

# Hands-on Programming Session

Laboratory of Bioinformatics I (Module 2)

*Author:*

Emidio Capriotti

[emidio.capriotti@unibo.it](mailto:emidio.capriotti@unibo.it)

University of Bologna (Italy)

# Contents

<b>1 Protein Structure Superimposition</b> . . . . .	<b>3</b>
<b>2 Protein Sequence Alignment Analysis</b> . . . . .	<b>6</b>
<b>3 Probabilistic Models</b> . . . . .	<b>8</b>
<b>4 CpG Island Hidden Markov Model</b> . . . . .	<b>11</b>
<b>5 BLAST-based Annotation</b> . . . . .	<b>15</b>
<b>6 Project Workflow</b> . . . . .	<b>20</b>
<b>7 Secondary Structure Propensity</b> . . . . .	<b>23</b>
<b>8 DSSP Data Analysis</b> . . . . .	<b>25</b>

# 1 Protein Structure Superimposition

**Exercise 1: Write a python script for superimposing two 3D structures.**

The script that should take in input two PDB structures, the relative chains and their residue intervals. The script has to include the following steps:

1. Parse the PDB file
2. Select the subset of residues
3. Calculate the transformation that minimizes the RMSD

The last step is should be performed using the [biopython](#) library. In particular we should use the module [SVDSuperimposer](#) that allow to calculate the best transformation and the RMSD.

For addressing this task we build a basic python script with two functions.

1. Parse the PDB file, select the subset of residues
2. Call SVDSuperimposer and return the transformation (Rotation matrix and Translation vector) and the RMSD.

As a reference for the selection of the coordinates in the PDB file consider the following fields in the [ATOM record](#):

- ATOM: 1-4
- ATOM TYPE: 13-16
- RESIDUE TYPE: 18-20
- CHAIN: 22
- RESIDUE NUMBER: 23-26
- X-COORD: 31-38
- Y-COORD: 39-46
- Z-COORD: 47-54

Below is reported a simple script to solve this exercise.

```
1  #!/usr/bin/env python
2  import sys
3  from Bio.SVDSuperimposer import SVDSuperimposer
4  import numpy as np
5
6  def get_list(naa):
7      vpos=[]
8      l1=naa.split(',')
9      for i in l1:
10         l2=i.split('-')
11         if len(l2)==1:
12             vpos=vpos+l2
13         elif len(l2)==2:
14             try:
15                 vpos=vpos+[str(j) for j in range(int(l2[0]),int(l2[1])+1)]
16             except:
17                 sys.stderr.write('ERROR: Incorrect input format '+naa)
18                 sys.exit(1)
19         else:
20             sys.stderr.write('ERROR: Incorrect input format '+naa)
21             sys.exit(1)
22     return vpos
23
24 def get_ca_atoms(pdbfile,chain,rlist,atom='CA'):
25     l_coord=[]
```

```

26 fpdb=open(pdbfile)
27 for line in fpdb:
28     if line[:4] != 'ATOM': continue
29     if line[21] != chain: continue
30     if line[22:26].strip() not in rlist: continue
31     if line[12:16].strip() != atom: continue
32     x=float(line[30:38])
33     y=float(line[38:46])
34     z=float(line[46:54])
35     l_coord.append([x,y,z])
36 return l_coord
37
38 def get_rmsd(coord1,coord2):
39     if len(coord1)!=len(coord2):
40         sys.stderr.write('ERROR: The sets of coordinates have
↪ different size.')
41         sys.exit(1)
42     svd=SVDSuperimposer()
43     svd.set(np.array(coord1),np.array(coord2))
44     svd.run()
45     rmsd=svd.get_rms()
46     rot,tran=svd.get_rotran()
47     return rot,tran,rmsd
48
49 if __name__ == '__main__':
50     if len(sys.argv)==7:
51         pdbfile1=sys.argv[1]
52         pdbfile2=sys.argv[2]
53         chain1=sys.argv[3]
54         chain2=sys.argv[4]
55         list1=get_list(sys.argv[5])
56         list2=get_list(sys.argv[6])
57         l_coord1=get_ca_atoms(pdbfile1,chain1,list1)
58         l_coord2=get_ca_atoms(pdbfile2,chain2,list2)
59         rot,tran,rmsd=get_rmsd(l_coord1,l_coord2)
60         print (f'R= [[ {rot[0][0]:7.3f}, {rot[0][1]:7.3f}, {rot
↪ [0][2]:7.3f} ],\n'+\
61             f'    [ {rot[1][0]:7.3f}, {rot[1][1]:7.3f}, {rot
↪ [1][2]:7.3f} ],\n'+\
62             f'    [ {rot[2][0]:7.3f}, {rot[2][1]:7.3f}, {rot
↪ [2][2]:7.3f} ]])')
63         print (f'T= [ {tran[0]:5.3f}, {tran[1]:5.3f}, {tran[2]:5.3f} ]'
↪ )
64         print (f'RMSD= {rmsd:5.3f}')
65     else:
66         print ("python super_pdb.py pdb1 pdb2 chain1 chain2 residues1
↪ residues2")

```

```

1 !python script/super_pdb.py data/3020.pdb data/3ZCF.pdb A A 10-60 10
↪ -60

```

```

R= [[ -0.216,  0.360,  0.908 ],
     [ -0.845, -0.534,  0.011 ],
     [  0.489, -0.765,  0.420 ]]
T= [ 32.090, 7.979, 0.263 ]

```

RMSD= 0.331

**WARNING:** This code is not safe in presence of Alternate locations in column 17 of the ATOM field.

## 2 Protein Sequence Alignment Analysis

### Exercise 2: Analyze a multiple sequence alignment to detect conserved sites.

We collected a set of 5 cytochromes for different species. The proteins have the following UniProt ID

- P99999 (human)
- P00004 (horse)
- P0C0X8 (Rhodobacter)
- P00091 (Rhodospseudomonas)
- Q93VA3 (Arabidopsis)

We build a unique fasta file containing all the sequences of the proteins downloading them automatically from the UniProt Rest API with the *wget* command.

```
1 for i in P99999 P00004 P0C0X8 P00091 Q93VA3
2 do
3     wget https://www.uniprot.org/uniprot/$i.fasta
4 done
5
6 cat P99999.fasta P00004.fasta P0C0X8.fasta P00091.fasta Q93VA3.
   ↪ fasta >cytc_aln.clw
```

You can now calculate the multiple sequence alignment of the five sequences using one of the most popular alignment methods such as [CLUSTALW](#), [MUSCLE](#) and [TCOFFEE](#).

