Week 10

Data science review

Agenda

- Probability
  - Joint probability
  - Conditional probability
- Machine learning
  - Supervised v.s. Unsupervised
- K-Means
- KNN
- Regression

Probability

Let S be a finite nonempty sample space of equally likely outcomes, and E is an event (i.e., a subset of S), then the *probability of* E *is* $p(E) = \frac{|E|}{|S|}$. *Obviously, we have* $0 \leq p(E) \leq 1$.

$$\text{The probability of E} = \frac{\text{The number of occurrence of E}}{\text{All possible outcomes}}$$

Joint probability

Let A and B be two events.

$$P(AB) = \frac{\text{The number of occurrence of A and B}}{\text{All possible outcomes}}$$

- If A and B are independent, then P(AB) = P(A)P(B).

Probability of the union of events

Let A and B be two events.

$$P(A \cup B) = P(A) + P(B) - P(AB)$$

Conditional probability

Let A and B be two events.

P(A|B): The probability of A happens when B happens.

- It is defined as

$$P(A|B) = \frac{P(AB)}{P(B)}$$

- Useful algebraic transforms:

$$P(AB) = P(A|B)P(B)$$

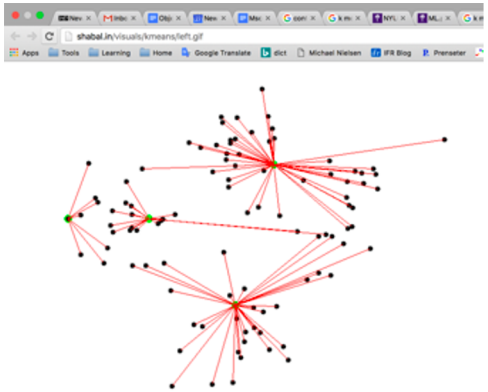
$$P(B|A) = \frac{P(A|B)P(B)}{P(A)}$$

K-means

Given a set of samples ($\mathbf{x}_1$, $\mathbf{x}_2$, …, $\mathbf{x}_n$), where each sample is a *d*-dimensional real vector, *k*-means clustering aims to partition the *n* samples into *k* (≤ *n*) clusters **C** = {C_1, C_2, …, C_k} so as to minimize the within-cluster sum of squares (WCSS) (i.e. variance).

Input: *k* (the number of clusters),
D (a set of lift ratios)
Output: a set of k clusters
Method:
Arbitrarily choose *k* objects from *D* as the initial cluster centers;
Repeat:
1. (re)assign each object to the cluster to which the object is the most similar, based on the mean value of the objects in the cluster;
2. Update the cluster means, i.e., calculate the mean value of the objects for each cluster

Until no change;



The code

```
15 class Sample(object):
16
17     def __init__(self, name, features, label=None):
18         #Assumes features is an array of numbers
19         self.name = name
20         self.features = features
21         self.label = label
22
23     def getDimensionality(self):
24         return len(self.features)
25
26     def getFeatures(self):
27         return self.features[:]
28
29     def getLabel(self):
30         return self.label
31
32     def setLabel(self, label):
33         self.label = label
34
35     def getName(self):
36         return self.name
37
38     def setName(self, name):
39         self.name = name
40
41     def distance(self, other):
42         def minkowskiDist(v1, v2, p=2):
43             """Assumes v1 and v2 are equal-length arrays of numbers
44                Returns Minkowski distance of order p between v1 and v2"""
45             dist = 0.0
46             for i in range(len(v1)):
47                 dist += abs(v1[i] - v2[i])**p
48             return dist**(1.0/p)
49         return minkowskiDist(self.features, other.getFeatures())
```

Magic methods

```
51     def __add__(self, other):
52         f = []
53         for i in range(self.getDimensionality()):
54             f.append(self.getFeatures()[i] + other.getFeatures()[i])
55         return Sample(self.name + '+' + other.name, f, self.label)
56
57     def __truediv__(self, n):
58         f = []
59         for e in self.getFeatures():
60             f.append(e/float(n))
61         return Sample(self.name + '/' + str(n), f)
62
63     def __sub__(self, other):
64         f = []
65         for i in range(self.getDimensionality()):
66             f.append(self.getFeatures()[i] - other.getFeatures()[i])
67         return Sample(self.name + '-' + other.name, f)
68
69     def __str__(self):
70         return self.name +': ' \
71             + str(self.features) + ': ' \
72             + str(self.label)
```

The magic methods (dunder) here, like, “__add__”, etc., can be considered as the overriding of the math operators. With such settings, the statement “a + b” means “a.__add__(b)” if “a” and “b” are instances of Sample class.

