

Mapping Crime Hotspots

Greg Ridgeway

Ruth Moyer

Li Sian Goh

2026-08-02

Table of contents

1	Introduction	1
2	Understanding an open data portal	3
3	Downloading and importing the Chicago crime data	3
4	Cleaning and validating the records	8
4.1	Identifiers and duplicates	8
4.2	Dates	13
4.3	Redundant information	18
4.4	Coordinates	27
5	From incident records to a point map	32
6	Hexagonal tiles and hotspot maps	35
6.1	A static, publication-friendly map	45
7	Smooth hotspot contours	46
7.1	Kernel density estimation	46
7.2	KDE in R	57
7.3	Comparing hotspots over time	67
8	Solutions to the exercises	79

1 Introduction

This lesson covers two complementary topics:

1. How to evaluate and work with city open data portals
2. How to create hotspot maps of crime incidents

City open data portals make government records available to anyone with an internet connection. Police departments, courts, transit agencies, health departments, and other public agencies use these portals to publish data in tables that can be viewed online or downloaded for analysis. These portals create opportunities for public-interest research, but a download button does not guarantee that a dataset is complete, stable, or ready to analyze.

The concept of determining the geographic locations where crime is most intense, a crime hotspot, is very important to criminological theory as well as to the very practical question of where to focus public safety efforts.

In this lesson we will use the City of Chicago's [Crimes, 2001 to Present](#) dataset. During our exploration of this dataset we will

- Learn how to evaluate a city open data portal and its documentation
- Download and load a large data file
- Clean and validate incident-level crime data
- Build point, hexagonal-bin, kernel-density, and time-comparison hotspot maps

Let's get started by loading several libraries that we will need.

```
library(dplyr)
library(tidyr)
library(lubridate)
library(ggplot2)

# for large datasets
library(data.table)
# for gzip compression
library(R.utils)

# for hotspot maps
library(sf)
library(isoband)
library(leaflet)
```


2 Understanding an open data portal

Many city portals use the Socrata platform, although the same general workflow applies to ArcGIS Hub and other systems. Before downloading anything, find the dataset's landing page and answer the following questions:

- Who created and maintains the data
- What event or unit does one row represent
- Which dates are covered
- How often the data are updated
- Which records are excluded
- Whether locations and sensitive information are obscured
- Whether classifications can change after publication

These data contain reported incidents from 2001 to the present, excluding the most recent seven days. The data are updated daily, addresses are shown only at the block level, and mapped locations are approximate.

These details affect interpretation. This is a dataset of crimes reported to the Chicago Police Department, not a measure of all victimization. Reporting, recording, enforcement, and classification practices all shape the records.

3 Downloading and importing the Chicago crime data

You can download the file in a browser or let R do it. Let's try having R download from the portal.

```
# this will take awhile... increase download timeout
options(timeout = max(3600, getOption("timeout")))

download.file(
  "https://data.cityofchicago.org/api/v3/views/ijzp-q8t2/export.csv",
  destfile = "Crimes_-_2001_to_Present.csv",
  method = "libcurl",
  mode = "wb",
  quiet = FALSE)
```

Just how big is this file?

```
file.info("Crimes_-_2001_to_Present.csv") |>
  transmute(size_gb = size / 109) # transmute = mutate + select
```

```
              size_gb
Crimes_-_2001_to_Present.csv 2.391954
```

You can save space by compressing the file.

```
R.utils::gzip("Crimes_-_2001_to_Present.csv",
  remove = FALSE,
  overwrite = TRUE)
```

```
file.info("Crimes_-_2001_to_Present.csv.gz") |>
  transmute(size_gb = size / 109)
```

```
              size_gb
Crimes_-_2001_to_Present.csv.gz 0.5307424
```

Let's peek at the first few rows. `scan()` can read directly from a compressed CSV file and makes no attempt at formatting or organizing the data.

```
scan(file="Crimes_-_2001_to_Present.csv.gz",
  nmax = 3,
  what="",
  sep="\n") |>
  gsub("'", "", x=_) # clear out quotes to be more readable
```

```
[1] "ID,Case Number,Date,Block,IUCR,Primary Type,Description,Location Description,Arrest,Dome
[2] "14276932,JK350639,07/23/2026 12:00:00 AM,020XX W WEBSTER AVE,0890,THEFT,FROM BUILDING,R
87.678225841,(41.921383493, -87.678225841)"
[3] "14271121,JK344338,07/23/2026 12:00:00 AM,016XX S MILLER ST,0710,THEFT,THEFT FROM MOTOR V
87.652227797,(41.85927278, -87.652227797)"
```

The data appear to be fairly clear, plus the portal provides a data dictionary. These fields are especially useful for our analysis.

Field	Meaning
ID	Unique row identifier

Field	Meaning
Case Number	Police Records Division number
Date	Estimated date and time of occurrence
Block	Partially redacted block address
IUCR	Illinois Uniform Crime Reporting code
Primary Type	Broad category associated with the IUCR code
Description	More detailed offense description
Location Description	Type of place where the incident occurred
Arrest	Whether an arrest was made
Domestic	Whether the incident was marked domestic
Beat	Small police patrol area
District	Police district
Ward	City Council ward
Community Area	One of Chicago's 77 community areas
Year	Calendar year associated with the record
Latitude, Longitude	Approximate mapped location

The complete file also includes x and y coordinates in a projected coordinate system, an update timestamp, and a combined `Location` field. We will keep only the columns needed for this lesson. Selecting columns during import reduces memory use.

The `fread()` function in the `data.table` package is much faster and more memory-efficient than base R's `read.csv()` for a file of this size. It can directly read compressed CSV files and also lets us pick which columns to import, which reduces memory use.

```
crime <- fread(
  "Crimes_-_2001_to_Present.csv.gz",
  select = c("ID", "Case Number", "Date", "Block", "IUCR", "Primary Type",
             "Description", "Location Description", "Arrest", "Domestic",
             "Beat", "District", "Ward", "Community Area", "FBI Code", "Year",
             "Latitude", "Longitude"),
  # make sure these get stored as characters
  colClasses = list(character = c("IUCR", "FBI Code", "Beat", "District",
                                   "Ward", "Community Area")),
  na.strings = c("", "NA"),
  showProgress = TRUE)

# set nicer names (no spaces)
setnames(crime,
  new = c("id", "case_number", "date", "block", "iucr",
           "primary_type", "description", "location_description",
           "arrest", "domestic", "beat", "district", "ward",
```

```
"community_area", "fbi_code", "year",  
"latitude", "longitude"))
```

Check how much memory the imported object occupies.

```
object.size(crime) |> format(units = "GB")
```

```
[1] "1.8 Gb"
```

How many rows and columns?

```
dim(crime)
```

```
[1] 8604907      18
```

What are the columns in this dataset?

```
str(crime)
```

Classes 'data.table' and 'data.frame': 8604907 obs. of 18 variables:

```
$ id          : int  14276932 14271121 14273747 14274625 14270932 14274370 14271781 ...  
$ case_number : chr   "JK350639" "JK344338" "JK347526" "JK348648" ...  
$ date        : chr   "07/23/2026 12:00:00 AM" "07/23/2026 12:00:00 AM" "07/23/2026 ...  
$ block       : chr   "020XX W WEBSTER AVE" "016XX S MILLER ST" "001XX W 111TH ST" ...  
$ iucr        : chr   "0890" "0710" "0610" "0580" ...  
$ primary_type: chr   "THEFT" "THEFT" "BURGLARY" "STALKING" ...  
$ description : chr   "FROM BUILDING" "THEFT FROM MOTOR VEHICLE" "FORCIBLE ENTRY" "S...  
$ location_description: chr "RESIDENCE - YARD (FRONT / BACK)" "STREET" "RESIDENCE" "RESIDE...  
$ arrest      : logi  FALSE FALSE FALSE FALSE FALSE FALSE ...  
$ domestic    : logi  FALSE FALSE FALSE TRUE TRUE FALSE ...  
$ beat        : chr   "1432" "1235" "0522" "0632" ...  
$ district    : chr   "014" "012" "005" "006" ...  
$ ward        : chr   "32" "25" "9" "6" ...  
$ community_area : chr "22" "31" "49" "44" ...  
$ fbi_code    : chr   "06" "06" "05" "08A" ...  
$ year        : int   2026 2026 2026 2026 2026 2026 2026 2026 2026 2026 ...  
$ latitude    : num   41.9 41.9 41.7 41.7 41.7 ...  
$ longitude   : num  -87.7 -87.7 -87.6 -87.6 -87.6 ...  
- attr(*, ".internal.selfref")=<pointer: 0x000001caae12c1b0>
```

Should we worry about missing data?

```
crime |>
  summarize(across(everything(), ~sum(is.na(.x)))) |>
  pivot_longer(everything(),
               names_to = "field",
               values_to = "missing") |>
  arrange(desc(missing)) |>
  print(n=Inf)
```

```
# A tibble: 18 x 2
  field      missing
  <chr>      <int>
1 ward      614813
2 community_area 613724
3 latitude    97159
4 longitude    97159
5 location_description 16398
6 district     47
7 id           0
8 case_number   0
9 date          0
10 block        0
11 iucr          0
12 primary_type  0
13 description   0
14 arrest        0
15 domestic      0
16 beat          0
17 fbi_code      0
18 year         0
```

The yearly totals in Figure 1 show that this extract spans 2001 through 2026. Counts generally decline over much of the period. What might explain the lower counts in 2026?

```
crime |>
  count(year) |>
  ggplot(aes(x = year, y = n)) +
  geom_col() +
  scale_x_continuous(breaks = seq(min(crime$year), max(crime$year), by = 2)) +
  scale_y_continuous(labels = scales::label_comma()) +
```

```
labs(x = "Year", y = "Reported crimes") +
theme_minimal()
```

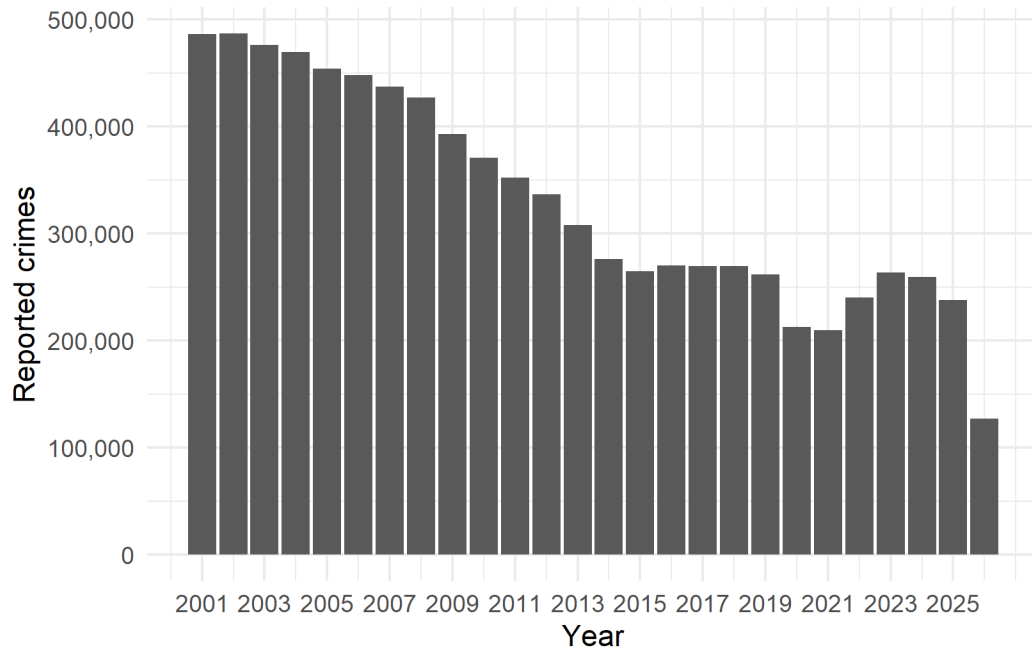


Figure 1: Reported crimes by year in the raw Chicago crime extract.

4 Cleaning and validating the records

In this section we will march through the columns and see if there is some cleanup and formatting to do before we can work with the data.

4.1 Identifiers and duplicates

The portal describes ID as the unique row identifier. Confirm that claim.

```
crime |>
  summarize(rows           = n(),
            unique_ids     = n_distinct(id),
            duplicated_ids  = sum(duplicated(id)),
            unique_case_numbers = n_distinct(case_number))
```

	rows	unique_ids	duplicated_ids	unique_case_numbers
1	8604907	8604907	0	8604281

id is unique, but case_number is not. Let's count how many records are associated with each case number.

```
case_number_counts <- crime |>
  count(case_number, name = "records") |>
  filter(records > 1)
head(case_number_counts)
```

	case_number	records
	<char>	<int>
1:	G023235	2
2:	G083440	2
3:	G137655	2
4:	G183906	2
5:	G219399	2
6:	G262626	2

```
# how many times are these case numbers duplicated?
case_number_counts |>
  count(records, name = "case_numbers")
```

	records	case_numbers
	<int>	<int>
1:	2	446
2:	3	56
3:	4	14
4:	5	4
5:	6	2

Most repeated case numbers occur twice, although a few occur as many as six times. How many records are involved, and what kinds of crimes are they?

```
# semi-join keeps rows first data frame when they match the second data frame
# It returns only columns from first data frame
repeated_case_records <- crime |>
  semi_join(case_number_counts, by = "case_number")
```

```
repeated_case_records |>
  summarize(repeated_case_numbers = n_distinct(case_number),
            records = n(),
            additional_records = n() - n_distinct(case_number))
```

	repeated_case_numbers	records	additional_records
1	522	1148	626

```
repeated_case_records |>
  count(primary_type, iucr, description, sort = TRUE)
```

	primary_type	iucr	description	n
	<char>	<char>	<char>	<int>
1:	HOMICIDE	0110	FIRST DEGREE MURDER	1148

In this extract, 522 case numbers occur more than once, involving 1148 records. All of them are homicides with IUCR code 0110 and the description **FIRST DEGREE MURDER**. This is consistent with the portal's statement that murder records represent victims rather than incidents. The largest case numbers illustrate the pattern.

```
largest_cases <- case_number_counts |>
  slice_max(records, n = 5, with_ties = FALSE)
largest_cases
```

	case_number	records
	<char>	<int>
1:	HJ590004	6
2:	HZ140230	6
3:	HP296582	5
4:	HS256531	5
5:	JC470284	5

```
repeated_case_records |>
  semi_join(largest_cases, by = "case_number") |>
  select(id, case_number, date, iucr, primary_type, description) |>
  arrange(case_number, date, id)
```


	id	case_number		date	iucr	primary_type
	<int>	<char>		<char>	<char>	<char>
1:	2346	HJ590004	08/27/2003	08:35:00 AM	0110	HOMICIDE
2:	2347	HJ590004	08/27/2003	08:35:00 AM	0110	HOMICIDE
3:	2348	HJ590004	08/27/2003	08:35:00 AM	0110	HOMICIDE
4:	2349	HJ590004	08/27/2003	08:35:00 AM	0110	HOMICIDE
5:	2350	HJ590004	08/27/2003	08:35:00 AM	0110	HOMICIDE
6:	2351	HJ590004	08/27/2003	08:35:00 AM	0110	HOMICIDE
7:	4498	HP296582	04/23/2008	06:15:00 PM	0110	HOMICIDE
8:	4499	HP296582	04/23/2008	06:15:00 PM	0110	HOMICIDE
9:	4500	HP296582	04/23/2008	06:15:00 PM	0110	HOMICIDE
10:	4501	HP296582	04/23/2008	06:15:00 PM	0110	HOMICIDE
11:	4502	HP296582	04/23/2008	06:15:00 PM	0110	HOMICIDE
12:	19332	HS256531	04/14/2010	04:25:00 AM	0110	HOMICIDE
13:	19333	HS256531	04/14/2010	04:25:00 AM	0110	HOMICIDE
14:	19334	HS256531	04/14/2010	04:25:00 AM	0110	HOMICIDE
15:	19335	HS256531	04/14/2010	04:25:00 AM	0110	HOMICIDE
16:	19369	HS256531	05/01/2010	05:45:00 PM	0110	HOMICIDE
17:	22313	HZ140230	02/04/2016	01:00:00 PM	0110	HOMICIDE
18:	22314	HZ140230	02/04/2016	01:00:00 PM	0110	HOMICIDE
19:	22316	HZ140230	02/04/2016	01:00:00 PM	0110	HOMICIDE
20:	22317	HZ140230	02/04/2016	01:00:00 PM	0110	HOMICIDE
21:	22318	HZ140230	02/04/2016	01:00:00 PM	0110	HOMICIDE
22:	22319	HZ140230	02/04/2016	01:00:00 PM	0110	HOMICIDE
23:	24790	JC470284	10/12/2019	06:28:00 PM	0110	HOMICIDE
24:	24792	JC470284	10/12/2019	06:37:00 PM	0110	HOMICIDE
25:	24791	JC470284	10/12/2019	06:39:00 PM	0110	HOMICIDE
26:	24789	JC470284	10/12/2019	07:16:00 PM	0110	HOMICIDE
27:	24794	JC470284	10/13/2019	10:28:00 AM	0110	HOMICIDE

	id	case_number		date	iucr	primary_type
	<int>	<char>		<char>	<char>	<char>
		description				
		<char>				
1:		FIRST DEGREE MURDER				
2:		FIRST DEGREE MURDER				
3:		FIRST DEGREE MURDER				
4:		FIRST DEGREE MURDER				
5:		FIRST DEGREE MURDER				
6:		FIRST DEGREE MURDER				
7:		FIRST DEGREE MURDER				
8:		FIRST DEGREE MURDER				
9:		FIRST DEGREE MURDER				
10:		FIRST DEGREE MURDER				

```

11: FIRST DEGREE MURDER
12: FIRST DEGREE MURDER
13: FIRST DEGREE MURDER
14: FIRST DEGREE MURDER
15: FIRST DEGREE MURDER
16: FIRST DEGREE MURDER
17: FIRST DEGREE MURDER
18: FIRST DEGREE MURDER
19: FIRST DEGREE MURDER
20: FIRST DEGREE MURDER
21: FIRST DEGREE MURDER
22: FIRST DEGREE MURDER
23: FIRST DEGREE MURDER
24: FIRST DEGREE MURDER
25: FIRST DEGREE MURDER
26: FIRST DEGREE MURDER
27: FIRST DEGREE MURDER

```

```

description
<char>

```

These are not duplicated rows. Every record has a different ID, and records sharing a case number can have different dates and times.

```

crime |>
  select(id, case_number, date, block, description) |>
  filter(case_number == "JC470284")

```

	id	case_number		date		block
	<int>	<char>		<char>		<char>
1:	24794	JC470284	10/13/2019	10:28:00 AM	067XX W	IRVING PARK RD
2:	24789	JC470284	10/12/2019	07:16:00 PM	067XX W	IRVING PARK RD
3:	24791	JC470284	10/12/2019	06:39:00 PM	067XX W	IRVING PARK RD
4:	24792	JC470284	10/12/2019	06:37:00 PM	067XX W	IRVING PARK RD
5:	24790	JC470284	10/12/2019	06:28:00 PM	067XX W	IRVING PARK RD

```

description
<char>
1: FIRST DEGREE MURDER
2: FIRST DEGREE MURDER
3: FIRST DEGREE MURDER
4: FIRST DEGREE MURDER
5: FIRST DEGREE MURDER

```

Deduplicating on `case_number` would incorrectly collapse multiple homicide victims into one record. Whether homicide should be counted by victim or by case depends on the research question.

4.2 Dates

Let's start by formatting the date and setting the time zone.

```
crime <- crime |>
  mutate(dateFormatted = mdy_hms(date, tz = "America/Chicago"))
```

Warning: There was 1 warning in `mutate()`.

i In argument: `dateFormatted = mdy\_hms(date, tz = "America/Chicago")`.

Caused by warning:

! 253 failed to parse.

There are some warnings about invalid dates. Let's check on why some of those date conversions failed.

```
crime |>
  filter(is.na(dateFormatted)) |>
  select(id, date, dateFormatted)
```

	id	date	dateFormatted
	<int>	<char>	<POSct>
1:	12002880	03/08/2020 02:30:00 AM	<NA>
2:	12003924	03/08/2020 02:30:00 AM	<NA>
3:	12002663	03/08/2020 02:15:00 AM	<NA>
4:	12018210	03/08/2020 02:00:00 AM	<NA>
5:	11623554	03/10/2019 02:30:00 AM	<NA>

249:	1454410	04/01/2001 02:00:00 AM	<NA>
250:	1454426	04/01/2001 02:00:00 AM	<NA>
251:	1454665	04/01/2001 02:00:00 AM	<NA>
252:	1456495	04/01/2001 02:00:00 AM	<NA>
253:	1459986	04/01/2001 02:00:00 AM	<NA>

Are the dates just missing in certain years?

```
crime |>
  summarize(n_missing = sum(is.na(dateFormatted)),
            .by = "year") |>
  arrange(desc(year))
```

	year	n_missing
1	2026	0
2	2025	0
3	2024	0
4	2023	0
5	2022	0
6	2021	0
7	2020	4
8	2019	2
9	2018	8
10	2017	9
11	2016	10
12	2015	8
13	2014	11
14	2013	17
15	2012	17
16	2011	15
17	2010	16
18	2009	16
19	2008	14
20	2007	13
21	2006	21
22	2005	19
23	2004	16
24	2003	14
25	2002	11
26	2001	12

Looks like problems with the dates are a thing of the past... and really not that common. But still, why are they missing?

I can see that the original date looks okay. I also notice that a lot of the dates that end up as NA after applying `mdy_hms()` are around 2:00 AM. Why would `mdy_hms()` have trouble with that? Let's run some tests.

```
mdy_hms("03/08/2020 02:30:00 AM", tz = "America/Chicago")
```

Warning: 1 failed to parse.

```
[1] NA
```

Something appears to be special about this particular time, because other times work just fine.

```
mdy_hms("03/08/2020 01:30:00 AM", tz = "America/Chicago")
```

```
[1] "2020-03-08 01:30:00 CST"
```

```
mdy_hms("03/08/2020 03:30:00 AM", tz = "America/Chicago")
```

```
[1] "2020-03-08 03:30:00 CDT"
```

```
mdy_hms("03/08/2020 02:30:00 PM", tz = "America/Chicago")
```

```
[1] "2020-03-08 14:30:00 CDT"
```

Do you notice something about these formatted dates that holds the secret? Note that the 1:30 AM and 3:30 AM times are formatted correctly. But the 1:30 AM timezone is Central Standard Time (CST) and the 3:30 AM timezone is Central Daylight Time (CDT). This is because daylight saving time started on March 8, 2020 at 2:00 AM. The clocks were moved forward one hour to 3:00 AM, so 2:30 AM cannot possibly happen on that day!

