weeks 3-5

March 16, 2021

[1]:
```python
import numpy as np
```

1 Task 1: Write a program to implement Needleman-Wunsch for proteins

The np library function is used to parse the text file containing a matrix of numbers into a numpy nd array, and then a dictionary is created mapping the letters of the blosum table in the correct order to indices so as to be able to access scores at an index of the blosum50 table corresponding to 2 chars with the get_blosum_score function:

[503]:
```python
#the log odds of one amino acid being substituted by another
blosum50 = np.loadtxt("blosum50.txt", dtype = 'i')
```

[32]:
```python
#map the characters with their indices
blosum_index_map = dict(zip(['A','R','N','D','C','Q','E','G','H','I','L','K',
                'M','F','P','S','T','W','Y','V'],
                  [i for i in range(len(blosum50))]))

#given the bases to consider swapping, get the blosum log probability score of␣
 ↪swapping them
#by looking up the corresponding index of the chars and then accessing that␣
 ↪index of the matrix
def get_blosum_score(char1, char2):
    i = blosum_index_map[char1]
    j = blosum_index_map[char2]
    return blosum50[i][j]
```

1.1 Needleman-Wunsch implementation

For implementing dynamic programming, I believe the most elegant technique is to store "pointers" directly in the elements of the matrix as was discussed in the lecture, so the "direction" that the current cell "came from" whose value results in the maximum value is marked with either "n", "w" or "nw" corresponing to north, west or north-west.

By doing this, once our "forward step" is finished, in our "backward step" we simply start from the bottom right corner of the matrix and can retrace this path direction: when moving "north west" no characters were substitued or eliminated so the character at the relative i/j indices in both strings stays the same, if we move “west” then we know a character was eliminated from the string along the “left” of the matrix so we append a dash to represent this, similarly if we move “north” then we know a character was eliminated from the string along the “top” of the matrix so we also apend a dash to represent this.

Once we reach the top left of the matrix our backward step is complete and the function prints the resulting strings that show eliminations/substitutions.

[508]:
```python
#assumes that the blosum50 field is accessible in its environment
def needleman_wunsch(str2, str1):

    #our penalty score
    d = -8

    #str1 is the chars on the "left" of the matrix
    #str2 is the chars along the "top" of the matrix

    height = len(str1) + 1
    width = len(str2) + 1

    #explicitly object data type so that we can store tuples
    matrix = np.zeros((height, width), dtype = "object")

    #initialization step, mark the cells at the sides of the matrix
    #as incrementing by the penalty, and mark the "arrow" directions all as␣
 ↪north/west
    count = 0
    for i in range(width):
        matrix[0][i] = (count,"w")
        count += d
    count = 0
    for i in range(height):
        matrix[i][0] = (count, "n")
        count +=d

    for i in range(1, height):

        for j in range(1, width):

            char1 = str1[i - 1]
            char2 = str2[j - 1]
            score = get_blosum_score(char2, char1)

            #direct implementation of the dynamic programming step
            matrix[i][j] = max([
                (matrix[i-1][j-1][0] + score,"nw"),
                (matrix[i-1][j][0] + d,"n"),
                (matrix[i][j-1][0] + d,"w")
            ], key = lambda tup: tup[0])

    #F matrix has been generated, now need to perform backwards algorithm
    #start at the bottom right of the matrix
    i = height - 1
    j = width - 1

    #path of cell transitions we will build up
    path = []

    #the 2 string substitution visualizations we will build up
    str1sub = ""
    str2sub = ""

    while True:

        #end condition, we have retraced the path
        if i == 0 and j == 0:
            break

        #the direction we need to go in
        dr = matrix[i][j][1]

        #chars stayed the same
        if dr == "nw":
            path.append((i,j))
            str1sub += str1[i - 1]
            str2sub += str2[j - 1]
            i -= 1
            j -= 1
            continue

        #eliminate char from str2
        if dr == "n":
            path.append((i,j))
            str1sub += str1[i - 1]
            str2sub += '-'
            i -= 1
            continue

        #eliminate char from str1
        if dr == "w":
            path.append((i,j))
            str1sub += '-'
            str2sub += str2[j - 1]
            j -= 1
    path.append((0,0))

    print("\nIndices of path found starting from bottom right:\n\n" + str(path)␣
↪+ "\n")

    print("\nString comparison, dashes represent if a character was eliminated:
↪\n\n")
    #reverse the strings as they have been built up backwards
    print(str2sub[::-1] + "\n")
    print(str1sub[::-1] + "\n\n")
```

