School of Engineering and Applied Science (SEAS), Ahmedabad University

Probability and Stochastic Processes (MAT 277)

Special Assignment Report

Group Name: hn_bio _08
Team Members:

- Milee Bajpai (AU2140158)
- Krishna Patel (AU2140170)
- Harsh Pandya (AU2140171)
- Yax Prajapati (AU2140230)

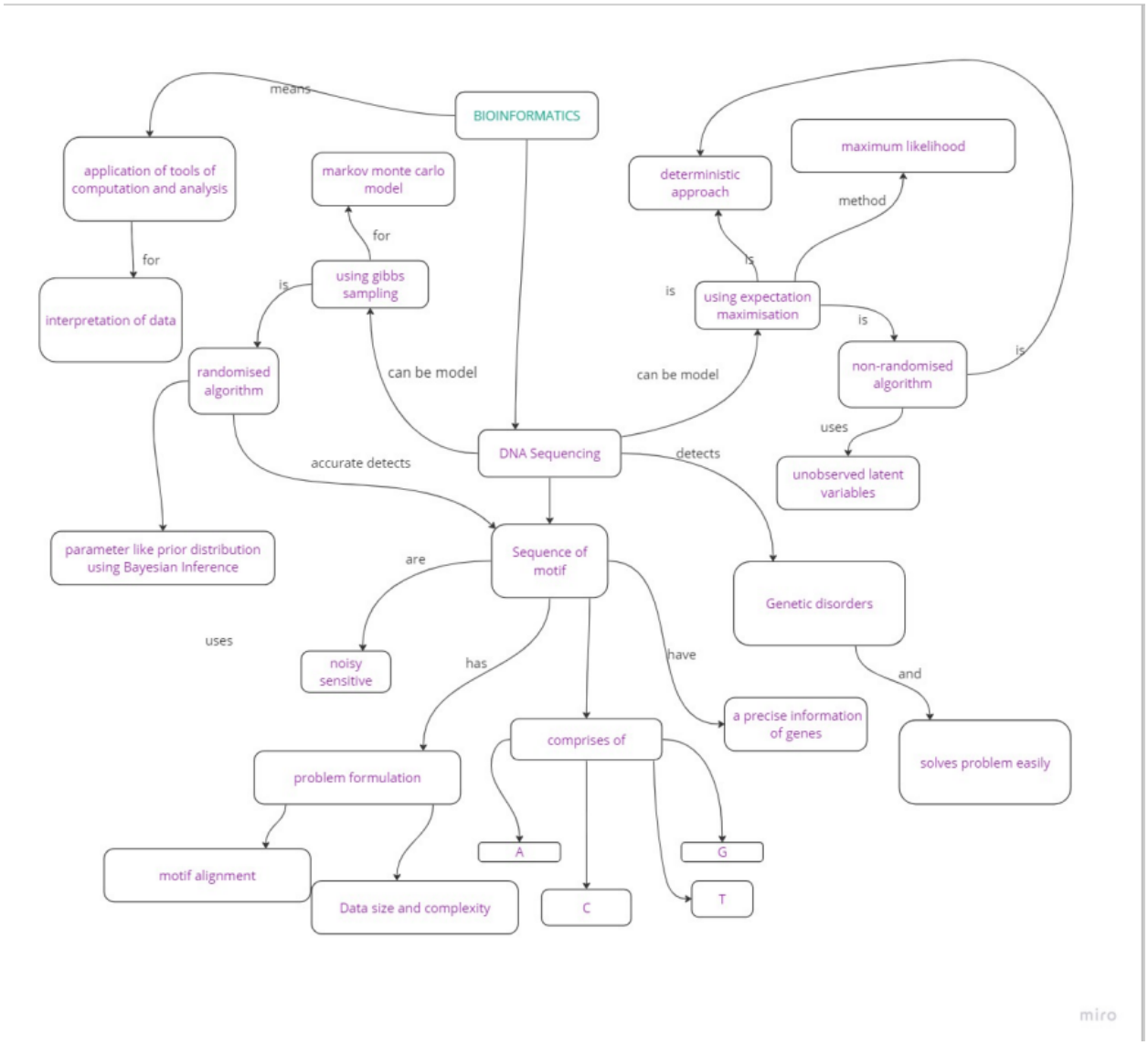
I. Team Activity Learning and Concept Map

The first stage involved choosing a domain. We thus had a team meeting. started a brainstorming session to identify the issues that a probabilistic approach might be able to resolve. Choosing one domain among the three was the aim of the first session.

We had decided on the bio informatics domain and had begun looking for research papers in this area. And discovered a study by the Oxford Press (metrics score: 142) about locating motif sequences. The paper was also reviewed by TAs. The objective at this point was to understand the paper, create a mathematical model, and learn how a probabilistic technique could be used. To understand the topics, we sought assistance from the TAs and conducted some online study. We divided the task since we needed to prepare for two different methods. A deterministic technique will be worked on by two members, and a randomised approach by the remaining members. We collectively learned the deterministic and randomised techniques in this manner. similar to how we handled the coding aspect. In the final section of the special assignment, we revised every single section and strengthened each member's areas of weakness.

Meeting group members and TAs helps in the completion of special assignment because we have encountered failures throughout the process. We had tried connecting the anecdotes numerous times before, but we finally succeeded after giving it our all and focusing on one issue at a time. Our team had a difficult time finding the randomised algorithm and debugging such code, but we were able to overcome some of these problems by watching a few films on gibbs sampling that were presented at MIT lectures.

