
11 coins left in the pile

Player 2 what do you play ? 4

Move #6: player 2 plays 4 :
7 coins left in the pile

Move #7: player 1 plays 1 :
6 coins left in the pile

Player 2 what do you play ? 2

Move #8: player 2 plays 2 :
4 coins left in the pile

Move #9: player 1 plays 3 :
1 coins left in the pile

Player 2 what do you play ? 1

Move #10: player 2 plays 1 :
0 coins left in the pile
```

As we can see, the computer wins the game because the user picked up the last coin.

Building a bot to play Tic-Tac-Toe

Tic-Tac-Toe (Noughts and Crosses) is probably one of the most famous games. Let's see how to build a game where the computer can play against the user. This is a minor variant of the Tic-Tac-Toe recipe given in the `easyAI` library.

Create a new Python file and import the following packages:

```
from easyAI import TwoPlayersGame, AI_Player, Negamax
from easyAI.Player import Human_Player
```

Define a class that contains all the methods to play the game. Start by defining the players and who starts the game:

```
class GameController(TwoPlayersGame):
    def __init__(self, players):
        # Define the players
        self.players = players

        # Define who starts the game
        self.nplayer = 1
```

We will be using a 3×3 board numbered from one to nine row-wise:

```
        # Define the board
        self.board = [0] * 9
```

Define a method to compute all the possible moves:

```
    # Define possible moves
    def possible_moves(self):
        return [a + 1 for a, b in enumerate(self.board) if b == 0]
```

Define a method to update the board after making a move:

```
    # Make a move
    def make_move(self, move):
        self.board[int(move) - 1] = self.nplayer
```

Define a method to see if somebody has lost the game. We will be checking if somebody has three in a row:

```
    # Does the opponent have three in a line?
    def loss_condition(self):
        possible_combinations = [[1,2,3], [4,5,6], [7,8,9],
            [1,4,7], [2,5,8], [3,6,9], [1,5,9], [3,5,7]]
        return any([all([(self.board[i-1] == self.nopponent)
               for i in combination]) for combination in
possible_combinations])
```

Check if the game is over using the `loss_condition` method:

```
    # Check if the game is over
    def is_over(self):
        return (self.possible_moves() == []) or self.loss_condition()
```

Define a method to show the current progress:

```
    # Show current position
    def show(self):
        print('\n'+'\n'.join([' '.join([['. ', 'O', 'X'][self.board[3*j +
i]]
                for i in range(3)]) for j in range(3)]))
```

Compute the score using the `loss_condition` method:

```
    # Compute the score
    def scoring(self):
        return -100 if self.loss_condition() else 0
```

Define the main function and start by defining the algorithm. We will be using Negamax as the AI algorithm for this game. We can specify the number of steps in advance that the algorithm should think. In this case, let's choose `7`:

```
if __name__ == "__main__":
    # Define the algorithm
    algorithm = Negamax(7)
```

Start the game:

```
    # Start the game
    GameController([Human_Player(), AI_Player(algorithm)]).play()
```

The full code is given in the file `tic_tac_toe.py`. It's an interactive game where you play against the computer. If you run the code, you will get the following output on your Terminal at the beginning:

```
. . .
. . .
. . .

Player 1 what do you play ? 5

Move #1: player 1 plays 5 :

. . .
. O .
. . .

Move #2: player 2 plays 1 :

X . .
. O .
. . .

Player 1 what do you play ? 9

Move #3: player 1 plays 9 :

X . .
. O .
. . O
```

If you scroll down, you will see the following printed on your Terminal once it finishes executing the code:

```
X O X
. O .
. X O

Player 1 what do you play ? 4

Move #7: player 1 plays 4 :

X O X
O O .
. X O

Move #8: player 2 plays 6 :

X O X
O O X
. X O

Player 1 what do you play ? 7

Move #9: player 1 plays 7 :

X O X
O O X
O X O
```

As we can see, the game ends in a draw.

Building two bots to play Connect Four™ against each other

Connect Four™ is a popular two-player game sold under the Milton Bradley trademark. It is also known by other names such as Four in a Row or Four Up. In this game, the players take turns dropping discs into a vertical grid consisting of six rows and seven columns. The goal is to get four discs in a line. This is a variant of the Connect Four recipe given in the easyAI library. Let's see how to build it. In this recipe, instead of playing against the computer, we will create two bots that will play against each other. We will use a different algorithm for each to see which one wins.
Create a new Python file and import the following packages:

```
import numpy as np
from easyAI import TwoPlayersGame, Human_Player, AI_Player, \
        Negamax, SSS
```

Define a class that contains all the methods needed to play the game:

```
class GameController(TwoPlayersGame):
    def __init__(self, players, board = None):
        # Define the players
        self.players = players
```

Define the board with six rows and seven columns:

```
        # Define the configuration of the board
        self.board = board if (board != None) else (
            np.array([[0 for i in range(7)] for j in range(6)]))
```

Define who's going to start the game. In this case, let's have player one start the game:

```
        # Define who starts the game
        self.nplayer = 1
```

Define the positions:

```
        # Define the positions
        self.pos_dir = np.array([[[i, 0], [0, 1]] for i in range(6)] +
                    [[[0, i], [1, 0]] for i in range(7)] +
                    [[[i, 0], [1, 1]] for i in range(1, 3)] +
                    [[[0, i], [1, 1]] for i in range(4)] +
                    [[[i, 6], [1, -1]] for i in range(1, 3)] +
                    [[[0, i], [1, -1]] for i in range(3, 7)])
```

Define a method to get all the possible moves:

```
# Define possible moves
def possible_moves(self):
    return [i for i in range(7) if (self.board[:, i].min() == 0)]
```

Define a method to control how to make a move:

```
# Define how to make the move
def make_move(self, column):
    line = np.argmin(self.board[:, column] != 0)
    self.board[line, column] = self.nplayer
```

Define a method to show the current status:

```
# Show the current status
def show(self):
    print('\n' + '\n'.join(
            ['0 1 2 3 4 5 6', 13 * '-'] +
            [' '.join([['.', 'O', 'X'][self.board[5 - j][i]]
            for i in range(7)]) for j in range(6)]))
```

Define a method to compute what a loss looks like. Whenever somebody gets four in a line, that player wins the game:

```
# Define what a loss_condition looks like
def loss_condition(self):
    for pos, direction in self.pos_dir:
        streak = 0
        while (0 <= pos[0] <= 5) and (0 <= pos[1] <= 6):
            if self.board[pos[0], pos[1]] == self.nopponent:
                streak += 1
                if streak == 4:
                    return True
            else:
                streak = 0

            pos = pos + direction

    return False
```

Check if the game is over by using the `loss_condition` method:

```
# Check if the game is over
def is_over(self):
    return (self.board.min() > 0) or self.loss_condition()
```

Compute the score:

```
    # Compute the score
    def scoring(self):
        return -100 if self.loss_condition() else 0
```

Define the main function and start by defining the algorithms. We will let two algorithms play against each other. We will use Negamax for the first computer player and **SSS*** algorithm for the second computer player. SSS* is basically a search algorithm that conducts a state space search by traversing the tree in a best-first style. Both methods take, as an input argument, the number of turns in advance to think about. In this case, let's use five for both:

```
if __name__ == '__main__':
    # Define the algorithms that will be used
    algo_neg = Negamax(5)
    algo_sss = SSS(5)
```

Start playing the game:

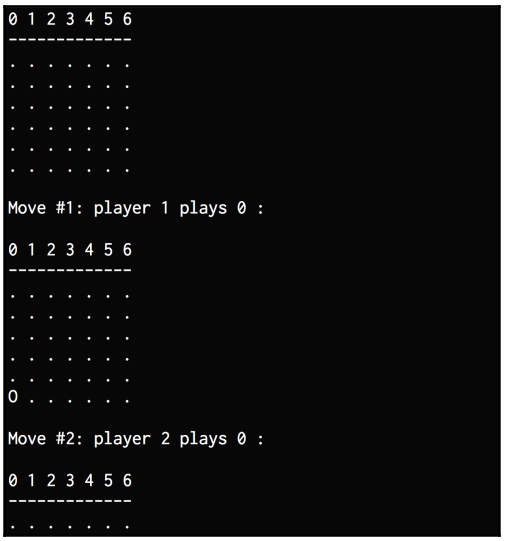
```
    # Start the game
    game = GameController([AI_Player(algo_neg), AI_Player(algo_sss)])
    game.play()
```

Print the result:

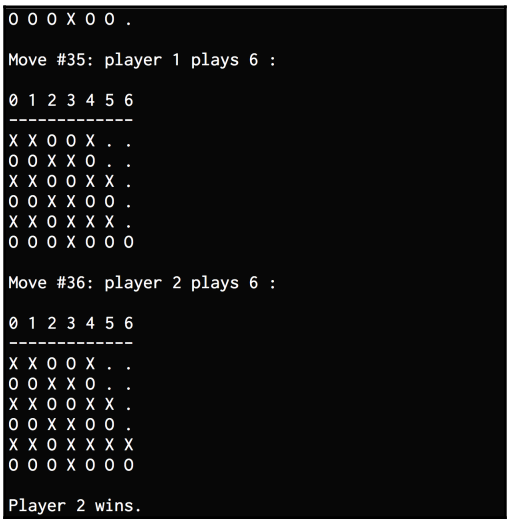
```
    # Print the result
    if game.loss_condition():
        print('\nPlayer', game.nopponent, 'wins. ')
    else:
        print("\nIt's a draw.")
```

The full code is given in the file `connect_four.py`. This is not an interactive game. We are just pitting one algorithm against another. Negamax algorithm is player one and SSS* algorithm is player two.

If you run the code, you will get the following output on your Terminal at the beginning:



If you scroll down, you will see the following towards the end:



As we can see, player two wins the game.

Building two bots to play Hexapawn against each other

Hexapawn is a two-player game where we start with a chessboard of size *NxM*. We have pawns on each side of the board and the goal is to advance a pawn all the way to the other end of the board. The standard pawn rules of chess are applicable here. This is a variant of the Hexapawn recipe given in the `easyAI` library. We will create two bots and pit an algorithm against itself to see what happens.

Create a new Python file and import the following packages:

```
from easyAI import TwoPlayersGame, AI_Player, \
        Human_Player, Negamax
```

Define a class that contains all the methods necessary to control the game. Start by defining the number of pawns on each side and the length of the board. Create a list of tuples containing the positions:

```
class GameController(TwoPlayersGame):
    def __init__(self, players, size = (4, 4)):
        self.size = size
        num_pawns, len_board = size
        p = [[(i, j) for j in range(len_board)] \
                for i in [0, num_pawns - 1]]
```

Assign the direction, goals, and pawns to each player:

```
        for i, d, goal, pawns in [(0, 1, num_pawns - 1,
                p[0]), (1, -1, 0, p[1])]:
            players[i].direction = d
            players[i].goal_line = goal
            players[i].pawns = pawns
```

Define the players and specify who starts first:

```
        # Define the players
        self.players = players

        # Define who starts first
        self.nplayer = 1
```

Define the alphabets that will be used to identify positions like B6 or C7 on a chessboard:

```
        # Define the alphabets
        self.alphabets = 'ABCDEFGHIJ'
```

Define a lambda function to convert strings to tuples:

```
        # Convert B4 to (1, 3)
        self.to_tuple = lambda s: (self.alphabets.index(s[0]),
                int(s[1:]) - 1)
```

Define a lambda function to convert tuples to strings:

```
        # Convert (1, 3) to B4
        self.to_string = lambda move: ' '.join([self.alphabets[
                move[i][0]] + str(move[i][1] + 1)
                for i in (0, 1)])
```

Define a method to compute the possible moves:

```
    # Define the possible moves
    def possible_moves(self):
        moves = []
        opponent_pawns = self.opponent.pawns
        d = self.player.direction
```

If you don't find an opponent pawn in a position, then that's a valid move:

```
        for i, j in self.player.pawns:
            if (i + d, j) not in opponent_pawns:
                moves.append(((i, j), (i + d, j)))

            if (i + d, j + 1) in opponent_pawns:
                moves.append(((i, j), (i + d, j + 1)))

            if (i + d, j - 1) in opponent_pawns:
                moves.append(((i, j), (i + d, j - 1)))

        return list(map(self.to_string, [(i, j) for i, j in moves]))
```

Define how to make a move and update the pawns based on that:

```
    # Define how to make a move
    def make_move(self, move):
        move = list(map(self.to_tuple, move.split(' ')))
        ind = self.player.pawns.index(move[0])
        self.player.pawns[ind] = move[1]

        if move[1] in self.opponent.pawns:
            self.opponent.pawns.remove(move[1])
```

Define the conditions for a loss. If a player gets 4 in a line, then the opponent loses:

```
    # Define what a loss looks like
```

```
    def loss_condition(self):
        return (any([i == self.opponent.goal_line
                for i, j in self.opponent.pawns])
                or (self.possible_moves() == []) )
```

Check if the game is over using the `loss_condition` method:

```
    # Check if the game is over
    def is_over(self):
        return self.loss_condition()
```

Print the current status:

```
    # Show the current status
    def show(self):
        f = lambda x: '1' if x in self.players[0].pawns else (
                '2' if x in self.players[1].pawns else '.')

        print("\n".join([" ".join([f((i, j))
                for j in range(self.size[1])])
                for i in range(self.size[0])]))
```

Define the main function and start by defining the scoring lambda function:

```
if __name__=='__main__':
    # Compute the score
    scoring = lambda game: -100 if game.loss_condition() else 0
```

Define the algorithm to be used. In this case, we will use Negamax that can calculate 12 moves in advance and uses a `scoring` lambda function for strategy:

```
    # Define the algorithm
    algorithm = Negamax(12, scoring)
```

Start playing the game:

```
    # Start the game
    game = GameController([AI_Player(algorithm),
            AI_Player(algorithm)])
    game.play()
    print('\nPlayer', game.nopponent, 'wins after', game.nmove, 'turns')
```

The full code is given in the file `hexapawn.py`. It's not an interactive game. We are pitting an AI algorithm against itself. If you run the code, you will get the following output on your Terminal at the beginning:

```
1 1 1 1
. . . .
. . . .
2 2 2 2

Move #1: player 1 plays A1 B1 :
. 1 1 1
1 . . .
. . . .
2 2 2 2

Move #2: player 2 plays D1 C1 :
. 1 1 1
1 . . .
2 . . .
. 2 2 2

Move #3: player 1 plays A2 B2 :
. . 1 1
1 1 . .
2 . . .
. 2 2 2

Move #4: player 2 plays D2 C2 :
. . 1 1
```

If you scroll down, you will see the following towards the end:

```
Move #4: player 2 plays D2 C2 :
. . 1 1
1 1 . .
2 2 . .
. . 2 2

Move #5: player 1 plays B1 C2 :
. . 1 1
. 1 . .
2 1 . .
. . 2 2

Move #6: player 2 plays C1 B1 :
. . 1 1
2 1 . .
. 1 . .
. . 2 2

Move #7: player 1 plays C2 D2 :
. . 1 1
2 1 . .
. . . .
. 1 2 2

Player 1 wins after 8 turns
```

As we can see, player one wins the game.

Summary

In this chapter, we discussed how to build games with artificial intelligence, and how to use search algorithms to effectively come up with strategies to win the games. We talked about combinatorial search and how it can be used to speed up the search process. We learned about Minimax and Alpha-Beta pruning. We learned how the Negamax algorithm is used in practice. We then used these algorithms to build bots to play Last Coin Standing and Tic-Tac-Toe.

We learned how to build two bots to play against each other in Connect Four and Hexapawn. In the next chapter, we will discuss natural language processing and how to use it to analyze text data by modeling and classifying it.

10
Natural Language Processing

In this chapter, we are going to learn about natural language processing. We will discuss various concepts such as tokenization, stemming, and lemmatization to process text. We will then discuss how to build a Bag of Words model and use it to classify text. We will see how to use machine learning to analyze the sentiment of a given sentence. We will then discuss topic modeling and implement a system to identify topics in a given document.

By the end of this chapter, you will know:

- How to install relevant packages
- Tokenizing text data
- Converting words to their base forms using stemming
- Converting words to their base forms using lemmatization
- Dividing text data into chunks
- Extracting document term matrix using the Bag of Words model
- Building a category predictor
- Constructing a gender identifier
- Building a sentiment analyzer
- Topic modeling using Latent Dirichlet Allocation

Introduction and installation of packages

Natural Language Processing (**NLP**) has become an important part of modern systems. It is used extensively in search engines, conversational interfaces, document processors, and so on. Machines can handle structured data well. But when it comes to working with free-form text, they have a hard time. The goal of NLP is to develop algorithms that enable computers to understand freeform text and help them understand language.

One of the most challenging things about processing freeform natural language is the sheer number of variations. The context plays a very important role in how a particular sentence is understood. Humans are great at these things because we have been trained for many years. We immediately use our past knowledge to understand the context and know what the other person is talking about.

To address this issue, NLP researchers started developing various applications using machine learning approaches. To build such applications, we need to collect a large corpus of text and then train the algorithm to perform various tasks like categorizing text, analyzing sentiments, or modeling topics. These algorithms are trained to detect patterns in input text data and derive insights from it.

In this chapter, we will discuss various underlying concepts that are used to analyze text and build NLP applications. This will enable us to understand how to extract meaningful information from the given text data. We will use a Python package called **Natural Language Toolkit** (**NLTK**) to build these applications. Make sure that you install this before you proceed. You can install it by running the following command on your Terminal:

```
$ pip3 install nltk
```

You can find more information about NLTK at `http://www.nltk.org`.

In order to access all the datasets provided by NLTK, we need to download it. Open up a Python shell by typing the following on your Terminal:

```
$ python3
```

We are now inside the Python shell. Type the following to download the data:

```
>>> import nltk
>>> nltk.download()
```

We will also use a package called `gensim` in this chapter. It's a robust semantic modeling library that's useful for many applications. You can install it by running the following command on your Terminal:

```
$ pip3 install gensim
```

You might need another package called `pattern` for `gensim` to function properly. You can install it by running the following command on your Terminal:

```
$ pip3 install pattern
```

You can find more information about `gensim` at `https://radimrehurek.com/gensim`. Now that you have installed the `NLTK` and `gensim`, let's proceed with the discussion.

Tokenizing text data

When we deal with text, we need to break it down into smaller pieces for analysis. This is where tokenization comes into the picture. It is the process of dividing the input text into a set of pieces like words or sentences. These pieces are called tokens. Depending on what we want to do, we can define our own methods to divide the text into many tokens. Let's take a look at how to tokenize the input text using `NLTK`.

Create a new Python file and import the following packages:

```
from nltk.tokenize import sent_tokenize, \
        word_tokenize, WordPunctTokenizer
```

Define some input text that will be used for tokenization:

```
# Define input text
input_text = "Do you know how tokenization works? It's actually quite
interesting! Let's analyze a couple of sentences and figure it out."
```

Divide the input text into sentence tokens:

```
# Sentence tokenizer
print("\nSentence tokenizer:")
print(sent_tokenize(input_text))
```

Divide the input text into word tokens:

```
# Word tokenizer
print("\nWord tokenizer:")
print(word_tokenize(input_text))
```

Divide the input text into word tokens using word punct tokenizer:

```
# WordPunct tokenizer
print("\nWord punct tokenizer:")
print(WordPunctTokenizer().tokenize(input_text))
```

The full code is given in the file `tokenizer.py`. If you run the code, you will get the following output on your Terminal:

```
Sentence tokenizer:
['Do you know how tokenization works?', "It's actually quite interesting!", "Let's analyze a couple of se
ntences and figure it out."]

Word tokenizer:
['Do', 'you', 'know', 'how', 'tokenization', 'works', '?', 'It', "'s", 'actually', 'quite', 'interesting'
, '!', 'Let', "'s", 'analyze', 'a', 'couple', 'of', 'sentences', 'and', 'figure', 'it', 'out', '.']

Word punct tokenizer:
['Do', 'you', 'know', 'how', 'tokenization', 'works', '?', 'It', "'", 's', 'actually', 'quite', 'interest
ing', '!', 'Let', "'", 's', 'analyze', 'a', 'couple', 'of', 'sentences', 'and', 'figure', 'it', 'out', '.
']
```

We can see that the sentence tokenizer divides the input text into sentences. The two word tokenizers behave differently when it comes to punctuation. For example, the word "It's" is divided differently in the punct tokenizer as compared to the regular tokenizer.

Converting words to their base forms using stemming

Working with text has a lot of variations included in it. We have to deal with different forms of the same word and enable the computer to understand that these different words have the same base form. For example, the word *sing* can appear in many forms such as *sang*, *singer*, *singing*, *singer*, and so on. We just saw a set of words with similar meanings. Humans can easily identify these base forms and derive context.

When we analyze text, it's useful to extract these base forms. It will enable us to extract useful statistics to analyze the input text. Stemming is one way to achieve this. The goal of a stemmer is to reduce words in their different forms into a common base form. It is basically a heuristic process that cuts off the ends of words to extract their base forms. Let's see how to do it using NLTK.

Create a new python file and import the following packages:

```
from nltk.stem.porter import PorterStemmer
from nltk.stem.lancaster import LancasterStemmer
from nltk.stem.snowball import SnowballStemmer
```

Define some input words:

```
input_words = ['writing', 'calves', 'be', 'branded', 'horse', 'randomize',
        'possibly', 'provision', 'hospital', 'kept', 'scratchy', 'code']
```

Create objects for **Porter**, **Lancaster**, and **Snowball** stemmers:

```
# Create various stemmer objects
porter = PorterStemmer()
lancaster = LancasterStemmer()
snowball = SnowballStemmer('english')
```

Create a list of names for table display and format the output text accordingly:

```
#Create a list of stemmer names for display
stemmer_names = ['PORTER', 'LANCASTER', 'SNOWBALL']
formatted_text = '{:>16}' * (len(stemmer_names) + 1)
print('\n', formatted_text.format('INPUT WORD', *stemmer_names),
        '\n', '='*68)
```

Iterate through the words and stem them using the three stemmers:

```
# Stem each word and display the output
for word in input_words:
    output = [word, porter.stem(word),
            lancaster.stem(word), snowball.stem(word)]
    print(formatted_text.format(*output))
```

The full code is given in the file `stemmer.py`. If you run the code, you will get the following output on your Terminal:

```
      INPUT WORD          PORTER       LANCASTER         SNOWBALL
====================================================================
         writing           write            writ           write
          calves            calv            calv            calv
              be              be              be              be
         branded           brand           brand           brand
           horse            hors            hors            hors
       randomize          random          random          random
        possibly        possibl            poss        possibl
       provision          provis          provid          provis
        hospital          hospit          hospit          hospit
            kept            kept            kept            kept
        scratchy        scratchi        scratchy        scratchi
            code            code             cod            code
```

Let's talk a bit about the three stemming algorithms that are being used here. All of them basically try to achieve the same goal. The difference between them is the level of strictness that's used to arrive at the base form.

The Porter stemmer is the least in terms of strictness and Lancaster is the strictest. If you closely observe the outputs, you will notice the differences. Stemmers behave differently when it comes to words like `possibly` or `provision`. The stemmed outputs that are obtained from the Lancaster stemmer are a bit obfuscated because it reduces the words a lot. At the same time, the algorithm is really fast. A good rule of thumb is to use the Snowball stemmer because it's a good trade off between speed and strictness.

Converting words to their base forms using lemmatization

Lemmatization is another way of reducing words to their base forms. In the previous section, we saw that the base forms that were obtained from those stemmers didn't make sense. For example, all the three stemmers said that the base form of *calves* is *calv*, which is not a real word. Lemmatization takes a more structured approach to solve this problem.

The lemmatization process uses a vocabulary and morphological analysis of words. It obtains the base forms by removing the inflectional word endings such as *ing* or *ed*. This base form of any word is known as the lemma. If you lemmatize the word *calves*, you should get *calf* as the output. One thing to note is that the output depends on whether the word is a verb or a noun. Let's take a look at how to do this using NLTK.

Create a new python file and import the following packages:

```
from nltk.stem import WordNetLemmatizer
```

Define some input words. We will be using the same set of words that we used in the previous section so that we can compare the outputs.

```
input_words = ['writing', 'calves', 'be', 'branded', 'horse', 'randomize',
        'possibly', 'provision', 'hospital', 'kept', 'scratchy', 'code']
```

Create a `lemmatizer` object:

```
# Create lemmatizer object
lemmatizer = WordNetLemmatizer()
```

Create a list of `lemmatizer` names for table display and format the text accordingly:

```
# Create a list of lemmatizer names for display
lemmatizer_names = ['NOUN LEMMATIZER', 'VERB LEMMATIZER']
formatted_text = '{:>24}' * (len(lemmatizer_names) + 1)
print('\n', formatted_text.format('INPUT WORD', *lemmatizer_names),
        '\n', '='*75)
```

Iterate through the words and lemmatize the words using Noun and Verb lemmatizers:

```
# Lemmatize each word and display the output
for word in input_words:
    output = [word, lemmatizer.lemmatize(word, pos='n'),
           lemmatizer.lemmatize(word, pos='v')]
    print(formatted_text.format(*output))
```

The full code is given in the file `lemmatizer.py`. If you run the code, you will get the following output on your Terminal:

```
              INPUT WORD        NOUN LEMMATIZER        VERB LEMMATIZER
===========================================================================
                 writing                writing                  write
                  calves                   calf                  calve
                      be                     be                     be
                branded                branded                  brand
                   horse                  horse                  horse
               randomize              randomize              randomize
                possibly               possibly               possibly
               provision              provision              provision
                hospital               hospital               hospital
                    kept                   kept                   keep
                scratchy               scratchy               scratchy
                    code                   code                   code
```

We can see that the noun `lemmatizer` works differently than the verb `lemmatizer` when it comes to words like *writing* or *calves*. If you compare these outputs to stemmer outputs, you will see that there are differences too. The `lemmatizer` outputs are all meaningful whereas stemmer outputs may or may not be meaningful.

Dividing text data into chunks

Text data usually needs to be divided into pieces for further analysis. This process is known as chunking. This is used frequently in text analysis. The conditions that are used to divide the text into chunks can vary based on the problem at hand. This is not the same as tokenization where we also divide text into pieces. During chunking, we do not adhere to any constraints and the output chunks need to be meaningful.

When we deal with large text documents, it becomes important to divide the text into chunks to extract meaningful information. In this section, we will see how to divide the input text into a number of pieces.

Create a new python file and import the following packages:

```
import numpy as np
from nltk.corpus import brown
```

Define a function to divide the input text into chunks. The first parameter is the text and the second parameter is the number of words in each chunk:

```
# Split the input text into chunks, where
# each chunk contains N words
def chunker(input_data, N):
    input_words = input_data.split(' ')
    output = []
```

Iterate through the words and divide them into chunks using the input parameter. The function returns a list:

```
    cur_chunk = []
    count = 0
    for word in input_words:
        cur_chunk.append(word)
        count += 1
        if count == N:
            output.append(' '.join(cur_chunk))
            count, cur_chunk = 0, []

    output.append(' '.join(cur_chunk))

    return output
```

Define the main function and read the input data using the Brown corpus. We will read `12,000` words in this case. You are free to read as many words as you want.

```
if __name__=='__main__':
    # Read the first 12000 words from the Brown corpus
    input_data = ' '.join(brown.words()[:12000])
```

Define the number of words in each chunk:

```
    # Define the number of words in each chunk
    chunk_size = 700
```

Divide the input text into chunks and display the output:

```
    chunks = chunker(input_data, chunk_size)
    print('\nNumber of text chunks =', len(chunks), '\n')
    for i, chunk in enumerate(chunks):
        print('Chunk', i+1, '==>', chunk[:50])
```

The full code is given in the file `text_chunker.py`. If you run the code, you will get the following output on your Terminal:

```
Number of text chunks = 18

Chunk 1 ==> The Fulton County Grand Jury said Friday an invest
Chunk 2 ==> '' . ( 2 ) Fulton legislators `` work with city of
Chunk 3 ==> . Construction bonds Meanwhile , it was learned th
Chunk 4 ==> , anonymous midnight phone calls and veiled threat
Chunk 5 ==> Harris , Bexar , Tarrant and El Paso would be $451
Chunk 6 ==> set it for public hearing on Feb. 22 . The proposa
Chunk 7 ==> College . He has served as a border patrolman and
Chunk 8 ==> of his staff were doing on the address involved co
Chunk 9 ==> plan alone would boost the base to $5,000 a year a
Chunk 10 ==> nursing homes In the area of `` community health s
Chunk 11 ==> of its Angola policy prove harsh , there has been
Chunk 12 ==> system which will prevent Laos from being used as
Chunk 13 ==> reform in recipient nations . In Laos , the admini
Chunk 14 ==> . He is not interested in being named a full-time
Chunk 15 ==> said , `` to obtain the views of the general publi
Chunk 16 ==> '' . Mr. Reama , far from really being retired , i
Chunk 17 ==> making enforcement of minor offenses more effectiv
Chunk 18 ==> to tell the people where he stands on the tax issu
```

The preceding screenshot shows the first 50 characters of each chunk.

Extracting the frequency of terms using a Bag of Words model

One of the main goals of text analysis is to convert text into numeric form so that we can use machine learning on it. Let's consider text documents that contain many millions of words. In order to analyze these documents, we need to extract the text and convert it into a form of numeric representation.

Machine learning algorithms need numeric data to work with so that they can analyze the data and extract meaningful information. This is where the Bag of Words model comes into picture. This model extracts a vocabulary from all the words in the documents and builds a model using a document term matrix. This allows us to represent every document as a *bag of words*. We just keep track of word counts and disregard the grammatical details and the word order.

Let's see what a document-term matrix is all about. A document term matrix is basically a table that gives us counts of various words that occur in the document. So a text document can be represented as a weighted combination of various words. We can set thresholds and choose words that are more meaningful. In a way, we are building a histogram of all the words in the document that will be used as a feature vector. This feature vector is used for text classification.

Consider the following sentences:

- Sentence 1: The children are playing in the hall
- Sentence 2: The hall has a lot of space
- Sentence 3: Lots of children like playing in an open space

If you consider all the three sentences, we have the following nine unique words:

- the
- children
- are
- playing
- in
- hall
- has

- a
- lot
- of
- space
- like
- an
- open

There are 14 distinct words here. Let's construct a histogram for each sentence by using the word count in each sentence. Each feature vector will be 14-dimensional because we have 14 distinct words overall:

- Sentence 1: [2, 1, 1, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0]
- Sentence 2: [1, 0, 0, 0, 0, 1, 1, 1, 1, 1, 1, 0, 0, 0]
- Sentence 3: [0, 1, 0, 1, 1, 0, 0, 0, 1, 1, 1, 1, 1, 1]

Now that we have extracted these feature vectors, we can use machine learning algorithms to analyze this data.

Let's see how to build a Bag of Words model in `NLTK`. Create a new python file and import the following packages:

```
import numpy as np
from sklearn.feature_extraction.text import CountVectorizer
from nltk.corpus import brown
from text_chunker import chunker
```

Read the input data from Brown corpus. We will read 5,400 words. You are free to read as many number of words as you want.

```
# Read the data from the Brown corpus
input_data = ' '.join(brown.words()[:5400])
```

Define the number of words in each chunk:

```
# Number of words in each chunk
chunk_size = 800
```

Divide the input text into chunks:

```
text_chunks = chunker(input_data, chunk_size)
```

Convert the chunks into dictionary items:

```
# Convert to dict items
chunks = []
for count, chunk in enumerate(text_chunks):
    d = {'index': count, 'text': chunk}
    chunks.append(d)
```

Extract the document term matrix where we get the count of each word. We will achieve this using the `CountVectorizer` method that takes two input parameters. The first parameter is the minimum document frequency and the second parameter is the maximum document frequency. The frequency refers to the number of occurrences of a word in the text.

```
# Extract the document term matrix
count_vectorizer = CountVectorizer(min_df=7, max_df=20)
document_term_matrix = count_vectorizer.fit_transform([chunk['text'] for
chunk in chunks])
```

Extract the vocabulary and display it. The vocabulary refers to the list of distinct words that were extracted in the previous step.

```
# Extract the vocabulary and display it
vocabulary = np.array(count_vectorizer.get_feature_names())
print("\nVocabulary:\n", vocabulary)
```

Generate the names for display:

```
# Generate names for chunks
chunk_names = []
for i in range(len(text_chunks)):
    chunk_names.append('Chunk-' + str(i+1))
```

Print the document term matrix:

```
# Print the document term matrix
print("\nDocument term matrix:")
formatted_text = '{:>12}' * (len(chunk_names) + 1)
print('\n', formatted_text.format('Word', *chunk_names), '\n')
for word, item in zip(vocabulary, document_term_matrix.T):
    # 'item' is a 'csr_matrix' data structure
    output = [word] + [str(freq) for freq in item.data]
    print(formatted_text.format(*output))
```

The full code is given in the file `bag_of_words.py`. If you run the code, you will get the following output on your Terminal:

```
Document term matrix:

        Word     Chunk-1     Chunk-2     Chunk-3     Chunk-4     Chunk-5     Chunk-6     Chunk-7

         and          23           9           9          11           9          17          10
         are           2           2           1           1           2           2           1
          be           6           8           7           7           6           2           1
          by           3           4           4           5          14           3           6
      county           6           2           7           3           1           2           2
         for           7          13           4          10           7           6           4
          in          15          11          15          11          13          14          17
          is           2           7           3           4           5           5           2
          it           8           6           8           9           3           1           2
          of          31          20          20          30          29          35          26
          on           4           3           5          10           6           5           2
         one           1           3           1           2           2           1           1
        said          12           5           7           7           4           3           7
       state           3           7           2           6           3           4           1
        that          13           8           9           2           7           1           7
         the          71          51          43          51          43          52          49
          to          11          26          20          26          21          15          11
         two           2           1           1           1           1           2           2
         was           5           6           7           7           4           7           3
       which           7           4           5           4           3           1           1
        with           2           2           3           1           2           2           3
```

We can see all the words in the document term matrix and the corresponding counts in each chunk.

Building a category predictor

A category predictor is used to predict the category to which a given piece of text belongs. This is frequently used in text classification to categorize text documents. Search engines frequently use this tool to order the search results by relevance. For example, let's say that we want to predict whether a given sentence belongs to sports, politics, or science. To do this, we build a corpus of data and train an algorithm. This algorithm can then be used for inference on unknown data.

In order to build this predictor, we will use a statistic called **TermFrequency – Inverse Document Frequency** (**tf-idf**). In a set of documents, we need to understand the importance of each word. The tf-idf statistic helps us understand how important a given word is to a document in a set of documents.

Let's consider the first part of this statistic. The **Term Frequency** (**tf**) is basically a measure of how frequently each word appears in a given document. Since different documents have a different number of words, the exact numbers in the histogram will vary. In order to have a level playing field, we need to normalize the histograms. So we divide the count of each word by the total number of words in a given document to obtain the term frequency.

The second part of the statistic is the **Inverse Document Frequency** (**idf**), which is a measure of how unique a word is to this document in the given set of documents. When we compute the term frequency, the assumption is that all the words are equally important. But we cannot just rely on the frequency of each word because words like *and* and *the* appear a lot. To balance the frequencies of these commonly occurring words, we need to reduce their weights and weigh up the rare words. This helps us identify words that are unique to each document as well, which in turn helps us formulate a distinctive feature vector.

To compute this statistic, we need to compute the ratio of the number of documents with the given word and divide it by the total number of documents. This ratio is essentially the fraction of the documents that contain the given word. Inverse document frequency is then calculated by taking the negative algorithm of this ratio.

We then combine term frequency and inverse document frequency to formulate a feature vector to categorize documents. Let's see how to build a category predictor.

Create a new python file and import the following packages:

```
from sklearn.datasets import fetch_20newsgroups
from sklearn.naive_bayes import MultinomialNB
from sklearn.feature_extraction.text import TfidfTransformer
from sklearn.feature_extraction.text import CountVectorizer
```

Define the map of categories that will be used for training. We will be using five categories in this case. The keys in this dictionary object refer to the names in the `scikit-learn` dataset.

