Bioinformatics
COMPGI10
Bioinformatics Report
Date: Friday 24th March, 2017
Graduate Assignment

Bioinformatics

Protein Classification By feature extraction

Sofiane MAHIOU

Computer Science, UCL, London, WC1E 6BT, UK

Abstract

Motivation : The goal of this assignment is to provide an automated system that is able to classify proteins (Amino Acid sequences) into four classes each being a subcellular locations : **[Cytosolic, Secreted, Nuclear, Mitochondrial]**. The subcellular location is an important information as it provides more knowledge about the protein function, it is therefore, very helpful to be able to automatically extract that information for any given protein.
Results : Using a *Random Forest Classifer* we manage to reach a **67% cross-validation accuracy**.
Improvements: In order to improve the results of the classifier, deepening the feature extraction method seems to be the way to go. Another method would be to use neural network techniques.
Contact: ucabsm1@ucl.ac.uk

1 Introduction

Currently, There is a growing need for fully automated methods to analyse amino acid sequences. One of the process that need to be automated is **the identification of the protein's subcellular location**. This problem can be splitted into two sub problems :

- **feature extraction:** the goal of this task is to choose the features that would allow an efficient classification, to be more precise, the chosen features should allow to easily seperate the sequences into classes or groups which will then be matched with the various subcellular locations
- **classification:** once the features obtained, it is then necessary to choose a fitting classification algorithm that will use the various features selected as a *vector representation* of each sequence that will then be fed to the classification algorithm during both training and testing.

It is however, possible to avoid splitting the problem into two sub-problems by using methods that have been designed to classify sequences of variables lengths such as :

- **HMM**
- **Recurrent neural networks & Seq2Seq Models**
- **1D Convolutional Neural Networks**

Although these methods usually yield better results than the methods presented before, the results obtained are far harder to interpret as these systems behave as **"black boxes"** and it's quite difficult to identify what was *learned*.

Therefore, the first approach was used in order to ease the analysis of the results and the task was splitted into a **feature-extraction** task and **classification** task. In order to identify what features might be useful to this problem, a research phase was realized where several research papers on the same subject have been studied, and used as a reference to select several features.

2 Sequence preprocessing :

It seems important to mention that before proceeding the **feature extraction** phase, it was necessary to preprocess the data due to the presence of unexpected characters : **U, B, X** in the amino acid sequences.

- **X:** Given that "X" refers to "any amino acid" we randomly replace it by a given amino acid among the 20 amino acids
- **B:** "B" refers to either *asparagine* and *aspartic acid*. It is therefore automatically replaced by one or the other
- **U:** the "U" amino acid is simply removed from the sequence for lack of a better solution

This preprocessing step, will only affect 64 sequences out of more than 9000, therefore these changes are unlikely to heavily influence the results but will allow and easier implementation of the various features.

3 Feature Extraction

We identify three different types of features:

- **Amino acids composition:** This refers to count or frequency of each amino acid in the sequence to analyse.
- **Protein's properties:** This refers to the various chimical and biological properties that a given protein can have such as *aromaticity, hydrophobicity, iso-electric point ...etc*
- **Subsequence-based features :** This refers to various existing methods that aim to extract relevant discriminative subsequences which *presence or count* will then be used as features to classify the proteins

The goal will then be combine several features from each type in order to optimize the results of the classification. The process through which the features to extract were selected was mainly based on what several research papers on this field advised and recommended. Once all the features were pre-selected and implemented, the final feature selection process was done while attempting classification, by trying to optmize both results and speed. Below you can find all the features that were considered as well as whether or not they were used in the final model.

3.1 sequence length: **Used**

The sequence found in the training set have a varying length, therefore, it seemed fitting the provide the length of a given sequence as a feature.
This was confirmed by the results obtained after classification using various classifiers, all of them providing better results when using this feature

3.2 Amino acids Counts:

This feature refers to the number of times each amino acid appears in the sequence. The sequences studied being of variable lengths, giving the raw count of each amino acid seemed unfit as each dimension would heavily vary.

3.3 Amino acids frequency: **Used**

In order to fix the problem cited above, we use the frequency instead as it was recommended. Indeed, the frequency being a value between 0 and 1, this leads each dimension to be normalized, allowing the model to compare **amino acid composition** of sequences of various length.

3.4 isoelectric point: **Used**

The isoelectronic point or isoionic point is the pH at which the amino acid does not migrate in an electric field.
As presented by Q.-B. Gao *et al.* (2005), **iso-electric point** is among the top features to use when attempting the predict the subcellular location of a protein.

3.5 Presence of specific Sequences : N-Grams **Used**

As explained in Saidi *et al.* (2010), The presence of specific sequences of a specified length as well as the number of appearances can be powerful features. The whole goal is then to identify whic sub sequences or patterns are relevant and discriminative enough.
To do so two techniques were attempted:

- **tf-idf:** tf-idf is a classic processing alogrithm that allows to extract "relevant" information from a sequence of "words" or "N-Grams" *i.e.* sequences of characters of length N. For example, for a sequence length or 3, this algorithm allowed us to extract the following sequences to look for : **["WW","MWW", "CWW", "WWM", "WMW", "WCW", "CWM", "WCM", "MCW", "WWH", "FWW"]**
- **Descriminative Descriptors:** this is another method that was presented in Saidi *et al.* (2010), Given a set of n sequences, assigned to P families/classes F1, F2 .., FP, this method consists of building substrings called Discriminative Descriptors DD which allow to discriminate a family F_i from other families F_j, with $i = 1..P$ and $i \neq j$. it behaves similarily to tf-idf but use the additional information that several classes exist.

The method chosen was a combination of both. Initially, a list of potentially interesting sequences is chosen using **tf-idf**. This is a method used to extract an first list of potentially interesting sub-sequences. These sub-sequences are then reduced to relevant sequences using the **DD** method approach which consists of only selecting sub-sequences which are far more frequent within a selected **class of sequences** then in the remaining sequences.

3.6 Nuclear Export Signals: **Used**

This feature that is presented in Xua *et al.* (2012), describes the following pattern as an efficient discriminative pattern : $\phi_1 - X_3 - \phi_2 - X_2 - \phi_3 - X - \phi_4$. Positions ϕ_3 and ϕ_4 of this prevalent pattern are dominated by the five traditional hydrophobic residues **Leu[L], Ile[I],Val[V], Met[M], and Phe[F]**.
This feature did lead to improvements, however they were not as significant as expected.

