Project 2 Protein Cleavage

Samuel Gaudin et Mathias Grau

May 19, 2023

1 Initialisation des données

On initialise les paramètres du modèle

[316]:
```
import numpy as np
p=13 # 13 acides aminés avant la liaison clivée
q=2 # 2 acides aminés après la liaison clivée
```

2 Lecture des fichiers à traiter

On créé une fonction qui va permettre d'extraire les données utiles d'un des fichier `.red` et qui renvoie un tableau de couples du type :

("séquence d'acides aminés","interprétation")

On définie les séquences de protéines de la manière suivante : avec $\mathcal{A} = \{A, ...Z\}$ l'ensemble des acides aminés.

Connaissant une séquence de protéine $(a_i)_{i\in 0,..,l-1}$ et la position de son site de clivage j : on peut définir la séquence:

$$a_{j-p}a_{j-p+1}...a_{j-1}a_j...a_{j+q-1} \in \mathcal{A}^{p+q}$$

[317]:
```
def parse_file(filename):
    pairs = []  # Tableau pour stocker les couples de chaînes de caractères

    with open(filename, 'r') as file:
        lines = file.readlines() # lit le fichier et stocke chaque ligne dans␣
↪une liste

        # Parcourir les lignes en sautant de 3 en 3
        for i in range(1, len(lines), 3):
            line2 = lines[i].strip()
            line3 = lines[i + 1].strip()
            pair = (line2, line3)
            pairs.append(pair)

    return pairs, len(pairs)
```

[318]:
```
# Exemple d'utilisation avec un fichier "red.txt"
filename = './data/SIG_13.red'
pairs,N = parse_file(filename)
print(f"Number of lines : {N}")
print(f"5 first pairs : ")
for i in range (5):
    print(pairs[i])
```

```
Number of lines : 1408
5 first pairs :
('MASKATLLLAFTLLFATCIARHQQRQQQQNQCQLQNIEALEPIEVIQAEA',
'SSSSSSSSSSSSSSSSSSSCMMMMMMMMMMMMMMMMMMMMMMMMMMM')
('MARSSLFTFLCLAVFINGCLSQIEQQSPWEFQGSEVWQQHRYQSPRACRLE',
'SSSSSSSSSSSSSSSSSSSSCMMMMMMMMMMMMMMMMMMMMMMMMMMM')
('MLVMAPRTVLLLLSAALALTETWAGSHSMRYFYTSVSRPGRGEPRFISVGYVDD',
'SSSSSSSSSSSSSSSSSSSSSSCMMMMMMMMMMMMMMMMMMMMMMMMMMM')
('MKLSKSTLVFSALLVILAAASAAPANQFIKTSCTLTTYPAVCEQSLSAYAKT',
'SSSSSSSSSSSSSSSSSSSSSCMMMMMMMMMMMMMMMMMMMMMMMMMMM')
('MANKLFLVCATLALCFLLTNASIYRTVVEFEEDDASNPVGPRQRCQKEFQQ',
'SSSSSSSSSSSSSSSSSSSSCMMMMMMMMMMMMMMMMMMMMMMMMMMM')
```

3 Statistiques sur l'acide aminé correspondant au site de clivage

[319]:
```
def find_letters_at_c_position(pairs):
    letters = []  # Tableau pour stocker les lettres correspondantes à␣
 ↪l'emplacement 'C'

    for pair in pairs:
        line2 = pair[0]
        line3 = pair[1]

        # Recherche de l'emplacement de la lettre 'C' dans le deuxième élément␣
 ↪du couple
        c_index = line3.find('C')

        if c_index != -1:
            # Ajout de la lettre correspondante dans le premier élément du␣
 ↪couple ainsi que sa position
            letters.append((line2[c_index],c_index))

    return letters
```

[320]:
```
# Utilisation avec la liste de couples 'pairs'
found_letters = find_letters_at_c_position(pairs)

# Affichage des lettres trouvées
```

```
print(f"Pour la première paire : \n{pairs[0][0]} \n{pairs[0][1]} \nOn a la␣
 ↪lettre {found_letters[0][0]} à la position {found_letters[0][1]}")
```

```
Pour la première paire :
MASKATLLLAFTLLFATCIARHQQRQQQQNQCQLQNIEALEPIEVIQAEA
SSSSSSSSSSSSSSSSSSSSCMMMMMMMMMMMMMMMMMMMMMMMMMMM
On a la lettre R à la position 20
```

On compte désormais les occurrences de chaque acide aminé en tant que site de clivage.