```
# Define the category map
category_map = {'talk.politics.misc': 'Politics', 'rec.autos': 'Autos',
        'rec.sport.hockey': 'Hockey', 'sci.electronics': 'Electronics',
        'sci.med': 'Medicine'}
```

Get the training dataset using `fetch_20newsgroups`:

```
# Get the training dataset
training_data = fetch_20newsgroups(subset='train',
        categories=category_map.keys(), shuffle=True, random_state=5)
```

Extract the term counts using the `CountVectorizer` object:

```
# Build a count vectorizer and extract term counts
count_vectorizer = CountVectorizer()
train_tc = count_vectorizer.fit_transform(training_data.data)
print("\nDimensions of training data:", train_tc.shape)
```

Create Term Frequency – Inverse Document Frequency (tf-idf) transformer and train it using the data:

```
# Create the tf-idf transformer
tfidf = TfidfTransformer()
train_tfidf = tfidf.fit_transform(train_tc)
```

Define some sample input sentences that will be used for testing:

```
# Define test data
input_data = [
    'You need to be careful with cars when you are driving on slippery
roads',
    'A lot of devices can be operated wirelessly',
    'Players need to be careful when they are close to goal posts',
    'Political debates help us understand the perspectives of both sides'
]
```

Train a Multinomial Bayes classifier using the training data:

```
# Train a Multinomial Naive Bayes classifier
classifier = MultinomialNB().fit(train_tfidf, training_data.target)
```

Transform the input data using the count `vectorizer`:

```
# Transform input data using count vectorizer
input_tc = count_vectorizer.transform(input_data)
```

Transform the vectorized data using the `tf-idf` transformer so that it can be run through the inference model:

```
# Transform vectorized data using tfidf transformer
input_tfidf = tfidf.transform(input_tc)
```

Predict the output using the `tf-idf` transformed vector:

```
# Predict the output categories
predictions = classifier.predict(input_tfidf)
```

Print the output category for each sample in the input test data:

```
# Print the outputs
for sent, category in zip(input_data, predictions):
    print('\nInput:', sent, '\nPredicted category:', \
            category_map[training_data.target_names[category]])
```

The full code is given in the file `category_predictor.py`. If you run the code, you will get the following output on your Terminal:

```
Dimensions of training data: (2844, 40321)

Input: You need to be careful with cars when you are driving on slippery roads
Predicted category: Autos

Input: A lot of devices can be operated wirelessly
Predicted category: Electronics

Input: Players need to be careful when they are close to goal posts
Predicted category: Hockey

Input: Political debates help us understand the perspectives of both sides
Predicted category: Politics
```

We can see intuitively that the predicted categories are correct.

Constructing a gender identifier

Gender identification is an interesting problem. In this case, we will use the heuristic to construct a feature vector and use it to train a classifier. The heuristic that will be used here is the last *N* letters of a given name. For example, if the name ends with *ia*, it's most likely a female name, such as *Amelia* or *Genelia*. On the other hand, if the name ends with *rk*, it's likely a male name such as *Mark* or *Clark*. Since we are not sure of the exact number of letters to use, we will play around with this parameter and find out what the best answer is. Let's see how to do it.

Create a new python file and import the following packages:

```
import random

from nltk import NaiveBayesClassifier
from nltk.classify import accuracy as nltk_accuracy
from nltk.corpus import names
```

Define a function to extract the last *N* letters from the input word:

```
# Extract last N letters from the input word
# and that will act as our "feature"
def extract_features(word, N=2):
    last_n_letters = word[-N:]
    return {'feature': last_n_letters.lower()}
```

Define the main function and extract training data from the `scikit-learn` package. This data contains labeled male and female names:

```
if __name__=='__main__':
    # Create training data using labeled names available in NLTK
    male_list = [(name, 'male') for name in names.words('male.txt')]
    female_list = [(name, 'female') for name in names.words('female.txt')]
    data = (male_list + female_list)
```

Seed the random number generator and shuffle the data:

```
    # Seed the random number generator
    random.seed(5)

    # Shuffle the data
    random.shuffle(data)
```

Create some sample names that will be used for testing:

```
    # Create test data
    input_names = ['Alexander', 'Danielle', 'David', 'Cheryl']
```

Define the percentage of data that will be used for training and testing:

```
    # Define the number of samples used for train and test
    num_train = int(0.8 * len(data))
```

We will be using the last *N* characters as the feature vector to predict the gender. We will vary this parameter to see how the performance varies. In this case, we will go from 1 to 6:

```
# Iterate through different lengths to compare the accuracy
for i in range(1, 6):
    print('\nNumber of end letters:', i)
    features = [(extract_features(n, i), gender) for (n, gender) in data]
```

Separate the data into training and testing:

```
        train_data, test_data = features[:num_train], features[num_train:]
```

Build a `NaiveBayes` Classifier using the training data:

```
classifier = NaiveBayesClassifier.train(train_data)
```

Compute the accuracy of the classifier using the inbuilt method available in `NLTK`:

```
# Compute the accuracy of the classifier
accuracy = round(100 * nltk_accuracy(classifier, test_data), 2)
print('Accuracy = ' + str(accuracy) + '%')
```

Predict the output for each name in the input test list:

```
# Predict outputs for input names using the trained classifier model
for name in input_names:
    print(name, '==>', classifier.classify(extract_features(name,
i)))
```

The full code is given in the file `gender_identifier.py`. If you run the code, you will get the following output on your Terminal:

```
Number of end letters: 1
Accuracy = 74.7%
Alexander ==> male
Danielle ==> female
David ==> male
Cheryl ==> male

Number of end letters: 2
Accuracy = 78.79%
Alexander ==> male
Danielle ==> female
David ==> male
Cheryl ==> female

Number of end letters: 3
Accuracy = 77.22%
Alexander ==> male
Danielle ==> female
David ==> male
Cheryl ==> female
```

The preceding screenshot shows the accuracy as well as the predicted outputs for the test data. Let's go further and see what happens:

```
Number of end letters: 4
Accuracy = 69.98%
Alexander ==> male
Danielle ==> female
David ==> male
Cheryl ==> female

Number of end letters: 5
Accuracy = 64.63%
Alexander ==> male
Danielle ==> female
David ==> male
Cheryl ==> female
```

We can see that the accuracy peaked at two letters and then started decreasing after that.

Building a sentiment analyzer

Sentiment analysis is the process of determining the sentiment of a given piece of text. For example, it can used to determine whether a movie review is positive or negative. This is one of the most popular applications of natural language processing. We can add more categories as well depending on the problem at hand. This technique is generally used to get a sense of how people feel about a particular product, brand, or topic. It is frequently used to analyze marketing campaigns, opinion polls, social media presence, product reviews on e-commerce sites, and so on. Let's see how to determine the sentiment of a movie review.

We will use a Naive Bayes classifier to build this classifier. We first need to extract all the unique words from the text. The NLTK classifier needs this data to be arranged in the form of a dictionary so that it can ingest it. Once we divide the text data into training and testing datasets, we will train the Naive Bayes classifier to classify the reviews into positive and negative. We will also print out the top informative words to indicate positive and negative reviews. This information is interesting because it tells us what words are being used to denote various reactions.

Create a new python file and import the following packages:

```
from nltk.corpus import movie_reviews
from nltk.classify import NaiveBayesClassifier
from nltk.classify.util import accuracy as nltk_accuracy
```

Define a function to construct a dictionary object based on the input words and return it:

```
# Extract features from the input list of words
def extract_features(words):
    return dict([(word, True) for word in words])
```

Define the main function and load the labeled movie reviews:

```
if __name__=='__main__':
    # Load the reviews from the corpus
    fileids_pos = movie_reviews.fileids('pos')
    fileids_neg = movie_reviews.fileids('neg')
```

Extract the features from the movie reviews and label it accordingly:

```
    # Extract the features from the reviews
    features_pos = [(extract_features(movie_reviews.words(
            fileids=[f])), 'Positive') for f in fileids_pos]
    features_neg = [(extract_features(movie_reviews.words(
            fileids=[f])), 'Negative') for f in fileids_neg]
```

Define the split between training and testing. In this case, we will allocate 80% for training and 20% for testing:

```
    # Define the train and test split (80% and 20%)
    threshold = 0.8
    num_pos = int(threshold * len(features_pos))
    num_neg = int(threshold * len(features_neg))
```

Separate the feature vectors for training and testing:

```
     # Create training and training datasets
    features_train = features_pos[:num_pos] + features_neg[:num_neg]
    features_test = features_pos[num_pos:] + features_neg[num_neg:]
```

Print the number of datapoints used for training and testing:

```
    # Print the number of datapoints used
    print('\nNumber of training datapoints:', len(features_train))
    print('Number of test datapoints:', len(features_test))
```

Train a `Naive Bayes` classifier using the training data and compute the accuracy using the inbuilt method available in `NLTK`:

```
# Train a Naive Bayes classifier
classifier = NaiveBayesClassifier.train(features_train)
print('\nAccuracy of the classifier:', nltk_accuracy(
        classifier, features_test))
```

Print the top `N` most informative words:

```
N = 15
print('\nTop ' + str(N) + ' most informative words:')
for i, item in enumerate(classifier.most_informative_features()):
    print(str(i+1) + '. ' + item[0])
    if i == N - 1:
        break
```

Define sample sentences to be used for testing:

```
# Test input movie reviews
input_reviews = [
    'The costumes in this movie were great',
    'I think the story was terrible and the characters were very weak',
    'People say that the director of the movie is amazing',
    'This is such an idiotic movie. I will not recommend it to anyone.'
]
```

Iterate through the sample data and predict the output:

```
print("\nMovie review predictions:")
for review in input_reviews:
    print("\nReview:", review)
```

Compute the probabilities for each class:

```
        # Compute the probabilities
        probabilities =
classifier.prob_classify(extract_features(review.split()))
```

Pick the maximum value among the probabilities:

```
        # Pick the maximum value
        predicted_sentiment = probabilities.max()
```

Print the predicted output class (positive or negative sentiment):

```
            # Print outputs
            print("Predicted sentiment:", predicted_sentiment)
            print("Probability:",
    round(probabilities.prob(predicted_sentiment), 2))
```

The full code is given in the file `sentiment_analyzer.py`. If you run the code, you will get the following output on your Terminal:

```
Number of training datapoints: 1600
Number of test datapoints: 400

Accuracy of the classifier: 0.735

Top 15 most informative words:
1. outstanding
2. insulting
3. vulnerable
4. ludicrous
5. uninvolving
6. astounding
7. avoids
8. fascination
9. symbol
10. seagal
11. affecting
12. anna
13. darker
14. animators
15. idiotic
```

The preceding screenshot shows the top 15 most informative words. If you scroll down your Terminal, you will see this:

```
Movie review predictions:

Review: The costumes in this movie were great
Predicted sentiment: Positive
Probability: 0.59

Review: I think the story was terrible and the characters were very weak
Predicted sentiment: Negative
Probability: 0.8

Review: People say that the director of the movie is amazing
Predicted sentiment: Positive
Probability: 0.6

Review: This is such an idiotic movie. I will not recommend it to anyone.
Predicted sentiment: Negative
Probability: 0.87
```

We can see and verify intuitively that the predictions are correct.

Topic modeling using Latent Dirichlet Allocation

Topic modeling is the process of identifying patterns in text data that correspond to a topic. If the text contains multiple topics, then this technique can be used to identify and separate those themes within the input text. We do this to uncover hidden thematic structure in the given set of documents.

Topic modeling helps us to organize our documents in an optimal way, which can then be used for analysis. One thing to note about topic modeling algorithms is that we don't need any labeled data. It is like unsupervised learning where it will identify the patterns on its own. Given the enormous volumes of text data generated on the Internet, topic modeling becomes very important because it enables us to summarize all this data, which would otherwise not be possible.

Latent Dirichlet Allocation is a topic modeling technique where the underlying intuition is that a given piece of text is a combination of multiple topics. Let's consider the following sentence – *Data visualization is an important tool in financial analysis*. This sentence has multiple topics like data, visualization, finance, and so on. This particular combination helps us identify this text in a large document. In essence, it is a statistical model that tries to capture this idea and create a model based on it. The model assumes that documents are generated from a random process based on these topics. A *topic* is basically a distribution over a fixed vocabulary of words. Let's see how to do topic modeling in Python.

We will use a library called `gensim` in this section. We have already installed this library in the first section of this chapter. Make sure that you have it before you proceed. Create a new python file and import the following packages:

```
from nltk.tokenize import RegexpTokenizer
from nltk.corpus import stopwords
from nltk.stem.snowball import SnowballStemmer
from gensim import models, corpora
```

Define a function to load the input data. The input file contains 10 line-separated sentences:

```
# Load input data
def load_data(input_file):
    data = []
    with open(input_file, 'r') as f:
        for line in f.readlines():
            data.append(line[:-1])

    return data
```

Define a function to process the input text. The first step is to tokenize it:

```
# Processor function for tokenizing, removing stop
# words, and stemming
def process(input_text):
    # Create a regular expression tokenizer
    tokenizer = RegexpTokenizer(r'\w+')
```

We then need to stem the tokenized text:

```
    # Create a Snowball stemmer
    stemmer = SnowballStemmer('english')
```

We need to remove the stop words from the input text because they don't add information. Let's get the list of stop-words:

```
    # Get the list of stop words
    stop_words = stopwords.words('english')
```

Tokenize the input string:

```
    # Tokenize the input string
    tokens = tokenizer.tokenize(input_text.lower())
```

Remove the stop-words:

```
    # Remove the stop words
    tokens = [x for x in tokens if not x in stop_words]
```

Stem the tokenized words and return the list:

```
    # Perform stemming on the tokenized words
    tokens_stemmed = [stemmer.stem(x) for x in tokens]

    return tokens_stemmed
```

Define the main function and load the input data from the file `data.txt` provided to you:

```
if __name__=='__main__':
    # Load input data
    data = load_data('data.txt')
```

Tokenize the text:

```
    # Create a list for sentence tokens
    tokens = [process(x) for x in data]
```

Create a dictionary based on the tokenized sentences:

```
    # Create a dictionary based on the sentence tokens
    dict_tokens = corpora.Dictionary(tokens)
```

Create a document term matrix using the sentence tokens:

```
    # Create a document-term matrix
    doc_term_mat = [dict_tokens.doc2bow(token) for token in tokens]
```

We need to provide the number of topics as the input parameter. In this case, we know that the input text has two distinct topics. Let's specify that.

```
    # Define the number of topics for the LDA model
    num_topics = 2
```

Generate the `LatentDirichlet` Model:

```
    # Generate the LDA model
    ldamodel = models.ldamodel.LdaModel(doc_term_mat,
            num_topics=num_topics, id2word=dict_tokens, passes=25)
```

Print the top 5 contributing words for each topic:

```
    num_words = 5
    print('\nTop ' + str(num_words) + ' contributing words to each topic:')
    for item in ldamodel.print_topics(num_topics=num_topics,
num_words=num_words):
        print('\nTopic', item[0])

        # Print the contributing words along with their relative
contributions
        list_of_strings = item[1].split(' + ')
        for text in list_of_strings:
            weight = text.split('*')[0]
            word = text.split('*')[1]
            print(word, '==>', str(round(float(weight) * 100, 2)) + '%')
```

The full code is given in the file `topic_modeler.py`. If you run the code, you will get the following output on your Terminal:

```
Top 5 contributing words to each topic:

Topic 0
mathemat ==> 2.7%
structur ==> 2.6%
set ==> 2.6%
formul ==> 2.6%
tradit ==> 1.6%

Topic 1
empir ==> 4.7%
expand ==> 3.3%
time ==> 2.0%
peopl ==> 2.0%
histor ==> 2.0%
```

We can see that it does a reasonably good job of separating the two topics – mathematics and history. If you look into the text, you can verify that each sentence is either about mathematics or history.

Summary

In this chapter, we learned about the various underlying concepts in natural language processing. We discussed tokenization and how to separate input text into multiple tokens. We learned how to reduce words to their base forms using stemming and lemmatization. We implemented a text chunker to divide input text into chunks based on predefined conditions.

We discussed the Bag of Words model and built a document term matrix for input text. We then learnt how to categorize text using machine learning. We constructed a gender identifier using a heuristic. We used machine learning to analyze the sentiments of movie reviews. We discussed topic modeling and implemented a system to identify topics in a given document.

In the next chapter, we will learn how to model sequential data using Hidden Markov Models and then use it to analyze stock market data.

11
Probabilistic Reasoning for Sequential Data

In this chapter, we are going to learn how to build sequence learning models. We will learn how to handle time-series data in Pandas. We will understand how to slice time-series data and perform various operations on it. We will discuss how to extract various stats from time-series data on a rolling basis. We will learn about Hidden Markov Models and then implement a system to build those models. We will understand how to use Conditional Random Fields to analyze sequences of alphabets. We will discuss how to analyze stock market data using the techniques learnt so far.

By the end of this chapter, you will learn about:

- Handling time-series data with Pandas
- Slicing time-series data
- Operating on time-series data
- Extracting statistics from time-series data
- Generating data using Hidden Markov Models
- Identifying alphabet sequences with Conditional Random Fields
- Stock market analysis

Understanding sequential data

In the world of machine learning, we encounter many types of data, such as images, text, video, sensor readings, and so on. Different types of data require different types of modeling techniques. Sequential data refers to data where the ordering is important. Time-series data is a particular manifestation of sequential data.

It is basically time-stamped values obtained from any data source such as sensors, microphones, stock markets, and so on. Time-series data has a lot of important characteristics that need to be modeled in order to effectively analyze the data.

The measurements that we encounter in time-series data are taken at regular time intervals and correspond to predetermined parameters. These measurements are arranged on a timeline for storage, and the order of their appearance is very important. We use this order to extract patterns from the data.

In this chapter, we will see how to build models that describe the given time-series data or any sequence in general. These models are used to understand the behavior of the time series variable. We then use these models to predict the future based on past behavior.

Time-series data analysis is used extensively in financial analysis, sensor data analysis, speech recognition, economics, weather forecasting, manufacturing, and many more. We will explore a variety of scenarios where we encounter time-series data and see how we can build a solution. We will be using a library called `Pandas` to handle all the time-series related operations. We will also use a couple of other useful packages like `hmmlearn` and `pystruct` during this chapter. Make sure you install them before you proceed.

You can install them by running the following commands on your Terminal:

```
$ pip3 install pandas
$ pip3 install hmmlearn
$ pip3 install pystruct
$ pip3 install cvxopt
```

If you get an error when installing `cvxopt`, you will find further instructions at `http://cvxopt.org/install`. Now that you have successfully installed the packages, let's go ahead to the next section.

Handling time-series data with Pandas

Let's get started by learning how to handle time-series data in Pandas. In this section, we will convert a sequence of numbers into time series data and visualize it. Pandas provides options to add timestamps, organize data, and then efficiently operate on it.

Create a new Python file and import the following packages:

```
import numpy as np
import matplotlib.pyplot as plt
import pandas as pd
```

Define a function to read the data from the input file. The parameter `index` indicates the column number that contains the relevant data:

```
def read_data(input_file, index):
    # Read the data from the input file
    input_data = np.loadtxt(input_file, delimiter=',')
```

Define a `lambda` function to convert strings to Pandas date format:

```
    # Lambda function to convert strings to Pandas date format
    to_date = lambda x, y: str(int(x)) + '-' + str(int(y))
```

Use this `lambda` function to get the start date from the first line in the input file:

```
    # Extract the start date
    start = to_date(input_data[0, 0], input_data[0, 1])
```

`Pandas` library needs the end date to be exclusive when we perform operations, so we need to increase the date field in the last line by one month:

```
    # Extract the end date
    if input_data[-1, 1] == 12:
        year = input_data[-1, 0] + 1
        month = 1
    else:
        year = input_data[-1, 0]
        month = input_data[-1, 1] + 1

    end = to_date(year, month)
```

Create a list of indices with dates using the start and end dates with a monthly frequency:

```
    # Create a date list with a monthly frequency
    date_indices = pd.date_range(start, end, freq='M')
```

Create pandas data series using the timestamps:

```
    # Add timestamps to the input data to create time-series data
    output = pd.Series(input_data[:, index], index=date_indices)

    return output
```

Define the main function and specify the input file:

```
if __name__=='__main__':
    # Input filename
    input_file = 'data_2D.txt'
```

Specify the columns that contain the data:

```
# Specify the columns that need to be converted
# into time-series data
indices = [2, 3]
```

Iterate through the columns and read the data in each column:

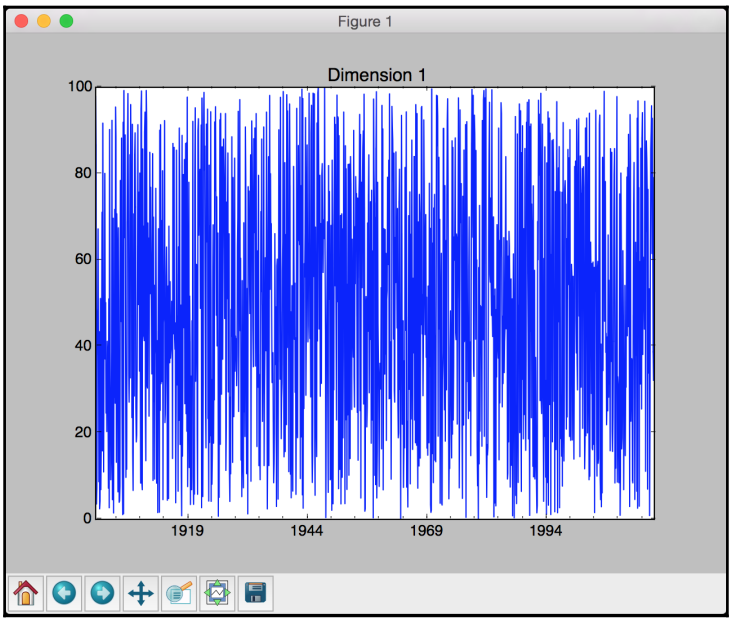
```
# Iterate through the columns and plot the data
for index in indices:
    # Convert the column to timeseries format
    timeseries = read_data(input_file, index)
```

Plot the time-series data:

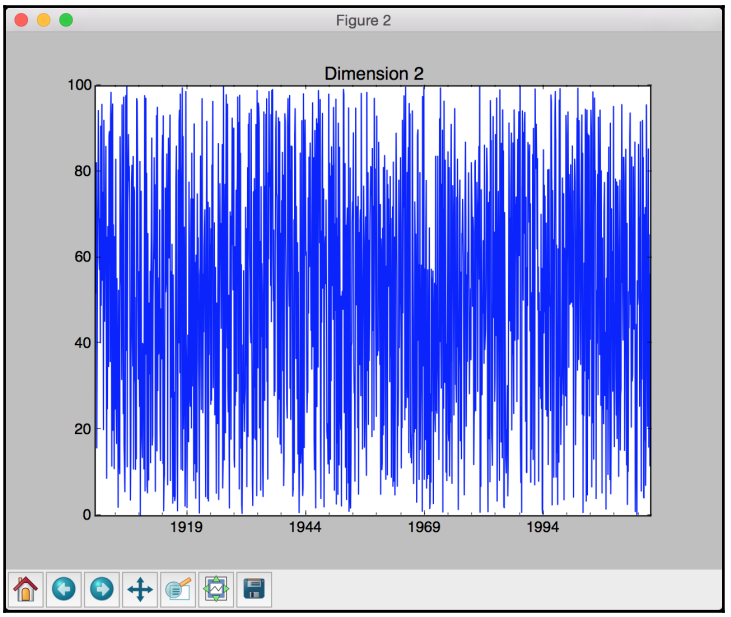
```
    # Plot the data
    plt.figure()
    timeseries.plot()
    plt.title('Dimension ' + str(index - 1))

plt.show()
```

The full code is given in the file `timeseries.py`. If you run the code, you will see two screenshots.

The following screenshot indicates the data in the first dimension:



The second screenshot indicates the data in the second dimension:



Slicing time-series data

Now that we know how to handle time-series data, let's see how we can slice it. The process of slicing refers to dividing the data into various sub-intervals and extracting relevant information. This is very useful when you are working with time-series datasets. Instead of using indices, we will use timestamp to slice our data.

Create a new Python file and import the following packages:

```
import numpy as np
import matplotlib.pyplot as plt
import pandas as pd

from timeseries import read_data
```

Load the third column (zero-indexed) from the input data file:

```
# Load input data
index = 2
data = read_data('data_2D.txt', index)
```

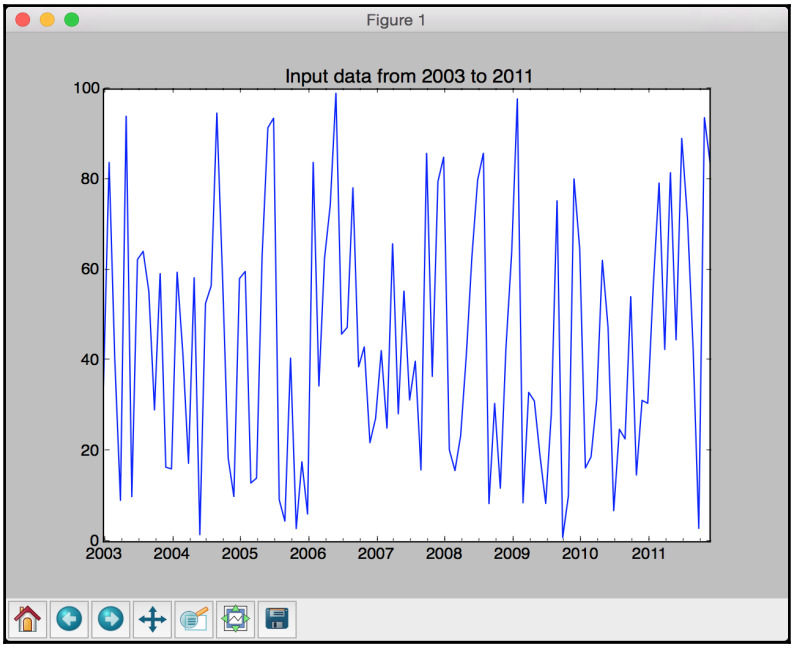
Define the start and end years, and then plot the data with year-level granularity:

```
# Plot data with year-level granularity
start = '2003'
end = '2011'
plt.figure()
data[start:end].plot()
plt.title('Input data from ' + start + ' to ' + end)
```

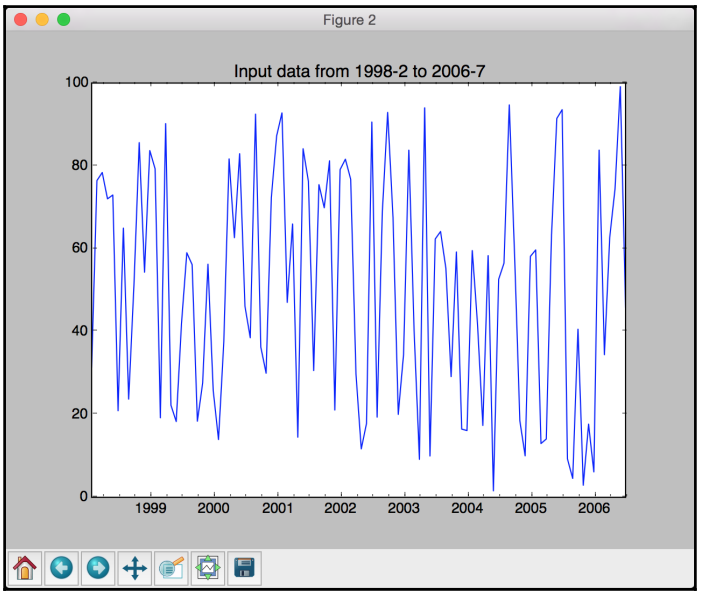
Define the start and end months, and then plot the data with month-level granularity:

```
# Plot data with month-level granularity
start = '1998-2'
end = '2006-7'
plt.figure()
data[start:end].plot()
plt.title('Input data from ' + start + ' to ' + end)

plt.show()
```

The full code is given in the file `slicer.py`. If you run the code, you will see two figures. The first screenshot shows the data from *2003* to *2011*:



The second screenshot shows the data from *February 1998* to *July 2006*:



Operating on time-series data

Pandas allows us to operate on time-series data efficiently and perform various operations like filtering and addition. You can simply set some conditions and Pandas will filter the dataset and return the right subset. You can add two time-series variables as well. This allows us to build various applications quickly without having to reinvent the wheel.

Create a new Python file and import the following packages:

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt

from timeseries import read_data
```

Define the input filename:

```
# Input filename
input_file = 'data_2D.txt'
```

Load the third and fourth columns into separate variables:

```
# Load data
x1 = read_data(input_file, 2)
x2 = read_data(input_file, 3)
```

Create a Pandas dataframe object by naming the two dimensions:

```
# Create pandas dataframe for slicing
data = pd.DataFrame({'dim1': x1, 'dim2': x2})
```

Plot the data by specifying the start and end years:

```
# Plot data
start = '1968'
end = '1975'
data[start:end].plot()
plt.title('Data overlapped on top of each other')
```

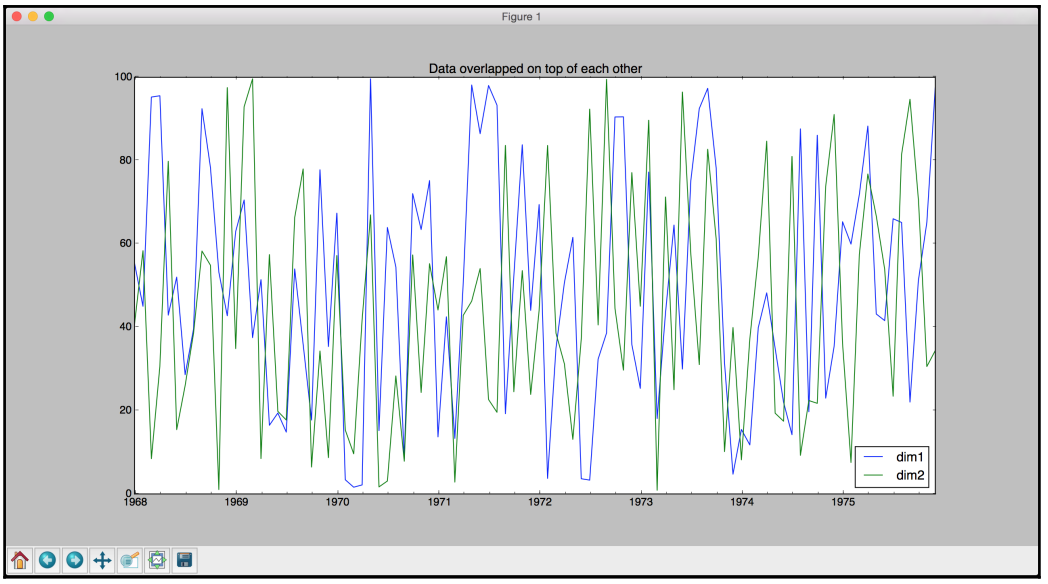
Filter the data using conditions and then display it. In this case, we will take all the datapoints in `dim1` that are less than `45` and all the values in `dim2` that are greater than `30`:

```
# Filtering using conditions
# - 'dim1' is smaller than a certain threshold
# - 'dim2' is greater than a certain threshold
data[(data['dim1'] < 45) & (data['dim2'] > 30)].plot()
plt.title('dim1 < 45 and dim2 > 30')
```

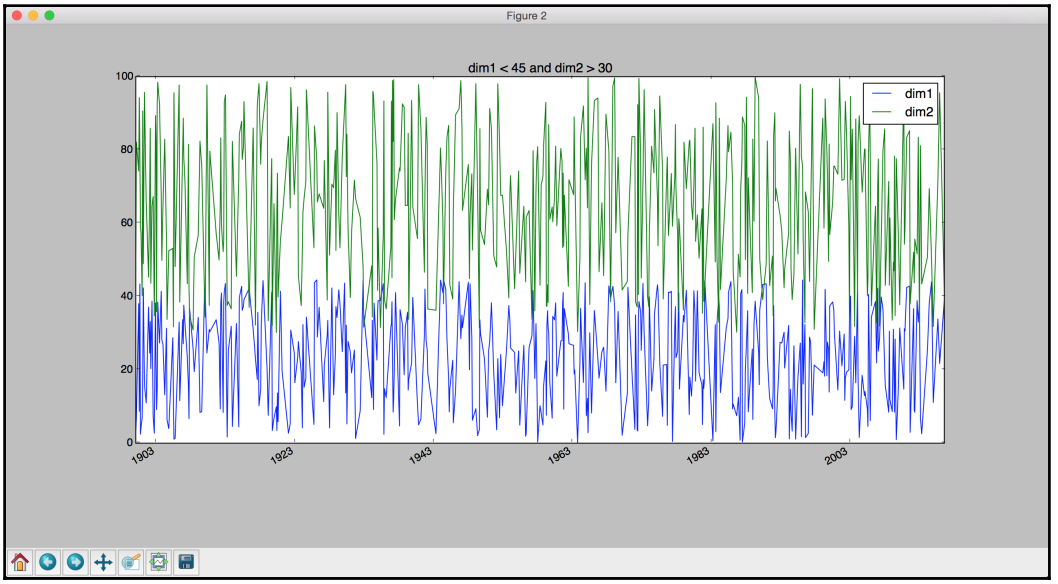
We can also add two series in `Pandas`. Let's add `dim1` and `dim2` between the given start and end dates:

```
# Adding two dataframes
plt.figure()
diff = data[start:end]['dim1'] + data[start:end]['dim2']
diff.plot()
plt.title('Summation (dim1 + dim2)')

plt.show()
```

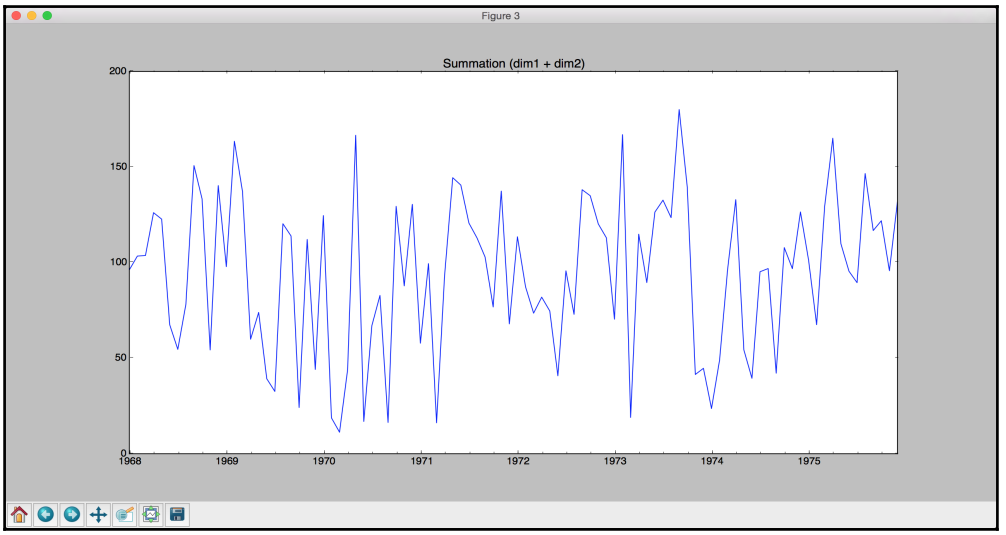
The full code is given in the file `operator.py`. If you run the code, you will see three screenshots. The first screenshot shows the data from *1968* to *1975:*



The second screenshot shows the filtered data:



The third screenshot shows the summation result:



Extracting statistics from time-series data

In order to extract meaningful insights from time-series data, we have to extract statistics from it. These stats can be things like mean, variance, correlation, maximum value, and so on. These stats have to be computed on a rolling basis using a window. We use a predetermined window size and keep computing these stats. When we visualize the stats over time, we will see interesting patterns. Let's see how to extract these stats from time-series data.

Create a new Python file and import the following packages:

```
import numpy as np
import matplotlib.pyplot as plt
import pandas as pd

from timeseries import read_data
```

Define the input filename:

```
# Input filename
input_file = 'data_2D.txt'
```

Load the third and fourth columns into separate variables:

```
# Load input data in time series format
x1 = read_data(input_file, 2)
x2 = read_data(input_file, 3)
```

Create a pandas dataframe by naming the two dimensions:

```
# Create pandas dataframe for slicing
data = pd.DataFrame({'dim1': x1, 'dim2': x2})
```

Extract maximum and minimum values along each dimension:

```
# Extract max and min values
print('\nMaximum values for each dimension:')
print(data.max())
print('\nMinimum values for each dimension:')
print(data.min())
```

Extract the overall mean and the row-wise mean for the first *12* rows:

```
# Extract overall mean and row-wise mean values
print('\nOverall mean:')
print(data.mean())
print('\nRow-wise mean:')
print(data.mean(1)[:12])
```

Plot the rolling mean using a window size of `24`:

```
# Plot the rolling mean using a window size of 24
data.rolling(center=False, window=24).mean().plot()
plt.title('Rolling mean')
```

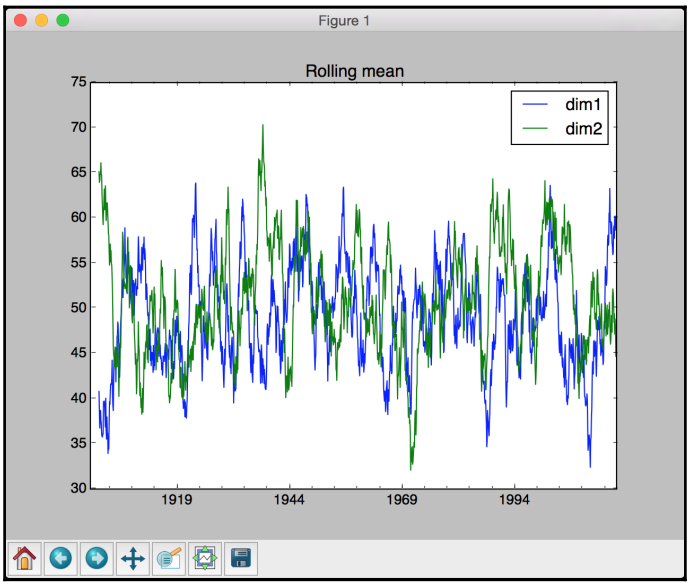
Print the correlation coefficients:

```
# Extract correlation coefficients
print('\nCorrelation coefficients:\n', data.corr())
```

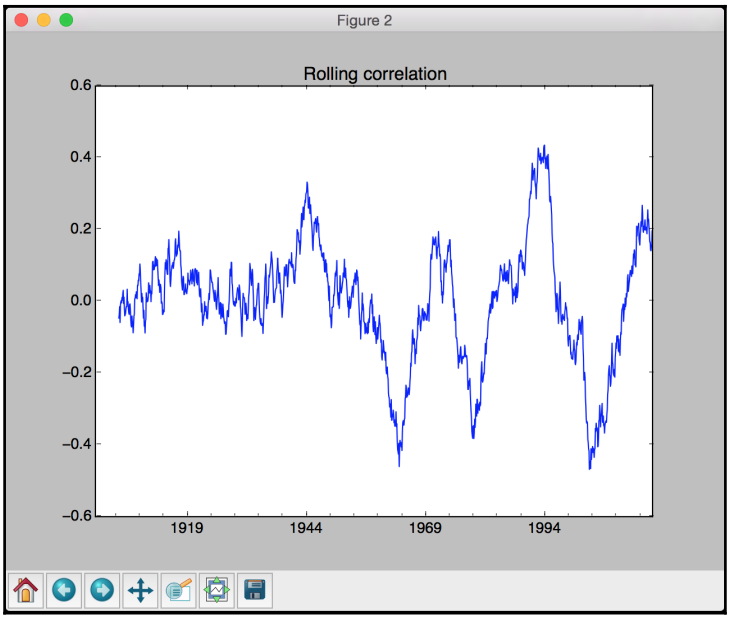
Plot the rolling correlation using a window size of *60*:

```
# Plot rolling correlation using a window size of 60
plt.figure()
plt.title('Rolling correlation')
data['dim1'].rolling(window=60).corr(other=data['dim2']).plot()

plt.show()
```

The full code is given in the file `stats_extractor.py`. If you run the code, you will see two screenshots. The first screenshot shows the rolling mean:



The second screenshot shows the rolling correlation:



You will see the following on your Terminal:

```
Maximum values for each dimension:
dim1    99.98
dim2    99.97
dtype: float64

Minimum values for each dimension:
dim1    0.18
dim2    0.16
dtype: float64

Overall mean:
dim1    49.030541
dim2    50.983291
dtype: float64
```

If you scroll down, you will see row-wise mean values and the correlation coefficients printed on your Terminal:

```
Row-wise mean:
1900-01-31    85.595
1900-02-28    75.310
1900-03-31    27.700
1900-04-30    44.675
1900-05-31    31.295
1900-06-30    44.160
1900-07-31    67.415
1900-08-31    56.160
1900-09-30    51.495
1900-10-31    61.260
1900-11-30    30.925
1900-12-31    30.785
Freq: M, dtype: float64

Correlation coefficients:
          dim1     dim2
dim1  1.00000  0.00627
dim2  0.00627  1.00000
```

The correlation coefficients in the preceding figures indicate the level of correlation of each dimension with all the other dimensions. A correlation of `1.0` indicates perfect correlation, whereas a correlation of `0.0` indicates that they the variables are not related to each other.

