Министерство науки и высшего образования Российской Федерации

Федеральное государственное автономное образовательное учреждение высшего образования
«КАЗАНСКИЙ (ПРИВОЛЖСКИЙ) ФЕДЕРАЛЬНЫЙ УНИВЕРСИТЕТ»

Институт Математики и Механики имени Н.И.Лобачевского
Кафедра математического анализа

Направление: 01.04.01 – Математика
Профиль: Анализ на многообразиях

МАГИСТЕРСКАЯ ДИССЕРТАЦИЯ

Кластеризация биоинформатических данных

Работа завершена:
Студент 2 курса
группы 05-712
"____"_____ 20____ г. _____ Ахметов И.З

Работа допущена к защите:
Научный руководитель
Кандидат физ.-мат. наук, ассистент
"____"_____ 20____ г. _____ Новиков А.А

Заведующий кафедрой
Доктор физ.-мат. наук, профессор
"____"_____ 20____ г. _____ Насыров С.Р

КАЗАНЬ – 2019

Содержание

1 Введение

Биоинформатика - молодая, бурноразвивающаяся наука. Мощный толчок к развитию она получила благодаря применению математических методов в биологии, до этого применявшихся в основном в физике. Целью данной работы является решение задачи биологии современными средствами математики и программирования.

В работе будет произведена кластеризация генов на k компонент методом средних. До этого предпринимались аналогичные попытки малазийскими учеными. Их труды были представлены в университетском журнале [7] , а также на научной конференции в сингапуре [6]. Особенностью данной работы будут использование языка программирования высокого уровня Python с применением технологии параллельного программирования, основанного на создании множества процессов, а также координатное представление нуклеотида, в котором нуклеотиду соответствует пятимерный вектор из вещественных чисел, принадлежащих интервалу $[0, 1]$.

2 Основные понятия и определения

Нуклеотид - это органическое соединение, элементарная составляющая нуклеиновой кислоты, по сути строительный кирпич ДНК. В ДНК каждый нуклеотид кодируется одним из четырех символов: A - Аденин, C - Цитозин, G - Гуанин, T - Тимин.

Кластеризация - это разбиение множества каких-либо объектов на непересекающиеся подмножества так, чтобы элементы, принадлежащие какому-либо подмножеству имели некоторый схожий признак, существенно отличающий их от элементов из другого подмножества.[1]

ДНК- это упорядоченная последовательность нуклеотидов, кодирующая генетический код какого-либо организма.

Алгоритм BLAST - алгоритм, предназначенный для поиска сходных участков генов для ДНК различных организмов, а так же приведения длин нескольких ДНК к единой длине с целью возможности применения метрики Хемминга для нахождения генетического расстояния.[12]

Генетическое расстояние - мера генетической близости $\rho > 0$ между двумя различными биологическими видами, введенная с целью изучения вопроса, насколько близкородственными являются различные виды.[4]

Python - построчно интерпретируемый язык программирования высокого уровня. В данной работе он будет применен для написания программы кластеризации. В данной работе будет использована версия компилятора - Python 3.7.3, IDE - JetBrains PyCharm Community Edition 2019.1.1

FASTA - формат файла, содержащий генетический код в следующем виде:

>(NameOfDNASequence1)

(DNASequenceCode1)

>(NameOfDNASequence2)

(DNASequenceCode2)

.....

>(NameOfDNASequenceN)

(DNASequenceCodeN)

3 Кластеризация данных

3.1 Постановка задачи кластеризации биоинформатических данных

Имеется N генетических последовательностей, сиквенсов (от английского sequence), задача - разбить их на K - кластеров, непересекающихся подмножеств таким образом, чтобы сумма расстояний от каждой генетической последовательности до соответствующего ей кластерного центроида (среднего арифметического всех последовательностей, принадлежащих данному кластеру) была минимальной.

Задача осложняется тем, что не существует определения нормы генетической последовательности. Длина строки в качестве нормы не подходит.

Однако, как станет ясно далее, при кластеризации методом K-means можно обойтись без использования нормы объекта в пространстве генетических последовательностей, достаточно определить их сумму и деление на число.

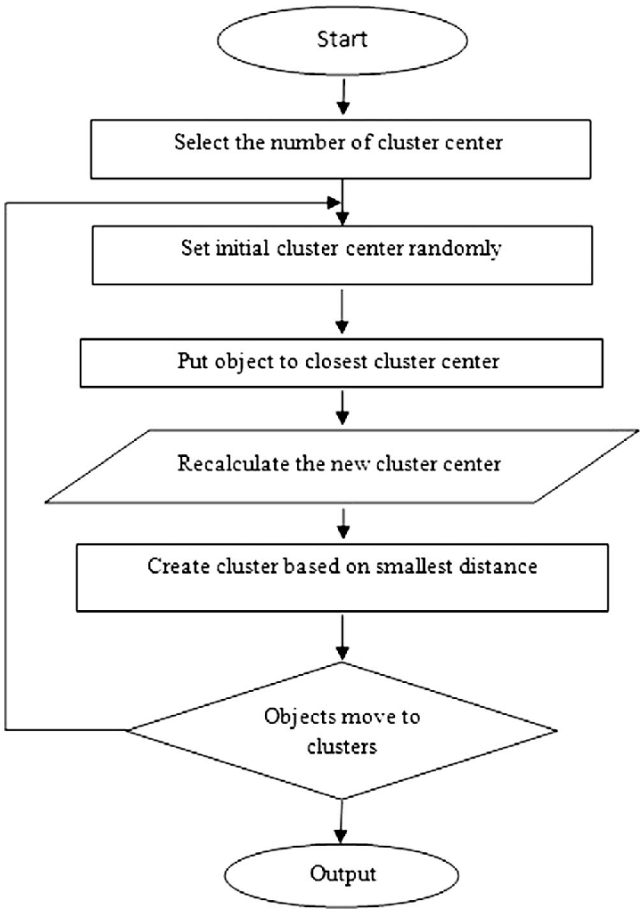
3.2 Алгоритм кластеризации K-means

Допустим, имеется N различных объектов, которые следует разбить на K кластеров, т.е непересекающихся множеств, объекты которых из всех кластеров наиболее близки по расстоянию к соответствующему им.

