Jacek Radajewski
Student number: 43612772
SCIE2100 Prac 8

Question 5

Chosen sequence was 1EVH.

1EVH Sequence

```
SEQSICQARAAVMVYDDANKKWVPAGGSTGFSRVHIYHHTGNNTFRVVGRKIQDHQVVIN
CAIPKGLKYNQATQTFHQWRDARQVYGLNFGSKEDANVFASAMMHALEVLN
```

1EVH secondary structure from the sstr3.fa

```
CEEEEEEEEEEEEEECCCCEEEEHHHCCCCEEEEEEECCCCEEEEEEECCCCCEEEE
EEECCCCCCECCCCCEEEEECCCCEEEEEECCHHHHHHHHHHHHHHHHHHC
```

Predicted structure from running my code

```
HEEEEEEEEEEEEEEEEHHHHHHHH----EEEEEEEEEE-HEEEEEEEEEEEEEEEEE
EEEEHEEEEEEEEEEEEEEEEHHHHHHHHHHHHHHHHHHHHHHHHHHHHH
```

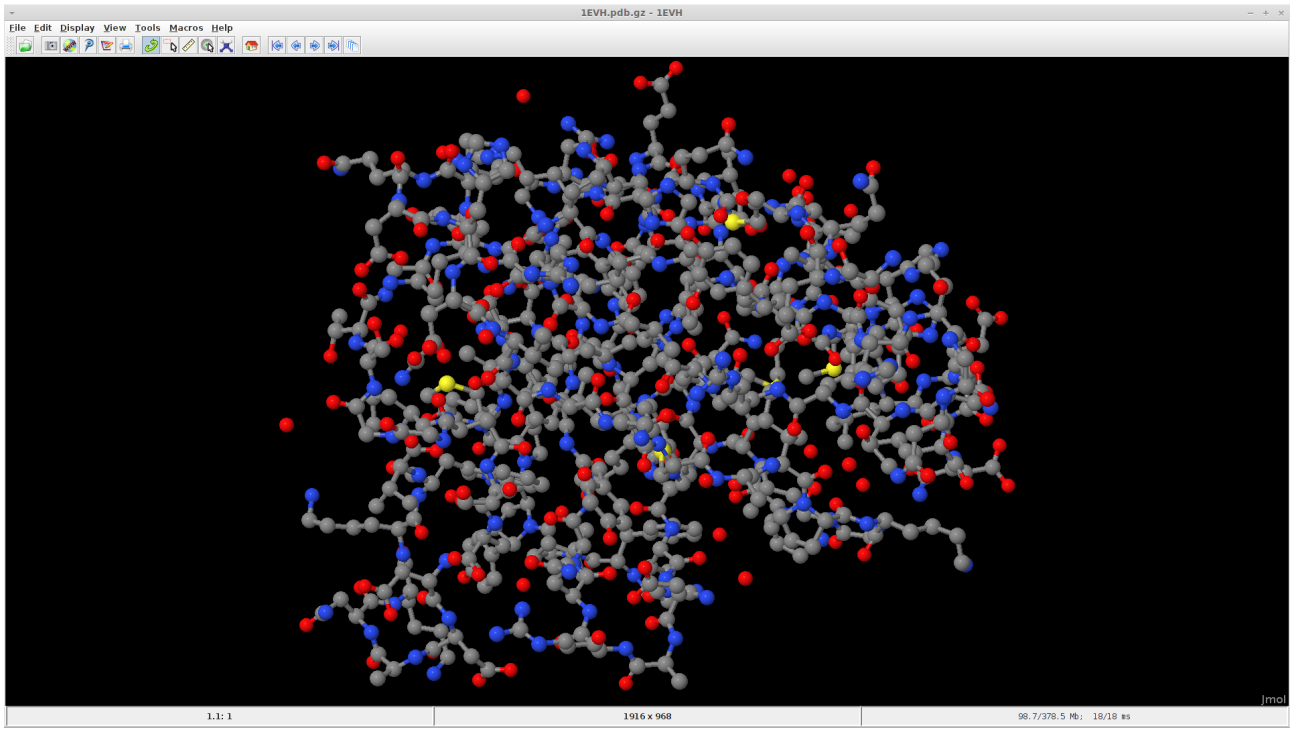
The accuracy was calculated at 61.26%. This was obtained by counting matches and dividing them by the total number of residues. A successful match was when expected and actual were both alpha helix or expected and actual were both beta sheets or when expected and actual were both neither an alpha helix or beta sheet.

Note that the alpha helix and beta sheet predictions obtained from the 1EVH sequence in sstr3.fa seemed to be identical to that obtained from the PDB web site (see below). Because of this I have read and used the sstr3.fa structure in my analysis (code).

