* Para encontrar cuantas veces se repite un patrón dentro de una secuencia

def PatternCount(Text, Pattern):

count = 0

for i in range(len(Text)-len(Pattern)+1):

if Text[i:i+len(Pattern)] == Pattern:

count = count+1

return count

print (PatternCount(Text, Pattern))

* Para buscar que patron se repite mas especificando el largo

n = len(Text) for i in range(n-k+1): Pattern = Text[i:i+k] freq[Pattern] = 0

* Este Código lo que hace primero es estalecer el rango de datos que va a tomar de acuerdo con la entrada y va tomando valores en la secuencia de acuerdo a esto, le va sumando 1 a la matriz cuando encuentra valores nuevos o iguales.

def FrequencyMap(Text, k):

freq = {}

n = len(Text)

for i in range(n-k+1):

Pattern = Text[i:i+k]

freq.setdefault(Pattern, 0)

if Text[i:i+k] == Pattern:

freq[Pattern] += 1

return freq

Text = 'CGATATATCCATAG'

k = 3

print (FrequencyMap(Text, k))

* Para traer de Vuelta solo los valores mas grandes que se obtuvieron se usa el siguiente código
* En este se sacan los valores numéricos máximos del arreglo que se tuvo antes, después de esto se evalua que el valor máximo obtenido sea igual a los del arreglo, si cumple con esto se guarda en un nuevo arreglo donde solo van a estar las palabras que mas se repiten.

def FrequentWords(Text, k):

words = []

freq = FrequencyMap(Text, k)

m = max(freq.values())

for key in freq:

if freq[key] == m:

pattern = key

words.append(pattern)

return words

def FrequencyMap(Text, k):

freq = {}

n = len(Text)

for i in range(n-k+1):

Pattern = Text[i:i+k]

freq.setdefault(Pattern, 0)

if Text[i:i+k] == Pattern:

freq[Pattern] += 1

return freq

Text = 'ACGTTGCATGTCGCATGATGCATGAGAGCT'

k = 4

print (FrequentWords(Text, k))

* Pattern to number: Introduce una secuencia y busca encontrar
* Para introducer una secuencia y que regrese la reversa se toma un valor, se agrega a la derecha y los siguientes se van sumando a la izquierda para que se vuelva el reverso

def Reverse\_String(Text):

reverse = ""

for char in Text:

reverse = char + reverse

return reverse

Text = 'AAAACCCGGT'

print (Reverse\_String(Text))

* Para obtener la secuencia complementaria se cambian las A por T y las C por G y al reves.

def Complement(Text):

comp = ''

for char in Text:

if char== 'A':

comp += 'T'

elif char == 'T':

comp += 'A'

elif char == 'G':

comp += 'C'

elif char == 'C':

comp += 'G'

else:

comp+= 'N'

return comp

Text = 'AAAACCCGGTO'

print (Complement(Text))

\*Al combiner las dos funciones anteriores

def Reverse\_Complement(Text):

RevComp = ''

Rev = Reverse(Text)

RevComp = Complement(Rev)

return RevComp

def Reverse(Text):

reverse = ""

for char in Text:

reverse = char + reverse

return reverse

def Complement(Text):

comp = ''

for char in Text:

if char== 'A':

comp += 'T'

elif char == 'T':

comp += 'A'

elif char == 'G':

comp += 'C'

elif char == 'C':

comp += 'G'

else:

comp+= 'N'

return comp

* Para encontrar la posición en la que esta un patron dentro de la secuencia se utiliza la siguiente función

def PatternMatching(Pattern,Text):

positions = []

for i in range(len(Text)-len(Pattern)+1):

if Text[i:i+(len(Pattern))] == Pattern:

positions.append(i)

return positions

* Este código va a identificar patrones dentro del genoma al contar la cantidad de nucleótidos (A T C G) que hay dentro, para esto toma una ventana que tiene de tamaño la mitad del genoma completo y que va avanzado cada posición para contar los nucleótidos dentro de ese rango. El extended genoma es el genoma completo mas la mitad para que se pueda completar la lectura de la ultima ventana.