На первом шаге случайным образом выбирается в качестве начальных центров кластеров - центроидов K из N объектов. Далее измеряется расстояние для каждого элемента множества объектов до текущих центроидов и ставится каждому элементу в соответствие наиболее близкий к нему центроид. Пересчитываются центроиды так же, как находим центр

масс - сумма всех элементов, принадлежащих на текущей итерации к X кластеру делится на количество элементов, принадлежащих на данной итерации X кластеру - это и будет значение нового центроида. Выполняется эта итерация повторно до тех пор, пока значения центроидов не перестанут изменяться. Когда центроиды перестанут изменять свое значение, считается сумму расстояний для всех объектов до соответствующих им центроидов.

Этот процесс запускается 20 раз и выбирается такое разбиение, что сумма расстояний минимальна. Схематическое изображение алгоритма:



4 Метрики на генах

4.1 Метрика Хемминга

Расстояние по хеммингу - количество различных символов в одинаковых индексах для двух строк одинаковой длины.[8] Оно определяется по формуле:

$$D = \sum_{k=1}^{N} |A_k - Bk| \tag{1}$$

где A и B - две строки длины N. Если соответствующие символы равны, то расстояние равно нулю, в противном случае - единице.

В данной работе мы переопределим расстояние Хемминга с целью удобства реализации и вычислений следующим образом. Каждому символу в строке поставим в соответствие пятимерный вектор:

$$x = (a_1, a_2, a_3, a_4, a_5) \tag{2}$$

каждая координата которого может принимать значения $a \in [0, 1]$ и соответствует символам $(A, C, G, T, -)$ соответственно в порядке перечисления. A означает Аденин, C - Цитозин, Г - Гуанин, T - Тимин. Символ $-$ соответствует пропуску, получается после процедуры выравнивания генетических последовательностей алгоритмом BLAST. Тогда псевдорасстояние между двумя произвольными нуклеотидами x и y будет определятся по формуле:

$$DC(x, y) = 1 - \sum_{k=1}^{5} x_k \times y_k \tag{3}$$

а расстояние между двумя произвольными последовательностями нуклеотидов X и Y по формуле:

$$DS = \sum_{k=1}^{N} DC(X_k, Y_k) \tag{4}$$

В данной работе мы будем применять именно это расстояние.

4.2 Метрика Левенштейна

Расстояние Левенштейна - это количество операций замены одного символа другим, вставки одного символа, удаления символа, минимально необходимых для превращения одной строки в другую.[9]

Расстояние Левенштейна широко применяется, например, в текстовых процессорах для нахождения грамматических ошибок, предложения наиболее подходящего слова, если слово, введенное пользователем недописано, в биоинформатике для нахождения минимального расстояния между двумя генетическими последовательностями.

Далее приводится реализация на языке программирования Python оптимизированного итерационного алгоритма Левенштейна, на каждом шаге потребляющий $2 \times M$ единиц памяти,
где $M = min(len(string1), len(string2))$. Алгоритм не использует рекурсию, так что нет опасности переполнения стека, и вместо $N \times M$ памяти, где N и M - длины строк, использует память, необходимую для хранения всего лишь двух строк.

```
def LevensteinDistance(string1,string2):
    if len(string1) > len(string2):
        string1,string2 = string2,string1
    distances = range(len(string1) + 1)
    for index2,char2 in enumerate(string2):
        newDistances = [index2+1]
        for index1,char1 in enumerate(string1):
            if char1 == char2:
                newDistances.append(distances[index1])
            else:
                newDistances.append(1 + min((distances[index1],
                                             distances[index1+1],
                                             newDistances[-1])))
```

```
            distances = newDistances
        return distances[-1]
```