The code

```
class Cluster(object):

    def __init__(self, samples):
        """Assumes samples is a list"""
        self.samples = samples
        """ centrold is also an instance of Sample class """
        self.centroid = self.computeCentroid()

    def size(self):
        return len(self.samples)

    def getCentroid(self):
        return self.centroid

    def getMembers(self):
        return self.samples

    #### Implement the centroid computing function here!
    def computeCentroid(self):
        '''
        return an instance of Sample, its features should be
        the center of all the samples in the cluster
        '''
        dim = self.samples[0].dimensionality()
        centroid = sample.Sample('centroid', [0.0]*dim)
        for e in self.samples:
            centroid += e
            centroid /= len(self.samples)
        return centroid
```

Computing the centroid,
recall the Dunders we defined in Sample

Computing the new centroid,
Calculate distance of the new centroid and old centroid

```
#### Implement the centroid updating function here!
def update(self, samples):
    """Replace the samples in the cluster by new samples
       Return: how much the centroid has changed"""
    oldCentroid = self.centroid
    self.samples = samples
    if len(samples) > 0:
        self.centroid = self.computeCentroid()
        return oldCentroid.distance(self.centroid)
    else:
        return 0.0

    #return helper.update(self, samples)

def __str__(self):
    names = []
    for e in self.samples:
        names.append(e.getName())
    names.sort()
    result = 'Cluster with centroid '\
             + str(self.centroid.getFeatures()) + ' contains:\n  '
    for e in names:
        result = result + e + ', '
    return result[:-2]
```

The code

```
25# Kmeans: take a list of samples and make k clusters
26def kmeans(samples, k, verbose):
27    """Assumes samples is a list of samples of class Sample,
28         k is a positive int, verbose is a Boolean
29       Returns a list containing k clusters. """
30    #Get k randomly chosen initial centroids
31    initialCentroids = random.sample(samples, k)
32    #Create a singleton cluster for each centroid
33    clusters = []
34    for e in initialCentroids:
35        clusters.append(cluster.Cluster([e]))
36    #Iterate until centroids do not change
37    converged = False
38    numIterations = 0
39    while not converged:
40        numIterations += 1
41        # replace the following line by implementing
42        # kmeans_iter(samples, clusters, k) in this file
43        converged = kmeans_iter(samples, clusters, k)
44        #  converged = helper.kmeans_iter(samples, clusters, k)
45        if verbose:
46            print('Iteration #' + str(numIterations))
47            for c in clusters:
48                print(c)
49            print('\n') #add blank line
50    return clusters
```

List of all data points

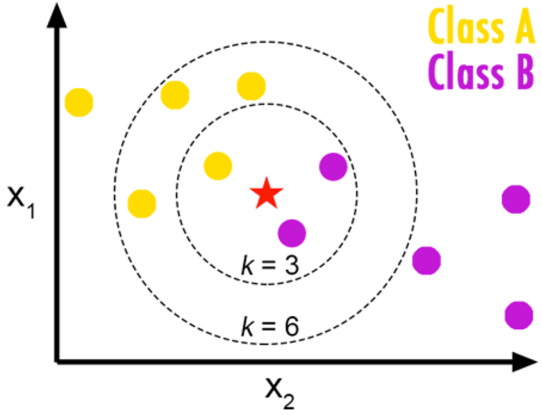
list of instances of Cluster

```
36def kmeans_iter(samples, clusters, k):
37    #Create a list containing k distinct empty lists
38    newClusters = []
39    for i in range(k):
40        newClusters.append([])
41    #Associate each sample with closest centroid
42    for e in samples:
43        #Find the centroid closest to e
44        smallestDistance = e.distance(clusters[0].getCentroid())
45        index = 0
46        for i in range(1, k):
47            distance = e.distance(clusters[i].getCentroid())
48            if distance < smallestDistance:
49                smallestDistance = distance
50                index = i
51        #Add e to the list of samples for the appropriate cluster
52        newClusters[index].append(e)
53    #Update each cluster; check if a centroid has changed
54    converged = True
55    for i in range(len(clusters)):
56        if clusters[i].update(newClusters[i]) > 0.0:
57            converged = False
58    return converged
```

Update the clusters

K-nn

The principle behind nearest neighbor methods is to find a predefined number (i.e., k) of training samples closest in distance to the new point, and predict the label from these.



k-Nearest Neighbor
Classify (**X**, **Y**, *x*) // **X**: training data, **Y**: class labels of **X**, *x*: unknown sample
for $i = 1$ **to** m **do**
  Compute distance $d(\mathbf{X}_i, x)$
end for
Compute set I containing indices for the k smallest distances $d(\mathbf{X}_i, x)$.
return majority label for $\{\mathbf{Y}_i$ where $i \in I\}$