[509]:
```
needleman_wunsch("HEAGAWGHEE", "PAWHEAE")
```

```
Indices of path found starting from bottom right:

[(7, 10), (6, 9), (5, 9), (4, 8), (3, 7), (3, 6), (2, 5), (1, 4), (1, 3), (0,
2), (0, 1), (0, 0)]


String comparison, dashes represent if a character was eliminated:


HEAGAWGHE-E

--P-AW-HEAE
```

[510]:
```
needleman_wunsch("SALPQPTTPVSSFTSGSMLGRTDTALTNTYSAL",␣
↪"PSPTMEAVTSVEASTASHPHSTSSYFATTYYHLY")
```

```
Indices of path found starting from bottom right:

[(34, 33), (33, 33), (32, 32), (31, 31), (30, 30), (29, 29), (28, 28), (27, 27),
(26, 26), (25, 25), (24, 24), (23, 23), (22, 22), (21, 21), (20, 20), (19, 19),
(18, 18), (17, 17), (16, 16), (15, 15), (14, 14), (13, 13), (13, 12), (12, 11),
(11, 10), (10, 9), (9, 8), (8, 7), (7, 6), (6, 5), (5, 4), (4, 3), (3, 2), (2,
1), (1, 0), (0, 0)]


String comparison, dashes represent if a character was eliminated:


-SALPQPTTPVSSFTSGSMLGRTDTALTNTYSAL-
```

```
PSPTMEAVTSVEA-STASHPHSTSSYFATTYYHLY
```

2 Task 2: Smith-Waterman implementation

As was dicussed in lectures, the Smith-Waterman algorithm is nearly identical except we now add an extra constraint to ensure no cells in our F matrix can be negative. When generating the backwards path, instead of starting from the bottom right of the matrix we now instead start from the maximum cell of the matrix so the location of this is stored in a variable such that once our "forward step" is finished, we know exactly which cell to start from and follow our direction strings stored in the tuple at each cell until we reach a cell containing a 0.

[511]:
```
#assumes that the blosum50 field is accessible in its environment
def smith_waterman(str2, str1):

    #our penalty score
    d = -8

    #str1 is the chars along the "left" of the matrix
    #str2 is the chars along the "right" of the matrix

    #initialize an empty matrix
    height = len(str1) + 1
    width = len(str2) + 1

    #explicitly object data type so that we can store tuples
    matrix = np.zeros((height, width), dtype = "object")

    #the cells at the edges are now marked as zeroes
    for i in range(width):
        matrix[0][i] = (0,"zero")
    for i in range(height):
        matrix[i][0] = (0, "zero")

    #keep track of the max seen
    max_seen = 0
    #also keep track of its index
    max_index = (0,0)

    for i in range(1, height):

        for j in range(1, width):

            char1 = str1[i - 1]
            char2 = str2[j - 1]
            score = get_blosum_score(char2, char1)
            #direct implementation of the dynamic programming step
            matrix[i][j] = max([
                #the new case added for smith waterman
                (0, "zero"),
                #each cell contains a tuple of score and direction so extract␣
↪the score
                (matrix[i-1][j-1][0] + score,"nw"),
                (matrix[i-1][j][0] + d,"n"),
                (matrix[i][j-1][0] + d,"w")
            ], key = lambda tup: tup[0])

            #keep track of the biggest cell value found as this
            #will now be our starting point
            if(matrix[i][j][0] > max_seen):
                max_seen = matrix[i][j][0]
                max_index = (i,j)

    #we are now starting at the index containing the max value
    #instead of the bottom right of the matrix
    i = max_index[0]
    j = max_index[1]

    #path of cell transitions we will build up
    path = []

    #the 2 string substitution visualizations we will build up
    str1sub = ""
    str2sub = ""

    while True:

        #the direction we need to go in (or a zero)
        dr = matrix[i][j][1]

        #termination condition, the path stops at 0
        if dr == "zero":
            break

        #chars stayed the same
        if dr == "nw":

            path.append((i,j))
            str1sub += str1[i - 1]
            str2sub += str2[j - 1]

            i -= 1
            j -= 1
            continue

        #eliminate char from the string along the "top" of the matrix
        if dr == "n":
            path.append((i,j))
            str1sub += str1[i - 1]
            str2sub += '-'
            i -= 1
            continue

        #eliminate char from the string along the "left" of the matrix
        if dr == "w":
            path.append((i,j))
            str1sub += '-'
            str2sub += str2[j - 1]
            j -= 1

    #we finished at indices i,j therefore it must be a 0 and the final
    #node in the path, so add it to the path
    path.append((i,j))

    print("\nIndices of path found starting from maximum:\n\n" + str(path) +␣
↪"\n")

    print("\nLocal match substituted strings, dashes represent if a character␣
↪was eliminated:\n")

    #reverse the strings as they have been built up backwards
    print(str2sub[::-1] + "\n")
    print(str1sub[::-1] + "\n\n")
```