All of the team members lacked domain expertise, but they made every effort to teach and coach one another.


means
BIOINFORMATICS
application of tools of computation and analysis
for
interpretation of data
markov monte carlo model
for
using gibbs sampling
is
randomised algorithm
can be model
DNA Sequencing
accurate detects
parameter like prior distribution using Bayesian Inference
are
Sequence of motif
uses
noisy sensitive
has
problem formulation
motif alignment
Data size and complexity
comprises of
A
C
G
T
have
a precise information of genes
detects
Genetic disorders
and
solves problem easily
deterministic approach
is
maximum likelihood
method
is
using expectation maximisation
is
non-randomised algorithm
is
can be model
uses
unobserved latent variables
miro

II. Background and Motivation

Background

Sequence motifs are short patterns that occur in DNA with certain frequency and that often have some sort of biological distinct function. For discovering motifs, probabilistic approaches are often used since they offer a mechanism to describe the uncertainty present in biological data. One approach to motif finding involves using a probabilistic model of the motif, such as a position weight matrix (PWM).The likelihood of a sequence given the motif is then determined as the sum of the probabilities of the various places. A PWM assigns a probability distribution across each possible nucleotide or amino acid at each position in the motif. Overall, because they provide the integration of many information sources and determine the level of uncertainty in the results, probabilistic approaches are an effective tool for motif discovery.

Motivation

Discovery of brief and recurrent patterns in DNA, RNA, or protein sequences is a challenge in computational biology known as motif discovery. The goal of motif discovery in the field of probability is to determine the likelihood of spotting a certain arrangement of nucleotide or amino acids in a biological sequence. We can learn more about the evolutionary and functional characteristics of biological sequences by finding motifs. An important functional component, such as a binding site for a transcription factor, may be indicated by a motif that is highly conserved throughout many species. Similar to this, motifs unique to certain species or groups of species may be helpful in determining evolutionary links or spotting horizontal gene transfer activities. Additionally, motif discovery is useful in industries like medication development and genetic engineering. Researchers can create novel medicines that target certain proteins or RNA molecules, as well as construct biological systems with desired traits, by locating and characterizing motifs.

III. Algorithm Description

Input

- dna: a list of DNA reads of equal length
- numSeeds: an integer indicating how many times to seed the genetic algorithm
- k: an integer indicating the motif length being searched for
- N: an integer indicating the number of iterations before returning the best motif

Algorithm description

- bestMotifs: a list of the closest matching motifs found in each string in dna
- Define a function, singleReplacementMotif(motifs, dna_i), that selects a replacement motif for the "motifs" in gibbsSampler, then returns the string "replacementKmer".
- Define a function, BuildProfile(motifs), that returns a dictionary with keys A, C, G, and T and their corresponding probability arrays based on the input "motifs".
- Define a function, profileProb(profile, dna_i), that takes in a dictionary "profile" and a DNA read "dna_i" and returns the probability of the most likely k-mer in dna_i based on profile.
- Define a function, selectRandomMotif(strand, k), that selects a random k-mer from "strand".
- Define a function, score(motifs), that takes in a list of "motifs" and returns the sum of the Hamming distances between each k-mer in the list and the consensus motif.
- Define a function, gibbsSampler(dna, k, N), that takes in a list of "dna" reads, an integer "k" indicating the motif length, and an integer "N" indicating the number of iterations before returning the best motif. The function should return a list "bestMotifs" that contains the closest motif match from each string in "dna".
- In the gibbsSampler function, Initialize "motifs" to a list containing a randomly chosen k-mer from each "dna" read.
- Initialize "bestMotifs" to a list containing the same k-mers as "motifs".
- Loop "N" times:
- Choose a random integer "i" from 0 to t-1, where "t" is the number of "dna" reads.
- Create a list "subsetMotifs" that contains all the k-mers from "motifs" except for the i-th k-mer.
- Create a dictionary "profile" based on "subsetMotifs".
- Calculate the probability density function "kmerDensities" for all possible k-mers in the i-th "dna" read using the "profileProb" function.
- Choose a new k-mer "replacement Motif" from the i-th "dna" read using the probability density function "kmerDensities".
- Replace the i-th k-mer in "motifs" with "replacementMotif".
- If the new "motifs" list has a lower "score" than the previous "bestMotifs" list, replace "bestMotifs" with "motifs".
- returns best motif.
- Define a function, multipleSeedsGibbsSampling(dna, numSeeds, k, N), that takes in "dna", "numSeeds", "k", and "N". The function should run "gibbsSampler" "numSeeds" times and return the list "bestMotifs" that contains the closest motif match from each string in "dna".

IV. Application

DNA sequencing has numerous applications across various fields, including:

- Gene regulation: Motifs are often found in promoter regions of genes and can be used to identify potential transcription factor binding sites. Understanding gene regulation is important for understanding how cells and organisms respond to different stimuli and for developing new treatments for diseases.
- Protein structure prediction: Motifs can be used to predict the structure and function of proteins. Protein structure prediction is important for drug discovery, as it allows researchers to design drugs that specifically target certain proteins.
- RNA splicing: Motifs are important for RNA splicing, the process by which pre-mRNA is edited to remove introns and splice together exons. Abnormal splicing can lead to disease, so understanding the motifs involved in splicing can help researchers develop treatments for these diseases.
- Protein-protein interactions: Motifs can also be used to predict protein-protein interactions. Understanding protein-protein interactions is important for understanding how cells and organisms function and for developing new treatments for diseases.
- Meta genomics: Motif detection can be used to identify conserved regions in metagenomic data, which is DNA isolated directly from environmental samples. Identifying conserved regions can help researchers understand the diversity of microorganisms in different environments and their potential roles in various ecological processes.
- Overall, motif detection is a powerful tool for understanding biological systems and has many potential applications in a wide range of fields, from medicine to environmental science.

