Syllabus UNIT III

Getting Data, stdin and stdout, Reading Files: The Basics of Text files, Delimited files, Scraping the Web, HTML and the Parsing Thereof, Example: Keeping Tabs on Congress, Using APIs, JSON (and XML), Using an Unauthenticated API, Finding APIs, Example: Using the Twitter APIs, working with Data, Exploring Your Data, Using Named Tuples, Data classes, Cleaning and Munging, Manipulating Data, Rescaling, An Aside: tqdm, Dimensionality Reduction.

stdin and stdout

- If you run your Python scripts at the command line, you can pipe data through them using ***sys.stdin*** and ***sys.stdout.***
- For example, here is a script **(egrep.py)** that reads in lines of text and spits back out the ones that match a regular expression:

```
# egrep.py
import sys, re
# sys.argv is the list of command-line arguments
# sys.argv[0] is the name of the program itself
# sys.argv[1] will be the regex specified at the command line
regex = sys.argv[1]
# for every line passed into the script
for line in sys.stdin:
    # if it matches the regex, write it to stdout
    if re.search(regex, line):
        sys.stdout.write(line)
```

You have a file named `input.txt` with the following content:

```nginx
apple
banana
Apple pie
grape
pineapple
applesauce
```

```
python3 egrep.py "apple|banana" < input.txt
```

Output:

```nginx
apple
banana
pineapple
applesauce
```

```
python3 egrep.py "(?i)apple" < input.txt
```

Output:

```nginx
apple
Apple pie
pineapple
applesauce
```

stdin and stdout

- And here's one **(line_count.py)** that counts the lines it receives and then writes out the count:

```
# line_count.py
import sys
count = 0
for line in sys.stdin:
  count += 1
# print goes to sys.stdout
print count
```

Assume a file `input.txt` **with the content:**

```arduino
first line
second line
third line
```

1. Using input redirection:

```bash
python3 line_count.py < input.txt
```

Output:

```
3
```

stdin and stdout

- You could then use these to count how many lines of a file contain numbers.
- In **Windows**, you'd use:

```
type input.txt | python egrep.py "[0-9]" | python line_count.py
```

- whereas in a **Unix** system you'd use:

```
cat input.txt | python egrep.py "[0-9]" | python line_count.py
```

- The | is the pipe character, which means "use the output of the left command as the input of the right command."
- You can build pretty elaborate data-processing pipelines this way.

Most Common Words

- Python Program that counts the words in its input and writes out the most common word.

```
$ cat the_bible.txt | python most_common_words.py 10
36397   the
30031   and
20163   of
7154    to
6484    in
5856    that
5421    he
5226    his
5060    unto
4297    shall
```

```
# most_common_words.py
import sys
from collections import Counter
# pass in number of words as first argument
try:
        num_words = int(sys.argv[1])
except:
        print("usage: most_common_words.py num_words")
        sys.exit(1) # nonzero exit code indicates error
counter = Counter(word.lower() # lowercase words
                for line in sys.stdin
                for word in line.strip().split() # split on spaces
                if word) # skip empty 'words'
for word, count in counter.most_common(num_words):
        sys.stdout.write(str(count))
        sys.stdout.write("\t")
        sys.stdout.write(word)
        sys.stdout.write("\n")
```

```
words = []
for line in sys.stdin:
    for word in line.strip().split():
        if word:
            words.append(word.lower())
counter = Counter(words)
```

Reading Files

- You can also explicitly read from and write to files directly in your code.
- Python makes working with files pretty simple.

<u>The Basics of Text Files</u>

- The first step to working with a text file is to obtain a file object using ***open***:

```python
# 'r' means read-only, it's assumed if you leave it out
file_for_reading = open('reading_file.txt', 'r')
file_for_reading2 = open('reading_file.txt')

# 'w' is write -- will destroy the file if it already exists!
file_for_writing = open('writing_file.txt', 'w')

# 'a' is append -- for adding to the end of the file
file_for_appending = open('appending_file.txt', 'a')

# don't forget to close your files when you're done
file_for_writing.close()
```

Reading Files

- Because it is easy to forget to close your files, you should **always use them in a with block**, at the end of which they will be closed automatically:

```
with open(filename) as f:
    data = function_that_gets_data_from(f)

# at this point f has already been closed, so don't try to use it
process(data)
```

with statement, is Python's context manager — used to automatically manage resources like files, network connections, or locks.

Reading Files

- If you need to read a whole text file, you can just iterate over the lines of the file using ***for.***
- Every line you get this way ends in a newline character, so you'll often want to ***strip*** it before doing anything with it.

```
starts_with_hash = 0
with open('input.txt') as f:
for line in f: # look at each line in the file
        if re.match("^#",line): # use a regex to see if it starts with '#'
                starts_with_hash += 1 # if it does, add 1 to the count
```

```
# This is a comment
This is regular text
# Another comment
Something else
#One more comment
Not a comment
```

```
Lines starting with #: 3
```

Reading Files

- **Python Program to access a file full of email addresses, one per line, and count the number of the domains.**

```
from collections import Counter

def get_domain(email_address: str) -> str:
    """Split on '@' and return the last piece"""
    return email_address.lower().split("@")[-1]
    # a couple of tests
assert get_domain('joelgrus@gmail.com') == 'gmail.com'
assert get_domain('joel@m.datasciencester.com') == 'm.datasciencester.com'
with open('email_addresses.txt', 'r') as f:
    domain_counts = Counter(get_domain(line.strip())
                            for line in f
                            if "@" in line)
```

Delimited Files

- The hypothetical email addresses file we just processed had one address per line.
- More frequently you'll work with files with lots of data on each line.
- These files are very often either ***comma-separated*** or ***tab-separated***: each line has several fields, with a comma or a tab indicating where one field ends and the next field starts.
- This starts to get complicated when you have fields with commas and tabs and newlines in them (which you inevitably will).
- For this reason, you should never try to parse them yourself.
- Instead, you should use Python's csv module.