[512]:
```
smith_waterman("HEAGAWGHEE", "PAWHEAE")
```

```
Indices of path found starting from maximum:

[(5, 9), (4, 8), (3, 7), (3, 6), (2, 5), (1, 4)]


Local match substituted strings, dashes represent if a character was eliminated:

AWGHE

AW-HE
```

[513]:
```
smith_waterman(
    "MQNSHSGVNQLGGVFVNGRPLPDSTRQKIVELAHSGARPCDISRILQVSNGCVSKILGRY",
    "TDDECHSGVNQLGGVFVGGRPLPDSTRQKIVELAHSGARPCDISRI"
)
```

```
Indices of path found starting from maximum:

[(46, 45), (45, 44), (44, 43), (43, 42), (42, 41), (41, 40), (40, 39), (39, 38),
(38, 37), (37, 36), (36, 35), (35, 34), (34, 33), (33, 32), (32, 31), (31, 30),
(30, 29), (29, 28), (28, 27), (27, 26), (26, 25), (25, 24), (24, 23), (23, 22),
(22, 21), (21, 20), (20, 19), (19, 18), (18, 17), (17, 16), (16, 15), (15, 14),
(14, 13), (13, 12), (12, 11), (11, 10), (10, 9), (9, 8), (8, 7), (7, 6), (6, 5),
(5, 4)]


Local match substituted strings, dashes represent if a character was eliminated:

HSGVNQLGGVFVNGRPLPDSTRQKIVELAHSGARPCDISRI

HSGVNQLGGVFVGGRPLPDSTRQKIVELAHSGARPCDISRI
```

3 Task 3: Testing the BLAST algorithm

These are my results of the BLAST sequence comparison for the Pax6 mouse protein and eyeless protein for fruit flies, alignment is apparent in query 5:

[506]:
```
alignment = open("3CYASRFB114-Alignment.txt", 'r').read()
print(alignment)
```

```
RID: 3CYASRFB114
Job Title: sp|P63015|PAX6_MOUSE Paired box protein
Program: BLASTP
Subject:tr|O96791|O96791_DROME Eyeless protein (Fragment) OS=Drosophila
melanogaster OX=7227 GN=ey PE=4 SV=1 ID: lcl|Query_38263(amino acid) Length: 101
Query #1: sp|P63015|PAX6_MOUSE Paired box protein Pax-6 OS=Mus musculus OX=10090
GN=Pax6 PE=1 SV=1 Query ID: lcl|Query_38261 Length: 422

Sequences producing significant alignments:
                                                                   Scientific
Common                       Max    Total Query   E   Per.   Acc.
Description                                                        Name
Name            Taxid        Score  Score cover Value Ident  Len       Accession
<tr id="dtr_<@dflnFrm_id@>" ind="<@dfln_blast_rank@>" class="<@trtp@> dflLnk" >
<td class="l c0"><span class="ind"><@dfln_blast_rank@></span><input
```

```
type="checkbox" id="chk_<@dfln_blast_rank@>" class="cb" name="getSeqGi"
value="<@dfln_id@>" onclick="configDescrLinks(event,this)" /></td>
<td class="lim l c2">
<div class="lim n">
<a title="Go to alignment for tr|O96791|O96791_DROME Eyeless protein (Fragment)
OS=Drosophil… " class="deflnDesc" hsp="<@dfln_hspnum@>" len="<@dfln_alnLen@>"
ind="<@dfln_blast_rank@>" accs="<@dflnAccs@>" seqFSTA="<@dflnFASTA_id@>"
gi="<@dfln_id@>"  seqID="<@dflnFrm_id@>" id="deflnDesc_<@dfln_blast_rank@>"
onclick="DisplayAlignFromDescription(this);"
href="#alnHdr_<@dflnFrm_id@>">tr|O96791|O96791_DROME Eyeless protein (Fragment)
OS=Drosophil… </a>
</div>
</td>
<td class="c3">-246.0 </td>
<td class="c7">9.00   %</td>
<td class="c1 l lim">
Query_38263
</td>
</tr>

Alignments:
>tr|O96791|O96791_DROME Eyeless protein (Fragment) OS=Drosophila melanogaster
OX=7227 GN=ey PE=4 SV=1
Sequence ID: Query_38263 Length: 101
Range 1: 1 to 101

<table class="alnParams">
<caption class="hdnHeader">Alignment statistics for match #1</caption>
<tr>
<th>NW Score</th><th>Identities</th><th>Positives</th><th>Gaps</th><th
class="aln_frame ">Frame</th>
</tr>
<tr>
<td>-246</td>
<td>42/478(9%)</td>
<td>43/478(8%)</td>
<td>429/478(89%)</td>
<td class="aln_frame "></td>
</tr>
</table>
Query  1        MQN---------------------------------------------------S  4
                M+N
Sbjct  1    NVIAMRNLPCLGTAGGSGLGGIAGKPSPTMEAVEASTASHPHSTSSYFATTYYHLTDDEC  60

Query  5    HSGVNQLGGVFVNGRPLPDSTRQKIVELAHSGARPCDISRILQVSNGCVSKILGRYYETG  64
            HSGVNQLGGVFV GRPLPDSTRQKIVELAHSGARPCDISRI
Sbjct  61   HSGVNQLGGVFVGGRPLPDSTRQKIVELAHSGARPCDISRI                    101
```

```
Query  65   SIRPRAIGGSKPRVATPEVVSKIAQYKRECPSIFAWEIRDRLLSEGVCTNDNIPSVSSIN  124


Query  125  RVLRNLASEKQQMGADGMYDKLRMLNGQTGSWGTRPGWYPGTSVPGQPTQDGCQQQEGGG  184


Query  185  ENTNSISSNGEDSDEAQMRLQLKRKLQRNRTSFTQEQIEALEKEFERTHYPDVFARERLA  244


Query  245  AKIDLPEARIQVWFSNRRAKWRREEKLRNQRRQASNTPSHIPISSSFSTSVYQPIPQPTT  304


Query  305  PVSSFTSGSMLGRTDTALTNTYSALPPMPSFTMANNLPMQPPVPSQTSSYSCMLPTSPSV  364


Query  365  NGRSYDTYTPPHMQTHMNSQPMGTSGTTSTGLISPGVSVPVQVPGSEPDMSQYWPRLQ  422
```

4 Task 4: Programming a HMM model to generate CG rich regions and AT rich regions

I found the most elegant way to model the HMM was using mutual recursion, where the 2 functions calling one another represents a change in state, and when n_remaining reaches 0 then the recursive stack that has been built up is returned (which will ultimately be a list of tuples of the string sections generated by each state, paired with a string stating the state that generated the section)

[431]:
```python
def run_HMM(length):

    #equal probability of starting in either the AT state or the CG state
    if(np.random.choice([True, False], p = [0.5, 0.5])):
        return generate_CGrich_region(length, [])
    else:
        return generate_ATrich_region(length, [])
```