The code

```
 7 import sample
 8 import random
 9 import util
10 #import knn_helper
11
12 def knn(p, data, k):
13     """ Compute the distance between p to every sample in data,
14         set p's label to the label of the maximum of samples
15         within the k nearest neighbors
16     """
17
18     """ Steps:
19         1. Iterate through samples in data and store the
20            distance from p in the dictionary "distance"; key is the
21            distance, value is the sample.
22         2. Creat a sorted list of samples according to ascending
23            order of the distances.
24         3. In the dictioary "label_votes", stores number of votes
25            in each label among the top-k nearest samples
26         4. Assign p the most popular label
27     """
28
29 #     max_label = util.LABELS[0]
30 #     p.setLabel(max_label)
31     # above forces a fixed label: remove them
32     # replace knn_helper.knn(p, data, k) with your own logic
33     # Calculating the distances b/w p & every pt. in data
```

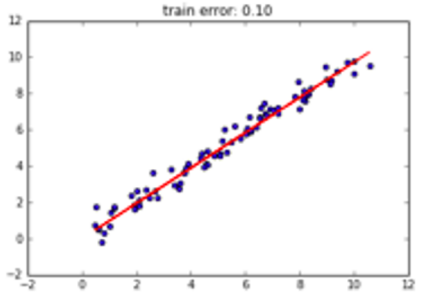
p is an instance of Sample; data is the labeled data and is a list whose elements are instances of Sample

```
34     distances = {}
35     for d in data:
36         if d.distance(p) not in distances.keys():
37             distances[ d.distance(p) ] = [ d ]
38         else:
39             distances[ d.distance(p) ].append(d)
40
41     # Sorting the k nearest neighbours
42     result = []
43     for key in sorted(distances.keys()):
44         result.extend( distances[key] )
45
46     k_nearest_neighbours = result[:k]
47
48     # Assigning a label to the new point based on the k neighbours
49     label_votes = { l:0 for l in util.LABELS }
50     for x in k_nearest_neighbours:
51         label_votes[ x.getLabel() ] += 1
52     max_label = sorted(label_votes, key = label_votes.get, reverse = True)[0]
53
54     p.setLabel(max_label)
```

voting

a dict comprehension

Regression

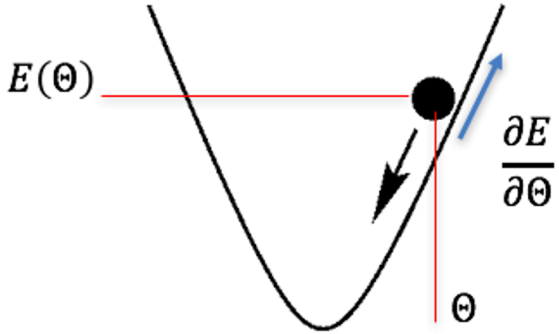


Minimizing the square of the total prediction error **E;**
Turned to be an optimization problem since **E** is a function of **w**.

- Model: $pred = wx$
- Model parameter: w
- $E(w) = \frac{1}{2n}\sum_{i=1}^{n}(wx_i - y_i)^2$

W can be obtained by gradient descent.

- $w = w - \lambda\frac{dE}{dW}$
- $\frac{dE}{dw} = \frac{1}{n}\sum_{i=1}^{n}(wx_i - y_i)x_i$
- $\frac{dE}{dw} = \frac{1}{n}\sum_{i}^{n}(pred_i - y_i)x_i$

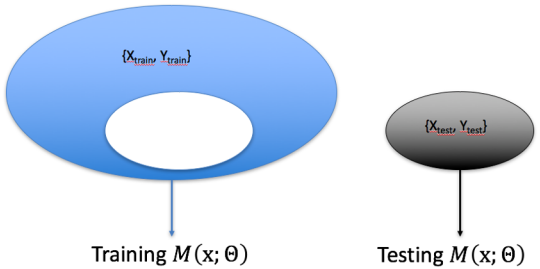


The code

```
60    # w is the parameter to be learned
61    w = 0.5
62
63    # training loop
64    while True:
65
66        # predict and compute error
67        pred_y = [w*x for x in d_x]
68        error = compute_error(pred_y, d_y)
69        error_history.append(error)
70
71        plt.scatter(d_x, d_y)
72        plt.plot(d_x, pred_y, 'r')
73        title_str = "train error: %0.2f" % error
74        plt.title(title_str)
75        plt.show()
76
77        steps += 1
78        if error <= margin or steps >= num_iter:
79            break
80
81        # gradient descent
82        grad = compute_grad(pred_y, d_y, d_x)
83        w -= learning_rate * grad
84
85        if VERBOSE and input("cont? (y/n) ") == 'n':
86            break
87
88    # testing on test data
89    d_x = [d.getFeatures()[0] for d in test_data]
90    d_y = [d.getFeatures()[1] for d in test_data]
91    pred_y = [w*x for x in d_x]
92    error = compute_error(pred_y, d_y)
```

```
 8 def compute_error(pred, truth):
 9     error = 0
10     n = len(pred)
11     for i in range(n):
12         error += (pred[i] - truth[i]) ** 2
13     error /= 2*n
14     return error
15
16 def compute_grad(pred, truth, x):
17     n = len(pred)
18     grad = sum([(pred[i] - truth[i]) * x[i] for i in range(n)])/n
19     return grad
```