def SymbolArray(Genome, symbol):

array = {}

n = len(Genome)

ExtendedGenome = Genome + Genome[0:n//2]

for i in range(n):

array[i] = PatternCount(symbol, ExtendedGenome[i:i+(n//2)])

return array

# Reproduce the PatternCount function here.

def PatternCount(Pattern, Text):

count = 0

for i in range(len(Text)-len(Pattern)+1):

if Text[i:i+len(Pattern)] == Pattern:

count = count+1

return count

* Skew ddiagram: Para este Código se crea un diccionario de un genoma en el que se asignan valores de acuerdo con la letra que tiene, si tiene ‘G’ se asigna el valor del puesto anterior + 1, si tiene ‘C’ se asigna el valor anterior – 1, para ‘A’ y ‘T’ el valor se vuelve igual al anterior. Al asignar el primer valor como 0 se logra que se coloque el nuevo valor sobre la siguiente posición.

def SkewArray(Genome):

skew = [0]

for i in range(len(Genome)):

if Genome[i] == 'G':

skew.append(skew[i] + 1)

elif Genome[i] == 'C':

skew.append(skew[i] - 1)

else:

skew.append(skew[i])

return skew

\*Skew diagram, min: Para este ejercicio se realiza primero el diagrama skew, se obtiene el valor que sea minimo y se busca dentro del código, este regresa la posición en donde este se encuentra.

def MinimumSkew(Genome):

positions = []

skew = SkewArray(Genome)

skmin = min(skew)

for i in range(len(skew)):

if skew[i] == skmin:

positions.append(i)

return positions

def SkewArray(Genome):

skew = [0]

for i in range(len(Genome)):

if Genome[i] == 'G':

skew.append(skew[i] + 1)

elif Genome[i] == 'C':

skew.append(skew[i] - 1)

else:

skew.append(skew[i])

return skew

* Para este ejercicio se busca contar las veces en las que dentro de dos secuencias hay discrepancias que luego se contabilizan.

def HammingDistance(Text1, Text2):

count = 0

for i in range(len(Text1)):

if Text1[i] != Text2[i]:

count += 1

return count

* Al comparar una secuencia con un patrón se pueden saber las posiciones en las que hay similitudes de hasta el rango de discrepancia que queramos. D es la cantidad de discrepancias que hay y se van evaluando rangos de la secuencia para ver en donde hay una cantidad igual o mayor de discrepancias establecidas por d.

def ApproximatePatternMatching(Text, Pattern, d):

positions = []

for i in range(len(Text)-len(Pattern)+1):

if HammingDistance(Text[i:i+ len(Pattern)],Pattern) <= d:

positions.append(i)

return positions

def HammingDistance(Text1, Text2):

count = 0

for i in range(len(Text1)):

if Text1[i] != Text2[i]:

count += 1

return count

* Count Motifs: Se brinda un listado de motifs los cuales se van a acomodar en una matriz, luego se va a contar por columna la cantidad de nucleótidos que tiene.

def Count(Motifs):

count = {}

k = len(Motifs[0])

for symbol in "ACGT":

count[symbol] = []

for j in range(k):

count[symbol].append(0)

t = len(Motifs)

for i in range(t):

for j in range(k):

symbol = Motifs[i][j]

count[symbol][j] += 1

return count

* Profile Motifs: Regresa la probabilidad con la que cada nucleótido aparece por cada columna

def Profile(Motifs):

t= len(Motifs)

k= len(Motifs[0])

Profile = Count(Motifs)

for i in Profile:

for j in range(k):

Profile[i][j] = Profile[i][j] / t

return Profile

def Count(Motifs):

count = {}

k = len(Motifs[0])

for symbol in "ACGT":

count[symbol] = []

for j in range(k):

count[symbol].append(0)

t = len(Motifs)

for i in range(t):

for j in range(k):

symbol = Motifs[i][j]

count[symbol][j] += 1

return count

* Consensus Motifs: El código compara los valores máximos en cada columna para poder determinar cual es la letra que mas aparece y poder decir en el consenso que el motif es con la mejor tendencia

def Consensus(Motifs):

consensus = ''

k = len(Motifs[0])

count = Count(Motifs)

for j in range(k):

m = 0

FreqSymbol = ""

for symbol in "ACGT":

if count[symbol][j] > m:

m = count[symbol][j]

