Отчёт по лабораторной работе №7 Компьютерный практикум по статистическому анализу данных

Введение в работу с данными

Выполнил: Махорин Иван Сергеевич,
НПИбд-02-21, 1032211221

Содержание

Список иллюстраций

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. Julia для науки о данных

1.1. Считывание данных

[2]:
```
# 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
```

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):

[5]:
```
# Считывание данных и их запись в структуру:
P = CSV.File("programminglanguages.csv") |> DataFrame
# 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
# Пример вызова функции и определение даты создания языка Python:
language_created_year(P,"Python")
```

[5]:
```
1991
```

Рис. 2.2: Считывание данных и запись в структуру

Пример для Julia (рис. 2.3):

```
[62]: # Пример вызова функции и определение даты создания языка Julia:
      language_created_year(P,"Julia")

[62]: 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):

```
[63]: language_created_year(P,"julia")
```

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):

```
[8]: # 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
     # Пример вызова функции и определение даты создания языка julia:
     language_created_year_v2(P,"julia")

[8]: 2012
```

Рис. 2.5: Изменение исходной функции

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

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

```
[9]: 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"
        ⋮
     2003        "Scala"
     2005        "F#"
     2006        "PowerShell"
     2007        "Clojure"
     2009        "Go"
     2010        "Rust"
     2011        "Dart"
     2011        "Kotlin"
     2011        "Red"
     2011        "Elixir"
     2012        "Julia"
     2014        "Swift"
```

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):

1.2. Запись данных в файл

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

```
[10]: "programming_languages_data2.csv"
```

Рис. 2.7: Запись данных в файл

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

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

```
[12]: # 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):

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

```
[13]: 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"
      ⋮
      2003    "Scala"
      2005    "F#"
      2006    "PowerShell"
      2007    "Clojure"
      2009    "Go"
      2010    "Rust"
      2011    "Dart"
      2011    "Kotlin"
      2011    "Red"
      2011    "Elixir"
      2012    "Julia"
      2014    "Swift"
```

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 клю-

чей и значений (рис. 2.10):

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

[14]: Dict{Integer, Vector{String}}()
```

Рис. 2.10: Инициализация словаря

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

```
[15]: # Инициализация словаря:
      dict2 = Dict()

[15]: 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):

```
[16]: # 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):

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

```
[17]: 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):

1.4. DataFrames

```
[18]: # Подгружаем пакет DataFrames:
using DataFrames
```

```
[20]: # Задаём переменную со структурой DataFrame:
df = DataFrame(year = P[:,1], language = P[:,2])
# Вывод всех значения столбца year:
df[!,:year]
# Получение статистических сведений о фрейме:
describe(df)
```

[20]: 2×7 DataFrame

Row	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	String31

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):

1.5. RDatasets

```
[21]: # Подгружаем пакет RDatasets:
      using RDatasets

[22]: # Задаём структуру данных в виде набора данных:
      iris = dataset("datasets", "iris")
      # Определения типа переменной:
      typeof(iris)

[22]: DataFrame
```

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):

```
[23]: describe(iris)
```

[23]: 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):

