Отчёт по лабораторной работе №7

Введение в работу с данными

Гань Чжаолун

Цель работы

Основной целью работы является специализированных пакетов Julia для обработки данных.

Выполнение лабораторной работы

7.2.1.1. Считывание данных

В Julia для работы с такого рода структурами данных используют пакеты CSV, DataFrames, RDatasets, FileIO:

```
# Обновление окружения:
using Pkg
Pkg.update

# Установка пакетов:
using Pkg
for p in ["CSV", "DataFrames", "RDatasets", "FileIO"]
Pkg.add(p)
end
```

```
   Updating registry at `~/.julia/registries/General`
######################################################################## 100.0%
  Resolving package versions...
No Changes to `~/.julia/environments/v1.5/Project.toml`
No Changes to `~/.julia/environments/v1.5/Manifest.toml`
  Resolving package versions...
No Changes to `~/.julia/environments/v1.5/Project.toml`
No Changes to `~/.julia/environments/v1.5/Manifest.toml`
  Resolving package versions...
No Changes to `~/.julia/environments/v1.5/Project.toml`
No Changes to `~/.julia/environments/v1.5/Manifest.toml`
  Resolving package versions...
No Changes to `~/.julia/environments/v1.5/Project.toml`
No Changes to `~/.julia/environments/v1.5/Manifest.toml`
```

```
using CSV, DataFrames, DelimitedFiles
```

Для заполнения массива данными для последующей обработки требуется считать данные из исходного файла и записать их в соответствующую структуру:

```
# Считывание данных и их запись в структуру:
P = CSV.File("programminglanguages.csv") |> DataFrame
```

73 rows × 2 columns

	year	language
	Int64	String
1	1951	Regional Assembly Language
2	1952	Autocode
3	1954	IPL

Далее приведём пример функции, в которой на входе указывается название языка программирования, а на выходе — год его создания:

Следует обратить внимание, что в данной функции мы никак не обрабатываем случай, когда языка программирования, переданного в функцию не существует.

```
# Функция определения по названию языка программирования года его создания:
function language_created_year(P,language::String)
    loc = findfirst(P[:,2].==language)
    return P[loc,1]
end
```

```
language_created_year (generic function with 1 method)
```

```
# Пример вызова функции и определение даты создания языка Python:
print(language_created_year(P,"Python"))
# Пример вызова функции и определение даты создания языка Julia:
language_created_year(P,"Julia")
```

```
1991
```

```
2012
```

В таблице содержится запись о языках программирования Julia и Python, вывод годов произошел успешно. Однако если мы попробуем написать julia с маленькой буквы, то получим ошибку, так как датафрейм не содержит такой записи.

```
language_created_year(P,"julia")
```

```
MethodError: no method matching getindex(::DataFrame, ::Nothing, ::Int64)
```

Для того, чтобы убрать в функции зависимость данных от регистра, необходимо изменить исходную функцию следующим образом:

```
# Функция определения по названию языка программирования
# года его создания (без учёта регистра):
function language_created_year_v2(P,language::String)
    loc = findfirst(lowercase.(P[:,2]).==lowercase.(language))
    return P[loc,1]
end
```

```
language_created_year_v2 (generic function with 1 method)
```

```
# Пример вызова функции и определение даты создания языка julia:
language_created_year_v2(P,"julia")
```

```
2012
```

Можно считывать данные построчно, с элементами, разделенными заданным разделителем:

```
# Построчное считывание данных с указанием разделителя:
Tx = readdlm("programminglanguages.csv", ',')
```

```
74×2 Array{Any,2}:
     "year"  "language"
  1951       "Regional Assembly Language"
  1952       "Autocode"
  1954       "IPL"
```

7.2.1.2. Запись данных в файл

Предположим, что требуется записать имеющиеся данные в файл. Для записи данных в формате CSV можно воспользоваться следующим вызовом:

```
# Запись данных в CSV-файл:
CSV.write("programming_languages_data2.csv", P)
```

```
"programming_languages_data2.csv"
```

Можно задать тип файла и разделитель данных:

```
# Пример записи данных в текстовый файл с разделителем ',':
writedlm("programming_languages_data.txt", Tx, ',')
```

```
# Пример записи данных в текстовый файл с разделителем '-':
writedlm("programming_languages_data2.txt", Tx, '-')
```

Можно проверить, используя readdlm, корректность считывания созданного текстового файла:

```
# Построчное считывание данных с указанием разделителя:
P_new_delim = readdlm("programming_languages_data2.txt", '-')
```

```
74×2 Array{Any,2}:
     "year"  "language"
 1951        "Regional Assembly Language"
 1952        "Autocode"
 1954        "IPL"
 1955        "FLOW-MATIC"
 1957        "FORTRAN"
```

7.2.1.3. Словари

При работе с данными бывает удобно записать их в формате словаря.

При инициализации словаря можно задать конкретные типы данных для ключей и значений:

```
# Инициализация словаря:
dict = Dict{Integer,Vector{String}}()
```

```
Dict{Integer,Array{String,1}}()
```

а можно инициировать пустой словарь, не задавая строго структуру:

```
# Инициализация словаря:
dict2 = Dict()
```

```
Dict{Any,Any}()
```

Далее требуется заполнить словарь ключами и годами, которые содержат все языки программирования, созданные в каждом году, в качестве значений:

```
# Заполнение словаря данными:
for i = 1:size(P,1)
    year,lang = P[i,:]
    if year in keys(dict)
        dict[year] = push!(dict[year],lang)
    else
        dict[year] = [lang]
    end
end
```

В результате при вызове словаря можно, выбрав любой год, узнать, какие языки программирования были созданы в этом году:

```
# Пример определения в словаре языков программирования, созданных в 2003 году:
dict[2003]
```

```
2-element Array{String,1}:
 "Groovy"
 "Scala"
```

7.2.1.4. DataFrames

На примере с данными о языках программирования и годах их создания зададим структуру DataFrame:

```
# Подгружаем пакет DataFrames:
using DataFrames
```

```
# Задаём переменную со структурой DataFrame:
df = DataFrame(year = P[:,1], language = P[:,2])
```

73 rows × 2 columns

	year	language
	Int64	String
1	1951	Regional Assembly Language
2	1952	Autocode
3	1954	IPL
4	1955	FLOW-MATIC
5	1957	FORTRAN
6	1957	COMTRAN
7	1958	LISP

Если требуется получить доступ к столбцам по имени заголовка, то необходимо добавить к имени заголовка двоеточие:

```
# Вывод всех значения столбца year:
df[!,:year]
```

```
73-element Array{Int64,1}:
 1951
 1952
 1954
 1955
```

Пакет DataFrames предоставляет возможность с помощью description получить основные статистические сведения о каждом столбце во фрейме данных:

```
# Получение статистических сведений о фрейме:
describe(df)
```

2 rows × 7 columns

	variable	mean	min	median	max	nmissing	eltype
	Symbol	Union...	Any	Union...	Any	Int64	DataType
1	year	1982.99	1951	1986.0	2014	0	Int64
2	language		ALGOL 58		dBase III	0	String

7.2.1.5. RDatasets

С данными можно работать также как с наборами данных через пакет RDatasets языка R:

```
# Подгружаем пакет RDatasets:
using RDatasets

# Задаём структуру данных в виде набора данных:
iris = dataset("datasets", "iris")
```

150 rows × 5 columns

	SepalLength	SepalWidth	PetalLength	PetalWidth	Species
	Float64	Float64	Float64	Float64	Cat...
1	5.1	3.5	1.4	0.2	setosa
2	4.9	3.0	1.4	0.2	setosa
3	4.7	3.2	1.3	0.2	setosa
4	4.6	3.1	1.5	0.2	setosa

В данном случает набор данных содержит сведения о цветах. При этом следует иметь в виду, что данные, загруженные с помощью набора данных, хранятся в виде DataFrame:

```
# Определения типа переменной:
typeof(iris)
```

```
DataFrame
```

Пакет RDatasets также предоставляет возможность с помощью description получить основные статистические сведения о каждом столбце в наборе данных:

```
describe(iris)
```

5 rows × 7 columns

	variable	mean	min	median	max	nmissing	eltype
	Symbol	Union...	Any	Union...	Any	Int64	DataType
1	SepalLength	5.84333	4.3	5.8	7.9	0	Float64
2	SepalWidth	3.05733	2.0	3.0	4.4	0	Float64
3	PetalLength	3.758	1.0	4.35	6.9	0	Float64
4	PetalWidth	1.19933	0.1	1.3	2.5	0	Float64
5	Species		setosa		virginica	0	CategoricalValue{String,UInt8}

7.2.1.6. Работа с переменными отсутствующего типа (Missing Values)

Пакет DataFrames позволяет использовать так называемый «отсутствующий» тип:

```
# Отсутствующий тип:
a = missing
typeof(a)
```

```
Missing
```

В операции сложения числа и переменной с отсутствующим типом значение также будет иметь отсутствующий тип:

```
# Пример операции с переменной отсутствующего типа:
a + 1
```

```
missing
```

Приведём пример работы с данными, среди которых есть данные с отсутствующим типом.