FreqSymbol = symbol

consensus += FreqSymbol

return consensus

def Count(Motifs):

count = {}

k = len(Motifs[0])

for symbol in "ACGT":

count[symbol] = []

for j in range(k):

count[symbol].append(0)

t = len(Motifs)

for i in range(t):

for j in range(k):

symbol = Motifs[i][j]

count[symbol][j] += 1

return count

* Score Motifs: Devuelve la suma de los valores que no se encuentran dentro de las tendencias por cada columna.

def Score(Motifs):

consensus = Consensus(Motifs)

score = 0

for i in Motifs:

for j in range(len(i)):

if i[j] != consensus[j]:

score += 1

return score

def Consensus(Motifs):

consensus = ''

k = len(Motifs[0])

count = Count(Motifs)

for j in range(k):

m = 0

FreqSymbol = ""

for symbol in "ACGT":

if count[symbol][j] > m:

m = count[symbol][j]

FreqSymbol = symbol

consensus += FreqSymbol

return consensus

def Count(Motifs):

count = {}

k = len(Motifs[0])

for symbol in "ACGT":

count[symbol] = []

for j in range(k):

count[symbol].append(0)

t = len(Motifs)

for i in range(t):

for j in range(k):

symbol = Motifs[i][j]

count[symbol][j] += 1

return count

* Probability of a profile: toma como input la matriz con los profile (diccionario de listas) y una secuencia con la misma longitud, va tomando la probabilidad de de cada letra a como esta acomodada la matriz y todos los números que recolecta los multiplica para saber cual es la probabilidad de que se forme esa secuencia.

def Pr(Text, Profile):

p = 1

for i in range(len(Text)):

p = p \* Profile[Text[i]][i]

return p

* **Profile-most probable k-mer**, al introducer un número de amplitud para una secuencia va a buscar cual secuencia dentro del genoma con esa longitud va a tener la mayor probabilidad de que suceda.

def ProfileMostProbablePattern(Text, k, Profile):

max\_kmer = []

max\_prob = -1

for i in range(len(Text)-k+1):

prob = Pr(Text[i:i+k], Profile)

if prob > max\_prob:

max\_prob = prob

max\_kmer = Text[i:i+k]

return max\_kmer

def Pr(Text, Profile):

p = 1

for i in range(len(Text)):

p = p \* Profile[Text[i]][i]

return p

* Best Motifs

def GreedyMotifSearch(Dna, k, t):

BestMotifs = []

for i in range(0, t):

BestMotifs.append(Dna[i][0:k])

n = len(Dna[0])

for i in range(n-k+1):

Motifs = []

Motifs.append(Dna[0][i:i+k])

for j in range(1, t):

P = Profile(Motifs[0:j])

Motifs.append(ProfileMostProbablePattern(Dna[j], k, P))

if Score(Motifs) < Score(BestMotifs):

BestMotifs = Motifs

return BestMotifs

def ProfileMostProbablePattern(Text, k, Profile):

max\_kmer = []

max\_prob = -1

for i in range(len(Text)-k+1):

prob = Pr(Text[i:i+k], Profile)

if prob > max\_prob:

max\_prob = prob

max\_kmer = Text[i:i+k]

return max\_kmer

def Pr(Text, Profile):

p = 1

for i in range(len(Text)):

p = p \* Profile[Text[i]][i]

return p

def Score(Motifs):

consensus = Consensus(Motifs)

score = 0

for i in Motifs:

for j in range(len(i)):

if i[j] != consensus[j]:

score += 1

return score

def Consensus(Motifs):

consensus = ''

k = len(Motifs[0])

count = Count(Motifs)

for j in range(k):

m = 0

FreqSymbol = ""

for symbol in "ACGT":

if count[symbol][j] > m:

m = count[symbol][j]

FreqSymbol = symbol

consensus += FreqSymbol

return consensus

def Profile(Motifs):

t= len(Motifs)

k= len(Motifs[0])