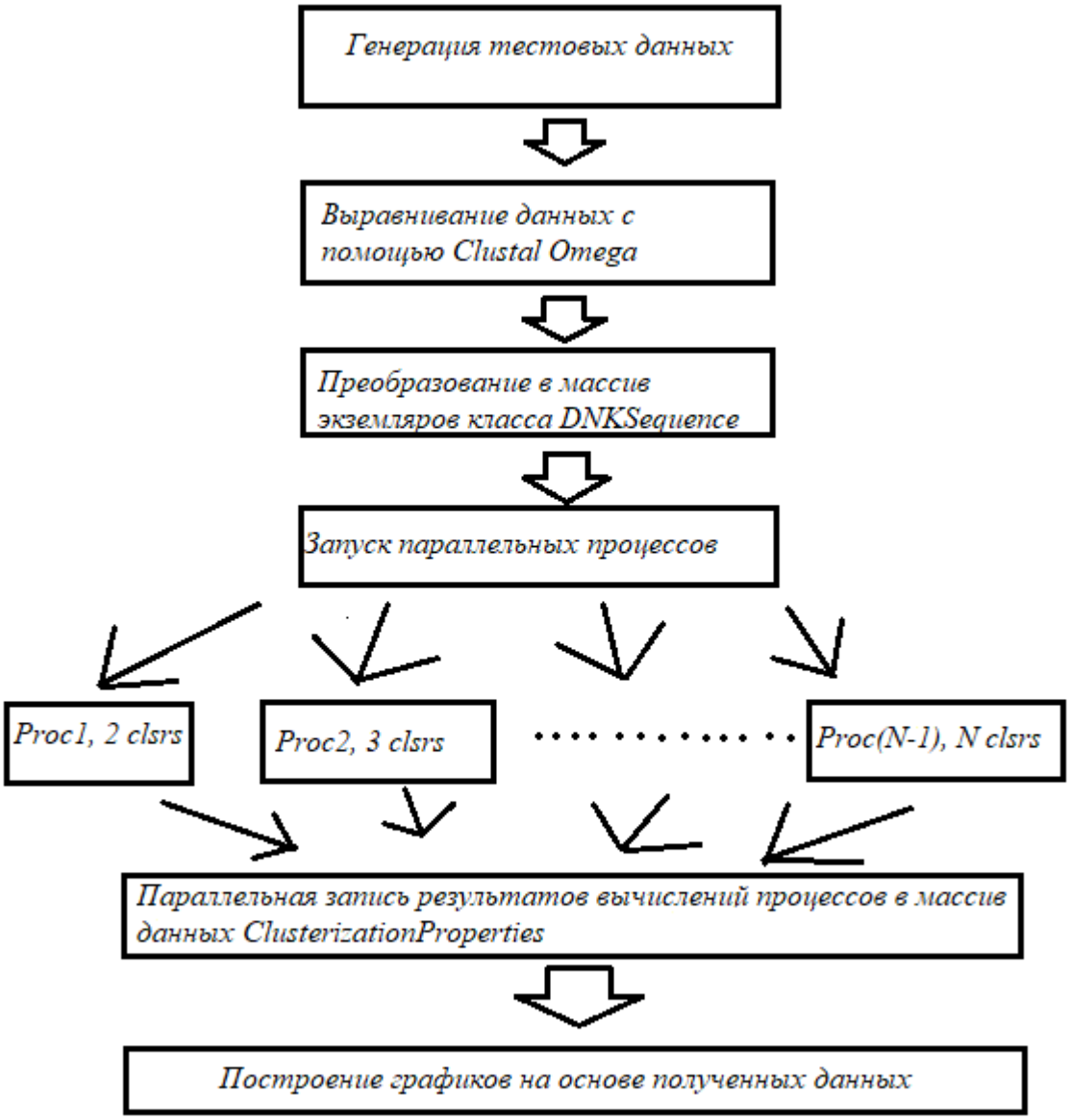
5 Код программы и документация

5.1 Общее описание

Программа для расчетов для удобочитаемости и возможности поддержки кода разбита на 6 скриптов: PlotFunction.py, GeneratingTest - Data.py, ReadSequencesFromFile.py, NucleotidClass.py, DNKSeqClass.py, Clusterization.py.

Разбивка производилась так, чтобы каждый скрипт отвечал за свою отдельную часть работы программы. Так же в папке с проектом, помимо скриптов находится программа Clustal-omega-1.2.2-win64 и текстовые файлы TestIn.txt и TestOut.txt в формате FASTA, генерируемые скриптами. Как видно из названия, они являются тестовыми данными, для проверки корректности работы программы. Данные файлы при надобности могут быть заменены реальными данными. В данном случае необходимо в скрипте ReadSequencesFromFile.py заменить в строке p = call('clustal-omega-1.2.2-win64\clustalo.exe -i TestIn.txt -o TestOut.txt -v –force') TestIn.txt и TestOut.txt на соответствующие имена файлов и "отключить"скрипт GeneratingTestData.py, закомментировав строку GeneratingTestData.Generate(NumberOfSequences,NuclsInSequence) в скрипте PlotFunction.py.

Далее приведена краткая блок-схема, идея работы программы:



Теперь рассмотрим листинг кода всех скриптов по отдельности и детально опишем работу каждого из них.

5.2 Clustal Omega

Была использована сторонняя программа clustal-omega-1.2.2-win64 для выравнивания массива последовательностей с помощью алгоритма BLAST. Выравнивание необходимо для поиска сходных участков генетического когда сразу для всех входящих генетических последовательностей и приведения их всех к одинаковой длине, чтобы можно было воспользоваться метрикой Хемминга. Программа написана на C++ группой

разработчиков из Ирландии. Выравнивание выполняется параллельно, создается количество потоков, равное числу ядер компьютера, на котором запущена программа. Все это делает данную программу высокопроизводительной.

5.3 NucleotidClass.py

В данном скрипте описан класс VecNucleotid, описывающий свойства Нуклеотида. определение суммы нуклеотидов, деления на число.

В конструкторе def __init__(self,Nucl) нуклеотид, представленный одной из букв A,C,G,T конвертируется в пятимерный вектор (a_1, a_2, a_3, a_4, a_5), каждый элемент a которого $a \in [0, 1]$. До конструктора определяется словарь NuclDict = {"A " : 0, "C " : 1, "G " : 2, "T " : 3, "– " : 4}, ставящий в соответствие нуклеотиду его координаты. Этот словарь нужен для работы конструктора класса.

Далее в методе def __add__(self,other) определяется сумма нуклеотидов как поэлементная сумма соответствующих им векторов, в результате которой получается новый нуклеотид. В действительности эта сумма определяется как поэлементная сумма векторов, деленная на их количество, однако это реализовано лишь в скрипте Clasterization, а не здесь для удобства определения класса, но здесь определено деление нуклеотида на число в методе def __truediv__(self, number) как поэлементное деление его координат на данное число. Эти две операции и дают возможность определить в скрипте для кластеризации сумму N нуклеотидов как сумму соответствующих координат, деленную на N.

В методе def distance(self,N1) определяется псевдометрика между

двумя нуклеотидами по формуле:

$$DC(x,y) = 1 - \sum_{k=1}^{5} x_k \times y_k \tag{5}$$

В методе def __eq__(self,OtherNucl) определяется, равны ли нуклеотиды между собой путем поэлементного сравнения их координат.

В методе def IfSpace(self) определяется, соответствуют ли координаты нуклеотида символу "— " . Это нужно для определения изначальных длин последовательностей ДНК и сортировки их по длине последовательности, для более оптимального начального выбора центров кластеров.