3.7 Nuclear Localization Signals: **Used**

This feature has also been presented in Xua *et al.* (2012), it refers to the count of subsequences of at least 5 **positively charged amino acids** *.ie* meaning one of the following : **lys[K], arg[R], his[H]**.
Again this feature lead to slight improvement over all models.

3.8 Protein's properties: **Used**

In addition to what was presented above the following properties were attempted as suggested by Q.-B. Gao *et al.* (2005)

- **Hydrophobicity**
- **Aromaticity**
- **Molecular Weight**

3.9 Beggining and End of sequences:

Each feature presented above was computed for the **full sequence** as well as the **first 50 amino acids** and the **last 50 amino acids** of each sequence. This aims to identify trends and patterns not only overall but also specific to the begining and the end of the sequences. Indeed, the length of sequences would make it difficult to extract information solely related to the begining or the end of a given sequence. Isolating the begining and the end of each sequence, before computing the same features, seems to be a proper way of removing the "noise" due to then length of most amino acid sequences.

4 Classification Methods

Several classification methods are available, using previous knowledge about **Machine Learning - Classifcation** problems as well as classifiers referenced by other research papers of this same field. Each classifier, is optimized in order to maximize its efficiency.

4.1 Random Forest

The *Random Forest Algorithm*, is an **ensemble** algorithm that will generate a specified number of random trees, each one using a specified number of **randomly selected features** in order first to train and then to predict the class of a given sample.
Once each **random decision tree** provides an answer, the algorithms average the results out generating a confidence or probability for each class. The most probable class is then chosen.

4.2 Logistic Regression Classifier

The logistic regression used in this case is a **One vs Rest** type of model. Basically for each class i , we generate a linear model $f_i(X_j) = \beta_i.X_j$ this model will then be used to provide the probability of a given sample X_j to belong to a given class i is then $ln(\frac{p_i}{1-p_i}) = \beta_i.X_j$. Again the class chosen is the one with highest probability.

4.3 Ridge Regression Classifier

The Ridge Regression is a simple **regularized linear regression**. It follows this equation :

$$\hat{x} = (A^\top A + \Gamma^\top \Gamma)^{-1} A^\top \mathbf{y}$$

$\Gamma = \alpha I$, the α parameter being manually chosen in order to optimize the result.

Once the A matrix is *learned*, we can compute $A\mathbf{x} = \mathbf{y}$, and choose for the class the dimension of y that has the maximum value. In order to get the probability, running a **softmax** function on the output seems to be fitting.

4.4 SVM Classifier

The SVM algorithm chosen uses the **rbf kernel** as well as a **One vs Rest** strategy. the rbf kernel follows the following expression :

$$K(\mathbf{x}, \mathbf{x'}) = \exp\left(-\frac{||\mathbf{x} - \mathbf{x'}||^2}{2\sigma^2}\right) = \varphi(\vec{x}_i) \cdot \varphi(\vec{x}_j).$$

The $\vec{w}$ classification vector is then defined as follow : $\vec{w} = \sum_{i=1}^{n} c_i y_i \varphi(\vec{x}_i)$, where the c_i are chosen in order to maximize :

$$f(c_1 \dots c_n) = \sum_{i=1}^{n} c_i - \frac{1}{2} \sum_{i=1}^{n} \sum_{j=1}^{n} y_i c_i k(\vec{x}_i, \vec{x}_j) y_j c_j$$

Once the parameters chosen, for each class we compute:

$$f_i = \left(\left[\sum_{i=1}^{n} c_i y_i k(\vec{x}_i, \vec{z}) \right] + b \right).$$

which will provide us a confidence value for each class.

4.5 Ensemble Model

An **ensemble or mixture** of the optimized models presented above were tried, two strategies were adopted:

- **linear combination:** Computation of the weighted average of the probabilities provided per each models, the weights have been chosen experimentally in order to maximize the results
- **maximized confidence:** For each sample the model with the best confidence is chosen

Although, the **maximized confidence** ensemble model seemed to yield better results, none of these two methods gave significantly better results than the random Forest while being slower to compute.

5 Results Analysis

Using the features presented above, we obtained the following results :

Table 1. Cross Validation Results

Model	Fold 1	Fold 2	Fold 3	Fold 4	Fold 5	**average**
Logistic Regression Classifier	0.628	0.642	0.589	0.602	0.62	**0.615**
Random Forest Classifier	0.676	0.680	0.676	0.665	0.669	**0.673**
SVM Classifier	0.638	0.655	0.645	0.641	0.663	**0.648**
Ridge Regression	0.664	0.640	0.646	0.644	0.618	**0.642**

As it can be seen the Random Forest seems to perform the best, with an average accurac of **0.673**. This however seems to be bellow results obtained in the various research paper focusing on this same task as they usually reach accuracies ranging between 75% an 80%.

5.1 Confusion Matrices

In order to understand the nature of the errors, the following confusion matrices have been computed and averaged over 5-Folds:

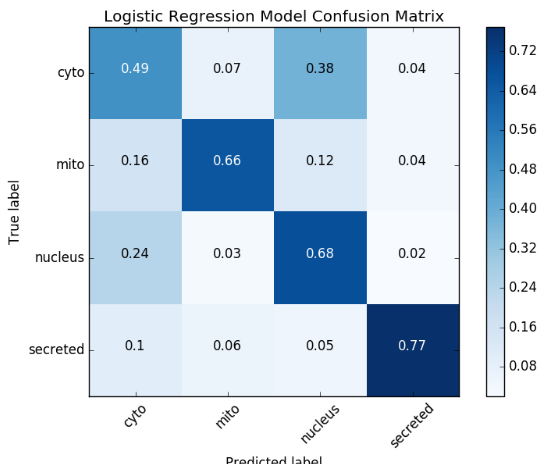

Fig. 1. Logistic Regression Model

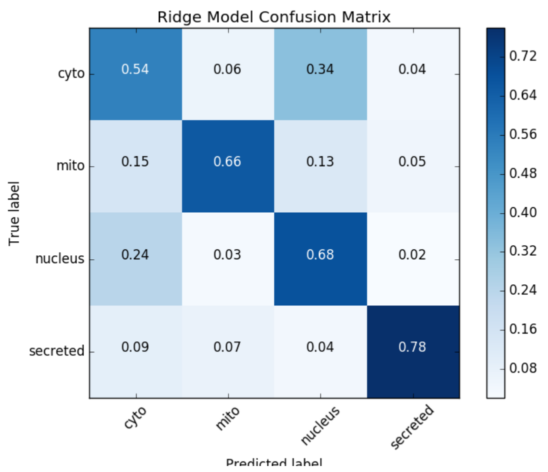

Fig. 2. Ridge Regression Model

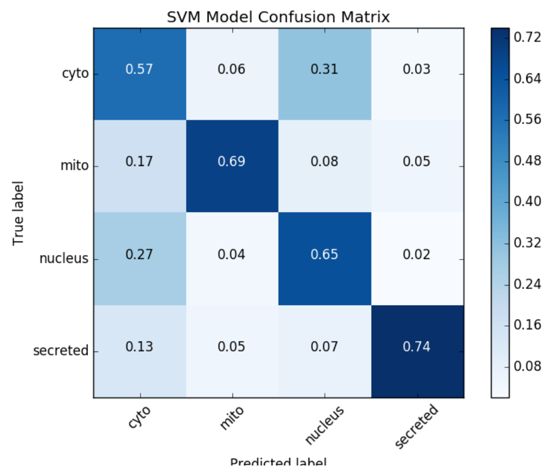

Fig. 3. SVM Model

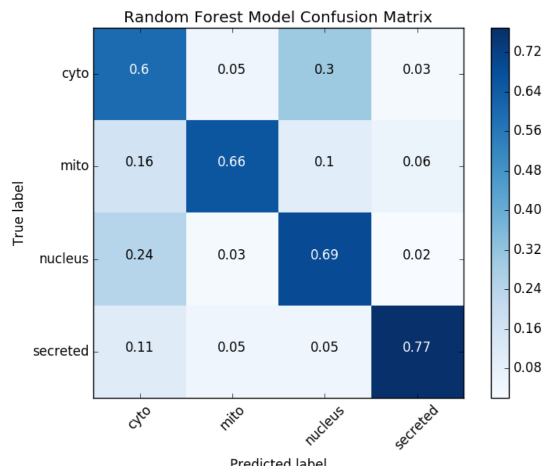

Fig. 4. Random Forest Model

As it can be noticed, all the models have very similar confusion matrices. All of them seem to struggle identifying **cyto** class properly and mainly confuses it with the **nucleus** class. This also explains why the **ensemble model** doesn't seem to help. Indeed, if all the models have the same **weaknesses**, combining them will not improve the overall results.

This suggests that some supplementary features are needed in order to distinguish the **cyto** and nucleus class from each other. According, to what is presented in the research literature on the subject, these features are most likely going to be **sequence-related** such as the presence of a specific patern in the protein.

5.2 Confidence on Mistakes

Table 2. Confidence on erroneous prediction

True Label	cyto	mito	nucleus	secreted	**average**
Random Forest Classifier	0.527	0.453	0.498	0.442	**0.497**

This idea is confirmed by the fact, on average, the model is not too confident when it is making a mistake as the confidence for these cases is around 50%. This confirms that, either a bigger training set size or adding some more features might help.

5.3 Prediction on blind test set

As required by the Assignment, you can find bellow the subcellular localization predictions realized on a blind test.

SEQ677 Cyto Confidence 36%
SEQ231 Cyto Confidence 30%
SEQ871 Secreted Confidence 68%
SEQ388 Nucleus Confidence 78%
SEQ122 Nucleus Confidence 69%
SEQ758 Nucleus Confidence 81%
SEQ333 Cyto Confidence 47%
SEQ937 Cyto Confidence 68%
SEQ351 Cyto Confidence 52%
SEQ202 Mito Confidence 71%
SEQ608 Mito Confidence 78%
SEQ402 Mito Confidence 67%
SEQ433 Secreted Confidence 35%
SEQ821 Secreted Confidence 79%
SEQ322 Nucleus Confidence 90%
SEQ982 Nucleus Confidence 82%
SEQ951 Nucleus Confidence 46%
SEQ173 Nucleus Confidence 57%
SEQ862 Mito Confidence 66%
SEQ224 Cyto Confidence 45%

References

R. Saidi, M. Maddouri and EM. Nguifo (2010)
Protein sequences classification by means of feature extraction with substitution matrices, *BMC Bioinformatics*

Q-B Gao, Z-Z Wang, C Yan, Y-H Du (2005)
Prediction of protein subcellular location using a combined feature of sequence, *FEBS Letters*

D Xua, A Farmera, G Colletta, N V. Grishinb, Y M Chooka (2012)
Sequence and structural analyses of nuclear export signals in the NESdb database , *MBoC*