Overall, DNA sequencing has transformed the study of biology and medicine and helped scientists comprehend the genetic foundations of a wide range of illnesses and features.

V. Mathematical Analysis

Mathematical model

hn_bio_08.

Date
Page

→ Each seed represents one monte carlo run.
numseeds = 20

→ "k" is a motif length.

k = 11

→ "N" no. of iteration before returning the best motif.

We firstly create an Empty array of DNA

∴ Dna = [ ]

This array has all the DNA Sequences

After that we have to go to gibbs.sampler which can extract the motif of desired length (substr)

Then, we have to calculate the score of a motif and in that we required a ('pattern') common motif from which we can compare one by one substr

Eg

Drop[list:26] = [GTCACTGTGTACTCTACTTGAA,
CCCGGGGTACGAA,
AAA.....,

.
.
.
.
.

CCCACGTGACTTAGGAATTA
CGACGTTTGACGGATTTG
A].

(K=11), i=0.

Substr[list:26] [GTCACTGTGTA, ...
....., CCCACGTGACT]

then for each position we have to calculate the number of each character in that position for all 26 substr sequence.

for Eg.

Position at i=0, no. of A = 10 T = 6
C = 9
G = 1

As 'A' is occuring more than other protein

Hence we can assume that in position 'i=0' of common motif

we have patten = ['A']

→ position at i=1

no. of A = 9
C = 9
G = 11
T = 12

As 'T' is occuring more than other protein

Hence we can assume that in position 'i=1' of common motif

we have patten = ['AT']

→ Similarly,

position at i=10

no of A = 9
C = 14
G = 8
T = 7

As C is occuring more then other protein
Hence we can assume that in position '1' of common motif
we have Pattern = ['AT ... C']

∴ common motif is
Pattern = ['ATTCCAGCCCC']

for score of motif.
we have to calculate hamming Distance

In hamming Distance individual substr & Pattern is compared

(score += hammingDistance(substr, pattern))

Eg Substr : CCTCACTCTCTA (zip type
= pattern : ATTCCACCCC comparison)

score : 9 (and it goes on in accumulate for all 26 DNA seq)

Higher = Higher
Score - Mismatches

It will compared for N iterations score of a motif will be compared, Minimum score will be the best motif.

if the previous motif score is not good then we have to replace the motif by the function of single replacement motif and for that 'build profile'

In that build profile, we have to create a Probability matrix of 4 x k
(4 ↓ rows, k → columns)

rows are the substitute, while the columns are the list

	0	1	2	3	4	5	6	7	8	9	10
A											
C											
G											
T											

4×K

firstly, we have to initialize the matrix with zero entry.

and we have to again find the common motif & divide that with its score.

	0	1	2	3	4	5	6	7	8	9	10
A	0.13	0.10	0.51	0.36	0.167	0.377	0.149	0.46	0.14	0.88	0.446
C	0.27	0.27	0.01	0.26	0.44	0.271	0.32	0.36	0.48	0.56	0.983
G	0.12	0.13	0.127	0.11	0.63	0.826	0.29	0.14	0.28	0.91	0.981
T	0.1	0.12	0.07	0.19	0.23	0.226	0.146	0.65	0.23	0.28	0.71

4×K

Now, Normalize this probability matrix

In Normalization,

Normalization total = (Sum(Kmer densities)) ∴ we have to add all the Probability of proteins a particular position.

∴ Kmer densities [i] = Kmer densities [i] / normalization total

∴ for 'A' at position 0

∴ Probability of A occuring at 0 will be $\frac{0.13}{0.13 + 0.27 + 0.12 + 0.10}$

$= \frac{0.13}{0.62} = 0.209$

At the output,

we will get the Algorithm best motif & it will match exactly with the Substr.

VI. Code(with description of each line)

DemmoGibbsSampler.py

Read: This script demos the Gibbs Sampler Motif finding algorithm. The test data is a collection of mouse DNA reads. Each read contains a binding site for the Zinc fingered GATA4 promoter. The algorithm locates and prints each binding site. You can compare the algorithm's answer to the real answer, by opening the solution file. In that file the real motif binding site appears in capital letters.