Листинг скрипта:

```
    NuclDict = {"A": 0, "C": 1, "G": 2, "T": 3, "-": 4} #possible values of
        Nucleotid


class VecNucleotid: #That class describes elementarty Nucleotid and his
    properties
    def __init__(self,Nucl):
        self.Nucls = [0.0] * 5
        self.Nucls[NuclDict[Nucl]]=1.0
    def __truediv__(self, number): #determine operation of division to Nucl
        self.Nucls=[CoordinateIndex/number for CoordinateIndex in self.Nucls]
        return self
    def __add__(self,OtherNucleotid): #determine result of sum of two nucl
        self.Nucls=[self.Nucls[CoordinateIndex] +
            OtherNucleotid.Nucls[CoordinateIndex]
                    for CoordinateIndex in range(5)]
        return self
    def distance(self,OtherNucleotid): # determine distance between two
        nucleotides
        return 1-sum([OtherNucleotid.Nucls[CoordinateIndex] *
```

```
            self.Nucls[CoordinateIndex]
                    for CoordinateIndex in range(5)])
    def __str__(self):
        return ' '.join(str(e) for e in self.Nucls)
    def __eq__(self,OtherNucleotid): #check if two Nuclteotids are equal
        return (False not in
            [self.Nucls[CoordinateIndex]==OtherNucleotid.Nucls[CoordinateIndex]
                    for CoordinateIndex in range(5)])
    def IfSpace(self): # Check if this Nucleotid is equal to
        # -. It's needed to sort Sequences by initial length
        if self.Nucls[4] == 1:
            return 1
        else:
            return 0
```

5.4 DNKSeqClass.py

Данный скрипт описывает класс DNKSequence, описывающий свойства ДНК-последовательности. Он использует уже определенные методы и свойства класса VecNucleotid для применения их уже к ДНК - последовательностям, а не отдельным нуклеотидам.

Конструктор def __init__(self,StrSeq,DNKName="Default"): отвечает за генерацию массива нуклеотидов из массива символов A,C,G,T,.

Метод def __truediv__(self, number): отвечает за деление ДНК - последовательности на натуральное число. По сути он делит каждый элемент массива нуклеотидов на это число, используя уже определенный метод деления в классе VecNucleotid.

Метод def __add__(self,OtherSeq): отвечает за поэлементное сложение соответствующих нуклеотидов в ДНК-последовательностях. Он и метод def __truediv__(self, number): совместно определяют сумму N по-

следовательностей как сумму их соответствующих элементов, деленных на N поэлементно.

Метод def SeqDistance(self,OtherSeq): определяет расстояние между двумя ДНК - последовательностями X и Y длины N как сумму расстояний между их нуклеотидами с одинаковыми индексами по формуле:

$$DS = \sum_{k=1}^{N} DC(X_k, Y_k) \tag{6}$$

где DC - псевдорасстояние между двумя нуклеотидами.

Метод def __eq__(self,OtherSeq): определяет, равны ли две ДНК-последовательности путем поэлементного сравнения их элементов. Если хотя бы одна пара элементов различна, т.е нуклеотиды имеют разные пятимерные координаты, то и последовательности различны.

Метод def __len__(self): определяет начальную длину невыравненной последовательности, т.е до применения к ней и другим ДНК-сиквенсам программы Clustal Omega и вставки в нее элемента −.

Листинг скрипта:

```
from NucleotidClass import VecNucleotid

class DNKSequence:
    def __init__(self,StrSeq,ThisName): # overloading constructor for DNK
        sequence
        self.seq=[VecNucleotid(StrSeq[Nucleotid]) for Nucleotid in
            range(len(StrSeq))]
        self.name=ThisName
    def __truediv__(self, DivNumber): # overloading divide operation for DNK
        sequences
        self.seq=[Nucleotid/DivNumber for Nucleotid in self.seq]
        return self
    def __add__(self,OtherSeq): #overloading add operation for DNK sequences
        self.seq=[self.seq[Nucleotid]+OtherSeq.seq[Nucleotid]
```

```
                for Nucleotid in range(len(self.seq))]
        return self
    def SeqDistance(self,OtherSeq): #Distance between two different DNK
        sequences
        return sum([self.seq[Nucleotid].distance(OtherSeq.seq[Nucleotid])
                for Nucleotid in range(len(self.seq))])
    def __eq__(self,OtherSeq): #check if two DNKSeqs are equal
        return ( False not in [self.seq[Nucleotid]==OtherSeq.seq[Nucleotid]
                for Nucleotid in range(len(OtherSeq.seq))] )
    def __len__(self): # Count number of "-" in NucleotidSequence to get its
        length
        return sum([Nucleotid.IfSpace() for Nucleotid in self.seq])
```

5.5 GeneratingTestData.py

Данный скрипт, как видно из названия, генерирует тестовые данные для проверки корректности работы программы. В начале для генерации случайных числе из пакета рандом импортируется метод randint: from random import randint. В цикле эта функция генерирует числа от 0 до 3 включительно, и в зависимости от числа мы получаем букву благодаря словарю: NuclDict={0:"A" 1:"C" 2:"G" 3:"T" }. Так же случайным образом генерируется число NucleSequences - количество нуклеотидов в текущей последовательности. Таких последовательностей генерируется length. Затем все сгенерированные строки, являющиеся последовательностями символов A,C,G,T заносятся в файл в формате FASTA. Листинг скритпа:

```
from random import randint
NuclDict={0:"A",1:"C",2:"G",3:"T",4:"-"}

def Generate(NumberOfSequences=30,NucleotidsInSequence=2000): # Generating Test
```

```
    Data
    with open("TestData.txt",mode="w") as f:
        for i in range(NumberOfSequences):
            teststr = [NuclDict[randint(0,4)]
                       for i in range(NucleotidsInSequence)] #Generating Test
                           DNKSeq
            f.write('>Sequence '+str(i)+'\n')
            f.write(''.join(teststr)+"\n")
```