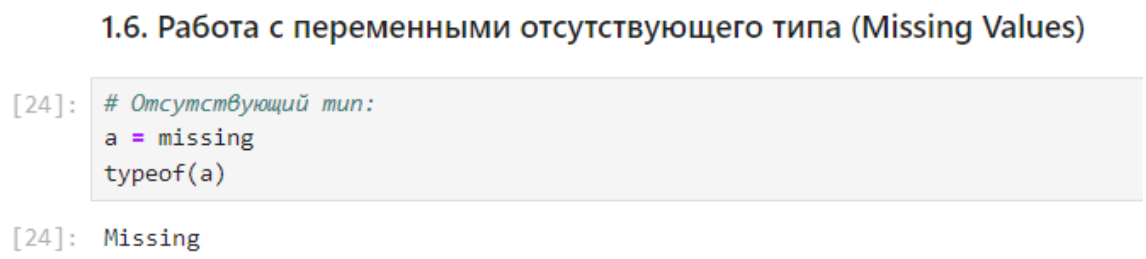

Рис. 2.17: Использование “отсутствующего” типа

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

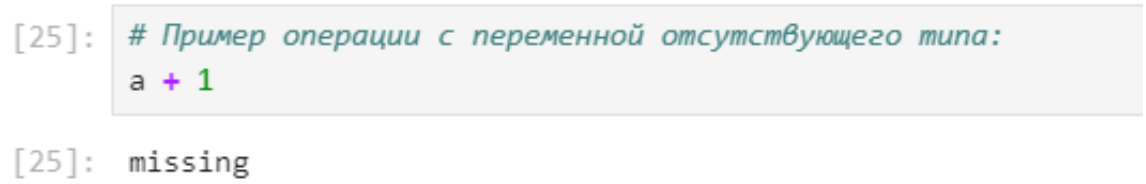

Рис. 2.18: Операция сложения числа и переменной с отсутствующим типом

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):

```
[26]: # Определение перечня продуктов:
      foods = ["apple", "cucumber", "tomato", "banana"]
      # Определение калорий:
      calories = [missing,47,22,105]
      # Определение типа переменной:
      typeof(calories)

[26]: Vector{Union{Missing, Int64}} (alias for Array{Union{Missing, Int64}, 1})
```

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):

```
[27]: # Подключаем пакет Statistics:
      using Statistics
      # Определение среднего значения:
      mean(calories)
      # Определение среднего значения без значений с отсутствующим типом:
      mean(skipmissing(calories))

[27]: 58.0
```

Рис. 2.20: Игнорирование отсутствующего типа

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

[29]:
```
# 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)
```

[29]: 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):

1.7. FileIO

[30]:
```
# Подключаем пакет FileIO:
using FileIO
```

[31]:
```
# Подключаем пакет ImageIO:
import Pkg
Pkg.add("ImageIO")
```

Рис. 2.22: Подключение пакетов

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

```
[35]: X1 = load("julialogo.png")
      display(X1)
```

Рис. 2.23: Загрузка изображения

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

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

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):

2. Обработка данных: стандартные алгоритмы машинного обученияв Julia

2.1. Кластеризация данных. Метод k-средних

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

Рис. 2.25: Подключение нужных пакетов

Затем загрузим данные (рис. 2.26):

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

[38]: 985×12 DataFrame — *960 rows omitted*

Row	street	city	zip	state	beds	baths	sq__ft	type	sale_date	price	latitude	longitude
	String	String15	Int64	String3	Int64	Int64	Int64	String15	String31	Int64	Float64	Float64
1	3526 HIGH ST	SACRAMENTO	95838	CA	2	1	836	Residential	Wed May 21 00:00:00 EDT 2008	59222	38.6319	-121.435
2	51 OMAHA CT	SACRAMENTO	95823	CA	3	1	1167	Residential	Wed May 21 00:00:00 EDT 2008	68212	38.4789	-121.431
3	2796 BRANCH ST	SACRAMENTO	95815	CA	2	1	796	Residential	Wed May 21 00:00:00 EDT 2008	68880	38.6183	-121.444
4	2805 JANETTE WAY	SACRAMENTO	95815	CA	2	1	852	Residential	Wed May 21 00:00:00 EDT 2008	69307	38.6168	-121.439
5	6001 MCMAHON DR	SACRAMENTO	95824	CA	2	1	797	Residential	Wed May 21 00:00:00 EDT 2008	81900	38.5195	-121.436
6	5828 PEPPERMILL CT	SACRAMENTO	95841	CA	3	1	1122	Condo	Wed May 21 00:00:00 EDT 2008	89921	38.6626	-121.328
7	6048 OGDEN NASH WAY	SACRAMENTO	95842	CA	3	2	1104	Residential	Wed May 21 00:00:00 EDT 2008	90895	38.6817	-121.352
8	2561 19TH AVE	SACRAMENTO	95820	CA	3	1	1177	Residential	Wed May 21 00:00:00 EDT 2008	91002	38.5351	-121.481
9	11150 TRINITY RIVER DR Unit 114	RANCHO CORDOVA	95670	CA	2	2	941	Condo	Wed May 21 00:00:00 EDT 2008	94905	38.6212	-121.271
10	7325 10TH ST	RIO LINDA	95673	CA	3	2	1146	Residential	Wed May 21 00:00:00 EDT 2008	98937	38.7009	-121.443
11	645 MORRISON AVE	SACRAMENTO	95838	CA	3	2	909	Residential	Wed May 21 00:00:00 EDT 2008	100309	38.6377	-121.452
12	4085 FAWN CIR	SACRAMENTO	95823	CA	3	2	1289	Residential	Wed May 21 00:00:00 EDT 2008	106250	38.4707	-121.459
13	2930 LA ROSA RD	SACRAMENTO	95815	CA	1	1	871	Residential	Wed May 21 00:00:00 EDT 2008	106852	38.6187	-121.436