Generating data using Hidden Markov Models

A **Hidden Markov Model** (**HMM**) is a powerful analysis technique for analyzing sequential data. It assumes that the system being modeled is a Markov process with hidden states. This means that the underlying system can be one among a set of possible states. It goes through a sequence of state transitions, thereby producing a sequence of outputs. We can only observe the outputs but not the states. Hence these states are hidden from us. Our goal is to model the data so that we can infer the state transitions of unknown data.

In order to understand HMMs, let's consider the example of a salesman who has to travel between the following three cities for his job — London, Barcelona, and New York. His goal is to minimize the traveling time so that he can be more efficient. Considering his work commitments and schedule, we have a set of probabilities that dictate the chances of going from city *X* to city *Y*. In the information given below, *P(X -> Y)* indicates the probability of going from city *X* to city *Y*:

P(London -> London) = 0.10

P(London -> Barcelona) = 0.70

P(London -> NY) = 0.20

P(Barcelona -> Barcelona) = 0.15

P(Barcelona -> London) = 0.75

P(Barcelona -> NY) = 0.10

P(NY -> NY) = 0.05

P(NY -> London) = 0.60

P(NY -> Barcelona) = 0.35

Let's represent this information with a transition matrix:

London Barcelona NY

London 0.10 0.70 0.20

Barcelona 0.75 0.15 0.10

NY 0.60 0.35 0.05

Now that we have all the information, let's go ahead and set the problem statement. The salesman starts his journey on Tuesday from London and he has to plan something on Friday. But that will depend on where he is. What is the probability that he will be in Barcelona on Friday? This table will help us figure it out.

If we do not have a Markov Chain to model this problem, then we will not know what his travel schedule looks like. Our goal is to say with a good amount of certainty that he will be in a particular city on a given day. If we denote the transition matrix by *T* and the current day by *X(i)*, then:

X(i+1) = X(i).T

In our case, Friday is *3* days away from Tuesday. This means we have to compute *X(i+3)*. The computations will looks like this:

X(i+1) = X(i).T

X(i+2) = X(i+1).T

X(i+3) = X(i+2).T

So in essence:

X(i+3) = X(i).T^3

We need to set *X(i)* as given here:

X(i) = [0.10 0.70 0.20]

The next step is to compute the cube of the matrix. There are many tools available online to perform matrix operations such as `http://matrix.reshish.com/multiplication.php`. If you do all the matrix calculations, then you will see that you will get the following probabilities for Thursday:

P(London) = 0.31

P(Barcelona) = 0.53

P(NY) = 0.16

We can see that there is a higher chance of him being in Barcelona than in any other city. This makes geographical sense as well because Barcelona is closer to London compared to New York. Let's see how to model HMMs in Python.

Create a new Python file and import the following packages:

```
import datetime

import numpy as np
import matplotlib.pyplot as plt
from hmmlearn.hmm import GaussianHMM

from timeseries import read_data
```

Load data from the input file:

```
# Load input data
data = np.loadtxt('data_1D.txt', delimiter=',')
```

Extract the third column for training:

```
# Extract the data column (third column) for training
X = np.column_stack([data[:, 2]])
```

Create a Gaussian HMM with 5 components and diagonal covariance:

```
# Create a Gaussian HMM
num_components = 5
hmm = GaussianHMM(n_components=num_components,
        covariance_type='diag', n_iter=1000)
```

Train the HMM:

```
# Train the HMM
print('\nTraining the Hidden Markov Model...')
hmm.fit(X)
```

Print the mean and variance values for each component of the HMM:

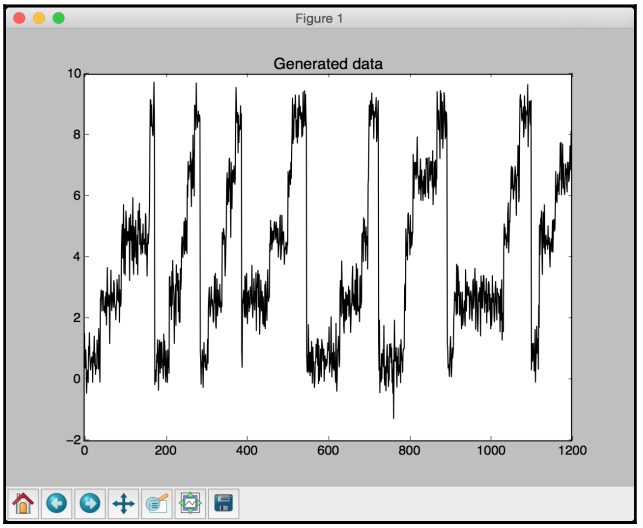
```
# Print HMM stats
print('\nMeans and variances:')
for i in range(hmm.n_components):
    print('\nHidden state', i+1)
    print('Mean =', round(hmm.means_[i][0], 2))
    print('Variance =', round(np.diag(hmm.covars_[i])[0], 2))
```

Generate `1200` samples using the trained HMM model and plot them:

```
# Generate data using the HMM model
num_samples = 1200
generated_data, _ = hmm.sample(num_samples)
plt.plot(np.arange(num_samples), generated_data[:, 0], c='black')
plt.title('Generated data')
```

```
plt.show()
```

The full code is given in the file `hmm.py`. If you run the code, you will see the following screenshot that shows the 1200 generated samples:



You will see the following printed on your Terminal:

```
Training the Hidden Markov Model...

Means and variances:

Hidden state 1
Mean = 4.6
Variance = 0.25

Hidden state 2
Mean = 6.59
Variance = 0.25

Hidden state 3
Mean = 0.6
Variance = 0.25

Hidden state 4
Mean = 8.6
Variance = 0.26

Hidden state 5
Mean = 2.6
Variance = 0.26
```

Identifying alphabet sequences with Conditional Random Fields

Conditional Random Fields (**CRF**s) are probabilistic models that are frequently used to analyze structured data. We use them to label and segment sequential data in various forms. One thing to note about CRFs is that they are discriminative models. This is in contrast to HMMs, which are generative models.

We can define a conditional probability distribution over a labeled sequence of measurements. We use this framework to build a CRF model. In HMMs, we have to define a joint distribution over the observation sequence and the labels.

One of the main advantages of CRFs is that they are conditional by nature. This is not the case with HMMs. CRFs do not assume any independence between output observations. HMMs assume that the output at any given time is statistically independent of the previous outputs. HMMs need this assumption to ensure that the inference process works in a robust way. But this assumption is not always true! Real world data is filled with temporal dependencies.

CRFs tend to outperform HMMs in a variety of applications such as natural language processing, speech recognition, biotechnology, and so on. In this section, we will discuss how to use CRFs to analyze sequences of alphabets. Create a new python file and import the following packages:

```
import os
import argparse
import string
import pickle

import numpy as np
import matplotlib.pyplot as plt
from pystruct.datasets import load_letters
from pystruct.models import ChainCRF
from pystruct.learners import FrankWolfeSSVM
```

Define a function to parse the input arguments. We can pass the `C` value as the input parameter. The C parameter controls how much we want to penalize misclassification. A higher value of C would mean that we are imposing a higher penalty for misclassification during training, but we might end up overfitting the model. On the other hand, if we choose a lower value for C, we are allowing the model to generalize well. But this also means that we are imposing a lower penalty for misclassification during training data points.

```
def build_arg_parser():
    parser = argparse.ArgumentParser(description='Trains a Conditional\
            Random Field classifier')
    parser.add_argument("--C", dest="c_val", required=False, type=float,
            default=1.0, help='C value to be used for training')
    return parser
```

Define a class to handle all the functionality of building the CRF model. We will use a chain CRF model with `FrankWolfeSSVM`:

```
# Class to model the CRF
class CRFModel(object):
    def __init__(self, c_val=1.0):
        self.clf = FrankWolfeSSVM(model=ChainCRF(),
                C=c_val, max_iter=50)
```

Define a function to load the training data:

```
    # Load the training data
    def load_data(self):
        alphabets = load_letters()
        X = np.array(alphabets['data'])
        y = np.array(alphabets['labels'])
        folds = alphabets['folds']

        return X, y, folds
```

Define a function to train the CRF model:

```
    # Train the CRF
    def train(self, X_train, y_train):
        self.clf.fit(X_train, y_train)
```

Define a function to evaluate the accuracy of the CRF model:

```
    # Evaluate the accuracy of the CRF
    def evaluate(self, X_test, y_test):
        return self.clf.score(X_test, y_test)
```

Define a function to run the trained CRF model on an unknown datapoint:

```
    # Run the CRF on unknown data
    def classify(self, input_data):
        return self.clf.predict(input_data)[0]
```

Define a function to extract a substring from the alphabets based on a list of indices:

```
# Convert indices to alphabets
def convert_to_letters(indices):
```

```
        # Create a numpy array of all alphabets
        alphabets = np.array(list(string.ascii_lowercase))
```

Extract the letters:

```
        # Extract the letters based on input indices
        output = np.take(alphabets, indices)
        output = ''.join(output)

        return output
```

Define the main function and parse the input arguments:

```
if __name__=='__main__':
    args = build_arg_parser().parse_args()
    c_val = args.c_val
```

Create the CRF model object:

```
        # Create the CRF model
        crf = CRFModel(c_val)
```

Load the input data and separate it into train and test sets:

```
        # Load the train and test data
        X, y, folds = crf.load_data()
        X_train, X_test = X[folds == 1], X[folds != 1]
        y_train, y_test = y[folds == 1], y[folds != 1]
```

Train the CRF model:

```
        # Train the CRF model
        print('\nTraining the CRF model...')
        crf.train(X_train, y_train)
```

Evaluate the accuracy of the CRF model and print it:

```
        # Evaluate the accuracy
        score = crf.evaluate(X_test, y_test)
        print('\nAccuracy score =', str(round(score*100, 2)) + '%')
```

Run it on some test datapoints and print the output:

```
        indices = range(3000, len(y_test), 200)
        for index in indices:
            print("\nOriginal  =", convert_to_letters(y_test[index]))
            predicted = crf.classify([X_test[index]])
            print("Predicted =", convert_to_letters(predicted))
```

The full code is given in the file `crf.py`. If you run the code, you will see the following output on your Terminal:

```
Training the CRF model...

Accuracy score = 77.93%

Original  = rojections
Predicted = rojectiong

Original  = uff
Predicted = ufr

Original  = kiing
Predicted = kiing

Original  = ecompress
Predicted = ecomertig

Original  = uzz
Predicted = vex

Original  = poiling
Predicted = aciting
```

If you scroll to the end, you will see the following on your Terminal:

```
Original  = abulously
Predicted = abuloualy

Original  = ormalization
Predicted = ormalisation

Original  = ake
Predicted = aka

Original  = afeteria
Predicted = ateteria

Original  = obble
Predicted = obble

Original  = hadow
Predicted = habow

Original  = ndustrialized
Predicted = ndusqrialyled

Original  = ympathetically
Predicted = ympnshetically
```

As we can see, it predicts most of the words correctly.

Stock market analysis

We will analyze stock market data in this section using Hidden Markov Models. This is an example where the data is already organized timestamped. We will use the dataset available in the `matplotlib` package. The dataset contains the stock values of various companies over the years. Hidden Markov models are generative models that can analyze such time series data and extract the underlying structure. We will use this model to analyze stock price variations and generate the outputs.

Create a new python file and import the following packages:

```
import datetime
import warnings

import numpy as np
import matplotlib.pyplot as plt
from matplotlib.finance import quotes_historical_yahoo_ochl\
        as quotes_yahoo
from hmmlearn.hmm import GaussianHMM
```

Load historical stock market quotes from September 4, 1970 to May 17, 2016. You are free to choose any date range you wish.

```
# Load historical stock quotes from matplotlib package
start = datetime.date(1970, 9, 4)
end = datetime.date(2016, 5, 17)
stock_quotes = quotes_yahoo('INTC', start, end)
```

Extract the closing quote each day and the volume of shares traded that day:

```
# Extract the closing quotes everyday
closing_quotes = np.array([quote[2] for quote in stock_quotes])

# Extract the volume of shares traded everyday
volumes = np.array([quote[5] for quote in stock_quotes])[1:]
```

Take the percentage difference of closing quotes each day:

```
# Take the percentage difference of closing stock prices
diff_percentages = 100.0 * np.diff(closing_quotes) / closing_quotes[:-1]
```

Since the differencing reduces the length of the array by `1`, you need to adjust the date array too:

```
# Take the list of dates starting from the second value
dates = np.array([quote[0] for quote in stock_quotes], dtype=np.int)[1:]
```

Stack the two data columns to create the training dataset:

```
# Stack the differences and volume values column-wise for training
training_data = np.column_stack([diff_percentages, volumes])
```

Create and train the Gaussian HMM with 7 components and diagonal covariance:

```
# Create and train Gaussian HMM
hmm = GaussianHMM(n_components=7, covariance_type='diag', n_iter=1000)
with warnings.catch_warnings():
    warnings.simplefilter('ignore')
    hmm.fit(training_data)
```

Use the trained HMM model to generate 300 samples. You can choose to generate any number of samples you want.

```
# Generate data using the HMM model
num_samples = 300
samples, _ = hmm.sample(num_samples)
```

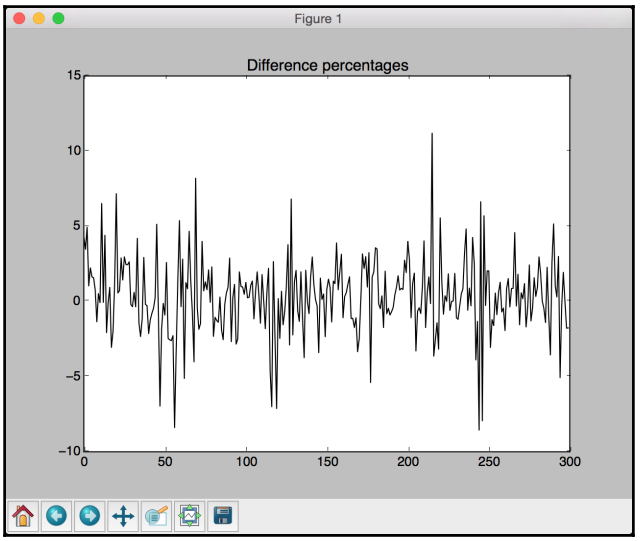
Plot the generated values for difference percentages:

```
# Plot the difference percentages
plt.figure()
plt.title('Difference percentages')
plt.plot(np.arange(num_samples), samples[:, 0], c='black')
```

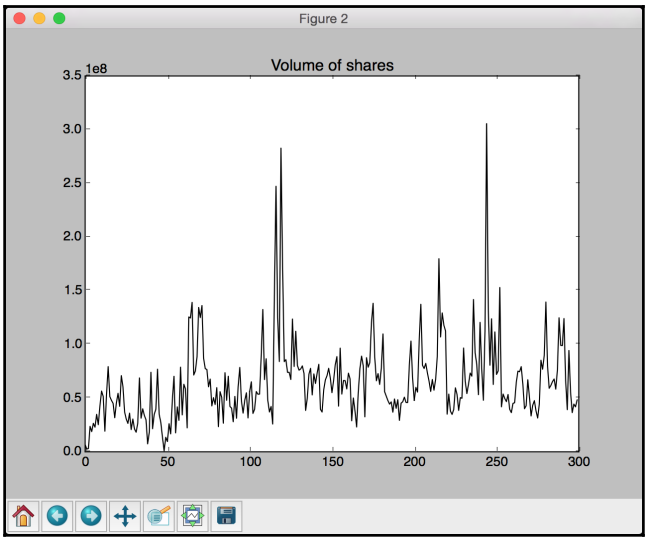
Plot the generated values for volume of shares traded:

```
# Plot the volume of shares traded
plt.figure()
plt.title('Volume of shares')
plt.plot(np.arange(num_samples), samples[:, 1], c='black')
plt.ylim(ymin=0)

plt.show()
```

The full code is given in the file `stock_market.py`. If you run the code, you will see the following two screenshots. The first screenshot shows the difference percentages generated by the HMM:



The second screenshot shows the values generated by the HMM for volume of shares traded:



Summary

In this chapter, we learned how to build sequence learning models. We understood how to handle time-series data in Pandas. We discussed how to slice time-series data and perform various operations on it. We learned how to extract various stats from time-series data in a rolling manner. We understood Hidden Markov Models and then implemented a system to build that model.

We discussed how to use Conditional Random Fields to analyze sequences of alphabets. We learned how to analyze stock market data using various techniques. In the next chapter, we will learn about speech recognition and build a system to automatically recognize spoken words.

12
Building A Speech Recognizer

In this chapter, we are going to learn about speech recognition. We will discuss how to work with speech signals and understand how to visualize various audio signals. By utilizing various techniques to process speech signals, we will learn how to build a speech recognition system.

By the end of this chapter, you will know about:

- Working with speech signals
- Visualizing audio signals
- Transforming audio signals to frequency domain
- Generating audio signals
- Synthesizing tones
- Extracting speech features
- Recognizing spoken words

Working with speech signals

Speech recognition is the process of understanding the words that are spoken by humans. The speech signals are captured using a microphone and the system tries to understand the words that are being captured. Speech recognition is used extensively in human computer interaction, smartphones, speech transcription, biometric systems, security, and so on.

It is important to understand the nature of speech signals before we analyze them. These signals happen to be complex mixtures of various signals. There are many different aspects of speech that contribute to its complexity. These things include emotion, accent, language, noise, and so on.

Hence it becomes difficult to robustly define a set of rules to analyze speech signals. But humans are really good at understanding speech even though it has so many variations. We seem to do it with relative ease. If we want our machines to do the same, we need to help them understand speech the same way we do.

Researchers work on various aspects and applications of speech, such as understanding spoken words, identifying who the speaker is, recognizing emotions, identifying accents, and so on. In this chapter, we will focus on understanding spoken words. Speech recognition represents an important step in the field of human computer interaction. If we want to build cognitive robots that can interact with humans, they need to talk to us in natural language. This is the reason that automatic speech recognition has been the center of attention for many researchers in recent years. Let's go ahead and see how to deal with speech signals and build a speech recognizer.

Visualizing audio signals

Let's see how to visualize an audio signal. We will learn how to read an audio signal from a file and work with it. This will help us understand how an audio signal is structured. When audio files are recorded using a microphone, they are sampling the actual audio signals and storing the digitized versions. The real audio signals are continuous valued waves, which means we cannot store them as they are. We need to sample the signal at a certain frequency and convert it into discrete numerical form.

Most commonly, speech signals are sampled at 44,100 Hz. This means that each second of the speech signal is broken down into 44,100 parts and the values at each of these timestamps is stored in an output file. We save the value of the audio signal every 1/44,100 seconds. In this case, we say that the sampling frequency of the audio signal is 44,100 Hz. By choosing a high sampling frequency, it will appear like the audio signal is continuous when humans listen to it. Let's go ahead and visualize an audio signal.

Create a new Python file and import the following packages:

```
import numpy as np
import matplotlib.pyplot as plt
from scipy.io import wavfile
```

Read the input audio file using the `wavefile.read` method. It returns two values – sampling frequency and the audio signal:

```
# Read the audio file
sampling_freq, signal = wavfile.read('random_sound.wav')
```

Print the shape of the signal, datatype, and the duration of the audio signal:

```
# Display the params
print('\nSignal shape:', signal.shape)
print('Datatype:', signal.dtype)
print('Signal duration:', round(signal.shape[0] / float(sampling_freq), 2),
'seconds')
```

Normalize the signal:

```
# Normalize the signal
signal = signal / np.power(2, 15)
```

Extract the first `50` values from the `numpy` array for plotting:

```
# Extract the first 50 values
signal = signal[:50]
```

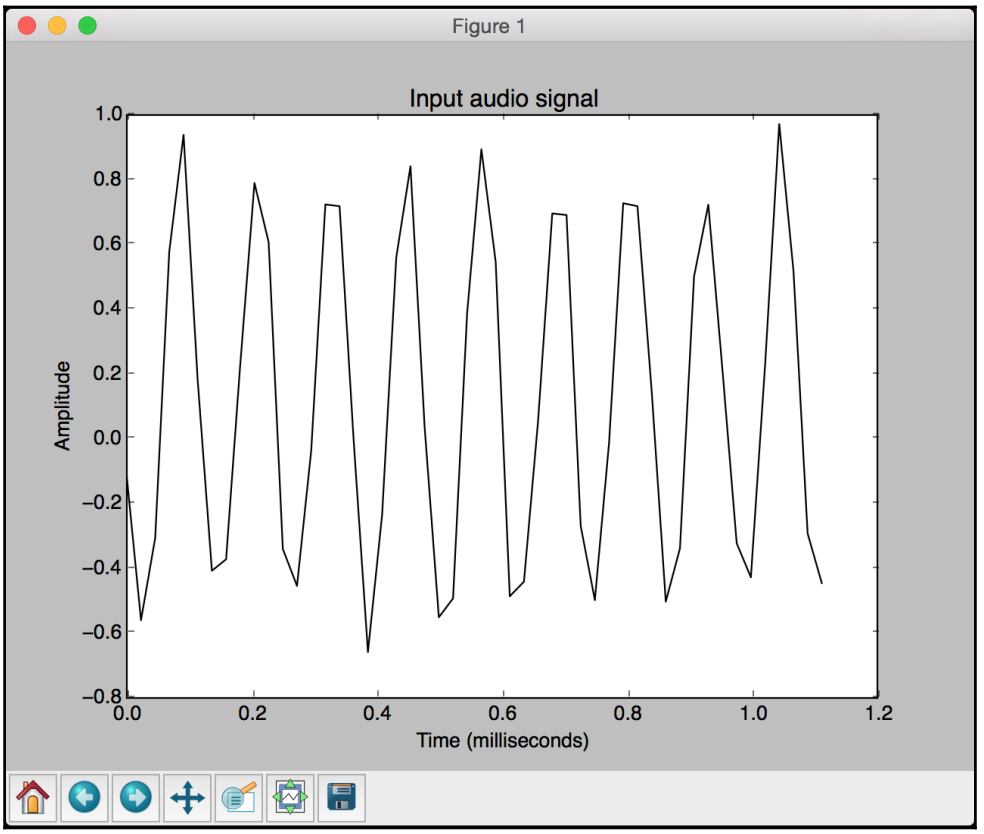
Construct the time axis in seconds for plotting:

```
# Construct the time axis in milliseconds
time_axis = 1000 * np.arange(0, len(signal), 1) / float(sampling_freq)
```

Plot the audio signal:

```
# Plot the audio signal
plt.plot(time_axis, signal, color='black')
plt.xlabel('Time (milliseconds)')
plt.ylabel('Amplitude')
plt.title('Input audio signal')
plt.show()
```

The full code is given in the file `audio_plotter.py`. If you run the code, you will see the following screenshot:



The preceding screenshot shows the first 50 samples of the input audio signal. You will see the following output on your Terminal:

```
Signal shape: (132300,)
Datatype: int16
Signal duration: 3.0 seconds
```

The output printed in the preceding figure shows the information that we extracted from the signal.

Transforming audio signals to the frequency domain

In order to analyze audio signals, we need to understand the underlying frequency components. This gives us insights into how to extract meaningful information from this signal. Audio signals are composed of a mixture of sine waves of varying frequencies, phases, and amplitudes.

If we dissect the frequency components, we can identify a lot of characteristics. Any given audio signal is characterized by its distribution in the frequency spectrum. In order to convert a time domain signal into the frequency domain, we need to use a mathematical tool like Fourier Transform. If you need a quick refresher on **Fourier Transform**, you can check out this link: `http://www.thefouriertransform.com`. Let's see how to transform an audio signal from the time domain to the frequency domain.

Create a new Python file and import the following packages:

```
import numpy as np
import matplotlib.pyplot as plt
from scipy.io import wavfile
```

Read the input audio file using the `wavefile.read` method. It returns two values – sampling frequency and the audio signal:

```
# Read the audio file
sampling_freq, signal = wavfile.read('spoken_word.wav')
```

Normalize the audio signal:

```
# Normalize the values
signal = signal / np.power(2, 15)
```

Extract the length and half-length of the signal:

```
# Extract the length of the audio signal
len_signal = len(signal)

# Extract the half length
len_half = np.ceil((len_signal + 1) / 2.0).astype(np.int)
```

Apply Fourier transform to the signal:

```
# Apply Fourier transform
freq_signal = np.fft.fft(signal)
```

Normalize the frequency domain signal and take the square:

```
# Normalization
freq_signal = abs(freq_signal[0:len_half]) / len_signal

# Take the square
freq_signal **= 2
```

Adjust the Fourier transformed signal for even and odd cases:

```
# Extract the length of the frequency transformed signal
len_fts = len(freq_signal)

# Adjust the signal for even and odd cases
if len_signal % 2:
    freq_signal[1:len_fts] *= 2
else:
    freq_signal[1:len_fts-1] *= 2
```

Extract the power signal in `dB`:

```
# Extract the power value in dB
signal_power = 10 * np.log10(freq_signal)
```

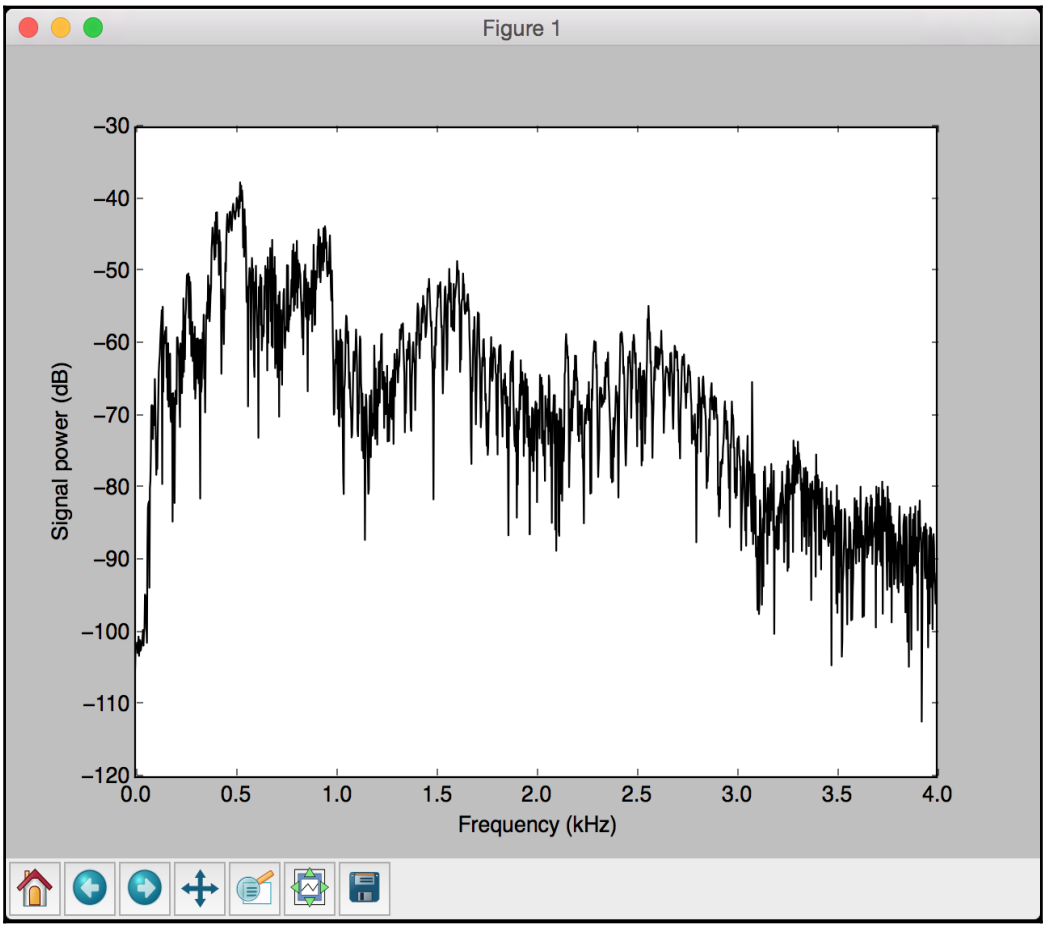
Build the *X* axis, which is frequency measured in `kHz` in this case:

```
# Build the X axis
x_axis = np.arange(0, len_half, 1) * (sampling_freq / len_signal) / 1000.0
```

Plot the figure:

```
# Plot the figure
plt.figure()
plt.plot(x_axis, signal_power, color='black')
plt.xlabel('Frequency (kHz)')
plt.ylabel('Signal power (dB)')
plt.show()
```

The full code is given in the file `frequency_transformer.py`. If you run the code, you will see the following screenshot:



The preceding screenshot shows the power of the signal across the frequency spectrum.

Generating audio signals

Now that we know how audio signals work, let's see how we can generate one such signal. We can use the `NumPy` package to generate various audio signals. Since audio signals are mixtures of **sinusoids,** we can use this to generate an audio signal with some predefined parameters.

Create a new Python file and import the following packages:

```
import numpy as np
import matplotlib.pyplot as plt
from scipy.io.wavfile import write
```

Define the output audio filename:

```
# Output file where the audio will be saved
output_file = 'generated_audio.wav'
```

Specify the audio parameters such as duration, sampling frequency, tone frequency, minimum value, and maximum value:

```
# Specify audio parameters
duration = 4  # in seconds
sampling_freq = 44100  # in Hz
tone_freq = 784
min_val = -4 * np.pi
max_val = 4 * np.pi
```

Generate the audio signal using the defined parameters:

```
# Generate the audio signal
t = np.linspace(min_val, max_val, duration * sampling_freq)
signal = np.sin(2 * np.pi * tone_freq * t)
```

Add some noise to the signal:

```
# Add some noise to the signal
noise = 0.5 * np.random.rand(duration * sampling_freq)
signal += noise
```

Normalize and scale the signal:

```
# Scale it to 16-bit integer values
scaling_factor = np.power(2, 15) - 1
signal_normalized = signal / np.max(np.abs(signal))
signal_scaled = np.int16(signal_normalized * scaling_factor)
```

Save the generated audio signal in the output file:

```
# Save the audio signal in the output file
write(output_file, sampling_freq, signal_scaled)
```

Extract the first `200` values for plotting:

```
# Extract the first 200 values from the audio signal
signal = signal[:200]
```

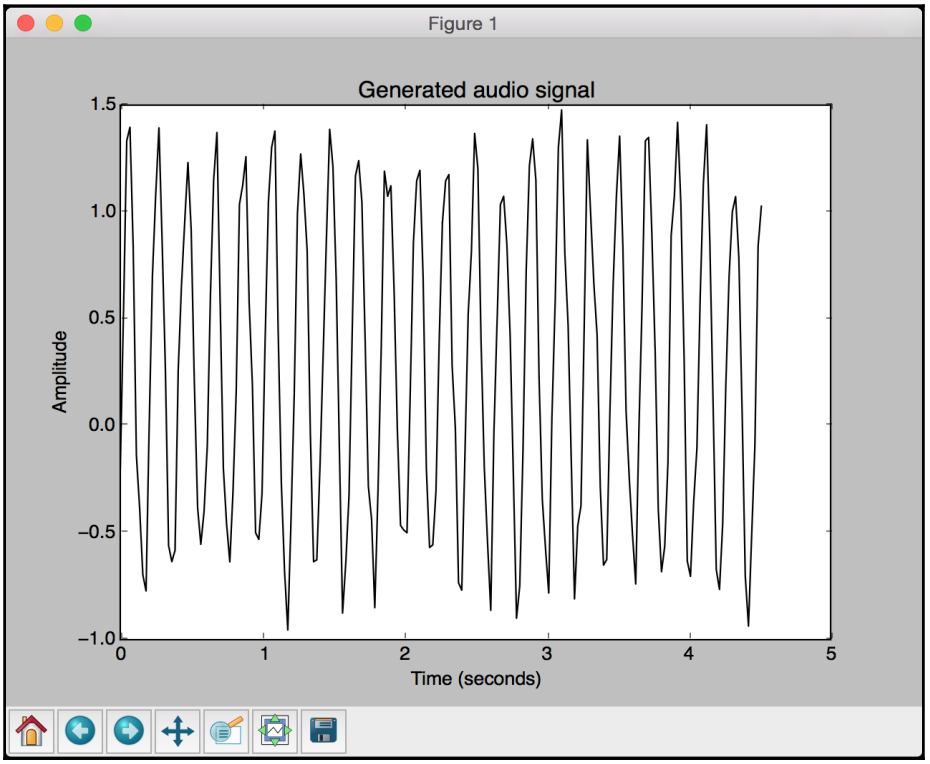
Construct the time axis in milliseconds:

```
# Construct the time axis in milliseconds
time_axis = 1000 * np.arange(0, len(signal), 1) / float(sampling_freq)
```

Plot the audio signal:

```
# Plot the audio signal
plt.plot(time_axis, signal, color='black')
plt.xlabel('Time (milliseconds)')
plt.ylabel('Amplitude')
plt.title('Generated audio signal')
plt.show()
```

The full code is given in the file `audio_generator.py`. If you run the code, you will see the following screenshot:



Play the file `generated_audio.wav` using your media player to see what it sounds like. It will be a signal that's a mixture of a *784 Hz* signal and the noise signal.

Synthesizing tones to generate music

The previous section described how to generate a simple monotone, but it's not all that meaningful. It was just a single frequency through the signal. Let's use that principle to synthesize music by stitching different tones together. We will be using standard tones like *A*, *C*, *G*, *F*, and so on to generate music. In order to see the frequency mapping for these standard tones, you can check out this link: `http://www.phy.mtu.edu/~suits/notefreqs.html`. Let's use this information to generate a musical signal.

Create a new Python file and import the following packages:

```
import json

import numpy as np
import matplotlib.pyplot as plt
from scipy.io.wavfile import write
```

Define a function to generate a tone based on the input parameters:

```
# Synthesize the tone based on the input parameters
def tone_synthesizer(freq, duration, amplitude=1.0, sampling_freq=44100):
    # Construct the time axis
    time_axis = np.linspace(0, duration, duration * sampling_freq)
```

Construct the audio signal using the parameters specified and return it:

```
    # Construct the audio signal
    signal = amplitude * np.sin(2 * np.pi * freq * time_axis)

    return signal.astype(np.int16)
```

Define the `main` function. Let's define the output audio filenames:

```
if __name__=='__main__':
    # Names of output files
    file_tone_single = 'generated_tone_single.wav'
    file_tone_sequence = 'generated_tone_sequence.wav'
```

We will be using a tone mapping file that contains the mapping from tone names (such as `A`, `C`, `G`, and so on) to the corresponding frequencies:

```
# Source: http://www.phy.mtu.edu/~suits/notefreqs.html
mapping_file = 'tone_mapping.json'
# Load the tone to frequency map from the mapping file
with open(mapping_file, 'r') as f:
    tone_map = json.loads(f.read())
```

Let's generate the `F` tone with a duration of 3 seconds:

```
# Set input parameters to generate 'F' tone
tone_name = 'F'
duration = 3     # seconds
amplitude = 12000
sampling_freq = 44100    # Hz
```

Extract the corresponding tone frequency:

```
# Extract the tone frequency
tone_freq = tone_map[tone_name]
```

Generate the tone using the tone synthesizer function that was defined earlier:

```
    # Generate the tone using the above parameters
    synthesized_tone = tone_synthesizer(tone_freq, duration, amplitude,
sampling_freq)
```

Write the generated audio signal to the output file:

```
# Write the audio signal to the output file
write(file_tone_single, sampling_freq, synthesized_tone)
```

Let's generate a tone sequence to make it sound like music. Let's define a tone sequence with corresponding durations in seconds:

```
    # Define the tone sequence along with corresponding durations in
seconds
    tone_sequence = [('G', 0.4), ('D', 0.5), ('F', 0.3), ('C', 0.6), ('A',
0.4)]
```

Construct the audio signal based on the tone sequence:

```
# Construct the audio signal based on the above sequence
signal = np.array([])
for item in tone_sequence:
    # Get the name of the tone
    tone_name = item[0]
```

For each tone, extract the corresponding frequency:

```
            # Extract the corresponding frequency of the tone
            freq = tone_map[tone_name]
```

Extract the corresponding duration:

```
            # Extract the duration
            duration = item[1]
```

Synthesize the tone using the tone `synthesizer` function:

```
            # Synthesize the tone
            synthesized_tone = tone_synthesizer(freq, duration, amplitude,
    sampling_freq)
```

Append it to the main output signal:

```
            # Append the output signal
            signal = np.append(signal, synthesized_tone, axis=0)
```

Save the main output signal to the output file:

```
        # Save the audio in the output file
        write(file_tone_sequence, sampling_freq, signal)
```

The full code is given in the file `synthesizer.py`. If you run the code, it will generate two output files — `generated_tone_single.wav` and `generated_tone_sequence.wav`. Play the audio files using a media player to hear what they sound like.