Two things to remember:

1. Time zones and daylight saving time can be tricky
2. Checking the original data often reveals the source of a problem

```
# generate all days since 2001
allDays <- ymd_hms("2001-01-01 12:00:00", tz="America/Chicago") +
  days(0:floor(26*365.25)) # for 26 years

head(allDays)
```

```
[1] "2001-01-01 12:00:00 CST" "2001-01-02 12:00:00 CST"
[3] "2001-01-03 12:00:00 CST" "2001-01-04 12:00:00 CST"
[5] "2001-01-05 12:00:00 CST" "2001-01-06 12:00:00 CST"
```

```
# on which days do we see DST change
spring_transitions <- which(diff(dst(allDays)) == 1)

# here is the day just before the change
allDays[spring_transitions]
```

```
[1] "2001-03-31 12:00:00 CST" "2002-04-06 12:00:00 CST"
[3] "2003-04-05 12:00:00 CST" "2004-04-03 12:00:00 CST"
[5] "2005-04-02 12:00:00 CST" "2006-04-01 12:00:00 CST"
[7] "2007-03-10 12:00:00 CST" "2008-03-08 12:00:00 CST"
[9] "2009-03-07 12:00:00 CST" "2010-03-13 12:00:00 CST"
[11] "2011-03-12 12:00:00 CST" "2012-03-10 12:00:00 CST"
[13] "2013-03-09 12:00:00 CST" "2014-03-08 12:00:00 CST"
[15] "2015-03-07 12:00:00 CST" "2016-03-12 12:00:00 CST"
[17] "2017-03-11 12:00:00 CST" "2018-03-10 12:00:00 CST"
[19] "2019-03-09 12:00:00 CST" "2020-03-07 12:00:00 CST"
[21] "2021-03-13 12:00:00 CST" "2022-03-12 12:00:00 CST"
[23] "2023-03-11 12:00:00 CST" "2024-03-09 12:00:00 CST"
[25] "2025-03-08 12:00:00 CST" "2026-03-07 12:00:00 CST"
```

```
# if we add one day...
allDays[spring_transitions] + days(1)
```

```
[1] "2001-04-01 12:00:00 CDT" "2002-04-07 12:00:00 CDT"
[3] "2003-04-06 12:00:00 CDT" "2004-04-04 12:00:00 CDT"
[5] "2005-04-03 12:00:00 CDT" "2006-04-02 12:00:00 CDT"
[7] "2007-03-11 12:00:00 CDT" "2008-03-09 12:00:00 CDT"
[9] "2009-03-08 12:00:00 CDT" "2010-03-14 12:00:00 CDT"
[11] "2011-03-13 12:00:00 CDT" "2012-03-11 12:00:00 CDT"
[13] "2013-03-10 12:00:00 CDT" "2014-03-09 12:00:00 CDT"
[15] "2015-03-08 12:00:00 CDT" "2016-03-13 12:00:00 CDT"
[17] "2017-03-12 12:00:00 CDT" "2018-03-11 12:00:00 CDT"
[19] "2019-03-10 12:00:00 CDT" "2020-03-08 12:00:00 CDT"
[21] "2021-03-14 12:00:00 CDT" "2022-03-13 12:00:00 CDT"
[23] "2023-03-12 12:00:00 CDT" "2024-03-10 12:00:00 CDT"
[25] "2025-03-09 12:00:00 CDT" "2026-03-08 12:00:00 CDT"
```

```
# spring DST dates
datesSpringDST <- date(allDays[spring_transitions] + days(1))
```

If we see any of these dates with a time between 2:00 AM and 2:59 AM, we will add one hour to them... Some officer did not set their watch ahead, but we can do that for them.

```
crime <- crime |>
  mutate(datetime0 = mdy_hms(date, tz = "UTC"), # UTC ignores DST
    datetime0 = if_else(
      date(datetime0) %in% datesSpringDST &
        hour(datetime0) == 2,
      datetime0 + hours(1),
      datetime0),
    # force_tz() keeps HMS, just changes the time zone
    dateFormatted = force_tz(datetime0, tzone = "America/Chicago"))

# any missing now?
sum(is.na(crime$dateFormatted))
```

```
[1] 0
```

No more missing dates. Let's tidy up before moving on.

```
crime <- crime |>
  select(-c(date, datetime0)) |>
  rename(date = dateFormatted) |>
  relocate(date, .after = case_number)
```

And let's do a final quick check that the dates agree with year.

```
crime |>
  summarize(sum(year != year(date)))
```

```
sum(year != year(date))
1          0
```

4.3 Redundant information

The data have columns for `iucr`, `primary_type`, `fbi_code`, and `description`. Let's peek at the first couple of rows.

```
crime |>
  select(iucr, primary_type, fbi_code, description) |>
  head()
```

	iucr	primary_type	fbi_code	description
	<char>	<char>	<char>	<char>
1:	0890	THEFT	06	FROM BUILDING
2:	0710	THEFT	06	THEFT FROM MOTOR VEHICLE
3:	0610	BURGLARY	05	FORCIBLE ENTRY
4:	0580	STALKING	08A	SIMPLE
5:	0810	THEFT	06	OVER \$500
6:	0710	THEFT	06	THEFT FROM MOTOR VEHICLE

These four fields are all connected and offer different descriptions about the same offense. `iucr`, the Illinois Uniform Crime Reporting code, is the key field that everything hooks on. Once you know the IUCR code, then the other three fields must follow. Let's check if there are any IUCR codes that have more than one label in the other three fields.

```
crime |>
  distinct(iucr, primary_type, fbi_code, description) |>
  count(iucr, name = "n_labels") |>
  filter(n_labels > 1) |>
  arrange(desc(n_labels))
```

	iucr	n_labels
	<char>	<int>
1:	0263	3
2:	0320	3
3:	033A	3
4:	0340	3
5:	0420	3

180:	5121	2
181:	5122	2
182:	5130	2
183:	5131	2
184:	5132	2

Looks like there are many fields with more than one label. Let's see what they are.

```
crime |>
  distinct(iucr, primary_type, fbi_code, description, year) |>
  filter(iucr=="0263") |>
  arrange(desc(year))
```

	iucr <char>	primary_type <char>	fbi_code <char>
1:	0263 CRIMINAL	SEXUAL ASSAULT	02
2:	0263 CRIMINAL	SEXUAL ASSAULT	02
3:	0263 CRIMINAL	SEXUAL ASSAULT	02
4:	0263 CRIMINAL	SEXUAL ASSAULT	02
5:	0263 CRIMINAL	SEXUAL ASSAULT	02
6:	0263 CRIMINAL	SEXUAL ASSAULT	02
7:	0263 CRIMINAL	SEXUAL ASSAULT	02
8:	0263 CRIM	SEXUAL ASSAULT	02
9:	0263 CRIM	SEXUAL ASSAULT	02
10:	0263 CRIMINAL	SEXUAL ASSAULT	02
11:	0263 CRIMINAL	SEXUAL ASSAULT	02
12:	0263 CRIM	SEXUAL ASSAULT	02
13:	0263 CRIMINAL	SEXUAL ASSAULT	02
14:	0263 CRIMINAL	SEXUAL ASSAULT	02
15:	0263 CRIM	SEXUAL ASSAULT	02
16:	0263 CRIM	SEXUAL ASSAULT	02
17:	0263 CRIMINAL	SEXUAL ASSAULT	02
18:	0263 CRIM	SEXUAL ASSAULT	02
19:	0263 CRIMINAL	SEXUAL ASSAULT	02
20:	0263 CRIM	SEXUAL ASSAULT	02
21:	0263 CRIMINAL	SEXUAL ASSAULT	02
22:	0263 CRIM	SEXUAL ASSAULT	02
23:	0263 CRIMINAL	SEXUAL ASSAULT	02
24:	0263 CRIM	SEXUAL ASSAULT	02
25:	0263 CRIMINAL	SEXUAL ASSAULT	02
26:	0263 CRIM	SEXUAL ASSAULT	02
27:	0263 CRIM	SEXUAL ASSAULT	02
28:	0263 CRIMINAL	SEXUAL ASSAULT	02
29:	0263 CRIM	SEXUAL ASSAULT	02
30:	0263 CRIM	SEXUAL ASSAULT	02
31:	0263 CRIMINAL	SEXUAL ASSAULT	02
32:	0263 CRIMINAL	SEXUAL ASSAULT	02
33:	0263 CRIM	SEXUAL ASSAULT	02

34:	0263	CRIM SEXUAL ASSAULT	02
35:	0263	CRIMINAL SEXUAL ASSAULT	02
36:	0263	CRIM SEXUAL ASSAULT	02
37:	0263	CRIMINAL SEXUAL ASSAULT	02
38:	0263	CRIM SEXUAL ASSAULT	02
39:	0263	CRIMINAL SEXUAL ASSAULT	02
40:	0263	CRIM SEXUAL ASSAULT	02
41:	0263	CRIMINAL SEXUAL ASSAULT	02
42:	0263	CRIM SEXUAL ASSAULT	02
43:	0263	CRIMINAL SEXUAL ASSAULT	02
44:	0263	CRIM SEXUAL ASSAULT	02
45:	0263	CRIMINAL SEXUAL ASSAULT	02
	iucr	primary_type	fbi_code
	<char>	<char>	<char>
		description	year
		<char>	<int>
1:	AGGRAVATED - KNIFE / CUTTING INSTRUMENT		2026
2:	AGGRAVATED - KNIFE / CUTTING INSTRUMENT		2025
3:	AGGRAVATED - KNIFE / CUTTING INSTRUMENT		2024
4:	AGGRAVATED - KNIFE / CUTTING INSTRUMENT		2023
5:	AGGRAVATED - KNIFE / CUTTING INSTRUMENT		2022
6:	AGGRAVATED - KNIFE / CUTTING INSTRUMENT		2021
7:	AGGRAVATED - KNIFE / CUTTING INSTRUMENT		2020
8:	AGGRAVATED: KNIFE/CUT INSTR		2020
9:	AGGRAVATED: KNIFE/CUT INSTR		2019
10:	AGGRAVATED - KNIFE / CUTTING INSTRUMENT		2019
11:	AGGRAVATED: KNIFE / CUTTING INSTRUMENT		2019
12:	AGGRAVATED: KNIFE/CUT INSTR		2018
13:	AGGRAVATED - KNIFE / CUTTING INSTRUMENT		2018
14:	AGGRAVATED - KNIFE / CUTTING INSTRUMENT		2017
15:	AGGRAVATED: KNIFE/CUT INSTR		2017
16:	AGGRAVATED: KNIFE/CUT INSTR		2016
17:	AGGRAVATED - KNIFE / CUTTING INSTRUMENT		2016
18:	AGGRAVATED: KNIFE/CUT INSTR		2015
19:	AGGRAVATED - KNIFE / CUTTING INSTRUMENT		2015
20:	AGGRAVATED: KNIFE/CUT INSTR		2014
21:	AGGRAVATED - KNIFE / CUTTING INSTRUMENT		2014
22:	AGGRAVATED: KNIFE/CUT INSTR		2013
23:	AGGRAVATED - KNIFE / CUTTING INSTRUMENT		2013
24:	AGGRAVATED: KNIFE/CUT INSTR		2012
25:	AGGRAVATED - KNIFE / CUTTING INSTRUMENT		2012
26:	AGGRAVATED: KNIFE/CUT INSTR		2011
27:	AGGRAVATED: KNIFE/CUT INSTR		2010

28:	AGGRAVATED - KNIFE / CUTTING INSTRUMENT	2010
29:	AGGRAVATED: KNIFE/CUT INSTR	2009
30:	AGGRAVATED: KNIFE/CUT INSTR	2008
31:	AGGRAVATED - KNIFE / CUTTING INSTRUMENT	2008
32:	AGGRAVATED - KNIFE / CUTTING INSTRUMENT	2007
33:	AGGRAVATED: KNIFE/CUT INSTR	2007
34:	AGGRAVATED: KNIFE/CUT INSTR	2006
35:	AGGRAVATED - KNIFE / CUTTING INSTRUMENT	2006
36:	AGGRAVATED: KNIFE/CUT INSTR	2005
37:	AGGRAVATED - KNIFE / CUTTING INSTRUMENT	2005
38:	AGGRAVATED: KNIFE/CUT INSTR	2004
39:	AGGRAVATED - KNIFE / CUTTING INSTRUMENT	2004
40:	AGGRAVATED: KNIFE/CUT INSTR	2003
41:	AGGRAVATED - KNIFE / CUTTING INSTRUMENT	2003
42:	AGGRAVATED: KNIFE/CUT INSTR	2002
43:	AGGRAVATED - KNIFE / CUTTING INSTRUMENT	2002
44:	AGGRAVATED: KNIFE/CUT INSTR	2001
45:	AGGRAVATED - KNIFE / CUTTING INSTRUMENT	2001

	description	year
	<char>	<int>

Looks like there have been some changes in spelling and abbreviations over time. This might take some work, but we need to harmonize the labels to avoid problems later on.

```
crime <- crime |>
  mutate(primary_type =
    replace_values(primary_type,
      "CRIM SEXUAL ASSAULT"~"CRIMINAL SEXUAL ASSAULT"),
    description =
      replace_values(description,
        "AGGRAVATED: KNIFE/CUT INSTR" ~
          "AGGRAVATED - KNIFE / CUTTING INSTRUMENT",
        "AGGRAVATED: KNIFE / CUTTING INSTRUMENT" ~
          "AGGRAVATED - KNIFE / CUTTING INSTRUMENT"))
```

Let's check whether that resolved the issue with IUCR 0263.

```
crime |>
  distinct(iucr, primary_type, fbi_code, description) |>
  filter(iucr=="0263")
```

iucr	primary_type	fbi_code
------	--------------	----------

```

    <char>          <char>    <char>
1:   0263 CRIMINAL SEXUAL ASSAULT      02
                                description
                                <char>
1: AGGRAVATED - KNIFE / CUTTING INSTRUMENT

```

There are a lot more to go so let's check on another IUCR code.

```

crime |>
  distinct(iucr, primary_type, fbi_code, description, year) |>
  filter(iucr=="0320") |>
  arrange(desc(year))

```

	iucr	primary_type	fbi_code	description	year
	<char>	<char>	<char>	<char>	<int>
1:	0320	ROBBERY	03	STRONG ARM - NO WEAPON	2026
2:	0320	ROBBERY	03	STRONG ARM - NO WEAPON	2025
3:	0320	ROBBERY	03	STRONG ARM - NO WEAPON	2024
4:	0320	ROBBERY	03	STRONG ARM - NO WEAPON	2023
5:	0320	ROBBERY	03	STRONG ARM - NO WEAPON	2022
6:	0320	ROBBERY	03	STRONG ARM - NO WEAPON	2021
7:	0320	ROBBERY	03	STRONG ARM - NO WEAPON	2020
8:	0320	ROBBERY	03	STRONGARM - NO WEAPON	2020
9:	0320	ROBBERY	03	STRONGARM: NO WEAPON	2020
10:	0320	ROBBERY	03	STRONGARM - NO WEAPON	2019
11:	0320	ROBBERY	03	STRONG ARM - NO WEAPON	2019
12:	0320	ROBBERY	03	STRONGARM - NO WEAPON	2018
13:	0320	ROBBERY	03	STRONG ARM - NO WEAPON	2018
14:	0320	ROBBERY	03	STRONGARM - NO WEAPON	2017
15:	0320	ROBBERY	03	STRONG ARM - NO WEAPON	2017
16:	0320	ROBBERY	03	STRONGARM - NO WEAPON	2016
17:	0320	ROBBERY	03	STRONG ARM - NO WEAPON	2016
18:	0320	ROBBERY	03	STRONGARM - NO WEAPON	2015
19:	0320	ROBBERY	03	STRONG ARM - NO WEAPON	2015
20:	0320	ROBBERY	03	STRONGARM - NO WEAPON	2014
21:	0320	ROBBERY	03	STRONG ARM - NO WEAPON	2014
22:	0320	ROBBERY	03	STRONGARM - NO WEAPON	2013
23:	0320	ROBBERY	03	STRONG ARM - NO WEAPON	2013
24:	0320	ROBBERY	03	STRONGARM - NO WEAPON	2012
25:	0320	ROBBERY	03	STRONG ARM - NO WEAPON	2012
26:	0320	ROBBERY	03	STRONGARM - NO WEAPON	2011
27:	0320	ROBBERY	03	STRONG ARM - NO WEAPON	2011

28:	0320	ROBBERY	03	STRONGARM - NO WEAPON	2010
29:	0320	ROBBERY	03	STRONG ARM - NO WEAPON	2010
30:	0320	ROBBERY	03	STRONGARM - NO WEAPON	2009
31:	0320	ROBBERY	03	STRONG ARM - NO WEAPON	2009
32:	0320	ROBBERY	03	STRONGARM - NO WEAPON	2008
33:	0320	ROBBERY	03	STRONG ARM - NO WEAPON	2008
34:	0320	ROBBERY	03	STRONGARM - NO WEAPON	2007
35:	0320	ROBBERY	03	STRONG ARM - NO WEAPON	2007
36:	0320	ROBBERY	03	STRONGARM - NO WEAPON	2006
37:	0320	ROBBERY	03	STRONG ARM - NO WEAPON	2006
38:	0320	ROBBERY	03	STRONGARM - NO WEAPON	2005
39:	0320	ROBBERY	03	STRONG ARM - NO WEAPON	2005
40:	0320	ROBBERY	03	STRONGARM - NO WEAPON	2004
41:	0320	ROBBERY	03	STRONG ARM - NO WEAPON	2004
42:	0320	ROBBERY	03	STRONGARM - NO WEAPON	2003
43:	0320	ROBBERY	03	STRONG ARM - NO WEAPON	2003
44:	0320	ROBBERY	03	STRONGARM - NO WEAPON	2002
45:	0320	ROBBERY	03	STRONG ARM - NO WEAPON	2002
46:	0320	ROBBERY	03	STRONGARM - NO WEAPON	2001
47:	0320	ROBBERY	03	STRONG ARM - NO WEAPON	2001

	iucr	primary_type	fbi_code	description	year
	<char>	<char>	<char>	<char>	<int>

Again we see some spelling changes. It is possible that spelling changes are driving all of these inconsistencies.

Do FBI codes change over time? Let's check on that.

```
crime |>
  distinct(iucr, fbi_code) |>
  count(iucr) |>
  filter(n > 1)
```

Empty data.table (0 rows and 2 cols): iucr,n

FBI codes look okay. There are 418 unique IUCR codes and each one of them is linked to a single FBI code (the reverse is not true... some FBI codes include multiple IUCR codes).

Is `primary_type` a problem?

```
crime |>
  distinct(iucr, primary_type) |>
  count(iucr) |>
  filter(n > 1)
```

Empty data.table (0 rows and 2 cols): iucr,n

No. It looks like that one edit to criminal sexual assault was the only one that needed fixing there. So it is just `description` that needs attention.

When dealing with IUCR 0263, I decided to use the spelling convention that was used most recently. Let's see if we can find the most recent spelling convention for each IUCR code and use that to harmonize `description`. We are going to make our own lookup table from the existing data.

```
iucrLookup <- crime |>
  group_by(iucr) |>
  filter(date == max(date)) |>
  select(iucr, fbi_code, primary_type, description) |>
  distinct() |>
  ungroup()
```

```
# are all IUCR codes unique?
any(duplicated(iucrLookup$iucr))
```

```
[1] FALSE
```

```
head(iucrLookup)
```

```
# A tibble: 6 x 4
  iucr fbi_code primary_type description
<chr> <chr>    <chr>      <chr>
1 0890 06      THEFT      FROM BUILDING
2 0710 06      THEFT      THEFT FROM MOTOR VEHICLE
3 0610 05      BURGLARY   FORCIBLE ENTRY
4 0580 08A     STALKING   SIMPLE
5 0810 06      THEFT      OVER $500
6 1310 14      CRIMINAL DAMAGE TO PROPERTY
```

Let's see whether the edits generally make sense

```

crime |>
  select(iucr, description) |>
  left_join(iucrLookup |>
    select(iucr, description),
    by = "iucr",
    # add _recent to new columns
    suffix = c("", "_recent")) |>
  filter(description != description_recent) |>
  distinct()

```

	iucr		description
	<char>		<char>
1:	0453	AGGRAVATED PO:	OTHER DANGEROUS WEAPON
2:	141A		UNLAWFUL USE HANDGUN
3:	0430	AGGRAVATED:	OTHER DANGEROUS WEAPON
4:	1210		THEFT OF LABOR/SERVICES
5:	502R		VEHICLE TITLE/REG OFFENSE

186:	0841	FINANCIAL IDENTITY THEFT:	\$300 & UNDER
187:	0510	AGG RIT MUT:	HANDS/FIST/FEET SERIOUS INJURY
188:	0820		\$300 AND UNDER
189:	0810		OVER \$300
190:	1521	KEEP PLACE OF JUV	PROSTITUTION
			description_recent
			<char>
1:		AGGRAVATED POLICE OFFICER -	OTHER DANGEROUS WEAPON
2:			UNLAWFUL USE - HANDGUN
3:		AGGRAVATED -	OTHER DANGEROUS WEAPON
4:			THEFT OF LABOR / SERVICES
5:		VEHICLE TITLE /	REGISTRATION OFFENSE

186:		FINANCIAL ID THEFT:	\$300 & UNDER
187:		AGG. RITUAL MUTILATION -	HANDS, FISTS, FEET, SERIOUS INJURY
188:			\$500 AND UNDER
189:			OVER \$500
190:		KEEPING PLACE OF JUVENILE	PROSTITUTION

They generally look like inconsequential spelling changes. However, note that some of them indicate changes in statutes, such as changes in theft limits from \$300 to \$500.

```

crime |>
  select(iucr, description) |>
  left_join(iucrLookup |>
    select(iucr, description),
    by = "iucr",
    # add _recent to new columns
    suffix = c("", "_recent")) |>
  filter(description != description_recent &
    grepl("\\$", description)) |>
  distinct()

```

	iucr <char>	description <char>	description_recent <char>
1:	0840	FINANCIAL IDENTITY THEFT: OVER \$300	FINANCIAL ID THEFT: OVER \$300
2:	0841	FINANCIAL IDENTITY THEFT: \$300 & UNDER	FINANCIAL ID THEFT:\$300 &UNDER
3:	0820	\$300 AND UNDER	\$500 AND UNDER
4:	0810	OVER \$300	OVER \$500

There are two routes to go here.

1. Use the lookup table to copy the most recent spelling conventions into the `crime` table
2. Delete `fbi_code`, `primary_type`, and `description` from `crime` and use `iucrLookup` whenever we need those details

We are going to go with option 2. One of the side benefits of this option is that it will reduce the size of the `crime` object in memory. We can also drop `year` since that information is already in `date`. In total, this will free up 229.8 Mb.

```

crime <- crime |>
  select(-c(fbi_code, primary_type, description, year))

```

i Exercises

1. What are the 10 most common crimes? Use `primary_type`
2. Which FBI code has the most IUCR codes associated with it? What are the primary types associated with that FBI code?
3. Which primary types result in the highest arrest rate? Give the top 10 primary types with at least 1000 incidents
4. Which `district` has the most thefts?

- Count the number of assaults since 2016 that have occurred on Fridays and Saturdays after 6pm. Report the assault counts by date, day of week, hour of the day, and year.

4.4 Coordinates

Missing coordinates cannot be placed on a map. Implausible coordinates can place a Chicago incident in another state, near the equator, or at (0, 0). For some cities that I have worked with, they mark their missing coordinates as (0, 0), at their city hall, and, in one case, Cinderella's Castle at Disney World. Let's see what happens in Chicago.

By clicking on Google Maps with Chicago in view, you can get a general idea what sensible coordinates would be for Chicago. Longitude is generally between -87.95 and -87.50 and latitude is generally between 41.60 and 42.05.