Рис. 2.26: Загрузка данных

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

[4]:
```
using CSV
using DataFrames
using Plots
# Загрузка данных из CSV
houses = CSV.read("houses.csv", DataFrame)
# Построение графика
plot(size=(500, 500), leg=false)
x = houses[:, :sq__ft]  # Столбец площади
y = houses[:, :price]   # Столбец цены
scatter(x, y, markersize=3)
```

[4]:

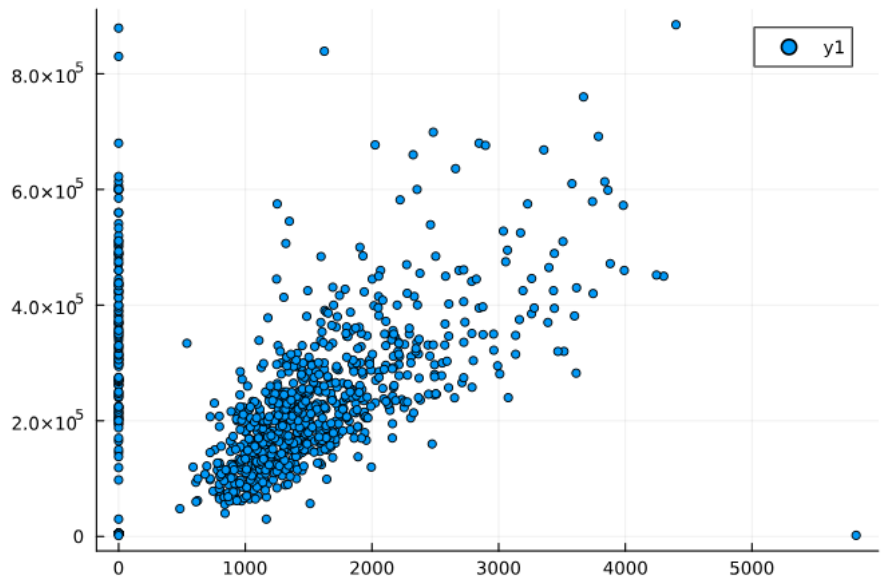

Рис. 2.27: Построение графика цен на недвижимость в зависимости от площади

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

[5]:
```
# Фильтрация данных по заданному условию:
filter_houses = houses[houses[!,:sq__ft].>0,:]
# Построение графика:
x = filter_houses[!,:sq__ft]
y = filter_houses[!,:price]
scatter(x,y)
```

[5]:

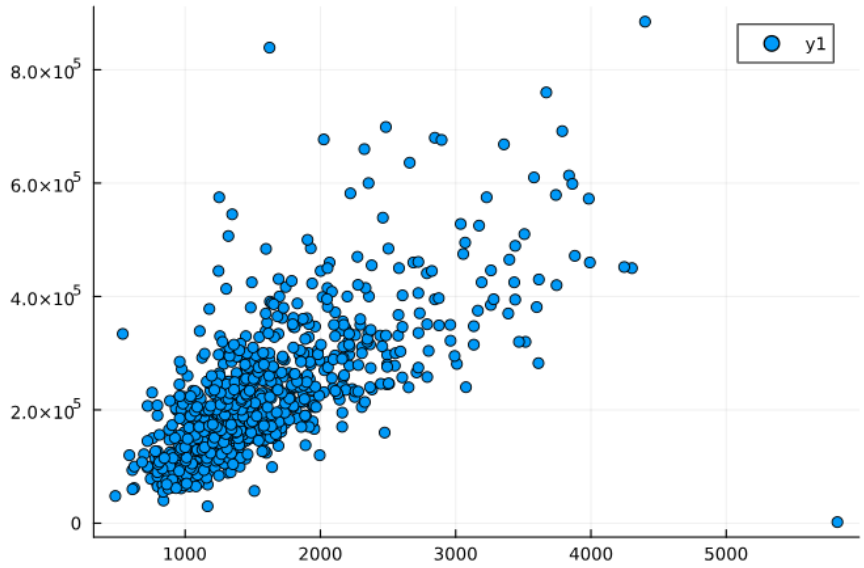

Рис. 2.28: Построение графика без “артефактов”