Tab-delimited

- If your file has no headers (which means you probably want each row as a list, and which places the burden on you to know what's in each column), you can use ***csv.reader*** to iterate over the rows, each of which will be an appropriately split list.
- For example, if we had a tab-delimited file of stock prices:

```
6/20/2014	AAPL	90.91
6/20/2014	MSFT	41.68
6/20/2014	FB	64.5
6/19/2014	AAPL	91.86
6/19/2014	MSFT	41.51
6/19/2014	FB	64.34
```

Tab-delimited

Python code snippet to access and process csv file using tab delimiter.

```
def process(date, symbol, closing_price):
    print(f"Date: {date}, Symbol: {symbol}, Closing Price: 
{closing_price}")

# Open the tab-delimited stock prices file
with open('tab_delimited_stock_prices.txt', 'r') as f:
    tab_reader = csv.reader(f, delimiter='\t')

    for row in tab_reader:
        date = row[0]
        symbol = row[1]
        closing_price = float(row[2])

        process(date, symbol, closing_price)
```

csv with headers

```
date:symbol:closing_price
6/20/2014:AAPL:90.91
6/20/2014:MSFT:41.68
6/20/2014:FB:64.5
```

- If your file has headers:

- you can either skip the header row with an initial call to ***reader.next***, or get each row as a ***dict*** (with the headers as keys) by using ***csv.DictReader***:

- **Python code snippet to access and process csv file having header.**

```python
with open('colon_delimited_stock_prices.txt') as f:
    colon_reader = csv.DictReader(f, delimiter=':')
    for dict_row in colon_reader:
        date = dict_row["date"]
        symbol = dict_row["symbol"]
        closing_price = float(dict_row["closing_price"])
        process(date, symbol, closing_price)
```

- Even if your file doesn't have headers, you can still use ***DictReader*** by passing it the keys as a ***fieldnames*** parameter.

csv writer

- You can similarly write out delimited data using ***csv.writer***:
- **Python code snippet to write out delimited data using csv.writer**

```python
todays_prices = {'AAPL': 90.91, 'MSFT': 41.68, 'FB': 64.5 }
with open('comma_delimited_stock_prices.txt', 'w') as f:
    csv_writer = csv.writer(f, delimiter=',')
    for stock, price in todays_prices.items():
        csv_writer.writerow([stock, price])
```

```
symbol,price
AAPL,90.91
MSFT,41.68
FB,64.5
```

- ***csv.writer*** will do the right thing if your fields themselves have commas in them.

Scraping the Web

- Another way to get data is by scraping it from web pages.
- Fetching web pages, it turns out, is pretty easy; getting meaningful structured information out of them less so.

HTML and the Parsing Thereof

- Pages on the web are written in HTML, in which text is (ideally) marked up into elements and their attributes:

```html
<html>
  <head>
    <title>A web page</title>
  </head>
  <body>
    <p id="author">Joel Grus</p>
    <p id="subject">Data Science</p>
  </body>
</html>
```

HTML and the Parsing Thereof

- In a perfect world, where all web pages were marked up semantically for our benefit, we would be able to extract data using rules like “find the **<p>** element whose **id** is **subject** and return the text it contains.”
- In the actual world, HTML is not generally well formed.
- To get data out of HTML, we will use the Beautiful Soup library, which builds a tree out of the various elements on a web page and provides a simple interface for accessing them.
- Requests library, is used for making HTTP requests.
- Python’s built-in HTML parser is not that lenient, as it doesn’t always cope well with HTML that’s not perfectly formed.
- For that reason, we’ll also install the html5lib parser.

BeautifulSoup

- To use Beautiful Soup, we pass a string containing HTML into the **BeautifulSoup** function.
- In our examples, this will be the result of a call to **requests.get**: after which we can get pretty far using a few simple methods.

```python
from bs4 import BeautifulSoup
import requests
# I put the relevant HTML file on GitHub. In order to fit
# the URL in the book I had to split it across two lines.
# Recall that whitespace-separated strings get concatenated.
url = ("https://raw.githubusercontent.com/"
       "joelgrus/data/master/getting-data.html")
html = requests.get(url).text
soup = BeautifulSoup(html, 'html5lib')
```

BeautifulSoup

- We'll typically work with Tag objects, which correspond to the tags representing the structure of an HTML page.
- For example, to find the first **<p>** tag (and its contents), you can use:

```
first_paragraph = soup.find('p')
print(first_paragraph)
```

```
0.4s
<p id="p1">This is the first paragraph.</p>
```

- You can get the text contents of a Tag using its text property:

```
first_paragraph_text = soup.p.text
first_paragraph_words = soup.p.text.split()
print(first_paragraph_text)
print(first_paragraph_words)
```

```
0.5s
This is the first paragraph.
['This', 'is', 'the', 'first', 'paragraph.']
```

BeautifulSoup

- And you can extract a tag's attributes by treating it like a dict:

```
first_paragraph_id = soup.p['id'] # raises KeyError if no 'id'
print(first_paragraph_id)
first_paragraph_id2 = soup.p.get('id') # returns None if no 'id'
print(first_paragraph_id2)
```

✓ 0.5s

```
p1
p1
```

- You can get multiple tags at once as follows:

```
all_paragraphs = soup.find_all('p') # or just soup('p')
print(all_paragraphs)
paragraphs_with_ids = [p for p in soup('p') if p.get('id')]
print(paragraphs_with_ids)
```

✓ 0.1s

```
[<p id="p1">This is the first paragraph.</p>, <p class="important">This is the second paragraph.</p>]
[<p id="p1">This is the first paragraph.</p>]
```

BeautifulSoup

- Frequently, you’ll want to find tags with a specific class

```
important_paragraphs = soup('p', {'class' : 'important'})
important_paragraphs2 = soup('p', 'important')
important_paragraphs3 = [p for p in soup('p')
                         if 'important' in p.get('class', [])]
print(important_paragraphs)
print(important_paragraphs2)
print(important_paragraphs3)
```

✓ 0.4s

```
[<p class="important">This is the second paragraph.</p>]
[<p class="important">This is the second paragraph.</p>]
[<p class="important">This is the second paragraph.</p>]
```

BeautifulSoup

- And you can combine these methods to implement more elaborate logic.
- For example, if you want to find every <span> element that is contained inside a <div> element, you could do this:

```python
# Warning: will return the same <span> multiple times
# if it sits inside multiple <div>s.
# Be more clever if that's the case.
spans_inside_divs = [span for div in soup('div') # for each <div> on the page
                     for span in div('span')] # find each <span> inside it
print(spans_inside_divs)
```

✓ 0.6s

```
[<span id="name">Joel</span>, <span id="twitter">@joelgrus</span>, <span id="email">joelgrus-at-gmail</span>]
```

Using APIs

- Many websites and web services provide application programming interfaces (APIs), which allow you to explicitly request data in a structured format.
- This saves you the trouble of having to scrape them!

JSON

- Because HTTP is a protocol for transferring text, the data you request through a web API needs to be serialized into a string format.
- Often this serialization uses *JavaScript Object Notation (JSON).*
- JavaScript objects look quite similar to Python ***dicts***, which makes their string representations easy to interpret
- We can parse JSON using Python's ***json*** module.
- In particular, we will use its ***loads*** function, which deserializes a string representing a JSON object into a Python object.