Execution

```
Terminal
File  Edit  View  Search  Terminal  Help
jacekrad@z400 ~/var/github/prac08/src $ python question5.py
     1EVH: SEQSICQARAAVMVYDDANKKWVPAGGSTGFSRVHIYHHTGNNTFRVVGRKIQDHQVVINCAIPKGLKYNQATQTFHQWRDARQVYGLNFGSKEDANVFASAMMHALEVLN
     1EVH: CEEEEEEEEEEEEEECCCCEEEEHHHCCCCEEEEEEECCCCEEEEEEECCCCCEEEEEECCCCCECCCCCEEEEECCCCEEEEECCHHHHHHHHHHHHHHHHHC
 alpha:    H----------------HHHHHHH-------------H--------------------H----------------HHHHHHHHHHHHHHHHHHHHHHHHHH
  beta:     -EEEEEEEEEEEEEEE-----------EEEEEEEEE--EEEEEEEEEEEEEEEEEEE-EEEEEEEEEEEEEEEE--------------------------
Combined:  HEEEEEEEEEEEEEEEEEHHHHHHHH----EEEEEEEEE-HEEEEEEEEEEEEEEEEEEEEEHEEEEEEEEEEEEEEEEEEHHHHHHHHHHHHHHHHHHHHHHHHHH
111  structures with  68  correctly matched.
Accuracy 61.26%
jacekrad@z400 ~/var/github/prac08/src $
```

JMOL structure view



Code

```
'''
Created on 20/05/2014
prac08 question 5

@author: s4361277
'''
from sequence import *
from symbol import *
from sstruct import *

# read both protein and structure sequences into lists
proteins = readFastaFile("prot2.fa", Protein_Alphabet)
structures =  readFastaFile("sstr3.fa", DSSP3_Alphabet)

# store protein and structure sequences in dictionary for easy retrieval
protein_map = {}
structure_map = {}

for protein in proteins:
    protein_map.update({protein.name:protein})

for structure in structures:
    structure_map.update({structure.name:structure})

protein = protein_map.get("1EVH")
structure = structure_map.get("1EVH")


alpha = getScores(protein, 0)
calls_a1 = markCountAbove(alpha, width=6, call_cnt=4)


alpha = getScores(protein, 0) # values from column 0
beta = getScores(protein, 1) # values from column 1

calls_a1 = markCountAbove(alpha, width=6, call_cnt=4)
calls_a2 = extendDownstream(alpha, calls_a1, width=4)
calls_a3 = extendUpstream(alpha, calls_a2, width=4)

calls_b1 = markCountAbove(beta, width=5, call_cnt=3)
calls_b2 = extendDownstream(beta, calls_b1, width=4)
calls_b3 = extendUpstream(beta, calls_b2, width=4)

avg_a = calcRegionAverage(alpha, calls_a3)
avg_b = calcRegionAverage(beta, calls_b3)

diff_a = [avg_a[i] - avg_b[i] for i in range(len(avg_a))]
diff_b = [avg_b[i] - avg_a[i] for i in range(len(avg_a))]

calls_a4 = checkSupport(calls_a3, diff_a)
calls_b4 = checkSupport(calls_b3, diff_b)

# print the sequence, structure from file and calculated alpha helix
# and beta sheet structures
print "    ", protein
print "    ", structure
alpha_string = makesstr(calls_a4, 'H')
beta_string = makesstr(calls_b4, 'E')
# create a combined string
combined_string = ""
```

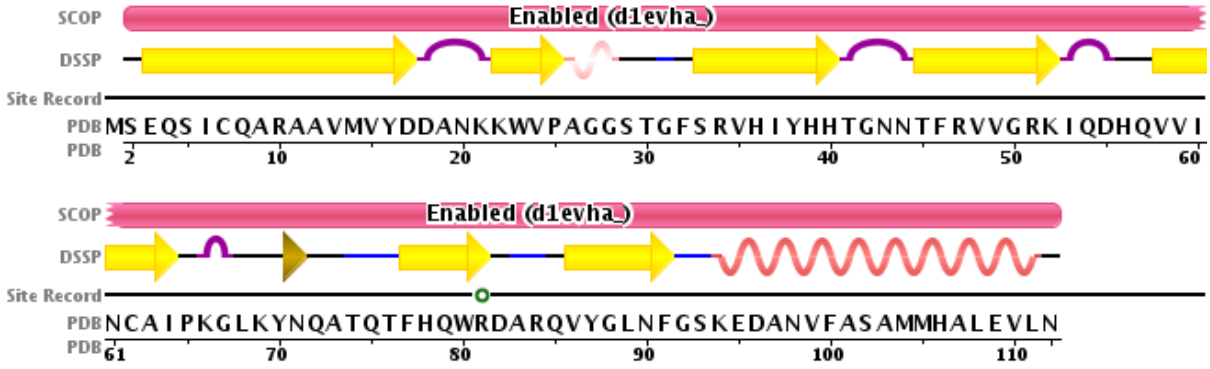
```
for i in range(0, len(alpha_string)):
    if beta_string[i] == 'E':
        combined_string += 'E'
    else:
        combined_string += alpha_string[i]

print " alpha:    ", alpha_string
print "  beta:    ", beta_string
print "Combined: ", combined_string
# to check the accuracy we simply compare our prediction to the one
# obtained from sstr3.fa

position = 0
match_count = 0
for element in structure.sequence:
    if element == "H" and calls_a4[position]:
        match_count += 1 # matched alpha helix
    if element == "E" and calls_b4[position]:
        match_count += 1 # matched beta sheet
    if element != "H" and element !="E" and not(calls_a4[position]) and
not(calls_b4[position]):
        match_count += 1 # matched other
    position += 1

print position, " structures with ", match_count, " correctly matched."
print "Accuracy %.2f%%" % ((float(match_count) /  position) * 100)
```