What are these records that are outside the Chicago area?

```
crime |>
  filter(!between(latitude, 41.60, 42.05) |
         !between(longitude, -87.95, -87.50))
```

	id	case_number		date		block	iucr
	<int>	<char>		<POS>		<char>	<char>
1:	12852880	JF425857	2022-10-07	15:55:00	0000X E CONGRESS PKWY	1150	
2:	12848210	JF420062	2022-10-02	19:00:00	001XX W CONGRESS PKWY	0320	
3:	11889657	JC508175	2019-11-12	18:22:00	0000X W IDA B WELLS DR	1821	
4:	11732417	JC318144	2019-06-23	09:03:00	009XX N CHRISTIANA AVE	0486	
5:	9801968	HX451303	2014-10-01	17:00:00	019XX S CANALPORT AVE	0820	

145:	937	G411262	2001-07-15	00:34:00	030XX S HARDING ST	0110	
146:	838	G311269	2001-05-29	23:35:00	059XX S MORGAN AV	0110	
147:	808	G259321	2001-05-06	01:30:00	020XX W 55 ST	0110	
148:	724	G159652	2001-03-20	18:17:00	067XX S JEFFERY AV	0110	
149:	637	G005960	2001-01-06	10:35:00	014XX N HARDING ST	0110	
	location_description	arrest	domestic	beat	district	ward	community_area
	<char>	<lgcl>	<lgcl>	<char>	<char>	<char>	<char>
1:	HOTEL / MOTEL	FALSE	FALSE	0113	001	4	32
2:	CTA TRAIN	FALSE	FALSE	0122	001	4	32
3:	STREET	TRUE	FALSE	0111	001	4	32
4:	OTHER	FALSE	TRUE	1121	011	27	23
5:	SIDEWALK	FALSE	FALSE	1235	012	25	31

```

---
145:          STREET FALSE FALSE 1031 010 <NA> <NA>
146:          DUMPSTER TRUE FALSE 0712 007 <NA> <NA>
147:          AUTO TRUE FALSE 0915 009 <NA> <NA>
148:          APARTMENT TRUE FALSE 0331 003 <NA> <NA>
149:          STREET TRUE FALSE 2535 025 <NA> <NA>
      latitude longitude
      <num>      <num>
1: 36.61945 -91.68657
2: 36.61945 -91.68657
3: 36.61945 -91.68657
4: 36.61945 -91.68657
5: 36.61945 -91.68657
---
145: 36.61945 -91.68657
146: 36.61945 -91.68657
147: 36.61945 -91.68657
148: 36.61945 -91.68657
149: 36.61945 -91.68657

```

```

crime |>
  select(longitude, latitude) |>
  filter(!between(latitude, 41.60, 42.05) |
         !between(longitude, -87.95, -87.50)) |>
  distinct()

```

```

      longitude latitude
      <num>      <num>
1: -91.68657 36.61945

```

Interesting. Over 100 cases are located outside Brandsville, Missouri, about 500 miles away from Chicago. From the addresses we can see that they have what seem to be Chicago addresses. In a later lesson we will cover geocoding, where you can take addresses and look up their longitude and latitude. That would be the right thing to do here. I save those skills for a later lesson and, for now, just flag cases that are clearly outside of Chicago or have no coordinates at all.

```

crime <- crime |>
  mutate(valid_coordinates =
    !is.na(latitude) &
    !is.na(longitude) &

```

```

# put a generous bounding box around Chicago
between(latitude, 41.60, 42.05) &
between(longitude, -87.95, -87.50))

crime |>
  count(valid_coordinates) |>
  mutate(percent = 100 * n / sum(n))

```

	valid_coordinates	n	percent
	<lgcl>	<int>	<num>
1:	FALSE	97308	1.130843
2:	TRUE	8507599	98.869157

```

crime |>
  filter(valid_coordinates) |>
  summarize(min_latitude = min(latitude, na.rm = TRUE),
            max_latitude = max(latitude, na.rm = TRUE),
            min_longitude = min(longitude, na.rm = TRUE),
            max_longitude = max(longitude, na.rm = TRUE))

```

	min_latitude	max_latitude	min_longitude	max_longitude
1	41.64459	42.02291	-87.93973	-87.52453

And let's do a quick check to see if these plotted points look like Chicago.

```

crime |>
  filter(valid_coordinates) |>
  slice_sample(n = 10000) |>
  ggplot(aes(x = longitude, y = latitude)) +
  geom_point(size = 0.5, alpha = 0.5) +
  coord_fixed() +
  labs(x = "Longitude", y = "Latitude") +
  theme_minimal()

```

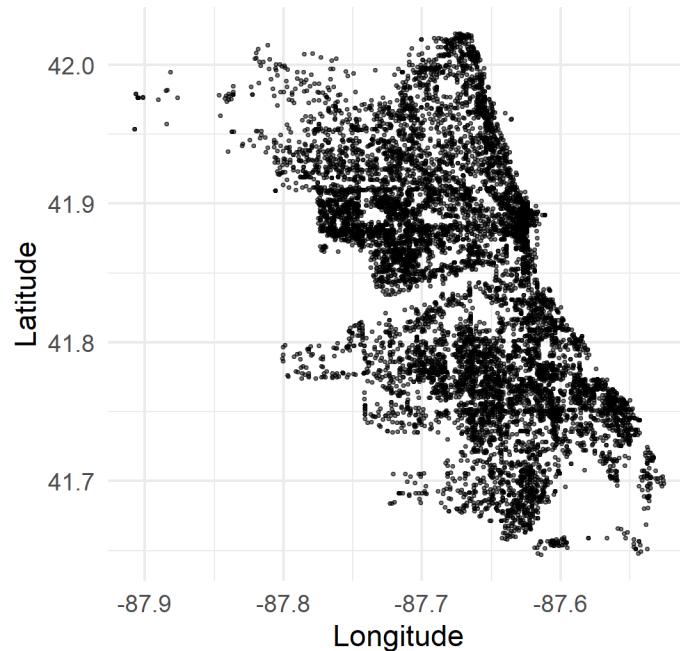


Figure 2: Basic plot of a sample of 10,000 Chicago crimes

i Exercises

6. Plot the longitude and latitude of all “ASSAULT”s for Ward 22.
7. What is the most common (longitude,latitude) for assaults in Ward 22? Add that point to the plot as a larger red dot.

Over many blocks of code we have run a lot of data cleanup steps to go from the original object we got from `fread()` to the cleaned up `crime` object we have now. Here is everything simplified to a single code block.

```
crime <- fread(
  "Crimes_-_2001_to_Present.csv.gz",
  select = c("ID", "Case Number", "Date", "Block", "IUCR",
    "Primary Type", "Description", "Location Description",
    "Arrest", "Domestic", "Beat", "District", "Ward",
    "Community Area", "FBI Code", "Year",
    "Latitude", "Longitude"),
  col.names = c("id", "case_number", "date", "block", "iucr",
    "primary_type", "description", "location_description",
    "arrest", "domestic", "beat", "district", "ward",
```

```

      "community_area", "fbi_code", "year",
      "latitude", "longitude"),
colClasses = list(character = c("IUCR", "FBI Code", "Beat", "District",
                                "Ward", "Community Area")),
na.strings = c("", "NA"),
showProgress = TRUE)

crime <- crime |>
  mutate(
    datetime0 = mdy_hms(date, tz = "UTC"), # UTC ignores DST

    # Correct records between 2am and 2:59am spring DST changes
    datetime0 = if_else(date(datetime0) %in% datesSpringDST &
                        hour(datetime0) == 2,
                        datetime0 + hours(1),
                        datetime0),

    date = force_tz(datetime0, tzone = "America/Chicago"),

    primary_type = replace_values(primary_type,
                                   "CRIM SEXUAL ASSAULT" ~ "CRIMINAL SEXUAL ASSAULT"),

    description = replace_values(description,
                                   "AGGRAVATED: KNIFE/CUT INSTR" ~
                                   "AGGRAVATED - KNIFE / CUTTING INSTRUMENT",
                                   "AGGRAVATED: KNIFE / CUTTING INSTRUMENT" ~
                                   "AGGRAVATED - KNIFE / CUTTING INSTRUMENT"),

    valid_coordinates = !is.na(latitude) &
                        !is.na(longitude) &
                        between(latitude, 41.60, 42.05) &
                        between(longitude, -87.95, -87.50)) |>
  select(-datetime0) |>
  relocate(date, .after = case_number)

iucrLookup <- crime |>
  group_by(iucr) |>
  filter(date == max(date)) |>
  select(iucr, fbi_code, primary_type, description) |>
  distinct() |>
  ungroup()

```

```
crime <- crime |>
  select(-fbi_code, -primary_type, -description, -year)
```

It is a good time to save our cleaned up data to a file so we can load it quickly in the future without having to repeat all of the cleanup steps.

```
save(crime, iucrLookup, file = "dataChicagoCrime.RData", compress=TRUE)
```

Whenever you need to work with this data again, you can load it quickly with `load("dataChicagoCrime.RData")`. This will recreate the `crime` and `iucrLookup` objects in your R session.

Now with our data cleaned up and our coordinates validated, we are ready to start mapping.

5 From incident records to a point map

Let's start small by creating a leaflet map just showing assaults in Ward 22.

```
crime |>
  filter(ward == 22 &
         valid_coordinates) |>
  semi_join(iucrLookup |>
            filter(primary_type == "ASSAULT"),
            by = "iucr") |>
  leaflet() |>
  setView(lng=-87.73, lat=41.83, # selected map's center
          zoom=13) |>          # zoom in to "neighborhood" level
  addTiles() |> # add the base street layer
  addCircleMarkers(~longitude, ~latitude,
                  radius=3,
                  stroke=FALSE,
                  fillOpacity = 1)
```

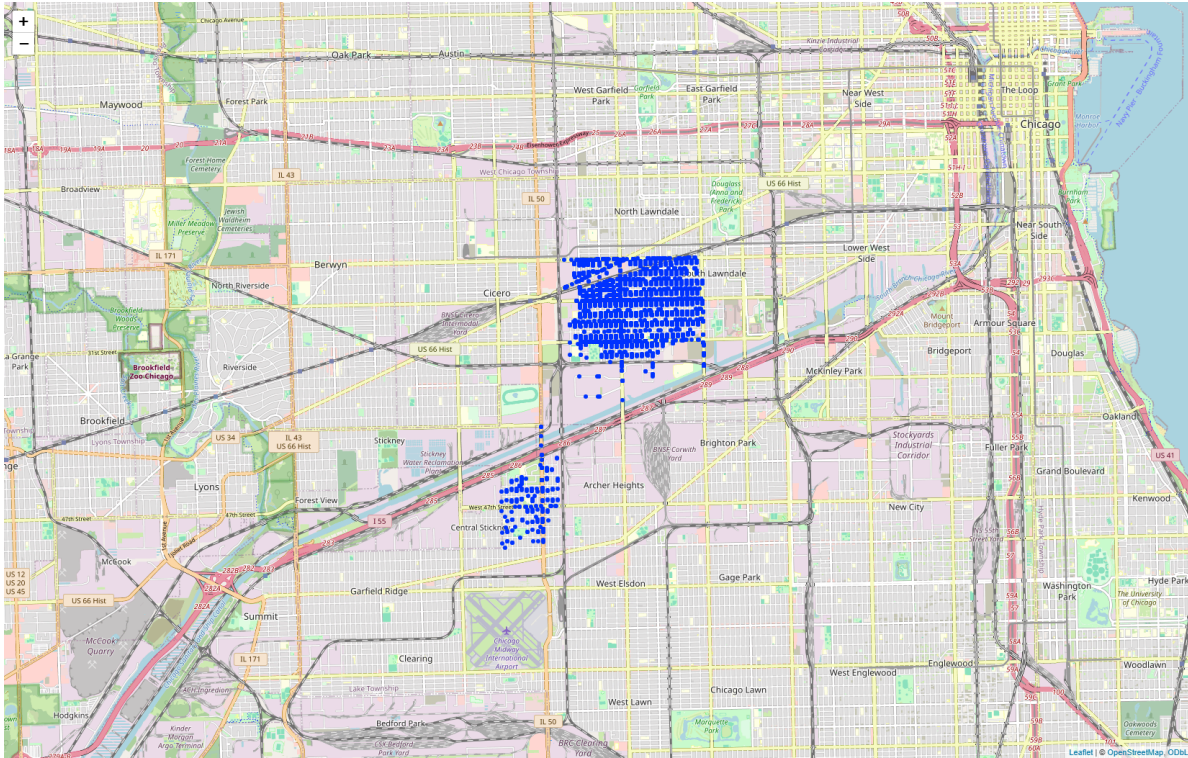


Figure 3: Reported assault locations in Chicago Ward 22

Leaflet has placed dots all over the map for ward 22. With so many dots, it is rather difficult to determine where exactly there is a higher crime density. It seems that crime is simply everywhere in ward 22.

Instead of showing all the dots, we are going to explore ways to color and highlight maps to reveal areas of higher crime density. Let's start by examining assaults in 2026 to date.

```
crime |>
  filter(valid_coordinates &
         year(date) == 2026) |>
  select(iucr, longitude, latitude) |>
  semi_join(iucrLookup |>
            filter(primary_type == "ASSAULT"),
            by = "iucr") |>
  leaflet() |>
  addProviderTiles(providers$CartoDB.Positron) |>
  setView(lng = -87.68, lat = 41.84, zoom = 10) |>
  addCircleMarkers(
    lng = ~longitude,
```



```
lat = ~latitude,
radius = 2,
stroke = FALSE,
fillOpacity = 0.25,
clusterOptions = markerClusterOptions())
```

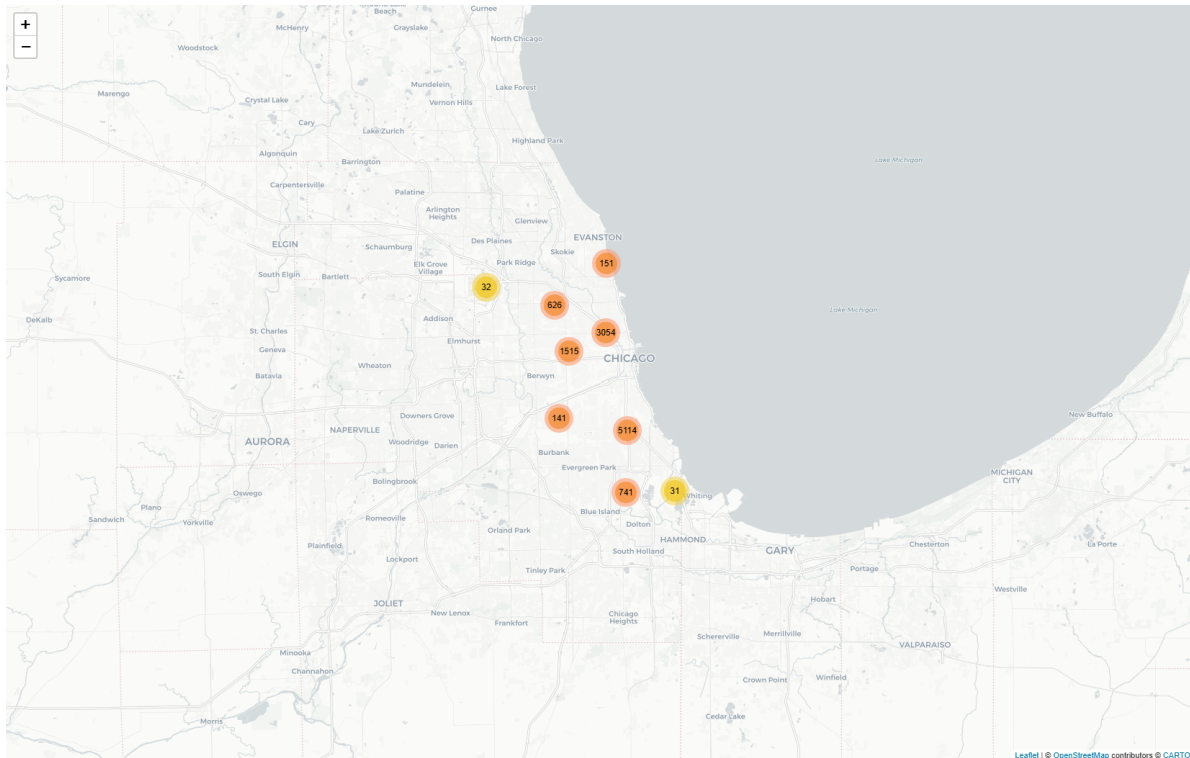


Figure 4: Clustered map of reported assaults in Chicago, 2026 to date

The clustered point map is useful for inspecting records and zooming to individual incidents. Also note that this leaflet map used CartoDB Positron base map instead of the default OpenStreetMap tiles. The CartoDB tiles can be a bit more subtle and less visually distracting than the default OpenStreetMap tiles.

i Exercise

8. Create a point map for motor vehicle theft in 2024. Compare the unclustered (by removing `clusterOptions = markerClusterOptions()`) and clustered versions and explain what each version hides

6 Hexagonal tiles and hotspot maps

One strategy for making a hotspot map is to chop up the Chicago map into a bunch of smaller tiles. For this we are going to overlay our leaflet map with hexagonal tiles. Then we will count how many dots land within each hexagonal tile and color the tile according to that count. Hexagonal bins are convenient because they produce smoother, more visually appealing density maps, especially when compared with square tiles. Each hexagon has a consistent distance to its neighbors and their geometry better approximates a circle, reducing distortion in how densities appear. This uniformity makes patterns in the underlying data easier to interpret, especially when mapping phenomena like crime incidents, where you want to highlight true spatial concentrations rather than artifacts of the grid shape.

This time we will make a hotspot map of all assaults in Chicago. First, we need to communicate to R that latitude and longitude are special geographic coordinates. Here we avoid saying too much about the `st_as_sf()` and the other functions from the `sf` package since we go into great detail about it in later parts of the course.

We will use a special coordinate system, the Universal Transverse Mercator coordinate system, for the part of the globe sharing Chicago's longitude area (CRS 26916 = UTM zone 16N) to avoid distortion (because we are projecting points on a round earth to a two-dimensional screen). The problem with using latitude and longitude coordinates is that differences in coordinates are measured in degrees rather than meters. At the equator 0.01° longitude is about 1 kilometer, but 0.01° gets shorter and shorter as you move toward the earth's poles. Therefore, all of our tiling and counting will be done in the UTM Zone 16N coordinate system that has distances measured in meters. After we have done all of our calculations, we will convert back to latitude and longitude to overlay our results on top of leaflet.

```
# convert to an sf object
#   tells R that Longitude and Latitude are special columns
sfAssaults <- crime |>
  filter(valid_coordinates) |>
  semi_join(iucrLookup |>
    filter(primary_type == "ASSAULT"),
    by = "iucr") |>
  st_as_sf(coords = c("longitude", "latitude"),
    crs = 4326, # lat/long coordinate system (WGS 84)
    remove = FALSE) |> # don't remove the lat/long columns
  st_transform(crs = 26916) # project to UTM 16N, Chicago
```

`sfAssaults` is now an `sf` object, which is a special type of data frame that can store geographic information. The `geometry` column contains the point locations of each assault in UTM coordinates.

```
sfAssaults |>
  head()
```

Simple feature collection with 6 features and 15 fields

Geometry type: POINT

Dimension: XY

Bounding box: xmin: 436803.1 ymin: 4625059 xmax: 449446.9 ymax: 4639665

Projected CRS: NAD83 / UTM zone 16N

	id	case_number		date		block	iucr
1	14271221	JK344593	2026-07-23	00:00:00		063XX S STATE ST	0530
2	14270842	JK344082	2026-07-22	22:45:00		013XX N HUDSON AVE	0560
3	14270920	JK344083	2026-07-22	22:10:00		065XX S LANGLEY AVE	0560
4	14272620	JK346369	2026-07-22	21:30:00		0000X N WACKER DR	0560
5	14270749	JK344012	2026-07-22	21:16:00		005XX S LOTUS AVE	0560
6	14270761	JK343925	2026-07-22	19:52:00		040XX W ROOSEVELT RD	0560

	location_description	arrest	domestic	beat	district	ward	community_area
1	VEHICLE - COMMERCIAL	FALSE	FALSE	0312	003	20	69
2	APARTMENT	FALSE	FALSE	1821	018	27	8
3	APARTMENT	FALSE	FALSE	0321	003	20	42
4	STREET	FALSE	FALSE	0122	001	42	32
5	APARTMENT	FALSE	FALSE	1522	015	29	25
6	APARTMENT	FALSE	TRUE	1011	010	24	29

	latitude	longitude	valid_coordinates		geometry
1	41.77980	-87.62538	TRUE	POINT	(448029.4 4625517)
2	41.90715	-87.63968	TRUE	POINT	(446946.1 4639665)
3	41.77576	-87.60828	TRUE	POINT	(449446.9 4625059)
4	41.88248	-87.63703	TRUE	POINT	(447145.8 4636924)
5	41.87358	-87.76158	TRUE	POINT	(436803.1 4636021)
6	41.86615	-87.72678	TRUE	POINT	(439683.2 4635170)

Next, we will create the set of hexagonal tiles that cover the assault locations. We need to set the size of the hexagons, specifically the distance between opposite edges of each hexagon. First, check what units of distance R will use with these data.

```
st_crs(sfAssaults)$units
```

```
[1] "m"
```

Confirming that the distance measurements are in meters, let's set the hexagon size to 0.5 km or 500 meters. After that, we can use `st_make_grid()` to make an R object (an `sf` object)

that creates the hexagons and stores them in a convenient format. The cell width is an analytic choice. Small cells show local detail but produce unstable rates. Large cells are more stable but can conceal small hotspots. This is known as a scale or zoning problem. A hotspot can appear, disappear, or move when the grid changes.

```
hex_width <- 500
# create hexagonal tiles
sfGrid <-
  st_make_grid(sfAssaults,
               cellsize = hex_width,
               square = FALSE) |> # prefer hexagons
  st_as_sf() |>
  rename(geometry = x)

sfGrid
```

Simple feature collection with 6762 features and 0 fields

Geometry type: POLYGON

Dimension: XY

Bounding box: xmin: 422095.9 ymin: 4610178 xmax: 456845.9 ymax: 4652757

Projected CRS: NAD83 / UTM zone 16N

First 10 features:

```
              geometry
1 POLYGON ((422345.9 4610611,...
2 POLYGON ((422345.9 4611477,...
3 POLYGON ((422345.9 4612343,...
4 POLYGON ((422345.9 4613209,...
5 POLYGON ((422345.9 4614075,...
6 POLYGON ((422345.9 4614941,...
7 POLYGON ((422345.9 4615807,...
8 POLYGON ((422345.9 4616673,...
9 POLYGON ((422345.9 4617539,...
10 POLYGON ((422345.9 4618405,...
```

`sfGrid` is also like a regular data frame with one column containing polygon objects as each row's data element. There is also a header that gives some summary information like how many hexagons there are in `sfGrid` (6762) and its coordinate system (NAD83 / UTM zone 16N). Let's check our work so far by displaying the hexagons over a leaflet map of Chicago, this time using the ESRI World Imagery base map.

```

leaflet() |>
  addProviderTiles(providers$Esri.WorldImagery) |>
  # zoom=11-> tight city level view
  setView(lng = -87.8, lat = 41.85, zoom = 10) |>
  addPolygons( # convert coordinates back to lat/long (CRS 4326)
    data = sfGrid |> st_transform(crs=4326),
    fillOpacity = 0, # don't color in the hexagons
    color = "darkgray", # border of hexagons
    weight = 2, # border thickness
    opacity = 1) # border transparency

```



Figure 5: A 500-meter hexagonal grid covering Chicago assault locations

We are making progress. We have covered the map of Chicago with 6762 hexagons. Some of our hexagons are tiling parts of Lake Michigan. We will filter those out later.

Now we need to count how many of the assaults land in each of these hexagonal tiles. `st_intersects()` will figure out which points in `sfAssaults` land inside each hexagon and `lengths()` will count their number. We will also compute the crime density (incidents per

square kilometer per year), eliminate any hexagons with 0 assaults (like all those in Lake Michigan), and project coordinates back to latitude/longitude, which leaflet requires.