Предположим есть перечень продуктов, для которых заданы калории:

```
# Определение перечня продуктов:
foods = ["apple", "cucumber", "tomato", "banana"]
# Определение калорий:
calories = [missing, 47, 22, 105]
```

```
4-element Array{Union{Missing, Int64},1}:
     missing
  47
  22
 105
```

В массиве значений калорий есть значение с отсутствующим типом:

```
# Определение типа переменной:
typeof(calories)
```

```
Array{Union{Missing, Int64},1}
```

При попытке получить среднее значение калорий, ничего не получится из-за наличия переменной с отсутствующим типом:

```
# Подключаем пакет Statistics:
using Statistics

# Определение среднего значения:
mean(calories)
```

```
missing
```

Для решения этой проблемы необходимо игнорировать отсутствующий тип:

```
# Определение среднего значения без значений с отсутствующим типом:
mean(skipmissing(calories))
```

```
58.0
```

Далее показано, как можно сформировать таблицы данных и объединить их в один фрейм:

```
# Задание сведений о ценах:
prices = [0.85, 1.6, 0.8, 0.6]
# Формирование данных о калориях:
dataframe_calories = DataFrame(item=foods, calories=calories)
# Формирование данных о ценах:
dataframe_prices = DataFrame(item=foods, price=prices)
# Объединение данных о калориях и ценах:
DF = innerjoin(dataframe_calories, dataframe_prices, on=:item)
```

4 rows × 3 columns

	item	calories	price
	String	Int64?	Float64
1	apple	*missing*	0.85
2	cucumber	47	1.6
3	tomato	22	0.8
4	banana	105	0.6

В данном пункте выдавалась ошибка относительно использования join с датафреймами. Я использовала функцию innerjoin, так как join использовать нельзя. Ссылка на документацию, в которой описаны методы присоединения таблиц для датафреймов:

https://dataframes.juliadata.org/stable/man/joins/

7.2.1.7. FileIO

В Julia можно работать с так называемыми «сырыми» данными, используя пакет FileIO:

```
# Подключаем пакет FileIO:
using FileIO
```

Попробуем посмотреть, как Julia работает с изображениями. Подключим соответствующий пакет:

```
# Подключаем пакет ImageIO:
import Pkg
Pkg.add("ImageIO")
```

```
  Resolving package versions...
No Changes to `~/.julia/environments/v1.5/Project.toml`
No Changes to `~/.julia/environments/v1.5/Manifest.toml`
```

Загрузим изображение (в данном случае логотип Julia):

```
# Загрузка изображения:
X1 = load("julialogo.png")
```

```
460×460 Array{RGBA{N0f8},2} with eltype ColorTypes.RGBA{FixedPointNumbers.Normed{UInt8,8}}:
 RGBA{N0f8}(0.0,0.0,0.0,0.0)  …  RGBA{N0f8}(0.0,0.0,0.0,0.0)
 RGBA{N0f8}(0.0,0.0,0.0,0.0)     RGBA{N0f8}(0.0,0.0,0.0,0.0)
 RGBA{N0f8}(0.0,0.0,0.0,0.0)     RGBA{N0f8}(0.0,0.0,0.0,0.0)
 RGBA{N0f8}(0.0,0.0,0.0,0.0)     RGBA{N0f8}(0.0,0.0,0.0,0.0)
 RGBA{N0f8}(0.0,0.0,0.0,0.0)     RGBA{N0f8}(0.0,0.0,0.0,0.0)
 RGBA{N0f8}(0.0,0.0,0.0,0.0)  …  RGBA{N0f8}(0.0,0.0,0.0,0.0)
 RGBA{N0f8}(0.0,0.0,0.0,0.0)     RGBA{N0f8}(0.0,0.0,0.0,0.0)
```

Julia хранит изображение в виде множества цветов:

```
# Определение типа и размера данных:
@show typeof(X1);
@show size(X1);
```

```
typeof(X1) = Array{ColorTypes.RGBA{FixedPointNumbers.Normed{UInt8,8}},2}
size(X1) = (460, 460)
```

7.2.2. Обработка данных: стандартные алгоритмы машинного обучения в Julia

7.2.2.1. Кластеризация данных. Метод k-средних

Рассмотрим задачу кластеризации данных на примере данных о недвижимости. Файл с данными houses.csv содержит список транзакций с недвижимостью в районе Сакраменто, о которых было сообщено в течение определённого числа дней.

Сначала подключим необходимые для работы пакеты:

```
# Загрузка пакетов:
import Pkg
Pkg.add("DataFrames")
Pkg.add("Statistics")
import Pkg
Pkg.add("Plots")
using DataFrames, CSV, Plots
```

```
  Resolving package versions...
No Changes to `~/.julia/environments/v1.5/Project.toml`
No Changes to `~/.julia/environments/v1.5/Manifest.toml`
  Resolving package versions...
No Changes to `~/.julia/environments/v1.5/Project.toml`
No Changes to `~/.julia/environments/v1.5/Manifest.toml`
  Resolving package versions...
No Changes to `~/.julia/environments/v1.5/Project.toml`
No Changes to `~/.julia/environments/v1.5/Manifest.toml`
```

Затем загрузим данные:

```
# Загрузка данных:
houses = CSV.File("houses.csv") |> DataFrame
```

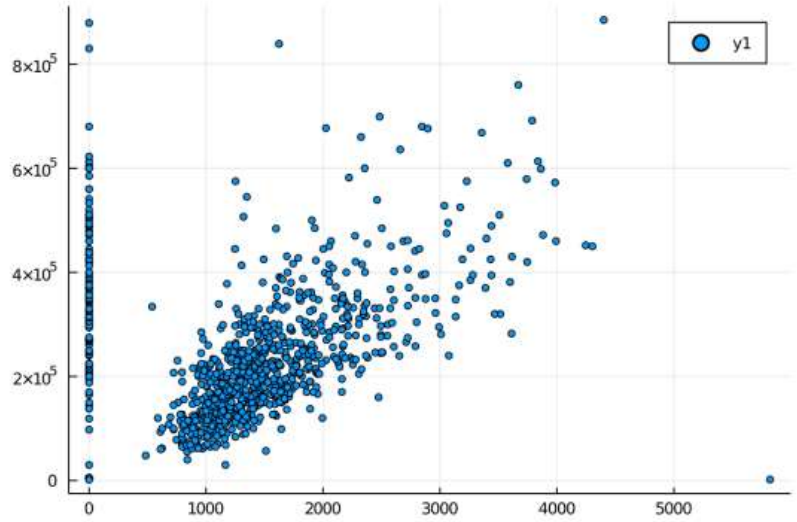
985 rows × 12 columns (omitted printing of 5 columns)

	street	city	zip	state	beds	baths	sqft
	String	String	Int64	String	Int64	Int64	Int64
1	3526 HIGH ST	SACRAMENTO	95838	CA	2	1	836
2	51 OMAHA CT	SACRAMENTO	95823	CA	3	1	1167
3	2796 BRANCH ST	SACRAMENTO	95815	CA	2	1	796
4	2805 JANETTE WAY	SACRAMENTO	95815	CA	2	1	852