[432]:
```python
def generate_ATrich_region(n_remaining, accumulator):
    #emission probabilities for each of the nucleotide bases respectively
    #when in this state
    emission_probs = [0.2698, 0.3237, 0.2080, 0.1985]

    #a continuous section that will be built up whilst in this state
    section_str = ""

    #loop until we change into the other state by chance
    while True:

        #whole recursive call finished and can be returned
        if n_remaining == 0:
            accumulator.append((section_str, "AT_rich"))
            return accumulator

        section_str += np.random.choice(nuc_bases, p = emission_probs)
        n_remaining -= 1

        #on each iteration we may by chance finish in this state
        #and go into the other state
        if np.random.choice([True, False], p = [0.0002, 0.9998]):
            break

    accumulator.append((section_str, "AT_rich"))
    return generate_CGrich_region(n_remaining, accumulator)
```

[433]:
```
def generate_CGrich_region(n_remaining, accumulator):

    #emission probabilities for each of the nucleotide bases respectively
    #when in this state
    emission_probs = [0.2459, 0.2079, 0.2478, 0.2984]

    #a continuous section that will be built up whilst in this state
    section_str = ""

    #loop until we change into the other state by chance
    while True:

        #whole recursive call finished and can be returned
        if n_remaining == 0:
            accumulator.append((section_str, "CG_rich"))
            return accumulator

        section_str += np.random.choice(nuc_bases, p = emission_probs)
        n_remaining -= 1

        #on each iteration we may by chance finish in this state
        #and go into the other state
        if np.random.choice([True, False], p = [0.0002, 0.9998]):
            break

    accumulator.append((section_str, "CG_rich"))
    return generate_ATrich_region(n_remaining, accumulator)
```

Let's generate a DNA sequence of length say, 30000 in order to test the Viterbi implementation:

[461]:
```
tups = run_HMM(30000)
tups

#Concatenate the strings generated in each section into one normal string in␣
 ↪order to test our Viterbi decoder
orig_sequence = "".join(list(map(lambda x: x[0], tups)))
```

5 Task 5: Viterbi algorithm implementation

The viterbi algorithm is divided into forwards and backwards parts, in the "forwards" part 2 lists of cumulative likelihoods are built up for both the AT_rich and CG_rich states using the log version of the formula from the slides to prevent underflowing, taking into account the state transition probabilities and emission probabilities within each state on each new "column" generated, keeping a pointer to the state that the current node came from.

The backwards algorithm simply retraces this path from the end of the two lists of cumulative likelihoods, starting with the node that has a greater cumulative likelihood and building up the decoded sequence from here by following the pointer directions.

Mutual recursion again turned out to be an elegant solution for modelling switching between the AT/CG states in the backwards algorithm.

[463]:
```
#takes as input a DNA sequence string and returns the estimated sections of the␣
 ↪sequence that are
#AT rich regions or CT rich regions, represented by the characters 'A' and 'C'␣
 ↪respectively
def viterbi_decode(sequence_string):
    CG_cumulative_probs, AT_cumulative_probs = viterbi_forwards(sequence_string)
    return viterbi_backwards(CG_cumulative_probs, AT_cumulative_probs)
```

[464]:
```
#takes as input the observed sequence string and returns 2 lists of cumulative␣
 ↪likelihoods for each of the
# "nodes" in the AT/CT region "rows" in the viterbi diagram
def viterbi_forwards(sequence_string):

    #nucleotide base symbol emission probabilities
    AT_emission_probs = {'A':0.2698,'T':0.3237,'C':0.2080,'G':0.1985}
    CG_emission_probs = {'A':0.2459,'T':0.2079,'C':0.2478,'G':0.2984}
    #state transition probabilities
    AT_stay_same = 0.9998
    AT_2_CG = 0.0002
    CG_stay_same = 0.9997
    CG_2_AT = 0.0003

    #how I will model the "rows" in the viterbi algorithm diagram, using 2␣
 ↪seperate lists
    #i.e. the cumulative probabilities for being in either state
    #and generating a letter at some point in the sequence

    #lets model the cumulative probability of the start state as 0
    #and the probability of transitioning from it to either the AT rich
    #or CG rich state as 0.5 for both
    AT_cumulative_probs = [(0,"end")]
    CG_cumulative_probs = [(0,"end")]

    #the "forward" part of the viterbi algorithm
    for char in sequence_string:

        #the cumulative probabilities in the predecessor AT/CG state nodes
        last_AT_val = AT_cumulative_probs[-1][0]
        last_CG_val = CG_cumulative_probs[-1][0]

        AT_max_val, AT_state_came_from = max([(last_CG_val + np.
 ↪log(CG_2_AT),"from_other"),(last_AT_val + np.
 ↪log(AT_stay_same),"from_same")], key = lambda tup: tup[0])
        CG_max_val, CG_state_came_from = max([(last_AT_val + np.
 ↪log(AT_2_CG),"from_other"),(last_CG_val + np.
 ↪log(CG_stay_same),"from_same")], key = lambda tup: tup[0])

        AT_cumulative_probs.append((np.log(AT_emission_probs[char]) +␣
 ↪AT_max_val, AT_state_came_from))
        CG_cumulative_probs.append((np.log(CG_emission_probs[char]) +␣
 ↪CG_max_val, CG_state_came_from))

    return CG_cumulative_probs, AT_cumulative_probs
```