We will need to know the number of years of data we have so we can normalize results to a “per year” rate.

```
nYears <- sfAssaults |>
  st_drop_geometry() |> # ignore the geographic info
  summarize(timespan = difftime(max(date, na.rm=TRUE),
                                min(date, na.rm=TRUE),
                                units = "days"),
            timespan = as.numeric(timespan / 365.25)) |>
  pull(timespan)
nYears
```

```
[1] 25.55499
```

`st_intersects()`, when used with polygon shapes (`sfGrid`) and points (`sfAssaults`), returns for each polygon a list of the indices of points that the polygon intersects. Here is what the results of `st_intersects()` look like for 10 hexagons plucked from the middle of Chicago. For some hexagons, the list is empty, signalling that no assaults landed in that hexagon. For other hexagons we see lists of points. These are the row numbers from `sfAssaults` of points that landed in that specific hexagonal tile.

```
sfGrid |>
  slice(3480:3489) |>
  st_intersects(sfAssaults)
```

Sparse geometry binary predicate list of length 10, where the predicate was ``intersects'`

```
1: (empty)
2: (empty)
3: (empty)
4: (empty)
5: (empty)
6: 6439, 31127, 34962, 36511, 40392, 44716, 47738, 53866, 60227, 63494, ...
7: 9939, 10973, 16622, 18586, 27371, 44668, 53151, 54078, 68902, 76387, ...
8: 5456, 33826, 35637, 41450, 51604, 89964, 112598, 118663, 133702, 181309, ...
9: 15381, 32116, 32702, 37333, 72709, 74311, 78181, 79091, 88412, 109323, ...
10: 210805, 436483, 445774
```

If we apply `lengths()` to this list it will return the number of points in each of these lists.

```
sfGrid |>
  slice(3480:3489) |>
  st_intersects(sfAssaults) |>
  lengths()
```

```
[1] 0 0 0 0 0 0 95 102 35 39 3
```

Now for all of our hexagons covering Chicago let's compute the number of assaults per km² per year.

```
sfGrid <- sfGrid |>
  # find which dots each hex intersects...
  mutate(nAssaults = geometry |>
    st_intersects(sfAssaults) |>
    lengths(), # ...and count them
  # get area in square kilometers
  areakm2 = as.numeric(st_area(geometry)) / 10^6,
  # compute incidents per km2 per year
  density = nAssaults / areakm2 / nYears) |>
  filter(nAssaults > 0) |> # drop those that are empty
  st_transform(crs=4326) # transform back to lat/long for leaflet

sfGrid
```

Simple feature collection with 2674 features and 3 fields

Geometry type: POLYGON

Dimension: XY

Bounding box: xmin: -87.94064 ymin: 41.64168 xmax: -87.51863 ymax: 42.02491

Geodetic CRS: WGS 84

First 10 features:

	geometry	nAssaults	areakm2	density
1	POLYGON ((-87.93757 41.9944...	1	0.2165064	0.1807398
2	POLYGON ((-87.93472 42.0061...	1	0.2165064	0.1807398
3	POLYGON ((-87.93153 41.9944...	1	0.2165064	0.1807398
4	POLYGON ((-87.92789 41.9516...	1	0.2165064	0.1807398
5	POLYGON ((-87.928 41.9594, ...	1	0.2165064	0.1807398
6	POLYGON ((-87.92846 41.9905...	1	0.2165064	0.1807398
7	POLYGON ((-87.92538 41.9867...	4	0.2165064	0.7229591
8	POLYGON ((-87.92561 42.0023...	3	0.2165064	0.5422193

```

9 POLYGON ((-87.92186 41.9516...      1 0.2165064 0.1807398
10 POLYGON ((-87.92197 41.9594...      1 0.2165064 0.1807398

```

Now `sfGrid` has three new columns associating with each hexagon an assault count, the area of the hexagon (all should be identical), and the assault density.

We want to break up that density into bins and assign a color to each bin. This way we can color-code each hexagon based on the assault density. `colorBin()` creates a function that will take a density value and return a color. The combination of `bins = 9` and `pretty = TRUE` will create some nice intervals for the different colors. Viridis is a family of color palettes that has been engineered to be perceptually uniform (equal steps in data values look like equal steps in color), colorblind-friendly, printer-friendly (maintains contrast in grayscale), and readable on screens and projectors. In its default direction, the viridis palette runs from dark purple for low values to yellow for high values. Therefore, the highest-density hexagons on this map will be the lightest and yellowest, while the lowest-density hexagons will be dark purple.

```

pal <- colorBin("viridis",
               domain = sfGrid$density,
               bins = 9, # with pretty(), might not be exactly 9
               pretty = TRUE) # choose "nice" breakpoints

```

So what is this `pal()` function? If we give `pal()` a number it will produce a color code.

```
pal(125)
```

```
[1] "#3B528B"
```

`pal(125)` produces the color code `#3B528B`, which gives the hexadecimal code for mixing the primary source colors red (3B), green (52), and blue (8B). That mix is like a deep indigo blue. When creating the hexagon overlay, each hexagon's density will be run through `pal()` which will determine how to color the hexagon. What breakpoints did `pal()` decide on?

```
pal |> attr("colorArgs")
```

```

$bins
[1] 0  50 100 150 200 250 300 350 400 450

$na.color
[1] "#808080"

```


The bins value shows that `pal()` will assign one color to 0-50, another color to 50-100, and so on up to the last bin 400-450. It also shows that if there happens to be any hexagons with missing values for density (there are in fact none), those hexagons will get colored with #808080, which is gray.

Let's make our leaflet map! This time I will use a dark base layer.

```
leaflet() |>
  addProviderTiles(providers$CartoDB.DarkMatterNoLabels) |>
  setView(lng = -87.8, lat = 41.85, zoom = 10) |>
  addPolygons(
    data = sfGrid,
    fillColor = ~pal(density),
    fillOpacity = 0.5,
    color = "", # no borders between hexagons
    # make popup box when hovering
    label = ~paste("Count:",
                    format(nAssaults, big.mark=","),
                    "<br>Density: ",
                    format(round(density,1), nsmall=1),
                    "/ km2 / year") |>
    lapply(htmltools::HTML)) |> # signal HTML so <br> is linebreak
  addLegend(position = "bottomright",
            pal = pal,
            values = sfGrid$density,
            title = "Incidents / km2 / year",
            opacity = 0.7,
            labFormat = labelFormat()) # use pal() to figure out format
```

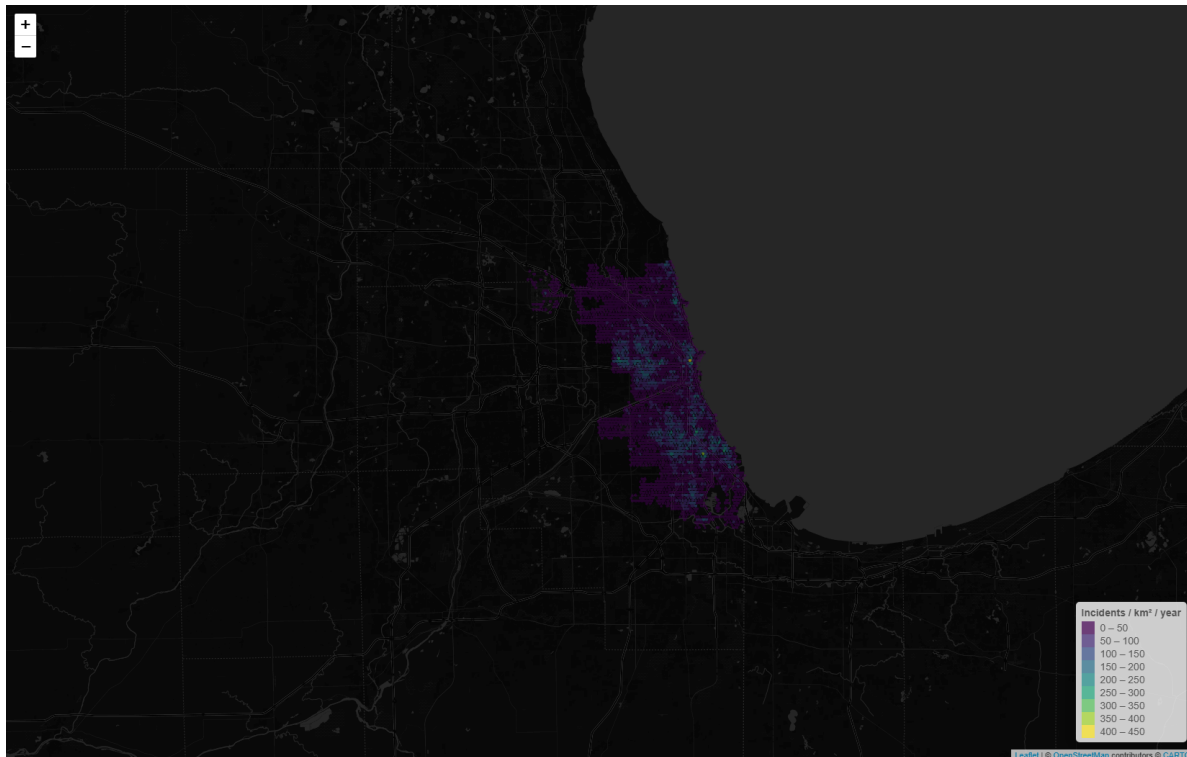



Figure 6: Reported assault density in 500-meter hexagons, Chicago

You can zoom in on specific neighborhoods or specific hexagons. You can also hover your mouse over a hexagon to see the assault density.

You can try to make your own color palettes, like

```
palGreg <- colorNumeric(
  palette= c("springgreen","hotpink"),
  domain = c(0,450))

leaflet() |>
  addTiles() |>
  setView(lng=-87.8, lat=41.85,zoom= 10) |>
  addPolygons(
    data = sfGrid,
    fillColor=~palGreg(density),
    fillOpacity=0.5,
    color="darkgray",
    weight=0.5,
    opacity=0.8,
```

```

label=~paste("Count:",
             format(nAssaults, big.mark=","),
             "<br>Density:",
             format(round(density,1),nsmall=1),
             "/km2/year") |>
  lapply(htmltools::HTML),
  highlightOptions = highlightOptions(color= "red",
                                       weight= 2,
                                       bringToFront= FALSE)) |>

addLegend(position="bottomright",
  pal=palGreg,
  values=sfGrid$density,
  title="Incidents/km2/year",
  opacity=0.7,
  labFormat=labelFormat())

```

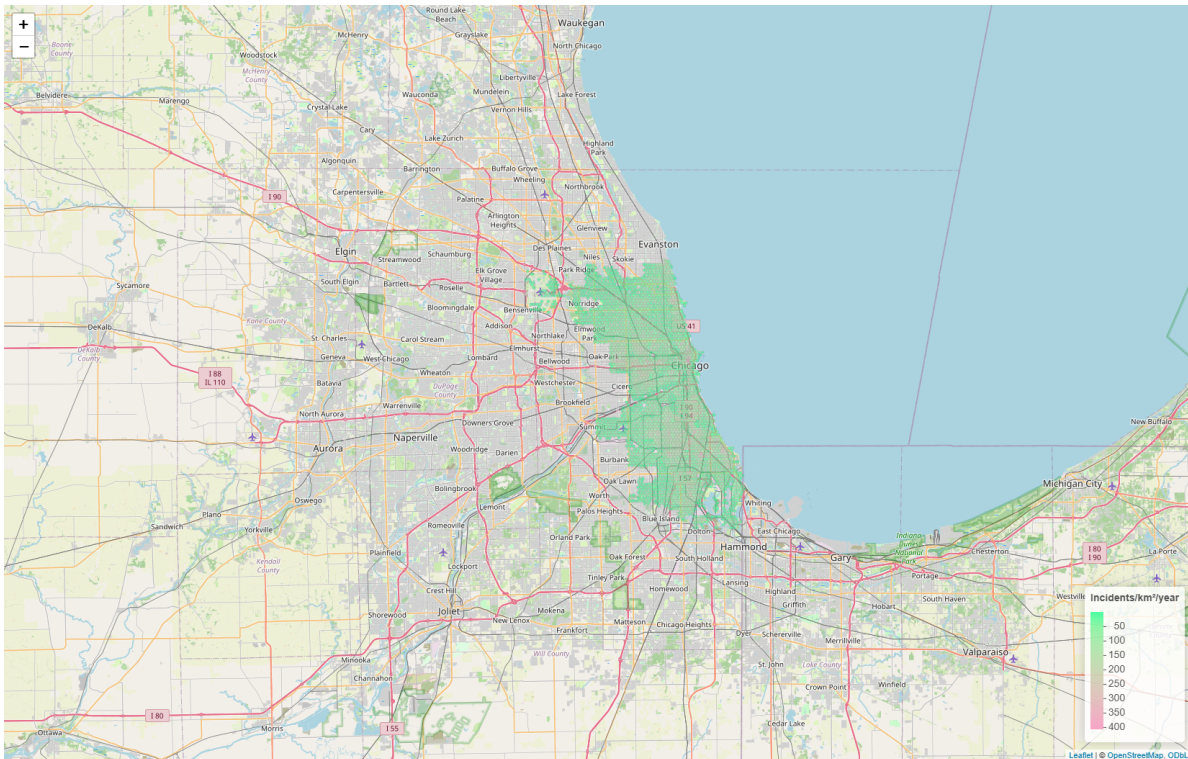


Figure 7: Reported assault density using a poorly differentiated green-to-pink palette

Yikes! I have made awful color choices here. The gradation of the palette is not sufficiently varied for us to see where the hotspots are, except for the few hexagons colored hot pink. I find

it best to rely on the built-in color palettes that have been more thoughtfully constructed.

6.1 A static, publication-friendly map

While leaflet maps are more engaging, you will often need static maps for your reports.

This map uses the magma palette in reverse. In its default direction, magma runs from nearly black for low values to pale yellow for high values. `direction = -1` reverses that mapping, making low densities light yellow and high densities dark purple. This keeps the highest hotspots from being confused with the white background outside the mapped hexagons.

The square-root transformation changes how density values are assigned to positions along the color scale. Compared with a linear scale, taking the square root spreads low and moderate values farther apart while compressing differences among high values. This makes variation outside the very highest-density hexagons easier to see. It does not change the density values themselves.

```
ggplot(sfGrid) +  
  geom_sf(aes(fill = density), color = NA) +  
  scale_fill_viridis_c(  
    option = "magma",  
    direction = -1, # light yellow low, dark purple high  
    # give more color resolution to low and moderate densities  
    transform = "sqrt",  
    name = "Assaults /\nkm2 / year") + # \n = line feed  
  coord_sf(datum = NA) +  
  theme_void()
```

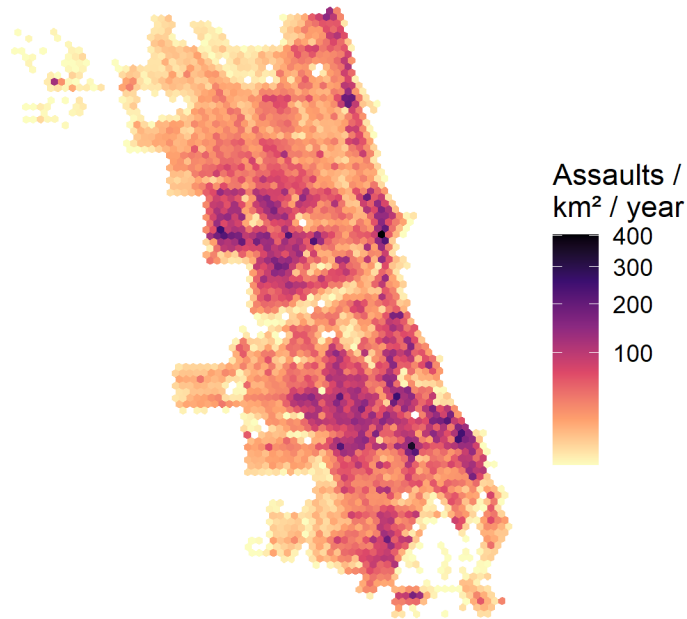


Figure 8: Reported assaults per square kilometer per year, Chicago

i Exercise

9. Rebuild the hexagonal map with 250-meter and 1,000-meter cells. Describe which features persist and which depend on cell size

7 Smooth hotspot contours

Hexagons make the aggregation areas visible. Kernel density estimation offers a smoothed version of the crime density.

7.1 Kernel density estimation

The hexagonal tiles are rather blocky. There is no smoothness to them where we would expect crime rates to be smoother. Kernel density estimation (KDE) transforms a set of incident locations into a smoothed estimate of an incident intensity, to highlight regions where incidents concentrate. KDE places a smooth bump, called a *kernel*, over each point and adds the heights of the bumps together at a large grid of points. The resulting surface estimates incident intensity across space.

Before applying KDE to a map, let's work through a one-dimensional example. Imagine a single street that is 6 kilometers long, with eight incidents at the locations marked below.

```
x <- c(1.0, 1.8, 2.0, 2.7, 3.1, 3.9, 4.2, 5.0)

plot(0, 0,
     type = "n",
     xlim = c(0, 6),
     ylim = c(0, 1),
     xlab = "Distance along street (km)",
     ylab = "",
     axes = FALSE)
axis(1)
rug(x, lwd = 2)
points(x, rep(0, length(x)), pch = 19)
```

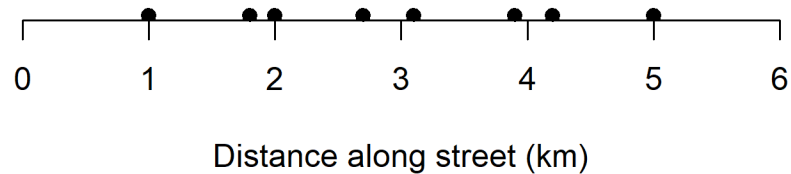


Figure 9: Eight incident locations along a six-kilometer street

Place a bell-shaped curve over each incident. This example uses a normal distribution, although other kernel shapes are possible. The `bandwidth` controls how widely each incident is spread across nearby locations. Each kernel has an area of one, so the total area under all eight kernels is eight incidents. Rather than treat the incident as occurring at one specific point, the kernels kind of smudge the incident so that the weight of any one incident is spread over nearby areas.

```
h <- 0.35
g <- seq(min(x) - 3 * h, max(x) + 3 * h, length.out = 1000)
```

```

kernels <- sapply(x, function(xi) dnorm((g - xi) / h) / h)

plot(0, 0,
     type = "n",
     xlim = c(0, 6),
     ylim = c(0, 2.5),
     xlab = "Distance along street (km)",
     ylab = "Kernel height")
rug(x, lwd = 2)
points(x, rep(0, length(x)), pch = 19)

apply(kernels, 2,
     function(y) lines(g, y,
                       col = rgb(0.2, 0.2, 0.2, 0.35),
                       lwd = 1.5)) |>
invisible() # don't print any text, just draw lines

```

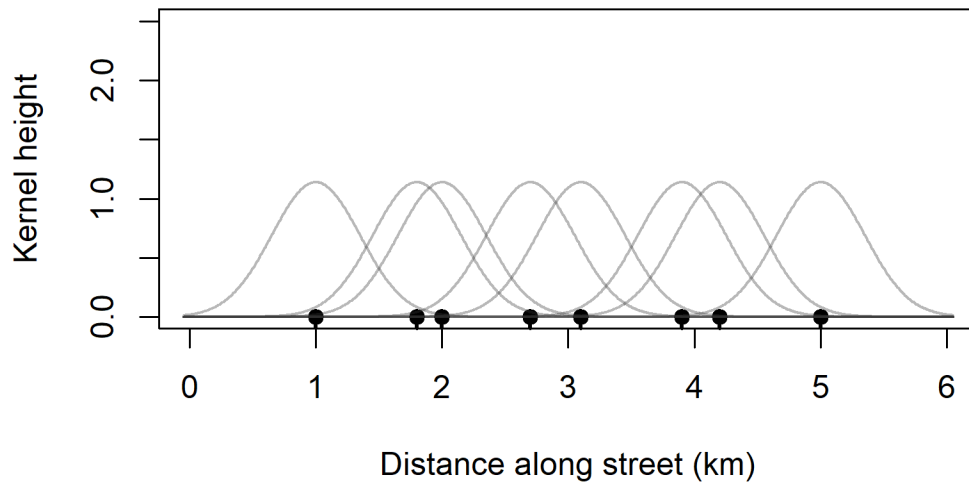


Figure 10: Normal kernels centered on eight incident locations

To estimate incident intensity at a particular location, add the heights of all eight kernels at that location. The translucent red points below show the eight contributions at the 2.5-kilometer marker (only four of them have non-negligible values). The solid red point shows their sum.

```

plot(0, 0,
     type = "n",
     xlim = c(0, 6),
     ylim = c(0, 2.5),
     xlab = "Distance along street (km)",
     ylab = "Incidents / km")
rug(x, lwd = 2)
points(x, rep(0, length(x)), pch = 19)

apply(kernels, 2,
      function(y) lines(g, y,
                        col = rgb(0.2, 0.2, 0.2, 0.35),
                        lwd = 1.5)) |>
invisible()

kde_at_25 <- which.min(abs(g - 2.5))
points(rep(2.5, length(x)),
       kernels[kde_at_25, ],
       col = rgb(1, 0, 0, 0.35),
       pch = 19)
points(2.5,
       sum(kernels[kde_at_25, ]),
       col = "red",
       pch = 19)

```

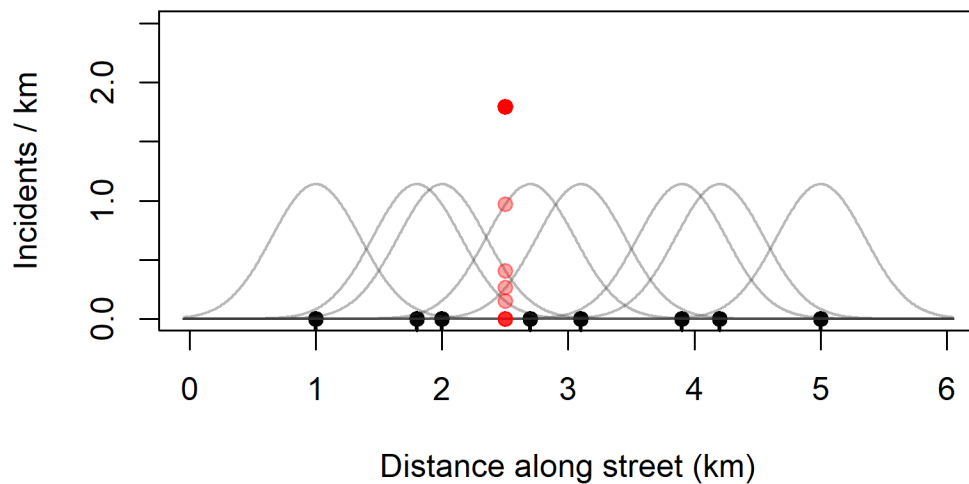


Figure 11: Individual kernel contributions and their sum at the 2.5-kilometer marker

At 2.5 kilometers, the estimated intensity is 1.8 incidents per kilometer. The units come from the kernels. Each kernel has an area of one incident and its horizontal axis is measured in kilometers.