JSON

```python
import json

serialized = """{
    "title" : "Data Science Book",
    "author" : "Joel Grus",
    "publicationYear" : 2019,
    "topics" : [ "data", "science", "data science" ]
}"""

# Parse the JSON to create a Python dict
deserialized = json.loads(serialized)

if deserialized["publicationYear"] == 2019:
    print("Success!")
else:
    print("Failed!")

if "data science" in deserialized["topics"]:
    print("Success!")
else:
    print("Failed!")
```

```
Success!
Success!
```

XML

```python
# XML data as a string
xml_data = """
<book>
    <title>Data Science Book</title>
    <author>Joel Grus</author>
    <publicationYear>2019</publicationYear>
    <topics>
        <topic>data</topic>
        <topic>science</topic>
        <topic>data science</topic>
    </topics>
</book>
"""


# Parse the XML
root = ET.fromstring(xml_data)

# Extract values
publication_year = int(root.find("publicationYear").text)
topics = [topic.text for topic in root.find("topics").findall("topic")]

# Check values
if publication_year == 2019:
    print("Success!")
else:
    print("Failed!")

if "data science" in topics:
    print("Success!")
else:
    print("Failed!")
```

```
Success!
Success!
```

Using an Unauthenticated API

- Most APIs these days require that you first authenticate yourself before you can use them.
- Accordingly, we'll start by taking a look at GitHub's API, with which you can do some simple things unauthenticated:
- At this point repos is a list of Python dicts, each representing a public repository in my GitHub account.
- (Substitute your username and get your GitHub repository data instead. You do have a GitHub account, right?)

Python program to access Github account and list the repositories, find out which months and days of the week the repository is created and list the languages of last five repositories

```
from collections import Counter
from dateutil.parser import parse
import requests, json
github_user = "SarangSpin"
endpoint = f"https://api.github.com/users/{github_user}/repos"
repos = json.loads(requests.get(endpoint).text)
dates = [parse(repo["created_at"]) for repo in repos]
month_counts = Counter(date.month for date in dates)
weekday_counts = Counter(date.weekday() for date in dates)
print(dates)
print(month_counts)
print(weekday_counts)
last_5_repositories = sorted(repos, key=lambda r: r["pushed_at"], reverse=True)[:5]
last_5_languages = [repo["language"] for repo in last_5_repositories]
print(last_5_languages)
```

Using an Unauthenticated API

```
[datetime.datetime(2024, 4, 3, 1, 40, 57, tzinfo=tzlocal()), datetime.datetime(2023, 11, 15, 10, 33, 38, tzinfo=tzlocal()),
Counter({9: 4, 11: 3, 2: 3, 8: 2, 12: 2, 7: 2, 4: 1, 3: 1, 5: 1})
Counter({5: 5, 4: 4, 2: 3, 0: 3, 3: 2, 6: 2})
['Dart', 'JavaScript', 'HTML', 'C++', 'JavaScript']
```

Exploring Your Data

- After you've identified the questions you're trying to answer and have gotten your hands on some data, you might be tempted to dive in and immediately start building models and getting answers.
- But you should resist this urge.
- Your first step should be to explore your data.

Exploring One-Dimensional Data

- The simplest case is when you have a one-dimensional dataset, which is just a collection of numbers.

2. Number of Times Each of a Collection of Data Science Tutorial Videos Was Watched

This refers to video analytics:

Video Title	Views
"Intro to Python for Data Science"	12,345
"Linear Regression Tutorial"	8,902
"Deep Learning with PyTorch"	5,678

Useful for:

- Identifying popular topics
- Improving content strategy
- Recommending content

Exploring One-Dimensional Data

📚 3. Number of Pages of Each of the Data Science Books in Your Library

This relates to metadata of resources:

Book Title	Pages
"Hands-On Machine Learning"	568
"Python Data Science Handbook"	510
"Deep Learning with Python"	384

Useful for:

- Estimating reading time
- Categorizing books by complexity
- Filtering search results

Exploring One-Dimensional Data

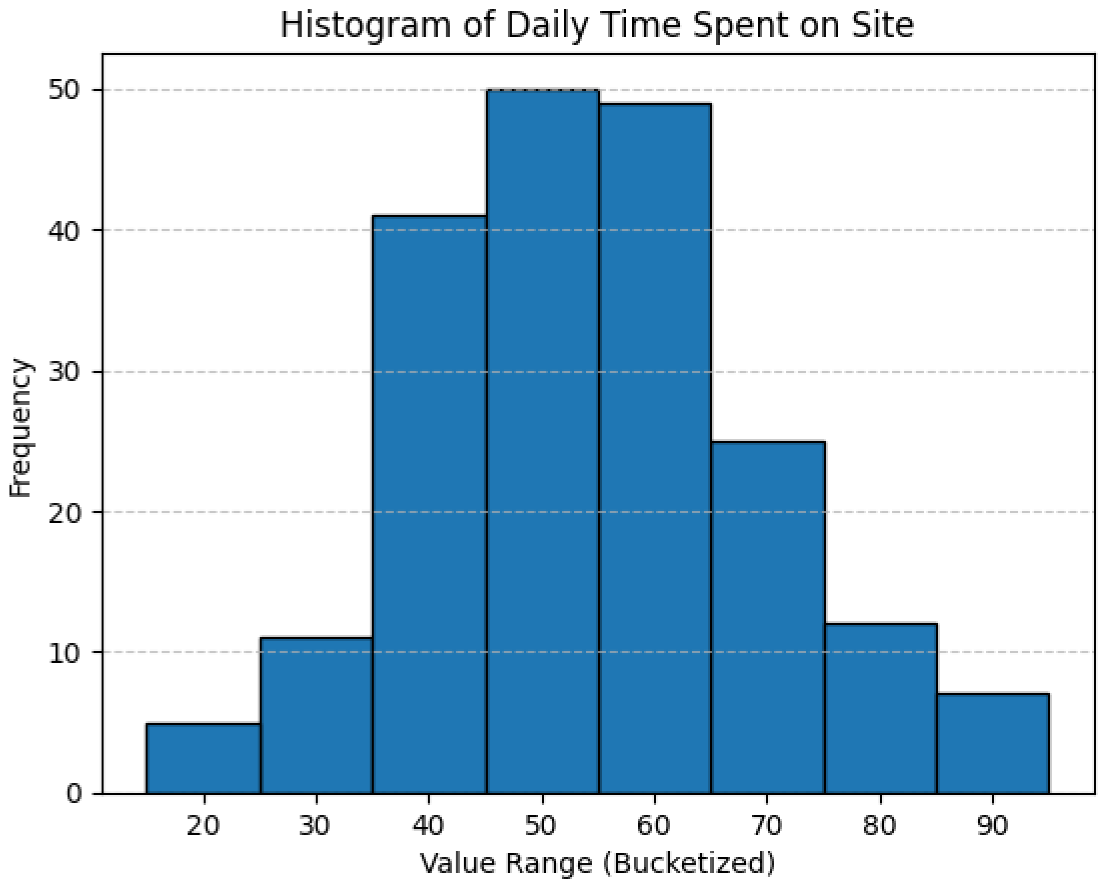
- An obvious first step is to compute a **few summary statistics.**
- You'd like to know how many data points you have, the **smallest, the largest, the mean, and the standard deviation.**
- But even these don't necessarily give you a great understanding.
- A good next step is to create a **histogram**, in which you group your data into discrete buckets and count how many points fall into each bucket:

Exploring One-Dimensional Data

```
import math
import matplotlib.pyplot as plt
from typing import List, Dict
from collections import Counter
import random
def bucketize(point: float, bucket_size: float) -> float:
    """Floor the point to the next lower multiple of bucket_size"""
    return bucket_size * math.floor(point / bucket_size)
def make_histogram(points: List[float], bucket_size: float) -> Dict[float, int]:
    """Buckets the points and counts how many in each bucket"""
    return Counter(bucketize(point, bucket_size) for point in points)
def plot_histogram(points: List[float], bucket_size: float, title: str = ""):
    """Plots the histogram"""
    histogram = make_histogram(points, bucket_size)
    plt.bar(histogram.keys(), histogram.values(), width=bucket_size,
edgecolor='black')
    plt.xlabel("Value Range (Bucketized)")
    plt.ylabel("Frequency")
    plt.title(title)
    plt.grid(axis='y', linestyle='--', alpha=0.7)
    plt.show()
# Simulate data: e.g., minutes spent on a site by users in a day
random.seed(0)
data_points = [random.gauss(60, 15) for _ in range(200)]  # mean=60 mins, stddev=15
bucket_size = 10
plot_histogram(data_points, bucket_size, title="Histogram of Daily Time Spent on
Site")
```

Exploring One-Dimensional Data

- shows the distribution:



Two Dimensions

- Now imagine you have a dataset with two dimensions. Maybe in addition to daily minutes you have years of data science experience.
- Of course you'd want to understand each dimension individually.
- But you probably also want to scatter the data.
- For example, consider another fake dataset:

```
def random_normal() -> float:
        """Returns a random draw from a standard normal
distribution"""
        return inverse_normal_cdf(random.random())
xs = [random_normal() for _ in range(1000)]
ys1 = [ x + random_normal() / 2 for x in xs]
ys2 = [-x + random_normal() / 2 for x in xs]
```

Two Dimensions

```
plt.scatter(xs, ys1, marker='.', color='black', label='ys1')
plt.scatter(xs, ys2, marker='.', color='gray', label='ys2')
plt.xlabel('xs')
plt.ylabel('ys')
plt.legend(loc=9)
plt.title("Very Different Joint Distributions")
plt.show()
```

```
from scratch.statistics import correlation
print(correlation(xs, ys1)) # about 0.9
print(correlation(xs, ys2)) # about -0.9
```

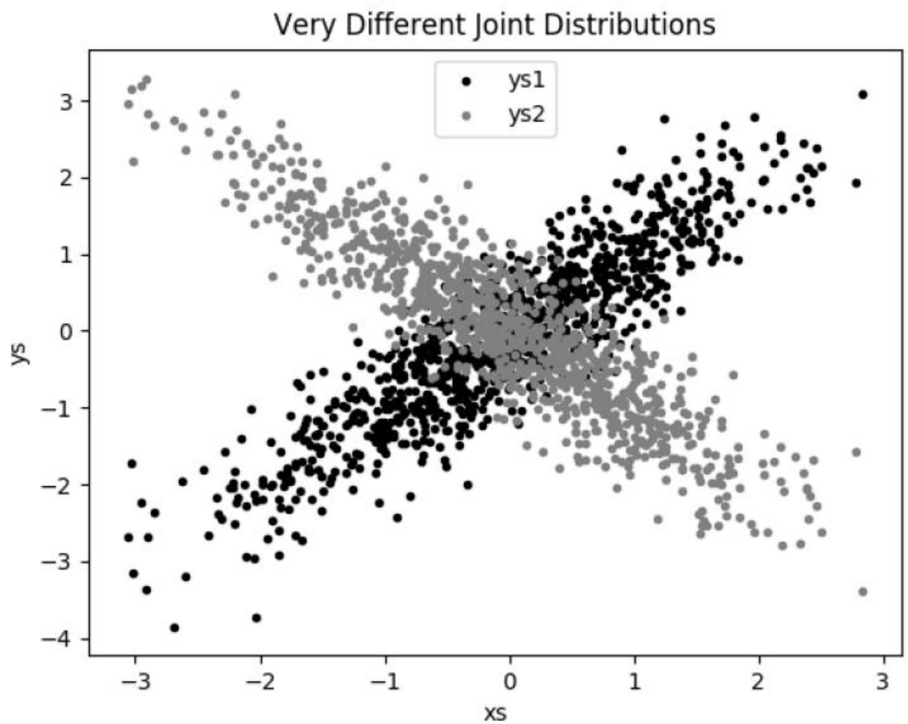


Figure 10-3. Scattering two different ys

Many Dimensions

- With many dimensions, you'd like to know how all the dimensions relate to one another.
- Use correlation matrix, in which the entry in row i and column j is the correlation between the ith dimension and the jth dimension of the data:

📌 **Formula for Pearson Correlation Coefficient**

Given two variables (or vectors) **X** = [x_1, x_2, ..., x_n] and **Y** = [y_1, y_2, ..., y_n], the formula is:

$$\text{correlation}(X, Y) = r = \frac{\sum_{i=1}^{n}(x_i - \bar{x})(y_i - \bar{y})}{(n-1)\cdot s_x \cdot s_y}$$

Where:

- $\bar{x}$ = mean of **X**
- $\bar{y}$ = mean of **Y**
- s_x = standard deviation of **X**
- s_y = standard deviation of **Y**
- n = number of data points

Many Dimensions

```
from scratch.linear_algebra import Matrix, Vector, make_matrix

def correlation_matrix(data: List[Vector]) -> Matrix:
    """
    Returns the len(data) x len(data) matrix whose (i, j)-th entry
    is the correlation between data[i] and data[j]
    """

    def correlation_ij(i: int, j: int) -> float:
        return correlation(data[i], data[j])

    return make_matrix(len(data), len(data), correlation_ij)
```

Many Dimensions

```
data = [
    [90, 85, 88, 92],
    [60, 80, 55, 75],
    [88, 82, 86, 90]
]
```

```
Correlation Matrix:
['1.00', '0.57', '1.00']
['0.57', '1.00', '0.57']
['1.00', '0.57', '1.00']
```