This algorithm is an improvement over the brute force algorithm reducing the brute force exponential runtime to a probablistic genetic algorithm polynomial run time. !!!!

```
import sys
import os
import inspect
import re

# determine path to script, then
# import GibbsSampler module
filename = inspect.getframeinfo(inspect.currentframe()).filename
path = os.path.dirname(os.path.abspath(filename))
sys.path.insert(0, path)
import GibbsSampler


def DemoMotifFinder(numSeeds, k, N):
        """ numSeeds" -- Is an integer indicating how many times to seed the genetic algorithm.
            Each seed represents one monte carlo run. (no default) """
        """ "k" -- An integer indicating the motif length being searched for. (no default) """
        """ "N" -- The number of iterations before returning the best motif. (no default) """
        """ This demos the gibbsSampler algorithm, but finding the common motif for the Zinc
            Fingered GATA4 promoter."""
        filename = inspect.getframeinfo(inspect.currentframe()).filename
        path = os.path.dirname(os.path.abspath(filename))
        # Load mm9 DNA from file.
        # Each read contains a motif, specifically a Zinc Fingered GATA4 promoter site.
        # This software will find that promoter site.
        mm9Loc = path + "/mm9Gata4MotifCollection.txt" #location of dataset.

        #mm9File =
            open("/Users/scottczopek/Documents/bioPython/pythonMotifFindingDemo/mm9Gata4MotifCollection.txt",'r')
        mm9File = open(mm9Loc, 'r')

        Dna = []
        for line in mm9File:
                if line[0:3] != '>mm':
                        line = line.strip()
                        Dna.append(line)

        BestMotif = GibbsSampler.multipleSeedsGibbsSampling(Dna, numSeeds, k, N)
        bestScore = GibbsSampler.score(BestMotif)

        mm9File.close()
        return BestMotif


numSeeds = 20
k = 11
N = 1000
BestMotif = DemoMotifFinder(numSeeds, k, N)
```

```
bestScore = GibbsSampler.score(BestMotif)
print("Gibbs Sampler Motifs")
print("BestScore: ", bestScore)
#print(BestMotif)

mm9SolLoc = path + "/mm9Gata4Solutions.txt"
mm9Solutions = open(mm9SolLoc, 'r')
SolutionsMotif = []
for line in mm9Solutions:
	if line[0:3] != '>mm':
		line = re.sub('[^A-Z]', '', line)
		SolutionsMotif.append(line)
print("Real Motifs")
print("Real Score: ", GibbsSampler.score(SolutionsMotif))
print()
print("Algorithms Best Pick", "  Match/Wrong     ", "Real Motif")
for i in range(len(SolutionsMotif)):
	if SolutionsMotif[i] == BestMotif[i]:
		print(BestMotif[i], "   Match     ", SolutionsMotif[i])
	else:
		print(BestMotif[i], "   Wrong     ", SolutionsMotif[i])
```