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

Средние цены для каждого типа домов:

4×2 DataFrame

Row	type	mean_price
	String15	Float64
1	Residential	2.39186e5
2	Condo	1.50082e5
3	Multi-Family	2.24535e5
4	Unkown	275000.0

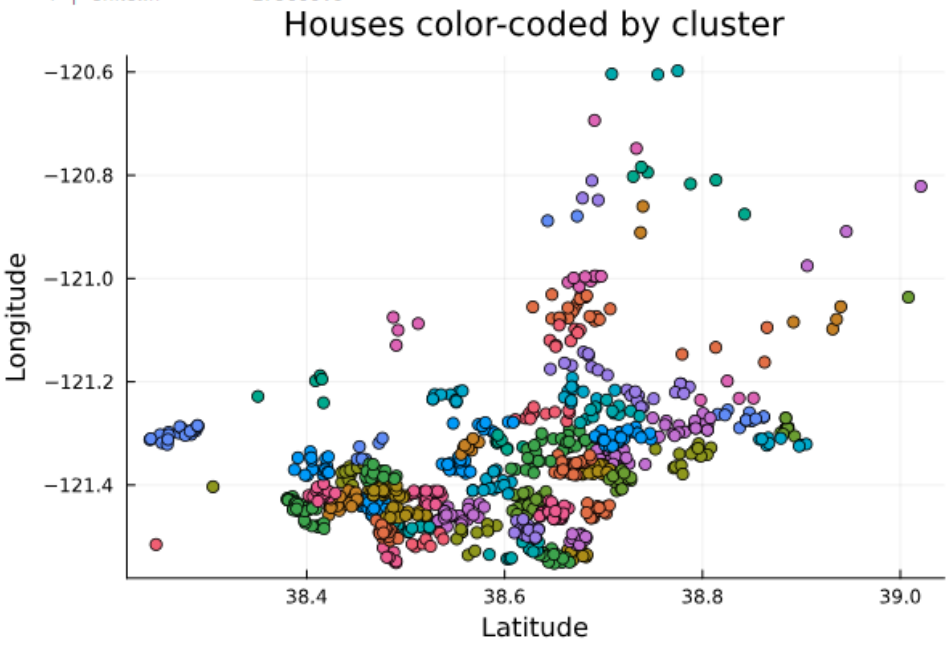

Рис. 2.29: Построение графика с кластерами разных цветов

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

```
[15]: 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)
```