Profile = Count(Motifs)

for i in Profile:

for j in range(k):

Profile[i][j] = Profile[i][j] / t

return Profile

def Count(Motifs):

count = {}

k = len(Motifs[0])

for symbol in "ACGT":

count[symbol] = []

for j in range(k):

count[symbol].append(0)

t = len(Motifs)

for i in range(t):

for j in range(k):

symbol = Motifs[i][j]

count[symbol][j] += 1

return count

* Pseudocount: Para este código se toma en cuenta que en los casos en los que hay una probabilidad cero de que ocurra haya un mínimo de probabilidad, esto mediante la propiedad de Laplace en la que se le suma uno a todas las variable y asi no se descarta la que da igual a 0. En el caso del profile ya que se añaden cuatro posiciones (+1) por cada letra en el sistema, se requiere dividir no solo por la longitud sino por esto mas 4

def ProfileWithPseudocounts(Motifs):

t= len(Motifs) + 4

k= len(Motifs[0])

Profile = CountWithPseudocounts(Motifs)

for i in Profile:

for j in range(k):

Profile[i][j] = Profile[i][j] / t

return Profile

def CountWithPseudocounts(Motifs):

count = {}

k = len(Motifs[0])

for symbol in "ACGT":

count[symbol] = []

for j in range(k):

count[symbol].append(1)

t = len(Motifs)

for i in range(t):

for j in range(k):

symbol = Motifs[i][j]

count[symbol][j] += 1

return count

* Greedy

def GreedyMotifSearchWithPseudocode(Dna, k, t):

BestMotifs = []

for i in range(0, t):

BestMotifs.append(Dna[i][0:k])

n = len(Dna[0])

for i in range(n-k+1):

Motifs = []

Motifs.append(Dna[0][i:i+k])

for j in range(1, t):

P = ProfileWithPseudocounts(Motifs[0:j])

Motifs.append(ProfileMostProbablePattern(Dna[j], k, P))

if Score(Motifs) < Score(BestMotifs):

BestMotifs = Motifs

return BestMotifs

def ProfileMostProbablePattern(Text, k, Profile):

max\_kmer = []

max\_prob = -1

for i in range(len(Text)-k+1):

prob = Pr(Text[i:i+k], Profile)

if prob > max\_prob:

max\_prob = prob

max\_kmer = Text[i:i+k]

return max\_kmer

def Pr(Text, Profile):

p = 1

for i in range(len(Text)):

p = p \* Profile[Text[i]][i]

return p

def Score(Motifs):

consensus = Consensus(Motifs)

score = 0

for i in Motifs:

for j in range(len(i)):

if i[j] != consensus[j]:

score += 1

return score

def Consensus(Motifs):

consensus = ''

k = len(Motifs[0])

count = CountWithPseudocounts(Motifs)

for j in range(k):

m = 0

FreqSymbol = ""

for symbol in "ACGT":

if count[symbol][j] > m:

m = count[symbol][j]

FreqSymbol = symbol

consensus += FreqSymbol

return consensus

def ProfileWithPseudocounts(Motifs):

t= len(Motifs) + 4

k= len(Motifs[0])

Profile = CountWithPseudocounts(Motifs)

for i in Profile:

for j in range(k):

Profile[i][j] = Profile[i][j] / t

return Profile

def CountWithPseudocounts(Motifs):

count = {}

k = len(Motifs[0])

for symbol in "ACGT":

count[symbol] = []

for j in range(k):

count[symbol].append(1)

t = len(Motifs)

for i in range(t):

for j in range(k):

symbol = Motifs[i][j]

count[symbol][j] += 1

return count

* Motifs. = Al ingresar un valor de profile, y un listado de secuencias regresa los Motifs mas probables que se puedan formar, esto al combinar el trabajo del Profile most probable pattern, para ir comparando cuales son los mas probables., para ir comparando cuales son los mas probables.

def Motifs(Profile, Dna):

Motifs = []

k = 4

for char in range(len(Dna)):

Motif = ProfileMostProbablePattern(Dna[char], k, Profile)