[321]:
```
def count_letter_occurrences(letter_positions):
    letter_counts = {}  # Dictionnaire pour stocker les occurrences des lettres

    for letter, _ in letter_positions:
        if letter in letter_counts:
            letter_counts[letter] += 1
        else:
            letter_counts[letter] = 1

    return letter_counts
```

[322]:
```
# Utilisation avec le tableau de couples 'letter_positions'
occurrences = count_letter_occurrences(found_letters)

# Affichage des occurrences des lettres
for letter, count in occurrences.items():
    print(f"Lettre : {letter}, Occurrence : {count}")
```

```
Lettre : R, Occurrence : 36
Lettre : Q, Occurrence : 173
Lettre : G, Occurrence : 65
Lettre : A, Occurrence : 298
Lettre : S, Occurrence : 116
Lettre : F, Occurrence : 38
Lettre : W, Occurrence : 11
Lettre : Y, Occurrence : 27
Lettre : V, Occurrence : 64
Lettre : E, Occurrence : 125
Lettre : H, Occurrence : 26
Lettre : M, Occurrence : 17
Lettre : I, Occurrence : 42
Lettre : D, Occurrence : 100
Lettre : L, Occurrence : 68
Lettre : T, Occurrence : 49
Lettre : N, Occurrence : 42
Lettre : K, Occurrence : 70
Lettre : P, Occurrence : 14
Lettre : C, Occurrence : 27
```

4 Statistique sur les N sous-séquences de peptides signaux

On dispose de N séquences d'acides aminés avec un site de clivage connu. On extrait les peptides signals qui correspondent à une succession d'acides aminés de taille $p+q$ avec p acides aminés avant le site de clivage et q acides aminés après.

[323]:
```
def extract_substrings(pairs, p, q):
    extracted_substrings = []  # Tableau pour stocker les sous-chaînes extraites

    for pair in pairs:
        line2 = pair[0]
        line3 = pair[1]

        # Recherche de l'emplacement de la lettre 'C' dans le deuxième élément␣
 ↪du couple
        c_index = line3.find('C')

        if c_index != -1:
            # Extraction des sous-chaînes dans le premier élément du couple
            start_index = max(1, c_index-p)
            end_index = min(len(line2), c_index + q )
            substring = line2[start_index:end_index]
            extracted_substrings.append(substring)

    return extracted_substrings


extracted_substrings_test = extract_substrings(pairs, p, q)

print(f"Pour la première paire : \n{pairs[0][0]} \n{pairs[0][1]} \nOn a la␣
 ↪lettre {found_letters[0][0]} à la position {found_letters[0][1]} ce qui donne␣
 ↪la sous-chaine \n{extracted_substrings_test[0]}")
```

```
Pour la première paire :
MASKATLLLAFTLLFATCIARHQQRQQQQNQCQLQNIEALEPIEVIQAEA
SSSSSSSSSSSSSSSSSSSSCMMMMMMMMMMMMMMMMMMMMMMMMMMMM
On a la lettre R à la position 20 ce qui donne la sous-chaine
LLAFTLLFATCIARH
```

On va maintenant créer un dictionnaire afin de stocker le nombre d'occurence de chaque acide aminé par rapport à leur position dans le peptide signal

[324]:
```
def count_letter_occurrences_with_position(extracted_substrings):
    letter_counts = {}  # Dictionnaire pour stocker les occurrences de chaque␣
 ↪lettre et sa position

    for substring in extracted_substrings:
        for position, letter in enumerate(substring):
```

```
            letter_position = (letter, position)

            if letter_position in letter_counts:
                letter_counts[letter_position] += 1
            else:
                letter_counts[letter_position] = 1

    return letter_counts
```

On définie les fonction f et c comme énoncées dans l'énoncé, qui compte le nombre d'occurences de l'acide aminé a $\in \mathcal{A}laposition i$

$$\begin{array}{rccc} c & : & \mathcal{A}, [[0, p+q]] & \to & \mathbb{N} \\ & & (a, i) & \mapsto & c(a, i) \end{array}$$