GibbsSampler.py

```
    # Need some random number functionality and the operator.add ability.
import random
import operator


def multipleSeedsGibbsSampling(dna, numSeeds, k, N):
    """A motif finding algorithm that finds one common motif and returns a list bestMotifs
        containing the closest motif match from each string in "dna". The library was written to
        support this function.

        Keyword arguments:
        "dna" -- A list of DNA reads that are the same length. All letters need to be upper case.
            (no default)
        "numSeeds" -- Is an integer indicating how many times to seed the genetic algorithm. Each
            seed represents one monte carlo run. (no default)
        "k" -- An integer indicating the motif length being searched for. (no default)
        "N" -- The number of iterations before returning the best motif. (no default)

        Return value:
        "bestMotif" -- For each "Dna" reads return the best length "k" motif match in a list
            BestMotifs.
                                    These list entries are the closest scoring matches after
                                        running the gibbsSampler
                                    algorithm "numSeeds" times. Each run gets its own unique
                                        starting seed, and that seed
                                    goes through N cycles before returning the local best match.
                                        The final result BestMotifs
                                    represent variants of a single DNA motif. (This assumes a
                                        common
                                    DNA motif be present in each DNA read.)
        """
    results = gibbsSampler(dna, k, N)
    bestScore = score(results)
    bestMotifs = list(results)
```

```
    for i in range(1, numSeeds):
        results = gibbsSampler(dna, k, N)
        if score(results) < bestScore:
            bestScore = score(results)
            bestMotifs = list(results)
    return bestMotifs


def gibbsSampler(dna, k, N):
    """A motif finding algorithm that finds one common motif in a list of "dna" reads, returns a
        list bestMotifs that contains the closes motif match from each string in "dna".
        Keyword arguments:
        "dna" -- A list of DNA reads that are the same length.
                                All letters need to be upper case. (no default)
        "k" -- An integer indicating the motif length being searched for. (no default)
        "N" -- The number of iterations before returning the best motif. (no default)

        Return Value:
        "bestMotifs" -- For each "Dna" reads return the best length "k" motif match in a list
            BestMotifs.
                                    These list entries are the closest scoring matches after
                                        running the gibbsSampler
                                    algorithm "N" times. They represent variants of a single DNA
                                        motif. (This assumes a common
                                    DNA motif be present in each DNA read.)
        """
    t = len(dna) # total number of dna seq. in file
    # randomly select k-mers Motifs = (Motif_1, , Motif_t) in each string from Dna
    motifs = []
    for strand in dna:
        i = random.randrange(len(strand) - k + 1)
        substr = strand[i:i + k]
        motifs.append(substr)

    bestMotifs = list(motifs)
    bestMotifsScore = score(bestMotifs)
    for j in range(1, N):
        i = random.randrange(t)
        subsetMotifs = motifs[0:i] + motifs[i + 1:t]
        replacementMotif = singleReplacementMotif(subsetMotifs, dna[i])
        motifs[i] = replacementMotif

        if score(motifs) < bestMotifsScore:
            bestMotifs = list(motifs)
            bestMotifsScore = score(bestMotifs)
    return bestMotifs


#
# helper functions
# functions that come after this line are helper functions
#


def singleReplacementMotif(motifs, dna\textunderscore i):
    """Select a replacement motif for the "motifs" variable in gibbsSampler, then return the
        string "replacementKmer".


        Detailed function summary:
```

```
        Select a replacement motif for the "motifs" in gibbsSampler, then return the string
            "replacementKmer". This replacement is a substring
        from the i_th DNA read ("dna\textunderscore i"). It will later replace the i_th "motifs"'s
        entry. Once this motif is replaced, the motifs' list converges a little closer
        to the best match. This best match is the common motif shared between the
        DNA reads.

        Keyword arguments:
        "motifs" -- Is the variable from the motifs' function.
                                    !!! Except that the i_th row has been removed. !!!
                                    "dna\textunderscore i" is the i_th DNA read. (no default)
        Return value: Return a replacement k-mer "replacementKmer" to replace the i_th "motif"
            entry.
        """

    k = len(motifs[0])
    profile = BuildProfile(motifs)

    # Calculate probilities for each k-mer in dna\textunderscore i
    kmerDensities = [0 for x in range(len(dna\textunderscore i) - k + 1)]
    for i in range(len(dna\textunderscore i) - k + 1):
        prob = 1
        for j in range(k):
            if dna\textunderscore i[i + j] == 'A':
                prob *= profile[j][0]
            elif dna\textunderscore i[i + j] == 'C':
                prob *= profile[j][1]
            elif dna\textunderscore i[i + j] == 'G':
                prob *= profile[j][2]
            elif dna\textunderscore i[i + j] == 'T':
                prob *= profile[j][3]
        kmerDensities[i] = prob

    # normalize probabilities
    normalizationTot = sum(kmerDensities)
    for i in range(len(dna\textunderscore i) - k + 1):
        kmerDensities[i] = kmerDensities[i] / normalizationTot

    # construct prefix sum for lookup
    kmerDensities = list(accumulate(kmerDensities))

    # randomly select a k-mer
    randVal = random.random()
    for i in range(len(dna\textunderscore i) - k + 1):
        if randVal < kmerDensities[i]:
            break
    replacementKmer = dna\textunderscore i[i:i + k]

    return replacementKmer


def BuildProfile(motif):
    """ Build the "profile variable for "gibbsSampler", return "profile". Please see special note.

        Special note:
        profile is a nested list
        profile[k][A,C,G,C]

        Keyword arguments:
        "motif" -- A list of strings. "motif", each entry on this list corresponds to a
```

```
                                        DNA read. It is a substring of that read, k letters
                                            long, which represents a
                                        common motif between different DNA sequences. (no
                                            default)
        Return value:
        "profile" -- A 4xk probability matrix. Each of the four rows corresponds to
                                    a probability that the k-mer is 'A','C','G', or 'T' at the nth
                                        position. The product
                                    of one entry from each column is the probability that a k-mer
                                        is a given sequence.
                                    This matrix is a nested list. The columns are the list, and
                                        the four rows are sub-list.
        """
    k = len(motif[0])
    profile = [[0 for y in range(4)] for x in range(k)]
    for count in range(k):
        A = 0
        C = 0
        G = 0
        T = 0

        # Add in Laplace counts to avoid
        # prob densities that are zero or one
        # accelerates alg runtime
        A += 1
        C += 1
        G += 1
        T += 1
        for string in motif:
            if string[count] == 'A':
                A += 1
            elif string[count] == 'C':
                C += 1
            elif string[count] == 'G':
                G += 1
            elif string[count] == 'T':
                T += 1
        # Insert frequencies if base A
        profile[count][0] = float(A) / (A + C + G + T)
        # Insert frequencies if base C
        profile[count][1] = float(C) / (A + C + G + T)
        # Insert frequencies if base G
        profile[count][2] = float(G) / (A + C + G + T)
        # Insert frequencies if base T
        profile[count][3] = float(T) / (A + C + G + T)
    return profile


def BuildMotifs(profile, dna, k):
    """ Build the "motifs" variable for the function gibbsSampler, return "motifs".
        Keyword arguments:
        "profile" -- A 4xk matrix containing the probabilities that
                                    the nth base in length k motif will be 'A','C','G', or 'T'.
                                    The prob that the motif is a particular seq is the product of
                                    one entry from each of the k rows. (no default)
        "dna" -- Is a list, length t, of Dna reads being searched for a
                                    commone motif. (no default)
        "k" -- An integer indicating the length of the motif being searched for. (no default)
        Return value:
        "motif" -- A list of length k Dna sub-strings. Each list entry
```

```
                                    corresponds to a substring in Dna. The first "motif" entry
                                        corresponds
                                    to the first "dna" entry, the second to the second, and so on.
                                        Each "motif"
                                    entry is chosen to be the most probable substring based on the
                                    probabilities given in "profile", representing the most
                                        probable motif
                                    shared between all the "dna" reads.
        """
    motif = []
    for string in dna:
        bestSubStr = ''
        for i in range(len(string) + 1 - k):
            substr = string[i:i + k]
            prob = 1
            bestProb = -1
            for j in range(k):
                if substr[j] == 'A':
                    prob *= profile[j][0]
                elif substr[j] == 'C':
                    prob *= profile[j][1]
                elif substr[j] == 'G':
                    prob *= profile[j][2]
                elif substr[j] == 'T':
                    prob *= profile[j][3]
            if prob > bestProb:
                bestProb = prob
                bestSubStr = substr
        motif.append(bestSubStr)
    return motif


def score(motifs):
    """ Counts the number of mismatches between strings in the list "motifs", returns the mismatch
         count as the "score".
        Keyword arguments:
        "motifs" -- A variable representing a collection of DNA sub-reads, length k. (no defaults)
        Return Value:
        "score" -- Returns the number of single base mismatches in "motifs". The higher the value
            the worse the score.
        """
    k = len(motifs[0])
    pattern = []
    for i in range(k):
        A = 0
        C = 0
        G = 0
        T = 0
        for string in motifs:
            if string[i] == 'A':
                A += 1
            elif string[i] == 'C':
                C += 1
            elif string[i] == 'G':
                G += 1
            elif string[i] == 'T':
                T += 1

        # """
        # For tie counts this chooses A over C
```

```
        # C over G, and G over T.
        # However, this does not introduce bias.
        # It has the same results as randomly breaking a tie,
        # but is much simpler to implement.
        if A >= C and A >= G and A >= T:
            pattern.append('A')
        elif C >= G and C >= T:
            pattern.append('C')
        elif G >= T:
            pattern.append('G')
        else:
            pattern.append('T')
    # """
    pattern = "".join(pattern)

    score = 0
    for string in motifs:
        score += hammingDistance(string, pattern)
    return score


def d(kmer, dna):
    """ The functiond d calculates the number of mismatches between the string "kmer" and the
        string list "dna", this number is returned as "totDist".
        Keyword arguments:
        "dna" -- A list of strings. Each entry in "dna" must be longer than "kmer". (no default)
        "kmer" -- A string representing a DNA kmer.

        Return value:
        "totDist" -- An integer which totals number of mismatches between "kmer" and each list
            entry in "dna".
        """
    k = len(kmer)
    motif = []
    totDist = 0
    for phrase in dna:
        localDist = len(phrase) + len(kmer)
        word = ""
        for i in range(len(phrase) - k + 1):
            subPattern = phrase[i:i + k]
            if localDist > hammingDistance(kmer, subPattern):
                localDist = hammingDistance(kmer, subPattern)
                word = subPattern
        motif.append(word)
        totDist += localDist
    return totDist


def hammingDistance(str1, str2):
    """
        Keyword arguments:
        "str1" -- A string. (no default)
        "str2" -- A string. (no default)

        Return value:
        "diffs" -- The number of mismatches between "str1" and "str2".
                                        Strings can be different lengths, but the mismatch
                                            count is
                                        only includes counts on the shortest length.
        """
```

```
    diffs = 0
    for ch1, ch2 in zip(str1, str2):
        if ch1 != ch2:
            diffs += 1
    return diffs


def accumulate(iterable, func=operator.add):
    """Count the # of differences between equal length strings str1 and str2.

        Keyword arguments:
        "iterable" -- An iterable "iterable" that can be summed through the
        function operator.add . (no default)

        Return value:
        yield the total "total".
        """
    'Return running totals'
    # accumulate([1,2,3,4,5]) --> 1 3 6 10 15
    # accumulate([1,2,3,4,5], operator.mul) --> 1 2 6 24 120
    it = iter(iterable)
    try:
        total = next(it)
    except StopIteration:
        return
    yield total
    for element in it:
        total = func(total, element)
        yield total
```