Repeat the calculation along the entire street to obtain the blue KDE curve.

```
kde <- rowSums(kernels)

plot(0, 0,
     type = "n",
     xlim = c(0, 6),
     ylim = c(0, 2.5),
     xlab = "Distance along street (km)",
     ylab = "Incidents / km")
rug(x, lwd = 2)
points(x, rep(0, length(x)), pch = 19)

apply(kernels, 2,
     function(y) lines(g, y,
                       col = rgb(0.2, 0.2, 0.2, 0.25),
                       lwd = 1.5)) |>
```



```
invisible()
lines(g, kde, col = "#2C7BB6", lwd = 3)
```

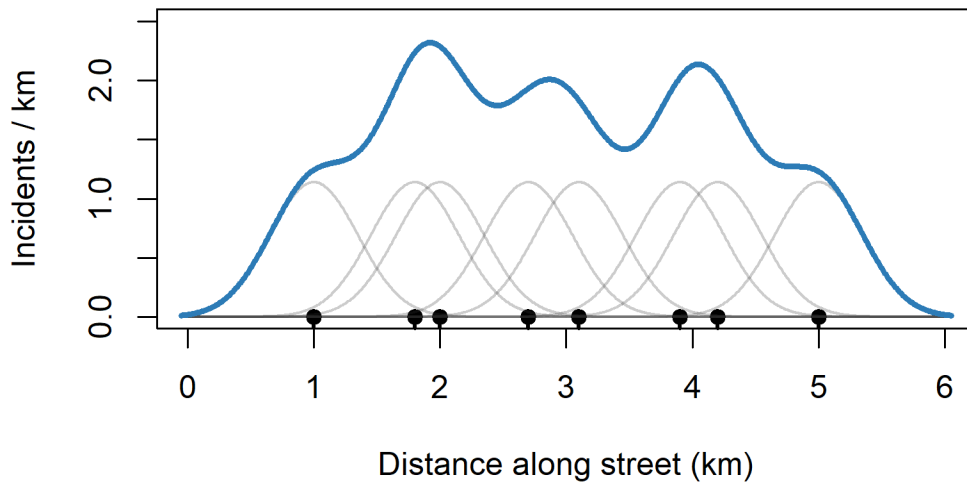


Figure 12: The kernel density estimate obtained by summing the individual kernels

Selecting a bandwidth requires judgment. A bandwidth that is too small produces an unstable, jagged estimate. A bandwidth that is too large smooths away potentially meaningful concentrations and can merge separate hotspots. The plot below compares three bandwidths using the same incidents.

```
plot(0, 0,
     type = "n",
     xlim = c(0, 6),
     ylim = c(0, 4.5),
     xlab = "Distance along street (km)",
     ylab = "Incidents / km")
rug(x, lwd = 2)
points(x, rep(0, length(x)), pch = 19)

bandwidths <- c(0.10, 0.35, 1.00)
bandwidth_colors <- c("#E66101", "#2C7BB6", "#4DAF4A")
```

```

for (j in seq_along(bandwidths))
{
  h_j <- bandwidths[j]
  kernels_j <- sapply(x, function(xi) dnorm((g - xi) / h_j) / h_j)
  lines(g,
        rowSums(kernels_j),
        col = bandwidth_colors[j],
        lwd = 3)
}

legend("topright",
      legend = paste0("h = ", format(bandwidths, nsmall = 2)),
      fill = bandwidth_colors,
      border = NA,
      bty = "n",
      title = "Bandwidth (km)")

```

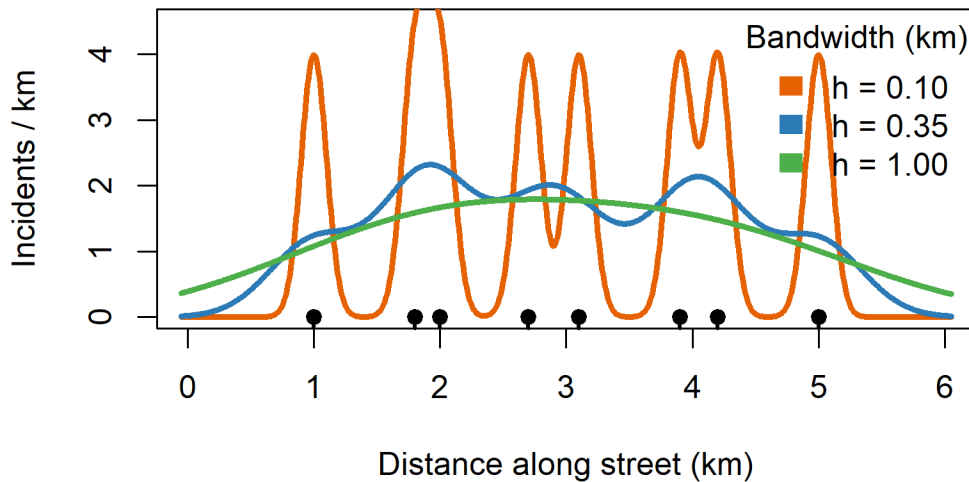


Figure 13: Kernel density estimates using three bandwidths

The height of the KDE curve can be translated into a color. Here, each horizontal band represents a range of incident intensities and the color that would be assigned to that range.

```

kde_breaks <- pretty(c(0, max(kde)), n = 5)
pal_1d_kde <- colorBin(palette = "viridis",
                      domain = kde,
                      bins = kde_breaks)

plot(0, 0,
     type = "n",
     xlim = c(0, 6),
     ylim = range(kde_breaks),
     xlab = "Distance along street (km)",
     ylab = "Incidents / km")

for (j in seq_len(length(kde_breaks) - 1))
{
  midpoint <- mean(kde_breaks[c(j, j + 1)])
  rect(par()$usr[1], kde_breaks[j],
       par()$usr[2], kde_breaks[j + 1],
       col = adjustcolor(pal_1d_kde(midpoint), alpha.f = 0.5),
       border = NA)
}

rug(x, lwd = 2)
points(x, rep(0, length(x)), pch = 19)
lines(g, kde, col = "#2C7BB6", lwd = 3)

```

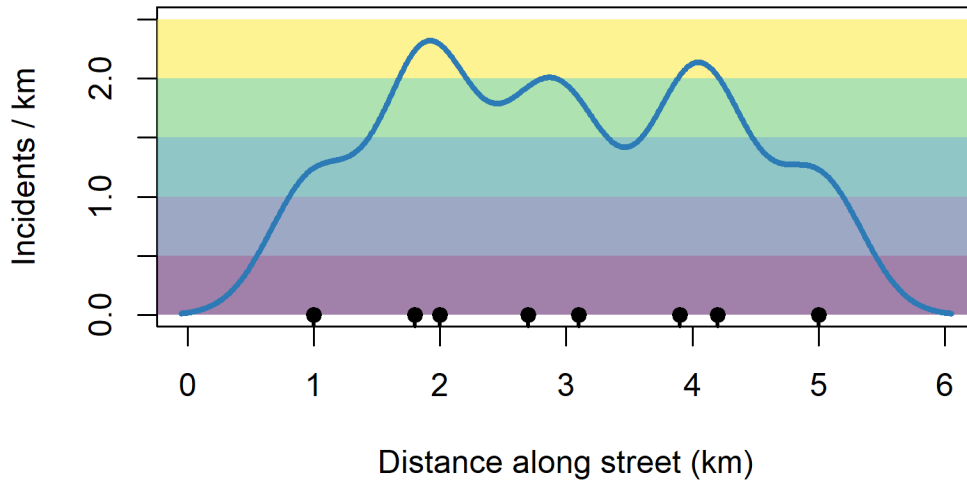


Figure 14: Mapping ranges of estimated incident intensity to colors

These breaks are approximately equally spaced in the original incident-intensity units. There is no reason to stick with that spacing. A square-root transformation gives more color resolution to low and moderate values and less resolution to the high end of the scale. To make four bands that are equally spaced on a square-root scale, we create equally spaced values between zero and the square root of the maximum, then square those values to return the breakpoints to the original units. The resulting bands are narrower at the low end and wider at the high end.

```
kde_breaks <- (seq(0, sqrt(max(kde)), length = 5))^2
pal_1d_kde <- colorBin(palette = "viridis",
                        domain = kde,
                        bins = kde_breaks)

plot(0, 0,
     type = "n",
     xlim = c(0, 6),
     ylim = range(kde_breaks),
     xlab = "Distance along street (km)",
     ylab = "Incidents / km")

for (j in seq_len(length(kde_breaks) - 1))
{
```

```

midpoint <- mean(kde_breaks[c(j, j + 1)])
rect(par()$usr[1], kde_breaks[j],
     par()$usr[2], kde_breaks[j + 1],
     col = adjustcolor(pal_1d_kde(midpoint), alpha.f = 0.5),
     border = NA)
}

rug(x, lwd = 2)
points(x, rep(0, length(x)), pch = 19)
lines(g, kde, col = "#2C7BB6", lwd = 3)

```

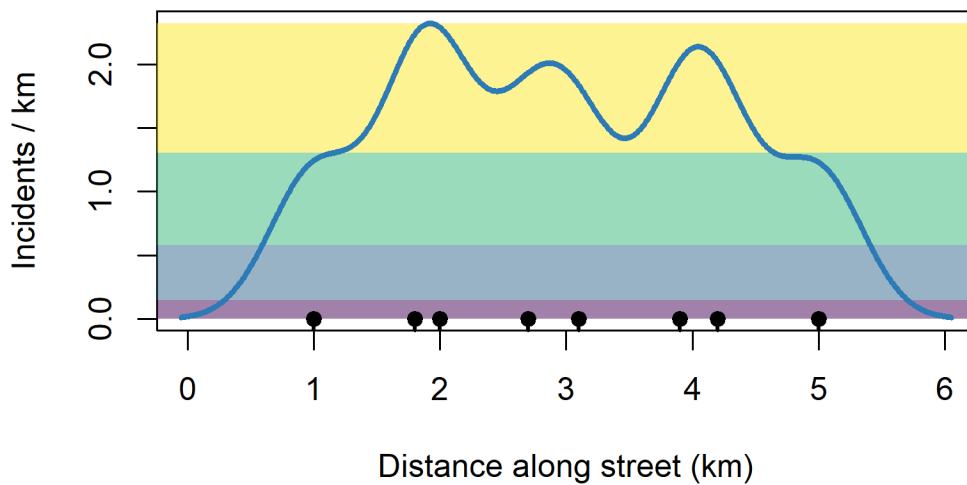


Figure 15: Mapping ranges of estimated incident intensity to colors, non-equal spacing

Finally, remove the curve and color the street itself by the estimated incident intensity. With the default direction of the viridis palette used here, the lighter yellow and green stretches have the highest estimated intensities and are the one-dimensional equivalent of hotspots on a map. The dark purple stretches have the lowest estimated intensities.

```

kde_groups <- cut(kde,
                  breaks = kde_breaks,
                  include.lowest = TRUE,
                  right = TRUE)

```

```

plot(0, 0,
     type = "n",
     xlim = c(0, 6),
     ylim = c(-0.15, 0.5),
     xlab = "Distance along street (km)",
     ylab = "",
     axes = FALSE)
axis(1)

segments(x0 = g[-length(g)],
         y0 = 0,
         x1 = g[-1],
         y1 = 0,
         col = pal_1d_kde(kde[-length(kde)]),
         lwd = 12,
         lend = "butt")
rug(x, lwd = 2)
points(x, rep(0, length(x)), pch = 19)

kde_midpoints <- 0.5*(kde_breaks[-1] + kde_breaks[-length(kde_breaks)])

legend("topright",
      legend = levels(kde_groups),
      fill = pal_1d_kde(kde_midpoints),
      border = NA,
      bty = "n",
      title = "Incidents / km")

```

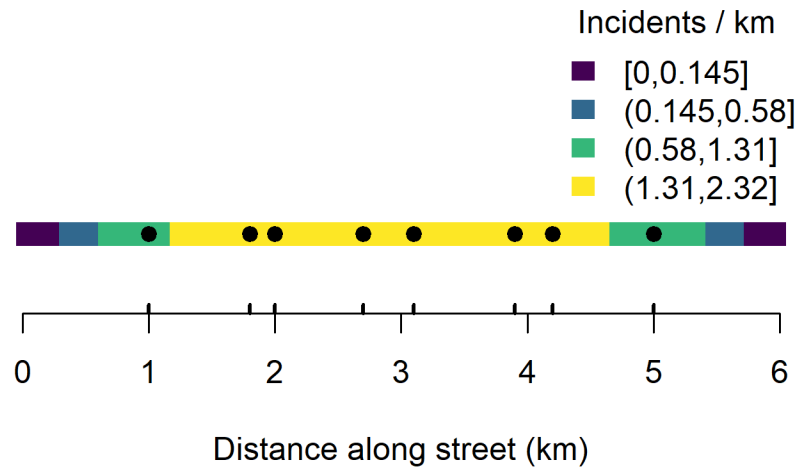


Figure 16: A street segment colored by estimated incident intensity

The two-dimensional calculation follows the same logic. Each incident receives a two-dimensional kernel, and the kernels are added across a grid of map locations. The resulting values are incident intensity per unit area instead of intensity per unit length.

7.2 KDE in R

The **MASS** package provides a two-dimensional KDE function `kde2d()`. However, the **MASS** package also has a `select()` function that clashes with **dplyr**'s `select()` function. This makes it either easy to cause bugs or confusing why an apparently simple line of code does not work. So we will not run `library(MASS)` and just prefix any calls to **MASS** functions with `MASS::` to avoid problems completely.

We will also use the **isoband** package. An *isoline*, also called a contour line, connects locations where the estimated incident intensity equals one specified value. An *isoband* is the area between two isolines, where the estimated intensity falls between a lower and an upper value. We will use isobands because they produce polygons that we can fill with colors representing ranges of incident intensity.

```
# for computing 2-dimensional kernel density estimates
# prefer not to load it, MASS::select() clash with dplyr::select()
```

```
# library(MASS)
# for creating polygonal isobands
library(isoband)
```

Here we use `MASS::bandwidth.nrd()` to select bandwidths, one for the x direction and one for the y direction. `kde2d()` will compute the KDE on a 200 x 200 grid of points.

```
xy <- sfAssaults |>
  st_coordinates() |>
  data.frame()

bandwidth <- c(
  MASS::bandwidth.nrd(xy$X),
  MASS::bandwidth.nrd(xy$Y))

kdeAssaults <- MASS::kde2d(
  x = xy$X, y = xy$Y,
  n = 200,
  h = bandwidth)
```

Stored in `kdeAssaults` is a component `z` that contains a probability density estimate. As we requested, `MASS::kde2d()` chopped the map of Chicago into a 200 by 200 grid and `z` contains the estimated crime density at those grid points. Those points are centered in a box with width and height equal to:

```
# width in meters
diff(kdeAssaults$x[1:2])
```

```
[1] 169.6576
```

```
# height in meters
diff(kdeAssaults$y[1:2])
```

```
[1] 211.274
```

`z` is scaled so that `z` times the width times the height equals the fraction of expected crime incidents in that box. So `z` is like the fraction of crime incidents per square meter. If we multiply each value of `z` by the width and height of the boxes (which all have the same size) and add them up, we should get a number close to 1.


```
sum(kdeAssaults$z * diff(kdeAssaults$x[1:2]) * diff(kdeAssaults$y[1:2]))
```

```
[1] 0.9974052
```

If we multiply all the values of z by the total number of crime incidents, then rather than describing the fraction of incidents per square meter (a hard-to-understand quantity), we estimate the expected number of incidents per square meter. Multiply that by 10^6 to convert square meters to square kilometers and divide by the number of years of data that we have so that we get an estimated number of crime incidents per km^2 per year, a measurement that describes the pace of incidents over space and time.

```
# scale so that the units are incidents/km2/year
kdeAssaults$zKM2Year <-
  kdeAssaults$z * nrow(sfAssaults) * 10^6 / nYears
hist(kdeAssaults$zKM2Year,
     xlab = "Crime rate",
     ylab = "Number of points",
     main = "")
```

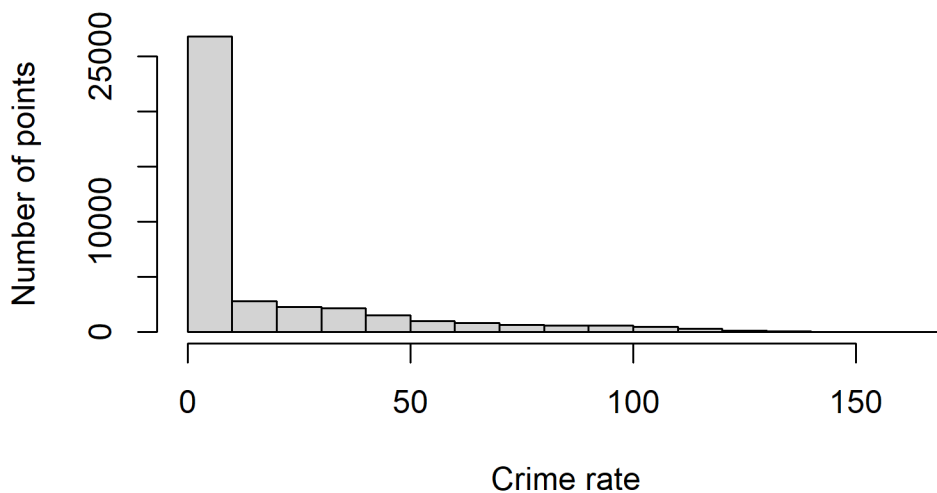


Figure 17: Histogram of estimated assault density per km^2 per year

The histogram shows that a large number of areas have very low crime rates, but this histogram has a long right tail, meaning some parts of Chicago have high crime rates. Let's determine some nice breakpoints for our map.

```
breaks <- kdeAssaults$zKM2Year |>
  range() |>
  pretty(n=10)
breaks
```

```
[1] 0 20 40 60 80 100 120 140 160 180
```

Now we ask `isobands()` to construct a polygonal band for each adjacent pair of breakpoints. For example, the first band contains locations where `zKM2Year` falls between `breaks[1]` and `breaks[2]`. The boundaries of a band follow the isolines at its lower and upper values. Unlike an isoline, however, each object returned here represents an area rather than a curve.

```
contourAssaults <-
  isobands(kdeAssaults$x,
           kdeAssaults$y,
           kdeAssaults$zKM2Year,
           levels_low = breaks[-length(breaks)],
           levels_high = breaks[-1]) |>
  iso_to_sfg() |> # convert to sf geometry object
  st_sfc(crs = 26916) |> # convert to sf data column
  st_sf(levels_low = breaks[-length(breaks)], # convert to sf object
        levels_high = breaks[-1],
        geometry = _) |>
  st_transform(4326)

# show the result so far
contourAssaults |>
  slice(-1) |> # drop the outer edge
  st_geometry() |>
  plot()
```



Figure 18: Isobands for Chicago assault incident rate

This resembles elevation contours on a topographical map. Nested bands whose values increase toward the center identify local peaks in estimated crime intensity. The band values and colors tell us how high the estimated intensity is. The spacing between band boundaries tells us how quickly the estimate changes over space. Closely spaced boundaries indicate a steep change, while widely spaced boundaries indicate a gradual change. Closely spaced boundaries do not, by themselves, indicate a higher crime intensity.

Let's create a color palette to shade the isobands. Here magma is used in its default direction, so low-intensity bands are dark purple or nearly black and high-intensity bands are light yellow. The brightest areas on the resulting maps represent the highest estimated incident intensities.

```
pal <- colorBin("magma",  
               domain = (contourAssaults$levels_low +  
                         contourAssaults$levels_high) / 2,  
               bins = breaks,  
               pretty = FALSE)
```

Now we are ready to overlay a colored contour map on top of our Chicago leaflet map. Here I have added multiple basemaps and a control to allow switching between layers and toggling the contour layer on and off.

```

leaflet() |>
  addProviderTiles(providers$Esri.WorldTopoMap, group="Topo") |>
  addProviderTiles(providers$CartoDB.Positron, group="Street") |>
  setView(lng=-87.8, lat=41.85, zoom= 10) |>
  addPolygons(
    data = contourAssaults,
    fillColor=~pal((levels_low + levels_high)/2),
    fillOpacity=0.4,
    color="#808080", #contour band edges
    weight=0.5,
    opacity=0.5,
    label=~paste0("Density: ",
                   format(levels_low, scientific=FALSE), "-",
                   format(levels_high, scientific=FALSE)) |>
    lapply(htmltools::HTML),
    highlightOptions =
      highlightOptions(weight=2, bringToFront= TRUE),
    group = "Crime density") |>
  addLegend(
    position="bottomright",
    pal=pal,
    values=(contourAssaults$levels_low +
            contourAssaults$levels_high)/2,
    title="Incidents/km2/ year", opacity=0.7) |>
  addLayersControl(
    baseGroups = c("Topo", "Street"),
    overlayGroups = "Crime density",
    options = layersControlOptions(collapsed = FALSE))

```

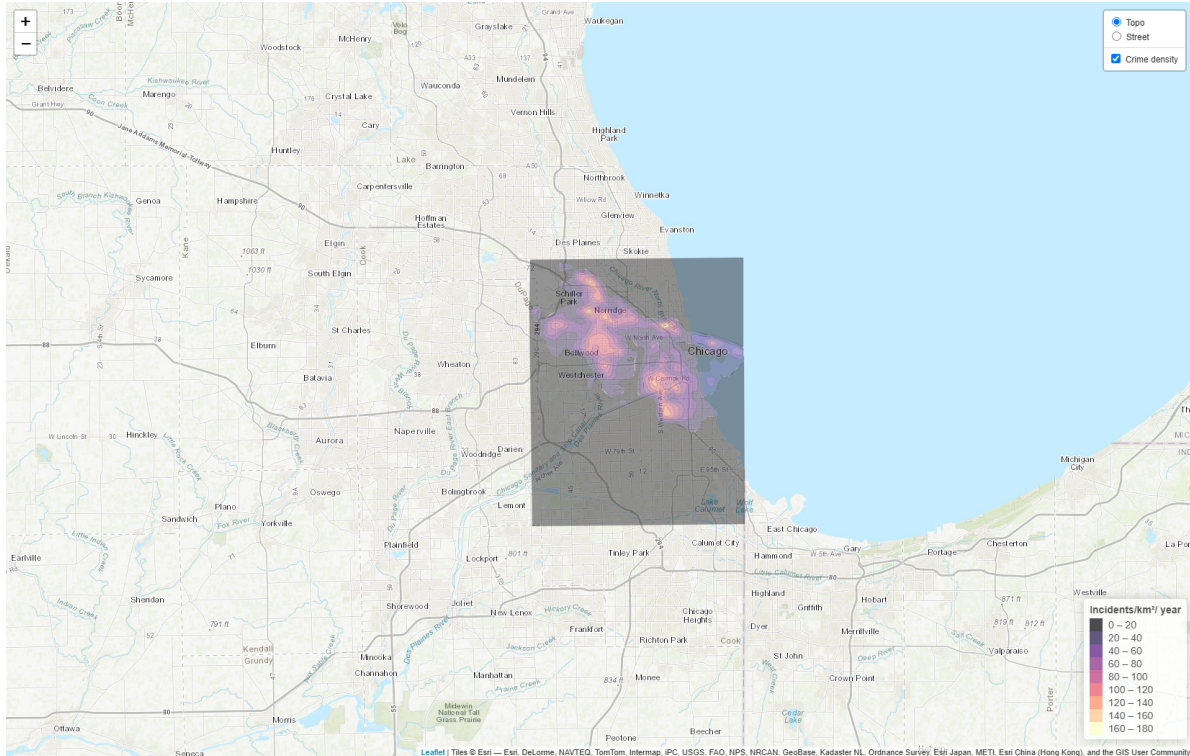


Figure 19: Kernel density estimate of reported assaults per square kilometer per year, Chicago

The isobands have been created over the rectangular Chicago bounding box, which ends up looking a little strange, with areas highlighted in Lake Michigan. We are going to trim them using a hull around the observed assault locations. A convex hull is the smallest convex polygon that contains every point. It acts like a rubber band stretched around the outside points, so it cannot bend inward where the point cloud does. A concave hull is more like shrink-wrap. It still contains every input point, but it can bend inward to follow the point cloud more closely.