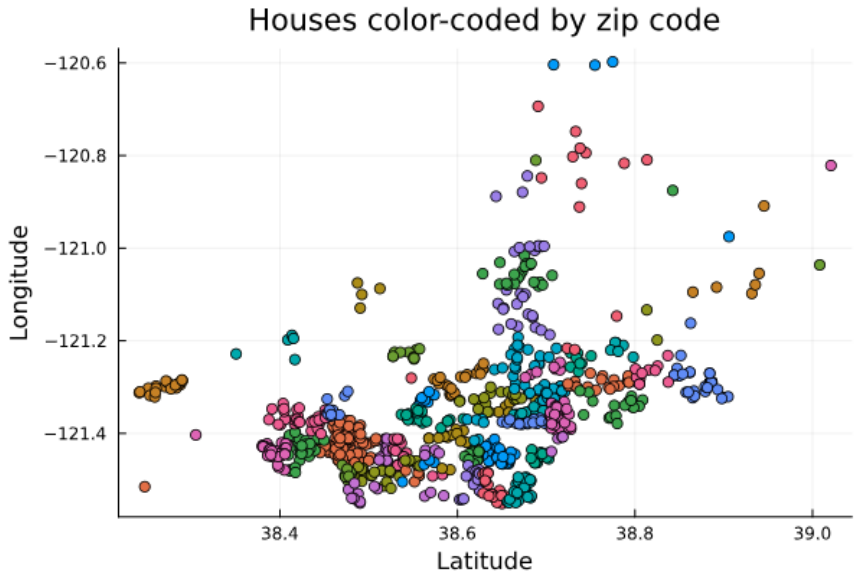

Рис. 2.30: Построение графика с кластерами разных цветов по почтовому индексу

2.10 Кластеризация данных. Метод k ближайших соседей

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

2.2. Кластеризация данных. Метод k ближайших соседей

[16]:
```
# Подключение пакета NearestNeighbors:
import Pkg
Pkg.add("NearestNeighbors")
using NearestNeighbors
knearest = 10
id = 70
point = X[:,id]
# Поиск ближайших соседей:
kdtree = KDTree(X)
idxs, dists = knn(kdtree, point, knearest, true)
# 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)
```

```
  Resolving package versions...
   Updating `C:\Users\Ivan\.julia\environments\v1.10\Project.toml`
 [b8a86587] + NearestNeighbors v0.4.21
 No Changes to `C:\Users\Ivan\.julia\environments\v1.10\Manifest.toml`
```

[16]:

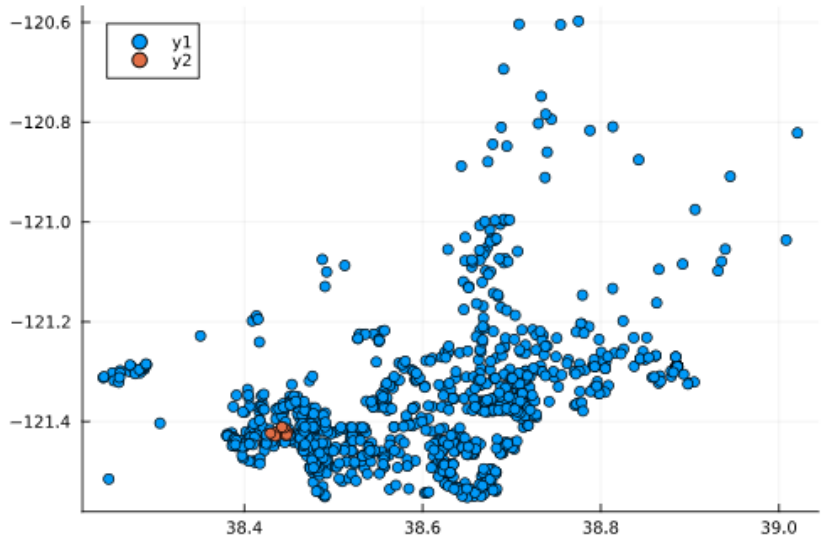

Рис. 2.31: Отображение на графике соседей выбранного объекта недвижимости

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

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

```
[17]: 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):

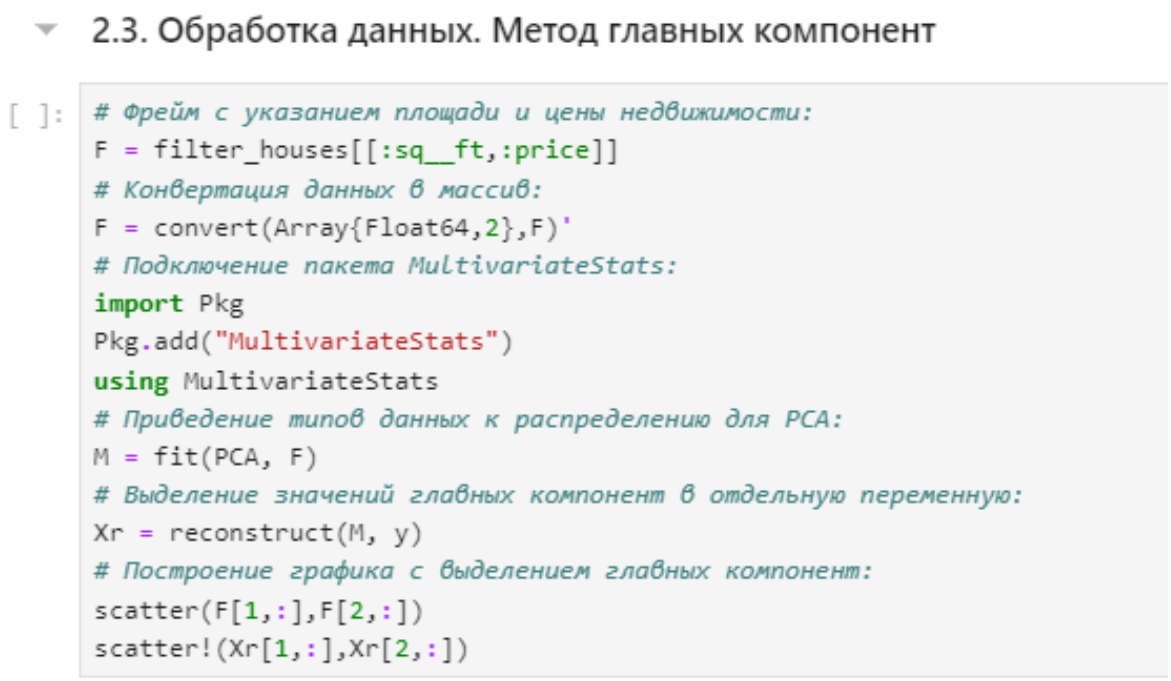

Рис. 2.33: Попытка уменьшения размера данных о цене и площади из набора данных домов

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 математическим 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):

2.4. Обработка данных. Линейная регрессия

[34]:
```
xvals = repeat(1:0.5:10,inner=2)
yvals = 3 .+ xvals + 2*rand(length(xvals)) .- 1
scatter(xvals,yvals,color=:black,leg=false)
```

[34]:

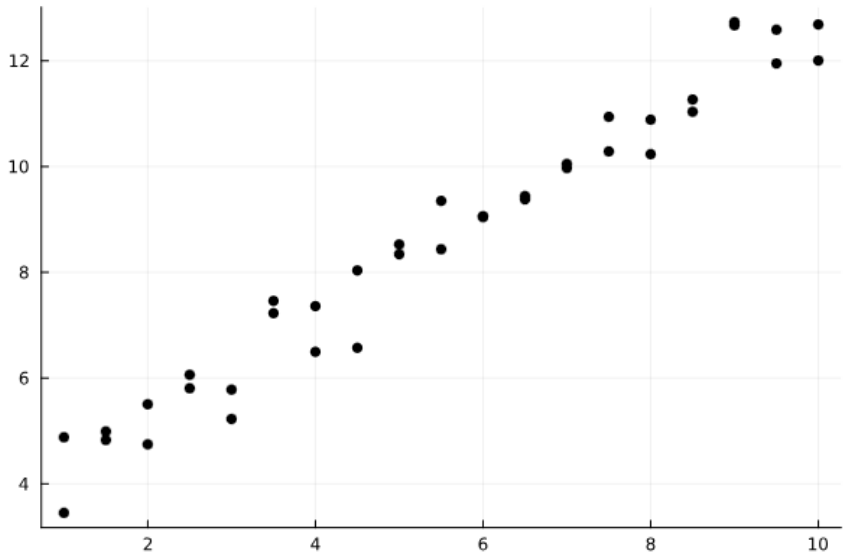

Рис. 2.34: Исходные данные

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

[36]:
```
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
a,b = find_best_fit(xvals,yvals)
ynew = a * xvals .+ b
plot!(xvals,ynew)
```

[36]:

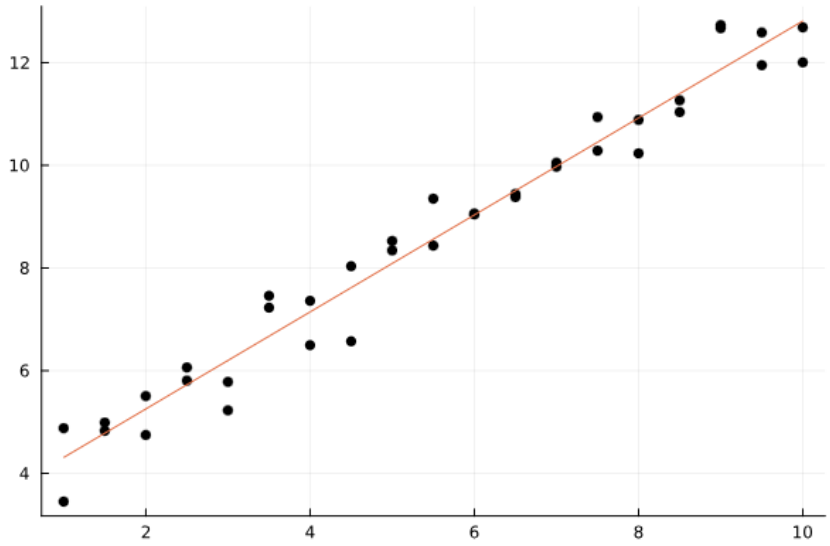

Рис. 2.35: Применение функции для построения графика

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):