You should save the output in clustalw format that contains separated columns representing the identifiers and the relative aligned sequences.

An example of the output is available [here](#).

Now you should write a python script that analyzes the multiple sequence alignment and calculates for each position the most abundant residue with its frequency and the information entropy ( $S$ ). Where  $S$  is

$$S = \sum_{i=0}^{20} -p_i \log(p_i)$$

and  $p$  are the frequencies of the amino acids. The program will have a function to parse the alignment file and a function that calculates the profile in each position of the alignment.

```
1 %%writefile parse_aln.py
2 #!/usr/bin/python
3 import sys
4 import numpy as np
5
6 def get_aln(alnfile):
7     d_aln={}
8     f=open(alnfile,'r')
9     for line in f:
10         if line.find('sp')!=0: continue
11         l=line.split()
12         sid=l[0]
13         seq=l[1]
14         d_aln[sid]=d_aln.get(sid,'')+seq
15     return d_aln
16
17 def get_profile(d_aln):
18     profile=[]
19     n=len(list(d_aln.values())[0])
```

```

20 sids=d_aln.keys()
21 for i in range(n):
22     aas=[d_aln[j][i] for j in sids]
23     vaas=get_iprofile(aas)
24     tot=float(vaas[:20].sum())
25     vaas[:20]=vaas[:20]/tot
26     profile.append(vaas)
27 return profile
28
29 def get_iprofile(aas,aa_list='ACDEFGHIKLMNPQRSTVWY-'):
30     v=np.zeros(len(aa_list))
31     for aa in aas:
32         pos=aa_list.find(aa)
33         if pos>-1: v[pos]=v[pos]+1
34     return v
35
36 def print_profile(profile,aa_list='ACDEFGHIKLMNPQRSTVWY-'):
37     n=len(profile)
38     for i in range(n):
39         pi=profile[i][:20]
40         s=0.0
41         for j in range(20):
42             if pi[j]>0: s=s-pi[j]*np.log(pi[j])
43         pm=pi.argmax()
44         print (i+1,aa_list[pm],s,pi[pm],profile[i][20])
45
46 if __name__ == '__main__':
47     alnfile=sys.argv[1]
48     d_aln=get_aln(alnfile)
49     profile=get_profile(d_aln)
50     print_profile(profile)

```

```

1 !python parse_aln.py ../data/cytc_aln.clw

```

```

1 M 0.0 1.0 4.0
2 R 0.0 1.0 4.0
3 L 0.0 1.0 4.0
4 V 0.0 1.0 4.0
5 L 0.0 1.0 4.0

```

The output of the program will return for each position the most frequent amino acid, the entropy, the frequency and the number of gaps.

Sorting this file you can detect the most conserved sites among the five proteins.