5.4 Main Code

6 Appendix: Code

6.1 Main Code

```
################### LOADING PYTHON PACKAGES #########################
import numpy as np
from time import time
import matplotlib.pyplot as plt
import matplotlib.patches as mpatches
import os.path as path
import os
import tensorflow as tf
import sys
import platform

from Bio import SeqIO
from Bio.SeqUtils.ProtParam import ProteinAnalysis


from sklearn.linear_model import RidgeClassifier, LogisticRegression
from sklearn import svm
from sklearn.model_selection import train_test_split, GridSearchCV, KFold, cross_val_score
from sklearn.ensemble import RandomForestClassifier as RFC
from sklearn.metrics import mean_squared_error, make_scorer, confusion_matrix


import re
import itertools

from keras.models import Sequential
from keras.layers import Dense, Activation
from keras.layers.normalization import BatchNormalization



##################### Useful Paths ################################
scriptPath = os.path.realpath(__file__)
rootPath = os.path.sep.join(scriptPath.split(os.path.sep)[:-1])
dataPath = rootPath + os.path.sep + 'data' +  os.path.sep
seed  = 42

###################### Auxiliary function ##########################
def plot_confusion_matrix(cm, classes, normalize=True, title='Confusion matrix', cmap=plt.cm.Blues):


        if normalize:
                cm = np.floor((cm.astype('float') / cm.sum(axis=1)[:, np.newaxis])*100).astype(int)/100
                print("Normalized confusion matrix")
        else:
                print('Confusion matrix, without normalization')

        #print(cm)

        plt.imshow(cm, interpolation='nearest', cmap=cmap)
        plt.title(title)
        plt.colorbar()
        tick_marks = np.arange(len(classes))
        plt.xticks(tick_marks, classes, rotation=45)
        plt.yticks(tick_marks, classes)

        thresh = cm.max() / 2.
        for i, j in itertools.product(range(cm.shape[0]), range(cm.shape[1])):
                plt.text(j, i, cm[i, j], horizontalalignment="center", color="white" if cm[i, j] > thresh else "black")
```

```
    plt.tight_layout()
    plt.ylabel('True label')
    plt.xlabel('Predicted label')


def sklearn_Grid_Search(model, parameters, X, Y, n_folds = 4):
    # grid search cross validation
    GSCV = GridSearchCV(model, parameters, cv=KFold(n_splits=n_folds, shuffle=True, random_state=42), verbose = 3)
    print(X.shape)
    print(Y.shape)
    GSCV.fit(X, Y)

    print("Best parameters set found on development set:")
    print(GSCV.best_params_)

    print("Grid scores on development set:")
    means = GSCV.cv_results_['mean_test_score']
    stds = GSCV.cv_results_['std_test_score']
    for mean, std, params in zip(means, stds, GSCV.cv_results_['params']):
        print("%0.3f (+/-%0.03f) for %r" % (mean, std * 2, params))

    return GSCV


def grid_search(model, parameters, X, Y, seed=seed, n_folds = 10):
    # grid search cross validation
    print('BIGINING GRID SEACH')
    grid_CV = GridSearchCV(model, parameters, cv=KFold(n_splits=n_folds, shuffle=True, random_state=seed), verbose = 3)
    grid_CV.fit(X, Y)

    print("Best parameters set found on development set:")
    print(grid_CV.best_params_)

    print("Grid scores on development set:")
    means = grid_CV.cv_results_['mean_test_score']
    stds = grid_CV.cv_results_['std_test_score']
    for mean, std, params in zip(means, stds, grid_CV.cv_results_['params']):
        print("%0.3f (+/-%0.03f) for %r" % (mean, std * 2, params))
    return grid_CV


def printResults(results ,index):
        if (platform.system().lower() == 'windows'):
                os.system('cls')
        else:
                os.system('clear')

        for i in range(index+1):
                print('--------------- EPOCH : ' + str(i) + ' ---------------' )
                print('Test-loss : %.4f'  %(results[i][0]) + ' - Test-Accuracy %.4f'  %(results[i][1]) )
                print('Train-loss : %.4f'  %(results[i][2]) + ' - Train-Accuracy %.4f'  %(results[i][3]) )


#################### LOADING FOLDERS ##############################

def loadFastaFiles(fileList):
        index = 0

        ## Initializing array
        tags = []
    names = []
    sequences = []
    lengthArray = []
    setOfAminoAcids = []
    ## For each file
    for name in fileList:
            print('LOADING␣ELEMENTS␣FROM␣FILE␣:␣' + name)
            secondIndex = 0
            ## LOAD file
            fasta_sequences =  SeqIO.parse(open(dataPath + name + '.fasta'),'fasta') # The function I miss
            for fasta in fasta_sequences:
                    seqName, sequence = fasta.id, str(fasta.seq)
                    sequence = re.sub('X', ["A", "R", "N", "D", "C", "Q", "E", "G", "H", "I", "L", "K", "M", "F", "P", "S", "T
                        ", "W", "Y", "V" ][np.random.randint(0,20)], sequence)
                    sequence = re.sub('U', '', sequence)
                    sequence = re.sub('B', ['N', 'D'][np.random.randint(0,2)], sequence)

                    setOfAminoAcids = list(set(setOfAminoAcids).union(set(sequence)))

                    secondIndex += 1

                    ## Adding sequqnce name to array
                    names.append(seqName)
                    ## addind sequence to sequence array
                    sequences.append(sequence)
                    ## Adding the tag
                    tagArray = np.zeros(len(fileList))
                    tagArray[index] = 1
                    tags.append(tagArray)

                    ## append the sequence length the length array
                    lengthArray.append(len(sequence))
                    ## append to the list of different amino-acids
            print('Number␣of␣Samples␣for␣'+ name + '␣:␣' ,secondIndex)
            index += 1

    return names, sequences, tags, lengthArray, setOfAminoAcids

def featurize(sequences,setOfAminoAcids,lengthArray, isForSVM = False):
    specific_sequences = ['WW','MWW', 'CWW', 'WWM', 'WMW', 'WCW', 'CWM', 'WCM', 'MCW', 'WWH', 'FWW', 'WCWE', 'WMYH', 'WMYI', '
        WMYL', 'WMYM', 'WMYQ', 'WMYV', 'WMYW', 'WMYY', 'QPWR']
    initial_sequences = ['WWF', 'WMM', 'WQW', 'HWC', 'FWC', 'HWD', 'CMW', 'VMW', 'MM', 'MHV']
    final_sequences = ['MW', 'CMM', 'WCM', 'WHF', 'PMW', 'AWW', 'CM', 'MVC', 'WFY', 'YWC']

    unambiguousAminoAcids = [letter for letter in 'ACDEFGHIKLMNPQRSTVWY']
    full_seq = np.array([ProteinAnalysis(sequences[j]) for j in range(len(sequences))])
    initial_seq = np.array([ProteinAnalysis(sequences[j][:50]) for j in range(len(sequences))])
    final_seq = np.array([ProteinAnalysis(sequences[j][-50:]) for j in range(len(sequences))])
