Отчет по лабораторной работе 7

Петрушов Дмитрий, 1032212287

Содержание

Список иллюстраций

Список таблиц

1 Цель работы

Основной целью работы является изучение специализированных пакетов Julia для обработки данных.

2 Выполнение лабораторной работы

2.1 Julia для науки о данных

В Julia для обработки данных используются наработки из других языков программирования, в частности, из R и Python.

2.2 Считывание данных

Перед тем, как начать проводить какие-либо операции над данными, необходимо их откуда-то считать и возможно сохранить в определённой структуре.

Довольно часто данные для обработки содержаться в csv-файле, имеющим текстовый формат, в котором данные в строке разделены, например, запятыми, и соответствуют ячейкам таблицы, а строки данных соответствуют строкам таблицы. Также данные могут быть представлены в виде фреймов или множеств.

В Julia для работы с такого рода структурами данных используют пакеты CSV, DataFrames, RDatasets, FileIO (рис. [2.1]):

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

```
   Resolving package versions...
  No Changes to `~/.julia/environments/v1.10/Project.toml`
  No Changes to `~/.julia/environments/v1.10/Manifest.toml`
Precompiling project...
  ✓ LLVMOpenMP_jll
  1 dependency successfully precompiled in 2 seconds. 530 already precompiled.
   Resolving package versions...
  No Changes to `~/.julia/environments/v1.10/Project.toml`
  No Changes to `~/.julia/environments/v1.10/Manifest.toml`
   Resolving package versions...
  No Changes to `~/.julia/environments/v1.10/Project.toml`
  No Changes to `~/.julia/environments/v1.10/Manifest.toml`
   Resolving package versions...
  No Changes to `~/.julia/environments/v1.10/Project.toml`
  No Changes to `~/.julia/environments/v1.10/Manifest.toml`
```

Рис. 2.1: Установка пакетов

Предположим, что у нас в рабочем каталоге с проектом есть файл с данными programminglanguages.csv, содержащий перечень языков программирования и год их создания. Тогда для заполнения массива данными для последующей обработки требуется считать данные из исходного файла и записать их в соответствующую структуру (рис. [2.2]):

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

[2]: 73×2 DataFrame *48 rows omitted*

Row	year	language
	Int64	String31
1	1951	Regional Assembly Language
2	1952	Autocode
3	1954	IPL
4	1955	FLOW-MATIC
5	1957	FORTRAN
6	1957	COMTRAN
7	1958	LISP
8	1958	ALGOL 58
9	1959	FACT

Рис. 2.2: Считывание данных и запись в структуру

Пример для Julia (рис. [2.3]):

```
# 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
```

Рис. 2.3: Пример

В следующем примере при вызове функции, в качестве аргумента которой указано слово julia, написанное со строчной буквы (рис. [2.4]):

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

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

Closest candidates are:
  getindex(::DataFrame, ::Colon, ::Union{AbstractString, Signed, Symbol, Unsigned})
   @ DataFrames ~/.julia/packages/DataFrames/kcA9R/src/dataframe/dataframe.jl:542
  getindex(::DataFrame, ::typeof(!), ::Union{Signed, Unsigned})
   @ DataFrames ~/.julia/packages/DataFrames/kcA9R/src/dataframe/dataframe.jl:548
  getindex(::DataFrame, ::InvertedIndex, ::Union{AbstractString, Signed, Symbol, Unsigned})
   @ DataFrames ~/.julia/packages/DataFrames/kcA9R/src/dataframe/dataframe.jl:538
  ...
```

Рис. 2.4: Поиск “julia” со строчной буквы

Для того, чтобы убрать в функции зависимость данных от регистра, необходимо изменить исходную функцию следующим образом (рис. [2.5]):

```
# 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
```

Рис. 2.5: Изменение исходной функции

Можно считывать данные построчно, с элементами, разделенными заданным разделителем (рис. [2.6]):

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

```
74×2 Matrix{Any}:
     "year"  "language"
 1951        "Regional Assembly Language"
 1952        "Autocode"
 1954        "IPL"
 1955        "FLOW-MATIC"
 1957        "FORTRAN"
 1957        "COMTRAN"
 1958        "LISP"
 1958        "ALGOL 58"
 1959        "FACT"
 1959        "COBOL"
 1959        "RPG"
 1962        "APL"
    ⋮
```

Рис. 2.6: Построчное считывание данных

2.3 Запись данных в файл

Предположим, что требуется записать имеющиеся данные в файл. Для записи данных в формате CSV можно воспользоваться следующим вызовом (рис. [2.7]):

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

```
"programming_languages_data2.csv"
```

Рис. 2.7: Запись данных в файл

Можно задать тип файла и разделитель данных (рис. [2.8]):

```
# 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, '-')
```

Рис. 2.8: Пример с указанием типа данных и разделителем данных

Можно проверить, используя readdlm, корректность считывания созданного текстового файла (рис. [2.9]):

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

```
74×2 Matrix{Any}:
     "year"  "language"
 1951        "Regional Assembly Language"
 1952        "Autocode"
 1954        "IPL"
 1955        "FLOW-MATIC"
 1957        "FORTRAN"
 1957        "COMTRAN"
 1958        "LISP"
 1958        "ALGOL 58"
 1959        "FACT"
 1959        "COBOL"
 1959        "RPG"
 1962        "APL"
    ⋮
```

Рис. 2.9: Проверка корректности считывания созданного текстового файла

2.4 Словари

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

Предположим, что словарь должен содержать перечень всех языков программирования и года их создания, при этом при указании года выводить все языки программирования, созданные в этом году.

При инициализации словаря можно задать конкретные типы данных для ключей и значений (рис. [2.10]):

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