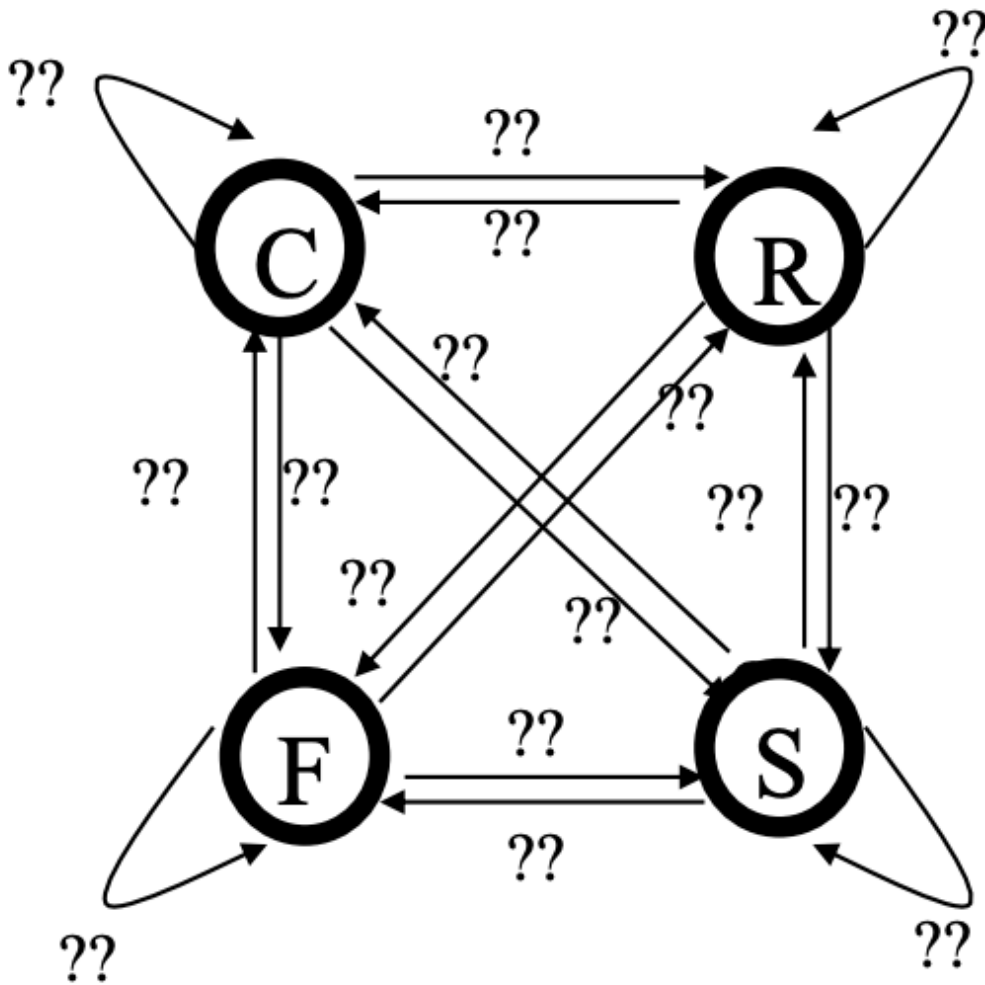


Figure 3.1: Probabilistic Model

### 3 Probabilistic Models

**Exercise 3: Write a python script to maximize the probability of a given sequence.**

Given the probabilistic model represented with the graph below, calculate the transition probabilities that maximize the probability of the following sequence of events

CCCFFCRRRCCSSSSFSFRRFFSSF

The python script takes in input a string representing a sequence of events and returns the transition matrix and the probability of the input sequence.

```

1  #!/usr/bin/python
2  import sys, random
3  import numpy as np
4
5  def get_alphabet(seq):
6      alpha=[]
7      for a in seq:
8          if a not in alpha: alpha.append(a)

```



```

9     alpha.sort()
10    return ''.join(alpha)
11
12    def get_tmatrix(seq, alpha):
13        n=len(alpha)
14        tm=np.zeros((n,n))
15        l=len(seq)
16        for i in range(l-1):
17            p1=alpha.find(seq[i])
18            p2=alpha.find(seq[i+1])
19            if p1>-1 and p2>-1: tm[p1][p2]+=1
20        for i in range(n):
21            tm[i,:]=tm[i,:]/np.sum(tm[i,:])
22        return tm
23
24    def get_prob(seq, tm, alpha):
25        p=1.0
26        l=len(seq)
27        for i in range(l-1):
28            p1=alpha.find(seq[i])
29            p2=alpha.find(seq[i+1])
30            if p1>-1 and p2>-1: p=p*tm[p1][p2]
31        return p
32
33    def print_logp(p):
34        if p>0.:
35            lp=-np.log10(p)
36        else:
37            lp=np.inf
38        return lp
39
40    def get_shuffle(seq):
41        l=[i for i in seq]
42        random.shuffle(l)
43        return ''.join(l)
44
45    def print_matrix(m, alpha):
46        print ('Transition Matrix:')
47        n=len(alpha)
48        for i in range(n):
49            for j in range(n):
50                print (alpha[i], '->', alpha[j], '%.3f\t' %m[i][j], end = '')
51            print ('')
52
53    if __name__ == '__main__':
54        if len(sys.argv)<2:
55            print ('python markov_model.py string_of_events')
56            sys.exit()
57        seq=sys.argv[1]
58        alpha=get_alphabet(seq)
59        tm=get_tmatrix(seq, alpha)
60        p=get_prob(seq, tm, alpha)
61        print_matrix(tm, alpha)
62        print ("\n-Log(Probability):", '%.3f' %print_logp(p))

```

The script from the impot sequence generates the alphabet with the function *get\_alphabet*.

The `get_tmatrix` function starting from position 0 and ending at the position n-1 counts all the transitions between state i and i+1. In the final step of the function the matrix is normalized by line.

The normalization corresponds to set to 1 the sum of probabilities of transition from one state.

The function `get_shuffle` is written to generate random shuffled sequence and show that their probabilities are in general lower than the probability of the input sequence.

```
1 !python markov_model.py CCCFFCRRRCSSSSFSFRRFFSSF
```

Transition Matrix:

```
C -> C 0.500 C -> F 0.167 C -> R 0.167 C -> S 0.167
F -> C 0.167 F -> F 0.333 F -> R 0.167 F -> S 0.333
R -> C 0.200 R -> F 0.200 R -> R 0.600 R -> S 0.000
S -> C 0.000 S -> F 0.429 S -> R 0.000 S -> S 0.571-
```

```
Log(Probability): 10.842
```

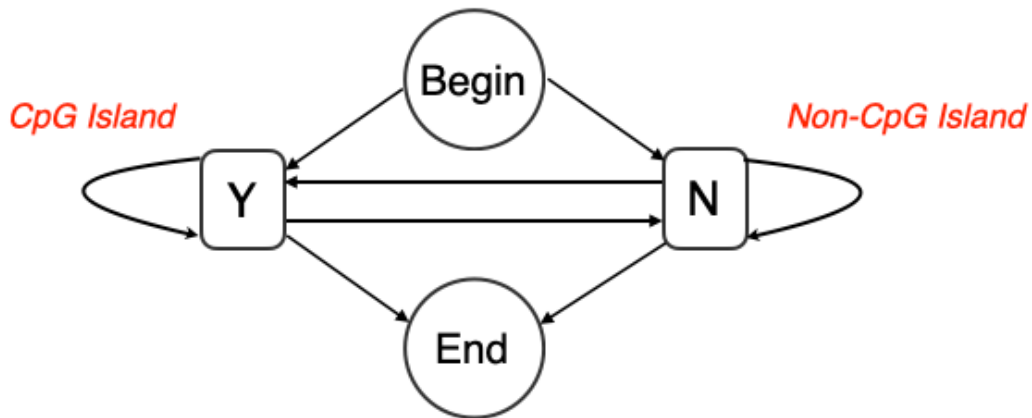


Figure 4.1: CpG-Island

## 4 CpG Island Hidden Markov Model

**Exercise 4: Write a script for generating and CpG Island HMM.**

For this exercise we consider as a training sequence the [human chromosome 21](#) downloaded from the ucsc genome browser. For the GpC Island annotation refer to the [cpgIslandExt.txt.gz](#) file.