Extracting speech features

We learnt how to convert a time domain signal into the frequency domain. Frequency domain features are used extensively in all the speech recognition systems. The concept we discussed earlier is an introduction to the idea, but real world frequency domain features are a bit more complex. Once we convert a signal into the frequency domain, we need to ensure that it's usable in the form of a feature vector. This is where the concept of **Mel Frequency Cepstral Coefficients** (**MFCCs**) becomes relevant. MFCC is a tool that's used to extract frequency domain features from a given audio signal.

In order to extract the frequency features from an audio signal, MFCC first extracts the power spectrum. It then uses filter banks and a **discrete cosine transform** (**DCT**) to extract the features. If you are interested in exploring MFCC further, you can check out this link: `http://practicalcryptography.com/miscellaneous/machine-learning/guide-mel-frequency-cepstral-coefficients-mfccs`.

We will be using a package called `python_speech_features` to extract the MFCC features. The package is available here: `http://python-speech-features.readthedocs.org/en/latest`. For ease of use, the relevant folder has been included with the code bundle. You will see a folder called `features` in the code bundle that contains the relevant files needed to use this package. Let's see how to extract MFCC features.

Create a new Python file and import the following packages:

```
import numpy as np
import matplotlib.pyplot as plt
from scipy.io import wavfile
from features import mfcc, logfbank
```

Read the input audio file and extract the first `10,000` samples for analysis:

```
# Read the input audio file
sampling_freq, signal = wavfile.read('random_sound.wav')

# Take the first 10,000 samples for analysis
signal = signal[:10000]
```

Extract the MFCC:

```
# Extract the MFCC features
features_mfcc = mfcc(signal, sampling_freq)
```

Print the MFCC parameters:

```
# Print the parameters for MFCC
print('\nMFCC:\nNumber of windows =', features_mfcc.shape[0])
print('Length of each feature =', features_mfcc.shape[1])
```

Plot the MFCC features:

```
# Plot the features
features_mfcc = features_mfcc.T
plt.matshow(features_mfcc)
plt.title('MFCC')
```

Extract the filter bank features:

```
# Extract the Filter Bank features
features_fb = logfbank(signal, sampling_freq)
```

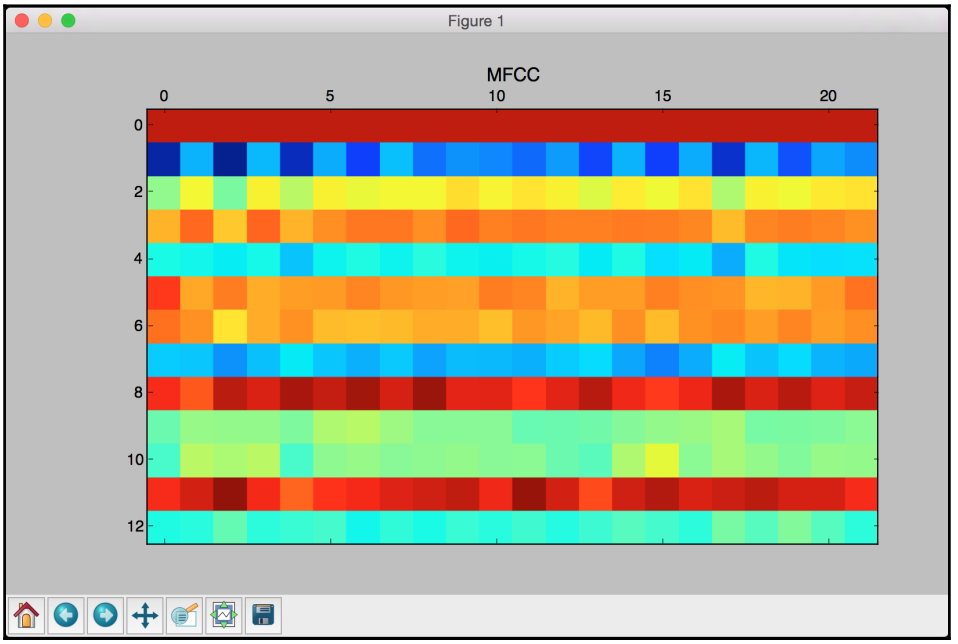
Print the parameters for the filter bank:

```
# Print the parameters for Filter Bank
print('\nFilter bank:\nNumber of windows =', features_fb.shape[0])
print('Length of each feature =', features_fb.shape[1])
```

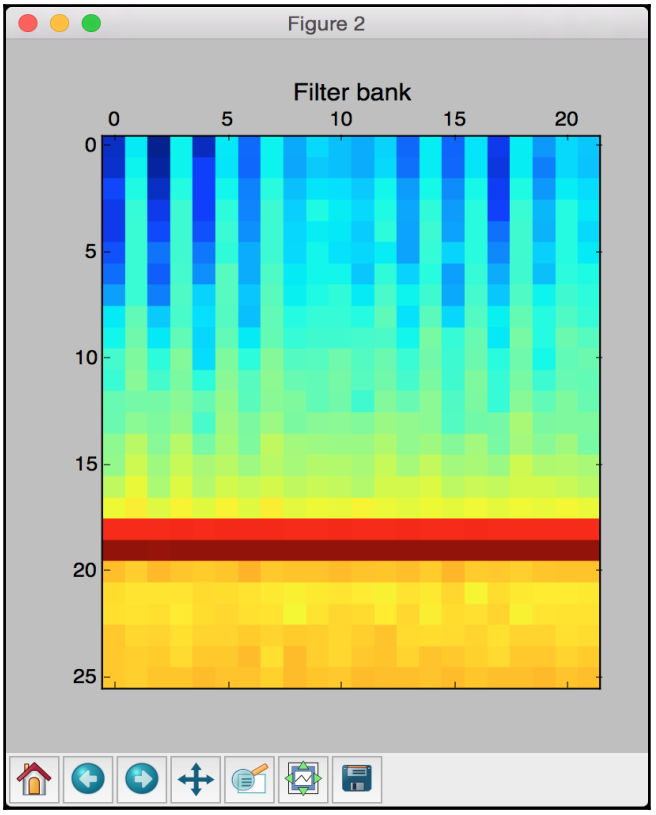
Plot the features:

```
# Plot the features
features_fb = features_fb.T
plt.matshow(features_fb)
plt.title('Filter bank')

plt.show()
```

The full code is given in the file `feature_extractor.py`. If you run the code, you will see two screenshots. The first screenshot shows the MFCC features:



The second screenshot shows the filter bank features:



You will see the following printed on your Terminal:

```
MFCC:
Number of windows = 22
Length of each feature = 13

Filter bank:
Number of windows = 22
Length of each feature = 26
```

Recognizing spoken words

Now that we have learnt all the techniques to analyze speech signals, let's go ahead and see how to recognize spoken words. Speech recognition systems take audio signals as input and recognize the words being spoken. We will use Hidden Markov Models (HMMs) for this task.

As we discussed in the previous chapter, HMMs are great at analyzing sequential data. An audio signal is a time series signal, which is a manifestation of sequential data. The assumption is that the outputs are being generated by the system going through a series of hidden states. Our goal is to find out what these hidden states are so that we can identify the words in our signal. If you are interesting in digging deeper, you can check out this link: `https://www.robots.ox.ac.uk/~vgg/rg/slides/hmm.pdf`.

We will be using a package called `hmmlearn` to build our speech recognition system. You can learn more about it here: `http://hmmlearn.readthedocs.org/en/latest`. You can install the package by running the following command on your Terminal:

```
$ pip3 install hmmlearn
```

In order to train our speech recognition system, we need a dataset of audio files for each word. We will use the database available at `https://code.google.com/archive/p/hmm-speech-recognition/downloads`. For ease of use, you have been provided with a folder called `data` in your code bundle that contains all these files. This dataset contains seven different words. Each word has a folder associated with it and each folder has 15 audio files. We will use 14 for training and one for testing in each folder. Note that this is actually a very small dataset. In the real world, you will be using much larger datasets to build speech recognition systems. We are using this dataset to get familiar with speech recognition and see how we can build a system to recognize spoken words.

We will go ahead and build an HMM model for each word. We will store all these models for reference. When we want to recognize the word in an unknown audio file, we will run it through all these models and pick the one with the highest score. Let's see how to build this system.

Create a new Python file and import the following packages:

```
import os
import argparse
import warnings

import numpy as np
from scipy.io import wavfile

from hmmlearn import hmm
from features import mfcc
```

Define a function to parse the input arguments. We need to specify the input folder containing the audio files required to train our speech recognition system:

```
# Define a function to parse the input arguments
def build_arg_parser():
    parser = argparse.ArgumentParser(description='Trains the HMM-based
            speech \ recognition system')
    parser.add_argument("--input-folder", dest="input_folder",
            required=True, help="Input folder containing the audio files
            for training")
    return parser
```

Define a class to train the HMMs:

```
# Define a class to train the HMM
class ModelHMM(object):
    def __init__(self, num_components=4, num_iter=1000):
        self.n_components = num_components
        self.n_iter = num_iter
```

Define the covariance type and the type of HMM:

```
        self.cov_type = 'diag'
        self.model_name = 'GaussianHMM'
```

Initialize the variable in which we will store the models for each word:

```
        self.models = []
```

Define the model using the specified parameters:

```
        self.model = hmm.GaussianHMM(n_components=self.n_components,
                covariance_type=self.cov_type, n_iter=self.n_iter)
```

Define a method to train the model:

```
    # 'training_data' is a 2D numpy array where each row is 13-dimensional
    def train(self, training_data):
        np.seterr(all='ignore')
        cur_model = self.model.fit(training_data)
        self.models.append(cur_model)
```

Define a method to compute the score for input data:

```
    # Run the HMM model for inference on input data
    def compute_score(self, input_data):
        return self.model.score(input_data)
```

Define a function to build a model for each word in the training dataset:

```
# Define a function to build a model for each word
def build_models(input_folder):
    # Initialize the variable to store all the models
    speech_models = []
```

Parse the input directory:

```
    # Parse the input directory
    for dirname in os.listdir(input_folder):
        # Get the name of the subfolder
        subfolder = os.path.join(input_folder, dirname)

        if not os.path.isdir(subfolder):
            continue
```

Extract the label:

```
        # Extract the label
        label = subfolder[subfolder.rfind('/') + 1:]
```

Initialize the variable to store the training data:

```
        # Initialize the variables
        X = np.array([])
```

Create a list of files to be used for training:

```
        # Create a list of files to be used for training
        # We will leave one file per folder for testing
        training_files = [x for x in os.listdir(subfolder) if
x.endswith('.wav')][:-1]

        # Iterate through the training files and build the models
```

```
        for filename in training_files:
            # Extract the current filepath
            filepath = os.path.join(subfolder, filename)
```

Read the audio signal from the current file:

```
            # Read the audio signal from the input file
            sampling_freq, signal = wavfile.read(filepath)
```

Extract the MFCC features:

```
            # Extract the MFCC features
            with warnings.catch_warnings():
                warnings.simplefilter('ignore')
                features_mfcc = mfcc(signal, sampling_freq)
```

Append the data point to the variable `X`:

```
            # Append to the variable X
            if len(X) == 0:
                X = features_mfcc
            else:
                X = np.append(X, features_mfcc, axis=0)
```

Initialize the HMM model:

```
        # Create the HMM model
        model = ModelHMM()
```

Train the model using the training data:

```
        # Train the HMM
        model.train(X)
```

Save the model for the current word:

```
        # Save the model for the current word
        speech_models.append((model, label))

        # Reset the variable
        model = None

    return speech_models
```

Define a function to run the tests on the test dataset:

```
# Define a function to run tests on input files
def run_tests(test_files):
    # Classify input data
    for test_file in test_files:
        # Read input file
        sampling_freq, signal = wavfile.read(test_file)
```

Extract the MFCC features:

```
        # Extract MFCC features
        with warnings.catch_warnings():
            warnings.simplefilter('ignore')
            features_mfcc = mfcc(signal, sampling_freq)
```

Define the variables to store the maximum score and the output label:

```
        # Define variables
        max_score = -float('inf')
        output_label = None
```

Iterate through each model to pick the best one:

```
        # Run the current feature vector through all the HMM
        # models and pick the one with the highest score
        for item in speech_models:
            model, label = item
```

Evaluate the score and compare against the maximum score:

```
            score = model.compute_score(features_mfcc)
            if score > max_score:
                max_score = score
                predicted_label = label
```

Print the output:

```
        # Print the predicted output
        start_index = test_file.find('/') + 1
        end_index = test_file.rfind('/')
        original_label = test_file[start_index:end_index]
        print('\nOriginal: ', original_label)
        print('Predicted:', predicted_label)
```

Define the `main` function and get the input folder from the input parameter:

```
if __name__=='__main__':
    args = build_arg_parser().parse_args()
    input_folder = args.input_folder
```

Build an HMM model for each word in the input folder:

```
    # Build an HMM model for each word
    speech_models = build_models(input_folder)
```

We left one file for testing in each folder. Use that file to see how accurate the model is:

```
    # Test files -- the 15th file in each subfolder
    test_files = []
    for root, dirs, files in os.walk(input_folder):
        for filename in (x for x in files if '15' in x):
            filepath = os.path.join(root, filename)
            test_files.append(filepath)

    run_tests(test_files)
```

The full code is given in the file `speech_recognizer.py`. Make sure that the `data` folder is placed in the same folder as the code file. Run the code as given below:

```
$ python3 speech_recognizer.py --input-folder data
```

If you run the code, you will see the following output:

```
Original:  apple
Predicted: apple

Original:  banana
Predicted: banana

Original:  kiwi
Predicted: kiwi

Original:  lime
Predicted: lime

Original:  orange
Predicted: orange

Original:  peach
Predicted: peach

Original:  pineapple
Predicted: pineapple
```

As we can see in the preceding screenshot, our speech recognition system identifies all the words correctly.

Summary

In this chapter, we learnt about speech recognition. We discussed how to work with speech signals and the associated concepts. We learnt how to visualize audio signals. We talked about how to transform time domain audio signals into the frequency domain using Fourier Transforms. We discussed how to generate audio signals using predefined parameters.

We then used this concept to synthesize music by stitching tones together. We talked about MFCCs and how they are used in the real world. We understood how to extract frequency features from speech. We learnt how to use all these techniques to build a speech recognition system. In the next chapter, we will learn about object detection and tracking. We will use those concepts to build an engine that can track objects in a live video.

13
Object Detection and Tracking

In this chapter, we are going to learn about object detection and tracking. We will start by installing OpenCV, a very popular library for computer vision. We will discuss frame differencing to see how we can detect the moving parts in a video. We will learn how to track objects using color spaces. We will understand how to use background subtraction to track objects. We will build an interactive object tracker using the `CAMShift` algorithm. We will learn how to build an optical flow based tracker. We will discuss face detection and associated concepts such as Haar cascades and integral images. We will then use this technique to build an eye detector and tracker.

By the end of this chapter, you will know about:

- Installing OpenCV
- Frame differencing
- Tracking objects using colorspaces
- Object tracking using background subtraction
- Building an interactive object tracker using the CAMShift algorithm
- Optical flow based tracking
- Face detection and tracking
- Using Haar cascades for object detection
- Using integral images for feature extraction
- Eye detection and tracking

Installing OpenCV

We will be using a package called **OpenCV** in this chapter. You can learn more about it here: `http://opencv.org`. Make sure to install it before you proceed. Here are the links to install OpenCV 3 with Python 3 on various operating systems:

- **Windows:**
  `https://solarianprogrammer.com/2016/09/17/install-opencv-3-with-python-3-on-windows`
- **Ubuntu:**
  `http://www.pyimagesearch.com/2015/07/20/install-opencv-3-0-and-python-3-4-on-ubuntu`
- **Mac:**
  `http://www.pyimagesearch.com/2015/06/29/install-opencv-3-0-and-python-3-4-on-osx`

Now that you have installed it, let's go to the next section.

Frame differencing

Frame differencing is one of the simplest techniques that can be used to identify the moving parts in a video. When we are looking at a live video stream, the differences between consecutive frames captured from the stream gives us a lot of information. Let's see how we can take the differences between consecutive frames and display the differences. The code in this section requires an attached camera, so make sure you have a camera on your machine.

Create a new Python file and import the following package:

```
import cv2
```

Define a function to compute the frame differences. Start by computing the difference between the current frame and the next frame:

```
# Compute the frame differences
def frame_diff(prev_frame, cur_frame, next_frame):
    # Difference between the current frame and the next frame
    diff_frames_1 = cv2.absdiff(next_frame, cur_frame)
```

Compute the difference between the current frame and the previous frame:

```
    # Difference between the current frame and the previous frame
    diff_frames_2 = cv2.absdiff(cur_frame, prev_frame)
```

Compute the bitwise-AND between the two difference frames and return it:

```
    return cv2.bitwise_and(diff_frames_1, diff_frames_2)
```

Define a function to grab the current frame from the webcam. Start by reading it from the video capture object:

```
# Define a function to get the current frame from the webcam
def get_frame(cap, scaling_factor):
    # Read the current frame from the video capture object
    _, frame = cap.read()
```

Resize the frame based on the scaling factor and return it:

```
    # Resize the image
    frame = cv2.resize(frame, None, fx=scaling_factor,
            fy=scaling_factor, interpolation=cv2.INTER_AREA)
```

Convert the image to grayscale and return it:

```
    # Convert to grayscale
    gray = cv2.cvtColor(frame, cv2.COLOR_RGB2GRAY)

    return gray
```

Define the main function and initialize the video capture object:

```
if __name__=='__main__':
    # Define the video capture object
    cap = cv2.VideoCapture(0)
```

Define the scaling factor to resize the images:

```
    # Define the scaling factor for the images
    scaling_factor = 0.5
```

Grab the current frame, the next frame, and the frame after that:

```
# Grab the current frame
prev_frame = get_frame(cap, scaling_factor)
# Grab the next frame
cur_frame = get_frame(cap, scaling_factor)
# Grab the frame after that
next_frame = get_frame(cap, scaling_factor)
```

Iterate indefinitely until the user presses the *Esc* key. Start by computing the frame differences:

```
# Keep reading the frames from the webcam
# until the user hits the 'Esc' key
while True:
    # Display the frame difference
    cv2.imshow('Object Movement', frame_diff(prev_frame,
            cur_frame, next_frame))
```

Update the frame variables:

```
    # Update the variables
    prev_frame = cur_frame
    cur_frame = next_frame
```

Grab the next frame from the webcam:

```
    # Grab the next frame
    next_frame = get_frame(cap, scaling_factor)
```

Check if the user pressed the *Esc* key. If so, exit the loop:

```
    # Check if the user hit the 'Esc' key
    key = cv2.waitKey(10)
    if key == 27:
        break
```

Once you exit the loop, make sure that all the windows are closed properly:

```
# Close all the windows
cv2.destroyAllWindows()
```

The full code is given in the file `frame_diff.py` provided to you. If you run the code, you will see an output window showing a live output. If you move around, you will see your silhouette as shown here:

The white lines in the preceding screenshot represent the silhouette.

Tracking objects using colorspaces

The information obtained by frame differencing is useful, but we will not be able to build a robust tracker with it. It is very sensitive to noise and it does not really track an object completely. To build a robust object tracker, we need to know what characteristics of the object can be used to track it accurately. This is where color spaces become relevant.

An image can be represented using various color spaces. The RGB color space is probably the most popular color space, but it does not lend itself nicely to applications like object tracking. So we will be using the HSV color space instead. It is an intuitive color space model that is closer to how humans perceive color. You can learn more about it here: `http://infohost.nmt.edu/tcc/help/pubs/colortheory/web/hsv.html`. We can convert the captured frame from RGB to HSV colorspace, and then use color thresholding to track any given object. We should note that we need to know the color distribution of the object so that we can select the appropriate ranges for thresholding.

Create a new Python file and import the following packages:

```
import cv2
import numpy as np
```

Define a function to grab the current frame from the webcam. Start by reading it from the video capture object:

```
# Define a function to get the current frame from the webcam
def get_frame(cap, scaling_factor):
    # Read the current frame from the video capture object
    _, frame = cap.read()
```

Resize the frame based on the scaling factor and return it:

```
    # Resize the image
    frame = cv2.resize(frame, None, fx=scaling_factor,
            fy=scaling_factor, interpolation=cv2.INTER_AREA)

    return frame
```

Define the main function. Start by initializing the video capture object:

```
if __name__=='__main__':
    # Define the video capture object
    cap = cv2.VideoCapture(0)
```

Define the scaling factor to be used to resize the captured frames:

```
    # Define the scaling factor for the images
    scaling_factor = 0.5
```

Iterate indefinitely until the user hits the *Esc* key. Grab the current frame to start:

```
# Keep reading the frames from the webcam
# until the user hits the 'Esc' key
while True:
    # Grab the current frame
    frame = get_frame(cap, scaling_factor)
```

Convert the image to HSV color space using the inbuilt function available in OpenCV:

```
    # Convert the image to HSV colorspace
    hsv = cv2.cvtColor(frame, cv2.COLOR_BGR2HSV)
```

Define the approximate HSV color range for the color of human skin:

```
    # Define range of skin color in HSV
    lower = np.array([0, 70, 60])
    upper = np.array([50, 150, 255])
```

Threshold the HSV image to create the mask:

```
    # Threshold the HSV image to get only skin color
    mask = cv2.inRange(hsv, lower, upper)
```

Compute bitwise-AND between the mask and the original image:

```
    # Bitwise-AND between the mask and original image
    img_bitwise_and = cv2.bitwise_and(frame, frame, mask=mask)
```

Run median blurring to smoothen the image:

```
    # Run median blurring
    img_median_blurred = cv2.medianBlur(img_bitwise_and, 5)
```

Display the input and output frames:

```
    # Display the input and output
    cv2.imshow('Input', frame)
    cv2.imshow('Output', img_median_blurred)
```

Check if the user pressed the *Esc* key. If so, then exit the loop:

```
# Check if the user hit the 'Esc' key
c = cv2.waitKey(5)
if c == 27:
    break
```

Once you exit the loop, make sure that all the windows are properly closed:

```
# Close all the windows
cv2.destroyAllWindows()
```

The full code is given in the file `colorspaces.py` provided to you. If you run the code, you will get two screenshot. The window titled **Input** is the captured frame:

The second window titled **Output** shows the skin mask:

Object tracking using background subtraction

Background subtraction is a technique that models the background in a given video, and then uses that model to detect moving objects. This technique is used a lot in video compression as well as video surveillance. It performs really well where we have to detect moving objects within a static scene. The algorithm basically works by detecting the background, building a model for it, and then subtracting it from the current frame to obtain the foreground. This foreground corresponds to moving objects.

One of the main steps here it to build a model of the background. It is not the same as frame differencing because we are not differencing successive frames. We are actually modeling the background and updating it in real time, which makes it an adaptive algorithm that can adjust to a moving baseline. This is why it performs much better than frame differencing.

Create a new Python file and import the following packages:

```
import cv2
import numpy as np
```

Define a function to grab the current frame:

```
# Define a function to get the current frame from the webcam
def get_frame(cap, scaling_factor):
    # Read the current frame from the video capture object
    _, frame = cap.read()
```

Resize the frame and return it:

```
    # Resize the image
    frame = cv2.resize(frame, None, fx=scaling_factor,
            fy=scaling_factor, interpolation=cv2.INTER_AREA)

    return frame
```

Define the main function and initialize the video capture object:

```
if __name__=='__main__':
    # Define the video capture object
    cap = cv2.VideoCapture(0)
```

Define the background subtractor object:

```
    # Define the background subtractor object
    bg_subtractor = cv2.createBackgroundSubtractorMOG2()
```

Define the history and the learning rate. The comment below is pretty self explanatory as to what "history" is all about:

```
    # Define the number of previous frames to use to learn.
    # This factor controls the learning rate of the algorithm.
    # The learning rate refers to the rate at which your model
    # will learn about the background. Higher value for
    # 'history' indicates a slower learning rate. You can
    # play with this parameter to see how it affects the output.
    history = 100

    # Define the learning rate
    learning_rate = 1.0/history
```

Iterate indefinitely until the user presses the *Esc* key. Start by grabbing the current frame:

```
# Keep reading the frames from the webcam
# until the user hits the 'Esc' key
while True:
    # Grab the current frame
    frame = get_frame(cap, 0.5)
```

Compute the mask using the background subtractor object defined earlier:

```
    # Compute the mask
    mask = bg_subtractor.apply(frame, learningRate=learning_rate)
```

Convert the mask from grayscale to RGB:

```
    # Convert grayscale image to RGB color image
    mask = cv2.cvtColor(mask, cv2.COLOR_GRAY2BGR)
```

Display the input and output images:

```
    # Display the images
    cv2.imshow('Input', frame)
    cv2.imshow('Output', mask & frame)
```

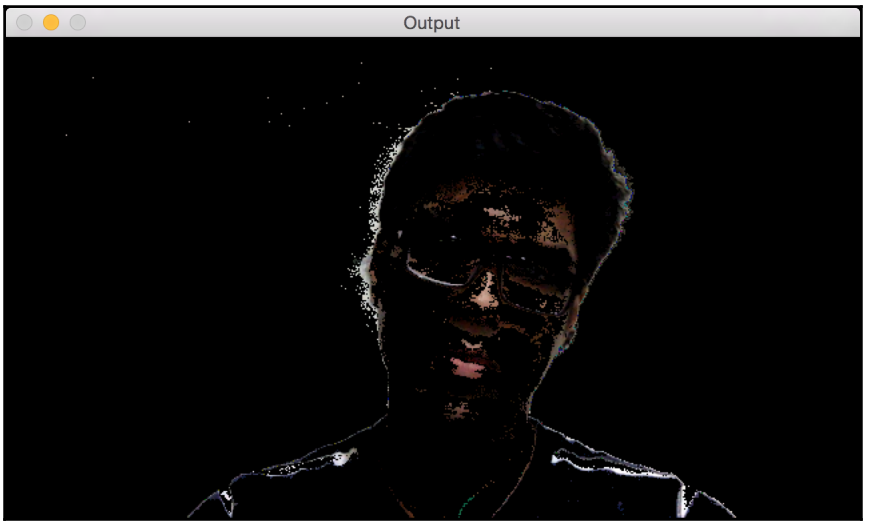
Check if the user pressed the *Esc* key. If so, exit the loop:

```
    # Check if the user hit the 'Esc' key
    c = cv2.waitKey(10)
    if c == 27:
        break
```

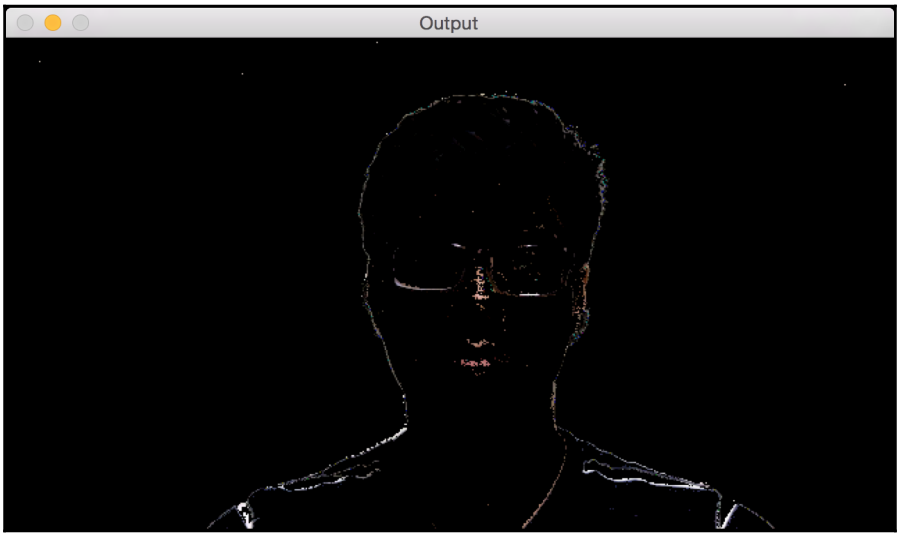
Once you exit the loop, make sure you release the video capture object and close all the windows properly:

```
# Release the video capture object
cap.release()
# Close all the windows
cv2.destroyAllWindows()
```

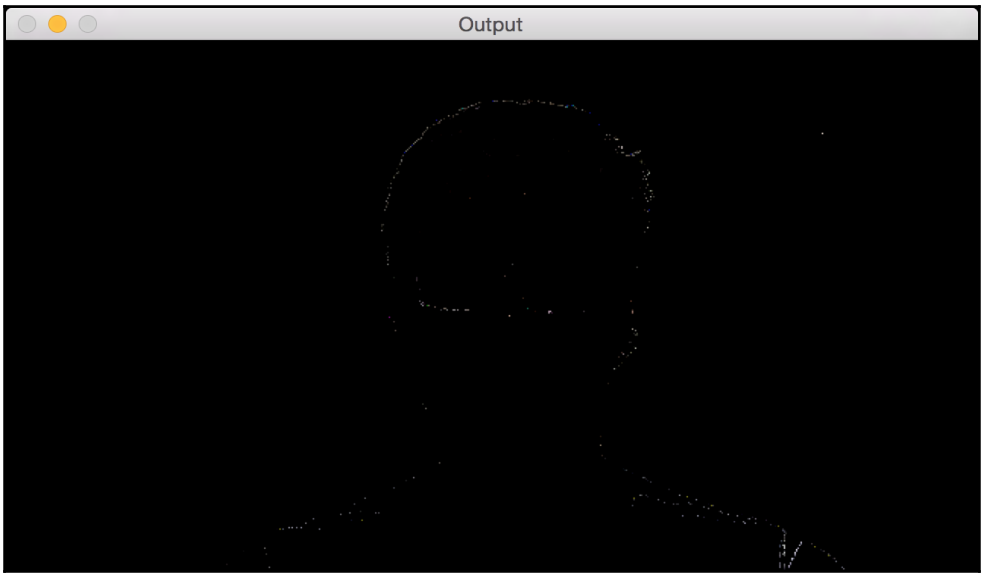
The full code is given in the file `background_subtraction.py` provided to you. If you run the code, you will see a window displaying the live output. If you move around, you will partially see yourself as shown here:



Once you stop moving around, it will start fading because you are now part of the background. The algorithm will consider you a part of the background and start updating the model accordingly:



As you remain still, it will continue to fade as shown here:



The process of fading indicates that the current scene is becoming part of the background model.

Building an interactive object tracker using the CAMShift algorithm

Color space based tracking allows us to track colored objects, but we have to define the color first. This seems restrictive! Let us see how we can select an object in a live video and then have a tracker that can track it. This is where the `CAMShift` algorithm, which stands for Continuously Adaptive Mean Shift, becomes relevant. This is basically an adaptive version of the Mean Shift algorithm.

In order to understand **CAMShift,** let's see how Mean Shift works. Consider a region of interest in a given frame. We have selected this region because it contains the object of interest. We want to track this object, so we have drawn a rough boundary around it, which is what “region of interest” refers to. We want our object tracker to track this object as it moves around in the video.

To do this, we select a set of points based on the color histogram of that region and then compute the centroid. If the location of this centroid is at the geometric center of this region, then we know that the object hasn't moved. But if the location of the centroid is not at the geometric center of this region, then we know that the object has moved. This means that we need to move the enclosing boundary as well. The movement of the centroid is directly indicative of the direction of movement of the object. We need to move our bounding box so that the new centroid becomes the geometric center of this bounding box. We keep doing this for every frame, and track the object in real time. Hence, this algorithm is called Mean Shift because the mean (i.e. the centroid) keeps shifting and we track the object using this.

Let us see how this is related to `CAMShift`. One of the problems with Mean Shift is that the size of the object is not allowed to change over time. Once we draw a bounding box, it will stay constant regardless of how close or far away the object is from the camera. This is why we need to use CAMShift because it can adapt the size of the bounding box to the size of the object. If you want to explore it further, you can check out this link: `http://docs.opencv.org/3.1.0/db/df8/tutorial_py_meanshift.html`. Let us see how to build a tracker.