[465]:
```
#takes as argument the accumulator string that will be built up with As and Cs␣
 ↪to represent which state the
#node is in
def cg_state(i, acc_string, cg_cumulative_probs, at_cumulative_probs):
    while True:
        acc_string = "C" + acc_string
        if(i == 0): return acc_string
        if(not cg_cumulative_probs[i][1] == "from_same"):
            i -= 1
            break
        else:
            i -= 1
    return at_state(i, acc_string, cg_cumulative_probs, at_cumulative_probs)

def at_state(i, acc_string, cg_cumulative_probs, at_cumulative_probs):
    while True:
        acc_string = "A" + acc_string
        if(i == 0): return acc_string
        if(not at_cumulative_probs[i][1] == "from_same"):
            i -= 1
            break
        else:
            i -= 1
    return cg_state(i, acc_string, cg_cumulative_probs, at_cumulative_probs)
```

[466]:
```
def viterbi_backwards(cg_cumulative_probs, at_cumulative_probs):

    #the index we start from, both lists are of the same length
    index = len(cg_cumulative_probs) - 1

    #start in the state that has the greatest cumulative likelihood
    if(cg_cumulative_probs[-1][0] > at_cumulative_probs[-1][0]):
        return cg_state(index, "", cg_cumulative_probs, at_cumulative_probs)
    else:
        return at_state(index, "", cg_cumulative_probs, at_cumulative_probs)
```

[467]:
```
decoded_sections = viterbi_decode(orig_sequence)
```

5.1 Getting information about the length of each section

Let’s write a simple function that we can use for getting information about the length of each section, this iwll be useful for comparing the generated sections with the guessed section sizes returned by the viterbi decoder. Unlikee the HMM model which returns a list of tuples, the viterbi decoder returns one long string where ‘A’ represents a nucleotide within an AT-rich state and ‘C’ represents a nucleotide within a CG-rich state, so we will design it to handle both types of data structures.

[468]:
```
def get_section_info(input):

    if(type(input[0]) is tuple):
        return list(map(lambda tup: (len(tup[0]), tup[1]), input))
    else:
        i = 0
        sections = []
        while i < len(input):
            if input[i] == 'A':
                size = 0
                while i < len(input) and input[i] == 'A':
                    size += 1
                    i += 1
                sections.append((size, "AT_rich"))
                continue
            if input[i] == 'C':
                size = 0
                while i < len(input) and input[i] == 'C':
                    size += 1
                    i += 1
                sections.append((size, "CG_rich"))
        return sections
```

Now using this function let's compare the sizes of sections in the actual sequence that was generated with the sequence that has been decoded into estimated state sections:

[469]:
```
#The original tuples generated by the HMM
tup_info = get_section_info(tups)
tup_info
```

[469]:
```
[(2986, 'AT_rich'),
 (3199, 'CG_rich'),
 (1070, 'AT_rich'),
 (2635, 'CG_rich'),
 (5251, 'AT_rich'),
 (6793, 'CG_rich'),
 (6325, 'AT_rich'),
 (769, 'CG_rich'),
 (972, 'AT_rich')]
```