```
Dict{Integer, Vector{String}}()
```

Рис. 2.10: Инициализация словаря

Можно инициировать пустой словарь, не задавая строго структуру (рис. [2.11]):

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

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

Рис. 2.11: Инициализация пустого словаря

Далее требуется заполнить словарь ключами и годами, которые содержат все языки программирования, созданные в каждом году, в качестве значений (рис. [2.12]):

```
# 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
```

Рис. 2.12: Заполнение словаря данными

В результате при вызове словаря можно, выбрав любой год, узнать, какие языки программирования были созданы в этом году (рис. [2.13]):

```
dict[2003]
```

```
2-element Vector{String}:
 "Groovy"
 "Scala"
```

Рис. 2.13: Пример работы словаря

2.5 DataFrames

Работа с данными, записанными в структуре DataFrame, позволяет использовать индексацию и получить доступ к столбцам по заданному имени заголовка или по индексу столбца.

На примере с данными о языках программирования и годах их создания зададим структуру DataFrame (рис. [2.14]):

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

73×2 DataFrame

Row	year	language
	Int64	String31
1	1951	Regional Assembly Language
2	1952	Autocode
3	1954	IPL
4	1955	FLOW-MATIC
5	1957	FORTRAN
6	1957	COMTRAN
7	1958	LISP
8	1958	ALGOL 58
9	1959	FACT
10	1959	COBOL
11	1959	RPG
12	1962	APL
13	1962	Simula
⋮	⋮	⋮

Рис. 2.14: Пример создания структуры DataFrame

2.6 RDatasets

С данными можно работать также как с наборами данных через пакет RDatasets языка R (рис. [2.15]):

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

150×5 DataFrame

Row	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
5	5.0	3.6	1.4	0.2	setosa
6	5.4	3.9	1.7	0.4	setosa
7	4.6	3.4	1.4	0.3	setosa
8	5.0	3.4	1.5	0.2	setosa
9	4.4	2.9	1.4	0.2	setosa
10	4.9	3.1	1.5	0.1	setosa
11	5.4	3.7	1.5	0.2	setosa
12	4.8	3.4	1.6	0.2	setosa
13	4.8	3.0	1.4	0.1	setosa

Рис. 2.15: Работа с пакетом RDatasets

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

(рис. [2.16]):

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

```
DataFrame
```

```
describe(iris)
```

5×7 DataFrame

Row	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}

Рис. 2.16: Получение основных статических сведений о каждом столбце в наборе данных

2.7 Работа с переменными отсутствующего типа (Missing Values)

Пакет DataFrames позволяет использовать так называемый «отсутствующий» тип (рис. [2.17]):

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

```
Missing
```

Рис. 2.17: Использование “отсутствующего” типа

В операции сложения числа и переменной с отсутствующим типом значение также будет иметь отсутствующий тип (рис. [2.18]):

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

```
missing
```

Рис. 2.18: Операция сложения числа и переменной с отсутствующим типом

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

Предположим есть перечень продуктов, для которых заданы калории. В массиве значений калорий есть значение с отсутствующим типом (рис. [2.19]):

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

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

Рис. 2.19: Пример работы с данными, среди которых есть данные с отсутствующим типом

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

Для решения этой проблемы необходимо игнорировать отсутствующий тип (рис. [2.20]):

```
# Подключаем пакет Statistics:
using Statistics
# Определение среднего значения:
mean(calories)
```

```
missing
```

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

```
58.0
```

Рис. 2.20: Игнорирование отсутствующего типа

Далее показано, как можно сформировать таблицы данных и объединить их в один фрейм (рис. [2.21]):

```
# 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)
```

4×3 DataFrame

Row	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

Рис. 2.21: Формирование таблиц данных и их объединение в один фрейм

2.8 FileIO

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

Попробуем посмотреть, как Julia работает с изображениями.

Подключим соответствующий пакет (рис. [2.22]):

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

```
   Resolving package versions...
  No Changes to `~/.julia/environments/v1.10/Project.toml`
  No Changes to `~/.julia/environments/v1.10/Manifest.toml`
```

Рис. 2.22: Подключение пакетов

Загрузим изображение (в данном случае логотип Julia) (рис. [2.23]):

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

```
460×460 Array{RGBA{N0f8},2} with eltype ColorTypes.RGBA{FixedPointNumbers.N0f8}
 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)
 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)
 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)
 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)
```

Рис. 2.23: Загрузка изображения

Julia хранит изображение в виде множества цветов (рис. [2.24]):

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

```
typeof(X1) = Matrix{ColorTypes.RGBA{FixedPointNumbers.N0f8}}
size(X1) = (460, 460)
```

Рис. 2.24: Определение типа и размера данных

2.9 Обработка данных: стандартные алгоритмы машинного обучения в Julia. Кластеризация данных. Метод k-средних

Задача кластеризации данных заключается в формировании однородной группы упорядоченных по какому-то признаку данных.

Метод k-средних позволяет минимизировать суммарное квадратичное отклонение точек кластеров от центров этих кластеров.

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 (рис. [2.25]):

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