5.6 ReadSequencesFromFile.py

Из пакета subprocess импортируется метод call для возможности запуска сторонних .exe файлов из Python. С помощью этого метода мы запускаем программу Clustal Omega с нужными нам параметрами: p = call('clustal-omega-1.2.2-win64\clustalo.exe -i TestIn.txt -o TestOut.txt -v –force') для выравнивания сгенерированных последовательностей в файле TestIn.txt для последующего выравнивания и записи результата в файл TestOut.txt. Создается пустой массив DNKSequences=[ ] из ДНК - последовательностей, элементы которого - экземпляры класса DNKSequence. Далее открывается файл TestOut.txt и в цикле считываются ДНК-последовательности по одной, из которых сразу же генерируются экземпляры класса DNKSequence и заносятся в массив DNKSequences. Когда все последовательности считаны, получившийся массив сортируется и возвращается функции, вызвавшей ReadData(), т.е переменной DNKSequences в скрипте PlotFunction.py.

Листинг скрипта:

```
from DNKSeqClass import DNKSequence
```

```
def ReadData():
    DNKSequences = [] # Array of DNKSequences
    with open("TestData.txt", "r") as filehandler:
        # Reading files and creating DNKSequences as Classes
        ThisName = ">Pseudo" # this we need to avoid program error
        ThisSeq = "AAT"
        line = filehandler.readline()
        while line:
            if line[0] == ">":
                DNKSequences.append(DNKSequence(ThisSeq, ThisName))
                ThisName = line;
                ThisSeq = ""
            else:
                ThisSeq = ThisSeq + line.strip('\n')
            line = filehandler.readline()
    # add the last seq because it hasn't been read within loop
    DNKSequences.append((DNKSequence(ThisSeq, ThisName)))
    DNKSequences.remove(DNKSequences[0]) # removing pseudo sequence
    # Sorting by lenght of seq without alignment incremently
    DNKSequences.sort(key=len, reverse=True)
    return DNKSequences
```

5.7 Clusterization.py

Импортирование библиотек. Для возможности суммирования произвольных объектов в функции reduce был [2] произведен import operator, дающий возможность использовать метод operator.add. Для генерации произвольных целых неотрицательных чисел и, тем самым, случайного выбора начальных центроидов был сделан импорт randint из библиотеки random. Из библиотеки functools импортирован метод reduce, который в паре с методом operator.add позволяет быстро суммировать объекты произвольной природы. Из библиотеки statistics был произведен

импорт stddev для нахождения среднеквадратичного отклонения диаметра(наибольшего расстояния между элементами кластера) от своего среднего значения. Из библиотеки itertoos был сделан импорт функции combinations для выбора всех возможных двух элементов в кластере. Это необходимо для нахождения попарного расстояния между элементами кластера и выбора наибольшего из них.

Алгоритм кластеризации: На вход функции GetClusterizationProperties поступает массив ДНК-последовательностей - DNKSequences, количество кластеров - NumberOfClusters, количество ДНК - последовательностей - NumberOfSequences, которое равно len(DNKSequences), т.е длине массива сиквенсов. Далее производится генерация начальных центроидов случайным образом, после чего по алгоритму K-means идет разбивка на кластеры. Из получившейся разбивки мы берем диаметр кластера - максимальное расстояние между элементами в кластере среди всех кластеров и радиус кластера - максимальное расстояние между центром кластера и элементом данного кластера среди всех кластеров. Эти данные записываем в массивы Diameters, Radii соответственно. Эту процедура повторяется 20 раз, т.е в массивах Diameters и Radii находится по 20 элементов. Для этих массивов находится средний диаметр, радиус и возвращается в вызывающую функцию в скрипте PlotFunction.py.

Листинг скрипта:

```
from DNKSeqClass import DNKSequence
import operator
from random import randint
from functools import reduce
from statistics import stdev
```

```
from itertools import combinations

def ArePrevAndCurCentersDiff(PrevC,CurC,NumberOfClusters):
    # checking if Centres on current and prev steps are not equal
    return (False in [PrevC[i]==CurC[i] for i in range(NumberOfClusters)])

def GetNearestCentr(Centers,ThisDNKSeq,NumberOfClusters): # Get index of
    nearest Cluster to DNKSequence
    return min([(ThisDNKSeq.SeqDistance(Centers[i]),i) for i in
        range(NumberOfClusters)])[1]

def GetClusterizationProperties(DNKSequences,ClusterizationProperties,
    NumberOfClusters=2,
                              NumberOfSequences=10, NuclSequences=1000):

    #Initilal Centers of Clusters
    PrevCenters=[DNKSequence("A"*(NuclSequences+1),"AAA") for x in
        range(NumberOfClusters)]
    ClusterIndexes=[0]*NumberOfSequences # ClusterNumber correspond to
        DNKSequences with the same index
    Iterations=20
    Diameters=[]
    RangeOfInitSubCluster=NumberOfSequences//NumberOfClusters
    Radii=[]
    while (Iterations>0):
        Iterations-=1
        #Initial random centers of Clusters
        CurrCenters=[DNKSequences[randint(i*RangeOfInitSubCluster,(i+1) *
            RangeOfInitSubCluster)]
                    for i in range(NumberOfClusters-1)]
        CurrCenters.append(DNKSequences[randint((NumberOfClusters-1) *
            RangeOfInitSubCluster,NumberOfSequences-1)])

        while
            (ArePrevAndCurCentersDiff(PrevCenters,CurrCenters,NumberOfClusters)):
```

```
        ##until centers stop to change
        for DNKSeqIndex in range(NumberOfSequences): #assigning
            VecNucleotids to Clusters
            ind=GetNearestCentr(CurrCenters, DNKSequences[DNKSeqIndex],
                NumberOfClusters)
            ClusterIndexes[DNKSeqIndex]=ind
        PrevCenters=[DNKSeqIndex for DNKSeqIndex in CurrCenters]
        for ClusterNumber in range(NumberOfClusters): #Recomputing Centers
            of clusters
            BelongToThisCluster=[ DNKSequences[i]
                    for i in range(NumberOfSequences) if ClusterIndexes[i] ==
                        ClusterNumber ]
            SequencesInCluster=len(BelongToThisCluster)
            if SequencesInCluster!=0:
                CurrCenters[ClusterNumber]=reduce(operator.add,
                    BelongToThisCluster)/SequencesInCluster


    MaxDiameters=[]
    MaxRadius=[]
    for ClusterNumber in range(NumberOfClusters): # Finding max diameter and
        max radius
        BelongToThisCluster=[ DNKSequences[i]
                            for i in range(NumberOfSequences) if
                                ClusterIndexes[i] == ClusterNumber ]
        if len(BelongToThisCluster)>1:
            MaxDiameters.append(max([ j[0].SeqDistance(j[1])
                            for j in list(combinations(BelongToThisCluster,
                                2)) ]))
            MaxRadius.append(max([ j.SeqDistance(CurrCenters[ClusterNumber])
                            for j in BelongToThisCluster ]))
    Diameters.append(max(MaxDiameters))
    Radii.append(max(MaxRadius))
AverageD=reduce(operator.add,Diameters)/20
AverageR=reduce(operator.add,Radii)/20
ClusterizationProperties[NumberOfClusters-2] = [AverageD, AverageR,
```

```
stdev(Diameters), stdev(Radii)]
```