On créé la fonction f définie dans le sujet qui calcul la fréquence d'apparition de l'acide aminé a\in $\mathcal{A}$\$laposition i$

$$\begin{array}{rccc} f & : & \mathcal{A}, [[0, p+q]] & \to & \mathbb{R} \\ & & (a, i) & \mapsto & \frac{c(a,i)}{N} \end{array}$$

[325]:
```
# Utilisation avec la liste de sous-chaînes extraites 'extracted_substrings'
occurrences_with_position =␣
 ↪count_letter_occurrences_with_position(extracted_substrings_test)

# Affichage des occurrences de chaque lettre avec sa position
k=0
for  letter_position, count in occurrences_with_position.items():
    letter, position = letter_position
    print(f"Lettre : {letter}, Position : {position}, Occurrences : {count}")
    k+=1
    if k>10:
        break
```

```
Lettre : L, Position : 0, Occurrences : 460
Lettre : L, Position : 1, Occurrences : 520
Lettre : A, Position : 2, Occurrences : 178
Lettre : F, Position : 3, Occurrences : 104
Lettre : T, Position : 4, Occurrences : 56
Lettre : L, Position : 5, Occurrences : 442
Lettre : L, Position : 6, Occurrences : 393
Lettre : F, Position : 7, Occurrences : 74
Lettre : A, Position : 8, Occurrences : 193
Lettre : T, Position : 9, Occurrences : 117
Lettre : C, Position : 10, Occurrences : 71
```

[326]:
```
def f(N,occurrences_with_position, lettre, position,alpha=0,d=1):
    letter_position = (lettre, position)
```

```
    if letter_position in occurrences_with_position:
        return (occurrences_with_position[letter_position]+alpha)/(N+alpha*d)
    else:
        return alpha/(N+alpha*d)
```

[327]:
```
# Utilisation avec le dictionnaire 'occurrences_with_position'
lettre = 'L'
position = 0
occurrence_count = f(N,occurrences_with_position, lettre, position,alpha=1,d=p+q)

print(f"Fréquence d'occurrences de la lettre '{lettre}' à la position {position}:
 ↪ {occurrence_count}")
```

```
Fréquence d'occurrences de la lettre 'L' à la position 0: 0.3239634574841883
```

On implémente maintenant la fonction g qui évalue la fréquence d'un acide aminé $a \in \mathcal{A}$ sans se soucier de sa position dans la chaine de caractère :

$$\begin{array}{rccc} g & : & \mathcal{A} & \rightarrow & \mathbb{R} \\ & & a & \mapsto & g(a) \end{array}$$

[328]:
```
def g(extracted_substrings, lettre, N, alpha=0,d=1):
    occurrence_count = 0
    for substring in extracted_substrings:
        occurrence_count += substring.count(lettre)
    if occurrence_count == 0:
        return alpha/(N+alpha*d)
    frequency = (occurrence_count +alpha )/ (N+alpha*d)
    return frequency
```

[329]:
```
# Utilisation avec 'extracted_substrings'
lettre = 'L'
result = g(extracted_substrings_test, lettre, N,alpha=1,d=p+q)

print(f"Fréquence d'occurrence de la lettre '{lettre}' par chaîne: {result}")
```

```
Fréquence d'occurrence de la lettre 'L' par chaîne: 3.068868587491216
```

On implémente ensuite la fonction s comme étant définie :

$$\forall a \in \mathcal{A}, \forall i \in [[0, p+q]] \quad \begin{array}{rccc} s & : & \mathcal{A}, [[0, p+q]] & \rightarrow & \mathbb{R} \\ & & (a, i) & \mapsto & log(f(a,i)) - log(g(a)) \end{array}$$

[330]:
```
def s(N,extract_substrings,lettre,position,alpha=1,d=1):
    ␣
 ↪occurrences_with_position=count_letter_occurrences_with_position(extract_substrings)
    return np.log(f(N,occurrences_with_position,lettre,position,alpha,d))-np.
 ↪log(g(extract_substrings,lettre,N,alpha,d))
```