```
  Resolving package versions...
 No Changes to `~/.julia/environments/v1.10/Project.toml`
 No Changes to `~/.julia/environments/v1.10/Manifest.toml`
  Resolving package versions...
 No Changes to `~/.julia/environments/v1.10/Project.toml`
 No Changes to `~/.julia/environments/v1.10/Manifest.toml`
  Resolving package versions...
 No Changes to `~/.julia/environments/v1.10/Project.toml`
 No Changes to `~/.julia/environments/v1.10/Manifest.toml`
```

Рис. 2.25: Подключение нужных пакетов

Затем загрузим данные (рис. [2.26]):

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

985×12 DataFrame *960 rows omitted*

Row	street	city	zip	state	beds	baths	sqft	type	sale_date	price	latit
	String	String15	Int64	String3	Int64	Int64	Int64	String15	String31	Int64	Floa
1	3526 HIGH ST	SACRAMENTO	95838	CA	2	1	836	Residential	Wed May 21 00:00:00 EDT 2008	59222	38.6
2	51 OMAHA CT	SACRAMENTO	95823	CA	3	1	1167	Residential	Wed May 21 00:00:00 EDT 2008	68212	38.4
3	2796 BRANCH ST	SACRAMENTO	95815	CA	2	1	796	Residential	Wed May 21 00:00:00 EDT 2008	68880	38.6

Рис. 2.26: Загрузка данных

Построим график цен на недвижимость в зависимости от площади (рис. [2.27]):

```
# Построение графика:
using Plots
plot(size=(500,500),leg=false)
x = houses[!,:sqft]
y = houses[!,:price]
scatter(x,y,markersize=3)
```

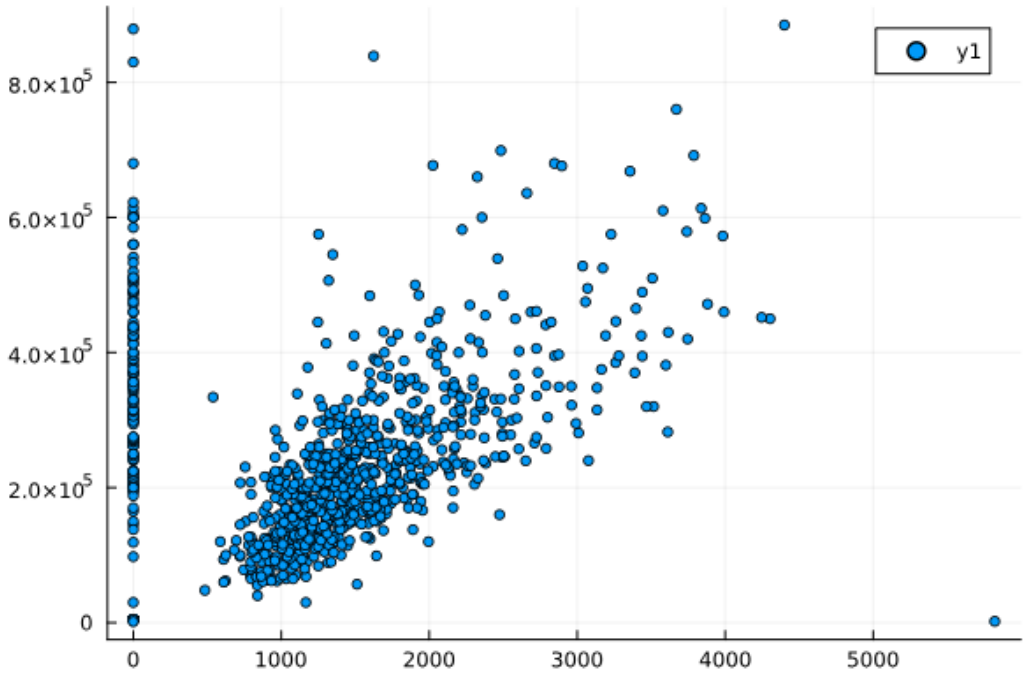

Рис. 2.27: Построение графика цен на недвижимость в зависимости от площади

Для того чтобы избавиться от “артефактов”, можно отфильтровать и исключить такие значения, получить более корректный график цен (рис. [2.28]):

```
# Фильтрация данных по заданному условию:
filter_houses = houses[houses[!,:sqft].>0,:]
# Построение графика:
x = filter_houses[!,:sqft]
y = filter_houses[!,:price]
scatter(x,y)
```

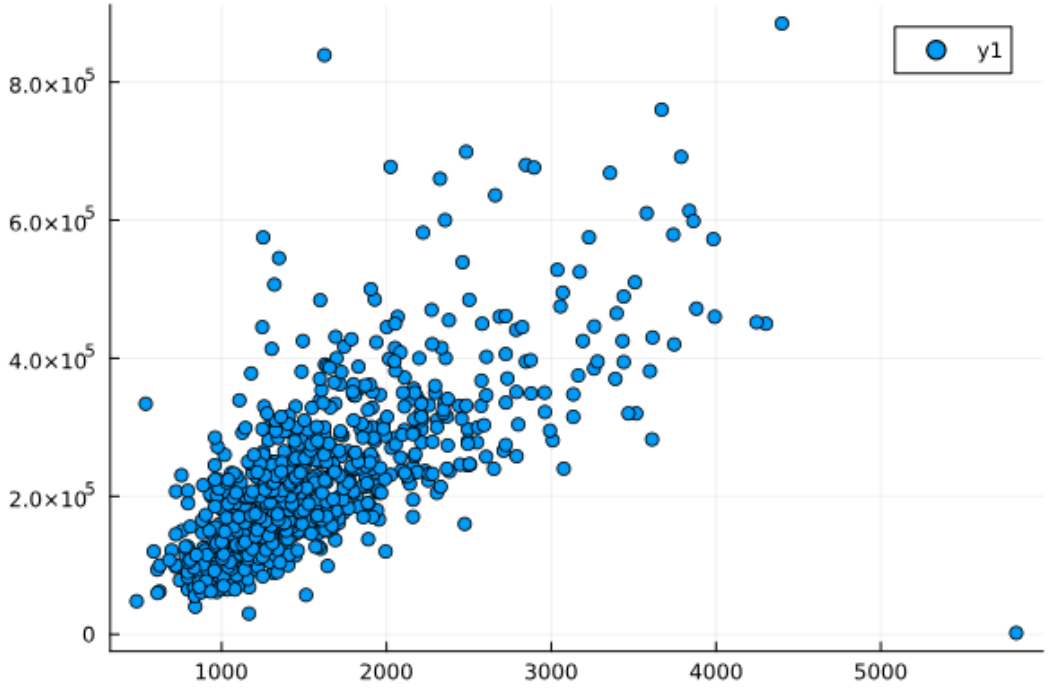

Рис. 2.28: Построение графика без “артефактов”

Построим график, обозначив каждый кластер отдельным цветом (рис. [2.29]):