`st_concave_hull()` works on each geometry separately. `st_combine()` first combines all the individual assault points into one multipoint geometry so that we get one hull around the entire point set. The [underlying GEOS algorithm](#) begins by making non-overlapping triangles with the points. The algorithm then works inward from the outside, removing eligible border triangles with long exposed edges while keeping the hull connected and retaining every point. The `ratio` argument controls the edge-length threshold for that removal. `ratio = 1` produces the convex hull. A `ratio = 0.2` puts the threshold 20% of the way from the shortest edge to the longest edge. Border triangles with exposed edges longer than the threshold can be removed.

This leaflet map shows the convex hull in gray and the tighter concave hull with `ratio = 0.2` in purple. A larger value such as 0.7 would produce a looser boundary closer to the convex

hull. The 200-meter buffer in the code moves each boundary slightly outward so that we do not trim the KDE surface exactly at the outermost incident locations. We could use either hull to remove isobands from places where assaults do not occur.

```
leaflet() |>
  addTiles() |>
  setView(lng = -87.8, lat = 41.85, zoom = 10) |>
  addPolygons(
    data = sfAssaults |>
      st_geometry() |>
      st_combine() |>
      st_convex_hull() |>
      st_buffer(200) |> # add a little extra at the edges
      st_transform(crs = 4326),
    color = "#808080",
    weight = 0.5,
    opacity = 0.9) |>
  addPolygons(
    data = sfAssaults |>
      st_geometry() |>
      st_combine() |>
      st_concave_hull(ratio=0.2) |>
      st_buffer(200) |> # add a little extra at the edges
      st_transform(crs = 4326),
    color = "purple",
    weight = 0.5,
    opacity = 0.9,
    highlightOptions =
      highlightOptions(weight = 2, bringToFront = TRUE))
```

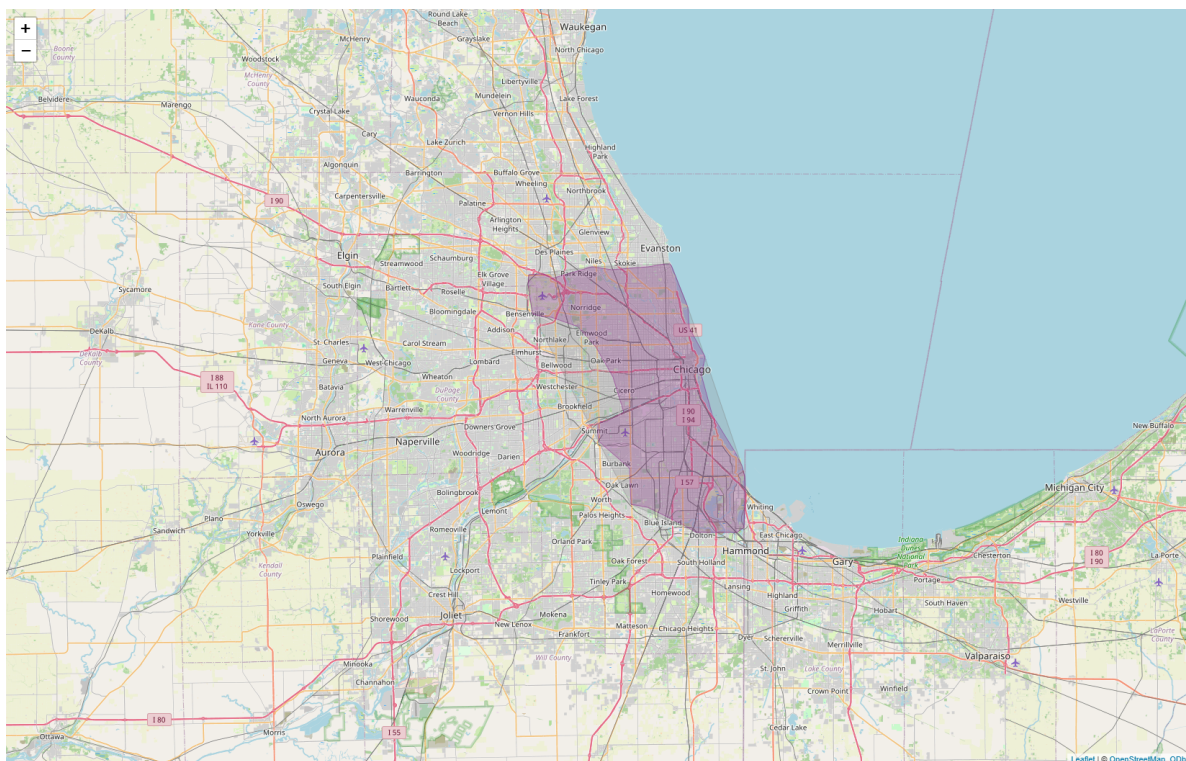


Figure 20: Convex and concave hulls surrounding reported assault locations

Also, let's make the hotspot map only highlight the hottest spots, those with a crime rate exceeding 80 incidents per km² per year. Zoom in on some of the brightest areas to see what is in these high-assault-rate areas.

```
legendMid <- (contourAssaults$levels_low +
              contourAssaults$levels_high)/2
legendMid <- legendMid[legendMid > 80]
legendCol <- pal(legendMid)
legendLabs <- paste0(contourAssaults$levels_low, "-",
                     contourAssaults$levels_high)
legendLabs <- legendLabs[contourAssaults$levels_low >=80]
leaflet() |>
  addTiles() |>
  setView(lng=-87.8, lat= 41.85, zoom=10) |>
  addPolygons(
    data= contourAssaults |>
      st_intersection(sfAssaults |>
        st_geometry() |>
```

```

        st_combine() |>
        st_concave_hull(ratio=0.2) |>
        st_buffer(200) |>
        st_transform(crs=4326)) |>
    filter(levels_low>= 80), # highlight just hottest spots
    fillColor = ~pal((levels_low + levels_high)/2),
    fillOpacity= 0.4,
    color="#808080",
    weight= 0.5,
    opacity= 0.5,
    label=~paste0("Density:",
                   format(levels_low, scientific=FALSE), "-",
                   format(levels_high, scientific=FALSE)),
    highlightOptions=
      highlightOptions(weight=2, bringToFront= TRUE)) |>
addLegend(
  position="bottomright",
  colors= legendCol,
  labels= legendLabs,
  title="Incidents/km2 /year",
  opacity= 0.7)

```

Warning: attribute variables are assumed to be spatially constant throughout all geometries

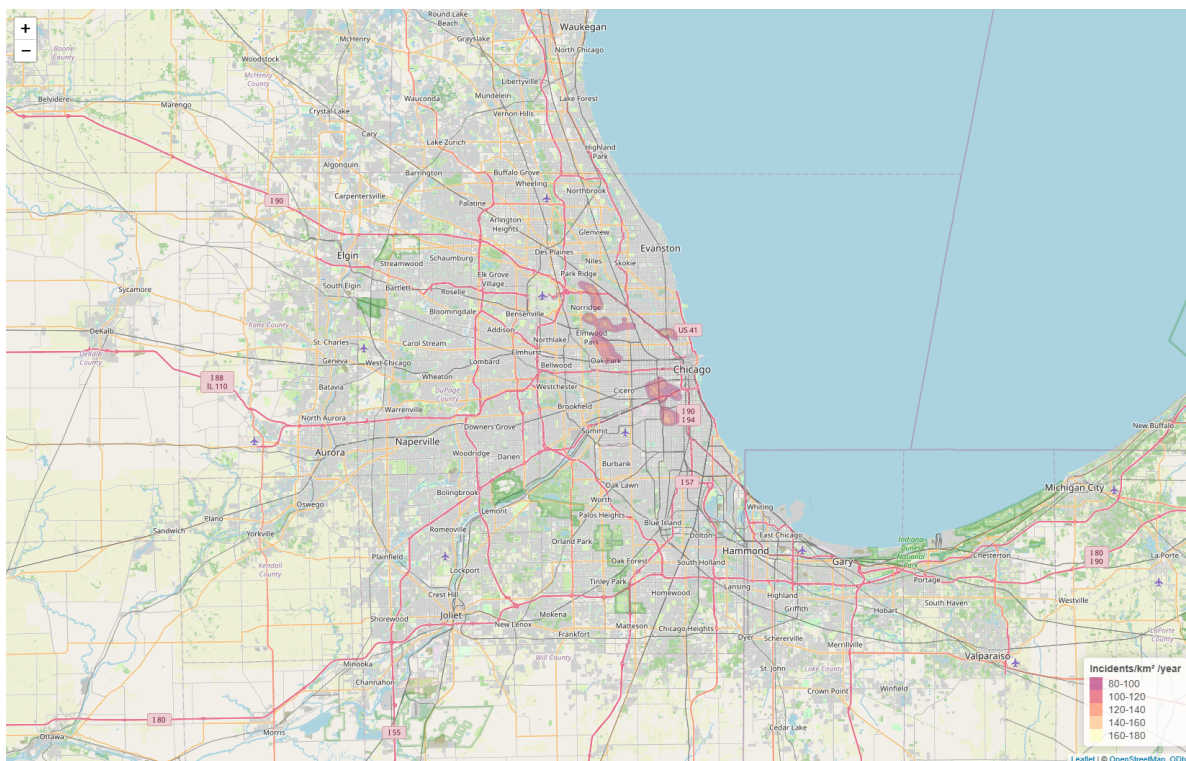


Figure 21: Highest-intensity reported assault hotspots clipped to the concave hull

Exercise

10. Multiply the KDE bandwidth by 0.5 and by 2. Rebuild the contours and explain how the substantive story changes

7.3 Comparing hotspots over time

Let's create hotspot maps with just motor vehicle theft, making a separate map for each year between 2018 and 2025. I make sure that the bandwidth, breaks, palette, and clipping boundary are the same for all of the maps.

```
sfMVT <- crime |>
  filter(valid_coordinates &
         between(year(date), 2018, 2025)) |>
  semi_join(iucrLookup |>
            filter(primary_type == "MOTOR VEHICLE THEFT"),
            by = "iucr") |>
```

```

select(longitude, latitude, date) |>
mutate(year = year(date)) |>
select(-date) |>
st_as_sf(coords = c("longitude", "latitude"),
          crs = 4326) |>
st_transform(crs = 26916)

# select breaks, bandwidth, and palette based on 2023
xy <- sfMVT |>
  filter(year == 2023) |> # for just one year
  st_coordinates() |>
  data.frame()

h <- c(MASS::bandwidth.nrd(xy$X),
      MASS::bandwidth.nrd(xy$Y))

breaks <- seq(0, 140, by=20)

pal <- colorBin("viridis",
               domain = (breaks[-length(breaks)]+breaks[-1])/2,
               bins = breaks,
               pretty = FALSE)

# construct a legend containing only the displayed isobands
legendLowMVT <- breaks[-length(breaks)]
legendHighMVT <- breaks[-1]
legendKeepMVT <- legendLowMVT >= 25
legendColMVT <- pal(
  ((legendLowMVT + legendHighMVT)/2)[legendKeepMVT])
legendLabsMVT <- paste0(legendLowMVT, "-", legendHighMVT)[legendKeepMVT]

```

We then loop through the years 2018 to 2025, subsetting the data to one of those years at a time, and generate a hotspot map.

```

maps <- lapply(2018:2025,
function(year0)
{
  message(paste("Mapping", year0))

  xy <- sfMVT |>
    filter(year == year0) |> # for just one year
    st_coordinates() |>

```

```

data.frame()
kdeCarTheft <- MASS::kde2d(xy$X, xy$Y, n = 200, h = h)

# no need to divide by year here... only one year of data
kdeCarTheft$zKM2Year <- kdeCarTheft$z * nrow(xy) * 10^6

contourCarTheft <-
  isobands(kdeCarTheft$x,
           kdeCarTheft$y,
           kdeCarTheft$zKM2Year,
           levels_low = breaks[-length(breaks)],
           levels_high = breaks[-1]) |>
  iso_to_sfg() |>
  st_sfc(crs = 26916) |>
  st_sf(levels_low = breaks[-length(breaks)],
        levels_high = breaks[-1],
        geometry = _) |>
  st_transform(4326) # back to lat/long

leaflet() |>
  addTiles() |>
  setView(lng = -87.8, lat = 41.85, zoom = 10) |>
  addPolygons(
    data = contourCarTheft |>
      st_intersection(sfMVT |>
        st_geometry() |>
        st_combine() |>
        st_concave_hull(ratio=0.7) |>
        st_buffer(200) |>
        st_transform(crs = 4326)) |>
    filter(levels_low >= 25), # highlight just the hottest spots
    fillColor = ~pal((levels_low + levels_high)/2),
    fillOpacity = 0.4,
    color = "#808080",
    weight = 0.5,
    opacity = 0.5,
    label = ~paste0("Density: ",
                     format(levels_low, scientific=FALSE), "-",
                     format(levels_high, scientific=FALSE)) |>
    lapply(htmltools::HTML),
    highlightOptions =
      highlightOptions(weight = 2, bringToFront = TRUE)) |>

```

```

    addLegend(
      position = "bottomright",
      colors = legendColMVT,
      labels = legendLabsMVT,
      title = paste(year0, "Incidents / km2 / year"),
      opacity = 0.7)
  })

dir.create("figs", showWarnings = FALSE)

yrs <- 2018:2025
pngs <- character(length(maps))

for (i in seq_along(maps)) {
  htmlf <- tempfile(fileext = ".html")
  pngf <- file.path("figs", paste0("car-theft-", yrs[i], ".png"))
  htmlwidgets::saveWidget(maps[[i]], htmlf, selfcontained = TRUE)
  webshot2::webshot(htmlf, pngf, vwidth = 1200, vheight = 900, zoom = 2)
  pngs[i] <- pngf
}

knitr::include_graphics(pngs)