Построим график цен на недвижимость в зависимости от площади :

```
# Построение графика:
plot(size=(500,500),leg=false)

x = houses[!,:sqft]
y = houses[!,:price]
scatter(x,y,markersize=3)
```

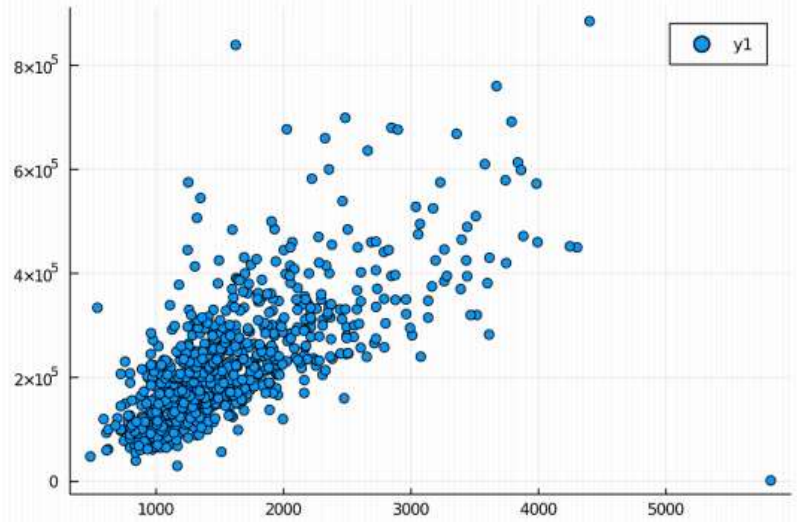


Как видно из графика, имеются так называемые «артефакты», т.е. проявляются отсутствующие или невозможные сведения в исходных данных, например, цены на недвижимость нулевой площади.

Для того чтобы избавиться от такого эффекта, можно отфильтровать и исключить такие значения, получить более корректный график цен :

```
# Фильтрация данных по заданному условию:
filter_houses = houses[houses[!,:sqft].>0,:]

# Построение графика:
x = filter_houses[!,:sqft]
y = filter_houses[!,:price]
scatter(x,y)
```



Используя для фильтрации значений функцию by пакета DataFrames и для вычисления среднего значения функцию mean пакета Statistics, можно посмотреть среднюю цену домов определённого типа:

```
# Подключение пакета Statistics:
using Statistics

# Определение средней цены для определённого типа домов:
combine(groupby(filter_houses, :type), filter_houses->mean(filter_houses[!,:price]))
```

3 rows × 2 columns

	type	x1
	String	Float64
1	Residential	2.34802e5
2	Condo	1.34213e5
3	Multi-Family	2.24535e5

В данном пункте также была ошибка, функция by не доступна для использования. Поэтому из предложенных вариантов я выбрала combine(groupby(), …).

Отфильтровав таким образом данные, можно приступить к формированию кластеров. Сначала подключаем необходимые пакеты и формируем данные в нужном виде:

```
# Подключение пакета Clustering:
import Pkg
Pkg.add("Clustering")
using Clustering

# Добавление данных :latitude и :longitude в новый фрейм:
X = filter_houses[!, [:latitude, :longitude]]
```

```
  Resolving package versions...
No Changes to `~/.julia/environments/v1.5/Project.toml`
No Changes to `~/.julia/environments/v1.5/Manifest.toml`
```

814 rows × 2 columns

	latitude	longitude
	Float64	Float64
1	38.6319	-121.435
2	38.4789	-121.431
3	38.6183	-121.444
4	38.6168	-121.439

```
# Конвертация данных в матричный вид:
X = convert(Matrix{Float64}, X)
```

```
814×2 Array{Float64,2}:
 38.6319  -121.435
 38.4789  -121.431
 38.6183  -121.444
 38.6168  -121.439
 38.5195  -121.436
 38.6626  -121.328
 38.6817  -121.352
 38.5351  -121.481
```

Каждая функция хранится в виде строки X, но можно транспонировать получившуюся матрицу, чтобы иметь возможность работать с столбцами данных X:

```
# Транспонирование матрицы с данными:
X = X'
```

```
2×814 LinearAlgebra.Adjoint{Float64,Array{Float64,2}}:
   38.6319    38.4789    38.6183  …    38.7088    38.417    38.6552
 -121.435   -121.431   -121.444      -121.257   -121.397  -121.076
```

В качестве критерия для формирования кластеров данных и определения количества кластеров попробуем использовать количество почтовых индексов:

```
# Задание количества кластеров:
k = length(unique(filter_houses[!,:zip]))
```

```
66
```

Для определения k-среднего можно воспользоваться соответствующей функцией пакета Statistics:

```
# Определение k-среднего:
C = kmeans(X,k)
```

```
KmeansResult{Array{Float64,2},Float64,Int64}([38.47325135714285 38.707247173913046 … 38.46755533333334 38.671155875;
-121.38218371428569 -121.30996678260867 … -121.31786866666668 -121.2249695], [8, 14, 8, 8, 49, 31, 41, 43, 16, 19  …
12, 36, 14, 14, 58, 62, 49, 37, 24, 47], [0.00015710817388026044, 0.00036038724283571355, 0.00035021260555367917, 0.0
0045296605458133854, 0.00017433685206924565, 0.00017304826178587973, 4.942436862620525e-5, 2.0411680452525616e-5, 0.0
0036852071207249537, 0.00011686658399412408  …  6.936186400707811e-5, 0.0015088486943568569, 0.00010623546768329106,
0.00026382742362329736, 0.00013415837747743353, 0.00010097602716996334, 0.00044728680222760886, 0.000177583664481062
4, 0.00030232657445594 67, 2.8308819310041144e-5], [14, 23, 4, 12, 19, 14, 5, 53, 34, 1  …  9, 7, 8, 3, 2, 24, 1, 1,
3, 8], [14, 23, 4, 12, 19, 14, 5, 53, 34, 1  …  9, 7, 8, 3, 2, 24, 1, 1, 3, 8], 0.21377727673461777, 13, true)
```

Далее сформируем новый фрейм, включающий исходные данные о недвижимости и столбец с данными о назначенном каждому дому кластере:

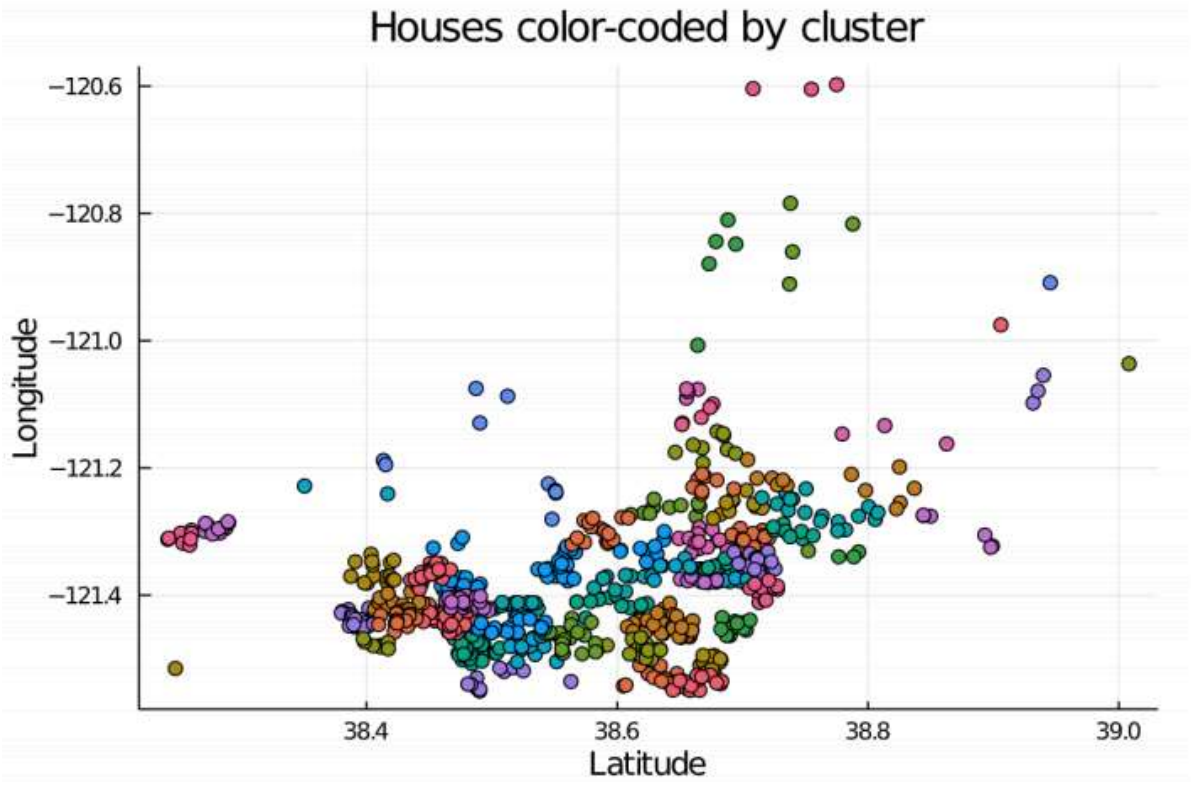
```
# Формирование фрейма данных:
df = DataFrame(cluster = C.assignments,city = filter_houses[!,:city],
latitude = filter_houses[!,:latitude],longitude = filter_houses[!,:longitude],zip = filter_houses[!,:zip])
```