From the `cpgIslandExt.txt` file extract the information relative to chromosome 21.

For modeling generation of CpG Island in nucleotide sequences we use the following Hidden Markov Model where the states Y and N can emit the letters representing the 4 nucleotides.

The script takes in input bedfile extracted from the `cpgIslandExt.txt` and the sequence of the chromosome 21 and calculates the transition and emission probabilities for each state.

```

1  #!/usr/bin/env python
2  import sys
3  import numpy as np
4  import pickle
5
6  def get_seq(seqfile):
7      seq=''
8      for line in open(seqfile):
9          if line[0]!='>':seq=seq+line.rstrip()
10     return seq
11
12     def get_ranges.bedfile):
13         l=[]
14         for line in open.bedfile):
15             v=line.split()
16             l.append([v[1],int(v[2]),int(v[3])])
17         return l
18
19     def match_seq.bed,seq):
20         s=''
21         n=len.bed)
22         p0=0
23         for i in range(n):
24             s=s+(bed[i][1]-p0-1)*'N'

```

```

25     s=s+(bed[i][2]-bed[i][1]+1)*'Y'
26     p0=bed[i][2]
27     s=s+(len(seq)-bed[i][2])*'N'
28     return s
29
30 def count(seq,cpg,nuc='ACGT',state='NY'):
31     t=np.zeros((len(state),len(state)))
32     e=np.zeros((len(state),len(nuc)))
33     n=len(seq)
34     for i in range(1,n):
35         ps=state.find(cpg[i])
36         pn=nuc.find(seq[i])
37         pf=state.find(cpg[i-1])
38         if ps>-1 and pn>-1:
39             e[ps][pn]+=1
40             if pf>-1: t[ps][pf]+=1
41     e[0]=e[0]/np.sum(e[0])
42     e[1]=e[1]/np.sum(e[1])
43     t[0]=t[0]/np.sum(t[0])
44     t[1]=t[1]/np.sum(t[1])
45     return e,t
46
47 def print_results(e,t,nuc='ACGT',state='NY'):
48     for i in range(len(state)):
49         for j in range(len(state)):
50             print (state[i],'->',state[j],'.6e\t' %t[i][j],end='')
51         print ('')
52     for i in range(len(state)):
53         print ('State:',state[i],end='\t')
54         for j in range(len(nuc)):
55             print (nuc[j]+':','.3f\t' %e[i][j],end='')
56         print ('')
57
58 if __name__ == '__main__':
59     if len(sys.argv)<3:
60         print ('python cpg-hmm.py bedfile seqfile')
61         sys.exit()
62     hmm={}
63     state='NY'
64     nuc='ACGT'
65     bedfile=sys.argv[1]
66     seqfile=sys.argv[2]
67     seq=get_seq(seqfile)
68     bed=get_ranges(bedfile)
69     cgp=match_seq(bed,seq)
70     seq=seq.upper()
71     e,t=count(seq,cgp)
72     print_results(e,t)
73     hmm['E']=e
74     hmm['T']=t
75     pickle.dump(hmm,open('../data/cpg-hmm.pik','wb'))

```

The first 3 functions of the scripr parse the bedfile and seqfile and generated 2 strings containing the DNA sequence and the CpG state. Only standard nucleic acids are considered for the statistics.

This calculation is performed on the first 50 lines of the cpgIslandExt.txt file and the first 10 million

nucleotides of chromosome 21. The pickle library is used to save the resulting HMM transition end probabilities on a file ('../data/cpg-hmm.pik') for future calculations.

```
1 !python cpg-hmm.py ../data/chr21_short.cpg ../data/chr21_short.fa
```

```
N -> N 9.999863e-01 N -> Y 1.370383e-05
Y -> N 1.208167e-03 Y -> Y 9.987918e-01
State: N A: 0.280 C: 0.216 G: 0.211 T: 0.293
State: Y A: 0.132 C: 0.357 G: 0.342 T: 0.170
```

We can now write a simple script that calculates the probability of the sequence giving the model using the forward algorithm.

```
1 #!/usr/bin/env python
2 import sys
3 import numpy as np
4 import pickle
5
6 def calculate_forward(seq,t,e,pb,pe,nuc='ACGT',state='NY'):
7     n=len(seq)
8     ns=len(state)
9     m=np.zeros((ns,n))
10    fp=0.0
11    for i in range(ns):
12        m[i][0]=pb[i]*e[i][nuc.find(seq[0])]
13    for i in range(1,n):
14        for j in range(ns):
15            for k in range(ns):
16                m[j][i]=m[j][i]+m[k][i-1]*t[k][j]*e[j][nuc.find(seq[i])]
17    for i in range(ns):
18        fp=fp+m[i][-1]*pe[i]
19    return m,fp
20
21 def calculate_logforward(seq,t,e,pb,pe,nuc='ACGT',state='NY'):
22    n=len(seq)
23    ns=len(state)
24    m=np.zeros((ns,n))
25    fp=0.0
26    for i in range(ns):
27        m[i][0]=-np.log10(pb[i]*e[i][nuc.find(seq[0])])
28    for i in range(1,n):
29        for j in range(ns):
30            tmp=0.0
31            for k in range(ns):
32                tmp=tmp+10**-(m[k][i-1]-m[0][i-1])*t[k][j]*e[j][nuc.find(seq
33                ↪ [i])]
34            m[j][i]=m[0][i-1]-np.log10(tmp)
35    tmp=0.0
36    for i in range(ns):
37        tmp=tmp+10**-(m[i][-1]-m[0][-1])*pe[i]
38    fp=m[0][-1]-np.log10(tmp)
39    return m,fp
40
41 if __name__ == '__main__':
42     if len(sys.argv)<3:
43         print ('python forward-hmm.py seqfile picklefile')
```

```
43     sys.exit()
44     state='NY'
45     nuc='ACGT'
46     seq=sys.argv[1]
47     picklefile=sys.argv[2]
48     hmm=pickle.load(open(picklefile,'rb'))
49     # Is assumed that pbegin is 0.5
50     pb=np.array([0.5,0.5])
51     # Is assumed that pend is 0.5 - thiis not correct
52     pe=np.array([0.5,0.5])
53     t=hmm['T']
54     e=hmm['E']
55     m,fp=calculate_logforward(seq,t,e,pb,pe)
56     print ('-Log10 Probability:', '%.3f' %fp)
```

```
1 !python forward-hmm.py ACGTG ../data/cpg-hmm.pik
```

```
-
Log10 Probability: 3.366
```