```

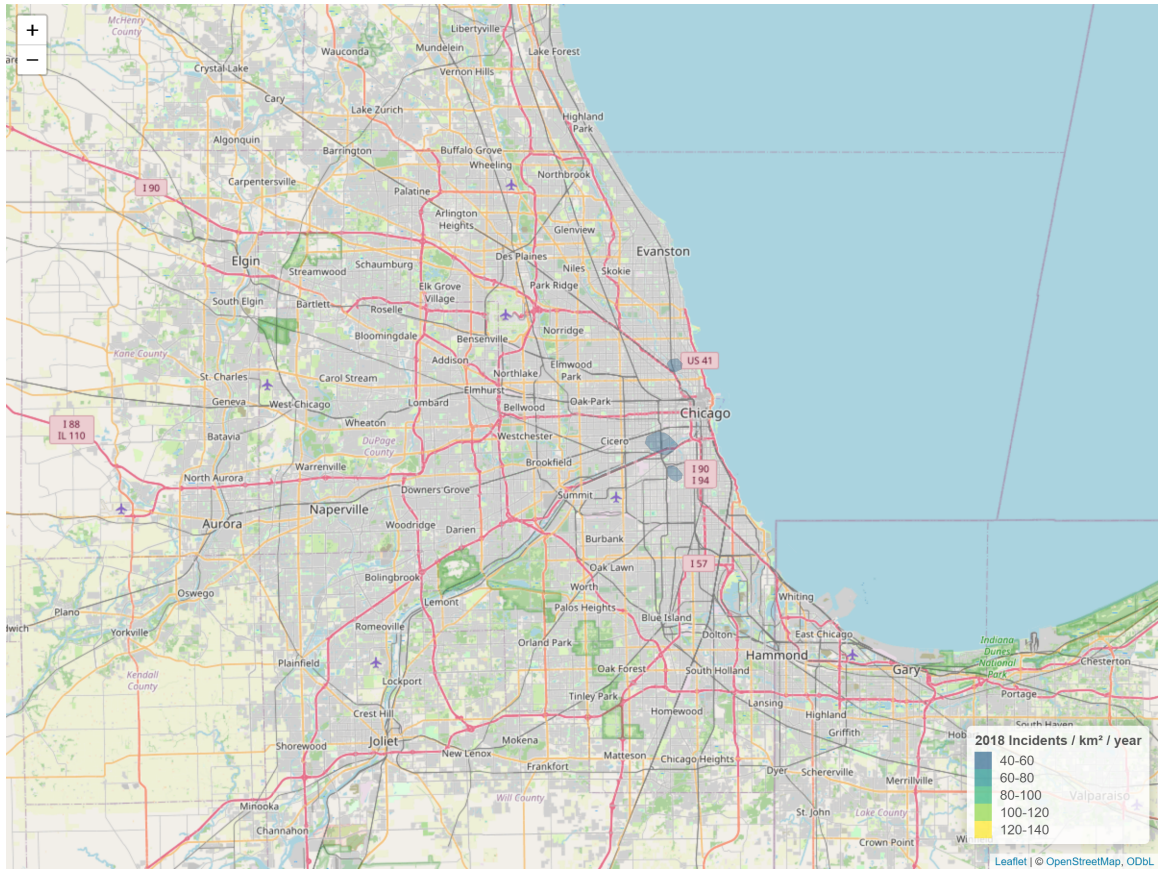


Figure 22: Annual motor vehicle theft hotspots in Chicago, 2018 through 2025

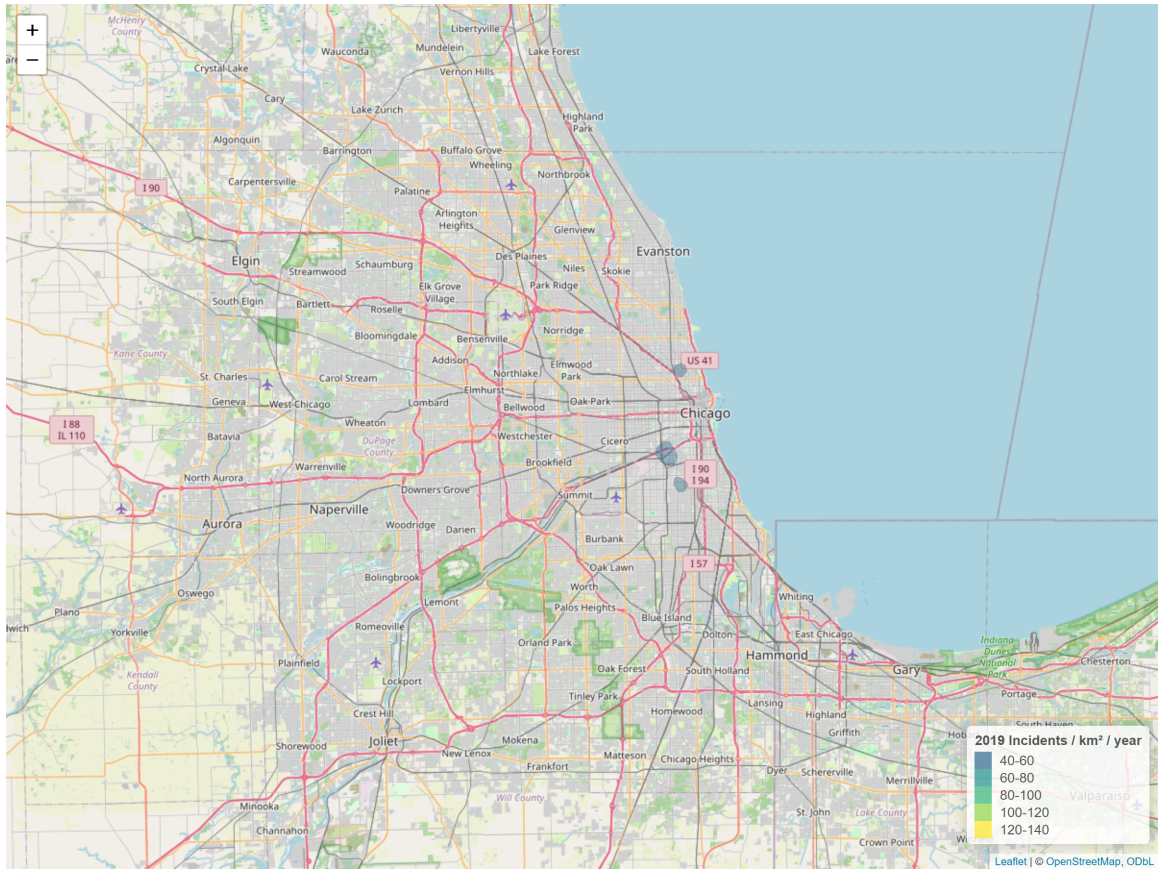


Figure 23: Annual motor vehicle theft hotspots in Chicago, 2018 through 2025

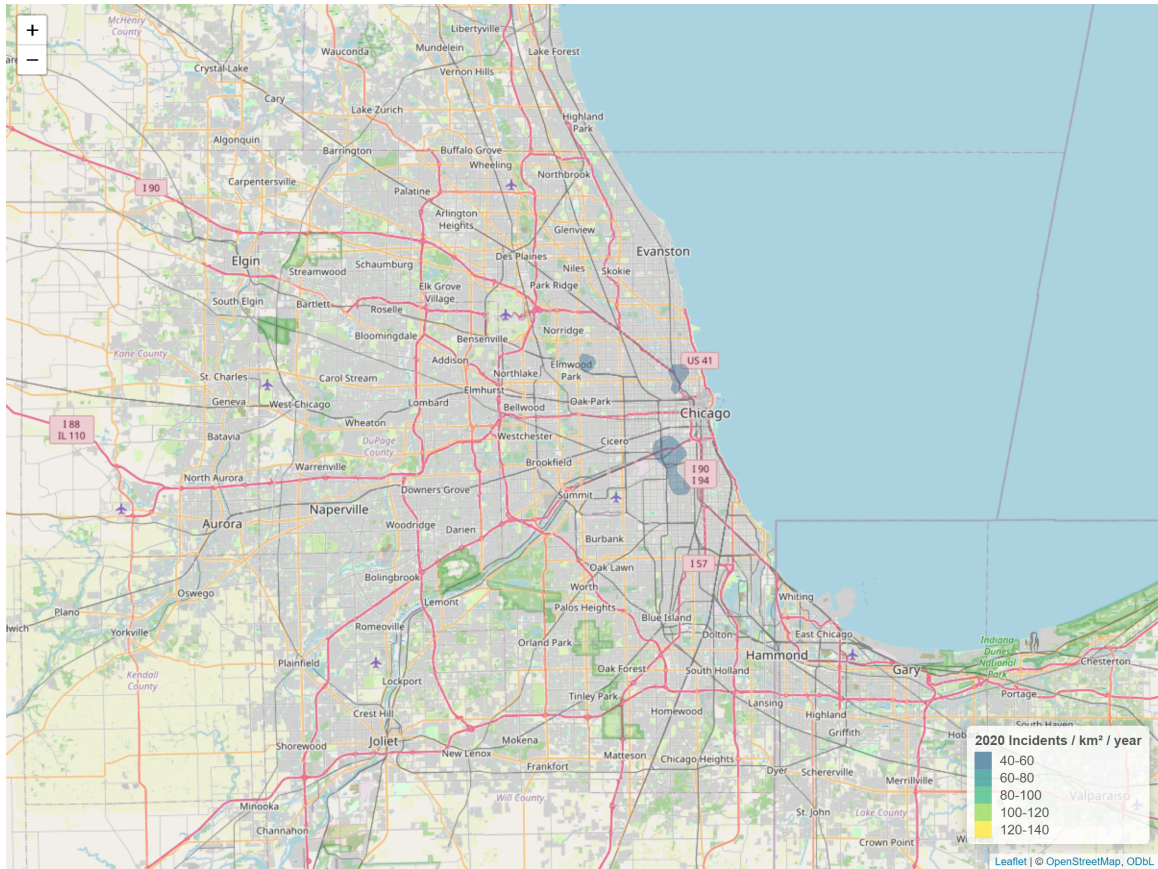


Figure 24: Annual motor vehicle theft hotspots in Chicago, 2018 through 2025

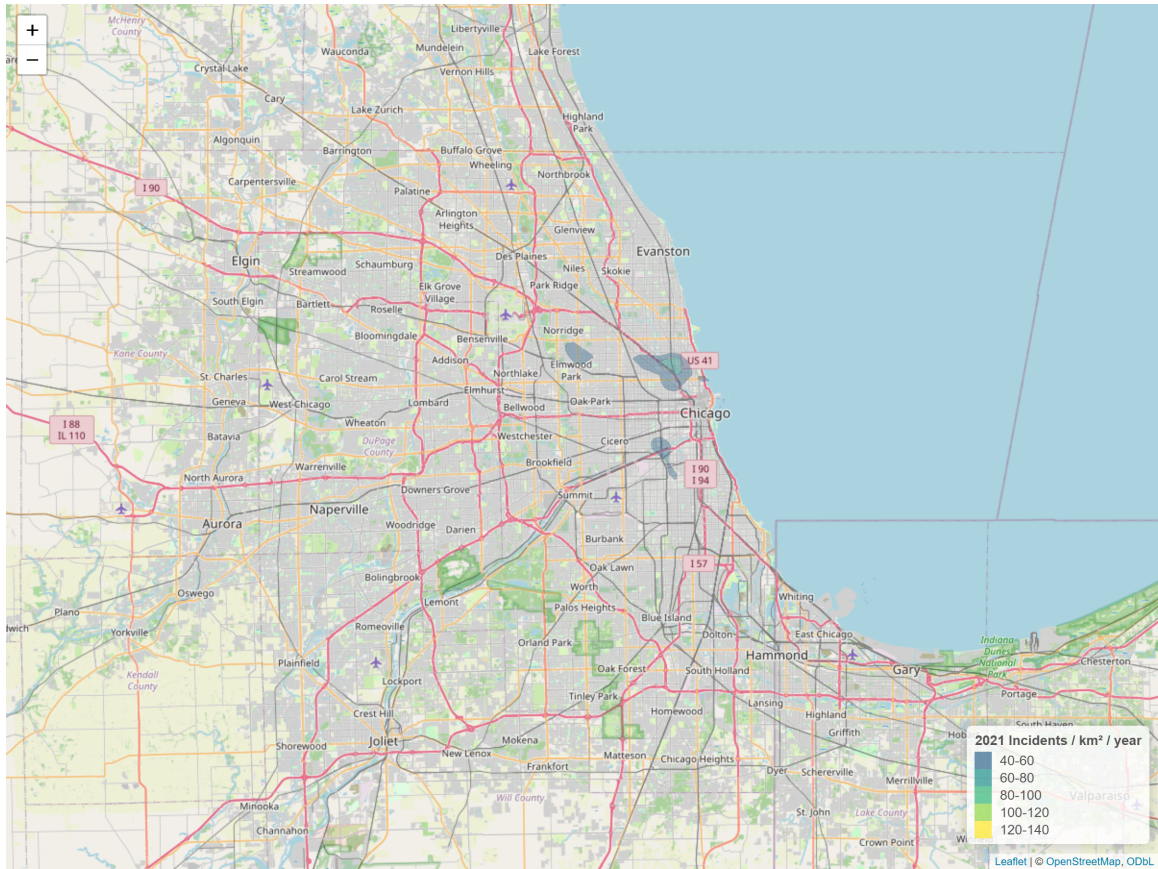


Figure 25: Annual motor vehicle theft hotspots in Chicago, 2018 through 2025

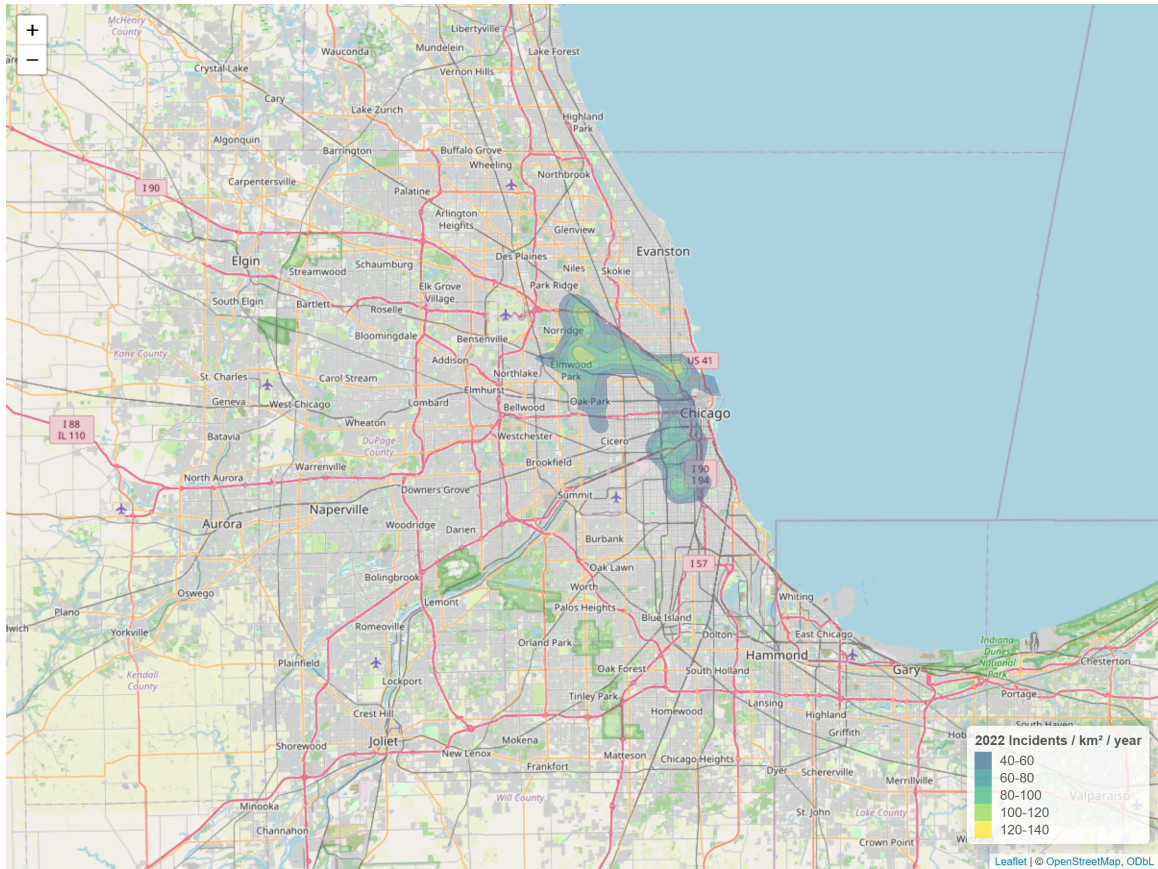


Figure 26: Annual motor vehicle theft hotspots in Chicago, 2018 through 2025

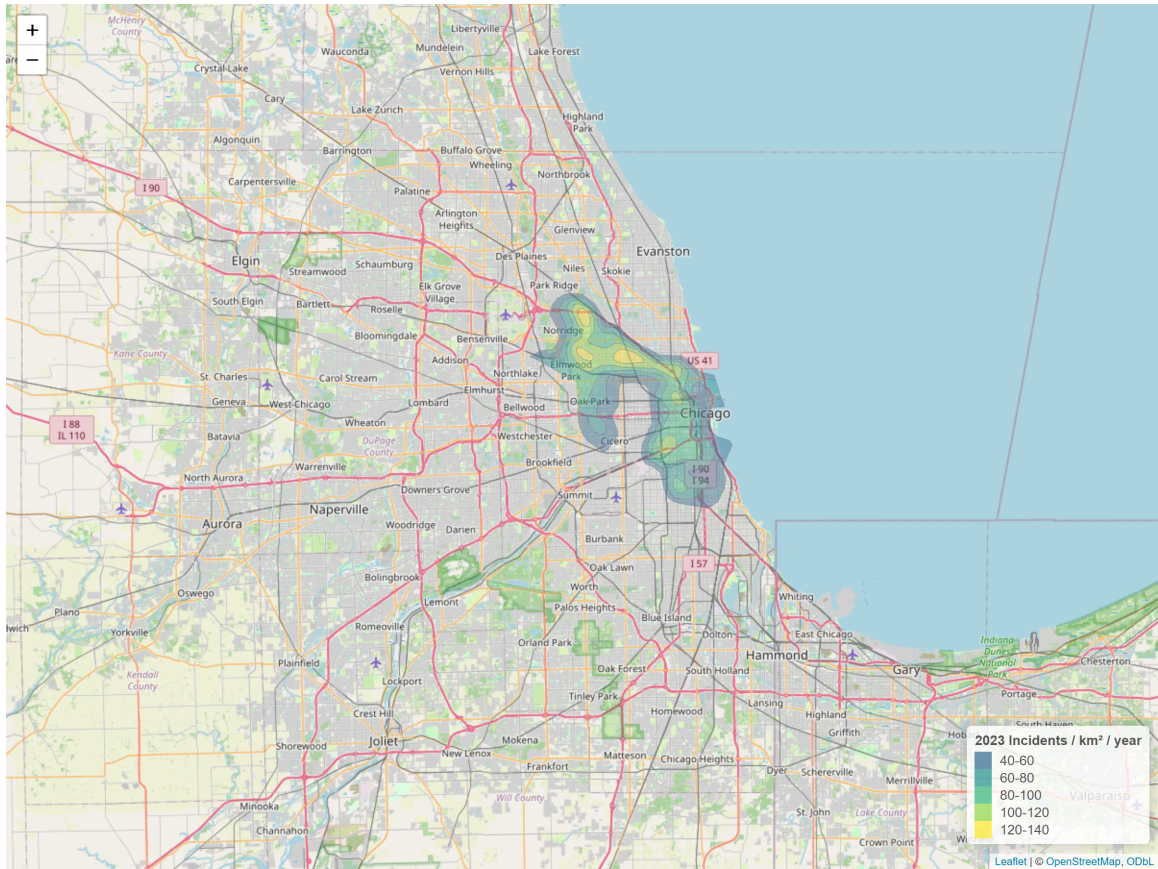


Figure 27: Annual motor vehicle theft hotspots in Chicago, 2018 through 2025

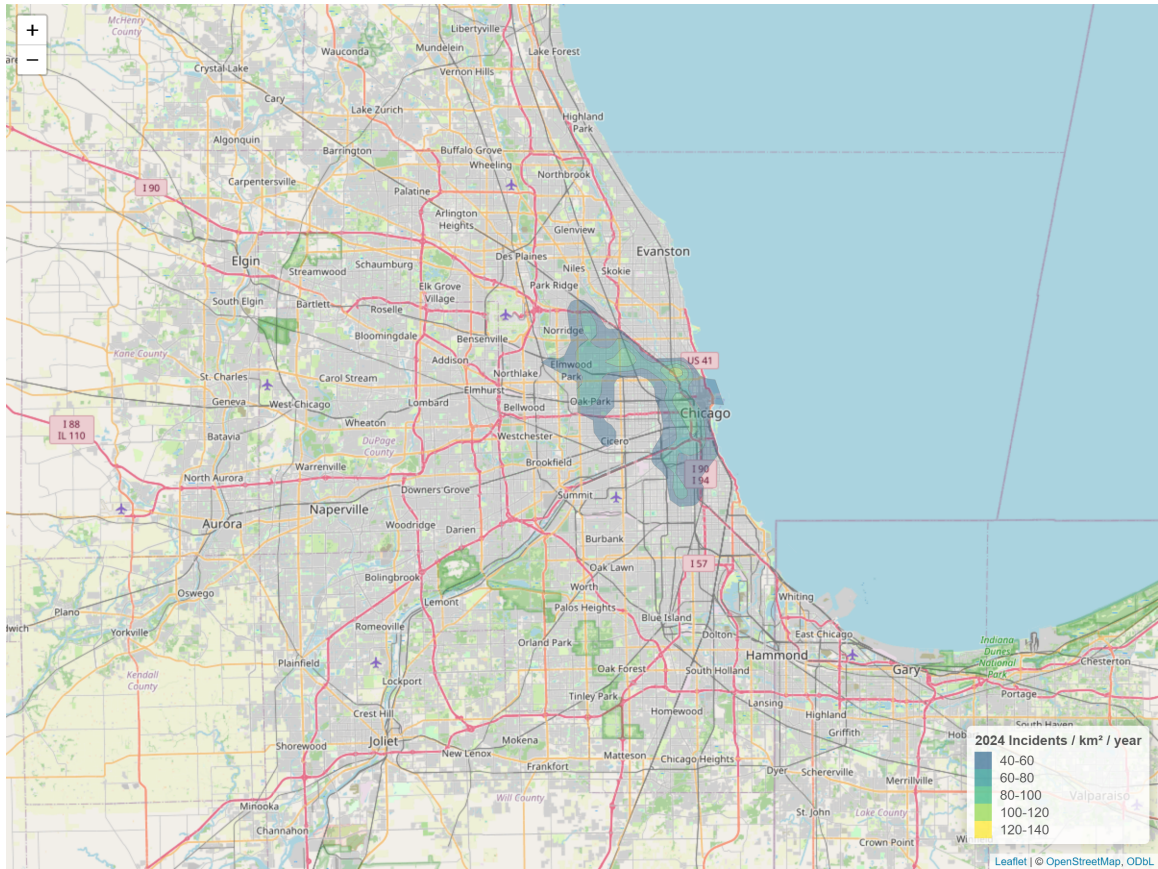


Figure 28: Annual motor vehicle theft hotspots in Chicago, 2018 through 2025

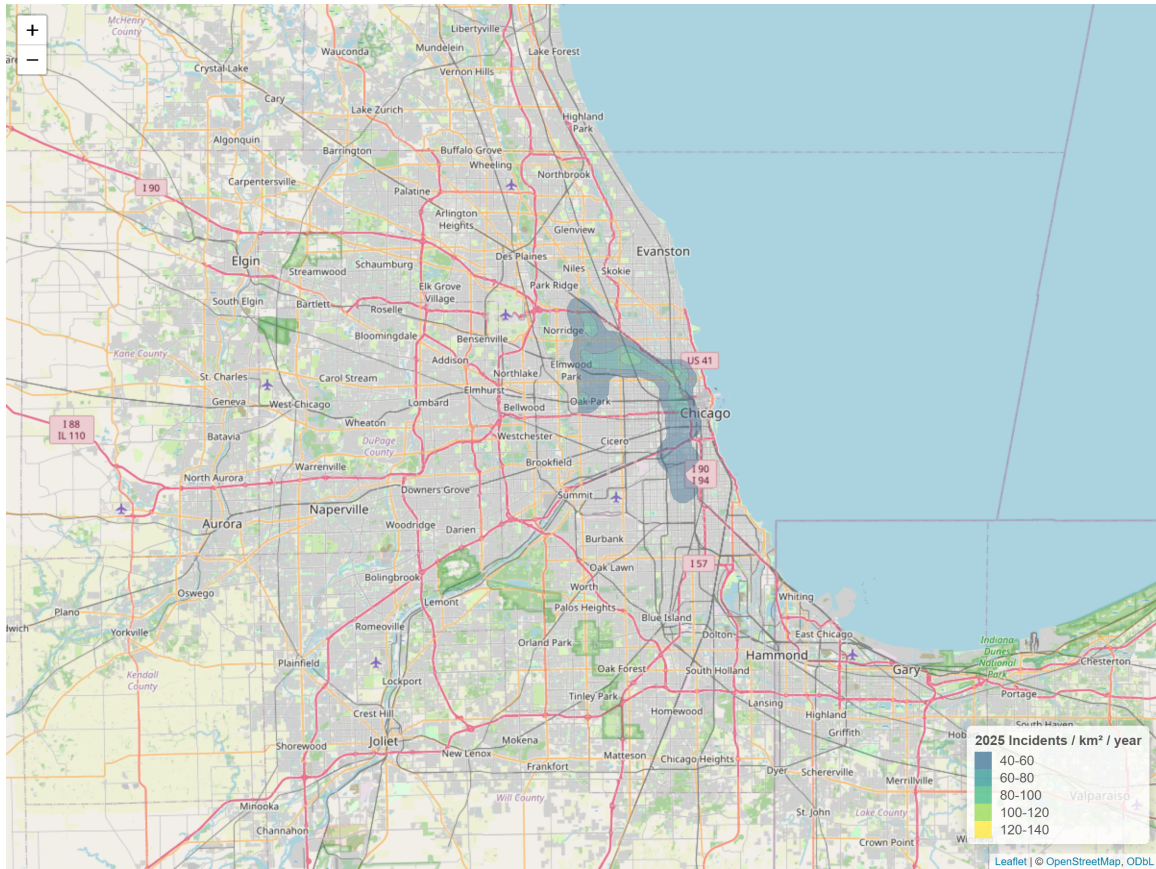


Figure 29: Annual motor vehicle theft hotspots in Chicago, 2018 through 2025

Car thefts spiked between 2022 and 2024.

```
sfMVT |>
  count(year) |>
  plot(n~year, data=_,
       xlab = "Year",
       ylab = "Number of car thefts")
```

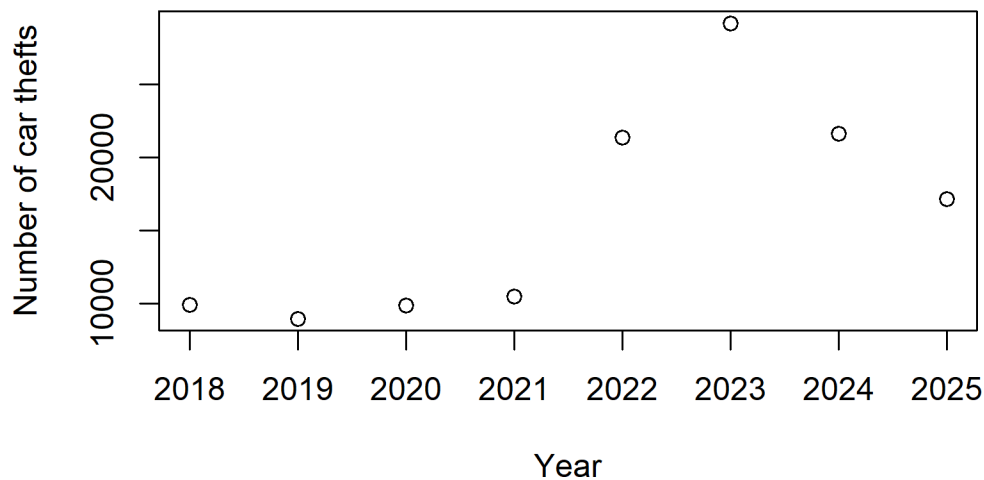


Figure 30: Reported motor vehicle thefts by year in Chicago, 2018 through 2025

That spike was fueled by thefts of Kia and Hyundai vehicles, which lacked passive immobilizer antitheft devices as standard equipment and led to a social media trend describing how to steal Kia and Hyundai vehicles.

Many cities other than Chicago have accessible incident-level data. You can easily modify the code here to make a hotspot map for Philadelphia, Los Angeles, Seattle, San Francisco, Baltimore, Washington DC, and many others.

i Exercise

11. Create common-scale hex maps for motor vehicle theft in each year from 2020 through 2025.

8 Solutions to the exercises

1. What are the 10 most common crimes? Use `primary_type`

```
crime |>
  left_join(iucrLookup |>
```

```

      select(iucr, primary_type),
      by = "iucr") |>
count(primary_type) |>
arrange(desc(n)) |>
slice_head(n = 10)

```

	primary_type	n
	<char>	<int>
1:	THEFT	1827852
2:	BATTERY	1567561
3:	CRIMINAL DAMAGE	977643
4:	NARCOTICS	768762
5:	ASSAULT	580273
6:	OTHER OFFENSE	537705
7:	BURGLARY	455566
8:	MOTOR VEHICLE THEFT	444886
9:	DECEPTIVE PRACTICE	400208
10:	ROBBERY	318137

2. Which FBI code has the most IUCR codes associated with it? What are the primary types associated with that FBI code?

```

iucrLookup |>
  count(fbi_code, name = "n_iucr_codes") |>
  arrange(desc(n_iucr_codes)) |>
  slice_max(n_iucr_codes)

```

```

# A tibble: 1 x 2
  fbi_code n_iucr_codes
  <chr>      <int>
1 26          75

```

```

iucrLookup |>
  semi_join(iucrLookup |>
    count(fbi_code, name = "n_iucr_codes") |>
    arrange(desc(n_iucr_codes)) |>
    slice_max(n_iucr_codes)) |>
  distinct(fbi_code, primary_type)

```

Joining with `by = join\_by(fbi\_code)`

```
# A tibble: 12 x 2
  fbi_code primary_type
  <chr>    <chr>
1 26      OTHER OFFENSE
2 26      CRIMINAL TRESPASS
3 26      STALKING
4 26      KIDNAPPING
5 26      HUMAN TRAFFICKING
6 26      OFFENSE INVOLVING CHILDREN
7 26      NARCOTICS
8 26      INTERFERENCE WITH PUBLIC OFFICER
9 26      PUBLIC PEACE VIOLATION
10 26     OBSCENITY
11 26     SEX OFFENSE
12 26     INTIMIDATION
```

3. Which primary types result in the highest arrest rate? Give the top 10 primary types with at least 1000 incidents

```
crime |>
  count(iucr, arrest, name = "crime_count") |>
  left_join(iucrLookup |>
    select(iucr, primary_type),
    by = "iucr") |>
  summarize(
    incidents = sum(crime_count),
    arrests   = sum(crime_count[arrest == TRUE]),
    arrest_rate = arrests / incidents,
    .by = "primary_type") |>
  filter(incidents >= 1000) |>
  arrange(desc(arrest_rate)) |>
  slice_head(n = 10)
```

	primary_type	incidents	arrests	arrest_rate
1	PROSTITUTION	70526	70206	0.9954627
2	NARCOTICS	768762	763490	0.9931422
3	GAMBLING	14674	14566	0.9926400
4	LIQUOR LAW VIOLATION	15534	15374	0.9897000
5	CONCEALED CARRY LICENSE VIOLATION	1823	1759	0.9648930
6	INTERFERENCE WITH PUBLIC OFFICER	20968	19222	0.9167303
7	WEAPONS VIOLATION	128627	93543	0.7272423
8	CRIMINAL TRESPASS	230928	155307	0.6725343

9	PUBLIC PEACE VIOLATION	55717	34702	0.6228261
10	HOMICIDE	14323	6937	0.4843259

4. Which district has the most thefts?

```
crime |>
  semi_join(iucrLookup |>
    filter(primary_type == "THEFT"),
    by = "iucr") |>
  filter(!is.na(district)) |>
  count(district, name = "thefts") |>
  slice_max(thefts)
```

```
district thefts
<char> <int>
1:      018 164222
```