Interpretation:

- **Diagonal = 1.00**: Each vector is perfectly correlated with itself.
- **Student A & C = 1.00**: Their scores are nearly identical.
- **Student A & B = 0.61**: Somewhat positively correlated but not perfe
- The matrix is **symmetric**, i.e., `corr(i, j) == corr(j, i)`

Many Dimensions

- A more visual approach (if you don't have too many dimensions) is to make a ***scatterplot matrix*** showing all the pairwise scatterplots.
- To do that we'll use ***plt.subplots,*** which allows us to create subplots of our chart.
- We give it the number of rows and the number of columns, and it returns a ***figure*** object (which we won't use) and a two dimensional array of ***axes*** objects (each of which we'll plot to):

```python
from typing import List

# Just some random data to show off correlation scatterplots
num_points = 100

def random_row() -> List[float]:
    row = [0.0, 0, 0, 0]
    row[0] = random_normal()
    row[1] = -5 * row[0] + random_normal()
    row[2] = row[0] + row[1] + 5 * random_normal()
    row[3] = 6 if row[2] > -2 else 0
    return row

random.seed(0)
# each row has 4 points, but really we want the columns
corr_rows = [random_row() for _ in range(num_points)]

corr_data = [list(col) for col in zip(*corr_rows)]
```

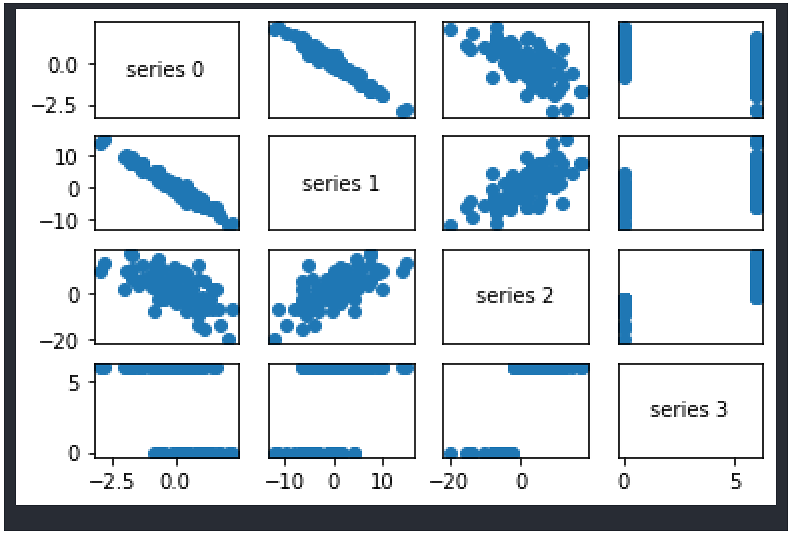
Many Dimensions

```python
# corr_data is a list of four 100-d vectors
num_vectors = len(corr_data)
fig, ax = plt.subplots(num_vectors, num_vectors)
for i in range(num_vectors):
    for j in range(num_vectors):
        # Scatter column_j on the x-axis vs. column_i on the y-axis
        if i != j:
            ax[i][j].scatter(corr_data[j], corr_data[i])
        # unless i == j, in which case show the series name
        else:
            ax[i][j].annotate("series " + str(i), (0.5, 0.5),
                              xycoords='axes fraction',
                              ha="center", va="center")

        # Then hide axis labels except left and bottom charts
        if i < num_vectors - 1:
            ax[i][j].xaxis.set_visible(False)
        if j > 0:
            ax[i][j].yaxis.set_visible(False)

# Fix the bottom-right and top-left axis labels, which are wrong because
# their charts only have text in them
ax[-1][-1].set_xlim(ax[0][-1].get_xlim())
ax[0][0].set_ylim(ax[0][1].get_ylim())
plt.show()
```

Many Dimensions



- Looking at the scatterplots, you can see that series 1 is very negatively correlated with series 0, series 2 is positively correlated with series 1, and series 3 only takes on the values 0 and 6, with 0 corresponding to small values of series 2 and 6 corresponding to large values.
- This is a quick way to get a rough sense of which of your variables are correlated.

Using NamedTuples

- One common way of representing data is using **dicts**:

```
import datetime
stock_price = {'closing_price': 102.06,
               'date': datetime.date(2014, 8, 29),
               'symbol': 'AAPL'}
```

- There are several reasons why this is less than ideal, however.

1. This is a slightly inefficient representation (a dict involves some overhead), so that if you have a lot of stock prices they'll take up **more memory** than they have to.
2. A larger issue is that accessing things by ***dict*** key is **error-prone.**

Using NamedTuples

3. Can type-annotate only for uniform dictionaries:

```
prices: Dict[datetime.date, float] = {}
```

- No helpful way to annotate dictionaries-as-data that have lots of different value types.
- So we also lose the power of type hints.
- As an alternative, Python includes a ***namedtuple*** class, which is like a tuple but with named slots:

NamedTuples

- A tuple is an immutable, ordered sequence of elements. Elements are accessed using indexes.

```
# Using a tuple
person = ('Alice', 30, 'Engineer')

# Accessing elements
print(person[0])  # Output: Alice
print(person[1])  # Output: 30
```

- A namedtuple is like a regular tuple, but with named fields for better readability and self-documentation. It is defined using collections

Using NamedTuples

namedtuple is a function from Python's collections module that creates tuple subclasses with named fields. It lets you access tuple elements using dot notation instead of only by index — making your code more readable. The syntax is as follows:

```
from typing import NamedTuple

class ClassName(NamedTuple):
    field1_name: field1_type
    field2_name: field2_type
    field3_name: field3_type
    # You can add as many fields as needed

    # Optional: define methods inside the class
    def method_name(self) -> return_type:
        # method logic here
        return something

# Example usage
obj = ClassName(field1_value, field2_value, field3_value)
```

Using NamedTuples: Program

```
from typing import NamedTuple
import datetime
class StockPrice(NamedTuple):
    symbol: str
    date: datetime.date
    closing_price: float

    def is_high_tech(self) -> bool:
        """Check if the stock symbol belongs to a high-tech company."""
        return self.symbol in ['MSFT', 'GOOG', 'FB', 'AMZN', 'AAPL']
# Create a StockPrice instance
price = StockPrice('MSFT', datetime.date(2018, 12, 14), 106.03)
# Assertions to check correctness
assert price.symbol == 'MSFT'
assert price.closing_price == 106.03
assert price.is_high_tech()
# Print details
print(f"Symbol: {price.symbol}")
print(f"Date: {price.date}")
print(f"Closing Price: {price.closing_price}")
print(f"Is High Tech: {price.is_high_tech()}")
```