## 5 BLAST-based Annotation

### Exercise 5: Build a BLAST-based method for protein annotation.

In preparation of the project we will build a BLAST-based method for the annotation of BPTI/Kinutz domain. For this purpose we collect a positive set of proteins containing the BPTI/Kinutz domain from UniProt and a negative set from the same database. Divide the human and non-human proteins to build a training and testing sets respectively.

#### Selecting training set from UniProt:

- Positives: `database:(type:pfam pf00014) organism:"Homo sapiens (Human) [9606]" AND reviewed:y`
- Negatives: `NOT database:(type:pfam pf00014) AND reviewed:yes AND organism:"Homo sapiens (Human) [9606]"`

#### Selecting testing set from UniProt:

- Positives: `database:(type:pfam pf00014) NOT organism:"Homo sapiens (Human) [9606]" AND reviewed:y`
- Negatives: `NOT database:(type:pfam pf00014) AND reviewed:yes NOT organism:"Homo sapiens (Human) [9606]"`

The composition of the sets will be the following:

- Human BPTI/Kunitz: 18
- BPTI/Kunitz Non Human: 18
- Not BPTI/Kunitz Human: 20347

For the negatives we divide the set in two (Training and Testing) selecting the first 10000 proteins as training and the remaining ones as Testing.

In my file I see that protein O95081 is the 10000th one and that sequence finishat line 109074, thus we split the file using the following command.

```
1 head -n 109074 Human_NotPF00014.fasta >Human_NotPF00014_Training.
   ↪ fasta
2 tail -n +109075 Human_NotPF00014.fasta >Human_NotPF00014_Testing.
   ↪ fasta
```

For performing the BLAST serch we need to format out positive training set of human BPTI/Kinutz proteins.

```
1 formatdb -i Human_PF00014.fasta
```

```
1 grep ">" Human_NotPF00014.fasta |head -n 10000 |tail -n 1
```

Run BLAST on the Human\_PF00014.fasta dataset as a consistency test. We want to check that all the human sequence are

```
1 # For checking purposes
2 blastpgp -i Human_PF00014.fasta -d Human_PF00014.fasta -o
   ↪ Human_PF00014.bl8 -m 8
3
4 # sort based on the 11 column to check the highst e-value
5 sort -grk 11 Human_PF00014.bl8| head -n 1
```

**WARNING:** we should use the option -g for sorting instead of -n. This could be different depending on linux distribution.

We now work on the negative Training set

```
1 # Run on negatives
2 blastpgp -i Human_NotPF00014_Training.fasta -d Human_PF00014.fasta -
  o Human_NotPF00014_Training.bl8 -m 8
3
4 # sort the output to chekc the sequence with lowest e-value
5 sort -gk 11 Human_NotPF00014_Training.bl8 |head -n 2 |awk '{print $1
  ↪ , $2, $11}'
6
7 sp|A6NMZ7|C06A6_HUMAN sp|P12111|C06A3_HUMAN 0.0
8 sp|A8TX70|C06A5_HUMAN sp|P12111|C06A3_HUMAN 0.0
```

We observe that there are many non BPTI/Kunitz proteins tha get a low e-value. Moving to the training set we can now run BLAST on both positive and negative sets.

```
1 # Run on positive testing set
2 blastpgp -i NotHuman_PF00014.fasta -d Human_PF00014.fasta -o
  ↪ NotHuman_PF00014.bl8 -m 8
3
4 # Run on negative testing set
5 blastpgp -i Human_NotPF00014_Testing.fasta -d Human_PF00014.fasta -o
  ↪ Human_NotPF00014_Testing.bl8 -m 8
```

We need first to rank the output on the posotive set of non Human BPTI/Kunitz proteins. To do so we grep based on the identifiers and select the match with lowest e-value.

```
1 for i in `awk '{print $1}' NotHuman_PF00014.bl8 |sort -u `
2 do
3     grep $i NotHuman_PF00014.bl8 |sort -gk 11 |head -n 1
4 done > NotHuman_PF00014.bl8.best
```

Now let's select an e-value of 0.001 as a classification threshold for the classification of BPTI/kunitz proteins. Accorddin to this assumption we calculated the number of proteins with maximu e-value below that threshold.

```
1 # Test on positives
2 awk '{if ($11<0.001) print $1}' NotHuman_PF00014.bl8.best |sort -u
  ↪ |wc -l
3     340
4
5 awk '{if ($11<0.001) print $1}' Human_NotPF00014_Training.bl8 |
  ↪ sort -u |wc
6     290
7
8 awk '{if ($11<0.001) print $1}' Human_NotPF00014_Tresting.bl8 |
  ↪ sort -u |wc -l
9     200
```

According to this results and considering the non Human BPTI/Kunits as a positive in training and testing we caculate the following confusion matrices



M-Training	Real_Positive	Real_Negative
Pred_Positive	340	290
Pred_Negative	1	9710

M-Testing	Real_Positive	Real_Negative
Pred_Positive	340	200
Pred_Negative	1	10147