Create a new python file and import the following packages:

```
import cv2
import numpy as np
```

Define a class to handle all the functionality related to object tracking:

```
# Define a class to handle object tracking related functionality
class ObjectTracker(object):
    def __init__(self, scaling_factor=0.5):
        # Initialize the video capture object
        self.cap = cv2.VideoCapture(0)
```

Capture the current frame:

```
        # Capture the frame from the webcam
        _, self.frame = self.cap.read()
```

Set the scaling factor:

```
        # Scaling factor for the captured frame
        self.scaling_factor = scaling_factor
```

Resize the frame:

```
# Resize the frame
self.frame = cv2.resize(self.frame, None,
        fx=self.scaling_factor, fy=self.scaling_factor,
        interpolation=cv2.INTER_AREA)
```

Create a window to display the output:

```
# Create a window to display the frame
cv2.namedWindow('Object Tracker')
```

Set the mouse callback function to take input from the mouse:

```
# Set the mouse callback function to track the mouse
cv2.setMouseCallback('Object Tracker', self.mouse_event)
```

Initialize variables to track the rectangular selection:

```
# Initialize variable related to rectangular region selection
self.selection = None
# Initialize variable related to starting position
self.drag_start = None
# Initialize variable related to the state of tracking
self.tracking_state = 0
```

Define a function to track the mouse events:

```
# Define a method to track the mouse events
def mouse_event(self, event, x, y, flags, param):
    # Convert x and y coordinates into 16-bit numpy integers
    x, y = np.int16([x, y])
```

When the left button on the mouse is down, it indicates that the user has started drawing a rectangle:

```
# Check if a mouse button down event has occurred
if event == cv2.EVENT_LBUTTONDOWN:
    self.drag_start = (x, y)
    self.tracking_state = 0
```

If the user is currently dragging the mouse to set the size of the rectangular selection, track the width and height:

```
# Check if the user has started selecting the region
if self.drag_start:
    if flags & cv2.EVENT_FLAG_LBUTTON:
        # Extract the dimensions of the frame
        h, w = self.frame.shape[:2]
```

Set the starting X and Y coordinates of the rectangle:

```
# Get the initial position
xi, yi = self.drag_start
```

Get the maximum and minimum values of the coordinates to make it agnostic to the direction in which you drag the mouse to draw the rectangle:

```
# Get the max and min values
x0, y0 = np.maximum(0, np.minimum([xi, yi], [x, y]))
x1, y1 = np.minimum([w, h], np.maximum([xi, yi], [x, y]))
```

Reset the selection variable:

```
# Reset the selection variable
self.selection = None
```

Finalize the rectangular selection:

```
# Finalize the rectangular selection
if x1-x0 > 0 and y1-y0 > 0:
    self.selection = (x0, y0, x1, y1)
```

If the selection is done, set the flag that indicates that we should start tracking the object within the rectangular region:

```
else:
    # If the selection is done, start tracking
    self.drag_start = None
    if self.selection is not None:
        self.tracking_state = 1
```

Define a method to track the object:

```
# Method to start tracking the object
def start_tracking(self):
    # Iterate until the user presses the Esc key
    while True:
        # Capture the frame from webcam
        _, self.frame = self.cap.read()
```

Resize the frame:

```
# Resize the input frame
self.frame = cv2.resize(self.frame, None,
        fx=self.scaling_factor, fy=self.scaling_factor,
        interpolation=cv2.INTER_AREA)
```

Create a copy of the frame. We will need it later:

```
# Create a copy of the frame
vis = self.frame.copy()
```

Convert the color space of the frame from RGB to HSV:

```
# Convert the frame to HSV colorspace
hsv = cv2.cvtColor(self.frame, cv2.COLOR_BGR2HSV)
```

Create the mask based on predefined thresholds:

```
# Create the mask based on predefined thresholds
mask = cv2.inRange(hsv, np.array((0., 60., 32.)),
            np.array((180., 255., 255.)))
```

Check if the user has selected the region:

```
# Check if the user has selected the region
if self.selection:
    # Extract the coordinates of the selected rectangle
    x0, y0, x1, y1 = self.selection

    # Extract the tracking window
    self.track_window = (x0, y0, x1-x0, y1-y0)
```

Extract the regions of interest from the HSV image as well as the mask. Compute the histogram of the region of interest based on these:

```
    # Extract the regions of interest
    hsv_roi = hsv[y0:y1, x0:x1]
    mask_roi = mask[y0:y1, x0:x1]


    # Compute the histogram of the region of
    # interest in the HSV image using the mask
    hist = cv2.calcHist( [hsv_roi], [0], mask_roi,
            [16], [0, 180] )
```

Normalize the histogram:

```
# Normalize and reshape the histogram
cv2.normalize(hist, hist, 0, 255, cv2.NORM_MINMAX);
self.hist = hist.reshape(-1)
```

Extract the region of interest from the original frame:

```
# Extract the region of interest from the frame
vis_roi = vis[y0:y1, x0:x1]
```

Compute bitwise-NOT of the region of interest. This is for display purposes only:

```
# Compute the image negative (for display only)
cv2.bitwise_not(vis_roi, vis_roi)
vis[mask == 0] = 0
```

Check if the system is in the tracking mode:

```
# Check if the system in the "tracking" mode
if self.tracking_state == 1:
    # Reset the selection variable
    self.selection = None
```

Compute the histogram backprojection:

```
# Compute the histogram back projection
hsv_backproj = cv2.calcBackProject([hsv], [0],
        self.hist, [0, 180], 1)
```

Compute bitwise-AND between the histogram and the mask:

```
# Compute bitwise AND between histogram
# backprojection and the mask
hsv_backproj &= mask
```

Define termination criteria for the tracker:

```
# Define termination criteria for the tracker
term_crit = (cv2.TERM_CRITERIA_EPS |
        cv2.TERM_CRITERIA_COUNT, 10, 1)
```

Apply the `CAMShift` algorithm to the backprojected histogram:

```
# Apply CAMShift on 'hsv_backproj'
track_box, self.track_window = cv2.CamShift(hsv_backproj,
        self.track_window, term_crit)
```

Draw an ellipse around the object and display it:

```
# Draw an ellipse around the object
cv2.ellipse(vis, track_box, (0, 255, 0), 2)
# Show the output live video
cv2.imshow('Object Tracker', vis)
```

If the user presses *Esc*, then exit the loop:

```
# Stop if the user hits the 'Esc' key
c = cv2.waitKey(5)
if c == 27:
    break
```

Once you exit the loop, make sure that all the windows are closed properly:

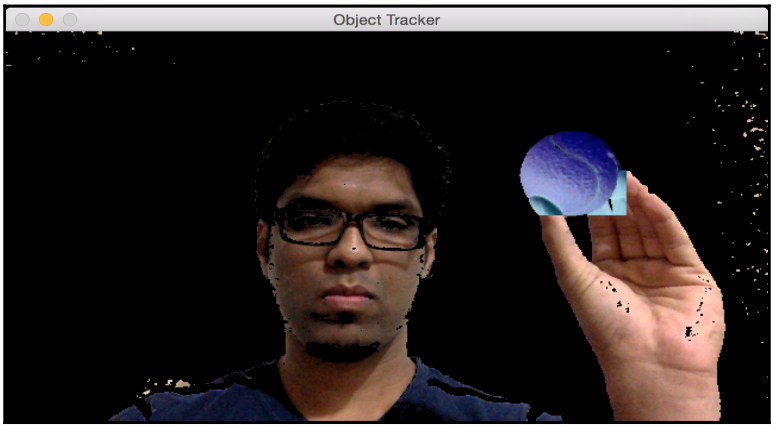
```
# Close all the windows
cv2.destroyAllWindows()
```

Define the main function and start tracking:

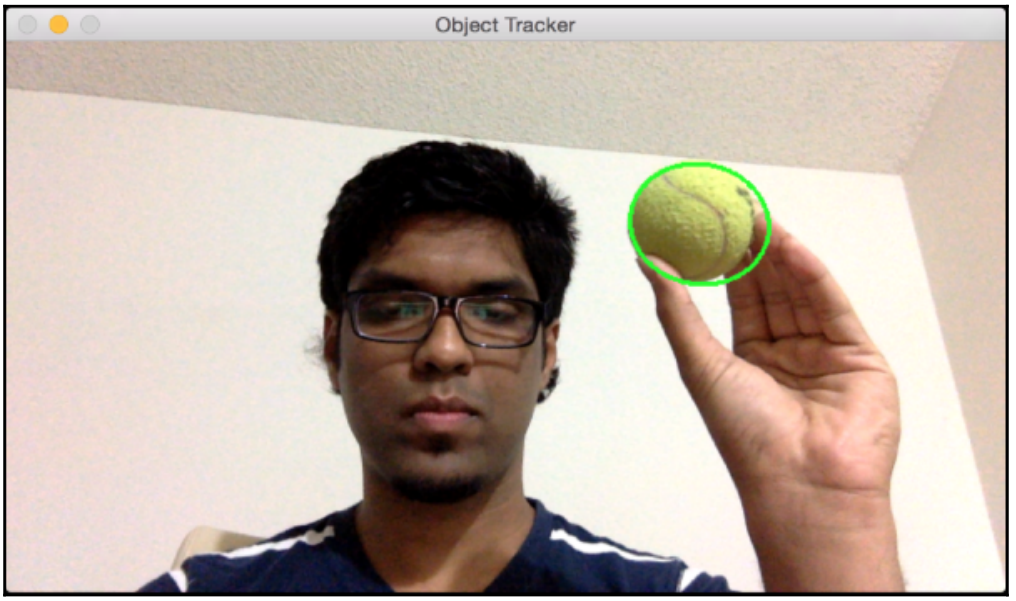
```
if __name__ == '__main__':
    # Start the tracker
    ObjectTracker().start_tracking()
```

The full code is given in the file `camshift.py` provided to you. If you run the code, you will see a window showing the live video from the webcam.

Take an object, hold it in your hand, and then draw a rectangle around it. Once you draw the rectangle, make sure to move the mouse pointer away from the final position. The image will look something like this:



Once the selection is done, move the mouse pointer to a different position to lock the rectangle. This event will start the tracking process as seen in the following image:



Let's move the object around to see if it's still being tracked:

Looks like it's working well. You can move the object around to see how it's getting tracked in real time.

Optical flow based tracking

Optical flow is a very popular technique used in computer vision. It uses image feature points to track an object. Individual feature points are tracked across successive frames in the live video. When we detect a set of feature points in a given frame, we compute the displacement vectors to keep track of it. We show the motion of these feature points between successive frames. These vectors are known as motion vectors. There are many different ways to perform optical flow, but the **Lucas-Kanade** method is perhaps the most popular. Here is the original paper that describes this technique: `http://cseweb.ucsd.edu/classes/sp02/cse252/lucaskanade81.pdf`.

The first step is to extract the feature points from the current frame. For each feature point that is extracted, a 3×3 patch (of pixels) is created with the feature point at the center. We are assuming that all the points in each patch have a similar motion. The size of this window can be adjusted depending on the situation.

For each patch, we look for a match in its neighborhood in the previous frame. We pick the best match based on an error metric. The search area is bigger than 3×3 because we look for a bunch of different 3×3 patches to get the one that is closest to the current patch. Once we get that, the path from the center point of the current patch and the matched patch in the previous frame will become the motion vector. We similarly compute the motion vectors for all the other patches.

Create a new python file and import the following packages:

```
import cv2
import numpy as np
```

Define a function to start tracking using optical flow. Start by initializing the video capture object and the scaling factor:

```
# Define a function to track the object
def start_tracking():
    # Initialize the video capture object
    cap = cv2.VideoCapture(0)

    # Define the scaling factor for the frames
    scaling_factor = 0.5
```

Define the number of frames to track and the number of frames to skip:

```
# Number of frames to track
num_frames_to_track = 5

# Skipping factor
num_frames_jump = 2
```

Initialize variables related to tracking paths and frame index:

```
# Initialize variables
tracking_paths = []
frame_index = 0
```

Define the tracking parameters like the window size, maximum level, and the termination criteria:

```
# Define tracking parameters
tracking_params = dict(winSize  = (11, 11), maxLevel = 2,
        criteria = (cv2.TERM_CRITERIA_EPS | cv2.TERM_CRITERIA_COUNT,
            10, 0.03))
```

Iterate indefinitely until the user presses the *Esc* key. Start by capturing the current frame and resizing it:

```
# Iterate until the user hits the 'Esc' key
while True:
    # Capture the current frame
    _, frame = cap.read()

    # Resize the frame
    frame = cv2.resize(frame, None, fx=scaling_factor,
            fy=scaling_factor, interpolation=cv2.INTER_AREA)
```

Convert the frame from RGB to grayscale:

```
    # Convert to grayscale
    frame_gray = cv2.cvtColor(frame, cv2.COLOR_BGR2GRAY)
```

Create a copy of the frame:

```
    # Create a copy of the frame
    output_img = frame.copy()
```

Check if the length of tracking paths is greater than zero:

```
if len(tracking_paths) > 0:
    # Get images
    prev_img, current_img = prev_gray, frame_gray
```

Organize the feature points:

```
    # Organize the feature points
    feature_points_0 = np.float32([tp[-1] for tp in \
            tracking_paths]).reshape(-1, 1, 2)
```

Compute the optical flow based on the previous and current images by using the feature points and the tracking parameters:

```
    # Compute optical flow
    feature_points_1, _, _ = cv2.calcOpticalFlowPyrLK(
            prev_img, current_img, feature_points_0,
            None, **tracking_params)
    # Compute reverse optical flow
    feature_points_0_rev, _, _ = cv2.calcOpticalFlowPyrLK(
            current_img, prev_img, feature_points_1,
            None, **tracking_params)

    # Compute the difference between forward and
    # reverse optical flow
    diff_feature_points = abs(feature_points_0 - \
            feature_points_0_rev).reshape(-1, 2).max(-1)
```

Extract the good feature points:

```
    # Extract the good points
    good_points = diff_feature_points < 1
```

Initialize the variable for the new tracking paths:

```
    # Initialize variable
    new_tracking_paths = []
```

Iterate through all the good feature points and draw circles around them:

```
        # Iterate through all the good feature points
        for tp, (x, y), good_points_flag in zip(tracking_paths,
                    feature_points_1.reshape(-1, 2), good_points):
            # If the flag is not true, then continue
            if not good_points_flag:
                continue
```

Append the X and Y coordinates and don't exceed the number of frames we are supposed to track:

```
            # Append the X and Y coordinates and check if
            # its length greater than the threshold
            tp.append((x, y))
            if len(tp) > num_frames_to_track:
                del tp[0]

            new_tracking_paths.append(tp)
```

Draw a circle around the point. Update the tracking paths and draw lines using the new tracking paths to show movement:

```
            # Draw a circle around the feature points
            cv2.circle(output_img, (x, y), 3, (0, 255, 0), -1)

        # Update the tracking paths
        tracking_paths = new_tracking_paths

        # Draw lines
        cv2.polylines(output_img, [np.int32(tp) for tp in \
                tracking_paths], False, (0, 150, 0))
```

Go into this `if` condition after skipping the number of frames specified earlier:

```
    # Go into this 'if' condition after skipping the
    # right number of frames
    if not frame_index % num_frames_jump:
        # Create a mask and draw the circles
        mask = np.zeros_like(frame_gray)
        mask[:] = 255
        for x, y in [np.int32(tp[-1]) for tp in tracking_paths]:
            cv2.circle(mask, (x, y), 6, 0, -1)
```

Compute the good features to track using the inbuilt function along with parameters like mask, maximum corners, quality level, minimum distance, and the block size:

```
# Compute good features to track
feature_points = cv2.goodFeaturesToTrack(frame_gray,
        mask = mask, maxCorners = 500, qualityLevel = 0.3,
        minDistance = 7, blockSize = 7)
```

If the feature points exist, append them to the tracking paths:

```
# Check if feature points exist. If so, append them
# to the tracking paths
if feature_points is not None:
    for x, y in np.float32(feature_points).reshape(-1, 2):
        tracking_paths.append([(x, y)])
```

Update the variables related to frame index and the previous grayscale image:

```
# Update variables
frame_index += 1
prev_gray = frame_gray
```

Display the output:

```
# Display output
cv2.imshow('Optical Flow', output_img)
```

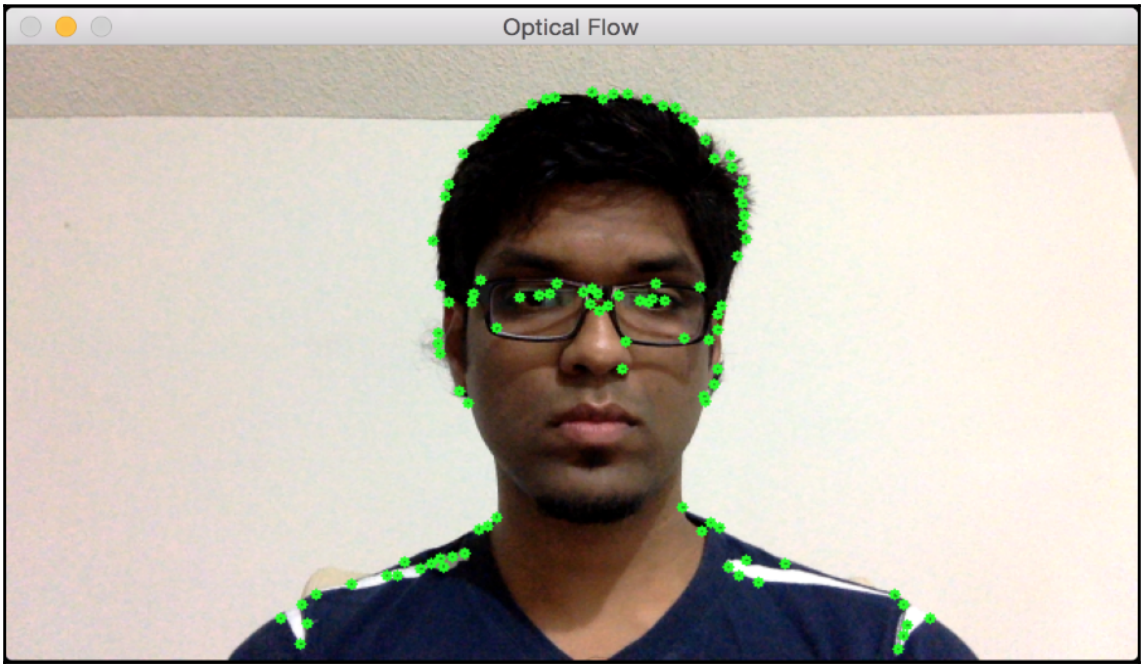
Check if the user pressed the *Esc* key. If so, exit the loop:

```
# Check if the user hit the 'Esc' key
c = cv2.waitKey(1)
if c == 27:
    break
```

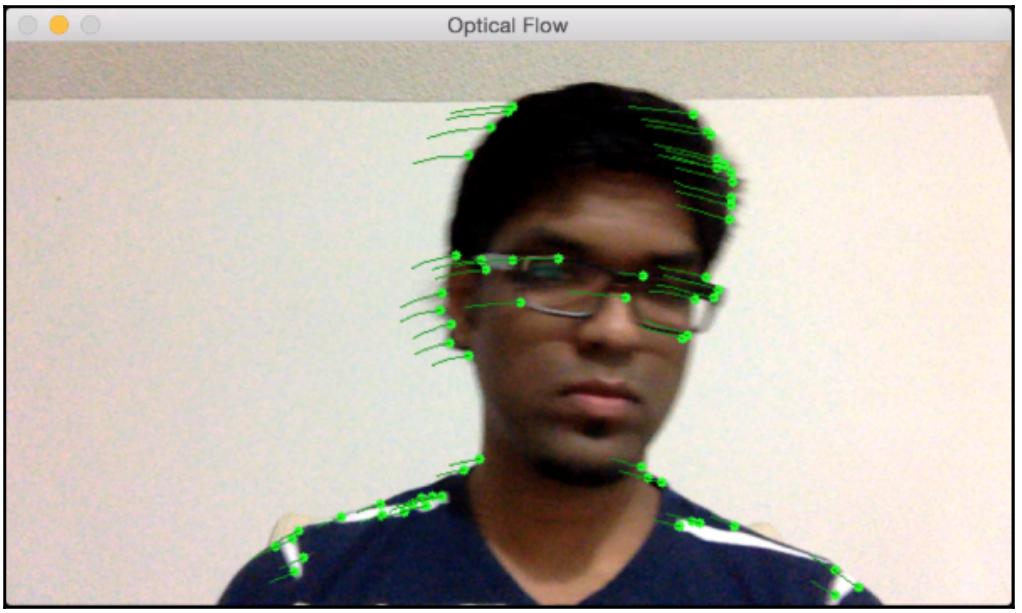
Define the main function and start tracking. Once you stop the tracker, make sure that all the windows are closed properly:

```
if __name__ == '__main__':
    # Start the tracker
    start_tracking()

    # Close all the windows
    cv2.destroyAllWindows()
```

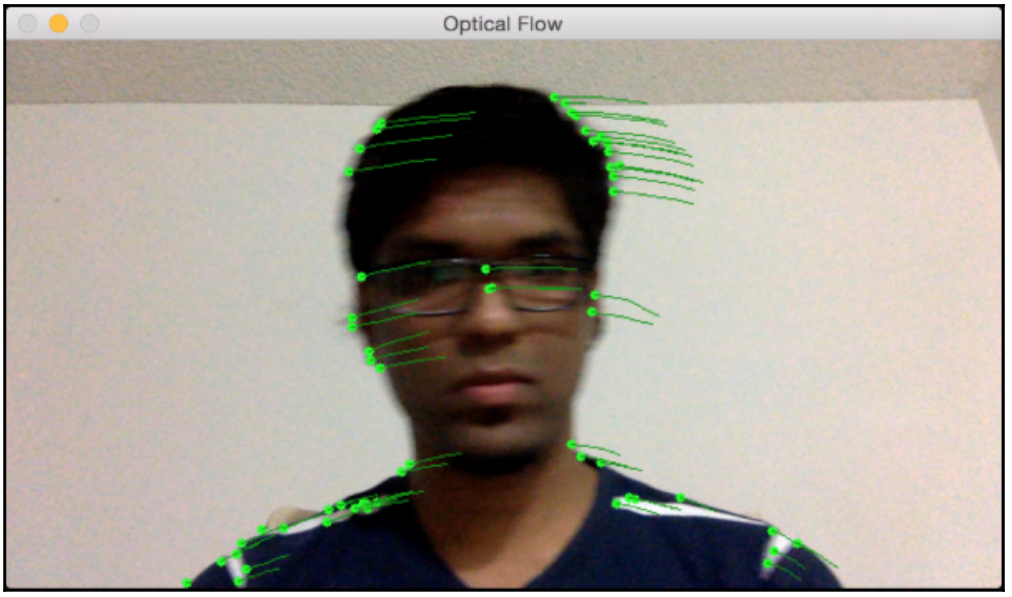
The full code is given in the file `optical_flow.py` provided to you. If you run the code, you will see a window showing the live video. You will see feature points as shown in the following screensot:



If you move around, you will see lines showing the movement of those feature points:



If you then move in the opposite direction, the lines will also change their direction accordingly:



Face detection and tracking

Face detection refers to detecting the location of a face in a given image. This is often confused with face recognition, which is the process of identifying who the person is. A typical biometric system utilizes both face detection and face recognition to perform the task. It uses face detection to locate the face and then uses face recognition to identify the person. In this section, we will see how to automatically detect the location of a face in a live video and track it.

Using Haar cascades for object detection

We will be using Haar cascades to detect faces in the video. Haar cascades, in this case, refer to cascade classifiers based on Haar features. *Paul Viola* and *Michael Jones* first came up with this object detection method in their landmark research paper in 2001. You can check it out here: `https://www.cs.cmu.edu/~efros/courses/LBMV07/Papers/viola-cvpr-01.pdf`. In their paper, they describe an effective machine learning technique that can be used to detect any object.

They use a boosted cascade of simple classifiers. This cascade is used to build an overall classifier that performs with high accuracy. The reason this is relevant is because it helps us circumvent the process of building a single-step classifier that performs with high accuracy. Building one such robust single-step classifier is a computationally intensive process.

Consider an example where we have to detect an object like, say, a tennis ball. In order to build a detector, we need a system that can learn what a tennis ball looks like. It should be able to infer whether or not a given image contains a tennis ball. We need to train this system using a lot of images of tennis balls. We also need a lot of images that don't contain tennis balls as well. This helps the system learn how to differentiate between objects.

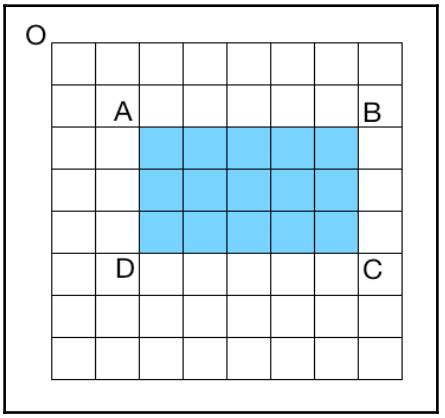
If we build an accurate model, it will be complex. Hence we won't be able to run it in real time. If it's too simple, it might not be accurate. This trade off between speed and accuracy is frequently encountered in the world of machine learning. The Viola-Jones method overcomes this problem by building a set of simple classifiers. These classifiers are then cascaded to form a unified classifier that's robust and accurate.

Let's see how to use this to do face detection. In order to build a machine learning system to detect faces, we first need to build a feature extractor. The machine learning algorithms will use these features to understand what a face looks like. This is where Haar features become relevant. They are just simple summations and differences of patches across the image. Haar features are really easy to compute. In order to make it robust to scale, we do this at multiple image sizes. If you want to learn more about this in a tutorial format, you can check out this link: `http://www.cs.ubc.ca/~lowe/425/slides/13-ViolaJones.pdf`.

Once the features are extracted, we pass them through our boosted cascade of simple classifiers. We check various rectangular sub-regions in the image and keep discarding the ones that don't contain faces. This helps us arrive at the final answer quickly. In order to compute these features quickly, they used a concept known as integral images.

Using integral images for feature extraction

In order to compute Haar features, we have to compute the summations and differences of many sub-regions in the image. We need to compute these summations and differences at multiple scales, which makes it a very computationally intensive process. In order to build a real time system, we use integral images. Consider the following figure:



If we want to compute the sum of the rectangle ABCD in this image, we don't need to go through each pixel in that rectangular area. Let's say OP indicates the area of the rectangle formed by the top left corner *O* and the point *P* on the diagonally opposite corners of the rectangle. To calculate the area of the rectangle ABCD, we can use the following formula:

Area of the rectangle ABCD = OC – (OB + OD – OA)

What's so special about this formula? If you notice, we didn't have to iterate through anything or recalculate any rectangle areas. All the values on the right hand side of the equation are already available because they were computed during earlier cycles. We directly used them to compute the area of this rectangle. Let's see how to build a face detector.

Create a new python file and import the following packages:

```
import cv2
import numpy as np
```

Load the Haar cascade file corresponding to face detection:

```
# Load the Haar cascade file
face_cascade = cv2.CascadeClassifier(
        'haar_cascade_files/haarcascade_frontalface_default.xml')

# Check if the cascade file has been loaded correctly
if face_cascade.empty():
   raise IOError('Unable to load the face cascade classifier xml file')
```

Initialize the video capture object and define the scaling factor:

```
# Initialize the video capture object
cap = cv2.VideoCapture(0)

# Define the scaling factor
scaling_factor = 0.5
```

Iterate indefinitely until the user presses the *Esc* key. Capture the current frame:

```
# Iterate until the user hits the 'Esc' key
while True:
    # Capture the current frame
    _, frame = cap.read()
```

Resize the frame:

```
    # Resize the frame
    frame = cv2.resize(frame, None,
            fx=scaling_factor, fy=scaling_factor,
            interpolation=cv2.INTER_AREA)
```

Convert the image to grayscale:

```
# Convert to grayscale
gray = cv2.cvtColor(frame, cv2.COLOR_BGR2GRAY)
```

Run the face detector on the grayscale image:

```
# Run the face detector on the grayscale image
face_rects = face_cascade.detectMultiScale(gray, 1.3, 5)
```

Iterate through the detected faces and draw rectangles around them:

```
# Draw a rectangle around the face
for (x,y,w,h) in face_rects:
    cv2.rectangle(frame, (x,y), (x+w,y+h), (0,255,0), 3)
```

Display the output:

```
# Display the output
cv2.imshow('Face Detector', frame)
```

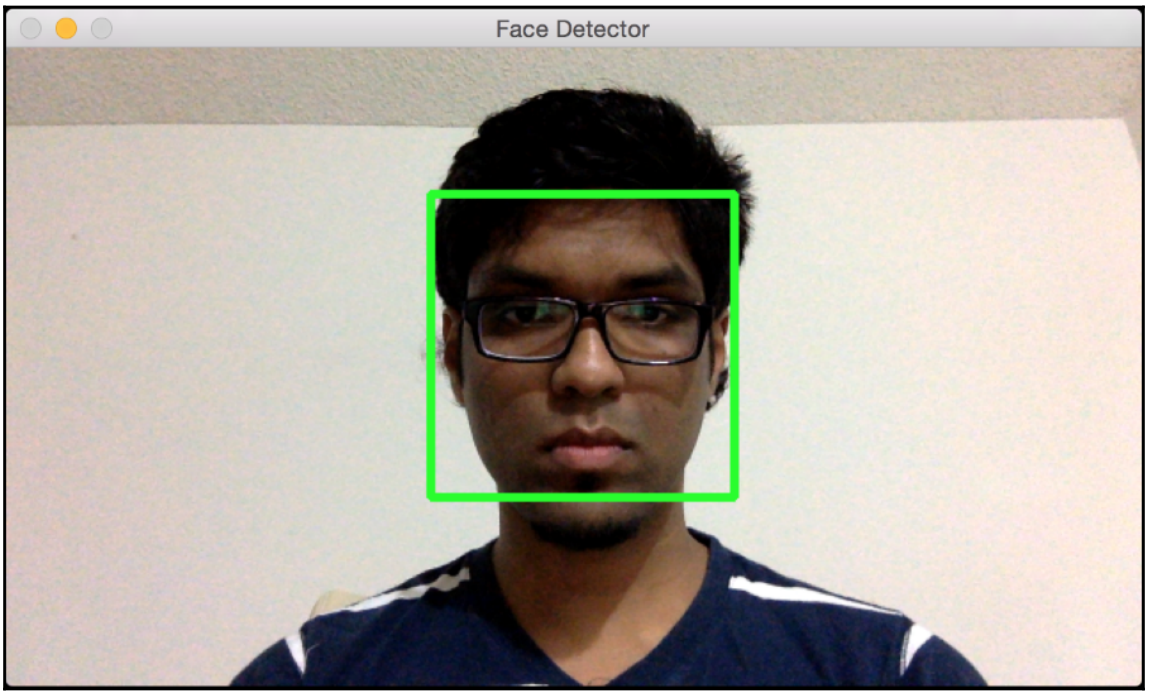
Check if the user pressed the *Esc* key. If so, exit the loop:

```
# Check if the user hit the 'Esc' key
c = cv2.waitKey(1)
if c == 27:
    break
```

Once you exit the loop, make sure to release the video capture object and close all the windows properly:

```
# Release the video capture object
cap.release()

# Close all the windows
cv2.destroyAllWindows()
```

The full code is given in the file `face_detector.py` provided to you. If you run the code, you will see something like this:



Eye detection and tracking

Eye detection works very similarly to face detection. Instead of using a face cascade file, we will use an eye cascade file. Create a new python file and import the following packages:

```
import cv2
import numpy as np
```

Load the Haar cascade files corresponding to face and eye detection:

```
# Load the Haar cascade files for face and eye
face_cascade =
cv2.CascadeClassifier('haar_cascade_files/haarcascade_frontalface_default.x
ml')
eye_cascade =
cv2.CascadeClassifier('haar_cascade_files/haarcascade_eye.xml')
```

```
# Check if the face cascade file has been loaded correctly
if face_cascade.empty():
   raise IOError('Unable to load the face cascade classifier xml file')

# Check if the eye cascade file has been loaded correctly
if eye_cascade.empty():
   raise IOError('Unable to load the eye cascade classifier xml file')
```

Initialize the video capture object and define the scaling factor:

```
# Initialize the video capture object
cap = cv2.VideoCapture(0)
# Define the scaling factor
ds_factor = 0.5
```

Iterate indefinitely until the user presses the *Esc* key:

```
# Iterate until the user hits the 'Esc' key
while True:
    # Capture the current frame
    _, frame = cap.read()
```

Resize the frame:

```
    # Resize the frame
    frame = cv2.resize(frame, None, fx=ds_factor, fy=ds_factor,
interpolation=cv2.INTER_AREA)
```

Convert the frame from RGB to grayscale:

```
    # Convert to grayscale
    gray = cv2.cvtColor(frame, cv2.COLOR_BGR2GRAY)
```

Run the face detector:

```
    # Run the face detector on the grayscale image
    faces = face_cascade.detectMultiScale(gray, 1.3, 5)
```

For each face detected, run the eye detector within that region:

```
    # For each face that's detected, run the eye detector
    for (x,y,w,h) in faces:
        # Extract the grayscale face ROI
        roi_gray = gray[y:y+h, x:x+w]
```

Extract the region of interest and run the eye detector:

```
# Extract the color face ROI
roi_color = frame[y:y+h, x:x+w]

# Run the eye detector on the grayscale ROI
eyes = eye_cascade.detectMultiScale(roi_gray)
```

Draw circles around the eyes and display the output:

```
# Draw circles around the eyes
for (x_eye,y_eye,w_eye,h_eye) in eyes:
    center = (int(x_eye + 0.5*w_eye), int(y_eye + 0.5*h_eye))
    radius = int(0.3 * (w_eye + h_eye))
    color = (0, 255, 0)
    thickness = 3
    cv2.circle(roi_color, center, radius, color, thickness)
# Display the output
cv2.imshow('Eye Detector', frame)
```

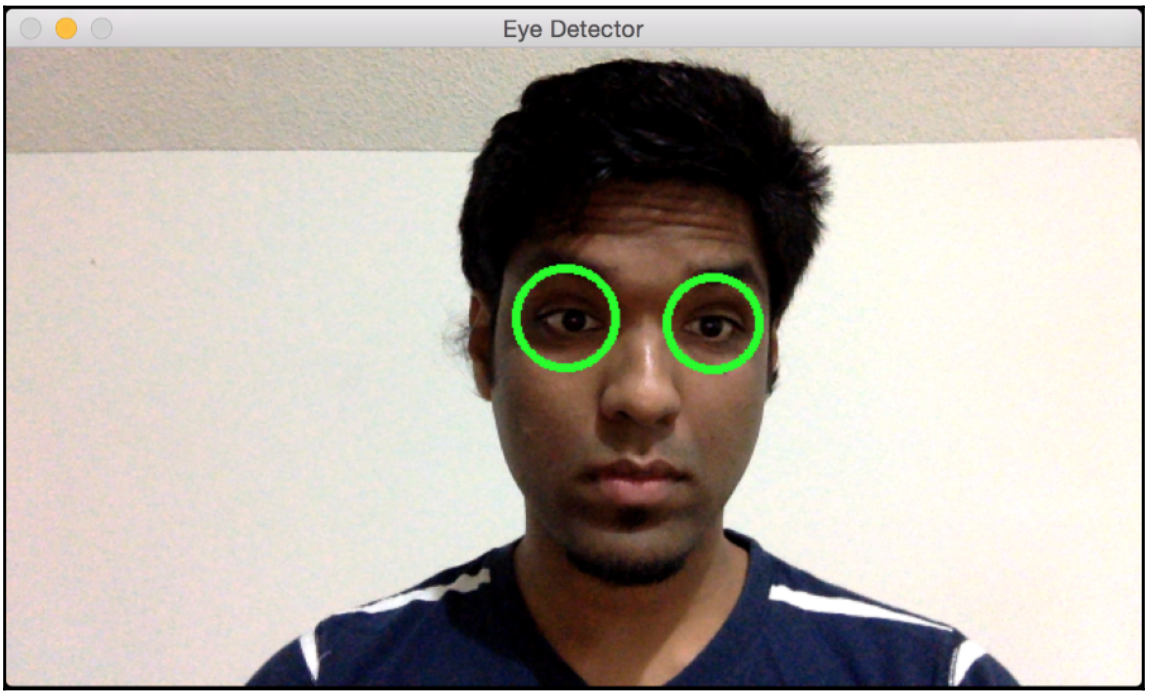
If the user presses the *Esc* key, exit the loop:

```
# Check if the user hit the 'Esc' key
c = cv2.waitKey(1)
if c == 27:
    break
```

Once you exit the loop, make sure to release the video capture object and close all the windows:

```
# Release the video capture object
cap.release()

# Close all the windows
cv2.destroyAllWindows()
```

The full code is given in the file `eye_detector.py` provided to you. If you run the code, you will see something like this:



Summary

In this chapter, we learnt about object detection and tracking. We understood how to install OpenCV with Python support on various operating systems. We learnt about frame differencing and used it to detect the moving parts in a video. We discussed how to track human skin using color spaces. We talked about background subtraction and how it can be used to track objects in static scenes. We built an interactive object tracker using the CAMShift algorithm.

We learnt how to build an optical flow based tracker. We discussed face detection techniques and understood the concepts of Haar cascades and integral images. We used this technique to build an eye detector and tracker. In the next chapter, we will discuss artificial neural networks and use those techniques to build an optical character recognition engine.

14
Artificial Neural Networks

In this chapter, we are going to learn about artificial neural networks. We will start with an introduction to artificial neural networks and the installation of the relevant library. We will discuss perceptrons and how to build a classifier based on them. We will learn about single layer neural networks and multilayer neural networks. We will see how to use neural networks to build a vector quantizer. We will analyze sequential data using recurrent neural networks. We will then use artificial neural networks to build an optical character recognition engine.

By the end of this chapter, you will know about:

- Introduction to artificial neural networks
- Building a Perceptron based classifier
- Constructing a single layer neural network
- Constructing a multilayer neural network
- Building a vector quantizer
- Analyzing sequential data using recurrent neural networks
- Visualizing characters in an Optical Character Recognition (OCR) database
- Building an Optical Character Recognition (OCR) engine

Introduction to artificial neural networks

One of the fundamental premises of Artificial Intelligence is to build machines that can perform tasks that require human intelligence. The human brain is amazing at learning new things. Why not use the model of the human brain to build a machine? An artificial neural network is a model designed to simulate the learning process of the human brain.

Artificial neural networks are designed such that they can identify the underlying patterns in data and learn from them. They can be used for various tasks such as classification, regression, segmentation, and so on. We need to convert any given data into numerical form before feeding it into the neural network. For example, we deal with many different types of data including visual, textual, time-series, and so on. We need to figure out how to represent problems in a way that can be understood by artificial neural networks.

Building a neural network

The human learning process is hierarchical. We have various stages in our brain's neural network and each stage corresponds to a different granularity. Some stages learn simple things and some stages learn more complex things. Let's consider an example of visually recognizing an object. When we look at a box, the first stage identifies simple things like corners and edges. The next stage identifies the generic shape and the stage after that identifies what kind of object it is. This process differs for different tasks, but you get the idea! By building this hierarchy, our human brain quickly separates the concepts and identifies the given object.

To simulate the learning process of the human brain, an artificial neural network is built using layers of neurons. These neurons are inspired by the biological neurons we discussed in the previous paragraph. Each layer in an artificial neural network is a set of independent neurons. Each neuron in a layer is connected to neurons in the adjacent layer.

Training a neural network

If we are dealing with N-dimensional input data, then the input layer will consist of N neurons. If we have M distinct classes in our training data, then the output layer will consist of M neurons. The layers between the input and output layers are called hidden layers. A simple neural network will consist of a couple of layers and a deep neural network will consist of many layers.

Consider the case where we want to use a neural network to classify the given data. The first step is to collect the appropriate training data and label it. Each neuron acts as a simple function and the neural network trains itself until the error goes below a certain value. The error is basically the difference between the predicted output and the actual output. Based on how big the error is, the neural network adjusts itself and retrains until it gets closer to the solution. You can learn more about neural networks at `http://pages.cs.wisc.edu/~bolo/shipyard/neural/local.html`.

We will be using a library called NeuroLab in this chapter. You can find more about it at `https://pythonhosted.org/neurolab`. You can install it by running the following command on your Terminal:

```
$ pip3 install neurolab
```

Once you have installed it, you can proceed to the next section.

Building a Perceptron based classifier

A **Perceptron** is the building block of an artificial neural network. It is a single neuron that takes inputs, performs computation on them, and then produces an output. It uses a simple linear function to make the decision. Let's say we are dealing with an N-dimension input data point. A Perceptron computes the weighted summation of those N numbers and it then adds a constant to produce the output. The constant is called the bias of the neuron. It is remarkable to note that these simple Perceptrons are used to design very complex deep neural networks. Let's see how to build a Perceptron based classifier using NeuroLab.