[470]:
```
decoded_info = get_section_info(decoded_sections)
decoded_info
```

[470]:
```
[(3017, 'AT_rich'),
 (3127, 'CG_rich'),
 (1111, 'AT_rich'),
 (2664, 'CG_rich'),
 (5222, 'AT_rich'),
 (6796, 'CG_rich'),
 (6324, 'AT_rich'),
 (761, 'CG_rich'),
 (979, 'AT_rich')]
```

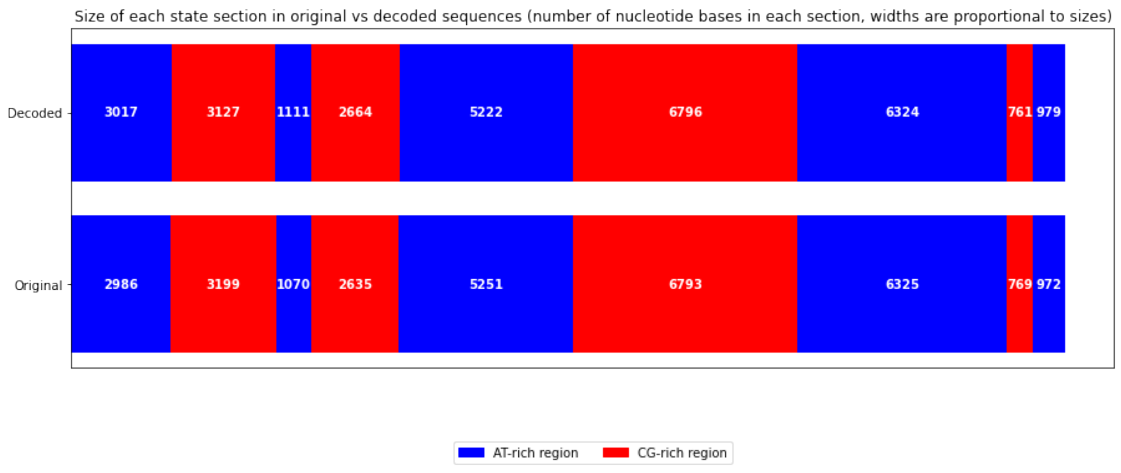
The sizes are remarkably similar so we can assume the Viterbi decoder is working! Now let's find a way of displaying this visually:

5.2 Visualizing the differences in state sections between the original sequence and the decoded sequence

[471]:
```
#HELPER FUNCTIONS FOR VISUALLY COMPARE TO USE

#generate a colour array corresponding to states
#Blue for AT rich state, Red for CG rich state
def gen_colour_arr(tups):
    states = list(map(lambda tup: tup[1], tups))
    colours = []
    for state in states:
        if state == "AT_rich":
            colours.append("blue")
        else:
            colours.append("red")
    return colours

#get the start x positions (left) for each of the rectangles
def extract_x_pos_arr(tups):
    y_pos_arr = [0]
    for i in range(len(tups) - 1):
        y_pos_arr.append(tups[i][0] + y_pos_arr[-1])
    return y_pos_arr
```