```
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)
```

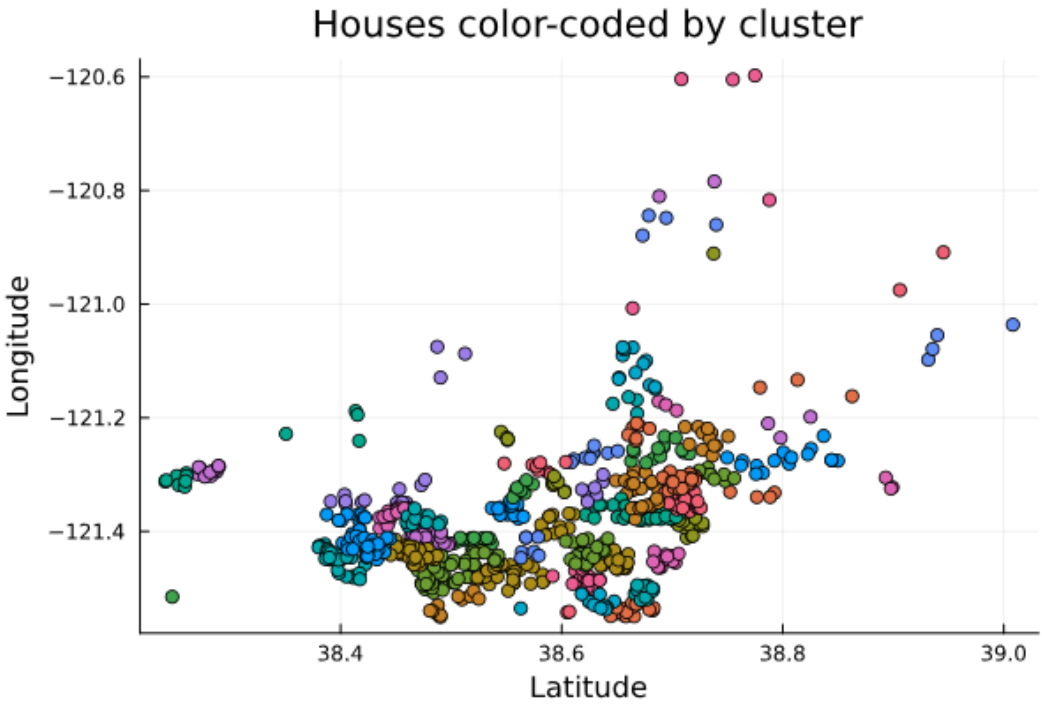

Рис. 2.29: Построение графика с кластерами разных цветов

Построим график, раскрасив кластеры по почтовому индексу (рис. [2.30]):

```
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)
```

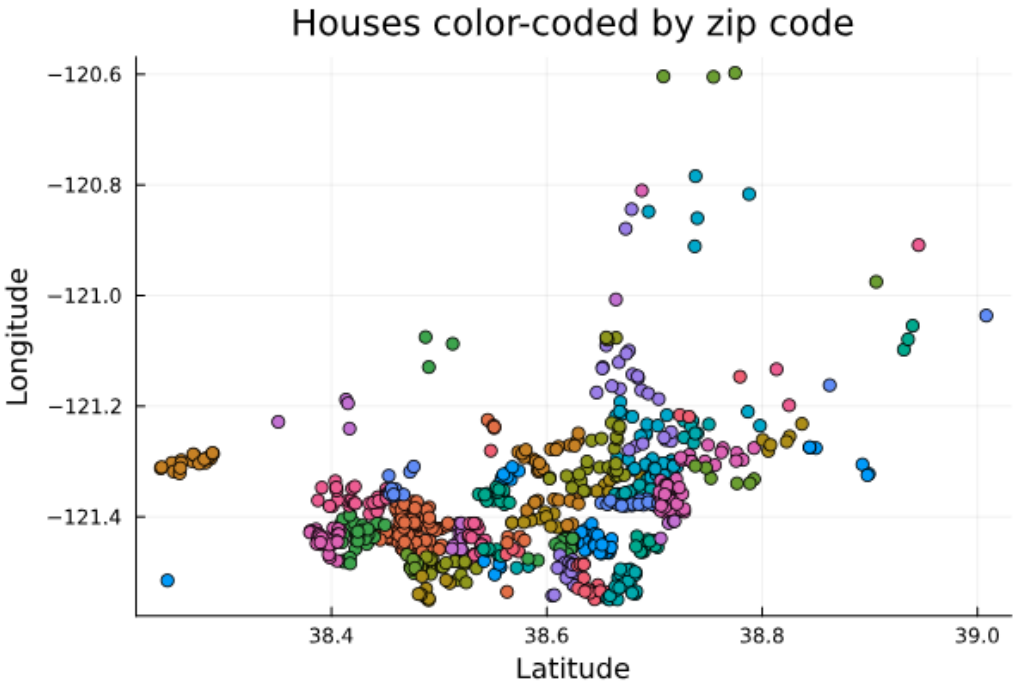

Рис. 2.30: Построение графика с кластерами разных цветов по почтовому индексу

2.10 Кластеризация данных. Метод k ближайших соседей

Отобразим на графике соседей выбранного объекта недвижимости (рис. [2.31]):