814 rows × 5 columns

	cluster	city	latitude	longitude	zip
	Int64	String	Float64	Float64	Int64
1	8	SACRAMENTO	38.6319	-121.435	95838
2	14	SACRAMENTO	38.4789	-121.431	95823
3	8	SACRAMENTO	38.6183	-121.444	95815
4	8	SACRAMENTO	38.6168	-121.439	95815
5	49	SACRAMENTO	38.5195	-121.436	95824
6	31	SACRAMENTO	38.6626	-121.328	95841
7	41	SACRAMENTO	38.6817	-121.352	95842
8	43	SACRAMENTO	38.5351	-121.481	95820

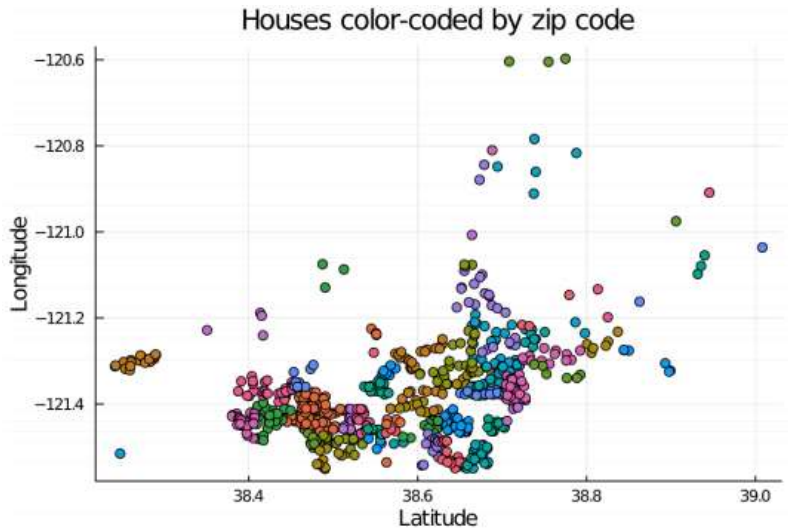
Построим график, обозначив каждый кластер отдельным цветом:

```
clusters_figure = plot(legend = false)
for i = 1:k
    clustered_houses = df[df[!,:cluster].== i,:]
    xvals = clustered_houses[!,:latitude]
    yvals = clustered_houses[!,:longitude]
    scatter!(clusters_figure,xvals,yvals,markersize=4)
end
xlabel!("Latitude")
ylabel!("Longitude")
title!("Houses color-coded by cluster")
display(clusters_figure)
```



Построим график, раскрасив кластеры по почтовому индексу:

```
unique_zips = unique(filter_houses[!,:zip])
zips_figure = plot(legend = false)
for uzip in unique_zips
    subs = filter_houses[filter_houses[!,:zip].==uzip,:]
    x = subs[!,:latitude]
    y = subs[!,:longitude]
    scatter!(zips_figure,x,y)
end
xlabel!("Latitude")
ylabel!("Longitude")
title!("Houses color-coded by zip code")
display(zips_figure)
```



7.2.2.2. Кластеризация данных. Метод k ближайших соседей

Рассмотрим использование метода k ближайших соседей на примере того же файла с данными об объектах недвижимости в Сакраменто.

```
# Подключение пакета NearestNeighbors:
import Pkg
Pkg.add("NearestNeighbors")
using NearestNeighbors
```

```
  Resolving package versions...
No Changes to `~/.julia/environments/v1.5/Project.toml`
No Changes to `~/.julia/environments/v1.5/Manifest.toml`
```

Найдём k-среднее одного из объектов недвижимости:

```
knearest = 10
id = 70
point = X[:,id]
```

```
2-element Array{Float64,1}:
   38.44004
 -121.421012
```

Определим ближайших соседей:

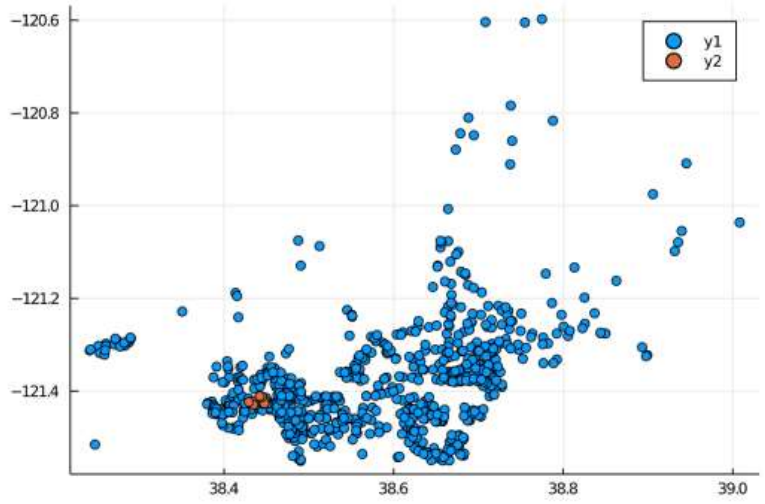
```
# Поиск ближайших соседей:
kdtree = KDTree(X)
idxs, dists = knn(kdtree, point, knearest, true)
```

```
([70, 764, 196, 125, 557, 368, 415, 92, 112, 683], [0.0, 0.006264891539364138, 0.00825320259050462, 0.008473585132630
057, 0.009164073548370188, 0.009405065124697706, 0.009921759722950759, 0.009941028618812013, 0.010332637707777167, 0.
011168993911721985])
```

Отобразим на графике соседей выбранного объекта недвижимости :

```
# Все объекты недвижимости:
x = filter_houses[!,:latitude];
y = filter_houses[!,:longitude];
scatter(x,y)

# Соседи:
x = filter_houses[idxs,:latitude];
y = filter_houses[idxs,:longitude];
scatter!(x,y)
```



Используя индексы idxs и функцию :city для индексации в DataFrame filter_houses, можно определить районы соседних домов:

```
# Фильтрация по районам соседних домов:
cities = filter_houses[idxs,:city]
```

```
10-element Array{String,1}:
 "SACRAMENTO"
 "ELK GROVE"
 "SACRAMENTO"
```

7.2.2.3. Обработка данных. Метод главных компонент

На примере с данными о недвижимости попробуем уменьшить размеры данных о цене и площади из набора данных домов:

```
# Фрейм с указанием площади и цены недвижимости:
F = filter_houses[!,[:sqft,:price]]
```

```
...
```

```
# Конвертация данных в массив:
F = convert(Array{Float64,2},F)'
```

```
2×814 LinearAlgebra.Adjoint{Float64,Array{Float64,2}}:
   836.0   1167.0    796.0    852.0    797.0  …    1216.0    1685.0    1362.0
 59222.0  68212.0  68880.0  69307.0  81900.0     235000.0  235301.0  235738.0
```