On définie logiquement la fonction score suivante : Pour tout mot

$$w = a_{j-p}a_{j-p+1}...a_{j-1}a_j...a_{j+q-1} \in \mathcal{A}^{p+q}$$

On a le $q-1$ score défini par :

$$\begin{array}{rccc} score & : & \mathcal{A}^{p+q} & \to & \mathbb{R} \\ & & w & \mapsto & \sum_{i=0}^{p+q-1} s(a_i, i) \end{array}$$

[331]:
```
def score(w,N,extracted_substrings,alpha=1,d=1):
    sum=0.0
    for i in range(len(w)):
        sum += s(N,extracted_substrings,w[i],i,alpha,d)
    return sum
```

[332]:
```
def threshold_classifier(w, N, extracted_substrings, alpha=1, d=1,␣
↪threshold_value=0):
    print("Beginning Test")

    score_sum = score(w, N, extracted_substrings, alpha, d)

    print(f"Score : {score_sum}")
    print(f"Threshold : {threshold_value}")

    if score_sum >= threshold_value:
        return 1  # Classe positive
    else:
        return 0  # Classe négative
```

[351]:
```
from tqdm import tqdm
# On calcule le minimum de score pour la séquence SIG_13
min=0
mot = ''
for substring in tqdm(extracted_substrings_test):
    score_sum = score(substring, N, extracted_substrings_test, alpha=1, d=p+q)
    if score_sum<min:
        min=score_sum
        mot=substring

# On obtient un score de -48.24458384914466 qui va servie de threshold pour la␣
↪suite
score_min= -48.24458384914466
print(f'Le score minimal obtenu dans la substring SIG_13 est {score_min}')
```

```
Le score minimal obtenu dans la substring SIG_13 est -48.24458384914466
```

Exemple d'utilisation de ce classifieur sur une chaine de caractère étant un peptide signal présent dans la séquence donnée

[334]:
```
w='LLAFTLLFATCIARH' #qui est un mot de la sous-chaine SIG_13 donc un peptide signal
threshold_value = score_min

score_obtenu = score(w, N, extracted_substrings_test, alpha=2, d=p+q)
print(f"Test sur le potentiel peptide signal : {w}")
test= threshold_classifier(w, N, extracted_substrings_test, alpha=1, d=p+q,threshold_value=threshold_value)
print('Réponse :',test)
```

```
Test sur le potentiel peptide signal : LLAFTLLFATCIARH
Beginning Test
Score : -37.08975645397594
Threshold : -48.24458384914466
Réponse : 1
```

5 Quelques SVM Kernels

5.1 Vectorisation des données

On crée les vecteurs de taille $26(p+q)$ qui est la concatenation de $p+q$ vecteurs de taille 26 où la ième valeur est 1 si elle correspond à la lettre considérée (dans l'ordre alphabétique) et 0 sinon.

[335]:
```
import string
def create_encoded_vectors(substring_extracted, p, q):
    alphabet = string.ascii_uppercase
    letter_to_index = {letter: index for index, letter in enumerate(alphabet)}

    encoded_vectors = []
    for substring in substring_extracted:
        vector = np.zeros(26 * (p + q))

        for i, letter in enumerate(substring):
            if letter in letter_to_index:
                index = letter_to_index[letter]
                vector[i * 26 + index] = 1

        encoded_vectors.append(vector)

    return encoded_vectors
```

[353]:
```
print(f"First extracted substring     {extracted_substrings_test[0]}")
alphabet = string.ascii_uppercase
letter_to_index = {letter: index for index, letter in enumerate(alphabet)}
encoded_vector = create_encoded_vectors(extracted_substrings_test, p, q)[0]
print(f"First extracted vector  {encoded_vector.astype(int)}")
indices = np.argwhere(encoded_vector == 1).flatten()%26
```

```
print(f'Verification {"".join([alphabet[indices[i]] for i in␣
 ↪range(len(indices))])}')
```

```
First extracted substring     LLAFTLLFATCIARH
First extracted vector  [0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0
 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0
 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0
 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0
 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0
 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0
 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
Verification LLAFTLLFATCIARH
```

Le produit scalaire entre 2 vecteurs donne le nombre de lettres communes entre les 2 chaines de caractères