```
# 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)
```

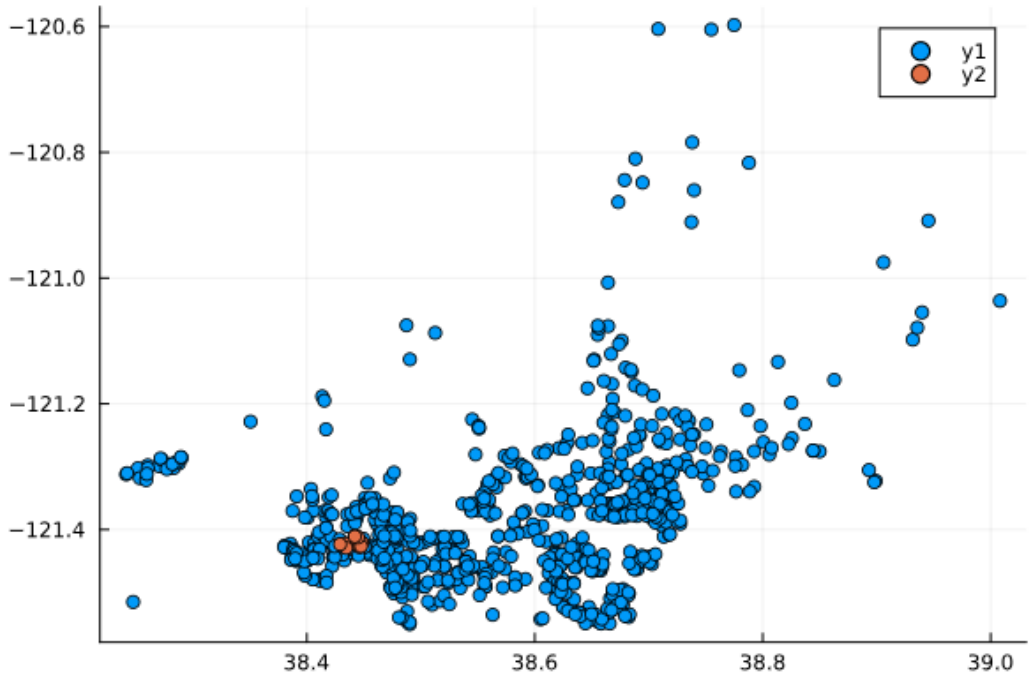

Рис. 2.31: Отображение на графике соседей выбранного объекта недвижимости

Используя индексы idxs и функцию :city для индексации в DataFrame filter_houses, можно определить районы соседних домов (рис. [2.32]):

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

```
10-element PooledArrays.PooledVector{String15, UInt32, Vector{UInt32}}:
 "SACRAMENTO"
 "ELK GROVE"
 "SACRAMENTO"
 "SACRAMENTO"
 "SACRAMENTO"
 "SACRAMENTO"
 "ELK GROVE"
 "ELK GROVE"
 "ELK GROVE"
 "ELK GROVE"
```

Рис. 2.32: Определение районов соседних домов

2.11 Обработка данных. Метод главных компонент

Метод главных компонент (Principal Components Analysis, PCA) позволяет уменьшить размерность данных, потеряв наименьшее количество полезной информации. Метод имеет широкое применение в различных областях знаний, например, при визуализации данных, компрессии изображений, в эконометрике, некоторых гуманитарных предметных областях, например, в социологии или в политологии.

На примере с данными о недвижимости попробуем уменьшить размеры данных о цене и площади из набора данных домов (рис. [2.33]):