    initial_seq_100 = np.array([ProteinAnalysis(sequences[j][:100]) for j in range(len(sequences))])
    final_seq_100 = np.array([ProteinAnalysis(sequences[j][-100:]) for j in range(len(sequences))])



    print('BUILDING␣DICTIONNARY␣')
    featureDictionnary = dict()

    print('LENGTH␣FEATURE')
    featureDictionnary['length'] = np.array([[len(sequences[j])] for j in range(len(sequences))])

    print('FREQUENCY␣FEATURE')
    featureDictionnary['frequency'] = np.array([[full_seq[j].get_amino_acids_percent()[i] for i in setOfAminoAcids] for j in
        range(len(sequences))])
    featureDictionnary['initialFrequency'] = np.array([[initial_seq[j].get_amino_acids_percent()[i] for i in setOfAminoAcids]
         for j in range(len(sequences))])
```

```
featureDictionnary['finalFrequency'] = np.array([[final_seq[j].get_amino_acids_percent()[i] for i in setOfAminoAcids]  for
     j in range(len(sequences))])
featureDictionnary['initialFrequency100'] = np.array([[initial_seq_100[j].get_amino_acids_percent()[i] for i in
     setOfAminoAcids]  for j in range(len(sequences))])
featureDictionnary['finalFrequency100'] = np.array([[final_seq_100[j].get_amino_acids_percent()[i] for i in
     setOfAminoAcids]  for j in range(len(sequences))])

print('COUNT_FEATURE')
featureDictionnary['counts'] = np.array([[full_seq[j].count_amino_acids()[i] for i in setOfAminoAcids] for j in range(len(
     sequences))])
featureDictionnary['initialcounts'] = np.array([[initial_seq[j].count_amino_acids()[i] for i in setOfAminoAcids]  for j in
     range(len(sequences))])
featureDictionnary['finalcounts'] = np.array([[final_seq[j].count_amino_acids()[i] for i in setOfAminoAcids]  for j in
     range(len(sequences))])
featureDictionnary['initialcounts100'] = np.array([[initial_seq_100[j].count_amino_acids()[i] for i in setOfAminoAcids]
     for j in range(len(sequences))])
featureDictionnary['finalcounts100'] = np.array([[final_seq_100[j].count_amino_acids()[i] for i in setOfAminoAcids]  for j
     in range(len(sequences))])

print('ISO-ELECTRIC_POINT_FEATURE')
featureDictionnary['iso'] =  np.array([[full_seq[j].isoelectric_point()] for j in range(len(sequences))])
featureDictionnary['initialIso'] =  np.array([[initial_seq[j].isoelectric_point()] for j in range(len(sequences))])
featureDictionnary['finalIso'] =  np.array([[final_seq[j].isoelectric_point()] for j in range(len(sequences))])
featureDictionnary['initialIso100'] =  np.array([[initial_seq_100[j].isoelectric_point()] for j in range(len(sequences))])
featureDictionnary['finalIso100'] =  np.array([[final_seq_100[j].isoelectric_point()] for j in range(len(sequences))])

print('GRAVY_FEATURE')
featureDictionnary['gravy']  = np.array([[full_seq[j].gravy()] for j in range(len(sequences))])
featureDictionnary['initialGravy']  = np.array([[initial_seq[j].gravy()] for j in range(len(sequences))])
featureDictionnary['finalGravy']  = np.array([[final_seq[j].gravy()] for j in range(len(sequences))])
featureDictionnary['initialGravy100']  = np.array([[initial_seq_100[j].gravy()] for j in range(len(sequences))])
featureDictionnary['finalGravy100']  = np.array([[final_seq_100[j].gravy()] for j in range(len(sequences))])

print('MOLECULAR_WEIGHT_FEATURE')

if not(isForSVM):

     featureDictionnary['weight']  = np.array([[full_seq[j].molecular_weight()] for j in range(len(sequences))])
     featureDictionnary['initialweight']  = np.array([[initial_seq[j].molecular_weight()] for j in range(len(sequences)
          )])
     featureDictionnary['finalweight']  = np.array([[final_seq[j].molecular_weight()] for j in range(len(sequences))])
     featureDictionnary['initialweight100']  = np.array([[initial_seq_100[j].molecular_weight()] for j in range(len(
          sequences))])
     featureDictionnary['finalweight100']  = np.array([[final_seq_100[j].molecular_weight()] for j in range(len(
          sequences))])

print('AROMATICITY_FEATURE')
featureDictionnary['aromaticity'] = np.array([[full_seq[j].aromaticity()] for j in range(len(sequences))])
featureDictionnary['initialAromaticity'] = np.array([[initial_seq[j].aromaticity()] for j in range(len(sequences))])
featureDictionnary['finalAromaticity'] = np.array([[final_seq[j].aromaticity()] for j in range(len(sequences))])

# print('INSTABILITY INDEX FEATURE')
# featureDictionnary['instability'] = np.array([[full_seq[j].instability_index()] for j in range(len(sequences))])
# featureDictionnary['initialinstability'] = np.array([[initial_seq[j].instability_index()] for j in range(len(sequences))
     ])
# featureDictionnary['finalinstability'] = np.array([[final_seq[j].instability_index()] for j in range(len(sequences))])


# print('SECONDARY-STRUCTURE FEATURE')
# featureDictionnary['secondary'] = np.array([full_seq[j].secondary_structure_fraction() for j in range(len(sequences))])

# print('SPECIFIC-SEQUENCES FEATURE')
# featureDictionnary['specific'] = np.array([[sequence.count(specific_sequence) for specific_sequence in
     specific_sequences] for sequence in sequences])
# featureDictionnary['initialSpecific'] = np.array([[sequence[:50].count(specific_sequence) for specific_sequence in
     initial_sequences] for sequence in sequences])
# featureDictionnary['finalSpecific'] = np.array([[sequence[-50:].count(specific_sequence) for specific_sequence in
     final_sequences] for sequence in sequences])

print('SPECIFIC-SEQUENCES_FEATURE')
featureDictionnary['specific'] = np.array([[ int(specific_sequence in sequence) for specific_sequence in
     specific_sequences] for sequence in sequences])
featureDictionnary['initialSpecific'] = np.array([[ int(specific_sequence in sequence[:50]) for specific_sequence in
     initial_sequences] for sequence in sequences])
featureDictionnary['finalSpecific'] = np.array([[ int(specific_sequence in sequence[-50:]) for specific_sequence in
     final_sequences] for sequence in sequences])

print('NES_COUNT_FEATURE')
```

```
        featureDictionnary['NES'] = np.array([[len(re.findall('[L|I|V|F|M][A-Z][A-Z][A-Z][L|I|V|F|M][A-Z][A-Z][L|I|V|F|M][A-Z][L|I
             |V|F|M]', sequence))] for sequence in sequences])
        featureDictionnary['InitialNES'] = np.array([[len(re.findall('[L|I|V|F|M][A-Z][A-Z][A-Z][L|I|V|F|M][A-Z][A-Z][L|I|V|F|M][A
             -Z][L|I|V|F|M]', sequence[:50]))] for sequence in sequences])
        featureDictionnary['finalNES'] = np.array([[len(re.findall('[L|I|V|F|M][A-Z][A-Z][A-Z][L|I|V|F|M][A-Z][A-Z][L|I|V|F|M][A-Z
             ][L|I|V|F|M]', sequence[-50:]))] for sequence in sequences])

        print('POS LIST FEATURE')

        tempPOS = [re.findall('[K|R|H]*', sequence) for sequence in sequences]