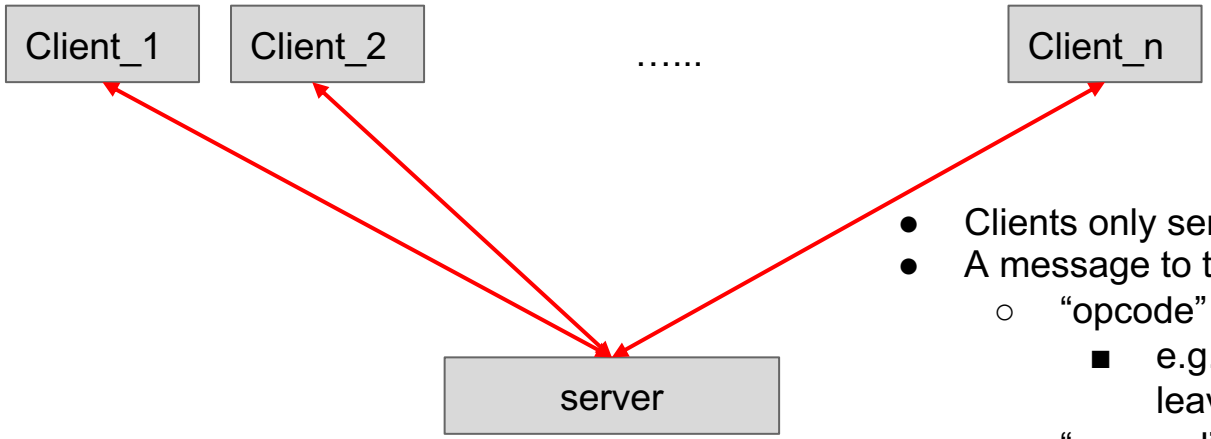
Training and testing



A model ***overfits*** if it performs well on training data, but poorly on testing data

UP3: An introduction

The framework of chatting



- Clients only send messages to server
- A message to the server should contain:
  - “opcode” (i.e., instructions)
    - e.g., to log in, to chat, or to leave, etc.
  - “operand” (i.e., arguments need for executing the corresponding opcode)
    - e.g., if the opcode is “to chat”, then, the message will contain the words that need to be sent.

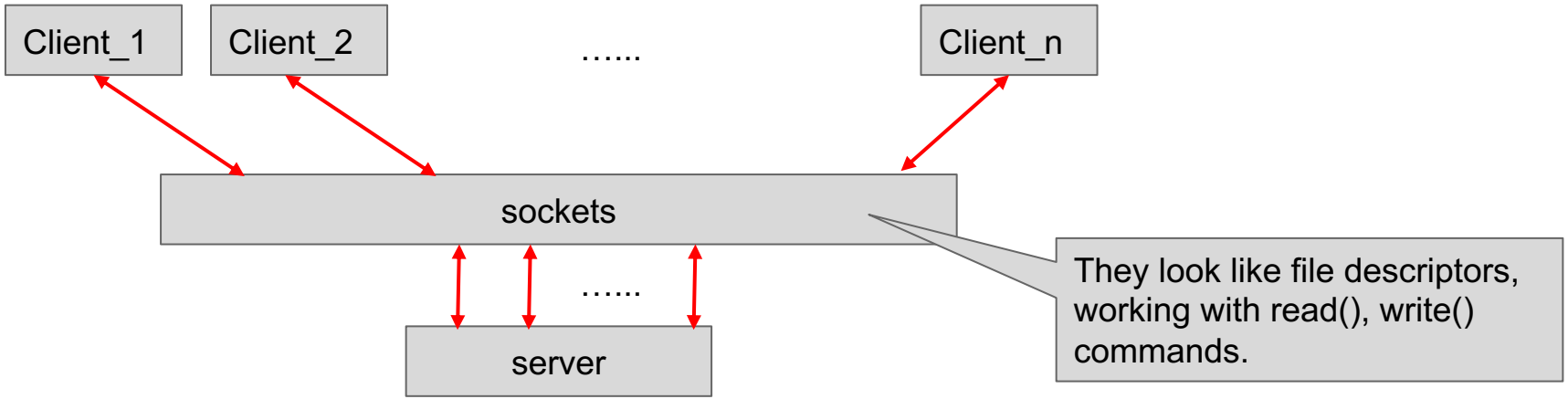
Two main tasks:

- Transfer the messages among clients and servers
- Manage clients and the chat

Transferring messages

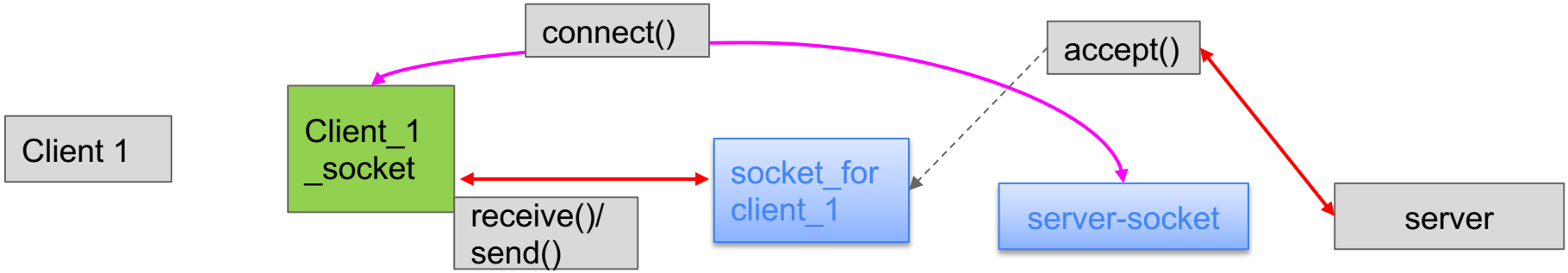
Most of the related codes are given; you don’t need to implement them. But knowing how they work is helpful as you need to use some of them in UP3.

socket: a package for transfer information

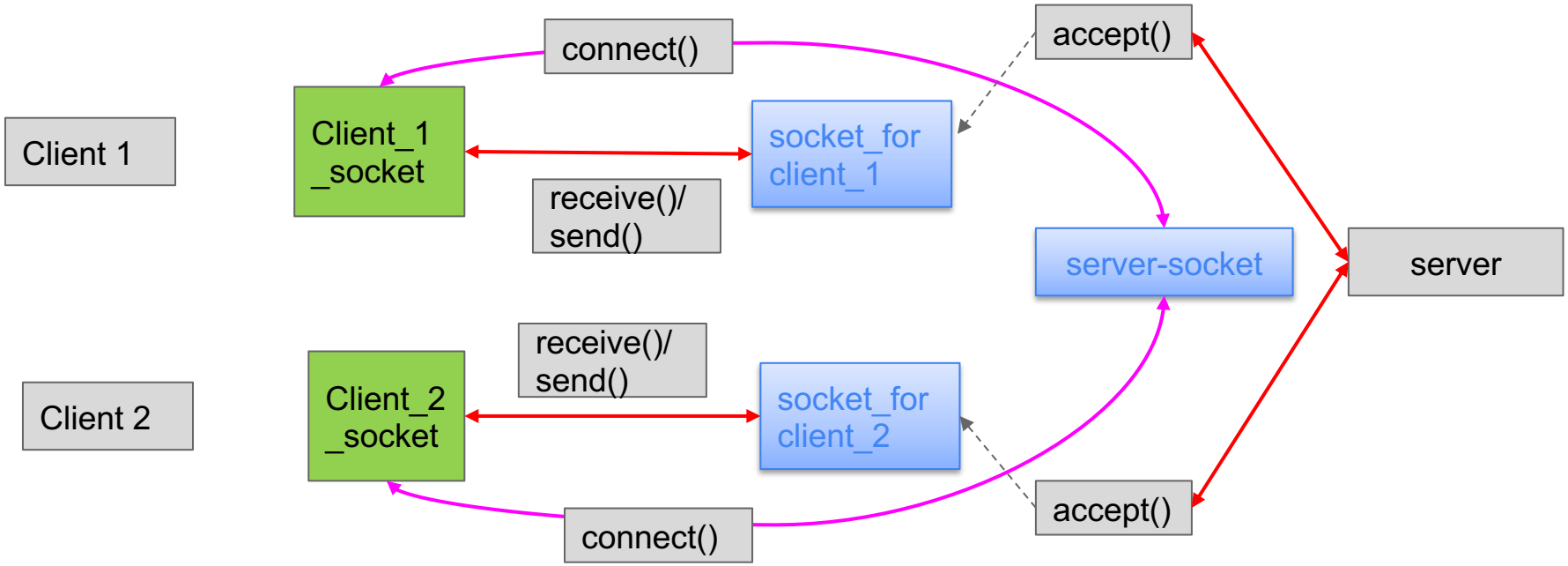


Sockets: allowing communication between two different processes on the same or different machines.