5.8 PlotFunction.py

Импортируются практически все скрипты, упомянутые раннее. Для измерения времени выполнения, т.е скорости работы, производительности программы импортируется функция time из библиотеки time. Для распараллеливания вычислений, т.е чтобы функции нахождения диаметров и радиусов с одним и тем же массивом данных, но разным числом кластеров выполнялись параллельно, импортируются классы Process, Manager и метод $cpu_count()$ из библиотеки multiprocessing[1]. Это кроссплатформенная библиотека позволяет распараллеливать вычисления в среде ОС Windows, по сути запуская сразу несколько интерпретаторов Python, каждый из которых запускает переданную ему в списке параметров функцию с указанными аргументами. Во всех 6 скриптах библиотеки не были импортированы полностью, а только лишь те функции и классы из них, необходимые для работы программы. Это сделано с целью повышения производительности. Для построения графиков импортируется сторонняя библиотека matplotlib.pyplot.

Вначале генерируются тестовые данные командой GeneratingTestData.Generate(NumberOfSequences,NuclsInSequence). Далее эти данные считываются и выравниваются командой DNKSequences = ReadSequencesFromFile.ReadData(). Затем в переменную ClusterizationProperties параллельно записываются данные, описывающие зависимость между числом кластеров и диаметром, радиусом кластеров. После чего по получившимся данным строятся графики командой

plt.plot b и выводятся на экран командой plt.show()

Ка известно, процесс обладает собственным пространством логической памяти, отдельной от других процессов. Здесь кроется отличие многопоточности от множества процессов - в многопоточности потоки работают в одном и том же адресном пространстве, с одними и теми же переменными, массивами, в общем данными. Процесс же при запуске копирует данные, переданные ему в качестве аргументов и работает уже с ними независимо от других процессов. Это может привести к слишком большому потреблению памяти. Чтобы этого избежать, в классе DNKSequence свойство класса NumberOfCluster вынесено в отдельную от класса переменную. В функции GetClusterizationProperties в скрипте Clusterization номера кластеров, к которым принадлежит данная ДНК-последовательность представлены в массиве ClusterIndexes. В скрипте PlotFunction ДНК-массив DNKSequences с помощью класса Manager и его метода list сделан общим для всех процессов, т.е Shared Memory. Это позволяет сэкономить $N \times Arr$ памяти, где N - количество запу?енных процессов, Arr - вес массива ДНК последовательностей в оперативной памяти. Далее процессы запускаются не все сразу, а порциями, не большими чем количество ядер компьютера. Это делается во избежание перегрузки компьютера, т.к в случае запуска множества процессов компьютер будет тратить время на переключение между процессами, а так же каждый процесс сам по себе потребляет при запуске ресурсы ПК. Представьте сразу 50 интепретаторов Python, одновременно запущенных на ПК, каждый из которых выполняет какие-либо операции. Довольно сильная нагрузка.

Листинг скрипта:

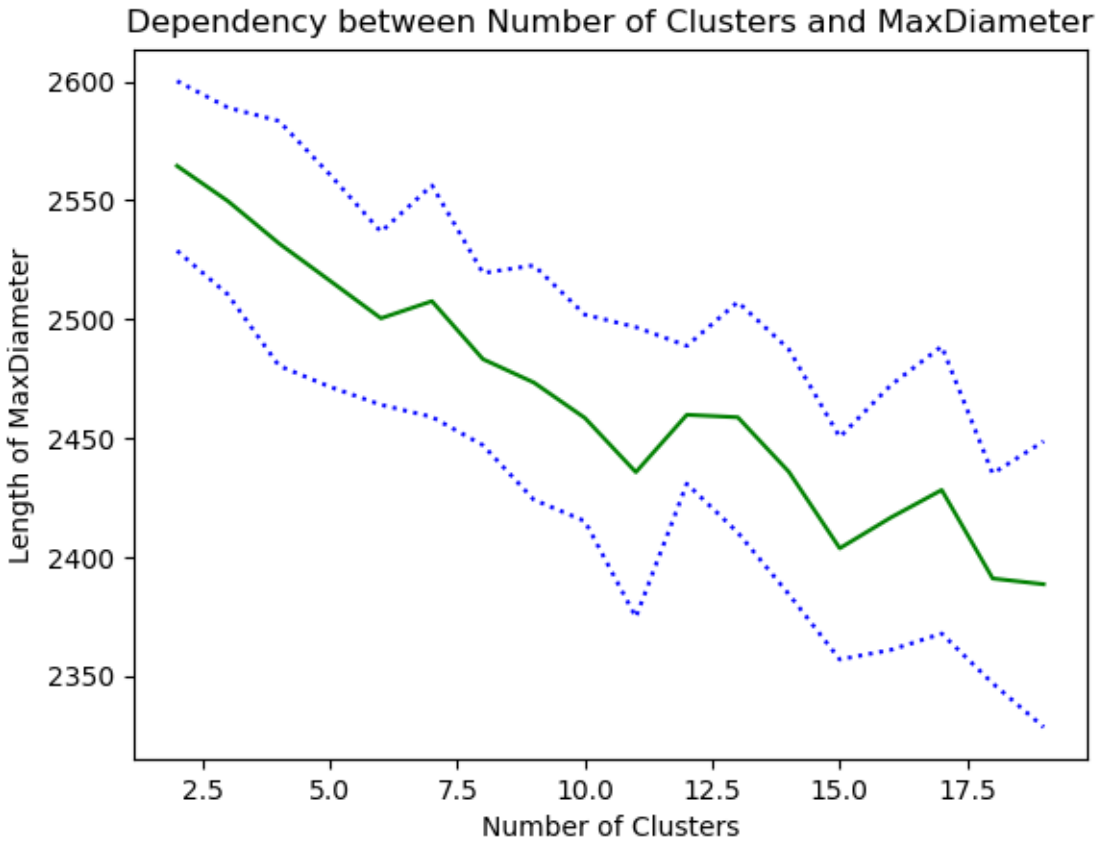
```
import Clusterization
import matplotlib.pyplot as plt
import GeneratingTestData
from multiprocessing import Process, Manager, cpu_count
import ReadSequencesFromFile
from time import time

NumberOfClusters=6
NumberOfSequences=12
NucleotidsInSequence=1000

if __name__ == '__main__':
    print("Program has been started")
    GeneratingTestData.Generate(NumberOfSequences,NucleotidsInSequence)
    print("Test Data is Generated")
    DNKSequences = Manager().list(ReadSequencesFromFile.ReadData()) # Read Data
        from file
    print("DNA-sequence array has been generated")
    ConcurProcs = cpu_count()
    ClusterizationProperties=Manager().dict()
    start = time()
    Processes=[Process(target=Clusterization.GetClusterizationProperties,args=
    (DNKSequences,ClusterizationProperties,
    ClustersNumber,NumberOfSequences,NucleotidsInSequence,))
              for ClustersNumber in range(2,NumberOfClusters+1)]

    ThisThreads = 0 # we need this to avoid overloading cpu and memory
    # so we will start processes partially
    print("Processes are starting up")
    while ((ThisThreads + ConcurProcs) <= NumberOfClusters - 1):
        for i in range(ThisThreads, ThisThreads + ConcurProcs):
            Processes[i].start()
        for i in range(ThisThreads, ThisThreads + ConcurProcs):
            Processes[i].join()
        ThisThreads += ConcurProcs
```

```
for i in range(ThisThreads, NumberOfClusters - 1):
    Processes[i].start()
for i in range(ThisThreads, NumberOfClusters - 1):
    Processes[i].join()
print("Processes have done their work")
print("Performance time " + str(time() - start) + " seconds")

Diam = [ClusterizationProperties[i][0] for i in ClusterizationProperties]
Rad = [ClusterizationProperties[i][1] for i in ClusterizationProperties]
StdErrD=[ClusterizationProperties[i][2] for i in ClusterizationProperties]
StdErrR = [ClusterizationProperties[i][3] for i in ClusterizationProperties]
Clusters=[i for i in range(2,NumberOfClusters+1)]

StdErrDP = [Diam[i] + StdErrD[i] for i in range(len(StdErrD))]
StdErrDM = [Diam[i] - StdErrD[i] for i in range(len(StdErrD))]
plt.plot(Clusters,Diam,'g-',label='Diameter')
plt.plot(Clusters,StdErrDM,'b:',label='StdErrD-')
plt.plot(Clusters, StdErrDP,'b:',label='StdErrD+')
plt.title("Dependency between Number of Clusters and MaxDiameter")
plt.xlabel("Number of Clusters")
plt.ylabel("Length of MaxDiameter")
plt.show()

StdErrRP = [Rad[i] + StdErrR[i] for i in range(len(StdErrR))]
StdErrRM = [Rad[i] - StdErrR[i] for i in range(len(StdErrR))]
plt.plot(Clusters, Rad,'g-',label='Radius')
plt.plot(Clusters, StdErrRM, 'b:', label='StdErrR-')
plt.plot(Clusters, StdErrRP, 'b:', label='StdErrR+')
plt.title("Dependency between Number of Clusters and MaxRadius")
plt.xlabel("Number of Clusters")
plt.ylabel("Length of MaxRadius")
plt.show()
```