Motifs.append(Motif)

return Motifs

def ProfileMostProbablePattern(Text, k, Profile):

max\_kmer = []

max\_prob = -1

for i in range(len(Text)-k+1):

prob = Pr(Text[i:i+k], Profile)

if prob > max\_prob:

max\_prob = prob

max\_kmer = Text[i:i+k]

return max\_kmer

def Pr(Text, Profile):

p = 1

for i in range(len(Text)):

p = p \* Profile[Text[i]][i]

return p

* Random Motifs: Genera Motifs al azar con una amplitud k que toma una lista de secuencias y de esta las va probando para ver la probabilidad con la que aparece dentro de otro sistema.

import random

def RandomMotifs(Dna, k, t):

t = len(Dna)

l = len(Dna[0])

RandMotifs = []

for i in range(t):

R = random.randint(1,l-k)

RandMotifs.append(Dna[i][R:R+k])

return RandMotifs

* RandomizedMotifSearch: Evalua

import random

def RandomizedMotifSearch(Dna, k, t):

M = RandomMotifs(Dna, k, t)

BestMotifs = M

while True:

Profile = ProfileWithPseudocounts(M)

M = Motifs(Profile, Dna)

if Score(M) < Score(BestMotifs):

BestMotifs = M

else:

return BestMotifs

def RandomMotifs(Dna, k, t):

t = len(Dna)

n = len(Dna[0])

RandMotifs = []

for i in range(t):

R = random.randint(0,n-k)

RandMotifs.append(Dna[i][R:R+k])

return RandMotifs

def Score(Motifs):

consensus = Consensus(Motifs)

score = 0

for i in Motifs:

for j in range(len(i)):

if i[j] != consensus[j]:

score += 1

return score

def Consensus(Motifs):

consensus = ''

k = len(Motifs[0])

count = CountWithPseudocounts(Motifs)

for j in range(k):

m = 0

FreqSymbol = ""

for symbol in "ACGT":

if count[symbol][j] > m:

m = count[symbol][j]

FreqSymbol = symbol

consensus += FreqSymbol

return consensus

def ProfileWithPseudocounts(Motifs):

t= len(Motifs) + 4

k= len(Motifs[0])

Profile = CountWithPseudocounts(Motifs)

for i in Profile:

for j in range(k):

Profile[i][j] = Profile[i][j] / t

return Profile

def CountWithPseudocounts(Motifs):

count = {}

k = len(Motifs[0])

for symbol in "ACGT":

count[symbol] = []

for j in range(k):

count[symbol].append(1)

t = len(Motifs)

for i in range(t):

for j in range(k):

symbol = Motifs[i][j]

count[symbol][j] += 1

return count

def Motifs(Profile, Dna):

Motifs = []

k = 4

for char in range(len(Dna)):

Motif = ProfileMostProbablePattern(Dna[char], k, Profile)

Motifs.append(Motif)

return Motifs

def ProfileMostProbablePattern(Text, k, Profile):

max\_kmer = []

max\_prob = -1

for i in range(len(Text)-k+1):

prob = Pr(Text[i:i+k], Profile)

if prob > max\_prob:

max\_prob = prob

max\_kmer = Text[i:i+k]

return max\_kmer

def Pr(Text, Profile):

p = 1

for i in range(0,len(Text)):

p \*= Profile[Text[i]][i]

return p

* Probabilities: Toma un diccionario como Profile, donde hay varias probabilidades y las divide cada una por el total para normalizar y que la suma de todos sea igual a 1

Probabilities = {'A': 0.25, 'C': 0.25, 'G': 0.25, 'T': 0.25}

def Normalize(Probabilities):

sumProb = 0

newProb = {}

for i in Probabilities:

sumProb += Probabilities[i]

for key in Probabilities:

newProb[key] = Probabilities[key] / sumProb

return newProb

* WeightedDie: Toma como input un diccionario normalizado que tiene distintas probabilidades para cada k-mer, este genera un numero aleatorio float que va a decidir en tomar un valor de los kmer con respecto a las probabilidades de cada uno.