More detail about sockets



More detail about sockets

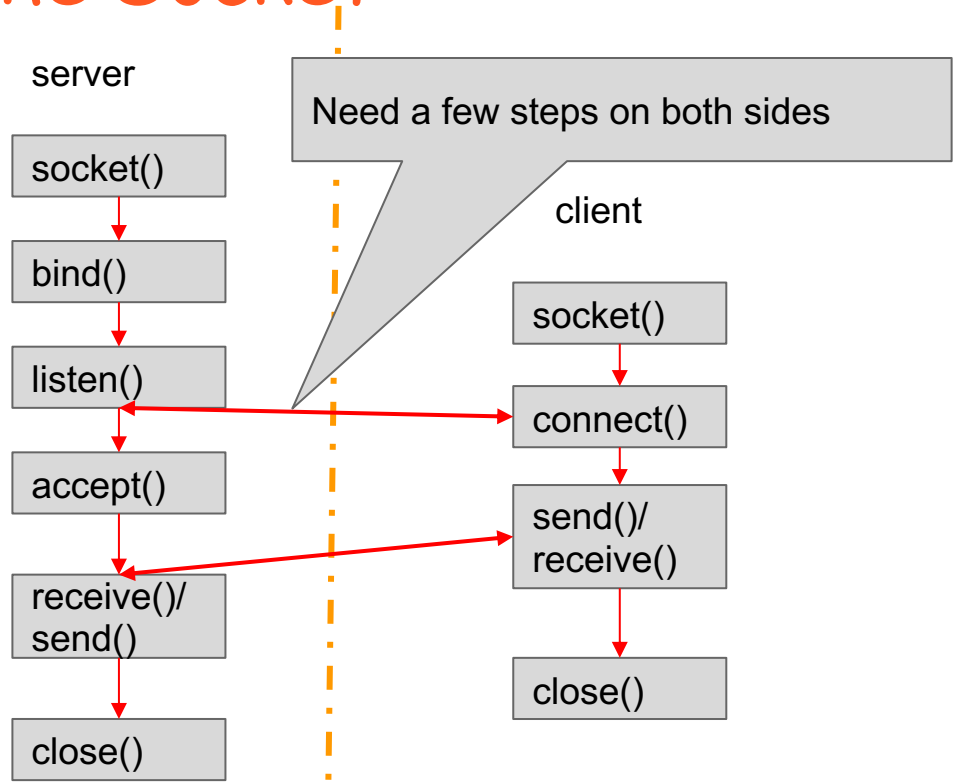


Communication via the Socket:

Operations on client side:

- Initialization
- setting up the socket
- connecting to the server
- login
- logout

and others.



An example

Implementing socket in chat_client_class.py

```
28      def init_chat(self):
29          self.socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM )
30          svr = SERVER if self.args.d == None else (self.args.d, CHAT_PORT)
31          self.socket.connect(svr)
32          self.sm = csm.ClientSM(self.socket)
33          reading_thread = threading.Thread(target=self.read_input)
34          reading_thread.daemon = True
35          reading_thread.start()
```

A socket is assigned to a client, then, the socket is connected to the server.

When sending/receiving messages, it needs to use the socket.
(Note: mysend() and myrecv() are defined in chat_utility.py)

```
def send(self, msg):
    mysend(self.socket, msg)

def recv(self):
    return myrecv(self.socket)
```

Implementing socket on server side (in `chat_server.py`)

```
19 class Server:
20     def __init__(self):
21         self.new_clients = []  # list of new sockets of which the user id is not known
22         self.logged_name2sock = {}  # dictionary mapping username to socket
23         self.logged_sock2name = {}  # dict mapping socket to user name
24         self.all_sockets = []
25         self.group = grp.Group()
26         # start server
27         self.server = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
28         self.server.bind(SERVER)
29         self.server.listen(5)
30         self.all_sockets.append(self.server)
31         # initialize past chat indices
32         self.indices = {}
33         # sonnet
34         self.sonnet = indexer.PIndex("AllSonnets.txt")
35
```

A client name is associated to a unique socket.