Create a new Python file and import the following packages:

```
import numpy as np
import matplotlib.pyplot as plt
import neurolab as nl
```

Load the input data from the text file `data_perceptron.txt` provided to you. Each line contains space separated numbers where the first two numbers are the features and the last number is the label:

```
# Load input data
text = np.loadtxt('data_perceptron.txt')
```

Separate the text into datapoints and labels:

```
# Separate datapoints and labels
data = text[:, :2]
labels = text[:, 2].reshape((text.shape[0], 1))
```

Plot the datapoints:

```
# Plot input data
plt.figure()
plt.scatter(data[:,0], data[:,1])
plt.xlabel('Dimension 1')
plt.ylabel('Dimension 2')
plt.title('Input data')
```

Define the maximum and minimum values that each dimension can take:

```
# Define minimum and maximum values for each dimension
dim1_min, dim1_max, dim2_min, dim2_max = 0, 1, 0, 1
```

Since the data is separated into two classes, we just need one bit to represent the output. So the output layer will contain a single neuron.

```
# Number of neurons in the output layer
num_output = labels.shape[1]
```

We have a dataset where the datapoints are two-dimensional. Let's define a Perceptron with two input neurons, where we assign one neuron for each dimension.

```
# Define a perceptron with 2 input neurons (because we
# have 2 dimensions in the input data)
dim1 = [dim1_min, dim1_max]
dim2 = [dim2_min, dim2_max]
perceptron = nl.net.newp([dim1, dim2], num_output)
```

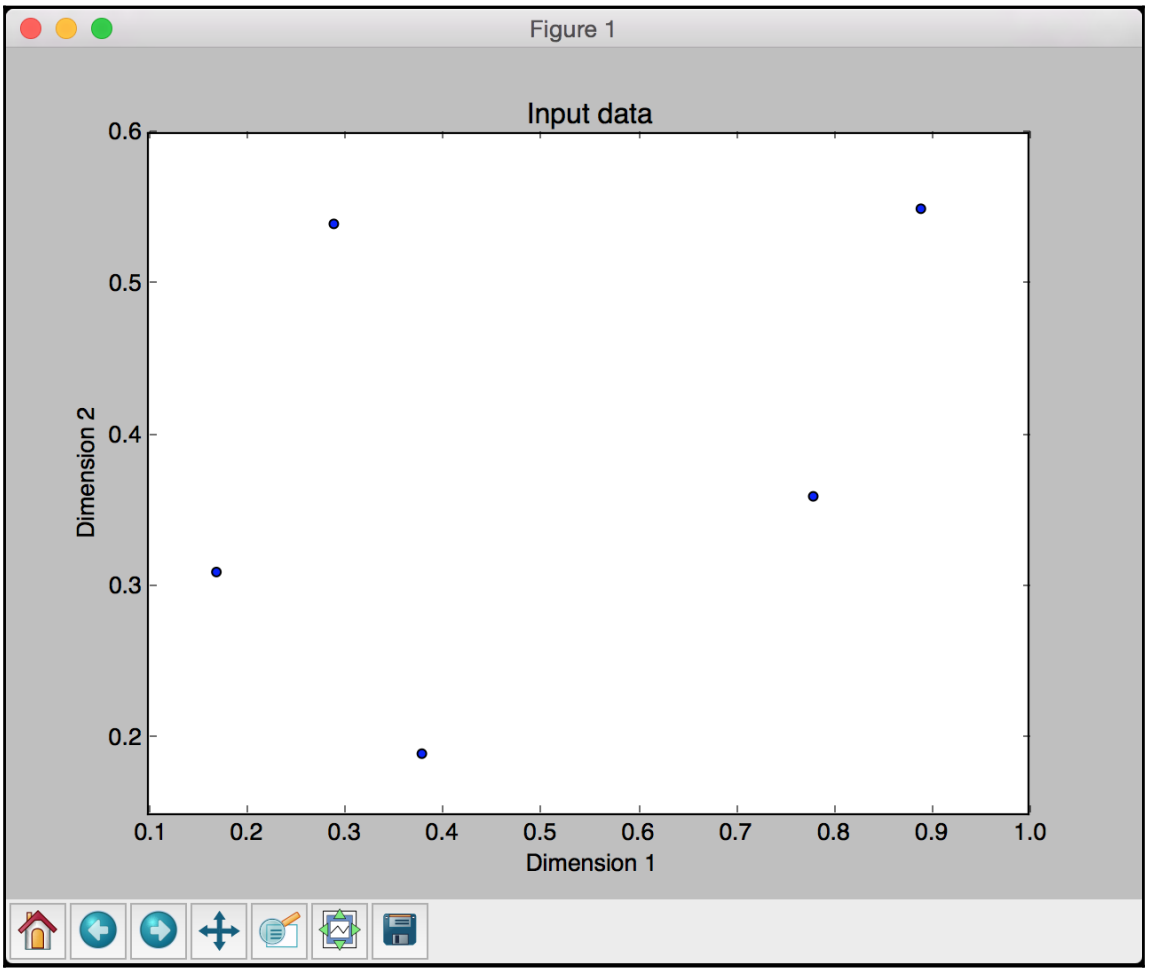
Train the perceptron with the training data:

```
# Train the perceptron using the data
error_progress = perceptron.train(data, labels, epochs=100, show=20,
lr=0.03)
```

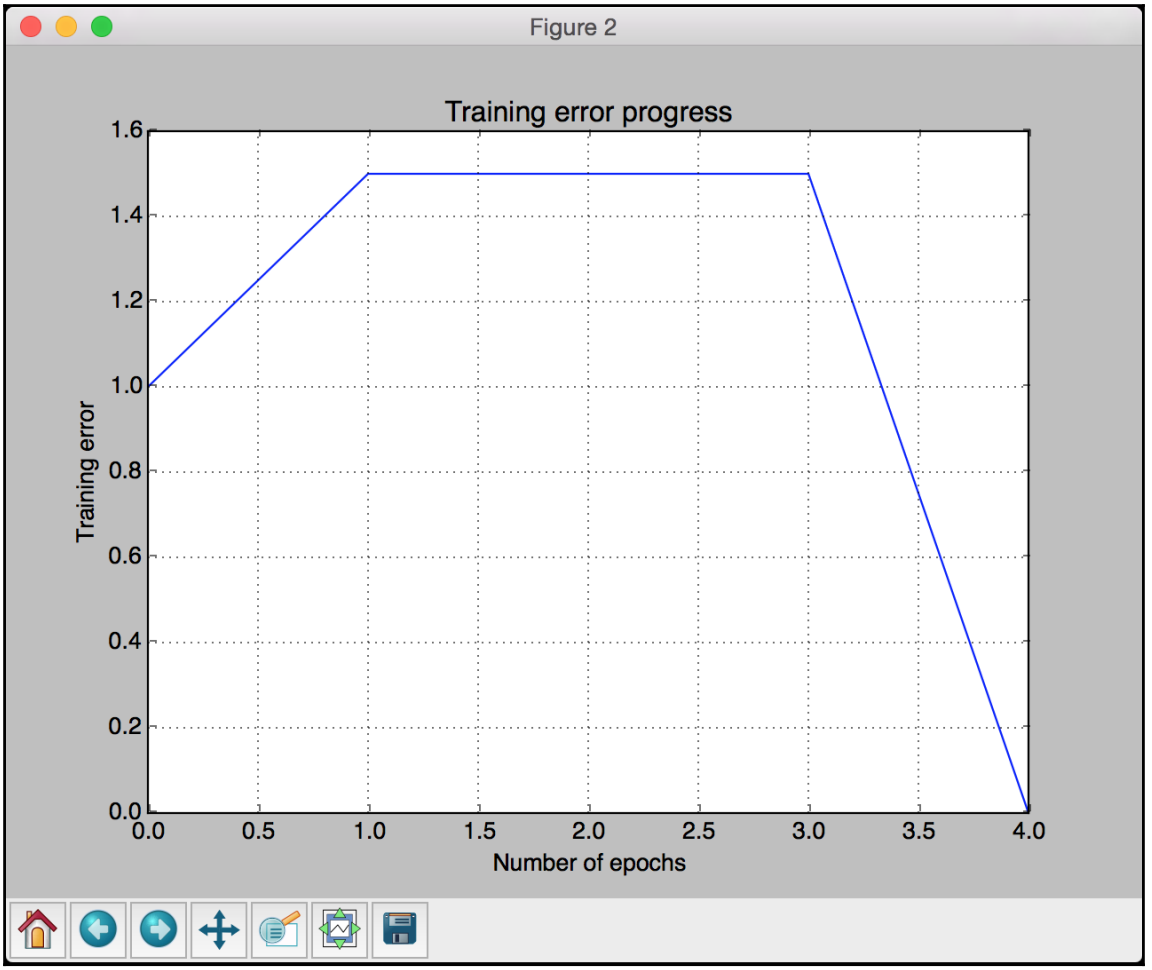
Plot the training progress using the error metric:

```
# Plot the training progress
plt.figure()
plt.plot(error_progress)
plt.xlabel('Number of epochs')
plt.ylabel('Training error')
plt.title('Training error progress')
plt.grid()

plt.show()
```

The full code is given in the file `perceptron_classifier.py`. If you run the code, you will get two output screenshot. The first screenshot indicates the input data points:



The second screenshot represents the training progress using the error metric:



We can observe from the preceding screenshot, the error goes down to **0** at the end of the fourth epoch.

Constructing a single layer neural network

A perceptron is a good start, but it cannot do much. The next step is to have a set of neurons act as a unit to see what we can achieve. Let's create a single neural network that consists of independent neurons acting on input data to produce the output.

Create a new Python file and import the following packages:

```
import numpy as np
import matplotlib.pyplot as plt
import neurolab as nl
```

We will use the input data from the file `data_simple_nn.txt` provided to you. Each line in this file contains four numbers. The first two numbers form the datapoint and the last two numbers are the labels. Why do we need to assign two numbers for labels? Because we have four distinct classes in our dataset, so we need two bits to represent them. Let us go ahead and load the data:

```
# Load input data
text = np.loadtxt('data_simple_nn.txt')
```

Separate the data into datapoints and labels:

```
# Separate it into datapoints and labels
data = text[:, 0:2]
labels = text[:, 2:]
```

Plot the input data:

```
# Plot input data
plt.figure()
plt.scatter(data[:,0], data[:,1])
plt.xlabel('Dimension 1')
plt.ylabel('Dimension 2')
plt.title('Input data')
```

Extract the minimum and maximum values for each dimension (we don't need to hardcode it like we did in the previous section):

```
# Minimum and maximum values for each dimension
dim1_min, dim1_max = data[:,0].min(), data[:,0].max()
dim2_min, dim2_max = data[:,1].min(), data[:,1].max()
```

Define the number of neurons in the output layer:

```
# Define the number of neurons in the output layer
num_output = labels.shape[1]
```

Define a single layer neural network using the above parameters:

```
# Define a single-layer neural network
dim1 = [dim1_min, dim1_max]
dim2 = [dim2_min, dim2_max]
nn = nl.net.newp([dim1, dim2], num_output)
```

Train the neural network using training data:

```
# Train the neural network
error_progress = nn.train(data, labels, epochs=100, show=20, lr=0.03)
```

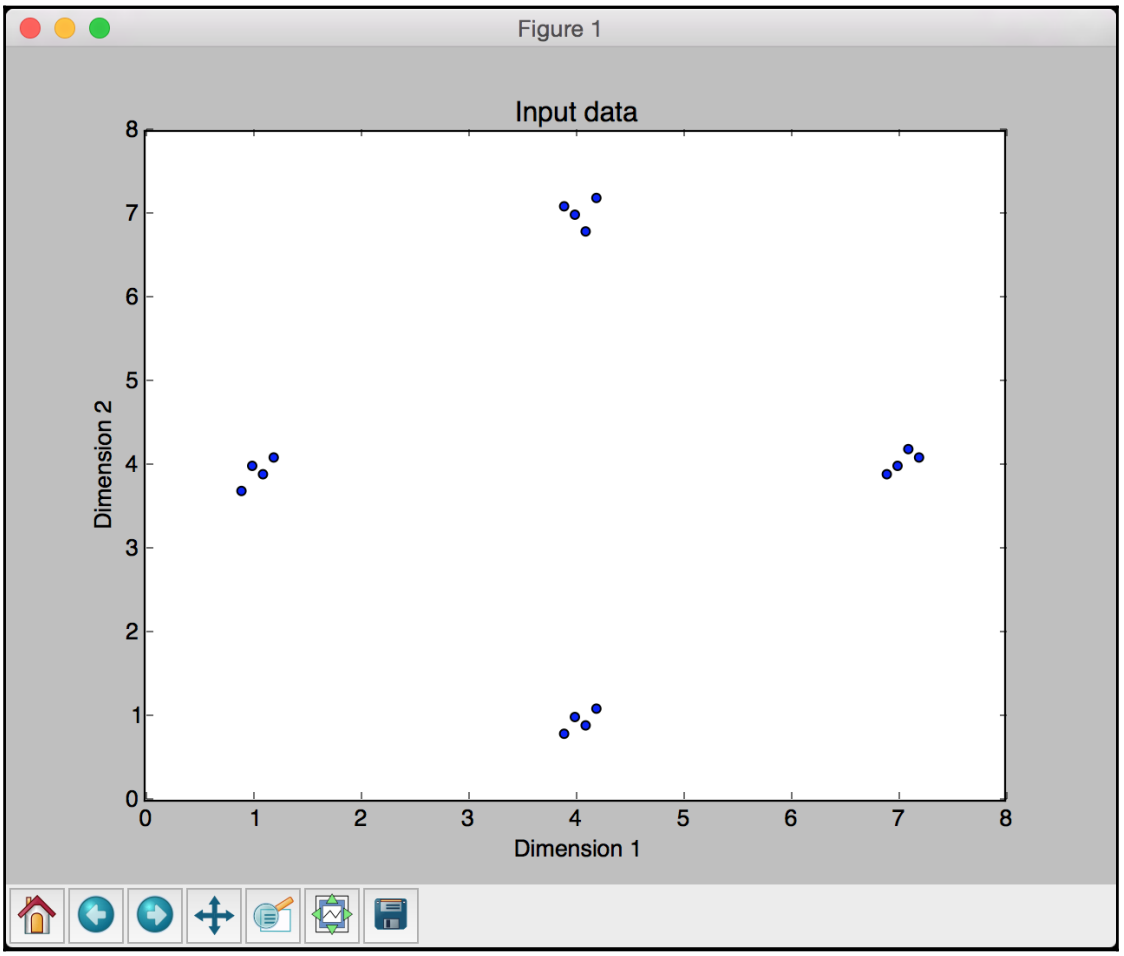
Plot the training progress:

```
# Plot the training progress
plt.figure()
plt.plot(error_progress)
plt.xlabel('Number of epochs')
plt.ylabel('Training error')
plt.title('Training error progress')
plt.grid()

plt.show()
```

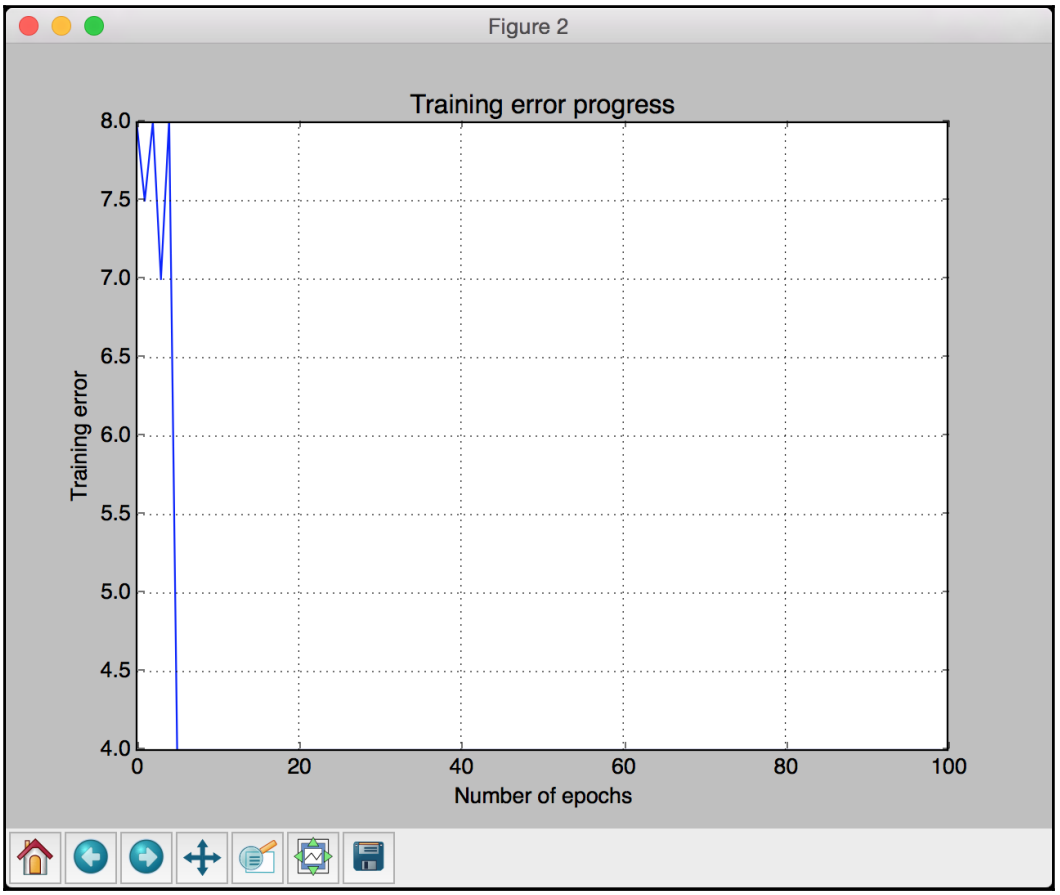
Define some sample test datapoints and run the network on those points:

```
# Run the classifier on test datapoints
print('\nTest results:')
data_test = [[0.4, 4.3], [4.4, 0.6], [4.7, 8.1]]
for item in data_test:
    print(item, '-->', nn.sim([item])[0])
```

The full code is given in the file `simple_neural_network.py`. If you run the code, you will get two screenshot. The first screenshot represents the input datapoints:



The second screenshot shows the training progress:



Once you close the graphs, you will see the following printed on your Terminal:

```
Epoch: 20; Error: 4.0;
Epoch: 40; Error: 4.0;
Epoch: 60; Error: 4.0;
Epoch: 80; Error: 4.0;
Epoch: 100; Error: 4.0;
The maximum number of train epochs is reached

Test results:
[0.4, 4.3] --> [ 0.  0.]
[4.4, 0.6] --> [ 1.  0.]
[4.7, 8.1] --> [ 1.  1.]
```

If you locate these test data points on a 2D graph, you can visually verify that the predicted outputs are correct.

Constructing a multilayer neural network

In order to enable higher accuracy, we need to give more freedom to the neural network. This means that a neural network needs more than one layer to extract the underlying patterns in the training data. Let's create a multilayer neural network to achieve that.

Create a new Python file and import the following packages:

```
import numpy as np
import matplotlib.pyplot as plt
import neurolab as nl
```

In the previous two sections, we saw how to use a neural network as a classifier. In this section, we will see how to use a multilayer neural network as a regressor. Generate some sample data points based on the equation $y = 3x^2 + 5$ and then normalize the points:

```
# Generate some training data
min_val = -15
max_val = 15
num_points = 130
x = np.linspace(min_val, max_val, num_points)
y = 3 * np.square(x) + 5
y /= np.linalg.norm(y)
```

Reshape the above variables to create a training dataset:

```
# Create data and labels
data = x.reshape(num_points, 1)
labels = y.reshape(num_points, 1)
```

Plot the input data:

```
# Plot input data
plt.figure()
plt.scatter(data, labels)
plt.xlabel('Dimension 1')
plt.ylabel('Dimension 2')
plt.title('Input data')
```

Define a multilayer neural network with two hidden layers. You are free to design a neural network any way you want. For this case, let's have 10 neurons in the first layer and 6 neurons in the second layer. Our task is to predict the value, so the output layer will contain a single neuron:

```
# Define a multilayer neural network with 2 hidden layers;
# First hidden layer consists of 10 neurons
# Second hidden layer consists of 6 neurons
# Output layer consists of 1 neuron
nn = nl.net.newff([[min_val, max_val]], [10, 6, 1])
```

Set the training algorithm to gradient descent:

```
# Set the training algorithm to gradient descent
nn.trainf = nl.train.train_gd
```

Train the neural network using the training data that was generated:

```
# Train the neural network
error_progress = nn.train(data, labels, epochs=2000, show=100, goal=0.01)
```

Run the neural network on the training datapoints:

```
# Run the neural network on training datapoints
output = nn.sim(data)
y_pred = output.reshape(num_points)
```

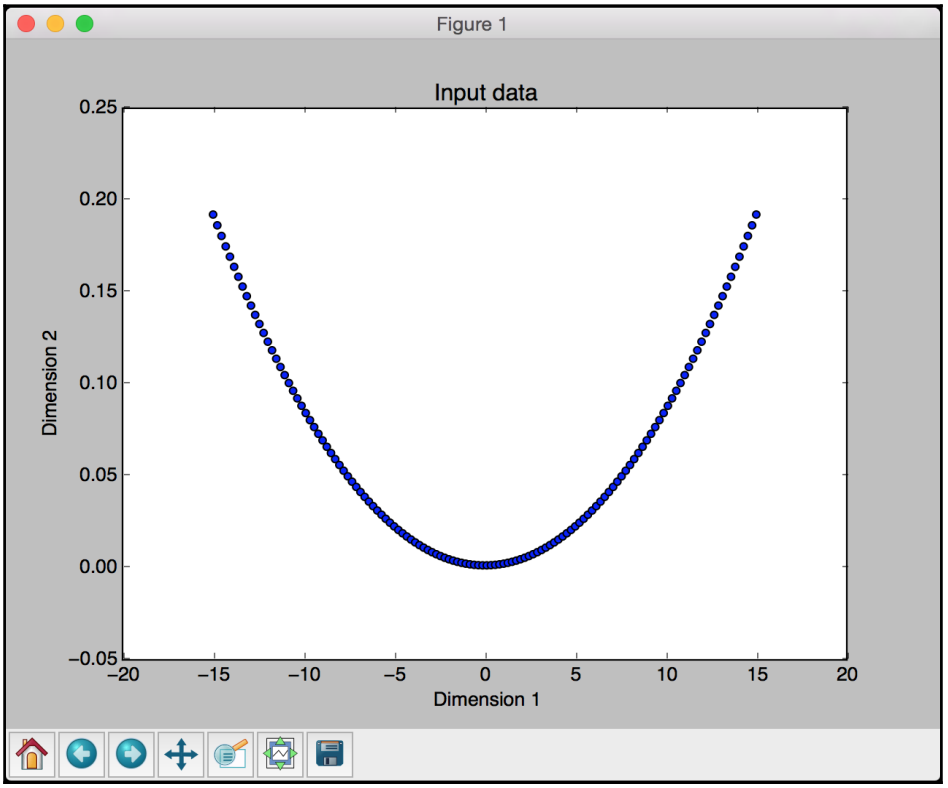
Plot the training progress:

```
# Plot training error
plt.figure()
plt.plot(error_progress)
plt.xlabel('Number of epochs')
plt.ylabel('Error')
plt.title('Training error progress')
```

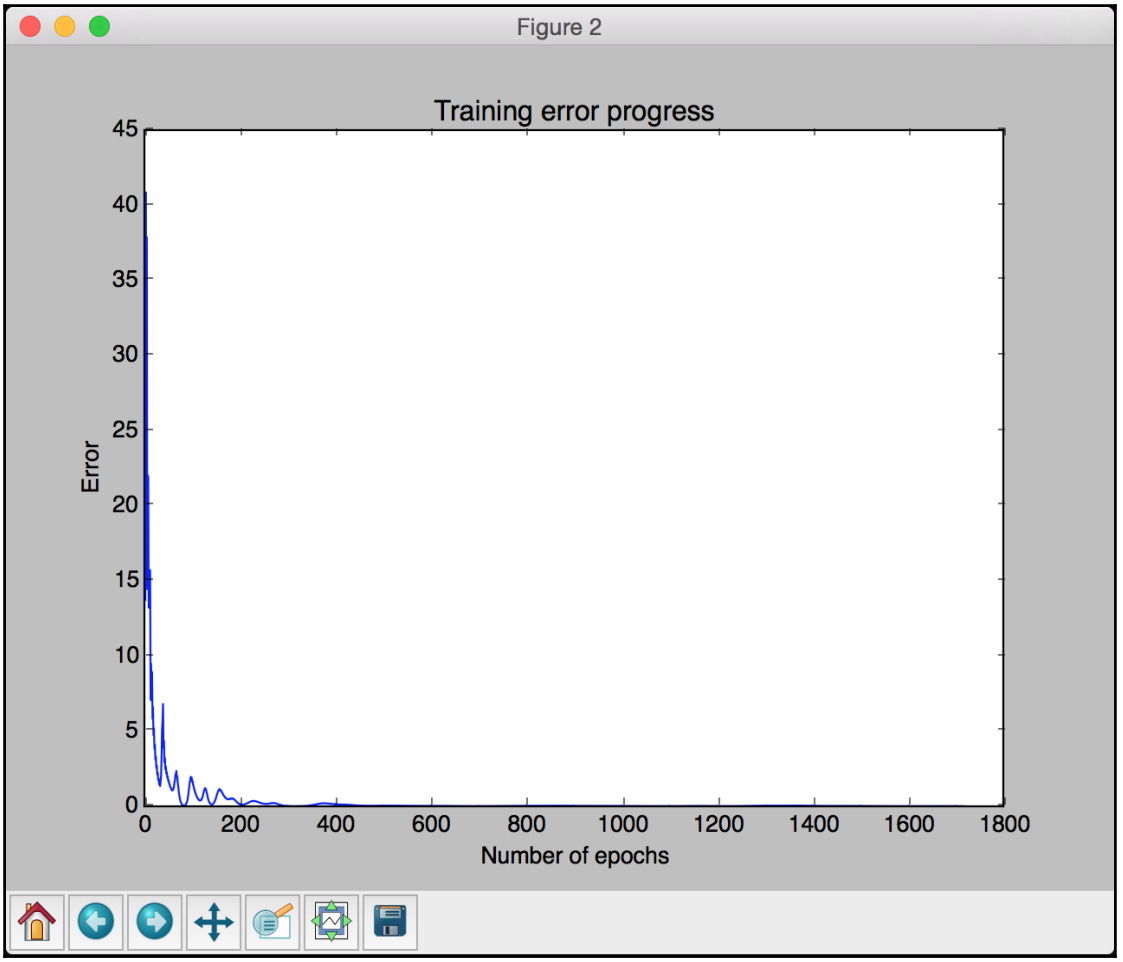
Plot the predicted output:

```
# Plot the output
x_dense = np.linspace(min_val, max_val, num_points * 2)
y_dense_pred =
nn.sim(x_dense.reshape(x_dense.size,1)).reshape(x_dense.size)

plt.figure()
plt.plot(x_dense, y_dense_pred, '-', x, y, '.', x, y_pred, 'p')
plt.title('Actual vs predicted')

plt.show()
```

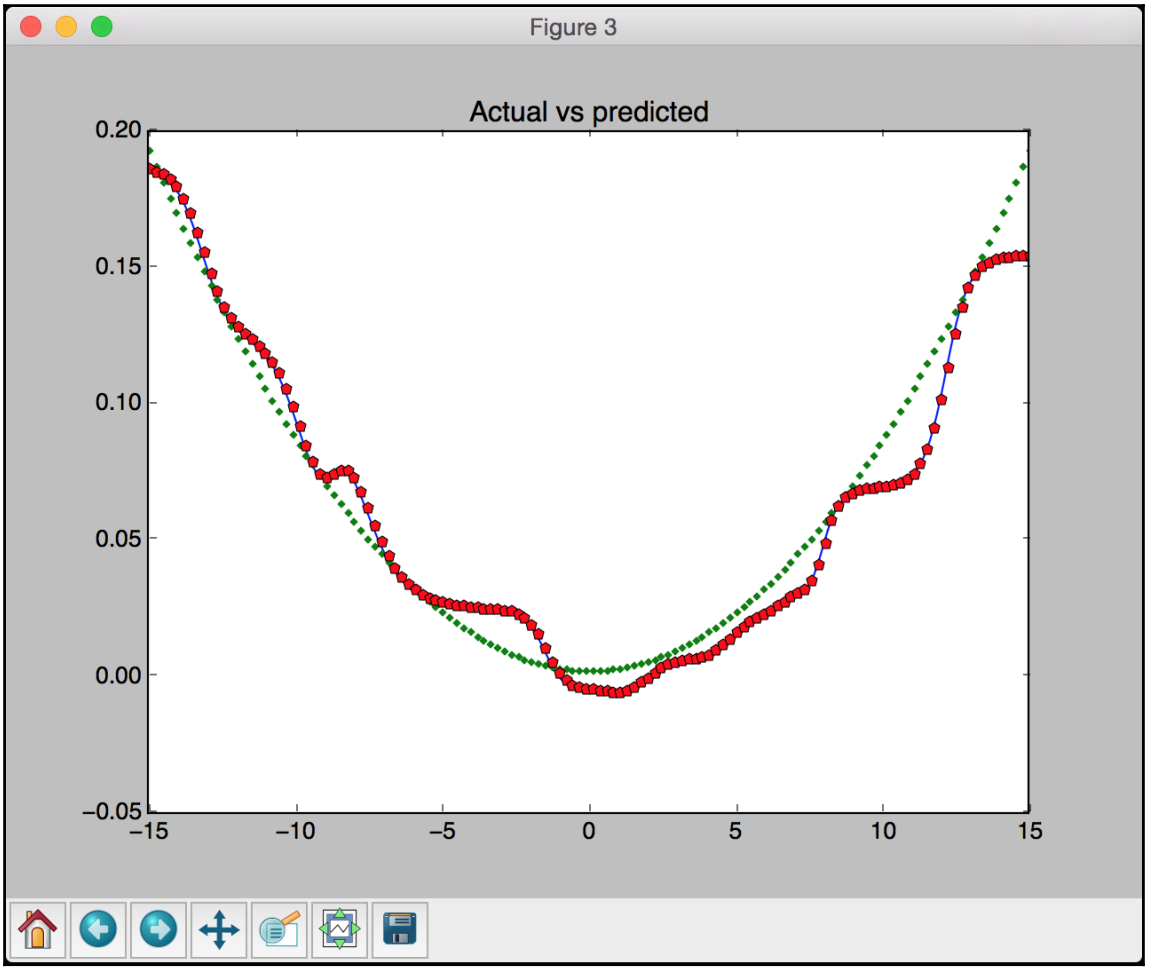
The full code is given in the file `multilayer_neural_network.py`. If you run the code, you will get three screenshot. The first screenshot shows the input data:



The second screenshot shows the training progress:



The third screenshot shows the predicted output overlaid on top of input data:



The predicted output seems to follow the general trend. If you continue to train the network and reduce the error, you will see that the predicted output will match the input curve even more accurately.

You will see the following printed on your Terminal:

```
Epoch: 100; Error: 1.9247718251621995;
Epoch: 200; Error: 0.15723294798079526;
Epoch: 300; Error: 0.021680213116912858;
Epoch: 400; Error: 0.1381761995539017;
Epoch: 500; Error: 0.04392553381948737;
Epoch: 600; Error: 0.02975401597014979;
Epoch: 700; Error: 0.014228560930227126;
Epoch: 800; Error: 0.03460207842970052;
Epoch: 900; Error: 0.035934053149433196;
Epoch: 1000; Error: 0.025833284445815966;
Epoch: 1100; Error: 0.013672412879982398;
Epoch: 1200; Error: 0.01776586425692384;
Epoch: 1300; Error: 0.04310242610384976;
Epoch: 1400; Error: 0.03799681296096611;
Epoch: 1500; Error: 0.02467030041520845;
Epoch: 1600; Error: 0.010094873168855236;
Epoch: 1700; Error: 0.01210866043021068;
The goal of learning is reached
```

Building a vector quantizer

Vector Quantization is a quantization technique where the input data is represented by a fixed number of representative points. It is the N-dimensional equivalent of rounding off a number. This technique is commonly used in multiple fields such as image recognition, semantic analysis, and data science. Let's see how to use artificial neural networks to build a vector quantizer.

Create a new Python file and import the following packages:

```
import numpy as np
import matplotlib.pyplot as plt
import neurolab as nl
```

Load the input data from the file `data_vector_quantization.txt`. Each line in this file contains six numbers. The first two numbers form the datapoint and the last four numbers form a one-hot encoded label. There are four classes overall.

```
# Load input data
text = np.loadtxt('data_vector_quantization.txt')
```

Separate the text into data and labels:

```
# Separate it into data and labels
data = text[:, 0:2]
labels = text[:, 2:]
```

Define a neural network with two layers where we have 10 neurons in the input layer and 4 neurons in the output layer:

```
# Define a neural network with 2 layers:
# 10 neurons in input layer and 4 neurons in output layer
num_input_neurons = 10
num_output_neurons = 4
weights = [1/num_output_neurons] * num_output_neurons
nn = nl.net.newlvq(nl.tool.minmax(data), num_input_neurons, weights)
```

Train the neural network using the training data:

```
# Train the neural network
_ = nn.train(data, labels, epochs=500, goal=-1)
```

In order to visualize the output clusters, let's create a grid of points:

```
# Create the input grid
xx, yy = np.meshgrid(np.arange(0, 10, 0.2), np.arange(0, 10, 0.2))
xx.shape = xx.size, 1
yy.shape = yy.size, 1
grid_xy = np.concatenate((xx, yy), axis=1)
```

Evaluate the grid of points using the neural network:

```
# Evaluate the input grid of points
grid_eval = nn.sim(grid_xy)
```

Extract the four classes:

```
# Define the 4 classes
class_1 = data[labels[:,0] == 1]
class_2 = data[labels[:,1] == 1]
class_3 = data[labels[:,2] == 1]
class_4 = data[labels[:,3] == 1]
```

Extract the grids corresponding to those four classes:

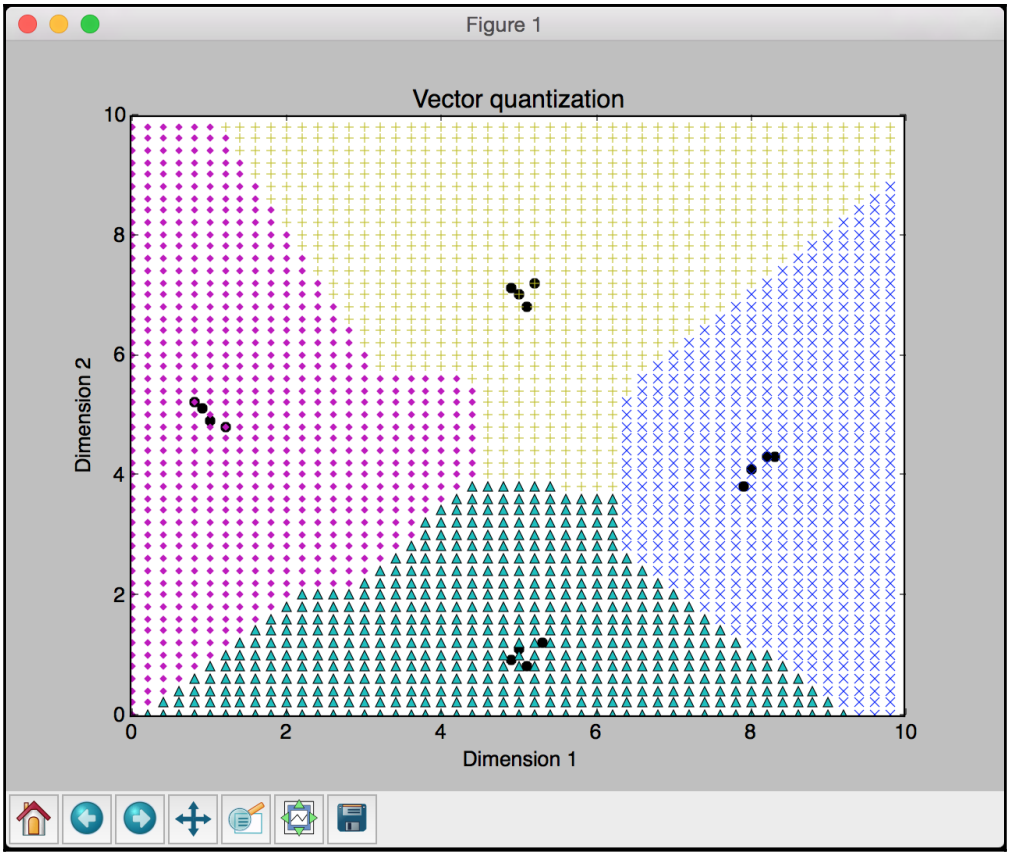
```
# Define X-Y grids for all the 4 classes
grid_1 = grid_xy[grid_eval[:,0] == 1]
grid_2 = grid_xy[grid_eval[:,1] == 1]
grid_3 = grid_xy[grid_eval[:,2] == 1]
grid_4 = grid_xy[grid_eval[:,3] == 1]
```

Plot the outputs:

```
# Plot the outputs
plt.plot(class_1[:,0], class_1[:,1], 'ko',
        class_2[:,0], class_2[:,1], 'ko',
```

```
            class_3[:,0], class_3[:,1], 'ko',
            class_4[:,0], class_4[:,1], 'ko')
    plt.plot(grid_1[:,0], grid_1[:,1], 'm.',
            grid_2[:,0], grid_2[:,1], 'bx',
            grid_3[:,0], grid_3[:,1], 'c^',
            grid_4[:,0], grid_4[:,1], 'y+')
    plt.axis([0, 10, 0, 10])
    plt.xlabel('Dimension 1')
    plt.ylabel('Dimension 2')
    plt.title('Vector quantization')

    plt.show()
```

The full code is given in the file `vector_quantizer.py`. If you run the code, you will get the following sccreenshot that shows the input data points and the boundaries between clusters:



You will see the following printed on your Terminal:

```
Epoch: 100; Error: 0.0;
Epoch: 200; Error: 0.0;
Epoch: 300; Error: 0.0;
Epoch: 400; Error: 0.0;
Epoch: 500; Error: 0.0;
The maximum number of train epochs is reached
```

Analyzing sequential data using recurrent neural networks

We have been dealing with static data so far. Artificial neural networks are good at building models for sequential data too. In particular, recurrent neural networks are great at modeling sequential data. Perhaps time-series data is the most commonly occurring form of sequential data in our world. You can learn more about recurrent neural networks at `http://www.wildml.com/2015/09/recurrent-neural-networks-tutorial-part-1-introduction-to-rnns`. When we are working with time-series data, we cannot just use generic learning models. We need to characterize the temporal dependencies in our data so that we can build a robust model. Let's see how to build it.