Diterministic Approach (Expectation maximization)

```
    import random
import time


def deterministic_em_motif(fasta_seqs, k, max_iterations=1000):
    # Initialize motifs with random k-mers from the input sequences
    motifs = [random.choice([seq[i:i + k] for i in range(len(seq) - k + 1)]) for seq in fasta_seqs]

    for iteration in range(max_iterations):
        # Compute the profile matrix from the current motifs
        profile = compute_profile(motifs)

        # Update the motifs by selecting the k-mer with the highest probability from the profile
            matrix
        motifs = [most_probable_kmer(seq, profile, k) for seq in fasta_seqs]

        # Check for convergence
        if iteration > 0 and motifs == old_motifs:
            break

        old_motifs = motifs

    consensus = compute_consensus(profile)

    # Calculate probability score for each motif based on the consensus motif
    scores = []
```

```
    for motif in motifs:
        score = 1
        for i, base in enumerate(motif):
            score *= profile[i][base]
        # Normalize the score by dividing it by the probability of generating the motif by a random
            sequence
        score /= (0.25 ** k * len(fasta_seqs))
        scores.append(score)

    # Normalize the scores by dividing each score by the sum of all scores
    sum_scores = sum(scores)
    normalized_scores = [score / sum_scores for score in scores]

    return motifs, normalized_scores


def compute_profile(motifs):
    counts = [{base: 1 for base in 'ACGT'} for i in range(len(motifs[0]))]

    for motif in motifs:
        for i, base in enumerate(motif):
            counts[i][base] += 1

    profile = [{base: count / len(motifs) for base, count in count_dict.items()} for count_dict in
        counts]

    return profile


def compute_consensus(profile):
    consensus = ''
    for i in range(len(profile)):
        base = max(profile[i], key=profile[i].get)
        consensus += base
    return consensus


def most_probable_kmer(seq, profile, k):
    max_prob = -1
    most_probable = None

    for i in range(len(seq) - k + 1):
        kmer = seq[i:i + k]
        prob = 1

        for j, base in enumerate(kmer):
            prob *= profile[j][base]

        if prob > max_prob:
            max_prob = prob
            most_probable = kmer

    return most_probable


# Example usage
fasta_seqs = [
    'GTCACTGTGTACTCTAGGCTCGTTGGTCCCCAAGCTTCTGGGTGGCTCTTTCTTATCTCCCGTCTTACTGTAAGAACAGATGGAGTGCTAGAACAAGTAGGATTGTGTCT
    'CCTCCTAGAGGCCTTTCACTCAAGTCCCTGCAGGATGAGGTACAAGCCTCTCTTATCTCCCCGCATCTATCCCTACAATCCTCCAGCCAGAGCAACCACTCACTGACTCA
    'ACCCTTCAGTTTTATCCGTTACTAACCAAGATAGAGTAAATTTGCTAAAATCTTATCTGTGAGAACTACAATGCAAACAAACCAAAGATTAAAAAGCCACACTGGACAGC
```