We can now calculate the performance of the method selecting a threshold of 0.001 in terms of accuracy and Matthews correlation coefficient.

```

1 import math
2
3 # Training confusion matrix
4 mtrain=[[340,290],[1,9710]]
5
6 # Training confusion matrix
7 mtest=[[340,200],[1,10147]]
8
9 def accuracy(m):
10     return float(m[0][0]+m[1][1])/(sum(m[0])+sum(m[1]))
11
12 def mcc(m):
13     d=(m[0][0]+m[1][0])*(m[0][0]+m[0][1])*(m[1][1]+m[1][0])*(m[1][1]+
14     ↪ m[0][1])
15     return (m[0][0]*m[1][1]-m[0][1]*m[1][0])/math.sqrt(d)
16
17 print ('ACC Training= %5.3f' %accuracy(mtrain))
18 print ('MCC Testing= %5.3f\n' %mcc(mtrain))
19
20 print ('ACC Training= %5.3f' %accuracy(mtest))
21 print ('MCC Testing= %5.3f\n' %mcc(mtest))

```

```

ACC Training= 0.972
MCC Testing= 0.723

```

```

ACC Training= 0.981
MCC Testing= 0.785

```

We can observe that the performance in training and testing are different especially in terms of Matthews correlation coefficient.

To solve the same problem with more general solution we can write a python script that analyze a file derived from the BLAST output.

The program will take in input a file with three columns representing the following data

1. Protein ID
2. The lowest e-value associated to each protein
3. The class (0: NotBPTI/Kunitz 1: BPTI/Kunitz)

```

1 #!/usr/bin/python
2 import sys, math
3

```

```

4 def get_blast(filename):
5     flist=[]
6     d={}
7     f=open(filename)
8     for line in f:
9         v=line.rstrip().split()
10        d[v[0]]=d.get(v[0],[])
11        d[v[0]].append([float(v[1]),int(v[2])])
12    for v in d.values():
13        v.sort()
14        flist.append(v[0])
15    return flist
16
17 def get_cm(data,th):
18     # CM=[[TP,FP],[FN,TN]]
19     # 0 = Negatives 1=Positives
20     cm=[[0.0,0.0],[0.0,0.0]]
21     for i in data:
22         if i[0]<th and i[1]==1:
23             cm[0][0]=cm[0][0]+1
24         if i[0]>=th and i[1]==1:
25             cm[1][0]=cm[1][0]+1
26         if i[0]<th and i[1]==0:
27             cm[0][1]=cm[0][1]+1
28         if i[0]>th and i[1]==0:
29             cm[1][1]=cm[1][1]+1
30     return cm
31
32 def get_acc(cm):
33     return float(cm[0][0]+cm[1][1])/(sum(cm[0])+sum(cm[1]))
34
35 def mcc(m):
36     d=(m[0][0]+m[1][0])*(m[0][0]+m[0][1])*(m[1][1]+m[1][0])*(m[1][1]+
37     ↪ m[0][1])
38     return (m[0][0]*m[1][1]-m[0][1]*m[1][0])/math.sqrt(d)
39
40 if __name__ == "__main__":
41     filename=sys.argv[1]
42     #th=float(sys.argv[2])
43     data=get_blast(filename)
44     for i in range(20):
45         th=10**-i
46         cm=get_cm(data,th)
47         print 'TH:',th,'ACC:',get_acc(cm),'MCC:',mcc(cm),cm

```

To run this progra we need to generate the input file form the the BLAST output extracting the protein ID from the 1st column and the e-value from the 11th column and adding 0 or 1 depending on the protein files we are considering.

In our case we should run the following commands to extract the data from the 3 files we have:

```

1 # Command for generating the positive set
2 awk '{split($1,v,'|'); print v[2],$11,1}' NotHuman_PF00014.b18 >
   ↪ positive.txt
3
4 # Command for generating the negative Training set

```

```

5 awk '{split($1,v,'|'); print v[2],$11,0}' Human_NotPF00014_Training
   ↪ .bl8 > negative_train.txt
6
7 # Command for generating the negative Testing set
8 awk '{split($1,v,'|'); print v[2],$11,0}' Human_NotPF00014_Testing.
   ↪ bl8 > negative_test.txt

```

Now you can merge positive and negatives in a unique file with the cat command and give the resulting file in input to the python script above.

```

1 cat positive.txt negative_train.txt > train_data.txt
2 python performance.py train_data.txt
3
4 cat positive.txt negative_test.txt > test_data.txt
5 python performance.py test_data.txt

```

**WARNING:** Remember that BLAST do not return any output for sequence that match with an e-value higher than 10 for this reason the total number of sequences you will have in the previous file will not sum up to the total number of proteins in your set.

You can use the *comm* command to determine the missing sequences and run again the script over the corrected files.

## 6 Project Workflow

### Main steps for the implementation of the project.

**Main Aim:** Development of a method for the detection of BPTI/Kunitz domain in proteins.

**Methodology:** Hidden Markov Model based on structural alignment.

The workflow of the project is summarized as follows

1. Selection of a representative set of protein structures from PDB.
2. Multiple structural alignment with web available tools.
3. Generation of the Hidden Markov Model for modeling BPTI/Kunitz domain.
4. Selection of training and testing set from UniProt.
5. Method optimization and assessment.