Using NamedTuples: Code Snippet

```
from typing import NamedTuple
class StockPrice(NamedTuple):
    symbol: str
    date: datetime.date
    closing_price: float
    def is_high_tech(self) -> bool:
    """It's a class, so we can add methods too"""
    return self.symbol in ['MSFT', 'GOOG', 'FB', 'AMZN', 'AAPL']
price = StockPrice('MSFT', datetime.date(2018, 12, 14), 106.03)
assert price.symbol == 'MSFT'
assert price.closing_price == 106.03
assert price.is_high_tech()
```

Dataclasses

- Dataclasses are (sort of) a mutable version of NamedTuple.
- The syntax is very similar to NamedTuple.
- But instead of inheriting from a base class, it uses a decorator.
- Syntax

```
from dataclasses import dataclass

@dataclass
class ClassName:
    field1: type
    field2: type
    field3: type = default_value  # optional default
```

Dataclasses: Code Snippet

```
from dataclasses import dataclass
@dataclass
class StockPrice2:
    symbol: str
    date: datetime.date
    closing_price: float
    def is_high_tech(self) -> bool:
    """It's a class, so we can add methods too"""
    return self.symbol in ['MSFT', 'GOOG', 'FB', 'AMZN', 'AAPL']
price2 = StockPrice2('MSFT', datetime.date(2018, 12, 14),
106.03)
assert price2.symbol == 'MSFT'
assert price2.closing_price == 106.03
assert price2.is_high_tech()
```

Dataclasses

- As mentioned, the big difference is that we can modify a dataclass instance's values:

```
# stock split
price2.closing_price /= 2
assert price2.closing_price == 51.03
```

- If we tried to modify a field of the NamedTuple version, we'd get an AttributeError.

Cleaning and Munging

- Real-world data is dirty. Often you'll have to do some work on it before you can use it.

Data cleaning means fixing or removing incorrect, inconsistent, incomplete, or duplicate data to improve data quality.

Typical tasks:

•Handle **missing values** (e.g., fill with mean/median, or drop rows).

•Correct **data entry errors** (e.g., "1234" instead of "1,234").

•Remove **duplicate rows or records**.

•Standardize **formats** (e.g., date formats, currency units).

•Fix **inconsistent labels** (e.g., "USA" vs. "United States").

Data Munging (also called Wrangling)

Data munging (or wrangling) is the process of transforming raw data into a useful format for analysis or modelling.

Typical tasks:

•Convert data types (e.g., string to number, timestamp).

•Merge or join multiple datasets.

•Create **new features** or columns.

•Filter and subset data to focus on relevant parts.

•Normalize or scale data.

Cleaning and Munging

For example, if we have comma-delimited stock prices with bad data:

```
AAPL,6/20/2014,90.91
MSFT,6/20/2014,41.68
FB,6/20/3014,64.5
AAPL,6/19/2014,91.86
MSFT,6/19/2014,n/a
FB,6/19/2014,64.34
```

Cleaning and Munging:Code Snippet

```
from typing import Optional
import re
def try_parse_row(row: List[str]) -> Optional[StockPrice]:
    symbol, date_, closing_price_ = row
    # Stock symbol should be all capital letters
    if not re.match(r"^[A-Z]+$", symbol):
        return None
    try:
        date = parse(date_).date()
    except ValueError:
        return None
    try:
        closing_price = float(closing_price_)
    except ValueError:
        return None
    return StockPrice(symbol, date, closing_price)
# Should return None for errors
assert try_parse_row(["MSFT0", "2018-12-14", "106.03"]) is None
assert try_parse_row(["MSFT", "2018-12--14", "106.03"]) is None
assert try_parse_row(["MSFT", "2018-12-14", "x"]) is None
# But should return same as before if data is good
assert try_parse_row(["MSFT", "2018-12-14", "106.03"]) == stock
```

Cleaning and Munging:Code Snippet

```
import csv
data: List[StockPrice] = []
with open("comma_delimited_stock_prices.csv") as f:
    reader = csv.reader(f)
    for row in reader:
        maybe_stock = try_parse_row(row)
        if maybe_stock is None:
                print(f"skipping invalid row: {row}")
        else:
                data.append(maybe_stock)
```

Manipulating Data

- One of the most important skills of a data scientist is manipulating data.
- It's more of a general approach than a specific technique, so we'll just work through a handful of examples to give you the flavor of it.
- Imagine we have a bunch of stock price data that looks like this:

```
data = [
    StockPrice(symbol='MSFT',
               date=datetime.date(2018, 12, 24),
               closing_price=106.03),
    # ...
]
```

Manipulating Data

- What are the largest and smallest one-day percent changes in our dataset.
- The percent change is **price_today / price_yesterday - 1**, which means we need some way of associating today's price and yesterday's price.

Manipulating Data

```
from typing import NamedTuple
import datetime
# Define a stock price record
class StockPrice(NamedTuple):
    symbol: str
    date: datetime.date
    closing_price: float
# Function to calculate percent change
def pct_change(yesterday: StockPrice, today: StockPrice) -> float:
    return round((today.closing_price / yesterday.closing_price - 1) * 100, 2)
# Simulated two-day data for AAPL
yesterday = StockPrice(symbol="AAPL", date=datetime.date(2025, 5, 5), closing_price=180.0)
today = StockPrice(symbol="AAPL", date=datetime.date(2025, 5, 6), closing_price=186.3)
# Calculate and display percentage change
change = pct_change(yesterday, today)
print(f"Stock: {today.symbol}")
print(f"Date: {today.date}")
print(f"Day-over-day % change: {change}%")
```

Manipulating Data

- Since the prices are tuples, they'll get sorted by their fields in order: first by symbol, then by date, then by price.
- This means that if we have some prices all with the same symbol, sort will sort them by date (and then by price, which does nothing, since we only have one per date), which is what we want.

```python
from typing import List
from collections import defaultdict
# Collect the prices by symbol
prices: Dict[str, List[StockPrice]] = defaultdict(list)

for sp in data:
    prices[sp.symbol].append(sp)
```

Manipulating Data

- which we can use to compute a sequence of day-over-day changes:

```
def pct_change(yesterday: StockPrice, today: StockPrice) -> float:
    return today.closing_price / yesterday.closing_price - 1


class DailyChange(NamedTuple):
    symbol: str
    date: datetime.date
    pct_change: float

def day_over_day_changes(prices: List[StockPrice]) -> List[DailyChange]:
    """
    Assumes prices are for one stock and are in order
    """
    return [DailyChange(symbol=today.symbol,
                        date=today.date,
                        pct_change=pct_change(yesterday, today))
            for yesterday, today in zip(prices, prices[1:])]
```