```
y= MultivariateStats.transform(M,F)
# Выделение значений главных компонент в отдельную переменную:
Xr = reconstruct(M, y)
# Построение графика с выделением главных компонент:
scatter(F[1,:],F[2,:])
scatter!(Xr[1,:],Xr[2,:])
```

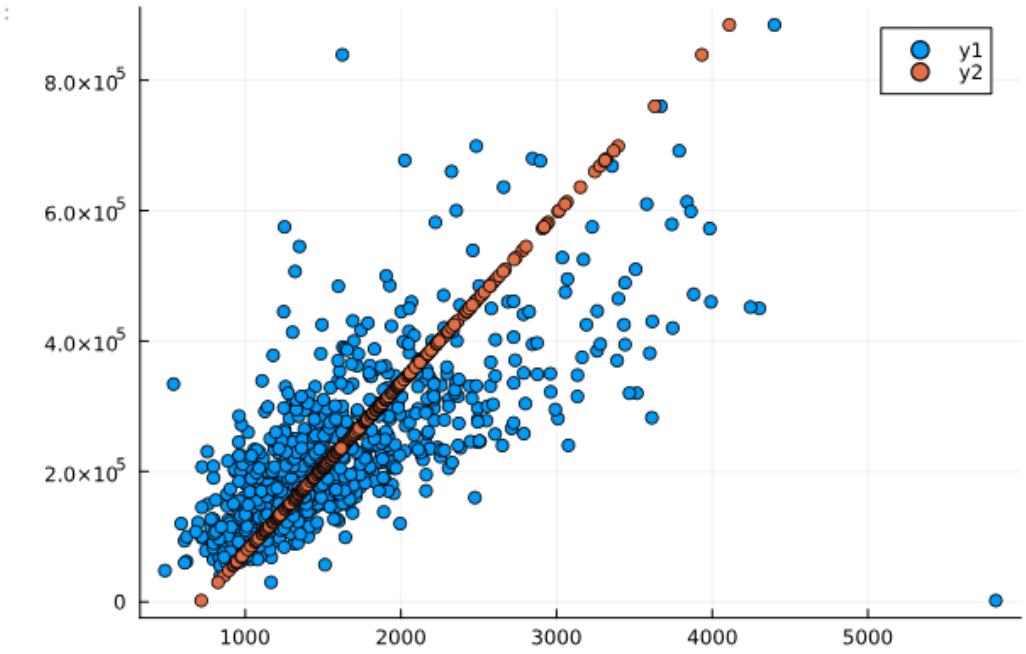

Рис. 2.33: Попытка уменьшения размера данных о цене и площади из набора данных домов

2.12 Обработка данных. Линейная регрессия

Регрессионный анализ представляет собой набор статистических методов исследования влияния одной или нескольких независимых переменных (регрессоров) на зависимую (критериальная) переменную. Терминология зависимых и независимых переменных отражает лишь математическую зависимость переменных, а не причинноследственные отношения.

Наиболее распространённый вид регрессионного анализа — линейная регрессия, когда находят линейную функцию, которая согласно определённым мате-

матическим критериям наиболее соответствует данным.

Зададим случайный набор данных (можно использовать и полученные экспериментальным путём какие-то данные). Попробуем найти для данных лучшее соответствие (рис. [2.34]):

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

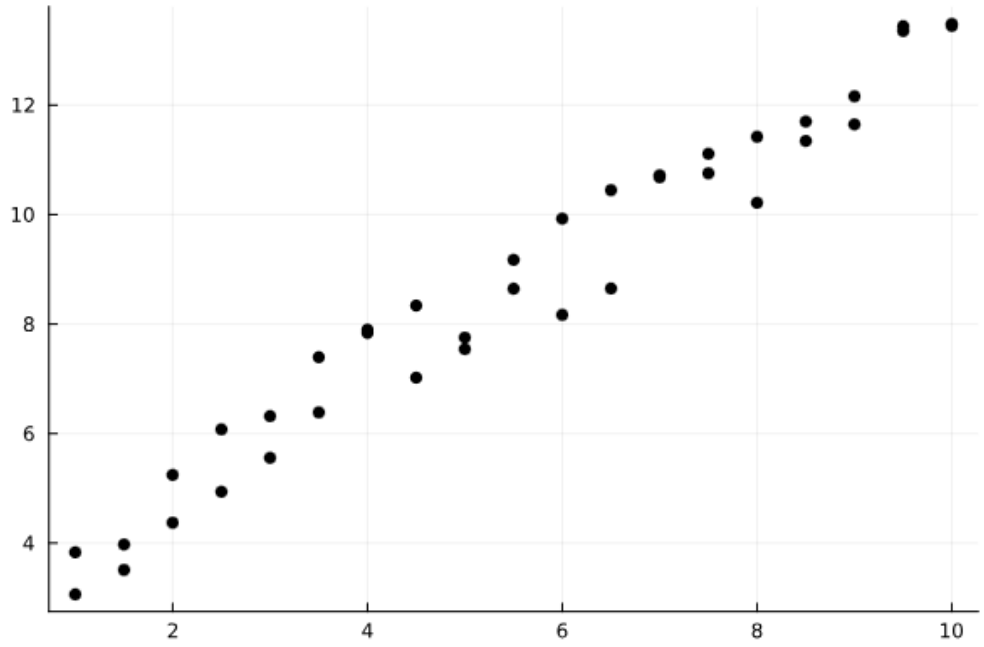

Рис. 2.34: Исходные данные

Определим функцию линейной регрессии. Применим функцию линейной регрессии для построения соответствующего графика значений (рис. [2.35]):