        featureDictionnary['5POS'] = np.array([[ len([subseq for subseq in tempPOS[i] if len(subseq) == 5])] for i in range(len(
             sequences))])
        featureDictionnary['4POS'] = np.array([[ len([subseq for subseq in tempPOS[i] if len(subseq) > 4])] for i in range(len(
             sequences))])
        tempInitialPOS = [re.findall('[K|R|H]*', sequence[:50]) for sequence in sequences]
        featureDictionnary['5InitialPOS'] = np.array([[ len([subseq for subseq in tempInitialPOS[i] if len(subseq) == 5])] for i
             in range(len(sequences))])
        featureDictionnary['4InitialPOS'] = np.array([[ len([subseq for subseq in tempInitialPOS[i] if len(subseq) > 4])] for i in
              range(len(sequences))])
        tempFinalPOS = [re.findall('[K|R|H]*', sequence[-50:]) for sequence in sequences]
        featureDictionnary['5FinalPOS'] = np.array([[ len([subseq for subseq in tempFinalPOS[i] if len(subseq) == 5])] for i in
             range(len(sequences))])
        featureDictionnary['4FinalPOS'] = np.array([[ len([subseq for subseq in tempFinalPOS[i] if len(subseq) > 4])] for i in
             range(len(sequences))])


        # print('N-TERMINUS SEQUENCE')
        # tempNTerminus = [re.findall('[V|I|L|M|F|A|C]*', sequence[:50]) for sequence in sequences]
        # featureDictionnary['NTerminus'] = np.array([[ len([subseq for subseq in tempNTerminus[i] if len(subseq) >= 5])] for i in
              range(len(sequences))])

        print('TRANSFORMING INTO ARRAY')

        featureSize = 0
        for key in featureDictionnary:
                featureSize += featureDictionnary[key].shape[1]

        print('NUMBER OF FEATURES : ', featureSize)

        featurized_sequence = np.zeros([len(sequences), featureSize])
        currentIndex = 0
        for key in featureDictionnary:
                featurized_sequence[:,currentIndex:currentIndex+featureDictionnary[key].shape[1]] = featureDictionnary[key]
                currentIndex += featureDictionnary[key].shape[1]

        return featurized_sequence


def findCommonSubstrig(sequences, sequenceLengths):
        initialSequenceIndex = np.argmin(sequenceLengths)
        initialSequence = sequences[initialSequenceIndex]

        longestSequence = initialSequence
        for i in range(len(sequences)):
                if i != initialSequenceIndex:
                        if not(longestSequence in sequences[i]):
                                print('blue')

###################### EXTRACTING ARGUMENTS #########################
args = sys.argv[1:] ## getting the arguments

ridge_option = True
svm_option = False
rf_option = False
nn_option = False
lr_option = False
ensemble_option = False
if ('-ridge' in args):
        ridge_option = True
```

```
elif('--svm' in args):
        ridge_option = False
        svm_option = True
elif('--rf' in args):
        ridge_option = False
        rf_option = True
elif('--nn' in args):
        ridge_option = False
        nn_option = True
elif('--lr' in args):
        ridge_option = False
        lr_option = True
elif('--ensemble' in args):
        ensemble_option = True
        ridge_option = False
        maxMode = False
        meanMode = True
        if ('--max' in args):
                meanMode = False
                maxMode = True

train = False
if('--train' in args) or ('-t' in args):
        train = True

nFolds = 2
foldString = '--nFold'
if (foldString in args):
        nFolds = int(args[args.index(foldString)+1])

testMode = False
if ('--test' in args):
        testMode = True


###################### Loading the Data ##############################

print('LOADING THE DATA SET')

dataNames = ['cyto', 'mito', 'nucleus', 'secreted']
names, sequences, tags, lengthArray, setOfAminoAcids  = loadFastaFiles(dataNames)
maxSeqLen = np.max(np.array(lengthArray))

print('Number of Amino Acids possible : ', len(setOfAminoAcids))
print('Number of Samples : ', len(sequences))
print('Maximum Sequence Length : ', maxSeqLen)
print('Number Of Different lengths : ', len(set(lengthArray)))
print('Shuffling the data : ')

print('FEATURIZATION')

X_train_all = featurize(sequences,setOfAminoAcids,lengthArray, isForSVM= svm_option)

if not(nn_option):
        Y_train_all = np.array(tags).argmax(axis = 1)
else:
        Y_train_all = np.array(tags)
names = np.array(names)
```

```
if train and not(nn_option):
        if ridge_option :
                classifier = RidgeClassifier()
                alpha_range = np.logspace(−6, 5, num=11, endpoint=False)
                model_params = {'alpha': alpha_range}

        elif svm_option :
                classifier = svm.SVC()
                model_params = {'C': [1, 2, 10, 100, 1000], 'gamma': [0.001, 0.0001], 'kernel': ['rbf']}

        elif rf_option :
                classifier = RFC()
                estimators_range = np.int_(np.linspace(200, 1000, num=9, endpoint=True))
                min_samples_leaf = np.int_(np.linspace(1, 40, num=20, endpoint=True))
                max_features = np.linspace(0.1, 0.8, num=8, endpoint=True)
                model_params = {'n_jobs' : [−1], 'n_estimators': estimators_range, 'min_samples_leaf': min_samples_leaf, '
                     max_features' : max_features}
        elif lr_option:
                c_range = np.int_(np.linspace(1, 1000, num=200, endpoint=True))
                model_params = {'n_jobs' : [−1], 'C': c_range}
                classifier = LogisticRegression(C=1)

        classifier = grid_search(classifier, model_params, X_train_all, Y_train_all, seed)
        print('BEST_PARAMS')
        print(classifier.best_params_)
        print('BEST_SCORE')
        print(classifier.best_score_)

else:
        if ridge_option :
                print('RIDGE_CLASSIFIER')

                classifier = RidgeClassifier(alpha = 1.0)
        elif svm_option :
                print('SVM_CLASSIFIER')

                classifier = svm.SVC(C = 10, gamma = 0.0001)
        elif rf_option :
                print('RANDOM_FOREST')

                classifier = RFC(max_features =  0.4, n_estimators =  750, min_samples_leaf =  1, n_jobs =  −1)
        elif lr_option:
                print('LOGISTIC_REGRESSION')
                classifier = LogisticRegression(C=1)
        elif nn_option:

                x_train, x_test, y_train, y_test = train_test_split(X_train_all, Y_train_all, test_size=0.1, random_state=0)

                print(x_train[0])
                print(y_train[0])
                model = Sequential()
                model.add(Dense(units=128, input_dim=X_train_all.shape[1]))
                model.add(Activation('relu'))