Manipulating Data

- and then collect them all:

```
all_changes = [change
               for symbol_prices in prices.values()
               for change in day_over_day_changes(symbol_prices)]
```

Manipulating Data

- At which point it's easy to find the largest and smallest:

```
max_change = max(all_changes, key=lambda change: change.pct_change)
# see, e.g. http://news.cnet.com/2100-1001-202143.html
print("Success!" if max_change.symbol == 'AAPL' else "Failure!")
print("Success!" if max_change.date == datetime.date(1997, 8, 6) else "Failure!")
print("Success!" if 0.33 < max_change.pct_change < 0.34 else "Failure!")

min_change = min(all_changes, key=lambda change: change.pct_change)
# see, e.g. http://money.cnn.com/2000/09/29/markets/techwrap/
print("Success!" if min_change.symbol == 'AAPL' else "Failure!")
print("Success!" if min_change.date == datetime.date(2000, 9, 29) else "Failure!")
print("Success!" if -0.52 < min_change.pct_change < -0.51 else "Failure!")
```

✓ 0.7s

```
Success!
Success!
Success!
Success!
Success!
Success!
```

Manipulating Data

- We can now use this new all_changes dataset to find which month is the best to invest in tech stocks. We'll just look at the average daily change by month:

```
changes_by_month: List[DailyChange] = {month: [] for month in range(1, 13)}
for change in all_changes:
    changes_by_month[change.date.month].append(change)
avg_daily_change = {
    month: sum(change.pct_change for change in changes) / len(changes)
    for month, changes in changes_by_month.items()
}
# October is the best month
print("Success!" if avg_daily_change[10] == max(avg_daily_change.values()) else "Failure!")

```

✓ 0.4s

```
Success!
```

Rescaling

- Many techniques are sensitive to the scale of your data. For example, imagine that you have a dataset consisting of the heights and weights of hundreds of data scientists, and that you are trying to identify clusters of body sizes.
- Intuitively, we'd like clusters to represent points near each other, which means that we need some notion of distance between points.
- We already have a Euclidean distance function, so a natural approach might be to treat (height, weight) pairs as points in two-dimensional space.
- Consider the people listed in Table (next slide)

Rescaling

Heights and weights

Person	Height (inches)	Height (centimeters)	Weight (pounds)
A	63	160	150
B	67	170.2	160
C	70	177.8	171

Rescaling

- If we measure height in inches, then B's nearest neighbor is A:

```python
from linear_algebra import distance
a_to_b = distance([63, 150], [67, 160]) # 10.77
a_to_c = distance([63, 150], [70, 171]) # 22.14
b_to_c = distance([67, 160], [70, 171]) # 11.40
```

- However, if we measure height in centimeters, then B's nearest neighbor is instead C:

```python
a_to_b = distance([160, 150], [170.2, 160]) # 14.28
a_to_c = distance([160, 150], [177.8, 171]) # 27.53
b_to_c = distance([170.2, 160], [177.8, 171]) # 13.37
```

Rescaling

- Obviously it's a problem if changing units can change results like this.
- For this reason, when dimensions aren't comparable with one another, we will sometimes rescale our data so that each dimension has mean 0 and standard deviation 1.
- This effectively gets rid of the units, converting each dimension to "standard deviations from the mean."
- To start with, we'll need to compute the mean and the standard_deviation for each position:

Rescaling

```
from typing import Tuple
from scratch.linear_algebra import vector_mean
from scratch.statistics import standard_deviation
def scale(data: List[Vector]) -> Tuple[Vector, Vector]:
    """returns the mean and standard deviation for each position"""
    dim = len(data[0])
    means = vector_mean(data)
    stdevs = [standard_deviation([vector[i] for vector in data])
              for i in range(dim)]
    return means, stdevs
vectors = [[-3, -1, 1], [-1, 0, 1], [1, 1, 1]]
means, stdevs = scale(vectors)
assert means == [-1, 0, 1]
assert stdevs == [2, 1, 0]
```

Rescaling

```python
def rescale(data: List[Vector]) -> List[Vector]:
    """
    Rescales the input data so that each position has
    mean 0 and standard deviation 1. (Leaves a position
    as is if its standard deviation is 0.)
    """
    dim = len(data[0])
    means, stdevs = scale(data)
    # Make a copy of each vector
    rescaled = [v[:] for v in data]
    for v in rescaled:
        for i in range(dim):
            if stdevs[i] > 0:
                v[i] = (v[i] - means[i]) / stdevs[i]
    return rescaled
```

Rescaling

- Of course, let's write a test to conform that rescale does what we think it should:

```
means, stdevs = scale(rescale(vectors))
assert means == [0, 0, 1]
assert stdevs == [1, 1, 0]
```

An Aside: tqdm

- Frequently we'll end up doing computations that take a **long time**.
- When you're doing such work, you'd like to know that you're **making progress** and how long you should expect to wait.
- One way of doing this is with the **tqdm library**, which generates **custom progress bars**.
- To start with, you'll need to install it:

python -m pip install tqdm

tqdm

- There are only a few features you need to know about.
- The first is that an iterable wrapped in tqdm.tqdm will produce a progress bar:
- which produces an output that looks like this:

```
import tqdm
for i in tqdm.tqdm(range(100)):
    # do something slow
    _ = [random.random() for _ in range(1000000)]
```

✓ 11.8s

```
39%|████      | 39/100 [00:04<00:08,  7.33it/s]
```

tqdm is a Python library used to display progress bars for loops.
When you wrap your loop with tqdm(), it tracks the number of iterations and provides a visual indication of progress, including information such as the elapsed time, estimated time remaining, and the number of iterations completed

tqdm

- In particular, it shows you what fraction of your loop is done (though it can't do this if you use a generator), how long it's been running, and how long it expects to run.
- In this case (where we are just wrapping a call to range) you can just use ***tqdm.trange***.
- You can also set the description of the progress bar while it's running.
- To do that, you need to capture the tqdm iterator in a with statement:

tqdm

```python
from typing import List

def primes_up_to(n: int) -> List[int]:
    primes = [2]
    with tqdm.trange(3, n) as t:
        for i in t:
            # i is prime if no smaller prime divides it
            i_is_prime = not any(i % p == 0 for p in primes)
            if i_is_prime:
                primes.append(i)
                t.set_description(f"{len(primes)} primes")
    return primes
my_primes = primes_up_to(100_000)
```

✓ 10.2s

```
9592 primes: 100%|██████████| 99997/99997 [00:10<00:00, 9817.12it/s]
```

- This adds a description like the following, with a counter that updates as new primes are discovered:

Dimensionality Reduction

- **What Is Dimensionality Reduction?**
- Dimensionality reduction is the process of reducing the number of input variables (features) in your dataset while preserving as much of the important information as possible.
- This is especially useful in high-dimensional data (e.g., hundreds or thousands of features).
- It helps simplify models, reduce computation, and remove noise.