```
[ ]: xvals = 1:100000;
xvals = repeat(xvals,inner=3);
yvals = 3 .+ xvals + 2*rand(length(xvals)) .- 1;
@show size(xvals)
@show size(yvals)
@time a,b = find_best_fit(xvals,yvals)
import Pkg
Pkg.add("PyCall")
Pkg.add("Conda")
using PyCall
using Conda
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
"""
xpy = PyObject(xvals)
ypy = PyObject(yvals)
@time a,b = find_best_fit_python(xpy,ypy)
import Pkg
Pkg.add("BenchmarkTools")
using BenchmarkTools
@btime a,b = find_best_fit_python(xvals,yvals)
@btime a,b = find_best_fit(xvals,yvals)
```

Рис. 2.36: Сравнение

2.13 Самостоятельное выполнение

Выполнение задания №1 (рис. 2.37):

Самостоятельное выполнение

1.1) Кластеризация

[39]:
```
# Загрузка данных
iris = dataset("datasets", "iris")
# Преобразуем DataFrame в матрицу
X = Matrix{Float64}(iris[:, 1:4])  # Используем только числовые признаки
# Применяем кластеризацию методом k-средних
k = 3  # Количество кластеров
result = kmeans(X, k)
# Визуализация кластеров
scatter(X[:, 1], X[:, 2], group = result.assignments, xlabel = "Sepal Length", ylabel = "Sepal Width", title = "K-means Clustering")
```

[39]:

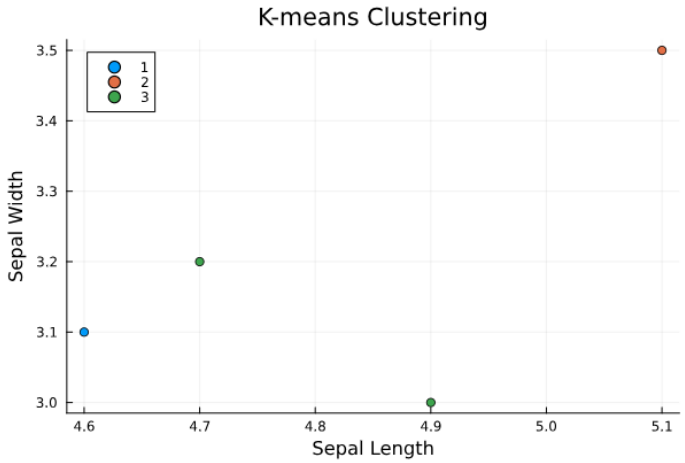

Рис. 2.37: Решение задания №1

Выполнение задания №2 (рис. 2.38 - рис. 2.39):

1.2) Регрессия (метод наименьших квадратов в случае линейной регрессии)

Часть 1.

[ ]:
```
# Генерация данных
X = randn(1000, 3)
a0 = rand(3)
y = X * a0 + 0.1 * randn(1000)
# Добавляем столбец единиц в X для учета свободного члена
X2 = hcat(ones(1000), X)
# Применяем ридж-регрессию с небольшим значением регуляризации
ridge_result = ridge(X2, y, 1e-4)
println("Ридж-регрессия, коэффициенты:")
println(ridge_result)
# Сравнение с использованием GLM.jl
# Преобразуем X2 в DataFrame, чтобы использовать его с GLM
df = DataFrame(X2 = hcat(ones(1000), X)..., y = y)  # Распаковываем X2 в отдельные столбцы
model = lm(@formula(y ~ X2), df)
println("Результаты с использованием GLM.jl:")
println(coef(model))
```

Рис. 2.38: Решение задания №2

Часть 2.

[49]:
```
# Генерация данных
X = rand(100)
y = 2 * X + 0.1 * randn(100)
# Построение графика данных
scatter(X, y, label="Data", xlabel="X", ylabel="y", title="Regression Plot")
# Линейная регрессия
X2 = hcat(ones(100), X)  # Добавляем столбец единиц
beta = (X2' * X2) \ (X2' * y)
# Добавление линии регрессии
plot!(X, x -> beta[1] + beta[2] * x, label="Regression Line", color=:red)
```

[49]:

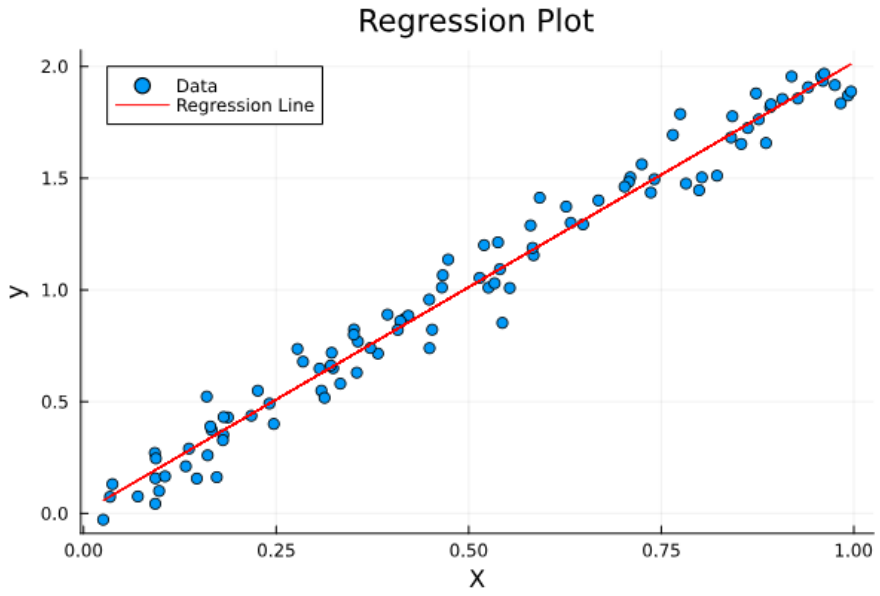

Рис. 2.39: Решение задания №2

Выполнение задания №3 (рис. 2.40 - рис. 2.42):

a.

[50]:
```
# Параметры модели
S = 100      # Начальная цена акции
T = 1        # Длительность в годах
n = 10000    # Количество периодов
sigma = 0.3  # Волатильность
r = 0.08     # Годовая процентная ставка
h = T / n    # Длина одного периода