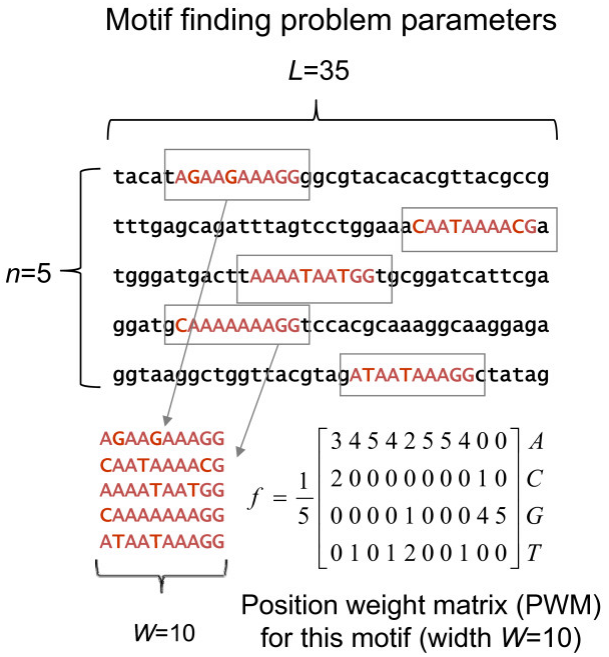
```
    'TTAGGAAGCTTAGCCCTGCTCTCTCCTAGGTTCATGCTGCCCACTTGCCCTCTTATCTGTTTAAAAGGATTTCAGCCACAGCTCTCAAGGCCTATTAATGGGTTGTATTC
    'CGTTTTCTAATGCTCATGGTAAGGATTAAATGTCACAATTCACGTGTCTCTCTTATCTGTTCTGTGCATGCGGTACAGCGGATCATCTGTAGCATACAGCTAGATGCTGA
    'CTAGAGATGCTATGGAGAATATCTGAAATTCATAGATAACCTTCCTCTTGCCTTATCTCTTGACATGAATGCAGATGACTAAAAATCTCATGCAGAAAATGCACCCCCCT
    'CCACCTGGGTTAATGGAGGTCATCACCAGCCCAGTATCTCAAACCAAAGTTCTTATCTCCCAGGACTCTGTCCTATCCCTTCCGGCACATTCCATAATCTCCTCGGCAGC
    'TGTCGGGCCAGGAGTATCAGCGCGCACACCAAATCTGCTCTGGTATGTCATCTTATCTCCCTTCCCCGCTGTTGTCCCCAAACGCTGCCTGTCAAGAGAGACGCCACCCC
    'AACCCCCTCCCAAAGAACAGCCTGGAGATAACCACAAAAATGGGGAAATATCTTATCTCAGAATGGGCCATCTGTGCTGGGCTGCCCTATCTCGTCCTAGAGTTAAGCGT
    'TGCTTGTCCCTTCCCACTCCTCGGCCAACCTCAGGCCTCCTCGTTCAGGGCCTTATCTCTTAGCCGCCGCCCCCCCAAAACGGGGCAAACATCTAGAAATGGCCTTCTGC
    'GCTGGCAGCCATGGCTGGCGGGTCCTTCCTAGCATGTGCTGCGGCCCTCACCTTATCTTCCGTCATCAGGGCACAGGCGCCGTGCAGGGCTATTCCGAGTCCAGGTAGAC
    'AGCACTCTGAAAAGGGGAACATCTCAGACTGCTCCCTGGAATGGCTTGGGTCTTATCTGTGTCATGACCTGAACTGGATAATATGTCAGCGTCTTCAGAACCTCCACAAA
    'GTAACTCAGCAAAGCAGGAAACAAAGCCCCATGGTCTCTGTTTCTGACATTCTTATCTGTGTGAGCCCAGTGACAAATGCCAGATTCCCATGTCACCAAAAGCTCTCAAC
    'CCCTTGGGAATCTCCCTGGCACCGCTGGGCACCGAGCCTATCTCTGCACCTCTTATCTCAGCTTCCCTCGGGGGCAAGCTCTGTCTCTACCGAACCCCCTTGAAATGCAC
    'CAGTATTCAGCAAATCCTGGGCTGGTTTCCATCTGGCCAAACCACAGTCCTCTTATCTCAGGAGGCAGGCAGTACTCTCTGGTTCTTGATTAGTAACAATGATTTTAAGC
    'ACACTCTTAAACAATCACCAACAAAAACCCCTGGTGCTCTCTGTGCACACTCTTATCTCAGAGATGACCTTTCCAAGCATAGAGACCCAAGGCTGAAGGACAGCAGGTTC
    'CTCTCACTTTGTACTTTTAAACACATACTTTAAACACACACGCCTTGTACCCTTATCTCTGTAGATAGAGCTCCCCGAGGCTGAGTCATACCAGTGTCTGGATCTGGCAC
    'GATGGGAATCGCGTAGGAGTGGCCGCTCTCTGCCCTCCCCACTCTACAACCCTTATCTTCCTCCTCTAGCCACTTGACTCCTCGCAGATACCTGTGGTGTGGGGGGTGGC
    'GCAGATGGCACCCGGCGACTACAGTTACTTAAGTTCTACCACTCAGTGGTTCTTATCTGTGTATAAACTGTTGATTGGCCGACATTCTGGCAAATCTATGAATGAGTCAC
    'CAGCTAAGACAATAAGGAACTGGGCTCCTCTGCAGGCCTCTCCTGCAGACCCTTATCTCTGGACAGTAACTCCCAAGGCCCCAGGCCTCTTCACACCCACAGGCATCTGA
    'ATTGTCATGCTTTTAAGTGTGTGCTTGCCATTTTGATAATTAGGTTCCATTCTTATCTGTTGCCCTGACCACTGACATAGACCCATCTCTGCCCAGTAGGTTGTGGTGCT
    'TGTTCGCTCTGCCTGTTTCTCTGGCTTGGGATCAACAAGAAAGAGTATTTCCTTATCTCTTTCCCTCAAGAGGATTGCTTTTCAGAGTTTCTCTTATCTTAAAAGGATGA
    'CTCTATCACAGCGTGCTCTTTGGTGAGAGGAGACGGTCAGGCACTGGGCCCCTTATCTCTTGGTCTTGTGCCTGGAGTCTGTCATGCAGTCAAGCGAACACTCCTGTTGC
    'AAACGACCCTGCACGGCCATGGTCTGGTCTTACTCAGACTGGTCTAAGTATCTTATCTGTGTTTCCACATTTGACCCTGTGTAGGTGTTATCCTTGATGTACTACTTGGA
    'AAACTACCCAGACCACTCTGGCAACAGTGTGTCTGCCTTGCTTCATGGTGTCTTATCTCAGTTCTGTCTTAGCTCTCTGTGTAGTGTGACCTCCAGGTCATCTACTCGGC
    'GTATCTTTTTTAATAAGGTCTGGCTGAGTAGCCCAGGCTGGCCCAGGTCATCTTATCTCAGCCTCCCGAGTGCTGGTACTGTGGGTGAGTGCCAGCCACCACACCCAGCT
]
k = 11

start = time.time()
motifs, scores = deterministic_em_motif(fasta_seqs, k)
end = time.time()
print('Identified motifs:', motifs)

print('Match probability scores:', scores)
print('Execution time is: ', (end - start) * 10 ** 3, 'ms')
```