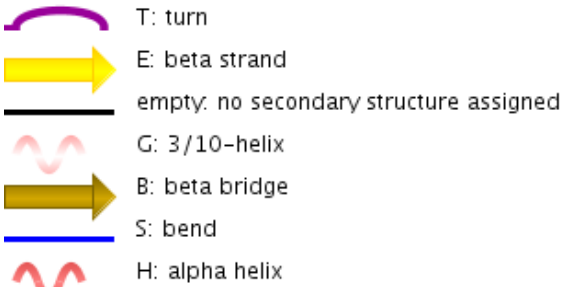
Secondary structure from Protein Data Bank



Site Record Legend

BINDING SITE FOR RESIDUE ACE B 1000 (SOFTWARE)

DSSP Legend

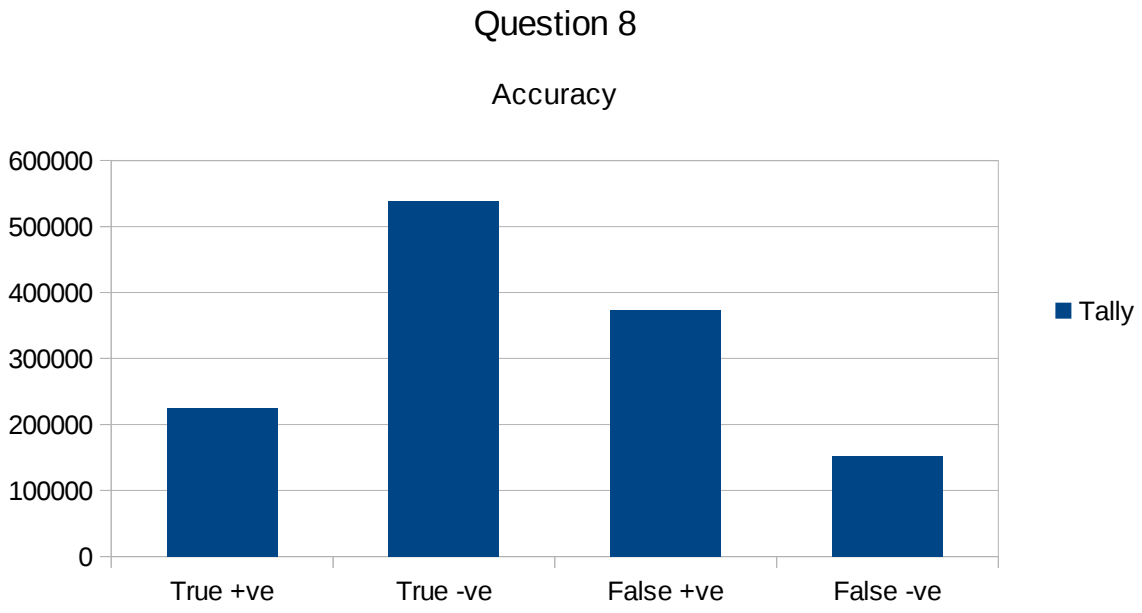


Question 7

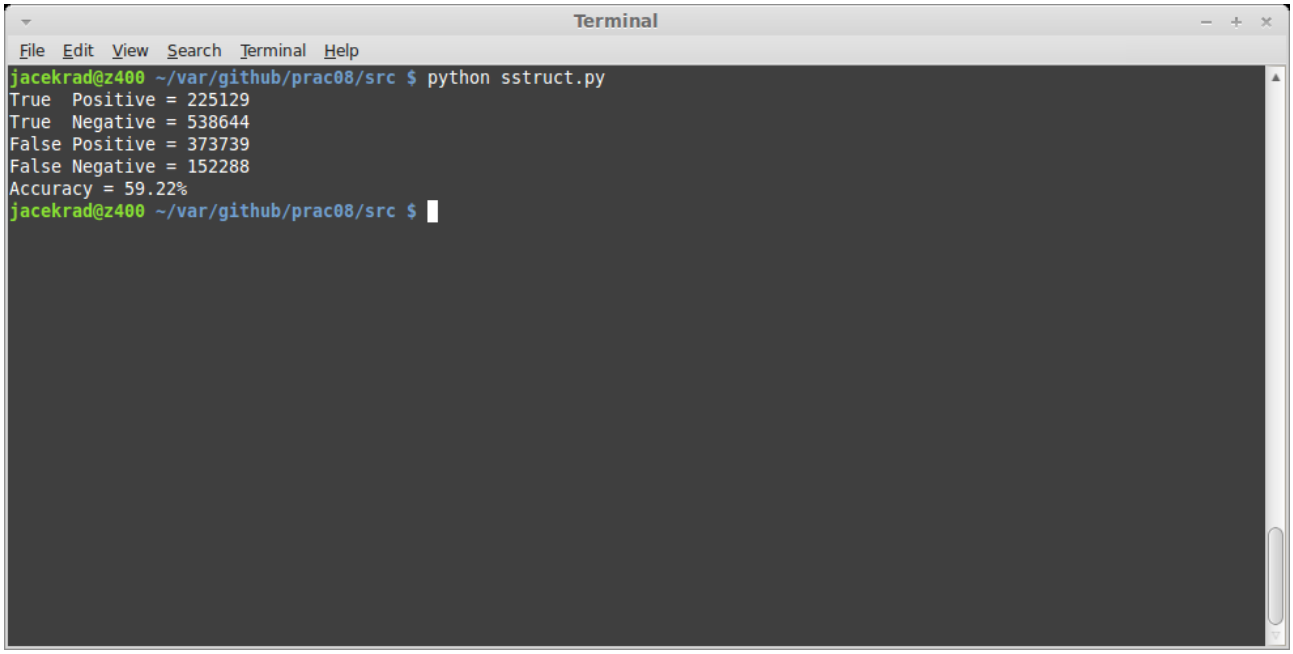
Baseline for a 3-class secondary structure (eg. Alpha helix, beta sheet and random coil) is the probability of random guessing a structure. Since we have 3 potential types our guess probability would be 1/3 or 33.33%. This means that if our prediction algorithm success rate should be compared to that of 33.33% baseline.