[484]:
```
#Takes in order the tuples and the decoded string generated
#and visually draws their differences
def visually_compare(original_tups, decoded_str):

    #list of states paired with lengths
    orig_tups = get_section_info(original_tups)
    decoded_tups = get_section_info(decoded_str)

    #list of widths, left x positions and colours for drawing the graphs
    orig_widths = list(map(lambda x: x[0], orig_tups))
    orig_lefts = extract_x_pos_arr(orig_tups)
    orig_colours = gen_colour_arr(orig_tups)
    decoded_widths = list(map(lambda x: x[0], decoded_tups))
    decoded_lefts = extract_x_pos_arr(decoded_tups)
    decoded_colours = gen_colour_arr(decoded_tups)

    #get the center lcoations of each rectangle so we can draw text on it
    orig_centers = []
    for i in range(len(orig_widths)):
        orig_centers.append((orig_widths[i] / 2) + orig_lefts[i])
    decoded_centers = []
    for i in range(len(decoded_widths)):
        decoded_centers.append((decoded_widths[i] / 2) + decoded_lefts[i])
    #draw our custom graph
    fig, ax = plt.subplots(figsize = (15,5))
    ax.barh(["Original"],
           orig_widths, left = orig_lefts, color = orig_colours)
    ax.barh(["Decoded"],
          decoded_widths, left = decoded_lefts, color = decoded_colours)

    #draw the state length in the sequence size on the rectangle
    for i in range(len(orig_centers)):
        ax.text(orig_centers[i], 0, orig_widths[i], ha = "center", va =␣
↪"center", color = "white"
               ,fontsize = "medium", fontweight = "bold")
    for i in range(len(decoded_centers)):
        ax.text(decoded_centers[i], 1, decoded_widths[i], ha = "center", va =␣
↪"center", color = "white",
                fontsize = "medium", fontweight = "bold")

    ax.legend(["AT-rich region", "CG-rich region"], bbox_to_anchor = (0.5,-0.
↪3), loc = "lower center", ncol = 2)
    ax.get_legend().legendHandles[0].set_color("blue")
    ax.get_legend().legendHandles[1].set_color("red")
    ax.title.set_text("Size of each state section in original vs decoded␣
↪sequences (number of nucleotide bases in each section, widths are␣
↪proportional to sizes)")
    ax.get_xaxis().set_visible(False)
```

[485]:
```
visually_compare(tups, decoded_sections)
```



6 Task 6: Running the Viterbi decoder on the lambda phage genome

[476]:
```
phage_lambda_seq = open("phaseLambda.fasta", 'r').read().replace('\n', '')
```

[477]:
```
decoded_phage = viterbi_decode(phage_lambda_seq)
```

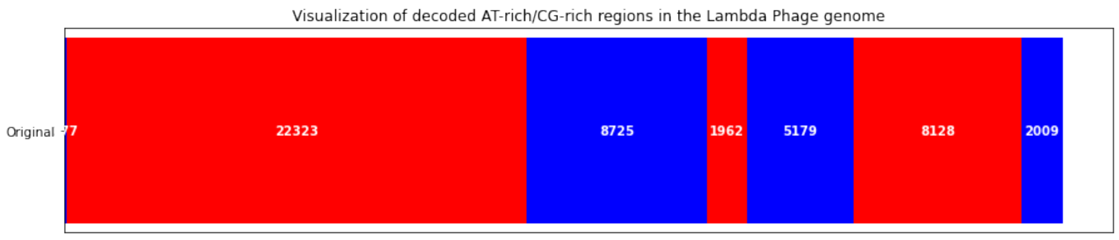
[478]:
```
phage_tups = get_section_info(decoded)
```

[479]:
```
#The predicted state section sizes in the phage lambda sequence
phage_tups
```

[479]:
```
[(177, 'AT_rich'),
 (22323, 'CG_rich'),
 (8725, 'AT_rich'),
 (1962, 'CG_rich'),
 (5179, 'AT_rich'),
 (8128, 'CG_rich'),
 (2009, 'AT_rich')]
```

Now, let's visualize the sections of the lambda phage as we did before when comparing our generated sequence and it's decoding:

[507]:
```
phage_widths = list(map(lambda x: x[0], phage_tups))
phage_lefts = extract_x_pos_arr(phage_tups)
phage_colors = gen_colour_arr(phage_tups)
phage_centers = []
for i in range(len(phage_widths)):
    phage_centers.append((phage_widths[i] / 2) + phage_lefts[i])
fig, ax = plt.subplots(figsize = (15,3))
ax.barh(["Original"],
       phage_widths, left = phage_lefts, color = phage_colors)
for i in range(len(phage_centers)):
        ax.text(phage_centers[i], 0, phage_widths[i], ha = "center", va =␣
 ↪"center", color = "white",
                fontsize = "medium", fontweight = "bold")
ax.get_xaxis().set_visible(False)
ax.title.set_text("Visualization of decoded AT-rich/CG-rich regions in the␣
 ↪Lambda Phage genome")
```



As before, blue represents an AT-rich region and red represents a CG-rich region. Notice the tiny spec of blue in the left representing the tiny AT-rich section found, it's too small for the text label to fit in it.