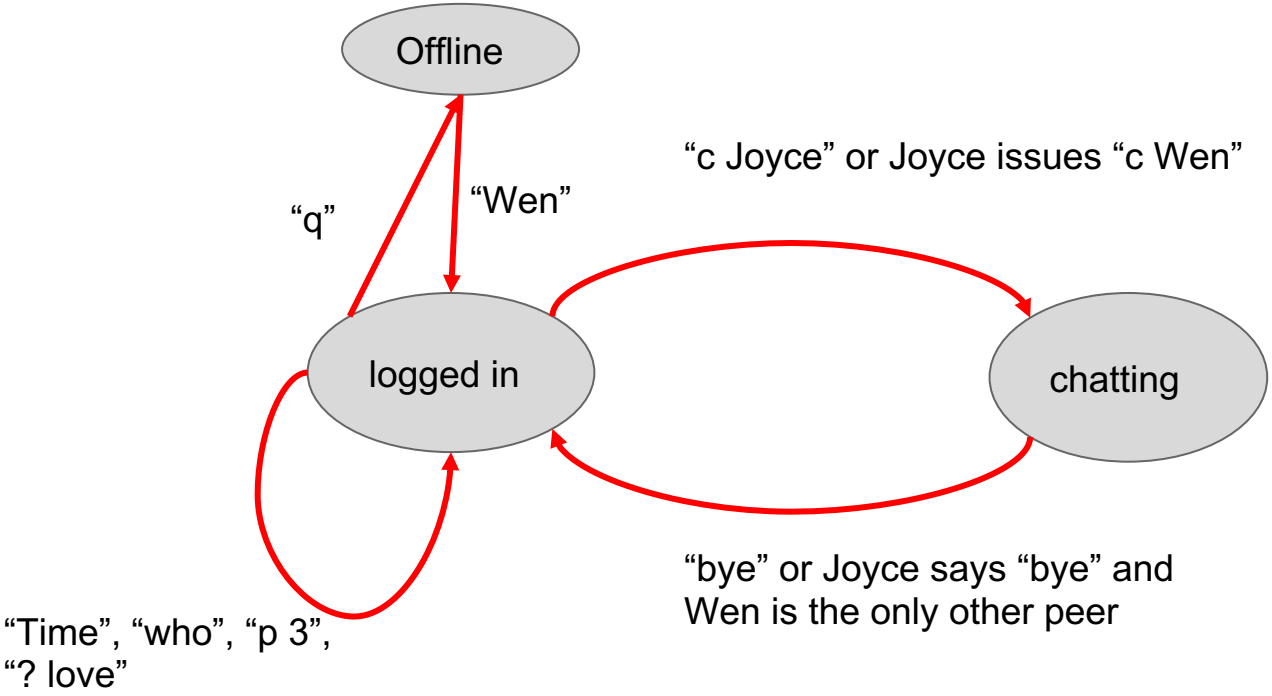
The socket of the server is binded to a IP and a Port.

Client side and chats management

You need to implement some parts in

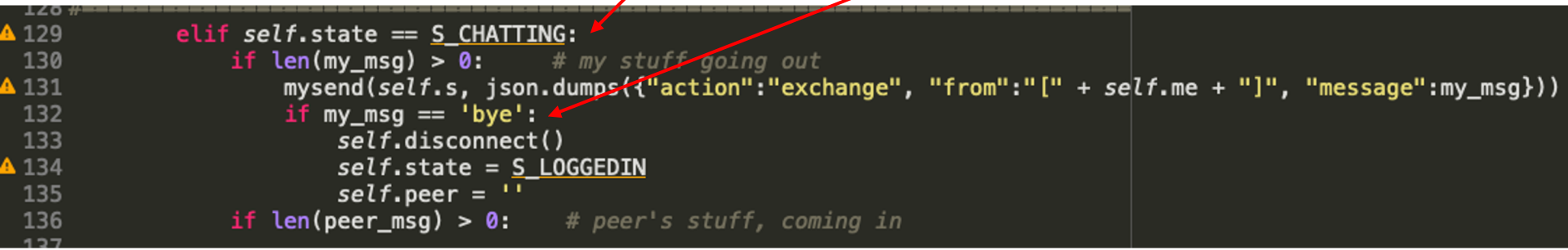
- client_state_machine.py
- chat_server.py.

Designing the steps: Protocol code



Finite state machine (FSM)

- In client side, there are three possible states: off-line, logged in, and chatting.
- The client side responses to the inputs based on its current state.
  - E.g., if the current state is chatting, then, the input “bye” will active the method disconnect().



Protocol code

A set of codes shared by client and server so that they can understand each other. A message between clients and server should follow the format that defined by the protocol

- It has to contain a key, “action”
  - There are seven actions including “connect”, “disconnect”, “poem”, etc.
- It may contain other keys, which depends on the type of action
  - e.g., if “action” is “connect”, then, the message should also has a key “target”, which specifies the peer that the client wants to connect to. (see the connect_to() in ClientSM class.)
- It is like the opcode + operand (i.e., the code of instructions; recalling the machine language we defined in lecture 3)

Your tasks in UP3

In client_state_machine.py, you need to complete code for the cases

- when a client is in the state of logged in, and some other client connects to her;
- when a client is in the state of chatting, and she receives a message from other clients;

In chat_server.py, you need to complete code for handling messages

- When the server receives messages whose “action” is “exchange”, “list”, “poem”, and “search” respectively.