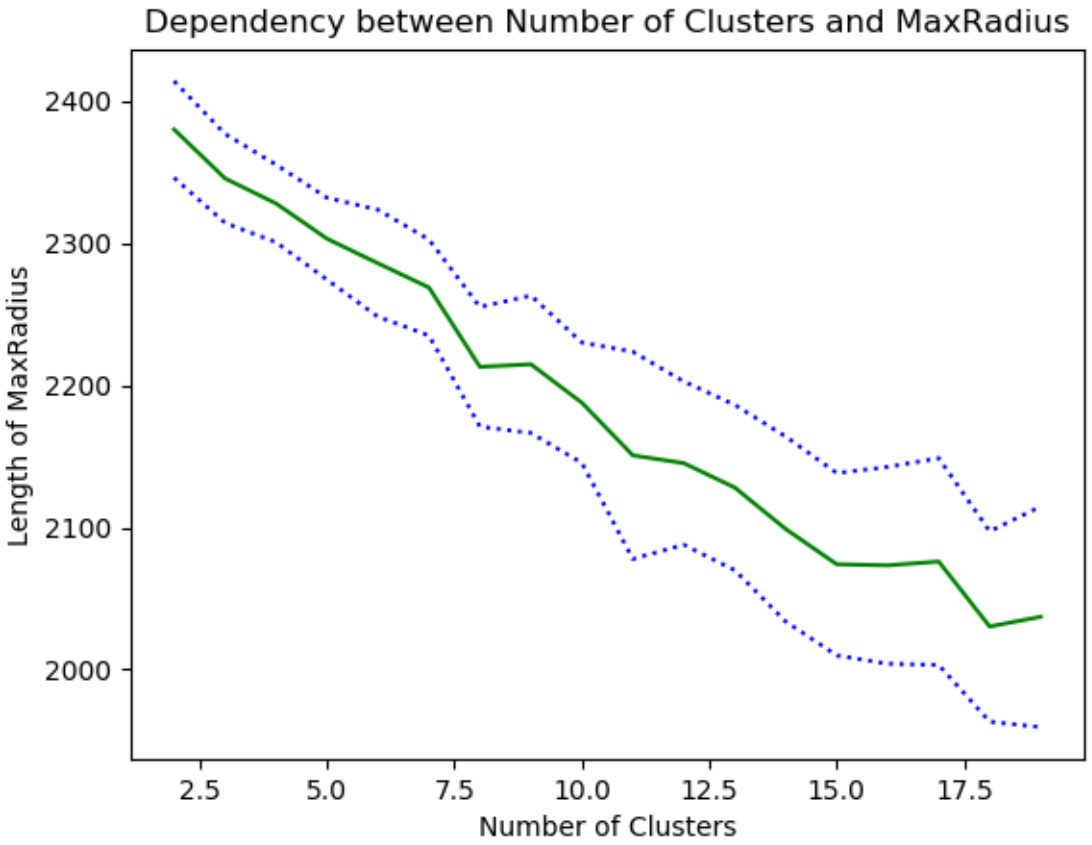
6 Результаты работы программы

Напоминание: диаметр кластера - максимальное расстояние между элементами в кластере. Радиус кластера - максимальное расстояние между центром кластера и элементами этого кластера.

Зависимость между количеством кластеров и диаметром кластера при длине последовательности, равной 2500 и количеством элементов, равным 50:



Зависимость между количеством кластеров и радиусом кластера при длине последовательности, равной 2500 и количеством элементов, равным 50:



Как видно из графиков, зависимость между количеством кластеров и длиной диаметра линейная.

7 Заключение

В данной работе был реализован алгоритм K-means для кластеризации генов. Для удобства вычислений нуклеотиды были заменены их векторными представлениями и на них была введена псевдометрика. Это сделало код гораздо более удобочитаемым и компактным, так как отпала необходимость вводить правило суммы для 16 возможных состояний нуклеотида. Результаты работы программы говорят нам о том, что количество кластеров и диаметр кластера зависят друг от друга по линейному закону, что ясно видно на полученных графиках. В ходе работы были применено множество библиотек Python, в том числе для распа-

раллеливания вычислений.

Список литературы

[1] Марк Лутц - Программирование на Python. Том 1, 4-е издание. Символ-Плюс, 2011

[2] Марк Лутц - Изучаем Python. Символ-Плюс, 2011

[3] Дурбин Р, Эдди Ш, Крог А, Митчисон Г. «Анализ биологических последовательностей». — М.-Ижевск: НИЦ «Регулярная и хаотичная динамика», 2006. — 480 с.

[4] Бородовский М., Екишева С. «Задачи и решения по анализу биологических последовательностей». — М.-Ижевск: НИЦ «Регулярная и хаотичная динамика», 2008. — 420 с

[5] Питер Брюс и Эндрю Брюс. “Практическая статистика для специалистов Data Science“, 2015

[6] Azilah Othman, Rosni Abdullah, Nur Aini Abdul Rashid, and Rosalina Abdul Salam - Parallel K-Means Clustering Algorithm on DNA Dataset, Conference: Parallel and Distributed Computing: Applications and Technologies, 5th International Conference, PDCAT 2004, Singapore, December 8-10, 2004, Proceedings

[7] Muhammad Faiz, Mohamed Saaid, Zuwairie Ibrahim, Marzuki Khalid, and NorHanizaSarmin - K-MEANS CLUSTERING FOR DNA COMPUTING READOUT METHOD IMPLEMENTEDON LIGHTCYCLER SYSTEM, UniversitiTeknologiMalaysia, 2003

[8] Chaoming Song, Shlomo Havlin, Hernan A. Makse - Self-similarity of complex networks, Journal Nature, Jan 27 2005

[9] M. Garey ; D. Johnson ; H. Witsenhausen - The complexity of the generalized Lloyd - Max problem, Published in: IEEE Transactions on Information Theory, 1982

[10] Adam Coates, Honglak Lee, Andrew Y. Ng - An Analysis of Single-Layer Networks in Unsupervised Feature Learning, Stanford University, 2011

[11] Bruce Alberts, Alexander Johnson, Julian Lewis, Martin Raff, Keith Roberts, and Peter Walter - Molecular Biology of the Cell, 4th edition,New York: Garland Science; 2002.

[12] Ilzins, O., Isea, R. and Hoebeke, J. Can Bioinformatics Be Considered as an Experimental Biological Science, 2015