[337]:
```
print(f"First extracted substring     {extracted_substrings_test[0]}")
print(f"Second extracted substring    {extracted_substrings_test[1]}")

first_extracted_vector = create_encoded_vectors(extracted_substrings_test, p,␣
 ↪q)[0]
second_extracted_vector = create_encoded_vectors(extracted_substrings_test, p,␣
 ↪q)[1]

num_common_letters = np.sum(first_extracted_vector * second_extracted_vector)

print(f"Number of common letters : {num_common_letters}")
```

```
First extracted substring     LLAFTLLFATCIARH
Second extracted substring    FLCLAVFINGCLSQI
Number of common letters : 2.0
```

5.2 Matrice de probabilité

On définit la matrice de probabilité $M(x, y)$ pour toute paire d'acide aminé (x, y)

[338]:
```
def similarity_matrix(string1,string2):
    alphabet = string.ascii_uppercase
    letter_to_index = {letter: index for index, letter in enumerate(alphabet)}
    matrix=np.zeros((len(alphabet),len(alphabet)))
    for i in range(len(string1)):
        indice1=letter_to_index[str(string1[i])]
        indice2=letter_to_index[str(string2[i])]
        if indice1==indice2:
```

```
            matrix[indice1][indice2]+=1
        else :
            matrix[indice1][indice2]+=1
            matrix[indice2][indice1]+=1
    return matrix
```

[339]:
```
word1=extracted_substrings_test[0]
word2=extracted_substrings_test[1]
sim_matrix=similarity_matrix(word1,word2)
print(f'Matrice de similitude entre {word1} et {word2} :')
print(sim_matrix.astype(int))
```

```
Matrice de similitude entre LLAFTLLFATCIARH et FLCLAVFINGCLSQI :
[[0 0 1 0 0 0 0 0 0 0 0 0 1 0 0 0 0 1 1 0 0 0 0 0 0]
 [0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
 [1 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
 [0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
 [0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
 [0 0 0 0 0 0 0 0 1 0 0 3 0 0 0 0 0 0 0 0 0 0 0 0 0]
 [0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0]
 [0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
 [0 0 0 0 0 1 0 1 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0]
 [0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
 [0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
 [0 0 0 0 0 3 0 0 1 0 0 1 0 0 0 0 0 0 0 0 1 0 0 0 0]
 [0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
 [1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
 [0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
 [0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
 [0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0]
 [0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0]
 [1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
 [1 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
 [0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
 [0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
 [0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
 [0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
 [0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
 [0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]]
```

On définit maintenant le score de similarité entre 2 séquences d'acides aminés en prenant la trace de la matrice de similarité :

$$\begin{array}{rccc} S & : & \mathcal{A}^{p+q}, \mathcal{A}^{p+q} & \to & \mathbb{N} \\ & & (a,b) & \mapsto & \sum_{i=0}^{n-1} M(a_i, b_i) \end{array}$$

[340]:
```
def S(string1,string2):
    return np.trace(similarity_matrix(string1,string2))
```

[341]:
```
print(f"Number of common letters at the same position by getting the trace of␣
↪the similarity matrix : {S(word1,word2)}")
```

```
Number of common letters at the same position by getting the trace of the
similarity matrix : 2.0
```

On définit finalement la fonction $log - kernel$ comme suit :

$$logK(a,b) = \sum_{i=-p}^{q-1} \phi_i(a_{p+i}, b_{p+i})$$

où

$$\phi_i(x,y) = \begin{cases} s(x,i) + s(y,i) & si\ x \neq y \\ s(x,i) + log(1 + e^{s(x,i)}) & si\ x = y \end{cases}$$

[342]:
```
def log_kernel(a,b, N, extracted_substring,p,q,alpha=1):
    sum=0.0
    d=p+q
    for i in range(len(a)):
        if a[i]!=b[i]:
            ␣
↪sum+=s(N,extracted_substring,a[i],i-p,alpha,d)+s(N,extracted_substring,b[i],i-p,alpha,d)
        else:
            sum+=s(N,extracted_substring,a[i],i-p,alpha,d)+np.log(1+np.
↪exp(s(N,extracted_substring,a[i],i-p,alpha,d)))

    return (sum)
```