Далее подключим пакет MultivariateStats, чтобы использовать метод главных компонент:

```
# Подключение пакета MultivariateStats:
import Pkg
Pkg.add("MultivariateStats")
using MultivariateStats
```

```
  Resolving package versions...
No Changes to `~/.julia/environments/v1.5/Project.toml`
No Changes to `~/.julia/environments/v1.5/Manifest.toml`
```

Далее используем специальную функцию fit и приведём имеющийся набор данных к распределению, к которому можно применить метод главных компонент (PCA):

```
# Приведение типов данных к распределению для PCA:
M = fit(PCA, F)
```

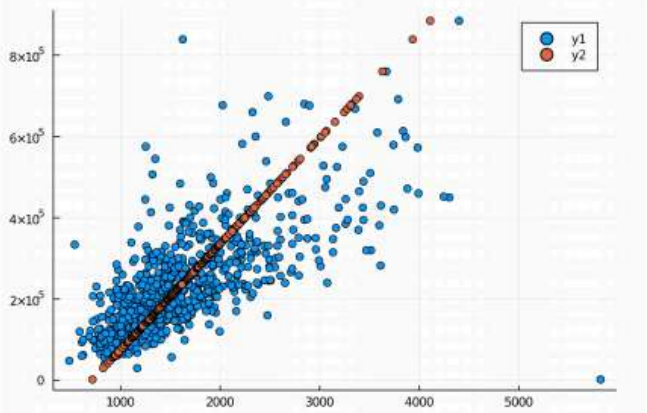
```
PCA(indim = 2, outdim = 1, principalratio = 0.9999840784692097)
```

Далее воспользуемся функцией reconstruct, чтобы выделить данные с главными компонентами в отдельную переменную Xr, значения которой в последствии можно вывести на графике :

```
# Выделение значений главных компонент в отдельную переменную:
y = MultivariateStats.transform(M, F)

Xr = reconstruct(M, y)

# Построение графика с выделением главных компонент:
scatter(F[1,:], F[2,:])
scatter!(Xr[1,:], Xr[2,:])
```

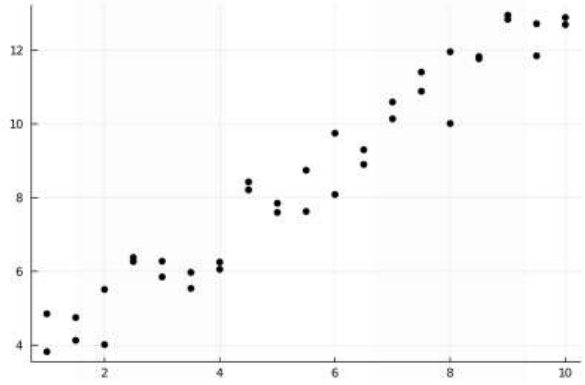


7.2.2.4. Обработка данных. Линейная регрессия

Зададим случайный набор данных (можно использовать и полученные экспериментальным путём какие-то данные). Попробуем найти для данных лучшее соответствие:

```
xvals = repeat(1:0.5:10,inner=2)
yvals = 3 .+ xvals + 2*rand(length(xvals)) .- 1

scatter(xvals,yvals,color=:black,leg=false)
```



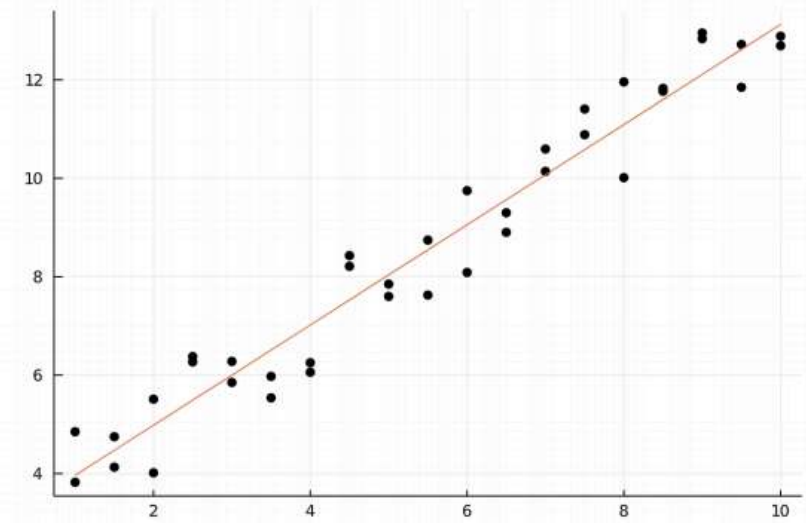
Определим функцию линейной регрессии:

```
function find_best_fit(xvals,yvals)
    meanx = mean(xvals)
    meany = mean(yvals)
    stdx = std(xvals)
    stdy = std(yvals)
    r = cor(xvals,yvals)
    a = r*stdy/stdx
    b = meany - a*meanx
    return a,b
end
```

```
find_best_fit (generic function with 1 method)
```

Применим функцию линейной регрессии для построения соответствующего графика значений:

```
a,b = find_best_fit(xvals,yvals)
ynew = a * xvals .+ b
plot!(xvals,ynew)
```



Сгенерируем больший набор данных:

```
xvals = 1:100000;
xvals = repeat(xvals,inner=3);
yvals = 3 .+ xvals + 2*rand(length(xvals)) .- 1;
```

```
@show size(xvals)
@show size(yvals)
```

```
size(xvals) = (300000,)
size(yvals) = (300000,)

(300000,)
```

```
@time a,b = find_best_fit(xvals,yvals)
```

```
  0.122454 seconds (245.55 k allocations: 13.098 MiB)

(0.9999999784106886, 3.000567530776607)
```

Для сравнения реализуем подобный код на языке Python:

```
import Pkg
Pkg.add("PyCall")
Pkg.add("Conda")
using PyCall
using Conda
```

```
  Resolving package versions...
No Changes to `~/.julia/environments/v1.5/Project.toml`
No Changes to `~/.julia/environments/v1.5/Manifest.toml`
  Resolving package versions...
No Changes to `~/.julia/environments/v1.5/Project.toml`
No Changes to `~/.julia/environments/v1.5/Manifest.toml`
```

```
py"""
import numpy
def find_best_fit_python(xvals,yvals):
    meanx = numpy.mean(xvals)
    meany = numpy.mean(yvals)
    stdx = numpy.std(xvals)
    stdy = numpy.std(yvals)
    r = numpy.corrcoef(xvals,yvals)[0][1]
    a = r*stdy/stdx
    b = meany - a*meanx
    return a,b
"""
```

```
find_best_fit_python = py"find_best_fit_python"
```

```
PyObject <function find_best_fit_python at 0x7fce61fa7c10>
```

```
xpy = PyObject(xvals)
ypy = PyObject(yvals)
@time a,b = find_best_fit_python(xpy,ypy)
```

```
  0.132079 seconds (199.06 k allocations: 10.347 MiB)

(0.9933169392041729, 337.15586937617627)
```

Используем пакет для анализа производительности, чтобы провести сравнение:

```
import Pkg
Pkg.add("BenchmarkTools")
using BenchmarkTools

@btime a,b = find_best_fit_python(xvals,yvals)
@btime a,b = find_best_fit(xvals,yvals)
```

```
  Resolving package versions...
No Changes to `~/.julia/environments/v1.5/Project.toml`
No Changes to `~/.julia/environments/v1.5/Manifest.toml`
```

```
  6.190 ms (27 allocations: 1.02 KiB)
  1.232 ms (1 allocation: 32 bytes)

(0.9999999784106886, 3.000567530776607)
```