5. Count the number of assaults since 2016 that have occurred on Fridays and Saturdays after 6pm. Report the assault counts by date, day of week, hour of the day, and year.

```
crime |>
  semi_join(iucrLookup |>
    filter(primary_type == "ASSAULT"),
    by = "iucr") |>
  mutate(calendar_date = date(date),
    weekday = lubridate::wday(date, label = TRUE, abbr = FALSE),
    hour = hour(date),
    calendar_year = year(date)) |>
  filter(calendar_year >= 2016,
    weekday %in% c("Friday", "Saturday"),
    hour >= 18) |>
  count(calendar_year, calendar_date, weekday, hour,
    name = "assaults") |>
  arrange(calendar_date, hour)
```

	calendar_year	calendar_date	weekday	hour	assaults
	<int>	<Date>	<ord>	<int>	<int>
1:	2016	2016-01-01	Friday	18	2
2:	2016	2016-01-01	Friday	19	3
3:	2016	2016-01-01	Friday	20	1
4:	2016	2016-01-01	Friday	21	3
5:	2016	2016-01-01	Friday	22	1


```

----
6077:      2026      2026-07-18 Saturday      19      1
6078:      2026      2026-07-18 Saturday      20      2
6079:      2026      2026-07-18 Saturday      21      3
6080:      2026      2026-07-18 Saturday      22      2
6081:      2026      2026-07-18 Saturday      23      5

```

6. Plot the longitude and latitude of all “ASSAULT”s for Ward 22.

```

crime |>
  filter(valid_coordinates & (ward == "22")) |>
  semi_join(iucrLookup |>
    filter(primary_type == "ASSAULT"),
    by = "iucr") |>
  ggplot(aes(x = longitude, y = latitude)) +
  geom_point(size = 0.5, alpha = 0.5) +
  coord_fixed() +
  labs(x = "Longitude", y = "Latitude") +
  theme_minimal()

```

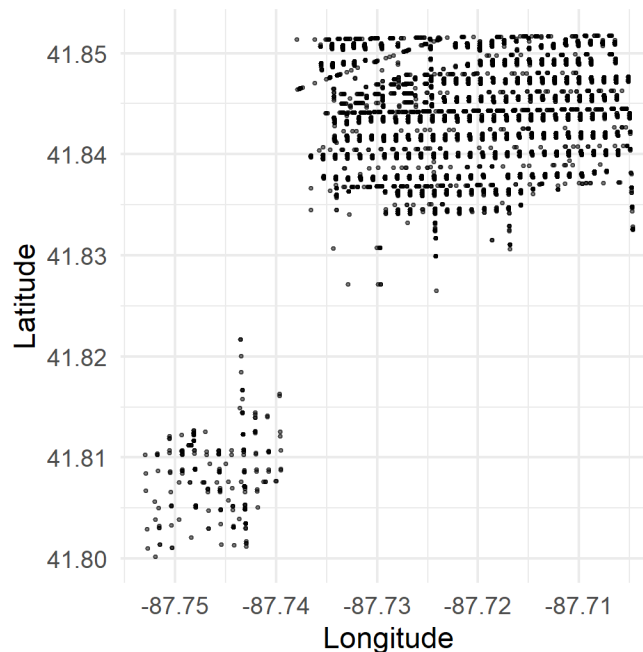


Figure 31: Reported assault locations in Chicago Ward 22

7. What is the most common (longitude,latitude) for assaults in Ward 22? Add that point to the plot as a larger red dot.

```

most_common_location <- crime |>
  filter(valid_coordinates & (ward == "22")) |>
  semi_join(iucrLookup |>
    filter(primary_type == "ASSAULT"),
    by = "iucr") |>
  count(longitude, latitude, name = "assaults") |>
  slice_max(assaults)

crime |>
  filter(valid_coordinates & (ward == "22")) |>
  semi_join(iucrLookup |>
    filter(primary_type == "ASSAULT"),
    by = "iucr") |>
ggplot(aes(x = longitude, y = latitude)) +
  geom_point(size = 0.5, alpha = 0.5) +
  geom_point(data = most_common_location,
    color = "red",
    size = 4) +
  coord_fixed() +
  labs(x = "Longitude", y = "Latitude") +
  theme_minimal()

```

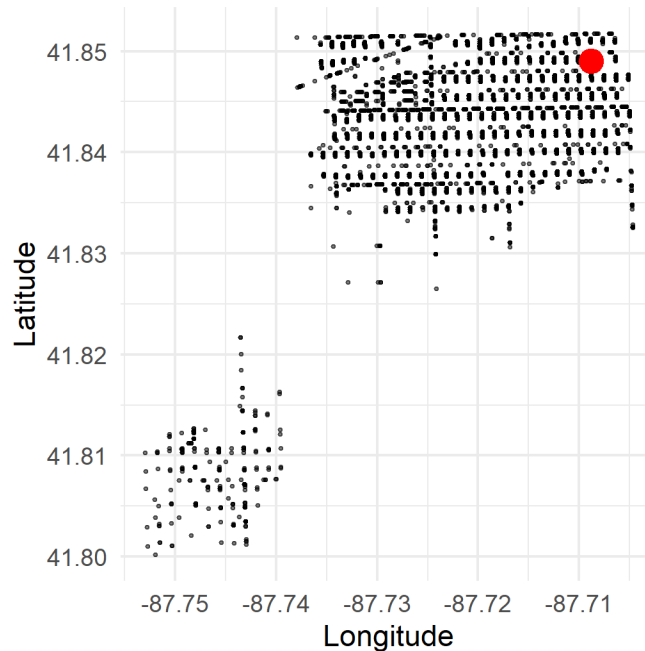


Figure 32: Most frequently repeated assault coordinates in Chicago Ward 22

8. Create a point map for motor vehicle theft in 2024. Compare the unclustered (by removing `clusterOptions = markerClusterOptions()`) and clustered versions and explain what each version hides

```
vehicle_theft <- crime |>
  filter(valid_coordinates & (year(date) == 2024)) |>
  semi_join(iucrLookup |>
    filter(primary_type == "MOTOR VEHICLE THEFT"),
    by = "iucr")

vehicle_theft_map <- leaflet(vehicle_theft) |>
  addProviderTiles(providers$CartoDB.Positron) |>
  setView(lng = -87.68, lat = 41.84, zoom = 10)

vehicle_theft_map |>
  addCircleMarkers(lng = ~longitude,
    lat = ~latitude,
    radius = 2,
    stroke = FALSE,
    fillOpacity = 0.25)
```

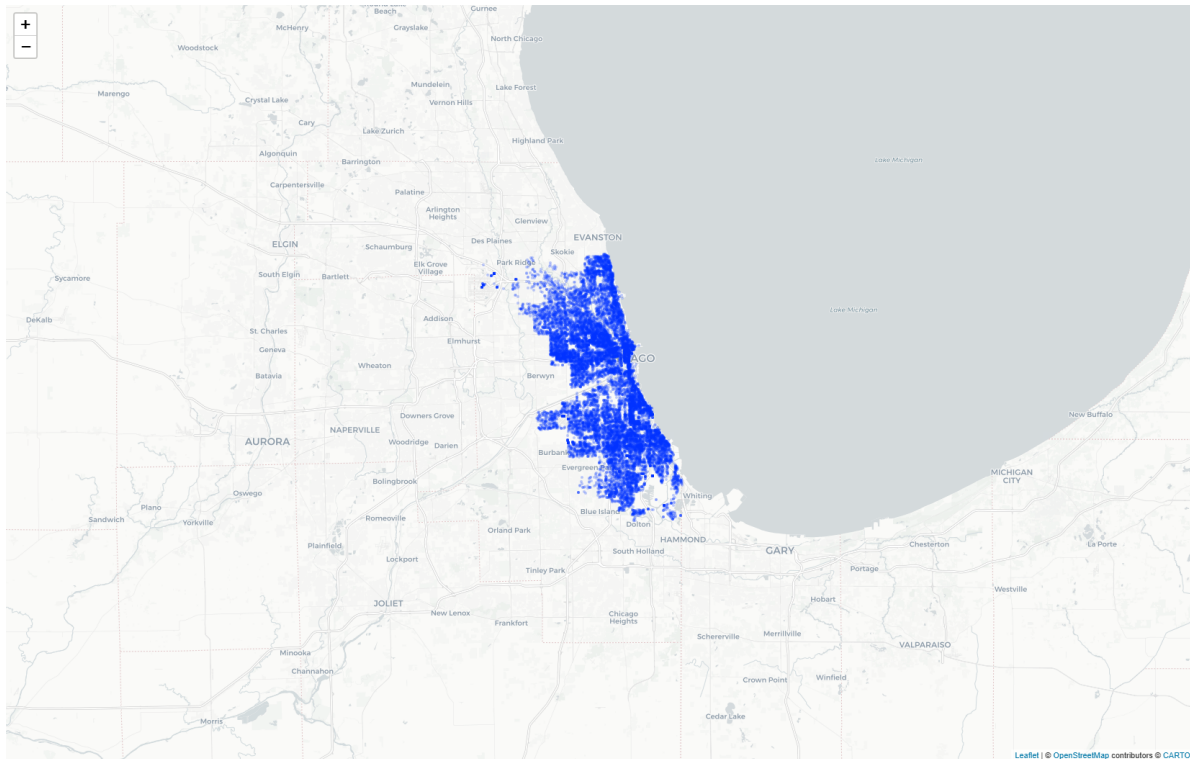


Figure 33: Unclustered motor vehicle theft point map, Chicago, 2024

```
vehicle_theft_map |>
  addCircleMarkers(lng = ~longitude,
                  lat = ~latitude,
                  radius = 2,
                  stroke = FALSE,
                  fillOpacity = 0.25,
                  clusterOptions = markerClusterOptions())
```

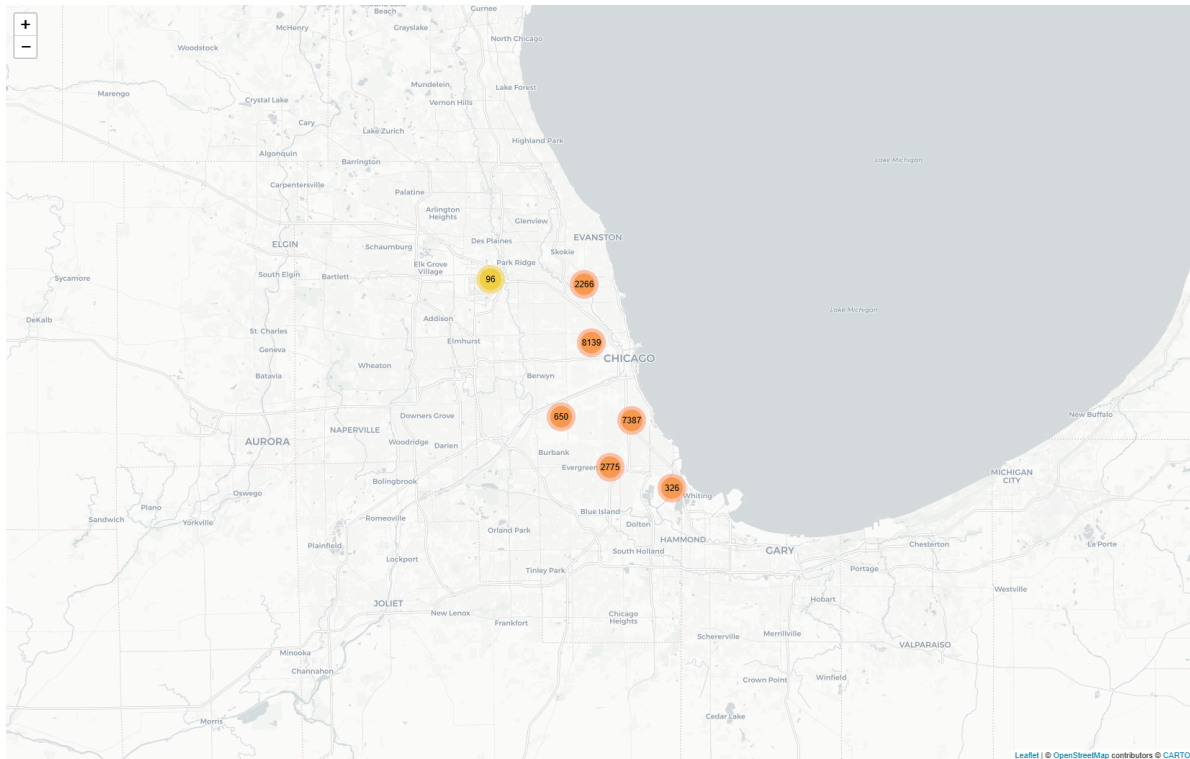


Figure 34: Clustered motor vehicle theft point map, Chicago, 2024

The unclustered map preserves individual locations, but overlapping markers hide how many incidents occur in dense areas. The clustered map summarizes nearby markers and is easier to navigate, but the cluster boundaries and counts change with zoom.

9. Rebuild the hexagonal map with 250-meter and 1,000-meter cells. Describe which features persist and which depend on cell size

Create both grids from the same assault points and calculate density in the same units. The shared color scale makes the two resolutions directly comparable.

```
hexWidths <- c("250-meter hexagons" = 250,
               "1,000-meter hexagons" = 1000)

solutionHexGrids <- lapply(seq_along(hexWidths),
                           function(i)
                           {
                               gridGeometry <- st_make_grid(sfAssaults,
                                                             cellsize = hexWidths[i],
                                                             square = FALSE)
```

```

st_sf(geometry = gridGeometry) |>
  mutate(
    nAssaults = lengths(st_intersects(geometry, sfAssaults)),
    areaKM2 = as.numeric(st_area(geometry)) / 10^6,
    density = nAssaults / areaKM2 / nYears,
    hex_width = names(hexWidths)[i]) |>
  filter(nAssaults > 0)
}) |>
do.call(rbind, args = _) |>
mutate(
  hex_width = factor(
    hex_width,
    levels = names(hexWidths)))

ggplot(solutionHexGrids) +
  geom_sf(aes(fill = density), color = NA) +
  scale_fill_viridis_c(
    option = "magma",
    direction = -1,
    transform = "sqrt",
    name = "Assaults /\nkm2 / year") +
  facet_wrap(~hex_width, nrow = 1) +
  coord_sf(datum = NA) +
  theme_void()

```

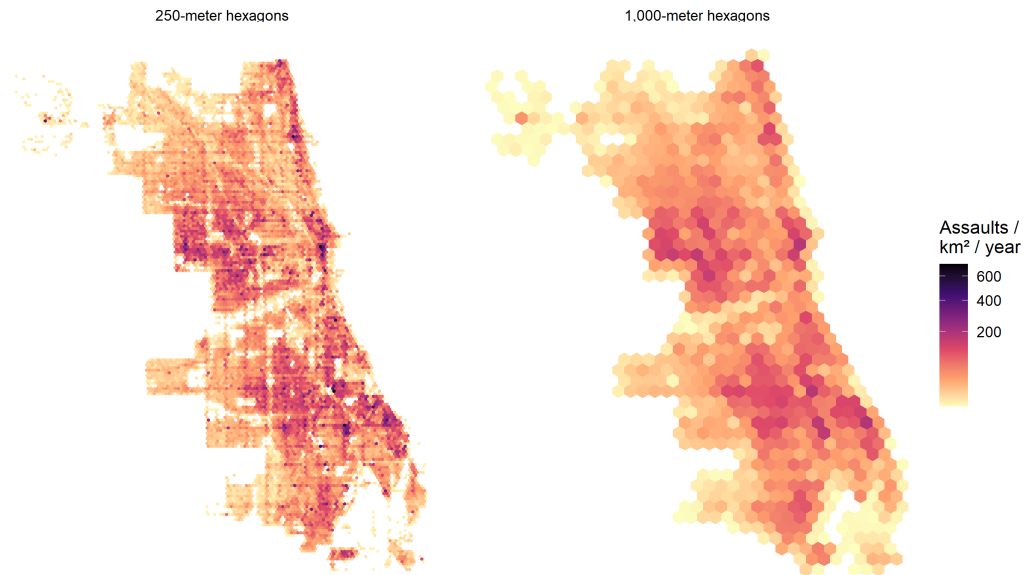


Figure 35: Reported assault density using 250-meter and 1,000-meter hexagons

The 250-meter hexagons reveal smaller concentrations and greater local variation, but isolated high-density cells may be sensitive to individual incidents. The 1,000-meter hexagons average over larger areas, creating a smoother map that can merge nearby concentrations or hide narrow features. Hotspots visible at both resolutions are less dependent on the chosen cell size.

10. Multiply the KDE bandwidth by 0.5 and by 2. Rebuild the contours and explain how the substantive story changes

To isolate the effect of the bandwidth, keep the evaluation grid, contour breaks, clipping boundary, and color scale the same in all three maps. Only the bandwidth changes.

```
bandwidthValues <- list("Half bandwidth"      = bandwidth * 0.5,
                        "Original bandwidth"   = bandwidth,
                        "Double bandwidth"     = bandwidth * 2)

xyAssaults <- sfAssaults |>
  st_coordinates() |>
  data.frame()

kdeBandwidth <- lapply(bandwidthValues,
  function(bandwidth0)
  {
    kde0 <- MASS::kde2d(
```

```

    x = xyAssaults$X,
    y = xyAssaults$Y,
    n = 200,
    h = bandwidth0)

    kde0$zKM2Year <- kde0$z * nrow(sfAssaults) * 10^6 / nYears
    kde0
  })

# Use common breaks so that the colors are comparable across maps
bandwidthBreaks <- pretty(
  c(0, max(vapply(kdeBandwidth,
                  function(kde0) max(kde0$zKM2Year),
                  numeric(1))))),
  n = 10)

bandwidthContours <- Map(
  function(kde0, bandwidthLabel)
  {
    isobands(
      kde0$x,
      kde0$y,
      kde0$zKM2Year,
      levels_low = bandwidthBreaks[-length(bandwidthBreaks)],
      levels_high = bandwidthBreaks[-1]) |>
    iso_to_sfg() |>
    st_sfc(crs = 26916) |>
    st_sf(
      levels_low = bandwidthBreaks[-length(bandwidthBreaks)],
      levels_high = bandwidthBreaks[-1],
      bandwidth = bandwidthLabel,
      geometry = _)
  },
  kdeBandwidth,
  names(kdeBandwidth)) |>
do.call(rbind, args = _)

assaultHull <- sfAssaults |>
  st_geometry() |>
  st_combine() |>
  st_concave_hull(ratio = 0.2) |>
  st_buffer(200)

```



```

bandwidthContours <- bandwidthContours |>
  st_intersection(assaultHull) |>
  mutate(
    density = (levels_low + levels_high) / 2,
    bandwidth = factor(
      bandwidth,
      levels = names(bandwidthValues)))

bandwidthContours <-
  bandwidthContours[!st_is_empty(bandwidthContours), ]

ggplot(bandwidthContours) +
  geom_sf(aes(fill = density), color = NA) +
  scale_fill_viridis_c(
    option = "magma",
    direction = -1,
    name = "Assaults /\nkm2 / year") +
  facet_wrap(~bandwidth, nrow = 1) +
  coord_sf(datum = NA) +
  theme_void()

```

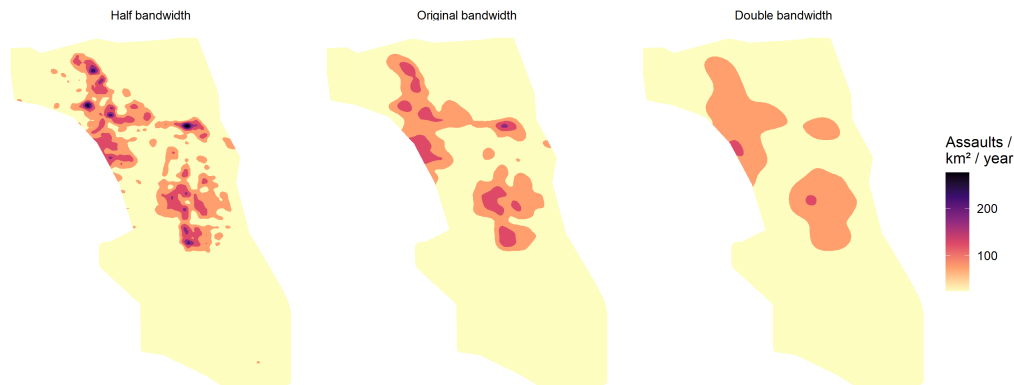


Figure 36: Assault hotspot contours using half, original, and double bandwidths

The half-bandwidth map contains more numerous, smaller peaks and sharper changes between neighboring areas. Some of those details may reflect real local concentrations, while others may be noise. Doubling the bandwidth produces broader regions with lower, more gradual peaks and can merge nearby hotspots. Concentrations that remain visible under all three bandwidths are less dependent on this modeling choice.

11. Create common-scale hex maps for motor vehicle theft in each year from 2020 through 2025.

```
vehicle_theft_sf <- crime |>
  filter(valid_coordinates &
    (year(date) %in% 2020:2025)) |>
  semi_join(iucrLookup |>
    filter(primary_type == "MOTOR VEHICLE THEFT"),
    by = "iucr") |>
  st_as_sf(coords = c("longitude", "latitude"),
    crs = 4326,
    remove = FALSE) |>
  st_transform(26916)

vehicle_hex_geometry <-
  st_make_grid(vehicle_theft_sf,
    cellsize = 500,
    square = FALSE)

vehicle_grid <- st_sf(
  hex_id = seq_along(vehicle_hex_geometry),
  geometry = vehicle_hex_geometry)

vehicle_grid_year <- lapply(2020:2025,
  function(year_i) {
    points_i <- vehicle_theft_sf |>
      filter(year(date) == year_i)

    vehicle_grid |>
      select(hex_id, geometry) |>
      mutate(
        year = year_i,
        n_thefts = lengths(st_intersects(geometry, points_i)),
        area_km2 = as.numeric(st_area(geometry)) / 10^6,
        density = n_thefts / area_km2)
  }) |>
  bind_rows()

ggplot(vehicle_grid_year) +
  geom_sf(aes(fill = density), color = NA) +
  scale_fill_viridis_c(
    option = "magma",
    trans = "sqrt",
```

```
limits = c(0, max(vehicle_grid_year$density))) +
facet_wrap(~year) +
coord_sf(datum = NA) +
theme_void()
```

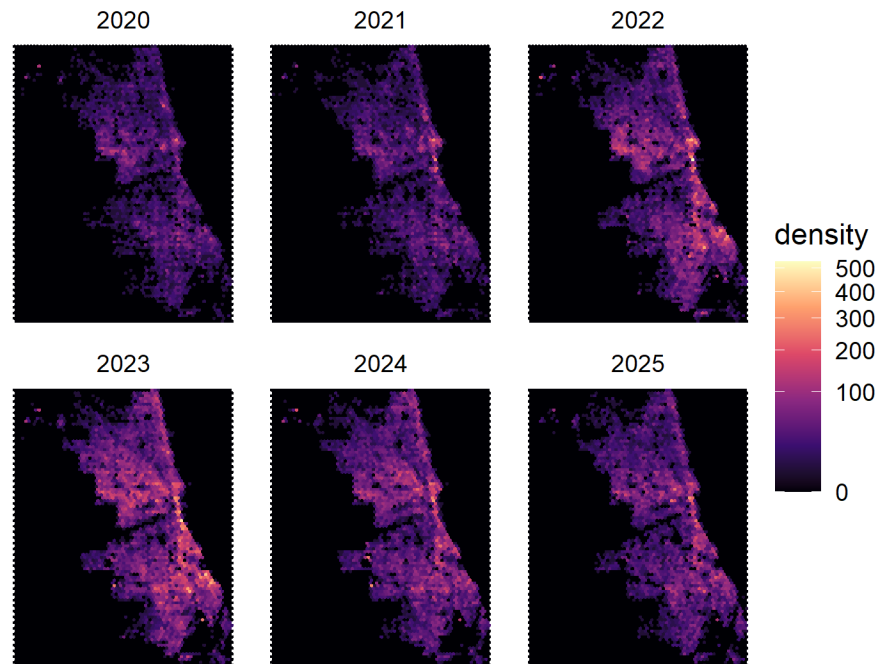


Figure 37: Annual motor vehicle theft density on a common hexagonal grid