Question 8

The accuracies for alpha-helices and beta-sheets for the proteins and structures in prot2.fa and sstr3.fa have been calculated to be 59.22%. This is an improvement of 25.89% over the baseline (33.33%) obtained in question 7. Frequency distribution for true positive, true negative, false positive and false negative matches is shown in a graph below. Code is shown in Appandix A. NOTE the code includes the question 9 implementation so it is not the original code used for question 8.



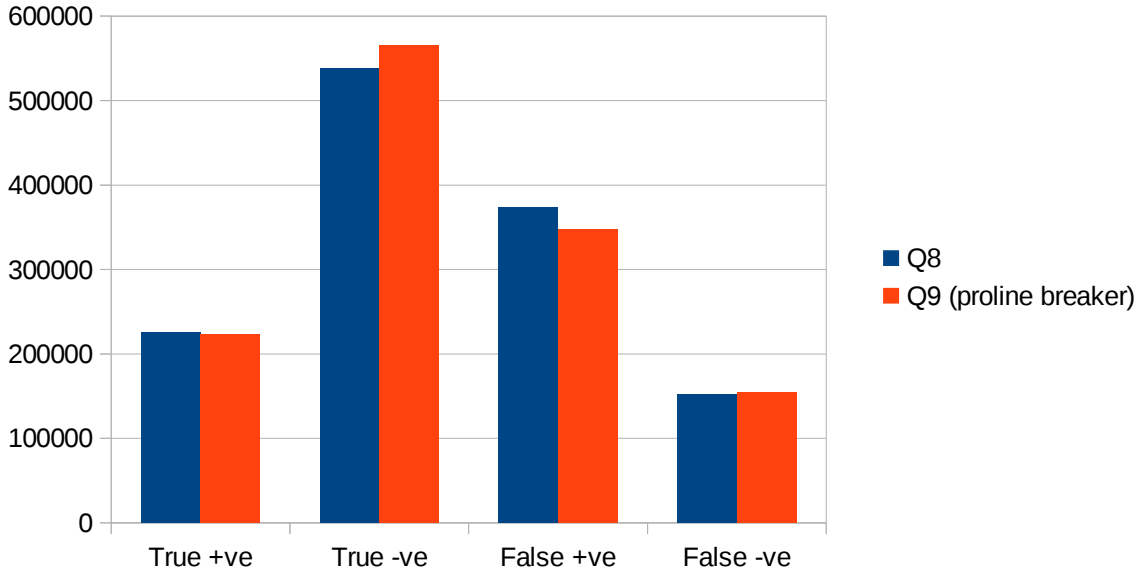
Execution:



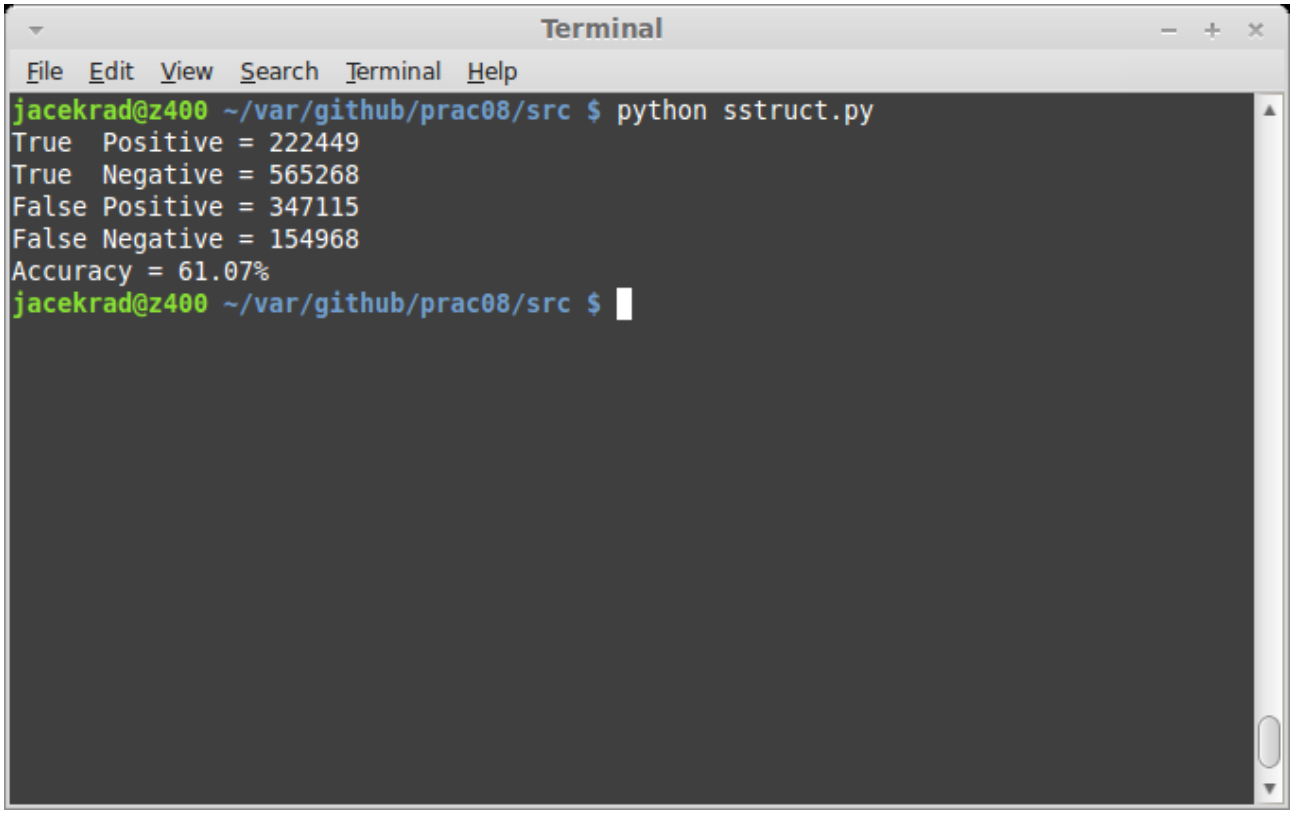
Question 9

The proline breaker functionality has been implemented (see appendix for full code listing) by creating a new version of extendDownstream function called extendDownstream9. This function accepts 2 additional parameters: boolean alpha and a list prolines. The proline breaker code is run only when alpha is set to True; i.e. we ignore it for beta-sheets. The prolines is a list of boolean values set true at the positions where prolines are detected in the sequence. This list has to be created prior to calling extendDownstream9. If a proline is detected whilst extending for alpha helices, current and next 5 locations are reset to False i.e. alpha helix is broken if there was one previously.

Running the new code gives us 61.07% accuracy which is 1.85% improvement over the results obtained in question 8. We can see (graph below) that there is a significant increase in true negative and reduction in false positive. Since the proline breaker code will create fewer alpha helix matches than the previous implementation these two changes are expected (assuming increase in accuracy which we did obtain).