7.4. Задания для самостоятельного выполнения

7.4.1. Кластеризация

1.Загружаем данные

```
using RDatasets
iris = dataset("datasets", "iris")
```

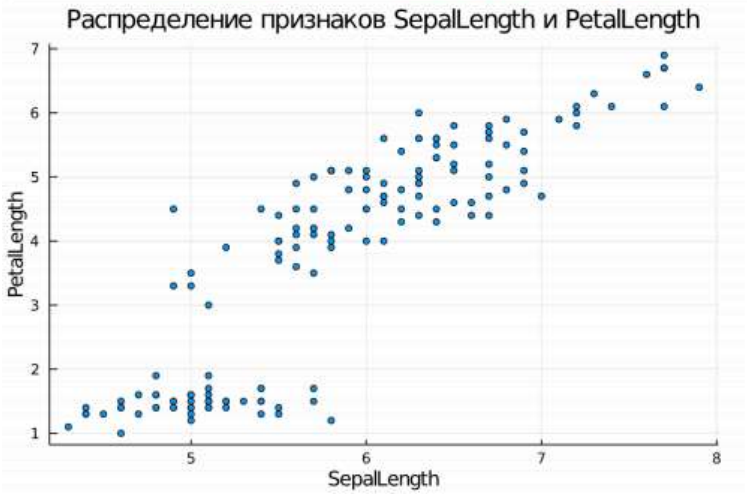
150 rows × 5 columns

	SepalLength	SepalWidth	PetalLength	PetalWidth	Species
	Float64	Float64	Float64	Float64	Cat...
1	5.1	3.5	1.4	0.2	setosa
2	4.9	3.0	1.4	0.2	setosa
3	4.7	3.2	1.3	0.2	setosa
4	4.6	3.1	1.5	0.2	setosa

2.Для кластеризации я выбрала 2 признака - SepalLength и PetalLength. Построим в начале точечный график их распределения:

```
# Построение графика:
plot(size=(500,500),leg=false)

x = iris[!,:SepalLength]
y = iris[!,:PetalLength]
scatter(x, y, markersize=3, title="Распределение признаков SepalLength и PetalLength",
    xlabel="SepalLength", ylabel="PetalLength", leg=false)
```



3.Добавим данные в новый фрейм:

```
# Добавление данных :latitude и :longitude в новый фрейм:
X = iris[!, [:SepalLength, :PetalLength]]
```

150 rows × 2 columns

	SepalLength	PetalLength
	Float64	Float64
1	5.1	1.4
2	4.9	1.4
3	4.7	1.3
4	4.6	1.5
5	5.0	1.4

4.Конвертируем данные в матричный вид и транспонируем их:

```
# Конвертация данных в матричный вид:
X = convert(Matrix{Float64}, X)
# Транспонирование матрицы с данными:
X = X'
```

```
2×150 LinearAlgebra.Adjoint{Float64,Array{Float64,2}}:
 5.1  4.9  4.7  4.6  5.0  5.4  4.6  5.0  …  6.8  6.7  6.7  6.3  6.5  6.2  5.9
 1.4  1.4  1.3  1.5  1.4  1.7  1.4  1.5     5.9  5.7  5.2  5.0  5.2  5.4  5.1
```

5.Я выдвинула гипотезу, что данные признаки могут зависеть от вида Ириса. Поэтому посчитаем количество уникальных видов и возьмем данное количество кластеров:

```
# Задание количества кластеров на основе вида ирисов:
k = length(unique(iris[!,:Species]))
```

```
3
```

```
# Определение k-среднего:
C = kmeans(X,k)
```

```
KmeansResult{Array{Float64,2},Float64,Int64}([5.874137931034483 6.839024390243902 5.007843137254902; 4.39310344827586
3 5.678048780487804 1.4921568627450983], [3, 3, 3, 3, 3, 3, 3, 3, 3, 3  …  2, 2, 1, 2, 2, 2, 1, 2, 2, 1], [0.01698577
4702035883, 0.02012302960400092, 0.13169165705497932, 0.16639753940791735, 0.008554402153016838, 0.1969857747020427,
0.1748289119569364, 0.00012302960399779295, 0.37796616685889717, 0.011691657054981874  …  0.0254193932183, 0.33785841
760854396, 0.5051991676575369, 0.05078524687684194, 0.01980963712077255, 0.24785841760854055, 0.5496819262782395, 0.3
4346817370612825, 0.48566329565733213, 0.5003715814506506], [58, 41, 51], [58, 41, 51], 53.80997864410648, 10, true)
```

6.Формируем фрейм данных с указанием кластера:

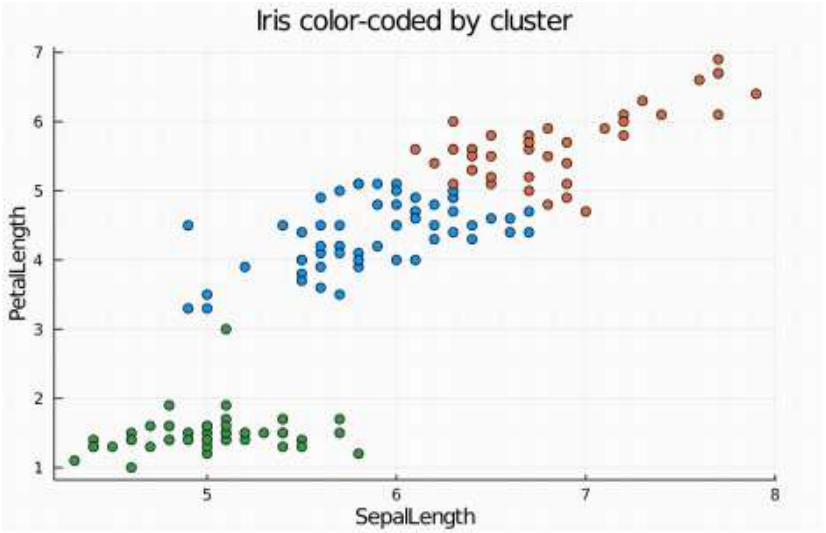
```
# Формирование фрейма данных:
iris_new = DataFrame(cluster = C.assignments,
SepalLength = iris[!,:SepalLength], PetalLength = iris[!,:PetalLength], Species = iris[!,:Species])
```

150 rows × 4 columns

	cluster	SepalLength	PetalLength	Species
	Int64	Float64	Float64	Cat...
1	3	5.1	1.4	setosa
2	3	4.9	1.4	setosa
3	3	4.7	1.3	setosa
4	3	4.6	1.5	setosa

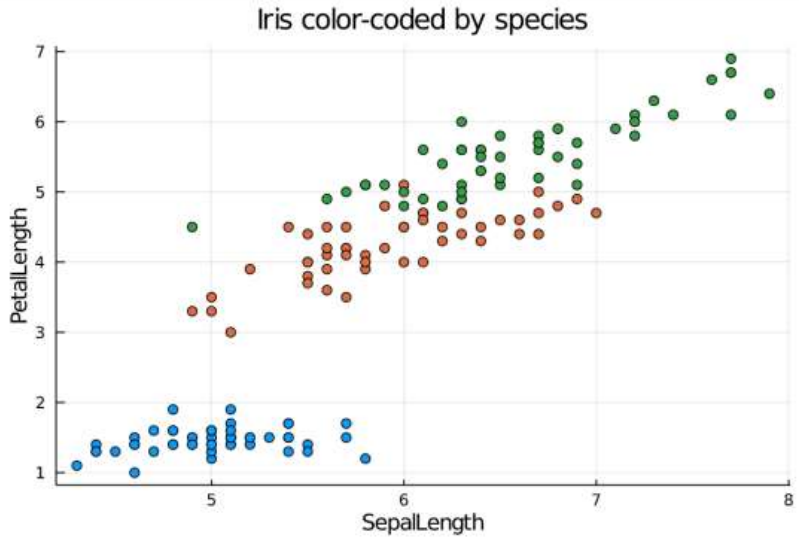
7.Построим график датафрейма с распределением цветов по кластерам:

```
clusters_figure = plot(legend = false)
for i = 1:k
    iris_new_clustered = iris_new[iris_new[!,:cluster].== i,:]
    xvals = iris_new_clustered[!,:SepalLength]
    yvals = iris_new_clustered[!,:PetalLength]
    scatter!(clusters_figure, xvals, yvals, markersize=4)
end
xlabel!("SepalLength")
ylabel!("PetalLength")
title!("Iris color-coded by cluster")
display(clusters_figure)
```



8.Теперь построим графики с распределением цветов по самому виду Ирисов:

```
unique_species = unique(iris[!,:Species])
species_figure = plot(legend = false)
for uspecies in unique_species
    iris_sp = iris[iris[!,:Species].==uspecies,:]
    x = iris_sp[!,:SepalLength]
    y = iris_sp[!,:PetalLength]
    scatter!(species_figure, x, y)
end
xlabel!("SepalLength")
ylabel!("PetalLength")
title!("Iris color-coded by species")
display(species_figure)
```



7.4.2. Регрессия (метод наименьших квадратов в случае линейной регрессии)

Часть 1

Пусть регрессионная зависимость является линейной. Матрица наблюдений факторов X имеет размерность N × 3 randn, массив результатов N × 1, регрессионная зависимость является линейной. Найдите МНК-оценку для линейной модели.