### Selection a representative set of protein structures from PDB.

For this task we use the advanced search on the PDB web page. It is possible to use different constraints in the search. For the selection of the structures it is important to consider:

- The resolution
- The length of the protein
- External identifiers for BPTI/Kunitz domain (e.g. PFAM, SCOP, CATH)

When a list of PDB structures is returned if it is feasible you should look to the proteins to check if they have a BPTI/Kunitz domain. From the PDB web page it is possible to download the sequence report it is possible to download the protein sequence and check for the possible redundancy in the set of proteins. To check for the redundancy in the dataset you can use *blastclust* algorithm. Blastclust takes in input a fasta file containing a set of sequences and clusters them according to the level of sequence identity and coverage between the proteins. The options *-S* and *-L* for the program are used to control the percentage of sequence identity and the coverage respectively.

```
1 #The command sed and awk can be used to clean the csv file returned
   ↪ by the PDB.
2
3 # Replace inner comma
4 sed 's/, /_/g' filename
5
6 # Remove spaces
7 sed 's/ //g' filename
8
9 # split by comma
10 awk -F "," '{print $1,$2}' filename
11
12 # After generating a fasta file run blastclust with sequence identity
   ↪ of 95% and 90% coverage.
13 blastclust -i seqfile.fasta -o seqfile.clust -S 95 -L 0.9
```

When the cluster file is returned select a representative structure for each cluster considering the resolution of the structure and the length of the protein.

### Multiple structural alignment with web available tools.

After collecting a clean set of protein structures calculate the multiple structure alignment using web available tools such as [PDBeFold](#), [PROMALS3D](#), [mTM-align](#). Check the returned alignment and look for the RMSD between pair of structures and the sequence identity. This information can be important to detect possible errors in the initial selection of the BPTI/Kunitz domain proteins. Finally download the fasta file representing the multiple structure alignment.

## Generation of the Hidden Markow Model for modeling BPTI/Kunitz domain.

When the alignment is returned it is important to manually check the alignment to look for the conserved residues and the possible errors in the alignment. After that you can generate the HMM using *hmmbuild* command from [HMMER](#) package.

```
1 # Generate the HMM
2 hmmbuild bpti-kunitz.hmm align-bpti-kunitz.fasta
```

## Selection of training and testing set from UniProt.

This task is performed using the advanced search of UniProt database using as a key search the Pfam identifier of the BPTI/Kunitz domain. The search has to be restricted to the protein in SwissProt that are manually annotated. With this search you will obtain ~350 protein containing BPTI/Kunitz domain that will represent the positive set. All the remaining proteins in SwissProt can be used as negatives.

For a fair test of your HMM model you should remove the sequence in the positive set that share high level of sequence identity with the protein structures collected for generating the model. This task can be performed using *blastpgp* with the option *-m 8* checking for sequences with low e-value (column 11) and high sequence identity (column 3).

```
1 # Run transforming the sequences of the selected structures in
  ↪ database
2 formatdb -i selected-bpti-kunits.fasta
3
4 # Run blastpgp to match the positive set against the selected bpti-
  ↪ kunitz.
5 blastpgp -i positives.fasta -d selected-bpti-kunits.fasta -m 8 -o
  ↪ positives.bl8
```

Using the output of *blastpgp* you can select the proteins that should be removed from the set of positives. *italicized text*

## Method optimization and assessment.

When the final benchmark set has been collected a basic testing procedure consists in the implementation of a 2-fold cross-validation test. It consists in splitting positives and negatives in 2 subsets, optimizing the classification threshold on one subset and testing the performance on the other subset.

When the dataset is splitted in two parts with the same size you can run the *hmmsearch* on the different subsets. It is better to run the command using the following options:

- *--max*: turns off all the heuristics for cutting of distantly related proteins
- *--noali*: exclude from the output the alignments
- *--tblout*: returns the output in tabular form
- *-Z*: It is important for normalizing the e-value output

An example of command for matching a set of sequences (*clean\_set.fasta*) with the HMM model (*bpti-kunitz.hmm*) is the following:

```
1 # hmmsearch command matching the model (bpti-kunitz.hmm) with
  ↪ sequence in clean_set.fasta
2
3 hmmsearch --noali --max --tblout output.txt -Z 1 bpti-kunitz.hmm
  ↪ clean_set.fasta
```

The output file can be parsed considering the lines without the hash character (#) at the beginning selecting and the column 1 corresponding to the protein identifier and either column 5 or 8 that correspond to the full-sequence and best-domain e-values respectively.

For the optimization and testing of the performance we need to check the *hmmsearch* output and include those proteins, presumably negatives, for which no match is returned. For the classification purposes you should assign to that proteins and e-value greater or equal than the e-value threshold used for running *hmmsearch*.

When all the results for the subset are collected in files containing protein ID, e-value and protein class (0/1), we can use the [performace.py script](#) previously developed to calculate the performance of our classification method at different e-value threshold.

## 7 Secondary Structure Propensity

**Exercise 7:** Write a python script for calculating the secondary structure propensity of the amino acids.

For calculating the amino acid secondary structure propensity we need to select a representative set of protein structures from the PDB.

We can use the advanced search for selecting high quality PDB structures containing only proteins with resolution below 2 .

To remove the redundancy, the clusters of proteins in the PDB are made available at this [page](#).

The information about the secondary structure of the protein can be downloaded from this [link](#). This file contains for each PDB entry its sequence and secondary structure.

After collecting a set of PDB and chains ids we can use a script for parsing fasta files to extract the information about the sequence and secondary structure of the proteins in our dataset.