Execution


Terminal
File Edit View Search Terminal Help
jacekrad@z400 ~/var/github/prac08/src $ python sstruct.py
True Positive = 222449
True Negative = 565268
False Positive = 347115
False Negative = 154968
Accuracy = 61.07%
jacekrad@z400 ~/var/github/prac08/src $

Appandix A – Code for Questions 8 & 9

Relevant changes are in bold

```
'''
Module sstruct -- methods for protein secondary structure
'''

import sequence
import symbol

cf_dict = {  # Chou-Fasman table
#     P(a), P(b), P(t),    f(i), f(i+1), f(i+2), f(i+3)
'A': (142, 83, 66, 0.060, 0.076, 0.035, 0.058),  # Alanine
'R': (98, 93, 95, 0.070, 0.106, 0.099, 0.085),  # Arginine
'N': (101, 54, 146, 0.147, 0.110, 0.179, 0.081),  # Aspartic Acid
'D': (67, 89, 156, 0.161, 0.083, 0.191, 0.091),  # Asparagine
'C': (70, 119, 119, 0.149, 0.050, 0.117, 0.128),  # Cysteine
'E': (151, 37, 74, 0.056, 0.060, 0.077, 0.064),  # Glutamic Acid
'Q': (111, 110, 98, 0.074, 0.098, 0.037, 0.098),  # Glutamine
'G': (57, 75, 156, 0.102, 0.085, 0.190, 0.152),  # Glycine
'H': (100, 87, 95, 0.140, 0.047, 0.093, 0.054),  # Histidine
'I': (108, 160, 47, 0.043, 0.034, 0.013, 0.056),  # Isoleucine
'L': (121, 130, 59, 0.061, 0.025, 0.036, 0.070),  # Leucine
'K': (114, 74, 101, 0.055, 0.115, 0.072, 0.095),  # Lysine
'M': (145, 105, 60, 0.068, 0.082, 0.014, 0.055),  # Methionine
'F': (113, 138, 60, 0.059, 0.041, 0.065, 0.065),  # Phenylalanine
'P': (57, 55, 152, 0.102, 0.301, 0.034, 0.068),  # Proline
'S': (77, 75, 143, 0.120, 0.139, 0.125, 0.106),  # Serine
'T': (83, 119, 96, 0.086, 0.108, 0.065, 0.079),  # Threonine
'W': (108, 137, 96, 0.077, 0.013, 0.064, 0.167),  # Tryptophan
'Y': (69, 147, 114, 0.082, 0.065, 0.114, 0.125),  # Tyrosine
'V': (106, 170, 50, 0.062, 0.048, 0.028, 0.053),  # Valine
'Y': (69, 147, 114, 0.082, 0.065, 0.114, 0.125),  # Tyrosine
'V': (106, 170, 50, 0.062, 0.048, 0.028, 0.053), }  # Valine

prot_alpha = symbol.Protein_Alphabet
sstr_alpha = symbol.DSSP3_Alphabet

def makesstr(seq, sym='*', gap='-'):
    """ Create a string from a list of booleans (seq) that indicate with sym what elements are true.
        gap is used for elements that are false.
    """
    sstr = ''
    for yes in seq:
        if yes:
            sstr += sym
        else:
            sstr += gap
    return sstr

def markCountAbove(scores, width=6, call_cnt=4):
    """ Create a list of booleans that mark all positions within a window
        of specified width that have scores above 100.
        scores: a list of scores (one for each position in sequence)
        width: width of window
        call_cnt: required number of positions with score 100 or more
        return: list of "calls" (positions in windows with at least call_cnt)
    """
    above = [False for _ in range(len(scores))]
    cnt = 0  # keep track of how many in the current window that are > 100
    for i in range(len(scores)):
        if scores[i] > 100: cnt += 1
        if i >= width:
            if scores[i - width] > 100: cnt -= 1
        if cnt >= call_cnt:
            for j in range(max(0, i - width + 1), i + 1):
                above[j] = True
    return above

def markAvgAbove(scores, width=4, call_avg=100.0):
    """ Create a list of booleans that mark all positions within a window of specified width
        that have an average score above specified call_avg.
    """
    above = [False for _ in range(len(scores))]
    sum = 0.0  #
    for i in range(len(scores)):
        sum += scores[i]
        if i >= width:  #
            sum -= scores[i - width]
        if sum >= call_avg * width:
            for j in range(max(0, i - width + 1), i + 1):
                above[j] = True
    return above
```

```
def extendDownstream9(scores, calls, prolines, width=4, alpha=True):
    """ Question 9 implementation
        Create a list of booleans that mark all positions that are contained
        in supplied calls list AND extend this list downstream containing a
        specified width average of 100.
        prolines: the list of prolines
        alpha: True iff we are extending for alpha helices
    """
    sum = 0.0
    order = range(0, len(calls) - 1, +1)  # we are extending calls downstream
    cnt = 0
    for i in order:  # extend to the right
        # proline handling
        # if we are dealing with alpha helix detection and proline is detected
        # 5 residues ahead then we clear all alpha helices between here and the
        # the proline; ie. set to False
        if alpha and i < (len(calls) - 5) and prolines[i + 5]:
            for j in range(0, 6): calls[i + j] = False
            cnt = 0
            sum = 0.0
        if calls[i]:  # to extend a call is required in the first place
            cnt += 1
            sum += scores[i]  # keep a sum to be able to average
            if cnt >= width:  # only average over a width
                sum -= scores[i - width + 1] # remove score from beyond the tail of the window
            if not calls[i + 1] and sum + scores[i + 1] > width * 100:  # check
                calls[i + 1] = True
        else:  # no call, reset sum
            cnt = 0
            sum = 0.0
    return calls

def extendDownstream(scores, calls, width=4):
    """ Create a list of booleans that mark all positions that are contained
        in supplied calls list AND extend this list downstream containing a
        specified width average of 100.
        prolines is the list of prolines
    """
    sum = 0.0
    order = range(0, len(calls) - 1, +1)  # we are extending calls downstream
    cnt = 0
    for i in order:  # extend to the right
        if calls[i]:  # to extend a call is required in the first place
            cnt += 1
            sum += scores[i]  # keep a sum to be able to average
            if cnt >= width:  # only average over a width
                sum -= scores[i - width + 1] # remove score from beyond the tail of the window
            if not calls[i + 1] and sum + scores[i + 1] > width * 100:  # check
                calls[i + 1] = True
        else:  # no call, reset sum
            cnt = 0
            sum = 0.0
    return calls


def extendUpstream(scores, calls, width=4):
    """ Create a list of booleans that mark all positions that are contained in supplied calls list
        AND extend this list upstream containing a specified width average of 100.
    """
    sum = 0.0
    order = range(len(calls) - 1, 0, -1)  # we are extending calls upstream/to-the-left
    cnt = 0
    for i in order:  # extend to the right
        if calls[i]:  # a requirement to extend is to have a call in the first place
            cnt += 1
            sum += scores[i]  # keep a sum to be able to average
            if cnt >= width:  # only average over a width
                sum -= scores[i + width - 1]
            if not calls[i - 1] and sum + scores[i - 1] > width * 100:  # check average
                calls[i - 1] = True
        else:  # no call, reset sum
            cnt = 0
            sum = 0.0
    return calls

def calcRegionAverage(scores, calls):
    """ Determine for each position in a calls list the average score over the region
        in which it is contained.
    """
    region_avg = []
    sum = 0.0
    cnt = 0
    # First determine the average for each region
    for i in range(len(scores)):  # go through each position
        if calls[i]:  # position is part of a "called" region
            sum += scores[i]  # add the score of that position to the average
            cnt += 1  # keep track of the number of positions in the region
        else:  # we are outside a "called" region
            if cnt > 0:  # if it is the first AFTER a called region
                region_avg.append(sum / cnt)  # save the average
            sum = 0.0  # reset average
            cnt = 0
    if cnt > 0:  # if it is the first AFTER a called region
        region_avg.append(sum / cnt)  # save the average
    # with all averages known, we'll populate the sequence of "averages"
    region = 0
    pos_avg = []
    cnt = 0
    for i in range(len(scores)):
        if calls[i]:
```

```
            pos_avg.append(region_avg[region])
            cnt += 1
        else:
            pos_avg.append(0)
            if cnt > 0:
                region += 1
            cnt = 0
    return pos_avg

def checkSupport(calls, diff):
    """ Create a list of booleans indicating if each true position is supported
        by a positive score """
    supported = []
    for i in range(len(calls)):  # go through each position
        supported.append(calls[i] and diff[i] > 0)
    return supported

def getScores(seq, index=0):
    """ Create a score list for a sequence by referencing the Chou-Fasman table.
    """
    return [cf_dict[s.upper()][index] for s in seq]


""" ----------------------------------------
Below is test code
---------------------------------------- """

# Read some protein sequence data
prot = sequence.readFastaFile('prot2.fa', symbol.Protein_Alphabet)