[343]:
```
res = log_kernel(word1, word2, N, extracted_substrings_test, p, q,alpha=2)
print(f'Résultat : {res} pour le mot {word1} et {word2}')
```

```
Résultat : -177.43938709139957 pour le mot LLAFTLLFATCIARH et FLCLAVFINGCLSQI
```

[354]:
```
def create_substrings_dataset(pairs, p, q):
    extracted_substrings = []  # Tableau pour stocker les sous-chaînes extraites

    for pair in pairs:
        line2 = pair[0]
        line3 = pair[1]

        # Recherche de l'emplacement de la lettre 'C' dans le deuxième élément␣
↪du couple
        c_index = line3.find('C')

        shift = np.random.choice([-4, 0, 4],p=[0.1, 0.8, 0.1])
        # On tire un nombre aléatoire entre -4, 0 et 4
        # avec une probabilité de 0.1, 0.8 et 0.1
        # respectivement pour décaler ou non la sous-chaîne extraite
```

```
        if shift == 0:
            y=1
        else:
            y=0

        if c_index != -1 and c_index + shift >= 0 and c_index + shift <␣
↪len(line2):
            # Extraction des sous-chaînes dans le premier élément du couple
            start_index = max(1, c_index-p+shift)
            end_index = min(len(line2), c_index + q+shift)
            substring = line2[start_index:end_index]
            extracted_substrings.append((substring,y))

    return extracted_substrings
```

[345]:
```
extracted_substring_set_test = create_substrings_dataset(pairs, p, q)
print(extracted_substring_set_test[:3])
```

```
[('LLAFTLLFATCIARH', 1), ('FLCLAVFINGCLSQI', 1), ('LLSAALALTETWAGS', 1)]
```

[346]:
```
test = []
for i in range(N):
    test.append(extracted_substring_set_test[i][0])
extracted_substring_vectorised = create_encoded_vectors(test, p, q)
extracted_substring_set_vectorised=[]
for i in range(N):
    extracted_substring_set_vectorised.
↪append((extracted_substring_vectorised[i],extracted_substring_set_test[i][1]))
```

[347]:
```
from sklearn.svm import SVC
from sklearn.metrics import accuracy_score

training_rate=0.8 # Taux d'exemples d'entraînement

X_train = []  # Caractéristiques des exemples d'entraînement
y_train = []  # Étiquettes des exemples d'entraînement
X_test = []   # Caractéristiques des exemples de test
y_test = []   # Étiquettes des exemples de test

# Préparation des données d'entraînement et de test
for i in range(int(training_rate*N)):
    X_train.append(extracted_substring_set_vectorised[i][0])  # Caractéristiques␣
↪des exemples d'entraînement
    y_train.append(extracted_substring_set_vectorised[i][1])  # Étiquettes des␣
↪exemples d'entraînement
for i in range(int(training_rate*N),N):
```

```
    X_test.append(extracted_substring_set_vectorised[i][0])    # Caractéristiques␣
↪des exemples de test
    y_test.append(extracted_substring_set_vectorised[i][1])    # Étiquettes des␣
↪exemples de test

# Création du modèle SVM avec un kernel gaussien (RBF)
svm_model = SVC(kernel='rbf')

# Entraînement du modèle
svm_model.fit(X_train, y_train)

# Prédiction des classes des exemples de test
y_pred = svm_model.predict(X_test)

# Évaluation du modèle
accuracy = accuracy_score(y_test, y_pred)
print(f"Accuracy : {accuracy}")
```

```
Accuracy : 0.8546099290780141
```