# Расчет u и d
u = exp(r * h + sigma * sqrt(h))
d = exp(r * h - sigma * sqrt(h))

# Цена акции на каждом шаге
function create_path(S, r, sigma, T, n)
    path = zeros(Float64, n+1)
    path[1] = S
    for i in 2:n+1
        if rand() > 0.5  # Пример случайного выбора направления
            path[i] = path[i-1] * u
        else
            path[i] = path[i-1] * d
        end
    end
    return path
end
# Генерация и построение траектории
path = create_path(S, r, sigma, T, n)
plot(path, label="Stock Price Path", xlabel="Time Step", ylabel="Price", title="Stock Price Trajectory")
```

[50]:

Рис. 2.40: Решение задания №3

b.

```
# Генерация 10 траекторий
paths = [create_path(S, r, sigma, T, n) for _ in 1:10]
# Построение всех траекторий на одном графике
for path in paths
    plot!(path, label="Trajectory", xlabel="Time Step", ylabel="Price", title="Multiple Stock Price Trajectories")
end
```

Рис. 2.41: Решение задания №3

c.

```
[ ]: using Threads
# Функция для параллельной генерации траекторий
function parallel_paths(n_paths)
    paths = Vector{Array{Float64, 1}}(undef, n_paths)
    Threads.@threads for i in 1:n_paths
        paths[i] = create_path(S, r, sigma, T, n)
    end
    return paths
end
# Генерация траекторий с использованием многозадачности
paths_parallel = parallel_paths(10)
# Построение всех параллельных траекторий
for path in paths_parallel
    plot!(path, label="Trajectory", xlabel="Time Step", ylabel="Price", title="Parallel Stock Price Trajectories")
end
```

Рис. 2.42: Решение задания №3

3 Вывод

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

4 Список литературы. Библиография

[1] Julia Documentation: https://docs.julialang.org/en/v1/