Dimensionality Reduction

- **Image Compression (e.g., Face Recognition)**
- **Scenario:**
- You have high-resolution images of people's faces. Each image (say, 100×100 pixels) is a vector of 10,000 pixel values.
- That’s 10,000 features per image!
- **Dimensionality Reduction:**
- Techniques like PCA or autoencoders reduce the image data to, say, 1000 key features.
- These compressed representations still preserve enough information for tasks like face recognition or classification.

Dimensionality Reduction

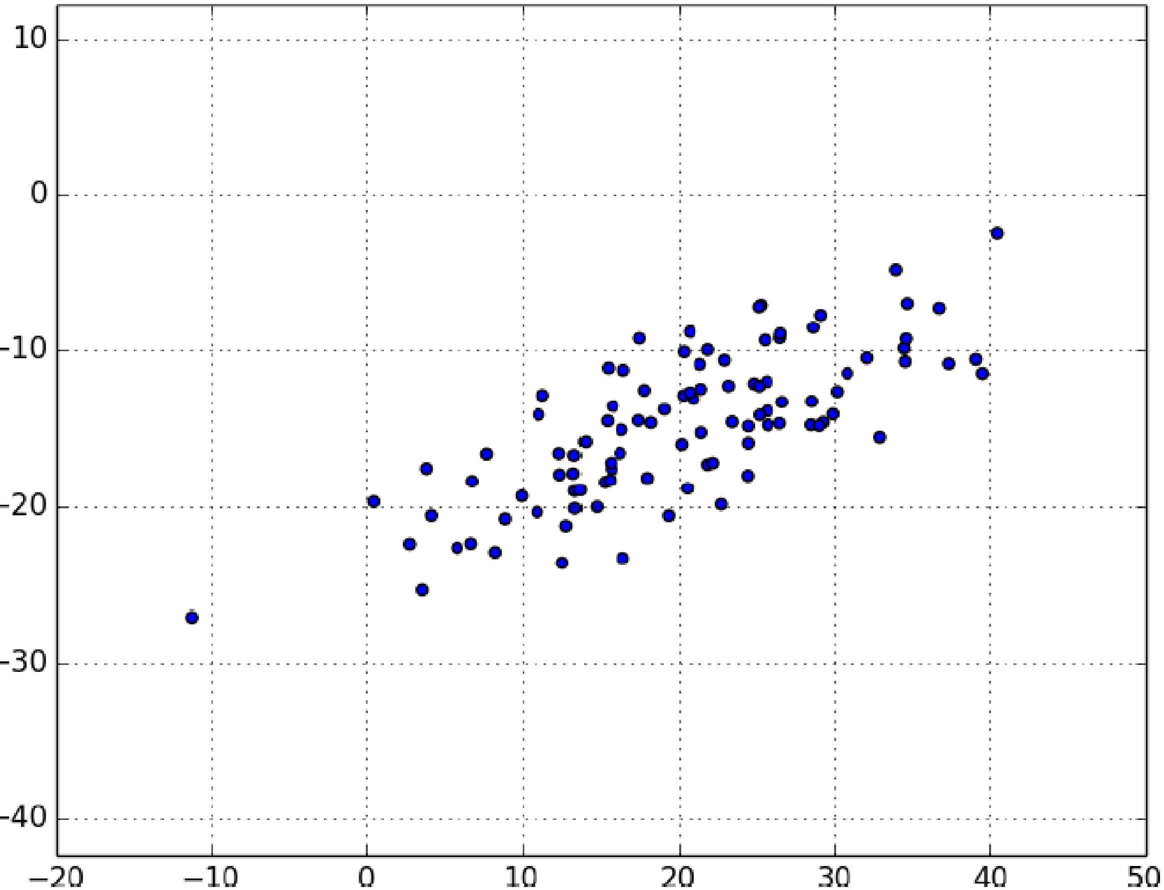


Figure 10-6. Data with the "wrong" axes

Data with the 'wrong' axes"

- Axis limits are poorly chosen
- Plot origin is misleading
- Scale mismatch between x and y axes
- Visual density is reduced

How to fix it:

- Adjust the **x-axis** and **y-axis limits** to closely match the range of the data.
- Use **equal or proportional scales** on both axes if comparing relationships.

Dimensionality Reduction

What is PCA?

Principal Component Analysis (PCA) is a **dimensionality reduction** technique used in data analysis and machine learning.

In simple terms, PCA transforms a dataset with possibly many variables into a smaller set of **new variables** (called **principal components**) that:

- capture **most of the variability (information)** in the data,
- are **uncorrelated** (orthogonal),
- and are ranked by how much variance they explain.

Imagine you have a cloud of points in 2D or 3D space. PCA:

- Finds the **direction** (line) along which the data varies the most → this is the **first principal component**.
- Finds the **next direction** that is perpendicular to the first and captures the next most variance → this is the **second principal component**.

And so on, for higher dimensions.

These principal components let you **compress** the data with minimal loss of important patterns.

Dimensionality Reduction

```
import numpy as np
import matplotlib.pyplot as plt
from sklearn.decomposition import PCA

# Step 1: Create a small 2D dataset (e.g., 10 points)
X = np.array([
    [2.5, 2.4],
    [0.5, 0.7],
    [2.2, 2.9],
    [1.9, 2.2],
    [3.1, 3.0],
    [2.3, 2.7],
    [2.0, 1.6],
    [1.0, 1.1],
    [1.5, 1.6],
    [1.1, 0.9]
])

# Step 2: Plot the original data
plt.figure(figsize=(12, 5))
plt.subplot(1, 2, 1)
plt.scatter(X[:, 0], X[:, 1], color='blue')
plt.title("Original Data")
plt.xlabel("X1")
plt.ylabel("X2")
plt.grid(True)
```

Dimensionality Reduction

```
# Step 3: Apply PCA to reduce to 1D (just for illustration)
pca = PCA(n_components=1)
X_pca = pca.fit_transform(X)

# Step 4: Project back to 2D for visualization
X_projected = pca.inverse_transform(X_pca)

# Step 5: Plot the PCA-projected data
plt.subplot(1, 2, 2)
plt.scatter(X[:, 0], X[:, 1], alpha=0.2, label='Original',
color='blue')
plt.scatter(X_projected[:, 0], X_projected[:, 1], color='red',
label='PCA Projection')
plt.title("PCA Projection (1D)")
plt.xlabel("X1")
plt.ylabel("X2")
plt.legend()
plt.grid(True)

plt.tight_layout()
plt.show()
```

Dimensionality Reduction