The secondary structure propensity score ( $PS(aa,ss)$ ) is calculated comparing the joint probability of having a residue  $aa$  with secondary structure  $ss$  ( $P(aa,ss)$ ) divided by the probability of independent events ( $P(aa)P(ss)$ ).

$$PS(aa,ss) = \frac{P(aa,ss)}{P(aa)P(ss)}$$

The python scripts take in input a file containing a list of PDB chain identifiers (PDBID:CHAIN) and the `ss_dis.txt` file.

```
1  #!/usr/bin/python
2  import sys
3
4  def get_ids(idfile):
5      ids=open(idfile).read().rstrip().split('\n')
6      return ids
7
8  def get_count(d):
9      daa={}
10     dss={}
11     das={}
12     for k in d.keys():
13         for aa in d[k][0]:
14             daa[aa]=daa.get(aa,0)+1
15         for ss in d[k][1]:
16             dss[ss]=dss.get(ss,0)+1
17         n=len(d[k][0])
18         for i in range(n):
19             aa=d[k][0][i]
20             ss=d[k][1][i]
21             das[(aa,ss)]=das.get((aa,ss),0)+1
22     return daa,dss,das
23
24 def get_seqs(ids,dbfile):
25     d={}
26     with open(dbfile, 'r') as fdb:
27         for line in fdb:
28             if line[0]=='>':
29                 #pid=line[1:].split(':')[2]
30                 pid=line[1:7]
31                 ts=line[8:].rstrip()
```

```

32         continue
33     if pid in ids:
34         d[pid]=d.get(pid,['',''])
35         if ts=='sequence': d[pid][0]=d[pid][0]+line.rstrip()
36         if ts=='secstr': d[pid][1]=d[pid][1]+line.rstrip('\n').
↪     replace(' ','C')
37     return d
38
39 if __name__ == '__main__':
40     idfile=sys.argv[1]
41     dbfile=sys.argv[2]
42     ids=get_ids(idfile)
43     d=get_seqs(ids,dbfile)
44     daa,dss,das=get_count(d)
45     na=float(sum(daa.values()))
46     ns=float(sum(dss.values()))
47     nas=float(sum(das.values()))
48     for k in das:
49         if k[1]!='H': continue
50         p=(das[k]/nas)/((daa[k[0]]/na)*(dss[k[1]]/ns))
51     print (k,p)

```

The *get\_ids* function extracts from the first file the list of identifiers.

The *get\_seqs* function generates a dictionary with the PDB chain id as key and a list of two strings including primary sequence and secondary structure as value.

Finally the *get\_count* functions generates 3 dictionaries including the counting of the amino acid composition (daa) the secondary structures (dss) and the join events (das).

The last *if* of the code is included to print only the propensity scale for the -helix.

You can compare the results returned by the previous script with [Chou-Fasman propensity scale](#) calculating the correlation coefficient. The [linregress function](#) of the `scipy.stats` library can be used for calculating the correlation coefficient.



## 8 DSSP Data Analysis

Emidio Capriotti - University of Bologna (Italy)

Laboratory of Bioinformatics I - Second Module

### Exercise 8: Write a python script for parsing a dssp file.

Generate a DSSP file from the PDB structure [1BI7](#) and write a python script to parse the DSSP file. The DSSP file can be generated using the [dssp](#) program. After downloading the dssp program the dssp can be generated using the following command:

```
1 dssp pdbfile > dsspfile
```

The dssp of the 1BI7 structure can be also downloaded for the [ftp repository](#) using the following [link](#).

The script for parsing thd DSSP file starts to extract the information after detecting the header ('# RESIDUE') excluding the row that contain the ! symbol in column 14. The script calculates the relative solvent accessibility of each residue dividing the accessibility by the maximum accessibility of each residue.

The dictionary containing the maximum solvent accessibility of each residue is stored in the Norm\_Acc dictionary.

```
1 #/usr/bin/python
2 import sys
3
4 Norm_Acc={"A" :106.0, "B" :160.0, \
5          "C" :135.0, "D" :163.0, "E" :194.0, \
6          "F" :197.0, "G" : 84.0, "H" :184.0, \
7          "I" :169.0, "K" :205.0, "L" :164.0, \
8          "M" :188.0, "N" :157.0, "P" :136.0, \
9          "Q" :198.0, "R" :248.0, "S" :130.0, \
10         "T" :142.0, "V" :142.0, "W" :227.0, \
11         "X" :180.0, "Y" :222.0, "Z" :196.0}
12
13 def get_dssp(dsspfile):
14     dssp=[]
15     f=open(dsspfile)
16     c=0
17     for line in f:
18         if line.find(' # RESIDUE')>-1:
19             c=1
20             continue
21         if c==0: continue
22         if len(line)<115: continue
23         rnum=line[5:10].strip()
24         ch=line[11]
25         aa=line[13]
26         if aa=='!': continue
27         ss=line[16]
28         if ss==' ': ss='C'
29         acc=float(line[35:38])
30         phi=float(line[103:109])
31         psi=float(line[109:115])
32         rsa=min(1.0, acc/Norm_Acc[aa])
33         dssp.append([rnum,ch,aa,ss,rsa,phi,psi])
34     return dssp
```

```

35
36 if __name__ == '__main__':
37     dsspfile=sys.argv[1]
38     dssp=get_dssp(dsspfile)
39     print("#N", 'CH', 'AA', 'SS', 'RSA', 'PHI', 'PSI', sep='\t')
40     for data in dssp:
41         (rnum, ch, aa, ss, rsa, phi, psi)=data
42         print(rnum, ch, aa, ss, '%.4f' %rsa, '%.1f' %phi, '%.1f' %psi, sep='\
↪ t')

```

We can now run the script to select the data relatives to the amino acid K.

```

1 !python parse_dssp.py ../data/1bi7.dssp |awk '{if ($3=="K") print
↪ $0}'

```

```

26  A K E 0.4049 -109.2 163.0
29  A K E 0.0146 -96.4 148.5
34  A K T 0.4634 -73.5 -33.6
43  A K E 0.1854 -127.0 100.3
93  A K E 0.4195 -66.9 126.9
111 A K H 0.3220 -72.2 -46.8
123 A K H 0.3024 -43.3 -41.9
147 A K C 0.0244 -79.5 165.8
160 A K E 0.0780 -142.4 130.6
216 A K S 0.4537 -165.2 128.0
230 A K H 0.2537 -35.6 -50.1
257 A K C 0.5171 -20.1 122.4
264 A K G 0.6244 -61.2 -28.0
274 A K H 0.2488 -49.3 -36.1
279 A K H 0.4195 -63.8 -25.5
287 A K T 0.6634 -79.6 -33.4

```