Для начала запишем данные:

```
X = randn(1000, 3)
a0 = rand(3)
print(a0)
y = X * a0 + 0.1 * randn(1000);
```

```
[0.6488768882014346, 0.04355456496646215, 0.5560984913725726]
```

Далее я создала функцию linear*regression*model. В которой изначально создаю матрицу X2, состоящую из единиц. Далее присоединяю этот столбец к X. А затем решаю систему линейных уравнений. Выходит результат: -0.00140559

```
function linear_regression_model(X, y)
    X2 = ones(1000)
    X = hcat(X, X2)
    return  X \ y
end
```

```
linear_regression_model (generic function with 1 method)
```

```
a = linear_regression_model(X, y)
print(a)
```

```
[0.6468629061323115, 0.04253975444213305, 0.5580332821634576, -0.0014055938037774972]
```

Добавление столбца с единицами обусловлено тем, что мы ищем свободный коэффициент. Суть метода наименьших квадратов:

Задача заключается в нахождении коэффициентов линейной зависимости, при которых функция двух переменных a и b принимает наименьшее значение. То есть, при данных a и b сумма квадратов отклонений экспериментальных данных от найденной прямой будет наименьшей. Таким образом, решение примера сводится к нахождению экстремума функции двух переменных.

Сравню полученные результаты с результатами использования llsq из MultivariateStats.jl.

```
using MultivariateStats
# solve using llsq
a = llsq(X, y)
print(a)
```

```
[0.6468629061323111, 0.04253975444213306, 0.5580332821634573, -0.0014055938037775108]
```

Как мы видим результат практически идентичен.

Сравню результаты с результатами использования регулярной регрессии наименьших квадратов из GLM.jl.

```
Pkg.add("GLM")
using GLM, DataFrames
```

```
  Resolving package versions...
No Changes to `~/.julia/environments/v1.5/Project.toml`
No Changes to `~/.julia/environments/v1.5/Manifest.toml`
```

После установки пакета создадим датафрейм, в который запишем y и разобьем X на 3 столбца - x1, x2 и x3.

Результат:

```
X1 = X[:,1]
X2 = X[:,2]
X3 = X[:,3]

data = DataFrame(y = y, x1 = X1, x2 = X2, x3 = X3);

lm(@formula(y ~ x1 + x2 + x3), data)
```

```
y ~ 1 + x1 + x2 + x3

Coefficients:
```

	Coef.	Std. Error	t	Pr(>|t|)	Lower 95%	Upper 95%
(Intercept)	-0.00140559	0.00310024	-0.45	0.6504	-0.00748934	0.00467815
x1	0.646863	0.00306265	211.21	<1e-99	0.640853	0.652873
x2	0.0425398	0.00328213	12.96	<1e-34	0.0360991	0.0489804
x3	0.558033	0.00308713	180.76	<1e-99	0.551975	0.564091

Результат получился таким же. Intercept во всех 3 случаях принимало значение -0.00140559.

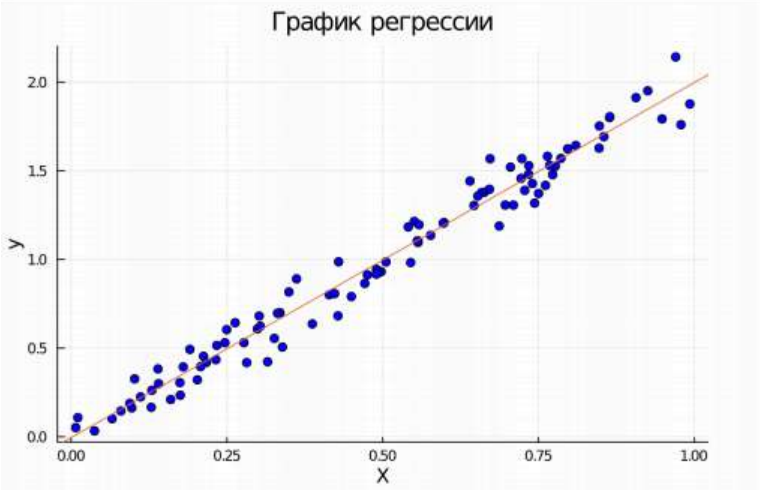
Часть 2

Найдите линию регрессии, используя данные (X, y). Постройте график (X, y), используя точечный график. Добавьте линию регрессии, используя abline!. Добавьте заголовок «График регрессии» и подпишите оси x и y.

```
X = rand(100);
y = 2*X + 0.1 * randn(100);
```

```
a, b = find_best_fit(X, y)
ynew = a * X .+ b

scatter(X, y, title="График регрессии", xlabel="X", ylabel="y", color=:blue, leg=false, line=:scatter)
Plots.abline!(a, b, line=:solid)
```



7.4.3. Модель ценообразования биномиальных опционов

а) Пусть S = 100, T = 1, n = 10000, σ = 0.3 и r = 0.08. Попробуйте построить траекторию курса акций. Функция rand () генерирует случайное число от 0 до 1. Вы можете использовать функцию построения графика из библиотеки графиков.

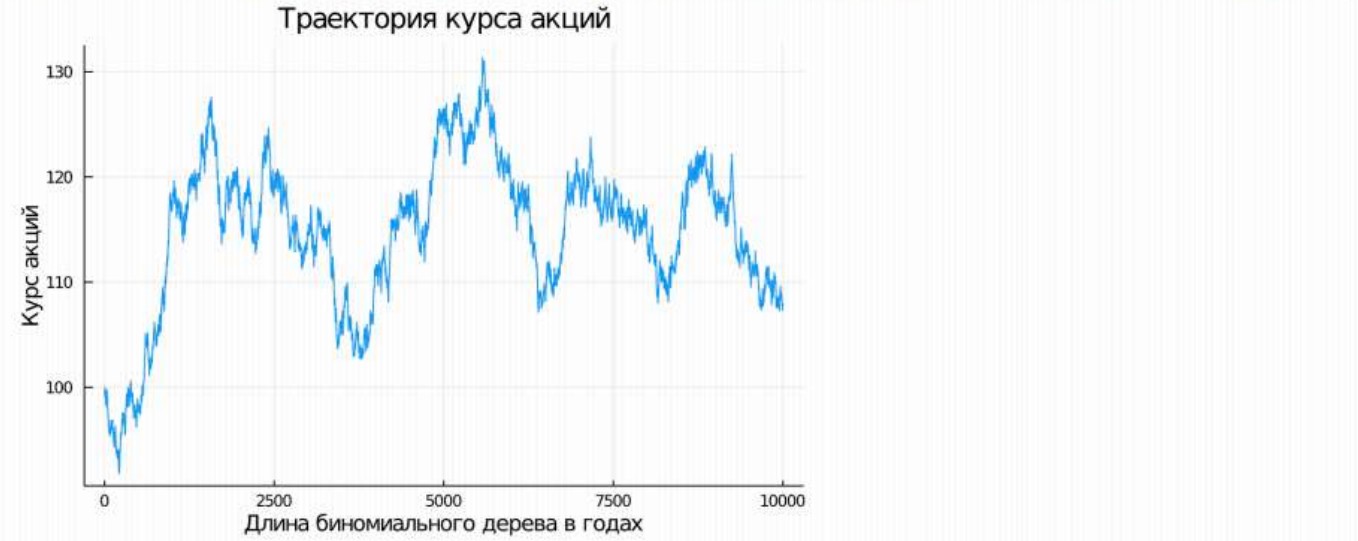
```julia
S = 100
T = 1
n = 10000
sigma = 0.3
r = 0.08

h = T / n # длина одного периода;
u = exp(r*h + sigma * sqrt(h))
d = exp(r*h - sigma * sqrt(h))
p = (exp(r*h) - d)/(u - d)

j = 0
stockTree = []
append!(stockTree, S)
for i in 1:n
    k = rand()
    if k < p
        append!(stockTree, S * (u^(i - j)) * (d ^ j))
    else
        append!(stockTree, S * (u^(i - j)) * (d ^ (j + 1)))
        j = j + 1
    end
end
```