Create a new python file and import the following packages:

```
import numpy as np
import matplotlib.pyplot as plt
import neurolab as nl
```

Define a function to generate the waveforms. Start by defining four sine waves:

```
def get_data(num_points):
    # Create sine waveforms
    wave_1 = 0.5 * np.sin(np.arange(0, num_points))
    wave_2 = 3.6 * np.sin(np.arange(0, num_points))
    wave_3 = 1.1 * np.sin(np.arange(0, num_points))
    wave_4 = 4.7 * np.sin(np.arange(0, num_points))
```

Create varying amplitudes for the overall waveform:

```
    # Create varying amplitudes
    amp_1 = np.ones(num_points)
    amp_2 = 2.1 + np.zeros(num_points)
    amp_3 = 3.2 * np.ones(num_points)
    amp_4 = 0.8 + np.zeros(num_points)
```

Create the overall waveform:

```
    wave = np.array([wave_1, wave_2, wave_3, wave_4]).reshape(num_points *
4, 1)
    amp = np.array([[amp_1, amp_2, amp_3, amp_4]]).reshape(num_points * 4,
1)

    return wave, amp
```

Define a function to visualize the output of the neural network:

```
# Visualize the output
def visualize_output(nn, num_points_test):
    wave, amp = get_data(num_points_test)
    output = nn.sim(wave)
    plt.plot(amp.reshape(num_points_test * 4))
    plt.plot(output.reshape(num_points_test * 4))
```

Define the main function and create a waveform:

```
if __name__=='__main__':
    # Create some sample data
    num_points = 40
    wave, amp = get_data(num_points)
```

Create a recurrent neural network with two layers:

```
    # Create a recurrent neural network with 2 layers
    nn = nl.net.newelm([[-2, 2]], [10, 1], [nl.trans.TanSig(),
nl.trans.PureLin()])
```

Set the initializer functions for each layer:

```
    # Set the init functions for each layer
    nn.layers[0].initf = nl.init.InitRand([-0.1, 0.1], 'wb')
    nn.layers[1].initf = nl.init.InitRand([-0.1, 0.1], 'wb')
    nn.init()
```

Train the neural network:

```
# Train the recurrent neural network
error_progress = nn.train(wave, amp, epochs=1200, show=100, goal=0.01)
```

Run the data through the network:

```
# Run the training data through the network
output = nn.sim(wave)
```

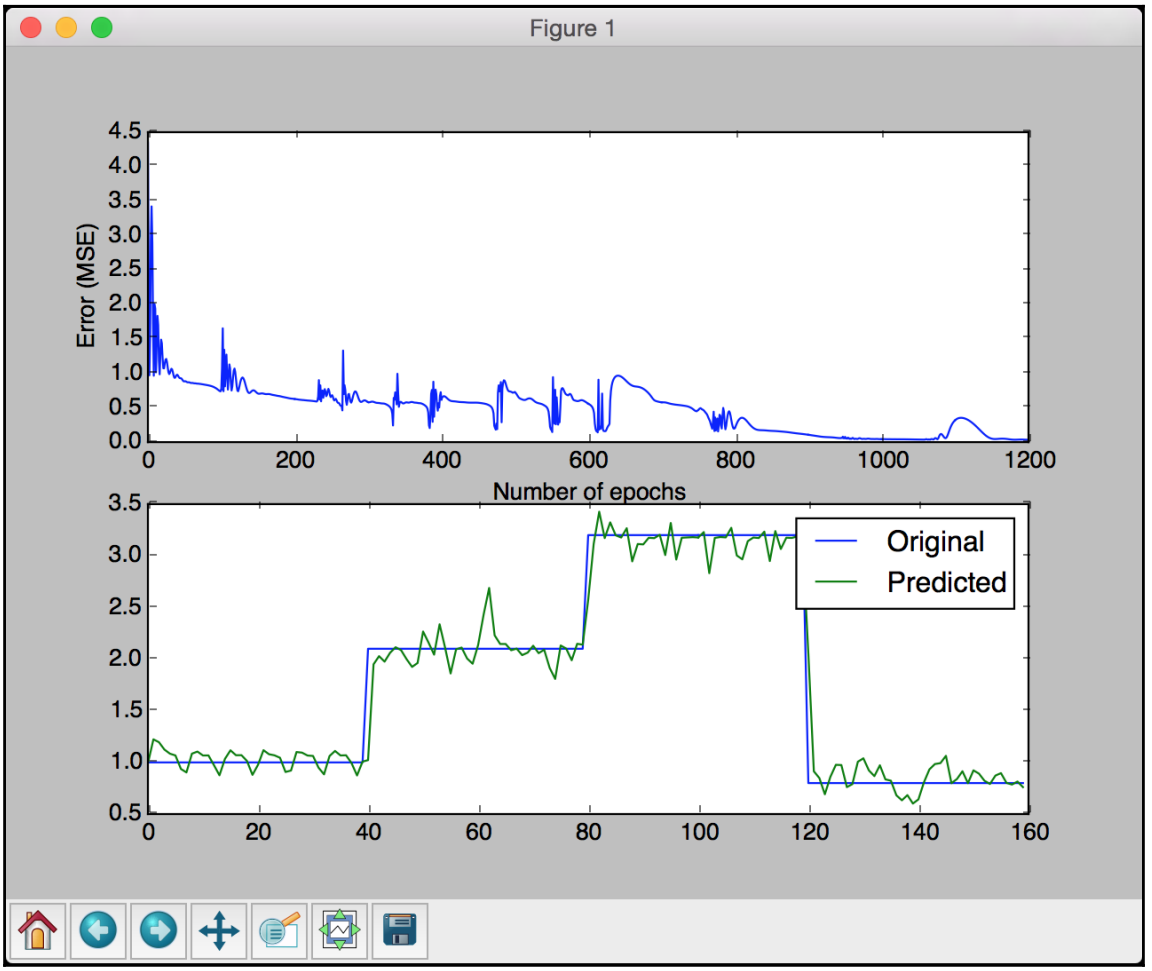
Plot the output:

```
# Plot the results
plt.subplot(211)
plt.plot(error_progress)
plt.xlabel('Number of epochs')
plt.ylabel('Error (MSE)')

plt.subplot(212)
plt.plot(amp.reshape(num_points * 4))
plt.plot(output.reshape(num_points * 4))
plt.legend(['Original', 'Predicted'])
```

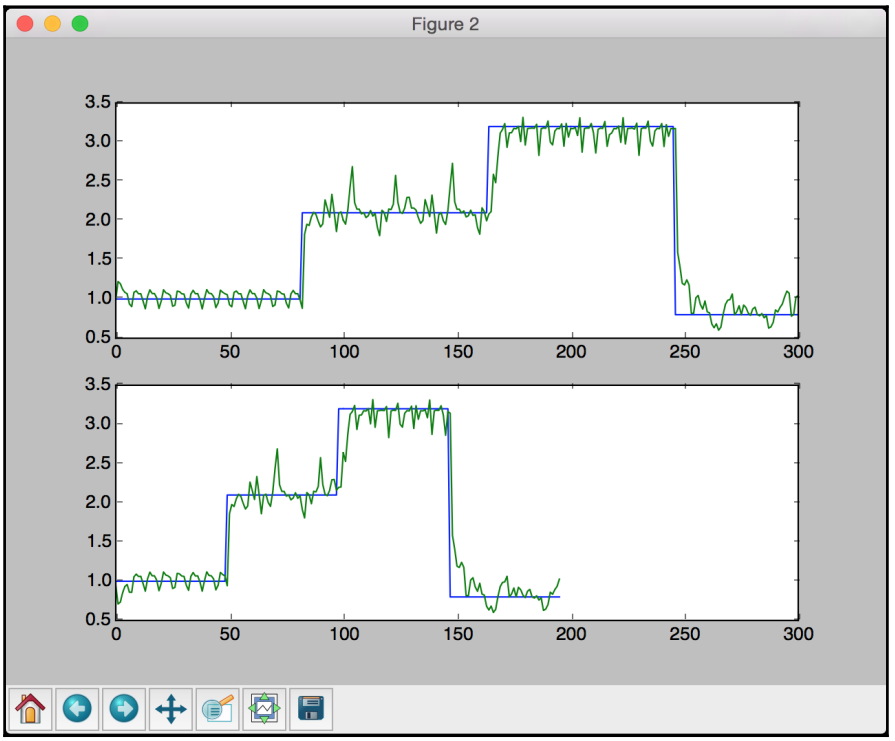
Test the performance of the neural network on unknown test data:

```
# Testing the network performance on unknown data
plt.figure()

plt.subplot(211)
visualize_output(nn, 82)
plt.xlim([0, 300])

plt.subplot(212)
visualize_output(nn, 49)
plt.xlim([0, 300])

plt.show()
```

The full code is given in the file `recurrent_neural_network.py`. If you run the code, you will see two output figures. The upper half of the first screenshot shows the training progress and the lower half shows the predicted output overlaid on top of the input waveform:



The upper half of the second screenshot shows how the neural network simulates the waveform even though we increase the length of the waveform. The lower half of the screenshot shows the same for decreased length.



You will see the following printed on your Terminal:

```
Epoch: 100; Error: 0.7378753203612153;
Epoch: 200; Error: 0.6276459886666788;
Epoch: 300; Error: 0.586316536629095;
Epoch: 400; Error: 0.7246461052491963;
Epoch: 500; Error: 0.7244266943409208;
Epoch: 600; Error: 0.5650581389122635;
Epoch: 700; Error: 0.5798180931911314;
Epoch: 800; Error: 0.19557566610789826;
Epoch: 900; Error: 0.10837074465396046;
Epoch: 1000; Error: 0.04330852391940663;
Epoch: 1100; Error: 0.3073835343028226;
Epoch: 1200; Error: 0.034685278416163604;
The maximum number of train epochs is reached
```

Visualizing characters in an Optical Character Recognition database

Artificial neural networks can use optical character recognition. It is perhaps one of the most commonly sited examples. **Optical Character Recognition (OCR)** is the process of recognizing handwritten characters in images. Before we jump into building that model, we need to familiarize ourselves with the dataset. We will be using the dataset available at `http://ai.stanford.edu/~btaskar/ocr`. You will be downloading a file called `letter.data`. For convenience, this file has been provided to you in the code bundle. Let's see how to load that data and visualize the characters.

Create a new python file and import the following packages:

```
import os
import sys

import cv2
import numpy as np
```

Define the input file containing the OCR data:

```
# Define the input file
input_file = 'letter.data'
```

Define the visualization and other parameters required to load the data from that file:

```
# Define the visualization parameters
img_resize_factor = 12
start = 6
end = -1
height, width = 16, 8
```

Iterate through the lines of that file until the user presses the Esc key. Each line in that file is tab separated. Read each line and scale it up to 255:

```
# Iterate until the user presses the Esc key
with open(input_file, 'r') as f:
    for line in f.readlines():
        # Read the data
        data = np.array([255 * float(x) for x in
line.split('\t')[start:end]])
```

Reshape the 1D array into a 2D image:

```
        # Reshape the data into a 2D image
        img = np.reshape(data, (height, width))
```

Scale the image for visualization:

```
            # Scale the image
            img_scaled = cv2.resize(img, None, fx=img_resize_factor,
    fy=img_resize_factor)
```

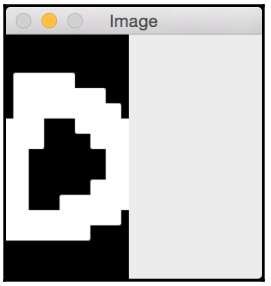
Display the image:

```
            # Display the image
            cv2.imshow('Image', img_scaled)
```

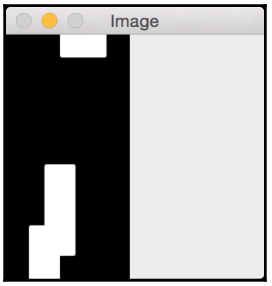
Check if the user has pressed the Esc key. If so, exit the loop:

```
            # Check if the user pressed the Esc key
            c = cv2.waitKey()
            if c == 27:
                break
```

The full code is given in the file `character_visualizer.py`. If you run the code, you will get an output screenshot displaying a character. You can keep pressing the space bar to see more characters. An `o` looks like this:



An `i` looks like this:



Building an Optical Character Recognition engine

Now that we have learned how to work with this data, let's build an optical character recognition system using artificial neural networks.

Create a new python file and import the following packages:

```
import numpy as np
import neurolab as nl
```

Define the input file:

```
# Define the input file
input_file = 'letter.data'
```

Define the number of datapoints that will be loaded:

```
# Define the number of datapoints to
# be loaded from the input file
num_datapoints = 50
```

Define the string containing all the distinct characters:

```
# String containing all the distinct characters
orig_labels = 'omandig'
```

Extract the number of distinct classes:

```
# Compute the number of distinct characters
num_orig_labels = len(orig_labels)
```

Define the train and test split. We will use 90% for training and 10% for testing:

```
# Define the training and testing parameters
num_train = int(0.9 * num_datapoints)
num_test = num_datapoints - num_train
```

Define the dataset extraction parameters:

```
# Define the dataset extraction parameters
start = 6
end = -1
```

Create the dataset:

```
# Creating the dataset
data = []
labels = []
with open(input_file, 'r') as f:
    for line in f.readlines():
        # Split the current line tabwise
        list_vals = line.split('\t')
```

If the label is not in our list of labels, we should skip it:

```
        # Check if the label is in our ground truth
        # labels. If not, we should skip it.
        if list_vals[1] not in orig_labels:
            continue
```

Extract the current label and append it to the main list:

```
        # Extract the current label and append it
        # to the main list
        label = np.zeros((num_orig_labels, 1))
        label[orig_labels.index(list_vals[1])] = 1
        labels.append(label)
```

Extract the character vector and append it to the main list:

```
        # Extract the character vector and append it to the main list
        cur_char = np.array([float(x) for x in list_vals[start:end]])
        data.append(cur_char)
```

Exit the loop once we have created the dataset:

```
        # Exit the loop once the required dataset has been created
        if len(data) >= num_datapoints:
            break
```

Convert the lists into numpy arrays:

```
# Convert the data and labels to numpy arrays
data = np.asfarray(data)
labels = np.array(labels).reshape(num_datapoints, num_orig_labels)
```

Extract the number of dimensions:

```
# Extract the number of dimensions
num_dims = len(data[0])
```

Create a feedforward neural network and set the training algorithm to gradient descent:

```
# Create a feedforward neural network
nn = nl.net.newff([[0, 1] for _ in range(len(data[0]))],
        [128, 16, num_orig_labels])

# Set the training algorithm to gradient descent
nn.trainf = nl.train.train_gd
```

Train the neural network:

```
# Train the network
error_progress = nn.train(data[:num_train,:], labels[:num_train,:],
        epochs=10000, show=100, goal=0.01)
```

Predict the output for test data:

```
# Predict the output for test inputs
print('\nTesting on unknown data:')
predicted_test = nn.sim(data[num_train:, :])
for i in range(num_test):
    print('\nOriginal:', orig_labels[np.argmax(labels[i])])
    print('Predicted:', orig_labels[np.argmax(predicted_test[i])])
```

The full code is given in the file `ocr.py`. If you run the code, you will see the following on your Terminal:

```
Epoch: 100; Error: 80.75182001223291;
Epoch: 200; Error: 49.823887961230206;
Epoch: 300; Error: 26.624261963923217;
Epoch: 400; Error: 31.131906412329677;
Epoch: 500; Error: 30.589610928772494;
Epoch: 600; Error: 23.129959531324324;
Epoch: 700; Error: 15.561849160600984;
Epoch: 800; Error: 9.52433563455828;
Epoch: 900; Error: 1.4032941634688987;
Epoch: 1000; Error: 1.1584148924740179;
Epoch: 1100; Error: 0.844934060039839;
Epoch: 1200; Error: 0.646187646028962;
Epoch: 1300; Error: 0.48881681329304894;
Epoch: 1400; Error: 0.4005475591737743;
Epoch: 1500; Error: 0.34145887283532067;
Epoch: 1600; Error: 0.29871068426249625;
Epoch: 1700; Error: 0.2657577763744411;
Epoch: 1800; Error: 0.23921810237252988;
Epoch: 1900; Error: 0.2172060084455509;
Epoch: 2000; Error: 0.19856823374761018;
Epoch: 2100; Error: 0.18253521958793384;
Epoch: 2200; Error: 0.16855895648078095;
```

It will keep going until 10,000 epochs. Once it's done, you will see the following on your Terminal:

```
Epoch: 9500; Error: 0.032460181065798295;
Epoch: 9600; Error: 0.027044816600106478;
Epoch: 9700; Error: 0.022026328910164213;
Epoch: 9800; Error: 0.018353324233938713;
Epoch: 9900; Error: 0.01578969259136868;
Epoch: 10000; Error: 0.014064205770213847;
The maximum number of train epochs is reached

Testing on unknown data:

Original: o
Predicted: o

Original: m
Predicted: n

Original: m
Predicted: m

Original: a
Predicted: d

Original: n
Predicted: n
```

As we can see in the preceding screenshot, it gets three of them right. If you use a bigger dataset and train longer, then you will get higher accuracy.

Summary

In this chapter, we learned about artificial neural networks. We discussed how to build and train neural networks. We talked about perceptrons and built a classifier based on that. We learned about single layer neural networks as well as multilayer neural networks. We discussed how neural networks could be used to build a vector quantizer. We analyzed sequential data using recurrent neural networks. We then built an optical character recognition engine using artificial neural networks. In the next chapter, we will learn about reinforcement learning and see how to build smart learning agents.

15
Reinforcement Learning

In this chapter, we are going to learn about reinforcement learning. We will discuss the premise of reinforcement learning. We will talk about the differences between reinforcement learning and supervised learning. We will go through some real world examples of reinforcement learning and see how it manifests itself in various forms. We will learn about the building blocks of reinforcement learning and the various concepts involved. We will then create an environment in python to see how it works in practice. We will then use these concepts to build a learning agent.

By the end of this chapter, you will know:

- Understanding the premise
- Reinforcement learning vs. supervised learning
- Real world examples of reinforcement learning
- Building blocks of reinforcement learning
- Creating an environment
- Building a learning agent

Understanding the premise

The concept of learning is fundamental to Artificial Intelligence. We want the machines to understand the process of learning so that they can do it on their own. Humans learn by observing and interacting with their surroundings. When you go to a new place, you quickly scan and see what's happening around you. Nobody is teaching you what to do here. You are observing and interacting with the environment around you. By building this connection with the environment, we tend to gather a lot of information about what's causing different things. We learn about cause and effect, what actions lead to what results, and what we need to do in order to achieve something.

We use this premise everywhere in our lives. We gather all this knowledge about our surroundings and, in turn, learn how we respond to that. Let's consider another example of an orator. Whenever good orators are giving speeches in public, they are aware of how the crowd is reacting to what they are saying. If the crowd is not responding to it, then the orator changes the speech in real time to ensure that the crowd is engaged. As we can see, the orator is trying to influence the environment through his/her behavior. We can say that the orator *learned* from interaction with the crowd in order to take action to achieve a certain *goal.* This is one of the most fundamental ideas in Artificial Intelligence on which many topics are based. Let's talk about reinforcement learning by keeping this in mind.

Reinforcement learning refers to the process of learning what to do and mapping situations to certain actions in order to maximize the reward. In most paradigms of machine learning, a learning agent is told what actions to take in order to achieve certain results. In the case of reinforcement leaning, the learning agent is not told what actions to take. Instead, it must discover what actions yield the highest reward by trying them out. These actions tend to affect the immediate reward as well as the next situation. This means that all the subsequent rewards will be affected too.

A good way to think about reinforcement learning is by understanding that we are defining a learning problem and not a learning method. So we can say that any method that can solve our problem can be considered as a reinforcement learning method. Reinforcement learning is characterized by two distinguishing features — trial and error learning, and delayed reward. A reinforcement learning agent uses these two features to learn from the consequences of its actions.

Reinforcement learning versus supervised learning

A lot of current research is focused on supervised learning. Reinforcement learning might seem a bit similar to supervised learning, but it is not. The process of supervised learning refers to learning from labeled samples provided by us. While this is a very useful technique, it is not sufficient to start learning from interactions. When we want to design a machine to navigate unknown terrains, this kind of learning is not going to help us. We don't have training samples available beforehand. We need an agent that can learn from its own experience by interacting with the unknown terrain. This is where reinforcement learning really shines.

Let's consider the exploration part where the agent has to interact with the new environment in order to learn. How much can it possibly explore? We do not even know how big the environment is, and in most cases, it is not possible to explore all the possibilities. So what should the agent do? Should it learn from its limited experience or wait until it explores further before taking action? This is one of the main challenges of reinforcement learning. In order to get a higher reward, an agent must favor the actions that have been tried and tested. But in order to discover such actions, it has to keep trying newer actions that have not been selected before. Researchers have studied this trade off between exploration and exploitation extensively over the years and it's still an active topic.

Real world examples of reinforcement learning

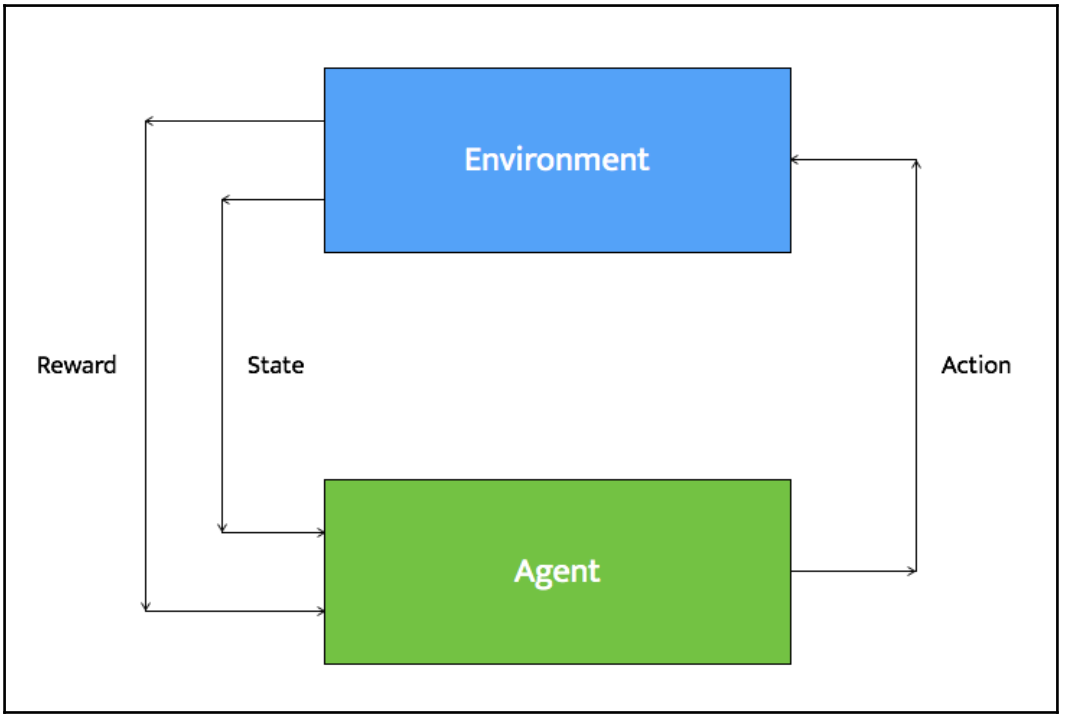
Let's see where reinforcement learning occurs in the real world. This will help us understand how it works and what possible applications can be built using this concept:

- **Game playing**: Let's consider a board game like **Go** or **Chess.** In order to determine the best move, the players need to think about various factors. The number of possibilities is so large that it is not possible to perform a brute-force search. If we were to build a machine to play such a game using traditional techniques, we need to specify a large number of rules to cover all these possibilities. Reinforcement learning completely bypasses this problem. We do not need to manually specify any rules. The learning agent simply learns by actually playing the game.
- **Robotics**: Let's consider a robot whose job is to explore a new building. It has to make sure it has enough power left to come back to the base station. This robot has to decide if it should make decisions by considering the trade off between the amount of information collected and the ability to reach back to base station safely.
- **Industrial controllers:** Consider the case of scheduling elevators. A good scheduler will spend the least amount of power and service the highest number of people. For problems like these, reinforcement learning agents can learn how to do this in a simulated environment. They can then take that knowledge to come up with optimal scheduling.
- **Babies:** Newborns struggle to walk in the first few months. They learn by trying it over and over again until they learn how to balance.

If you observe these examples closely, you will see there are some common traits. All of them involve interacting with the environment. The learning agent aims to achieve a certain goal even though there's uncertainty about the environment. The actions of an agent will change the future state of that environment. This impacts the opportunities available at later times as the agent continues to interact with the environment.

Building blocks of reinforcement learning

Now that we have seen a few examples, let's dig into the building blocks of a reinforcement learning system. Apart from the interaction between the agent and the environment, there are other factors at play here:



A typical reinforcement learning agent goes through the following steps:

- There is a set of states related to the agent and the environment. At a given point of time, the agent observes an input state to sense the environment.
- There are policies that govern what action needs to be taken. These policies act as decision making functions. The action is determined based on the input state using these policies.
- The agent takes the action based on the previous step.
- The environment reacts in a particular way in response to that action. The agent receives reinforcement, also known as reward, from the environment.
- The agent records the information about this reward. It's important to note that this reward is for this particular pair of state and action.

Reinforcement learning systems can do multiple things simultaneously — learn by performing a trial and error search, learn the model of the environment it is in, and then use that model to plan the next steps.

Creating an environment

We will be using a package called `OpenAI Gym` to build reinforcement learning agents. You can learn more about it here: `https://gym.openai.com`. We can install it using pip by running the following command on the Terminal:

```
$ pip3 install gym
```

You can find various tips and tricks related to its installation here: `https://github.com/openai/gym#installation`. Now that you have installed it, let's go ahead and write some code.

Create a new python file and import the following package:

```
import argparse

import gym
```

Define a function to parse the input arguments. We will be able to specify the type of environment we want to run:

```
def build_arg_parser():
    parser = argparse.ArgumentParser(description='Run an environment')
    parser.add_argument('--input-env', dest='input_env', required=True,
            choices=['cartpole', 'mountaincar', 'pendulum', 'taxi',
'lake'],
            help='Specify the name of the environment')
    return parser
```

Define the main function and parse the input arguments:

```
if __name__=='__main__':
    args = build_arg_parser().parse_args()
    input_env = args.input_env
```

Create a mapping from input argument string to the names of the environments as specified in the `OpenAI Gym` package:

```
    name_map = {'cartpole': 'CartPole-v0',
                'mountaincar': 'MountainCar-v0',
                'pendulum': 'Pendulum-v0',
                'taxi': 'Taxi-v1',
                'lake': 'FrozenLake-v0'}
```

Create the environment based on the input argument and reset it:

```
    # Create the environment and reset it
    env = gym.make(name_map[input_env])
    env.reset()
```

Iterate 1000 times and take action during each step:

```
    # Iterate 1000 times
    for _ in range(1000):
        # Render the environment
        env.render()

        # take a random action
        env.step(env.action_space.sample())
```

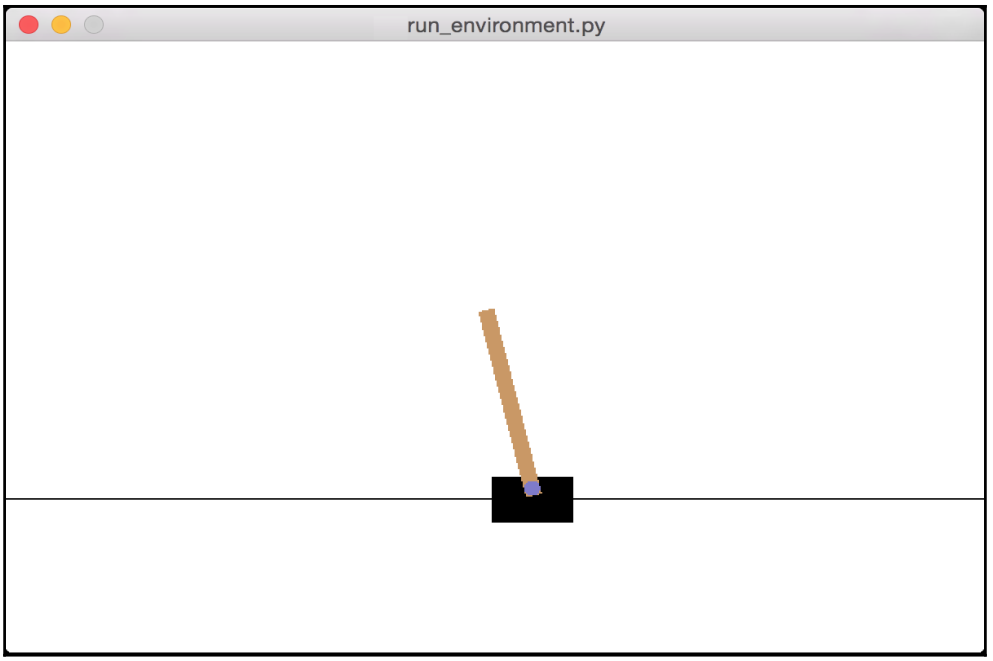
The full code is given in the file `run_environment.py`. If you want to know how to run the code, run it with the help argument as shown in the following figure:

```
$ python3 run_environment.py --help
usage: run_environment.py [-h] --input-env
                          {cartpole,mountaincar,pendulum,taxi,lake}

Run an environment

optional arguments:
  -h, --help            show this help message and exit
  --input-env {cartpole,mountaincar,pendulum,taxi,lake}
                        Specify the name of the environment
```

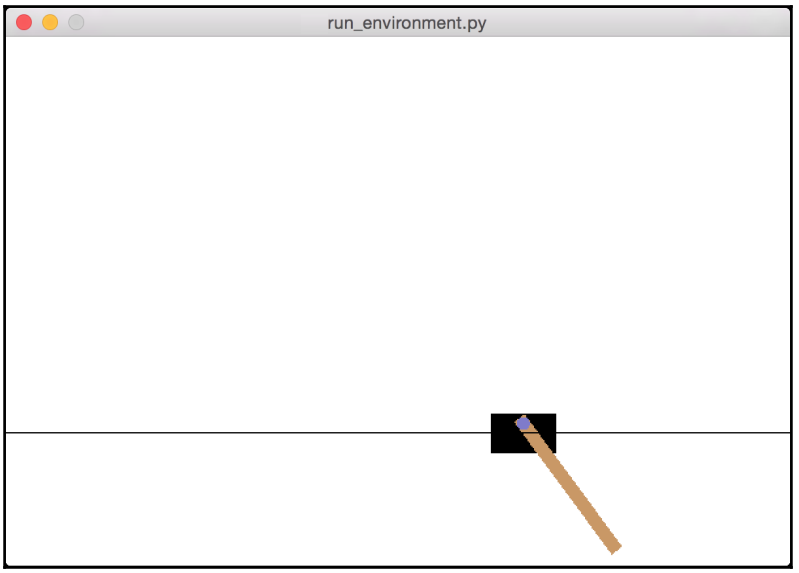
Let's run it with the cartpole environment. Run the following command on your Terminal:

```
$ python3 run_environment.py --input-env cartpole
```

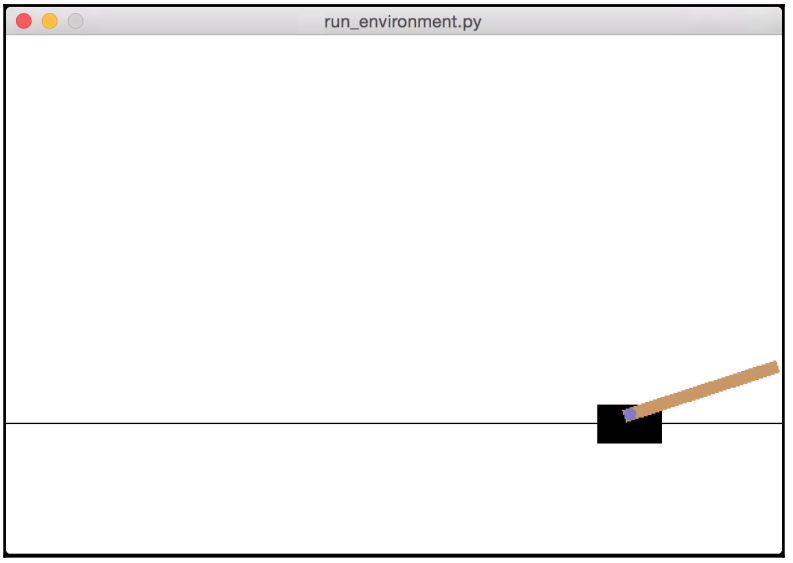
If you run it, you will see a window showing a **cartpole** moving to your right. The following screenshot shows the initial position:



In the next second or so, you will see it moving as shown in the following screenshot:



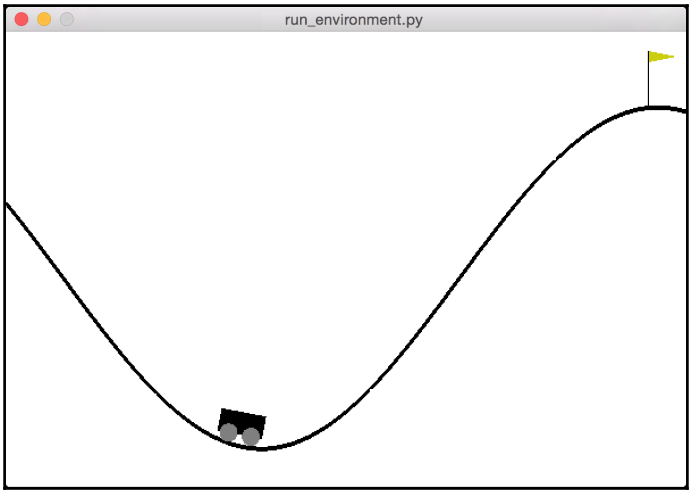
Towards the end, you will see it going out of the window as shown in the following screenshot:



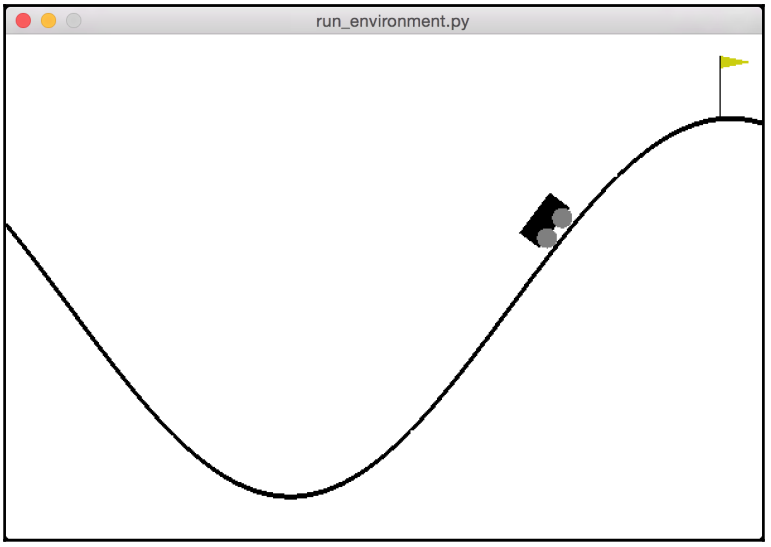
Let's run it with the mountain car argument. Run the following command on your Terminal:

```
$ python3 run_environment.py --input-env mountaincar
```

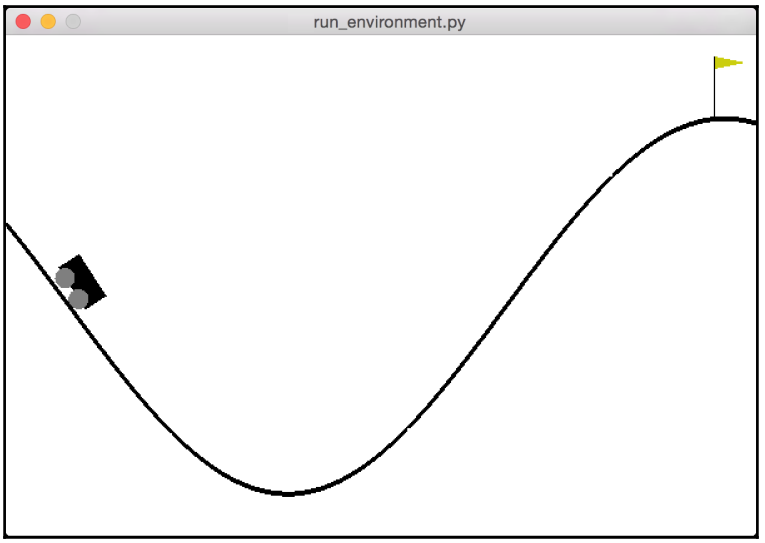
If you run the code, you will see the following figure initially:



If you let it run for a few seconds, you will see that the car oscillates more in order to reach the flag:



It will keep taking longer strides as shown in the following figure:



Building a learning agent

Let's see how to build a learning agent that can achieve a goal. The learning agent will learn how to achieve a goal. Create a new python file and import the following package:

```
import argparse

import gym
```

Define a function to parse the input arguments:

```
def build_arg_parser():
    parser = argparse.ArgumentParser(description='Run an environment')
    parser.add_argument('--input-env', dest='input_env', required=True,
            choices=['cartpole', 'mountaincar', 'pendulum'],
            help='Specify the name of the environment')
    return parser
```

Parse the input arguments:

```
if __name__=='__main__':
    args = build_arg_parser().parse_args()
    input_env = args.input_env
```

Build a mapping from the input arguments to the names of the environments in the OpenAI Gym package:

```
name_map = {'cartpole': 'CartPole-v0',
            'mountaincar': 'MountainCar-v0',
            'pendulum': 'Pendulum-v0'}
```

Create the environment based on the input argument:

```
# Create the environment
env = gym.make(name_map[input_env])
```

Start iterating by resetting the environment:

```
# Start iterating
for _ in range(20):
    # Reset the environment
    observation = env.reset()
```

For each reset, iterate 100 times. Start by rendering the environment:

```
    # Iterate 100 times
    for i in range(100):
        # Render the environment
        env.render()
```

Print the current observation and take action based on the available action space:

```
        # Print the current observation
        print(observation)

        # Take action
        action = env.action_space.sample()
```

Extract the consequences of taking the current action:

```
        # Extract the observation, reward, status and
        # other info based on the action taken
        observation, reward, done, info = env.step(action)
```

Check if we have achieved our goal:

```
        # Check if it's done
        if done:
            print('Episode finished after {} timesteps'.format(i+1))
            break
```

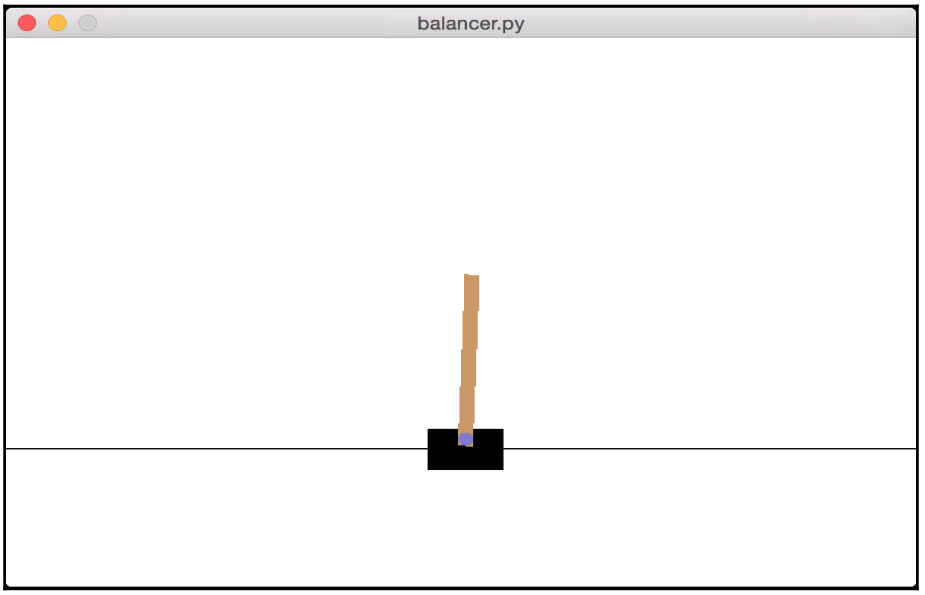
The full code is given in the file `balancer.py`. If you want to know how to run the code, run it with the help argument as shown in the following screenshot:

```
$ python3 balancer.py --help
usage: balancer.py [-h] --input-env {cartpole,mountaincar,pendulum}

Run an environment

optional arguments:
  -h, --help            show this help message and exit
  --input-env {cartpole,mountaincar,pendulum}
                        Specify the name of the environment
```

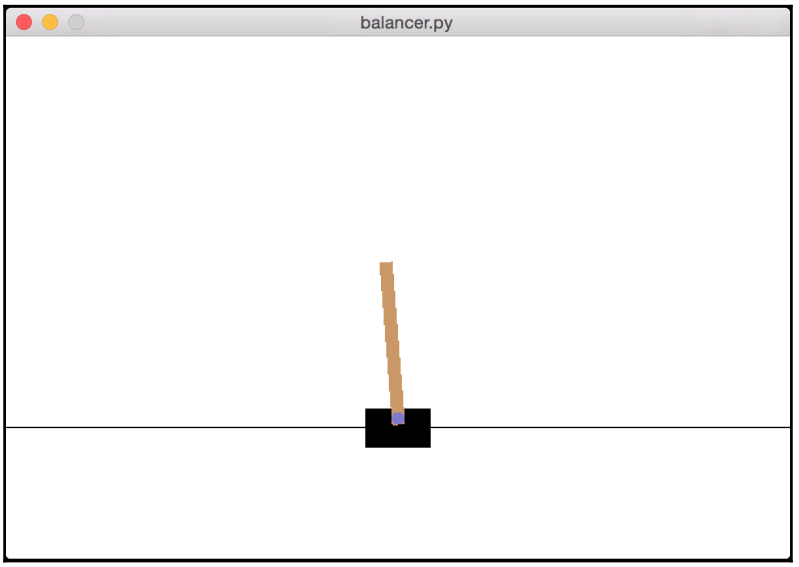
Let's run the code with the `cartpole` environment. Run the following command on your Terminal:

```
$ python3 balancer.py --input-env cartpole
```

If you run the code, you will see that the cartpole balances itself:



If you let it run for a few seconds, you will see that the cartpole is still standing as shown in the following screenshot:



You will see a lot of information printed on your Terminal. If you look at one of the episodes, it will look something like this:

```
[ 0.01704777  0.03379922 -0.01628054  0.02868271]
[ 0.01772375 -0.16108552 -0.01570689  0.31618481]
[ 0.01450204  0.03425659 -0.00938319  0.01859014]
[ 0.01518717 -0.16072954 -0.00901139  0.30829785]
[ 0.01197258 -0.35572194 -0.00284543  0.59812526]
[ 0.00485814 -0.16056029  0.00911707  0.30454742]
[ 0.00164694 -0.35581098  0.01520802  0.60009165]
[-0.00546928 -0.16090505  0.02720986  0.31223756]
[-0.00868738 -0.35640386  0.03345461  0.61337594]
[-0.01581546 -0.55197696  0.04572213  0.91640525]
[-0.026855   -0.3575021   0.06405023  0.63843544]
[-0.03400504 -0.16332896  0.07681894  0.36659087]
[-0.03727162 -0.3594537   0.08415076  0.68247294]
[-0.04446069 -0.5556372   0.09780022  1.00041801]
[-0.05557344 -0.75192055  0.11780858  1.32214352]
[-0.07061185 -0.55846765  0.14425145  1.06853119]
[-0.0817812  -0.36551752  0.16562207  0.82437502]
[-0.08909155 -0.56247052  0.18210957  1.16423244]
[-0.10034096 -0.75943464  0.20539422  1.50803784]
Episode finished after 19 timesteps
```

Different episodes take a different number of steps to finish. If you scroll through the information printed on your Terminal, you will be able to see that.

Summary

In this chapter, we learnt about reinforcement learning systems. We discussed the premise of reinforcement learning and how we can set it up. We talked about the differences between reinforcement learning and supervised learning. We went through some real world examples of reinforcement learning and saw how various systems use it in different forms.

We discussed the building blocks of reinforcement learning and concepts such as agent, environment, policy, reward, and so on. We then created an environment in python to see it in action. We used these concepts to build a reinforcement learning agent.

16
Deep Learning with Convolutional Neural Networks

In this chapter, we are going to learn about Deep Learning and **Convolutional Neural Networks** (**CNNs**). CNNs have gained a lot of momentum over the last few years, especially in the field of image recognition. We will talk about the architecture of CNNs and the type of layers used inside. We are going to see how to use a package called `TensorFlow`. We will build a perceptron based linear regressor. We are going to learn how to build an image classifier using a single layer neural network. We will then build an image classifier using a CNN.

By the end of this chapter, you will know:

- What are Convolutional Neural Networks (CNNs)?
- The architecture of CNNs
- The types of layers in a CNN
- Building a perceptron based linear regressor
- Building an image classifier using a single layer neural network
- Building an image classifier using a Convolutional Neural Network

What are Convolutional Neural Networks?

We saw how neural networks work in the last two chapters. Neural networks consist of neurons that have weights and biases. These weights and biases are tuned during the training process to come up with a good learning model. Each neuron receives a set of inputs, processes it in some way, and then outputs a value.

If we build a neural network with many layers, it's called a deep neural network. The branch of Artificial Intelligence dealing with these deep neural networks is referred to as deep learning.

One of the main disadvantages of ordinary neural networks is that they ignore the structure of input data. All data is converted to a single dimensional array before feeding it into the network. This works well for regular data, but things get difficult when we deal with images.

Let's consider grayscale images. These images are 2D structures and we know that the spatial arrangement of pixels has a lot of hidden information. If we ignore this information, we will be losing a lot of underlying patterns. This is where **Convolutional Neural Networks (CNNs)** come into the picture. CNNs take the 2D structure of the images into account when they process them.

CNNs are also made up of neurons consisting of weights and biases. These neurons accept input data, process it, and then output something. The goal of the network is to go from the raw image data in the input layer to the correct class in the output layer. The difference between ordinary neural networks and CNNs is in the type of layers we use and how we treat the input data. CNNs assume that the inputs are images, which allows them to extract properties specific to images. This makes CNNs way more efficient in dealing with images. Let's see how CNNs are built.

Architecture of CNNs

When we are working with ordinary neural networks, we need to convert the input data into a single vector. This vector acts as the input to the neural network, which then passes through the layers of the neural network. In these layers, each neuron is connected to all the neurons in the previous layer. It is also worth noting that the neurons within each layer are not connected to each other. They are only connected to the neurons in the adjacent layers. The last layer in the network is the output layer and it represents the final output.

If we use this structure for images, it will quickly become unmanageable. For example, let's consider an image dataset consisting of *256×256* RGB images. Since these are 3 channel images, there would be *256 * 256 * 3 = 196,608* weights. Note that this is just for a single neuron! Each layer will have multiple neurons, so the number of weights tends to increase rapidly. This means that the model will now have an enormous number of parameters to tune during the training process. This is why it becomes very complex and time-consuming. Connecting each neuron to every neuron in the previous layer, called full connectivity, is clearly not going to work for us.

CNNs explicitly consider the structure of images when processing the data. The neurons in CNNs are arranged in 3 dimensions — width, height, and depth. Each neuron in the current layer is connected to a small patch of the output from the previous layer. It's like overlaying an *NxN* filter on the input image. This is in contrast to a fully connected layer where each neuron is connected to all the neurons of the previous layer.

Since a single filter cannot capture all the nuances of the image, we do this *M* number of times to make sure we capture all the details. These *M* filters act as feature extractors. If you look at the outputs of these filters, we can see that they extract features like edges, corners, and so on. This is true for the initial layers in the CNN. As we progress through layers of the network, we will see that the later layers extract higher level features.

Types of layers in a CNN

Now that we know about the architecture of a CNN, let's see what type of layers are used to construct it. CNNs typically use the following types of layers:

- **Input layer:** This layer takes the raw image data as it is.
- **Convolutional layer:** This layer computes the convolutions between the neurons and the various patches in the input. If you need a quick refresher on image convolutions, you can check out this link: `http://web.pdx.edu/~jduh/courses/Archive/geog481w07/Students/Ludwig_ImageConvolution.pdf`. The convolutional layer basically computes the dot product between the weights and a small patch in the output of the previous layer.
- **Rectified Linear Unit layer:** This layer applies an activation function to the output of the previous layer. This function is usually something like *max(0, x)*. This layer is needed to add non-linearity to the network so that it can generalize well to any type of function.
- **Pooling layer:** This layer samples the output of the previous layer resulting in a structure with smaller dimensions. Pooling helps us to keep only the prominent parts as we progress in the network. Max pooling is frequently used in the pooling layer where we pick the maximum value in a given *KxK* window.
- **Fully Connected layer:** This layer computes the output scores in the last layer. The resulting output is of the size *1x1xL*, where *L* is the number of classes in the training dataset.

As we go from the input layer to the output layer in the network, the input image gets transformed from pixel values to the final class scores. Many different architectures for CNNs have been proposed and it's an active area of research. The accuracy and robustness of a model depends on many factors — the type of layers, depth of the network, the arrangement of various types of layers within the network, the functions chosen for each layer, training data, and so on.

Building a perceptron-based linear regressor

We will see how to build a linear regression model using perceptrons. We have already seen linear regression in previous chapters, but this section is about building a linear regression model using a neural network approach.

We will be using `TensorFlow` in this chapter. It is a popular deep learning package that's widely used to build various real world systems. In this section, we will get familiar with how it works. Make sure to install it before you proceed. The installation instructions are given here: `https://www.tensorflow.org/get_started/os_setup`. Once you verify that it's installed, create a new python and import the following packages:

```
import numpy as np
import matplotlib.pyplot as plt
import tensorflow as tf
```

We will be generating some datapoints and see how we can fit a model to it. Define the number of datapoints to be generated:

```
# Define the number of points to generate
num_points = 1200
```

Define the parameters that will be used to generate the data. We will be using the model of a line: $y = mx + c$:

```
# Generate the data based on equation y = mx + c
data = []
m = 0.2
c = 0.5
for i in range(num_points):
    # Generate 'x'
    x = np.random.normal(0.0, 0.8)
```

Generate some noise to add some variation in the data:

```
# Generate some noise
noise = np.random.normal(0.0, 0.04)
```

Compute the value of `y` using the equation:

```
# Compute 'y'
y = m*x + c + noise

data.append([x, y])
```

Once you finish iterating, separate the data into input and output variables:

```
# Separate x and y
x_data = [d[0] for d in data]
y_data = [d[1] for d in data]
```

Plot the data:

```
# Plot the generated data
plt.plot(x_data, y_data, 'ro')
plt.title('Input data')
plt.show()
```

Generate weights and biases for the perceptron. For weights, we will use a uniform random number generator and set the biases to zero:

```
# Generate weights and biases
W = tf.Variable(tf.random_uniform([1], -1.0, 1.0))
b = tf.Variable(tf.zeros([1]))
```

Define the equation using `TensorFlow` variables:

```
# Define equation for 'y'
y = W * x_data + b
```

Define the loss function that can be used during the training process. The optimizer will try to minimize this value as much as possible.

```
# Define how to compute the loss
loss = tf.reduce_mean(tf.square(y - y_data))
```

Define the gradient descent optimizer and specify the loss function:

```
# Define the gradient descent optimizer
optimizer = tf.train.GradientDescentOptimizer(0.5)
train = optimizer.minimize(loss)
```

All the variables are in place, but they haven't been initialized yet. Let's do that:

```
# Initialize all the variables
init = tf.initialize_all_variables()
```

Start the TensorFlow session and run it using the initializer:

```
# Start the tensorflow session and run it
sess = tf.Session()
sess.run(init)
```

Start the training process:

```
# Start iterating
num_iterations = 10
for step in range(num_iterations):
    # Run the session
    sess.run(train)
```

Print the progress of the training process. The `loss` parameter will continue to decrease as we go through iterations:

```
    # Print the progress
    print('\nITERATION', step+1)
    print('W =', sess.run(W)[0])
    print('b =', sess.run(b)[0])
    print('loss =', sess.run(loss))
```

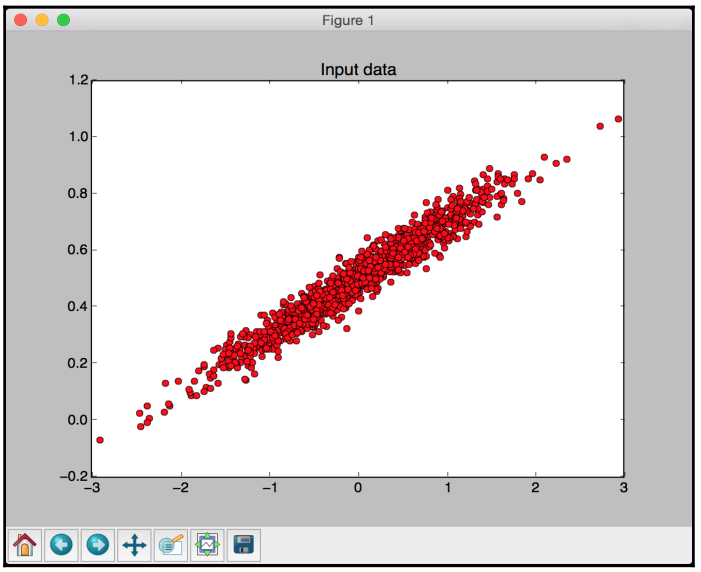
Plot the generated data and overlay the predicted model on top. In this case, the model is a line:

```
    # Plot the input data
    plt.plot(x_data, y_data, 'ro')

    # Plot the predicted output line
    plt.plot(x_data, sess.run(W) * x_data + sess.run(b))
```

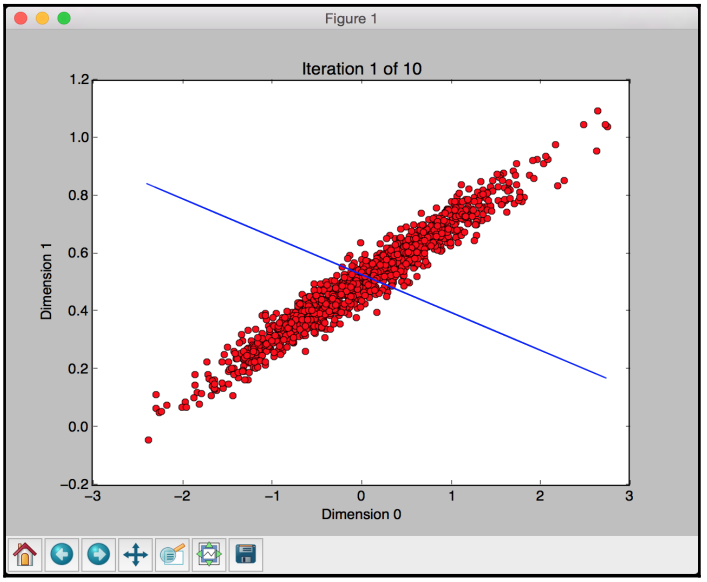
Set the parameters for the plot:

```
    # Set plotting parameters
    plt.xlabel('Dimension 0')
    plt.ylabel('Dimension 1')
    plt.title('Iteration ' + str(step+1) + ' of ' + str(num_iterations))
    plt.show()
```

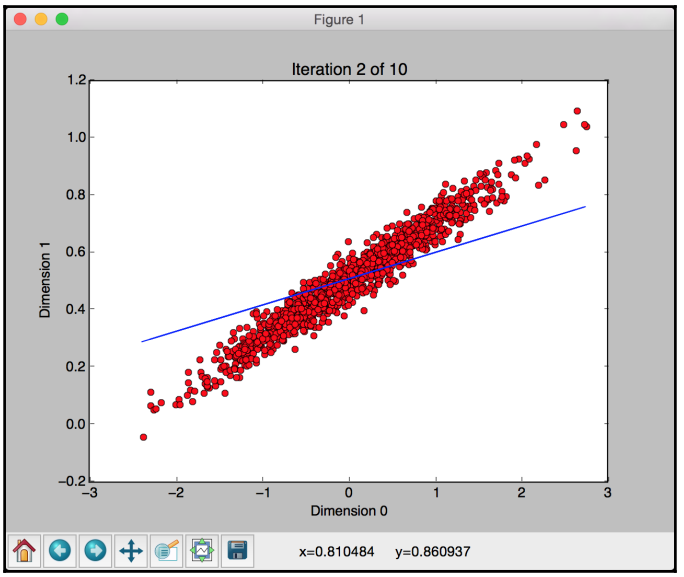
The full code is given in the file `linear_regression.py`. If you run the code, you will see following screenshot showing input data:



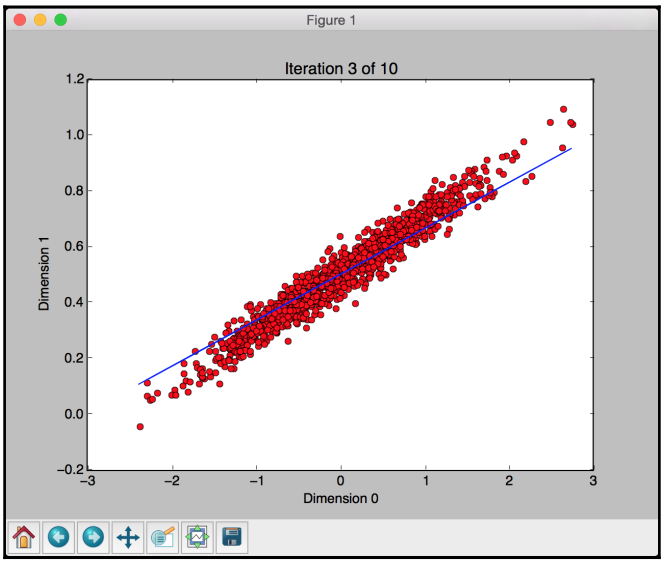
If close you this window, you will see the training process. The first iteration looks like this:



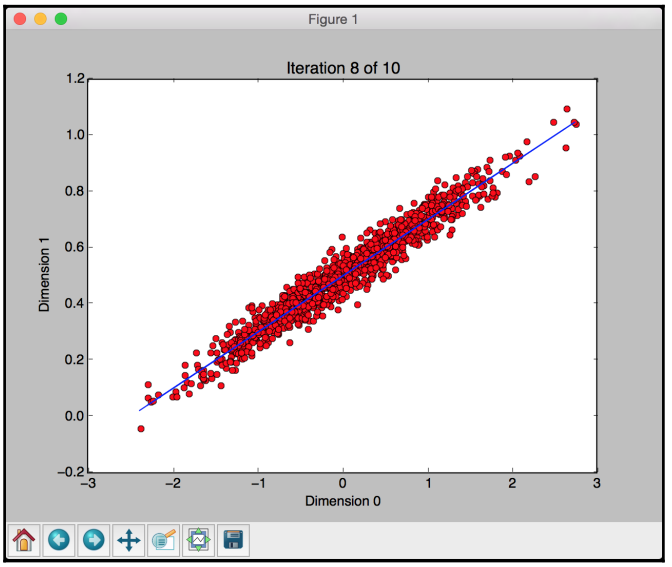
As we can see, the line is completely off. Close this window to go to the next iteration:



The line seems better, but it's still off. Let's close this window and continue iterating:



It looks like the line is getting closer to the real model. If you continue iterating like this, the model gets better. The eighth iteration looks like this:



The line seems to fit the data pretty well. You will see the following printed on your Terminal in the beginning:

```
ITERATION 1
W = -0.130961
b = 0.53005
loss = 0.0760343

ITERATION 2
W = 0.0917911
b = 0.508959
loss = 0.00960302

ITERATION 3
W = 0.164665
b = 0.502555
loss = 0.00250165

ITERATION 4
W = 0.188492
b = 0.500459
loss = 0.0017425
```

Once it finishes training, you will see the following on your Terminal:

```
ITERATION 7
W = 0.199662
b = 0.499477
loss = 0.00165175

ITERATION 8
W = 0.199934
b = 0.499453
loss = 0.00165165

ITERATION 9
W = 0.200023
b = 0.499445
loss = 0.00165164

ITERATION 10
W = 0.200052
b = 0.499443
loss = 0.00165164
```

Building an image classifier using a single layer neural network

Let's see how to create a single layer neural network using `TensorFlow` and use it to build an image classifier. We will be using MNIST image dataset to build our system. It is dataset containing handwritten images of digits. Our goal is to build a classifier that can correctly identify the digit in each image.

Create a new python and import the following packages:

```
import argparse

import tensorflow as tf
from tensorflow.examples.tutorials.mnist import input_data
```

Define a function to parse the input arguments:

```
def build_arg_parser():
    parser = argparse.ArgumentParser(description='Build a classifier using
             \MNIST data')
    parser.add_argument('--input-dir', dest='input_dir', type=str,
             default='./mnist_data', help='Directory for storing data')
    return parser
```

Define the main function and parse the input arguments:

```
if __name__ == '__main__':
    args = build_arg_parser().parse_args()
```

Extract the MNIST image data. The `one_hot` flag specifies that we will be using one-hot encoding in our labels. It means that if we have `n` classes, then the label for a given datapoint will be an array of length `n`. Each element in this array corresponds to a particular class. To specify a class, the value at the corresponding index will be set to *1* and everything else will be *0*:

```
    # Get the MNIST data
    mnist = input_data.read_data_sets(args.input_dir, one_hot=True)
```

The images in the database are *28 x 28*. We need to convert it to a single dimensional array to create the input layer:

```
    # The images are 28x28, so create the input layer
    # with 784 neurons (28x28=784)
    x = tf.placeholder(tf.float32, [None, 784])
```

Create a single layer neural network with weights and biases. There are 10 distinct digits in the database. The number of neurons in the input layer is *784* and the number of neurons in the output layer is *10*:

```
    # Create a layer with weights and biases. There are 10 distinct
    # digits, so the output layer should have 10 classes
    W = tf.Variable(tf.zeros([784, 10]))
    b = tf.Variable(tf.zeros([10]))
```

Create the equation to be used for training:

```
    # Create the equation for 'y' using y = W*x + b
    y = tf.matmul(x, W) + b
```

Define the loss function and the gradient descent optimizer:

```
    # Define the entropy loss and the gradient descent optimizer
    y_loss = tf.placeholder(tf.float32, [None, 10])
    loss = tf.reduce_mean(tf.nn.softmax_cross_entropy_with_logits(y,
y_loss))
    optimizer = tf.train.GradientDescentOptimizer(0.5).minimize(loss)
```

Initialize all the variables:

```
    # Initialize all the variables
    init = tf.initialize_all_variables()
```

Create a `TensorFlow` session and run it:

```
# Create a session
session = tf.Session()
session.run(init)
```

Start the training process. We will train using batches where we run the optimizer on the current batch and then continue with the next batch for the next iteration. The first step in each iteration is to get the next batch of images to train on:

```
# Start training
num_iterations = 1200
batch_size = 90
for _ in range(num_iterations):
    # Get the next batch of images
    x_batch, y_batch = mnist.train.next_batch(batch_size)
```

Run the optimizer on this batch of images:

```
    # Train on this batch of images
    session.run(optimizer, feed_dict = {x: x_batch, y_loss: y_batch})
```

Once the training process is over, compute the accuracy using the test dataset:

```
# Compute the accuracy using test data
predicted = tf.equal(tf.argmax(y, 1), tf.argmax(y_loss, 1))
accuracy = tf.reduce_mean(tf.cast(predicted, tf.float32))
print('\nAccuracy =', session.run(accuracy, feed_dict = {
        x: mnist.test.images,
        y_loss: mnist.test.labels}))
```

The full code is given in the file `single_layer.py`. If you run the code, it will download the data to a folder called `mnist_data` in the current folder. This is the default option. If you want to change it, you can do so using the input argument. Once you run the code, you will get the following output on your Terminal:

```
Extracting ./mnist_data/train-images-idx3-ubyte.gz
Extracting ./mnist_data/train-labels-idx1-ubyte.gz
Extracting ./mnist_data/t10k-images-idx3-ubyte.gz
Extracting ./mnist_data/t10k-labels-idx1-ubyte.gz

Accuracy = 0.921
```

As printed on your Terminal, the accuracy of the model is 92.1%.

Building an image classifier using a Convolutional Neural Network

The image classifier in the previous section didn't perform well. Getting *92.1%* on MNIST dataset is relatively easy. Let's see how we can use Convolutional Neural Networks (CNNs) to achieve a much higher accuracy. We will build an image classifier using the same dataset, but with a CNN instead of a single layer neural network.

Create a new python and import the following packages:

```
import argparse

import tensorflow as tf
from tensorflow.examples.tutorials.mnist import input_data
```

Define a function to parse the input arguments:

```
def build_arg_parser():
    parser = argparse.ArgumentParser(description='Build a CNN classifier \
            using MNIST data')
    parser.add_argument('--input-dir', dest='input_dir', type=str,
            default='./mnist_data', help='Directory for storing data')
    return parser
```

Define a function to create values for weights in each layer:

```
def get_weights(shape):
    data = tf.truncated_normal(shape, stddev=0.1)
    return tf.Variable(data)
```

Define a function to create values for biases in each layer:

```
def get_biases(shape):
    data = tf.constant(0.1, shape=shape)
    return tf.Variable(data)
```

Define a function to create a layer based on the input shape:

```
def create_layer(shape):
    # Get the weights and biases
    W = get_weights(shape)
    b = get_biases([shape[-1]])

    return W, b
```

Define a function to perform 2D-convolution:

```
def convolution_2d(x, W):
    return tf.nn.conv2d(x, W, strides=[1, 1, 1, 1],
            padding='SAME')
```

Define a function to perform a 2×2 max pooling operation:

```
def max_pooling(x):
    return tf.nn.max_pool(x, ksize=[1, 2, 2, 1],
            strides=[1, 2, 2, 1], padding='SAME')
```

Define the main function and parse the input arguments:

```
if __name__ == '__main__':
    args = build_arg_parser().parse_args()
```

Extract the *MNIST* image data:

```
    # Get the MNIST data
    mnist = input_data.read_data_sets(args.input_dir, one_hot=True)
```

Create the input layer with *784* neurons:

```
    # The images are 28x28, so create the input layer
    # with 784 neurons (28x28=784)
    x = tf.placeholder(tf.float32, [None, 784])
```

We will be using convolutional neural networks that take advantage of the 2D structure of images. So let's reshape `x` into a 4D tensor where the second and third dimensions specify the image dimensions:

```
    # Reshape 'x' into a 4D tensor
    x_image = tf.reshape(x, [-1, 28, 28, 1])
```

Create the first convolutional layer that will extract *32* features for each *5×5* patch in the image:

```
    # Define the first convolutional layer
    W_conv1, b_conv1 = create_layer([5, 5, 1, 32])
```

Convolve the image with weight tensor computed in the previous step, and then add the bias tensor to it. We then need to apply the **Rectified Linear Unit (ReLU)** function to the output:

```
    # Convolve the image with weight tensor, add the
    # bias, and then apply the ReLU function
    h_conv1 = tf.nn.relu(convolution_2d(x_image, W_conv1) + b_conv1)
```

Apply the 2×2 max pooling operator to the output of the previous step:

```
# Apply the max pooling operator
h_pool1 = max_pooling(h_conv1)
```

Create the second convolutional layer to compute 64 features for each 5×5 patch:

```
# Define the second convolutional layer
W_conv2, b_conv2 = create_layer([5, 5, 32, 64])
```

Convolve the output of the previous layer with weight tensor computed in the previous step, and then add the bias tensor to it. We then need to apply the **Rectified Linear Unit** (**ReLU**) function to the output:

```
# Convolve the output of previous layer with the
# weight tensor, add the bias, and then apply
# the ReLU function
h_conv2 = tf.nn.relu(convolution_2d(h_pool1, W_conv2) + b_conv2)
```

Apply the 2×2 max pooling operator to the output of the previous step:

```
# Apply the max pooling operator
h_pool2 = max_pooling(h_conv2)
```

The image size is now reduced to *7×7*. Create a fully connected layer with 1024 neurons.

```
# Define the fully connected layer
W_fc1, b_fc1 = create_layer([7 * 7 * 64, 1024])
```

Reshape the output of the previous layer:

```
# Reshape the output of the previous layer
h_pool2_flat = tf.reshape(h_pool2, [-1, 7*7*64])
```

Multiply the output of the previous layer with the weight tensor of the fully connected layer, and then add the bias tensor to it. We then apply the Rectified Linear Unit (ReLU) function to the output:

```
# Multiply the output of previous layer by the
# weight tensor, add the bias, and then apply
# the ReLU function
h_fc1 = tf.nn.relu(tf.matmul(h_pool2_flat, W_fc1) + b_fc1)
```

In order to reduce overfitting, we need to create a dropout layer. Let's create a `TensorFlow` placeholder for the probability values that specify the probability of a neuron's output being kept during dropout:

```
# Define the dropout layer using a probability placeholder
# for all the neurons
keep_prob = tf.placeholder(tf.float32)
h_fc1_drop = tf.nn.dropout(h_fc1, keep_prob)
```

Define the readout layer with 10 output neurons corresponding to 10 classes in our dataset. Compute the output:

```
# Define the readout layer (output layer)
W_fc2, b_fc2 = create_layer([1024, 10])
y_conv = tf.matmul(h_fc1_drop, W_fc2) + b_fc2
```

Define the loss function and optimizer function:

```
# Define the entropy loss and the optimizer
y_loss = tf.placeholder(tf.float32, [None, 10])
loss = tf.reduce_mean(tf.nn.softmax_cross_entropy_with_logits(y_conv,
y_loss))
optimizer = tf.train.AdamOptimizer(1e-4).minimize(loss)
```

Define how the accuracy should be computed:

```
# Define the accuracy computation
predicted = tf.equal(tf.argmax(y_conv, 1), tf.argmax(y_loss, 1))
accuracy = tf.reduce_mean(tf.cast(predicted, tf.float32))
```

Create and run a session after initializing the variables:

```
# Create and run a session
sess = tf.InteractiveSession()
init = tf.initialize_all_variables()
sess.run(init)
```

Start the training process:

```
# Start training
num_iterations = 21000
batch_size = 75
print('\nTraining the model....')
for i in range(num_iterations):
    # Get the next batch of images
    batch = mnist.train.next_batch(batch_size)
```

Print the accuracy progress every 50 iterations:

```
        # Print progress
        if i % 50 == 0:
            cur_accuracy = accuracy.eval(feed_dict = {
                    x: batch[0], y_loss: batch[1], keep_prob: 1.0})
            print('Iteration', i, ', Accuracy =', cur_accuracy)
```

Run the optimizer on the current batch:

```
        # Train on the current batch
        optimizer.run(feed_dict = {x: batch[0], y_loss: batch[1],
keep_prob: 0.5})
```

Once the training process is over, compute the accuracy using the test dataset:

```
    # Compute accuracy using test data
    print('Test accuracy =', accuracy.eval(feed_dict = {
            x: mnist.test.images, y_loss: mnist.test.labels,
            keep_prob: 1.0}))
```

The full code is given in the file `cnn.py`. If you run the code, you will get the following output on your Terminal:

```
Extracting ./mnist_data/train-images-idx3-ubyte.gz
Extracting ./mnist_data/train-labels-idx1-ubyte.gz
Extracting ./mnist_data/t10k-images-idx3-ubyte.gz
Extracting ./mnist_data/t10k-labels-idx1-ubyte.gz

Training the model....
Iteration 0 , Accuracy = 0.0533333
Iteration 50 , Accuracy = 0.813333
Iteration 100 , Accuracy = 0.8
Iteration 150 , Accuracy = 0.906667
Iteration 200 , Accuracy = 0.84
Iteration 250 , Accuracy = 0.92
Iteration 300 , Accuracy = 0.933333
Iteration 350 , Accuracy = 0.866667
Iteration 400 , Accuracy = 0.973333
Iteration 450 , Accuracy = 0.933333
Iteration 500 , Accuracy = 0.906667
Iteration 550 , Accuracy = 0.853333
Iteration 600 , Accuracy = 0.973333
Iteration 650 , Accuracy = 0.973333
Iteration 700 , Accuracy = 0.96
Iteration 750 , Accuracy = 0.933333
```

As you continue iterating, the accuracy keeps increasing as shown in the following screenshot:

```
Iteration 2900 , Accuracy = 0.973333
Iteration 2950 , Accuracy = 1.0
Iteration 3000 , Accuracy = 0.973333
Iteration 3050 , Accuracy = 1.0
Iteration 3100 , Accuracy = 0.986667
Iteration 3150 , Accuracy = 1.0
Iteration 3200 , Accuracy = 1.0
Iteration 3250 , Accuracy = 1.0
Iteration 3300 , Accuracy = 1.0
Iteration 3350 , Accuracy = 1.0
Iteration 3400 , Accuracy = 0.986667
Iteration 3450 , Accuracy = 0.946667
Iteration 3500 , Accuracy = 0.973333
Iteration 3550 , Accuracy = 0.973333
Iteration 3600 , Accuracy = 1.0
Iteration 3650 , Accuracy = 0.986667
Iteration 3700 , Accuracy = 1.0
Iteration 3750 , Accuracy = 1.0
Iteration 3800 , Accuracy = 0.986667
Iteration 3850 , Accuracy = 0.986667
Iteration 3900 , Accuracy = 1.0
```

Now that we have the output, we can see that the accuracy of a convolutional neural network is much higher than a simple neural network.

Summary

In this chapter, we learnt about Deep Learning and CNNs. We discussed what CNNs are and why we need them. We talked about the architecture of CNNs. We learnt about the various type of layers used within a CNN. We discussed how to use TensorFlow. We used it to build a perceptron-based linear regressor. We learnt how to build an image classifier using a single layer neural network. We then built an image classifier using a CNN.

Index

D

E

L

M

N

O

P

R

S

T

U

V

W