Please read the UP3 specification carefully before you start.

How to run your code

We cannot test the code in the IDEs such as Spyder or IDLE.

- This time, we have to run the scripts in the terminal (or cmd for windows users)

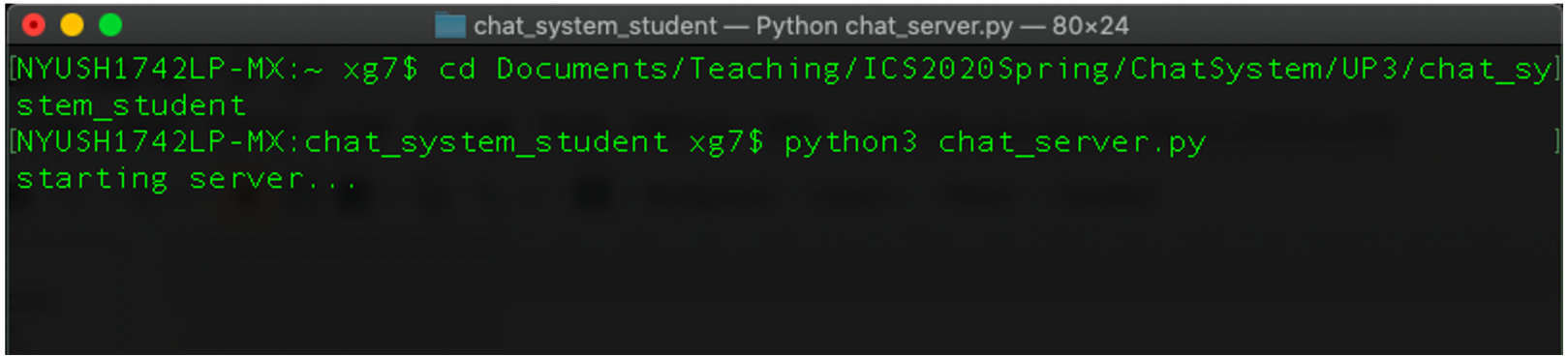
To run your script,

1. Open a terminal, then change current directory to the folder where you store your script (using the command “cd path”)
2. In the terminal, input “**python3** the_name_of_the_script” (for Windows users, python the_name_of_the_script)

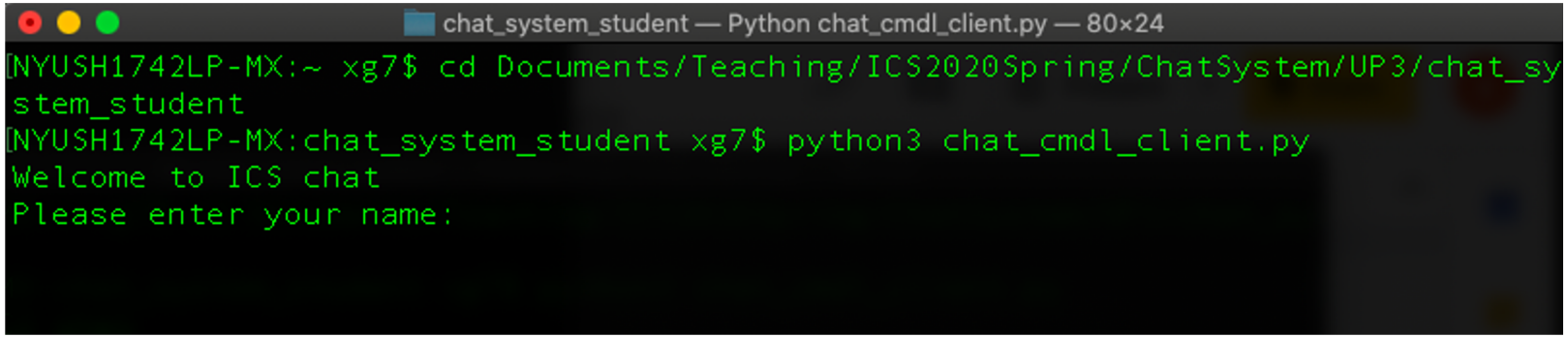
How to run your code

- run chat_server.py in one terminal, then, run chat_cmdl_client.py in another

1



2



Debugging in terminal

- run chat_server.py in one terminal, then, run chat_cmdl_client.py in another
- print() is very helpful
- press “Ctrl+c”, if you want to stop the server (or close the terminal directly)

Important: setting `CHAT_IP`

For Mac users:

- Please set the `CHAT_IP = ""` in `chat_utils.py`

For PC users:

- Please set the `CHAT_IP = socket.gethostbyname(socket.gethostbyname())` in `chat_utils.py`