```
plot!(xvals, ynew)
```

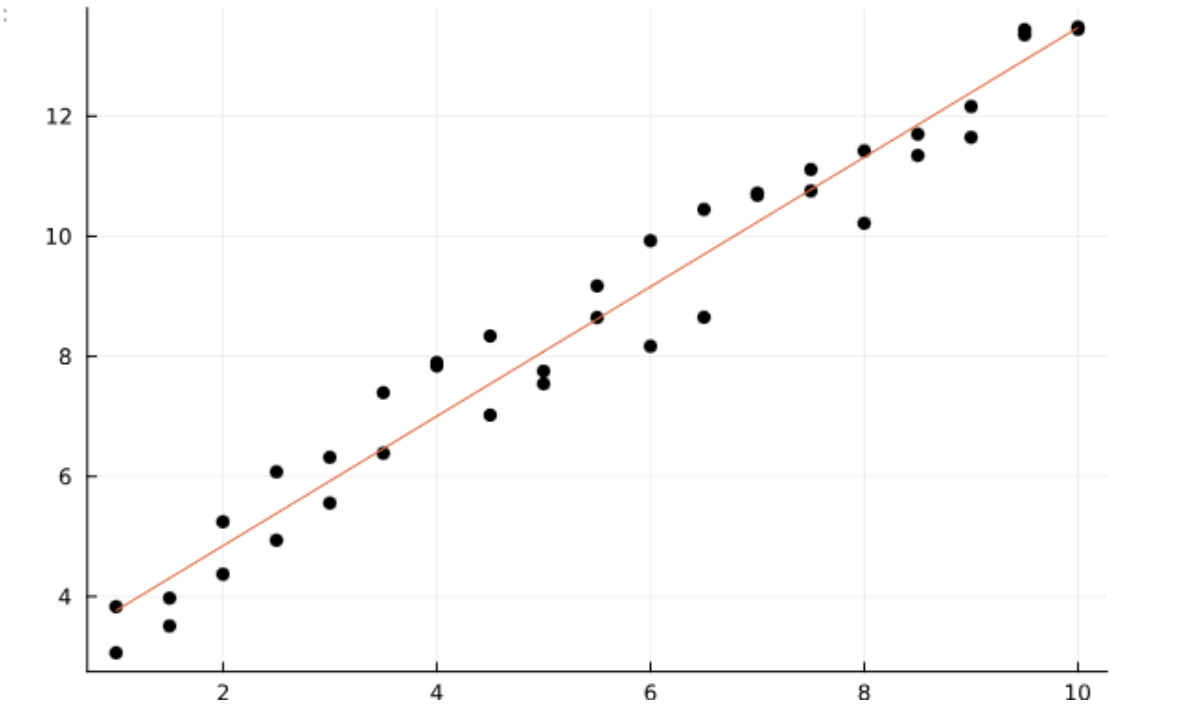

Рис. 2.35: Применение функции для построения графика

Сгенерируем больший набор данных. Определим, сколько времени потребуется, чтобы найти соответствие этим данным. Для сравнения реализуем подобный код на языке Python. Используем пакет для анализа производительности, чтобы провести сравнение (рис. [2.36]):

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

```
  0.064046 seconds (74.75 k allocations: 5.179 MiB, 91.19% compilation time)
```

```
(0.9999999961460347, 3.000038831247366)
```

```
import Pkg
Pkg.add("BenchmarkTools")
using BenchmarkTools
@btime a,b = find_best_fit_python(xvals,yvals)
@btime a,b = find_best_fit(xvals,yvals)
```

```
  2.810 ms (25 allocations: 832 bytes)
  429.597 μs (1 allocation: 32 bytes)
   Resolving package versions...
  No Changes to `~/.julia/environments/v1.10/Project.toml`
  No Changes to `~/.julia/environments/v1.10/Manifest.toml`
```

```
(0.9999999961460344, 3.000038831269194)
```

Рис. 2.36: Сравнение

2.13 Самостоятельное выполнение

Выполнение задания №1 (рис. [2.37] - рис. [2.39]):

```
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

Рис. 2.37: Решение задания №1

```
# 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)
```

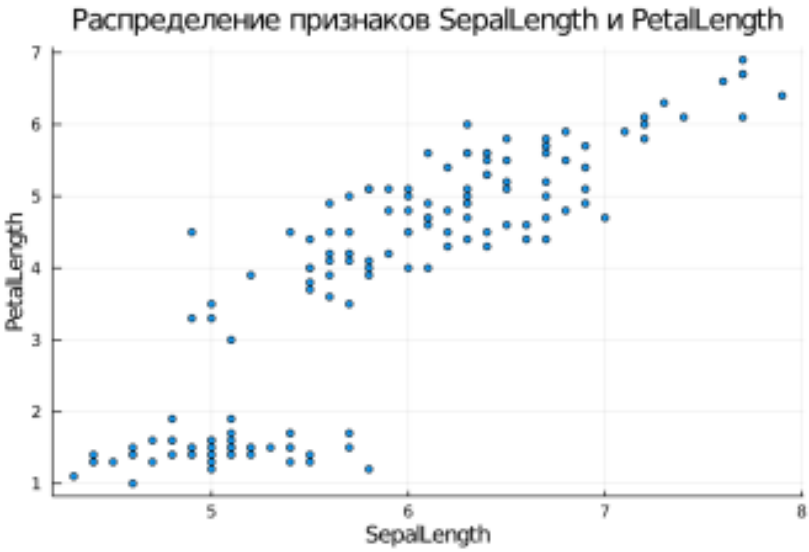

Рис. 2.38: Решение задания №1

```
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)
```