def WeightedDie(Probabilities):

n = random.uniform(0, 1)

for p in Probabilities:

n -= Probabilities[p]

if n <= 0:

return p

* ProfileGeneratedString: Recibe una secuencia, un profile con proabilidades y una k para el largo del kmer que van a generar un k-mer aleatorio basándose en las probabilidades que se proporcionan para dar con el mas certero, se genera un arreglo con las probabilidades con la función Pr , se normaliza y con base a este arreglo se regresa el kmer aleatorio.

def ProfileGeneratedString(Text, profile, k):

n = len(Text)

probabilities = {}

for i in range(0,n-k+1):

probabilities[Text[i:i+k]] = Pr(Text[i:i+k], profile)

probabilities = Normalize(probabilities)

return WeightedDie(probabilities)

def WeightedDie(Probabilities):

n = random.uniform(0, 1)

for p in Probabilities:

n -= Probabilities[p]

if n <= 0:

return p

def Normalize(Probabilities):

sumProb = 0

newProb = {}

for i in Probabilities:

sumProb += Probabilities[i]

for key in Probabilities:

newProb[key] = Probabilities[key] / sumProb

return newProb

def Pr(Text, Profile):

p = 1

for i in range(0,len(Text)):

p \*= profile[Text[i]][i]

return p

Gibbs Sampler: Con las formulas anteriores se evalua para hallar el mejor motif

def GibbsSampler(Dna, k, t, N):

BestMotifs = []

Motifs = RandomMotifs(Dna, k, t)

BestMotifs = Motifs

for i in range(0,N):

j = random.randint(0,t-1)

del Motifs[j]

profile = ProfileWithPseudocounts(Motifs)

Motifs.insert(j,ProfileGeneratedString(Dna[j], profile, k))

if Score(Motifs) < Score(BestMotifs):

BestMotifs = Motifs

return BestMotifs

def RandomMotifs(Dna, k, t):

t = len(Dna)

n = len(Dna[0])

RandMotifs = []

for i in range(t):

R = random.randint(0,n-k)

RandMotifs.append(Dna[i][R:R+k])

return RandMotifs

def ProfileGeneratedString(Text, profile, k):

n = len(Text)

probabilities = {}

for i in range(0,n-k+1):

probabilities[Text[i:i+k]] = Pr(Text[i:i+k], profile)

probabilities = Normalize(probabilities)

return WeightedDie(probabilities)

def WeightedDie(Probabilities):

n = random.uniform(0, 1)

for p in Probabilities:

n -= Probabilities[p]

if n <= 0:

return p

def Normalize(Probabilities):

sumProb = 0

newProb = {}

for i in Probabilities:

sumProb += Probabilities[i]

for key in Probabilities:

newProb[key] = Probabilities[key] / sumProb

return newProb

def Pr(Text, Profile):

p = 1

for i in range(len(Text)):

p = p \* Profile[Text[i]][i]

return p

def ProfileWithPseudocounts(Motifs):

t= len(Motifs) + 4

k= len(Motifs[0])

Profile = CountWithPseudocounts(Motifs)

for i in Profile:

for j in range(k):

Profile[i][j] = Profile[i][j] / t

return Profile

def CountWithPseudocounts(Motifs):

count = {}

k = len(Motifs[0])

for symbol in "ACGT":

count[symbol] = []

for j in range(k):

count[symbol].append(1)

t = len(Motifs)

for i in range(t):

for j in range(k):

symbol = Motifs[i][j]

count[symbol][j] += 1

return count

def Score(Motifs):

k = len(Motifs[0])

count = CountWithPseudocounts(Motifs)

consensus = Consensus(Motifs)

sum=0

row=0

for i in consensus:

sum1=0

for j in count:

if j!= i:

sum1 += count[j][row]

sum += sum1

row +=1

return sum

def Consensus(Motifs):

count = CountWithPseudocounts(Motifs)

consensus = ""

k = len(Motifs[0])

for j in range(k):

m = 0

frequentSymbol = ""

for symbol in "ACGT":

if count[symbol][j] > m:

m = count[symbol][j]

frequentSymbol = symbol

consensus += frequentSymbol

return consensus