5.3 Etude pour différentes fonctions

[361]:
```
def svm(kernel):
    filenames = ['./data/SIG_13.red','./data/EUKSIG_13.red','./data/GRAM-SIG_13.
↪red','./data/GRAM+SIG_13.red']
    print(f"Kernel : {kernel}")
    print()
    for i,filename in enumerate(filenames):
        pairs,N = parse_file(filename)
        extracted_substring_set_test = create_substrings_dataset(pairs, p, q)
        test = []
        for i in range(N):
            test.append(extracted_substring_set_test[i][0])
        extracted_substring_vectorised = create_encoded_vectors(test, p, q)
        extracted_substring_set_vectorised=[]
        for i in range(N):
            extracted_substring_set_vectorised.
↪append((extracted_substring_vectorised[i],extracted_substring_set_test[i][1]))
        X_train = []  # Caractéristiques des exemples d'entraînement
        y_train = []  # Étiquettes des exemples d'entraînement
        X_test = []   # Caractéristiques des exemples de test
        y_test = []   # Étiquettes des exemples de test

        # Préparation des données d'entraînement et de test
        for i in range(int(training_rate*N)):
            X_train.append(extracted_substring_set_vectorised[i][0])  #␣
↪Caractéristiques des exemples d'entraînement
```

```
            y_train.append(extracted_substring_set_vectorised[i][1])   #␣
↪Étiquettes des exemples d'entraînement
        for i in range(int(training_rate*N),N):
            X_test.append(extracted_substring_set_vectorised[i][0])    #␣
↪Caractéristiques des exemples de test
            y_test.append(extracted_substring_set_vectorised[i][1])    #␣
↪Étiquettes des exemples de test

        # Création du modèle SVM avec un kernel gaussien (RBF)
        svm_model = SVC(kernel=kernel)

        # Entraînement du modèle
        svm_model.fit(X_train, y_train)

        # Prédiction des classes des exemples de test
        y_pred = svm_model.predict(X_test)

        # Évaluation du modèle
        accuracy = accuracy_score(y_test, y_pred)
        print(f'Filename : {filename}')
        print(f"Accuracy : {accuracy}")
        print(f'False positive rate : {1-accuracy}')
        print()
```

[362]:
```
svm('rbf')
```

```
Kernel : rbf

Filename : ./data/SIG_13.red
Accuracy : 0.8262411347517731
False positive rate : 0.17375886524822692

Filename : ./data/EUKSIG_13.red
Accuracy : 0.8706467661691543
False positive rate : 0.12935323383084574

Filename : ./data/GRAM-SIG_13.red
Accuracy : 0.8490566037735849
False positive rate : 0.15094339622641506

Filename : ./data/GRAM+SIG_13.red
Accuracy : 0.8571428571428571
False positive rate : 0.1428571428571429
```

[363]:
```
svm('linear')
```

```
Kernel : linear
```

```
Filename : ./data/SIG_13.red
Accuracy : 0.875886524822695
False positive rate : 0.12411347517730498

Filename : ./data/EUKSIG_13.red
Accuracy : 0.8756218905472637
False positive rate : 0.12437810945273631

Filename : ./data/GRAM-SIG_13.red
Accuracy : 0.9433962264150944
False positive rate : 0.05660377358490565

Filename : ./data/GRAM+SIG_13.red
Accuracy : 0.8214285714285714
False positive rate : 0.1785714285714286
```

[364]:
```
svm('sigmoid')
```

```
Kernel : sigmoid

Filename : ./data/SIG_13.red
Accuracy : 0.900709219858156
False positive rate : 0.099290780141844

Filename : ./data/EUKSIG_13.red
Accuracy : 0.8557213930348259
False positive rate : 0.14427860696517414

Filename : ./data/GRAM-SIG_13.red
Accuracy : 0.9433962264150944
False positive rate : 0.05660377358490565

Filename : ./data/GRAM+SIG_13.red
Accuracy : 0.8214285714285714
False positive rate : 0.1785714285714286
```

[365]:
```
svm('poly')
```

```
Kernel : poly

Filename : ./data/SIG_13.red
Accuracy : 0.7943262411347518
False positive rate : 0.2056737588652482

Filename : ./data/EUKSIG_13.red
Accuracy : 0.8059701492537313
```

```
False positive rate : 0.19402985074626866

Filename : ./data/GRAM-SIG_13.red
Accuracy : 0.7358490566037735
False positive rate : 0.26415094339622647

Filename : ./data/GRAM+SIG_13.red
Accuracy : 0.6071428571428571
False positive rate : 0.3928571428571429
```

5.4 Conclusion

Nous sommes donc parvenus à étudier des peptides signaux de longueur $p + q$ pour en faire des ensembles d'entrainement afin de pouvoir juger si une séquence aléatoire consitue, ou non, un peptide signal. Les taux de précision sont aux alentour de 90% pour les dataset utilisés