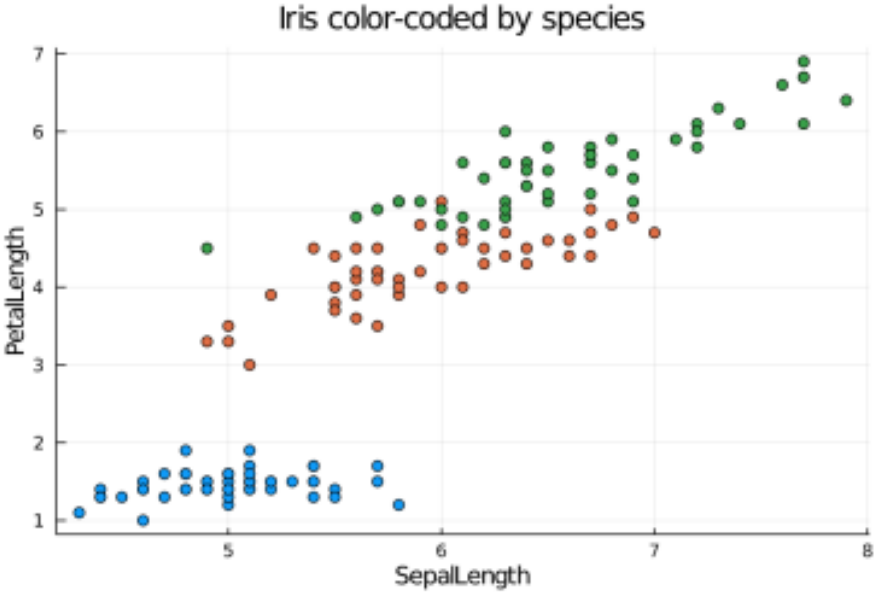

Рис. 2.39: Решение задания №1

Выполнение задания №2 (рис. [2.40] - рис. [2.41]):

```
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]
```

Рис. 2.40: Решение задания №2

```
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

Рис. 2.41: Решение задания №2

Выполнение задания №2-2 (рис. [2.42] - рис. [2.45]):

```
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)
```

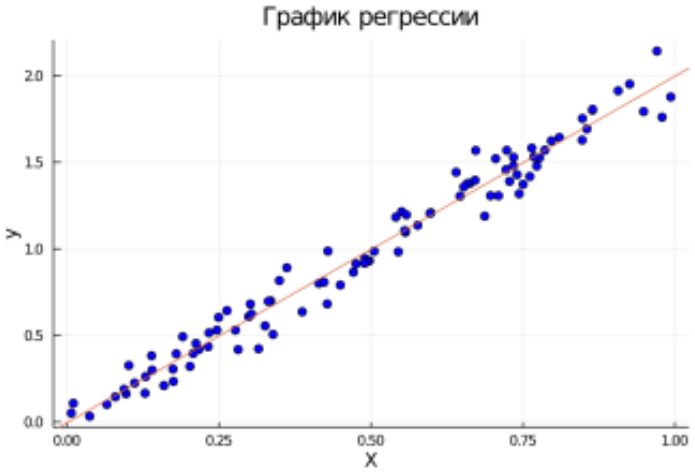

Рис. 2.42: Решение задания №2-2

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

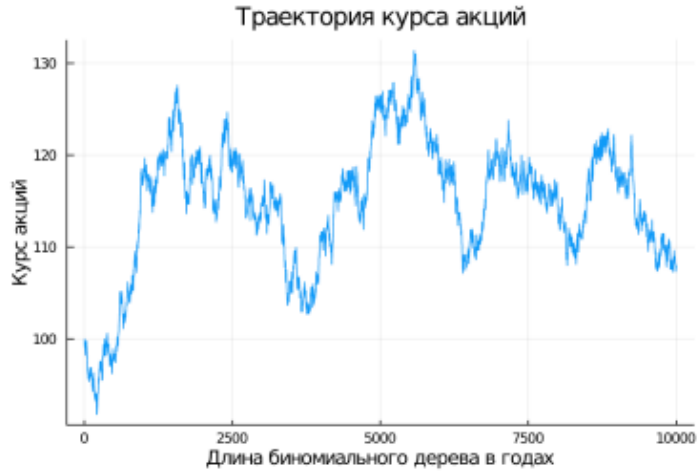

Рис. 2.43: Решение задания №2-2а

```julia
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
```

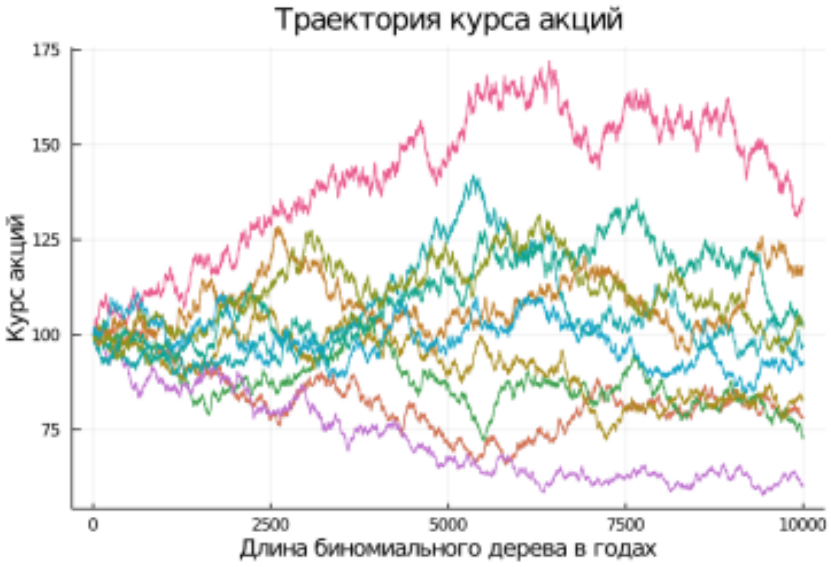

Рис. 2.44: Решение задания №2-2b

```julia
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
```

Рис. 2.45: Решение задания №2-2c

3 Вывод

В ходе выполнения лабораторной работы были изучены специализированные пакеты Julia для обработки данных.

Список литературы