VII. Results and Inferences

- we have derived that for detecting motifs in a sequence of DNA, the applied randomized approach is more effective and efficient than the traditional deterministic approach. Overall, this code provides a way to find common motifs in DNA sequences, which can be useful in many biological applications such as identifying transcription factor binding sites or regulatory regions.



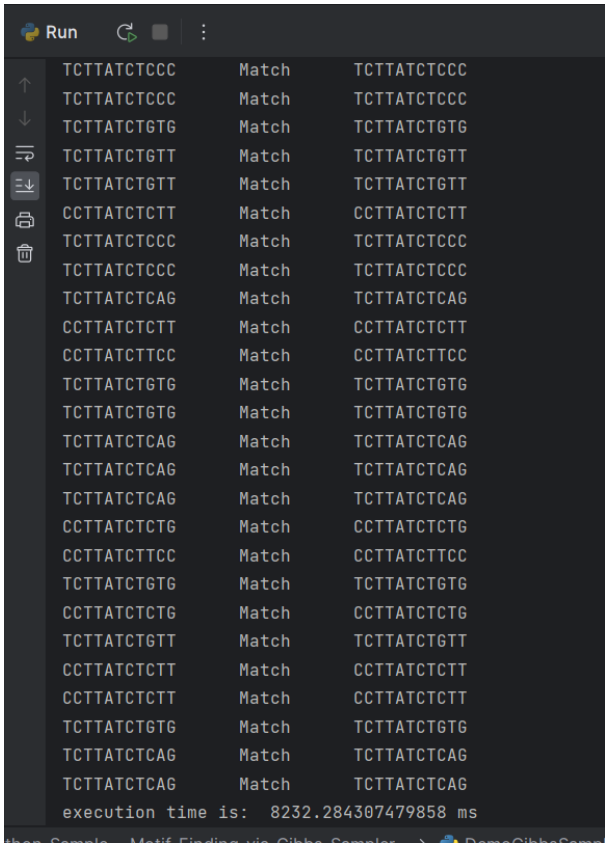
(a) Probability Weighted Matrix

```
G:\Python\temp\venv\Scripts\python.exe G:\Python\
Identified motifs: ['CTCTTTCTTAT', 'GCCTCTCTTAT',
Match probability scores: [0.3219421131412239, 0.
Execution time is:  15.62643051147461 ms
```

(B) Execution time and output with deterministic approach for the 1 iteration only

No	Motif	Pattern	Score
00	'CGTCTTACTGT'	'TCAGTTGTTAC'	8
01	'TCTTATCTCCC'		14
02	'AACCAAGATAG'		22
03	'TGGGTTGTATT'		27
04	'TAAATGTCACA'		35
05	'AGAAAATGCAC'		43
06	'ATGGAGGTCAT'		50
07	'CCCGCTGTTGT'		55
08	'AAAATGGGGAA'		62
09	'TCTAGAAATGG'		70
10	'ACAGGCGCCGT'		77
11	'TATCTGTGTCA'		85
12	'CTGTGTGAGCC'		93
13	'CTCTGTCTCTA'		102
14	'TCTTGATTAGT'		110
15	'GCTGAAGGACA'		118
16	'GTGTCTGGATC'		126
17	'CCACTTGACTC'		131
18	'TATAAACTGTT'		140
19	'GCAGACCCTTA'		147
20	'TTGATAATTAG'		153
21	'AGAGTATTTCC'		158
22	'TGAGAGGAGAC'		163
23	'CGACCCTGCAC'		171
24	'GTGTGTCTGCC'		179
25	'AGCCTCCCGAG'		188

(C) Score Calculation Using hamming distance



Output using randomised approach for 1000 iteration.

VIII. References

https://academic.oup.com/bioinformatics/article/21/10/2240/206744
https://youtu.be/1EMonM7qAU8
https://www.youtube.com/watch?v=d5NMrA2HkG4
https://youtu.be/d5NMrA2HkG4
https://www.youtube.com/watch?v=vupAgqunSGM
https://github.com/sczopek/Python-Sample$_{-Motif-Finding-via-Gibbs-Sampler}$