# read the secondary structure data for the proteins above (indices should agree)
sstr = sequence.readFastaFile('sstr3.fa', symbol.DSSP3_Alphabet)

#prot = [sequence.Sequence('PNKRKGFSEGLWEIENNPTVKASGY', symbol.Protein_Alphabet, '2NLU_r76')]
#sstr = [sequence.Sequence('CCCCHHHHHHHHHHCCCCCCCCC', symbol.DSSP3_Alphabet, '2NLU_s76')]

#prot = [sequence.Sequence("SEQSICQARAAVMVYDDANKKWVPAGGSTGFSRVHIYHHTGNNTFRVVGRKIQDHQVVIN" +\
#    "CAIPKGLKYNQATQTFHQWRDARQVYGLNFGSKEDANVFASAMMHALEVLN", symbol.Protein_Alphabet, "1EVH")]
#sstr = [sequence.Sequence("CEEEEEEEEEEEEEECCCCEEEEHHHCCCCEEEEEEEECCCCEEEEEEEECCCCCEEEEEEE" +\
#    "CCCCCCECCCCCEEEEECCCCEEEEEECCHHHHHHHHHHHHHHHHHC", symbol.DSSP3_Alphabet, "1EVH")]

tp = 0  # number of true positives (correctly identified calls)
tn = 0  # number of true negatives (correctly missed no-calls)
fp = 0  # number of false positives (incorrectly identified no-calls)
fn = 0  # number of false negatives (incorrectly missed calls)

for index in range(len(prot)):

    myprot = prot[index]
    mysstr = sstr[index]
    myalpha = [sym == 'H' for sym in sstr[index]]
    mybeta = [sym == 'E' for sym in sstr[index]]
    prolines = [sym == 'P' for sym in prot[index]]

    """
     1. Assign all of the residues in the peptide the appropriate set of parameters.
    """
    alpha = getScores(myprot, 0)
    beta = getScores(myprot, 1)
    turn = getScores(myprot, 2)

    """
     2. Scan through the peptide and identify regions where 4 out of 6 contiguous residues have P(a-helix) > 100.
        That region is declared an alpha-helix.
        Extend the helix in both directions until a set of four contiguous residues that have an average P(a-helix) < 100 is
reached.
        That is declared the end of the helix.
        If the segment defined by this procedure is longer than 5 residues and the average P(a-helix) > P(b-sheet) for that
segment,
        the segment can be assigned as a helix.

     3. Repeat this procedure to locate all of the helical regions in the sequence.
    """
    calls_a1 = markCountAbove(alpha, width=6, call_cnt=4)
    calls_a2 = extendDownstream9(alpha, calls_a1, prolines, width=4, alpha=True)
    calls_a3 = extendUpstream(alpha, calls_a2, width=4)

#    print calls_a1
#    print calls_a2
#    print calls_a3

    """
     4. Scan through the peptide and identify a region where 3 out of 5 of the residues have a value of P(b-sheet) > 100.
        That region is declared as a beta-sheet.
        Extend the sheet in both directions until a set of four contiguous residues that have an average P(b-sheet) < 100 is
reached.
        That is declared the end of the beta-sheet.
        Any segment of the region located by this procedure is assigned as a beta-sheet
        if the average P(b-sheet) > 105 and the average P(b-sheet) > P(a-helix) for that region.
    """
    calls_b1 = markCountAbove(beta, width=5, call_cnt=3)
    calls_b2 = extendDownstream9(beta, calls_b1, prolines, width=4, alpha=False)
    calls_b3 = extendUpstream(beta, calls_b2, width=4)

    """
     5. Any region containing overlapping alpha-helical and beta-sheet assignments are taken to be helical
        if the average P(a-helix) > P(b-sheet) for that region.
        It is a beta sheet if the average P(b-sheet) > P(a-helix) for that region.
    """
    avg_a = calcRegionAverage(alpha, calls_a3)
    avg_b = calcRegionAverage(beta, calls_b3)
```

```
    diff_a = [avg_a[i] - avg_b[i] for i in range(len(avg_a))]
    diff_b = [avg_b[i] - avg_a[i] for i in range(len(avg_a))]
    calls_a4 = checkSupport(calls_a3, diff_a)
    calls_b4 = checkSupport(calls_b3, diff_b)

    # For accuracy calculation, Exercise 6 and 8
    i = 0
    for call in myalpha:
        if call == True:
            if calls_a4[i] == True:
                tp += 1
            else:
                fn += 1
        else:
            if calls_a4[i] == False:
                tn += 1
            else:
                fp += 1
        i += 1

    i = 0
    for call in mybeta:
        if call == True:
            if calls_b4[i] == True:
                tp += 1
            else:
                fn += 1
        else:
            if calls_b4[i] == False:
                tn += 1
            else:
                fp += 1
        i += 1

####### Accuracy calculations (Q8) #######
print "True  Positive = %d" % tp
print "True  Negative = %d" % tn
print "False Positive = %d" % fp
print "False Negative = %d" % fn
print "Accuracy = %.2f%%" % (float(tp + tn) * 100 / (tp + tn + fp + fn))
```