                model.add(BatchNormalization())
                # model.add(Dense(units=64, input_dim=X_train_all.shape[1]))
                # model.add(Activation('relu'))
                # model.add(BatchNormalization())
                model.add(Dense(units=4))
                model.add(Activation('softmax'))
                model.compile(loss='categorical_crossentropy',
              optimizer='adadelta',
                metrics=['accuracy'])
                  model.fit(x_train, y_train, epochs=50, batch_size=10, validation_data=(x_test, y_test))


if ensemble_option:
        X_svm_all = featurize(sequences,setOfAminoAcids,lengthArray, isForSVM =  True)


train_ratio = 0.9
confusion_matrices = np.zeros([nFolds, 4,4])
y_proba = np.zeros([nFolds, Y_train_all.shape[0] - int(np.floor(Y_train_all.shape[0]*train_ratio)), 4])
y_mistake = np.zeros([nFolds])
y_mistake_perClass = np.zeros([nFolds, 4])

if not(nn_option) and not(testMode):
        print('CROSS_VALIDATING_....')
        for i in range(nFolds):
                if ensemble_option:
                        random_permutation = np.random.permutation(Y_train_all.shape[0])
                        train_index = random_permutation[:int(np.floor(Y_train_all.shape[0]*train_ratio))]
                        test_index = random_permutation[int(np.floor(Y_train_all.shape[0]*train_ratio)):]

                        x_svm_train = X_svm_all[train_index]
                        x_train = X_train_all[train_index]

                        x_svm_test = X_svm_all[test_index]
                        x_test = X_train_all[test_index]

                        y_train = Y_train_all[train_index]
                        y_test = Y_train_all[test_index]

                        # x_train, x_test, y_train, y_test = train_test_split(X_train_all, Y_train_all, test_size=0.1,
                              random_state=0)

                        listOfClassiferNames = ["SVMs", "Random_Forest" ,"Ridge_Classifier", "Logistic_Regression"]
                        listOfClassifer = [svm.SVC(C = 10, gamma = 0.0001, probability = True), RFC(max_features =  0.4,
                              n_estimators =  750, min_samples_leaf =  1, n_jobs =  -1), RidgeClassifier(alpha = 1.0) ,
                              LogisticRegression(C=1)]
                        listOfFeatures_train = [x_svm_train] + [x_train] * (len(listOfClassifer)-1)


                        for j in range(len(listOfClassifer)):
                                print("Model_Being_Trained_:_", listOfClassiferNames[j])
                                listOfClassifer[j].fit(listOfFeatures_train[j],y_train)

                        listOfFeatures_test = [x_svm_test] + [x_test] * (len(listOfClassifer) - 1)

                        probabilities = np.zeros([len(listOfClassifer), x_test.shape[0], 4])
                        for j in range(len(listOfClassifer)):
                                print("Model_Predicting_:_", listOfClassiferNames[j])
                                try:
                                        probabilities[j] = listOfClassifer[j].predict_proba(listOfFeatures_test[j])
                                except:
                                        scores = np.exp(listOfClassifer[j].decision_function(listOfFeatures_test[j]))
                                        probabilities[j] = scores/(1+scores)

                        coefList = [0.25,0.25,0.25,0.25]
                        if meanMode :
                                predictions = np.average(probabilities, axis = 0, weights = coefList).argmax(axis = 1)
                        else:
                                predictions = np.max(probabilities, axis = 0).argmax(axis = 1)

                        dev_accuracy = sum((predictions == y_test).astype(int))/predictions.shape[0]
        else:

            temp_classifier = classifier
            x_train, x_test, y_train, y_test = train_test_split(X_train_all, Y_train_all, test_size=0.1)
            temp_classifier.fit(x_train,y_train)
            dev_accuracy =  temp_classifier.score(x_test, y_test)
            predictions = temp_classifier.predict(x_test)
            # if ridge_option:
            #       print(temp_classifier.decision_function(x_test))
            try:
                y_proba[i] = classifier.predict_proba(x_test)
            except:
                scores = classifier.decision_function(x_test)
                y_proba[i] = scores/(1+scores)
            y_mistake[i] = np.mean(y_proba[i][y_proba[i].argmax(axis = 1) != y_test].max(axis = 1))
            testArray = np.array([np.mean(y_proba[i][y_test == j][y_proba[i][y_test == j].argmax(axis = 1) != j].max(
               axis = 1)) for j in range(4)])
            y_mistake_perClass[i] = testArray

        confusion_matrices[i] = confusion_matrix(y_test, predictions)
        print('Fold NÂ°',str(i))
        print('SCORE : ', dev_accuracy)

    if not(ensemble_option):

        y_full_mistake = np.mean(y_mistake)
        y_mistake_perClass = np.mean(y_mistake_perClass, axis = 0)

        print('Average False Confidence: ', y_full_mistake)
        print('Average False Confidence Per Class: ', y_mistake_perClass)

    final_confusion_matrix = np.mean(confusion_matrices,axis = 0)
    text = 'Ridge Model'
    textFile = 'Ridge_cc'

    if svm_option:
        textFile = 'SVM_cc'
        text = 'SVM Model'
    elif rf_option:
        textFile = 'RF_cc'
        text = 'Random Forest Model'
    elif lr_option:
        textFile = 'LR_cc'
        text = 'Logistic Regression Model'
    elif ensemble_option:
        textFile = 'ENSEMBLE_cc'
        if maxMode:
            text = 'Ensemble Model - Max'
        else:
            text = 'Ensemble Model - Mixture'

    text += ' Confusion Matrix'
    textFile += '.png'
    fig = plt.figure()


    plot_confusion_matrix(final_confusion_matrix,dataNames, title= text)
    fig.savefig(textFile)
elif testMode:
    x_train = X_train_all
    y_train = Y_train_all
y_train = Y_train_all
print('TEST␣FEATURIZATION')
test_names, test_sequences, test_tags, test_lengthArray, test_setOfAminoAcids  = loadFastaFiles(['blind'])
x_test = featurize(test_sequences,setOfAminoAcids,test_lengthArray, isForSVM= svm_option)
print('FITTING')
classifier.fit(x_train,y_train)
print('PREDICTION')
try:
        y_pred = classifier.predict_proba(x_test)
except:
        scores = classifier.decision_function(x_test)
        y_pred = scores/(1+scores)

for i in range(y_pred.shape[0]):
        print(test_names[i] + '␣' + dataNames[y_pred[i].argmax()] + '␣Confidence␣' +  str(int(y_pred[i].max() * 100)) + '%
             ')
```