Построим график:

```julia
using Plots
plot(stockTree, title="Траектория курса акций", xlabel="Длина биномиального дерева в годах", ylabel="Курс акций", leg=false)
```



b) Создайте функцию createPath (S :: Float64, r :: Float64, sigma :: Float64, T :: Float64, n :: Int64), которая создает траекторию цены акции с учетом начальных параметров. Используйте createPath, чтобы создать 10 разных траекторий и построить их все на одном графике.

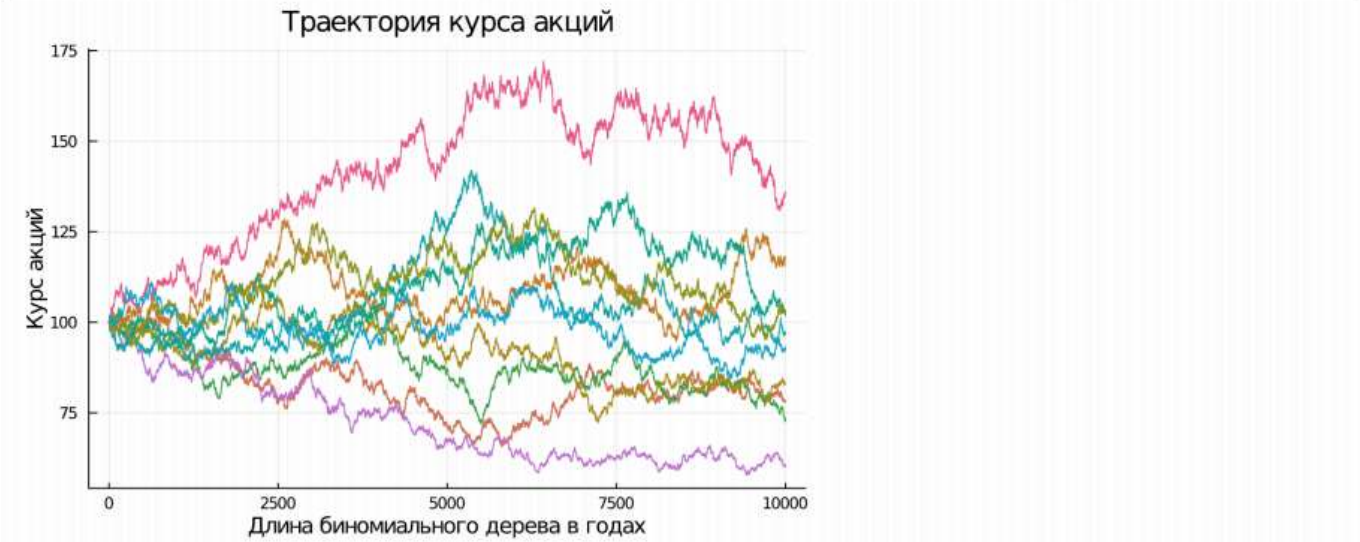
```
function createPath(S::Int64, T::Int64, n::Int64, sigma::Float64, r::Float64)
    # S - начальная цена акции;
    # T - длина биномиального дерева в годах;
    # n - длина одного периода;
    # sigma - волатильность акции;
    # r — годовая процентная ставка;

    h = T / n # длина одного периода;
    u = exp(r*h + sigma * sqrt(h))
    d = exp(r*h - sigma * sqrt(h))
    p = (exp(r*h) - d)/(u - d)
    stockTree = []
    append!(stockTree, S)
    j = 0

    for i in 1:n
        k = rand()
        if k < p
            append!(stockTree, S * (u^(i - j)) * (d ^ j))
        else
            j = j + 1
            append!(stockTree, S * (u^(i - j)) * (d ^ j))
        end
    end
    return stockTree
end
```

Нарисуем 10 траекторий на 1 графике с помощью цикла:

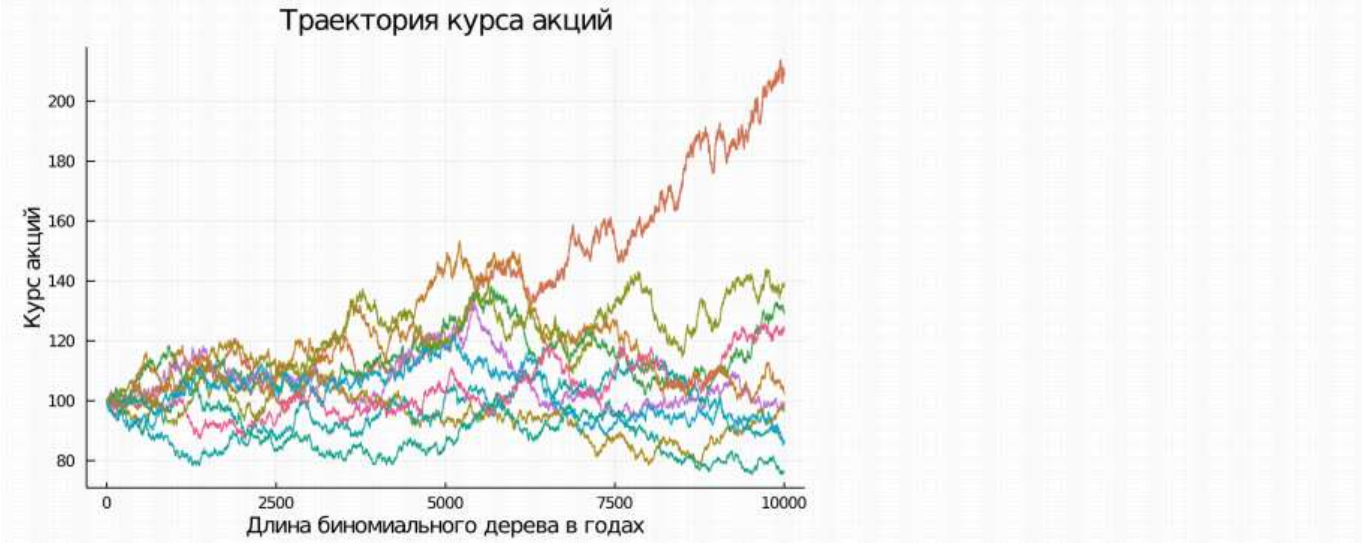
```
for i in 1:10
    IJulia.clear_output(true)
    traj = createPath(100, 1, 10000, 0.3, 0.08)
    if i == 1
        p = plot(traj, title="Траектория курса акций", xlabel="Длина биномиального дерева в годах",
            ylabel="Курс акций", leg=false)
    end
    p = plot!(traj)
    display(p)
end
```



c) Распараллельте генерацию траектории. Можете использовать Threads.@threads, pmap и @parallel.

```
using Base.Threads

Threads.@threads for i in 1:10
    IJulia.clear_output(true)
    traj = createPath(100, 1, 10000, 0.3, 0.08)
    if i == 1
        g = plot(traj, title="Траектория курса акций", xlabel="Длина биномиального дерева в годах",
            ylabel="Курс акций", leg=false)
    end
    g = plot!(traj)
    display(g)
end
```



Вывод

Я познакомилась с пакетами для обработки данных в Julia, а также изучил модель ценообразования биномиальных опционов.