6.2 Tf-IDF Code

```
import numpy as np
from sklearn.feature_extraction.text import TfidfTransformer, TfidfVectorizer
import sys
import re
import os
from Bio import SeqIO
import kmr
######### PATH #################
scriptPath = os.path.realpath(__file__)
rootPath = os.path.sep.join(scriptPath.split(os.path.sep)[:−1])
dataPath = rootPath + os.path.sep + 'data' +  os.path.sep
###################### EXTRACTING ARGUMENTS ########################
args = sys.argv[1:] ## getting the arguments

N_gram = 3
if ('–ngram' in args):
        N_gram = int(args[args.index('–ngram') + 1])

sequenceType = 'all'
if ('–type' in args):
        sequenceType = args[args.index('–type')+1]

numberToDisplay = 10
if ('–display' in args):
        numberToDisplay = int(args[args.index('–display') + 1])

length = 0
if ('–length' in args):
        length = int(args[args.index('–length') + 1])

selection = False
if('–select' in args):
        selection = True
        coef = 10
        if('–coef' in args):
                coef = float(args[args.index('–coef') + 1])
###################### Loading the Data ###############################


def loadFastaFiles(fileList):
        index = 0

        ## Initializing array
        tags = []
        names = []
        sequences = []
        lengthArray = []
        setOfAminoAcids = []
        ## For each file
        for name in fileList:
                print('LOADING␣ELEMENTS␣FROM␣FILE␣:␣' + name)
                secondIndex = 0
                ## LOAD file
                fasta_sequences =  SeqIO.parse(open(dataPath + name + '.fasta'),'fasta') # The function I miss
                for fasta in fasta_sequences:
                        seqName, sequence = fasta.id, str(fasta.seq)

                        sequence = re.sub('X', ["A", "R", "N", "D", "C", "Q", "E", "G", "H", "I", "L", "K", "M", "F", "P", "S", "T
                              ", "W", "Y", "V" ][np.random.randint(0,20)], sequence)
                        sequence = re.sub('U', '', sequence)
                        sequence = re.sub('B', ['N', 'D'][np.random.randint(0,2)], sequence)
```

```
            setOfAminoAcids = list(set(setOfAminoAcids).union(set(sequence)))

            secondIndex += 1

            ## Adding sequqnce name to array
            names.append(seqName)
            ## addind sequence to sequence array
            sequences.append(sequence)
            ## Adding the tag
            tagArray = np.zeros(len(fileList))
            tagArray[index] = 1
            tags.append(tagArray)

            ## append the sequence length the length array
            lengthArray.append(len(sequence))
            ## append to the list of different amino-acids
        print('Number␣of␣Samples␣for␣'+ name + '␣:␣' ,secondIndex)
        index += 1

    return names, sequences, tags, lengthArray, setOfAminoAcids

def computeTFIDF(sequenceList,N,numberToDisplay,length=0):
    print('building␣the␣N-GRAM␣sequence')
    if length == 0:
        corpus = ["␣".join(["".join(sequence[i:i+N]) for i in range(len(sequence)-N)]) for sequence in sequenceList]
    else:
        if length >0:
            corpus = ["␣".join(["".join(sequence[:length][i:i+N]) for i in range(len(sequence)-N)]) for sequence in
                sequenceList]
        else:
            corpus = ["␣".join(["".join(sequence[length:][i:i+N]) for i in range(len(sequence)-N)]) for sequence in
                sequenceList]
    print('COMPUTING␣TF-IDF')
    vectorizer = TfidfVectorizer(min_df=1)
    X = vectorizer.fit_transform(corpus)
    idf = vectorizer.idf_
    sortedIndex = np.array(np.argsort(-idf))
    sequence_names = np.array(vectorizer.get_feature_names())[sortedIndex]
    idf = np.array(idf)[sortedIndex]
    #print(dict(zip(sequence_names[:numberToDisplay], idf[:numberToDisplay])))
    toReturn = [word.upper() for word in sequence_names[:numberToDisplay]]
    return toReturn

def select(listOfClasses,listOfSequences,tags, allSequenceList, coef=10):
    ClassDict = dict()
    fullList = []
    for i in range(len(listOfClasses)):
        print('CLASS␣:␣', listOfClasses[i])
        ClassDict[listOfClasses[i]] = []
        tempList = []
        candidateSequences = listOfSequences[i]
        targetedSequences = allSequenceList[tags.argmax(axis = 1) == i]
        ReamainingSequences = allSequenceList[tags.argmax(axis = 1) != i]
        for seq in candidateSequences:
            alphaSum = np.sum([int(seq in target) for target in targetedSequences])
            alphaValue = alphaSum/targetedSequences.shape[0]
            betaSum = np.sum([int(seq in target) for target in ReamainingSequences])
            betaValue = betaSum/ReamainingSequences.shape[0]
            Bool = alphaValue>=betaValue*coef
            if Bool:
                ClassDict[listOfClasses[i]].append(seq)
```

```
                                fullList.append(seq)

        return ClassDict, fullList
print('LOADING␣THE␣DATA␣SET')

dataNames = ['cyto', 'mito', 'nucleus', 'secreted']
names, sequences, tags, lengthArray, setOfAminoAcids  = loadFastaFiles(dataNames)


if selection :
        offeredSequences = []
        listOfSequences = []
        nameList = []
        for i in range(len(dataNames)):
                dataIndex = i
                name = dataNames[i]
                print('COMPUTING␣TF_IDF␣FOR␣:␣', name)
                sequenceList = np.array(sequences)[np.array(tags).argmax(axis = 1) == dataIndex]
                offeredSequences.append(computeTFIDF(sequenceList,N_gram,numberToDisplay,length))
                listOfSequences.append(sequenceList)
                nameList.append(name)
        selectedPerClass, FullSelectedList = select(nameList,offeredSequences,np.array(tags), np.array(sequences), coef )
        print(selectedPerClass)
        print(FullSelectedList)
elif sequenceType == 'all':
        computeTFIDF(sequences,N_gram,numberToDisplay,length)
else:
        print('COMPUTING␣TF-IDF␣for␣:␣', sequenceType)
        dataIndex = dataNames.index(sequenceType)
        sequenceList = np.array(sequences)[np.array(tags).argmax(axis = 1) == dataIndex]
        computeTFIDF(sequenceList,N_gram,numberToDisplay, length)
```