CSCI1951A - Data Science

Lucas Brito
lucas_brito@brown.edu

September 5, 2021

Contents

Preface

These notes are based on the course CSCI1951A - Data Science as taught in the summer of 2021. The course is described as an integrated introduction to techniques commonly used in data science and related fields; and it is loosely structured as follows:

1. Data processing
2. Statistics
3. Machine Learning
4. Bells and whistles

The majority of the notes are based on material covered in lectures and demonstrations, but some content is derived from labs. If you see any issues with the notes don't hesitate to contact me at `lucas_brito@brown.edu`.

Chapter 1

Databases and SQL

1.1 Requirement Engineering

Before you start working with code, get an informal picture of the data domain, think of how to organize it:

- What objects in the data do you care about?
- What properties/attributes of those objects are you measuring?
- What are the relationships between these properties/objects?
- What assumptions are we making?
- What is the workload on the database—read only/write only?

EXAMPLE: Take as an example an election. The **objects** used are: people, tweets, candidates.

The **domains of attributes** of these objects: tweets have timestamps, authors, text, attachments, hashtags.

Identifiers/references/relationships: people have unique IDs, people author tweets, people retweet, people follow people, tweets mention people, so on.

Cardinalities: every tweet has exactly one author, each author can have multiple tweets, each tweet can mention multiple candidates.

Distributions: some people tweet a lot, others just follow; some candidates are mentioned more than others. Account for follower/followee asymmetry.

Workload: scrape and populate once, read often.

Priorities and service level agreements: "right to be forgotten" rules (if someone deletes a tweet you have to delete it too).

1.2 Conceptual Modelling

Entity-relationship model: One has an entity, an attribute, and a relationship between that entity and attribute. For example, with a tweet, one may have multiple mentions in it. You might have composite attributes: time, for example, has minutes, seconds, and hours. Do we make dedicated columns for seconds, minutes, hours? The answer depends on the application, not a clear answer.

Such models are often diagrammatically represented as depicted in figure 1.1. Here ellipses represent attributes, rectangles represent entities, and diamonds represent relationships. The double ellipse represents a multivalued attribute.

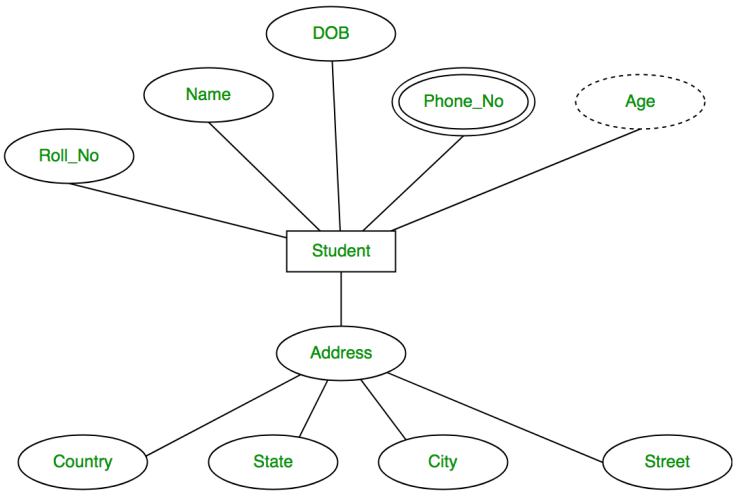

Figure 1.1: From GeeksForGeeks.

Key attribute: This is a designated attribute that uniquely identifies the entry. Time, for example, would be a poor key attribute because two people could tweet at the same time. One could combine two attributes to make a key attribute— for example the time and the author together. You could also just create a new key—when you enter something into your database, give it a unique hash.

Relationships: Is a mention a relationship between a tweet and a person, or a relationship between two people? Do we want to measure how many times a candidate has been mentioned, as an attribute of the candidate, versus as an attribute of a tweet? You could also make a mention a first-class relationship, which is its own entity in the database. Then the relationship itself can also have attributes—a direct or indirect mention, the sentiment of the tweet, so on. Or we could have an `author_of` relationship which relates a tweet and a person.

Design decisions: You can't talk about things that don't exist! It is important to anticipate the entire spectrum of analysis that you will make in order to design your database such that you can actually complete this task. This will depend on the application and use case, so there is no right answer.

1.3 Relational Algebra (For SQL)

SQL is based on something called a **relational model**. In the following case, this is a relation of tweets; the set of column names can be thought of as the **relation schema** which defines what it means to be, say, a tweet, the table itself is the **relation state**, the state of the tweets we are representing, and the table itself is the **relation instance**.

ID	Timestamp	Author	Text	Mentions
104	1/1/19 12:43	Bob	Hey	NULL
412	1/4/21 04:12	Maria	lol	[Bob]
818	1/2/19 1:04	Yu	:-D	NULL

1.4 SQL

SQL (Structured Query Language, pronounced "sequel") can be thought of as two types of language at once:

- Data definition language (DDL): define data types and relation schemas.
- Data manipulation and query language (DML): populating/updating databases and querying data bases.

1.4.1 Creating and Manipulating Data Tables

SQL supports the following **data types**:

- **Numeric**: `INT, FLOAT, REAL, DOUBLE`
- **Character strings**: `CHAR(n), VARCHAR(n)` (variable length; takes up less space for smaller string. `CHAR` requires precisely n characters whereas `VARCHAR` can have *up to* n characters.), `CLOB(size)` (for large objects). `CHAR` is faster for it uses static memory allocation: no length checks in operations. `VARCHAR` uses less space on average.
- **Bit strings**: `BIT(n)`, `BIT VARYING(n)`, `BLOB` (same as above).
- **Boolean**.
- **Dates**: `DATE`, `TIME`, `TIMESTAMP`, `TIME WITH TIME ZONE`

Creating tables: We want a tweet with an ID, a time, and a text. We write

```
create table TWEET (
    ID INT,
    Time TIMESTAMP,
    Text VARCHAR(140)
);
```

Now we want to create a table for the author relation. This is a relationship between tweets and people:

```
create table AUTHOR (
    Tweet INT;
    Person VARCHAR(100),
);
```

To **populate the table**, we write

```
insert into TWEET values(
    149414,
    2019-91-01 12:34:56,
    "hey"
);
```

If you didn't specify a value the value will be `NULL`. You can assign default values if you'd like:

```
create table TWEET (
    ID INT DEFAULT 0,
    ...
);
```

Or, to insert into specific attributes:

```
insert into TWEET(timestamp, Text) values(
    2019-01-01 12:34:41,
    "lol"
);
```

Where the empty attributes will be filled with `NULL`. You can make SQL enforce non-null attributes with `ID INT NOT NULL`.

You can also create tables from existing tables:

```
CREATE TABLE new_table_name AS
    SELECT column1, column2, ...
    FROM existing_table_name
    WHERE ...;
```

1.4.2 Primary Keys

The primary key is what we use to identify each row in the database uniquely. We specify them by

```
create table TWEET (
    ID INT PRIMARY KEY,
    Time TIMESTAMP,
    Text VARCHAR(140)
);

--OR

create table TWEET (
    ID INT,
    Time TIMESTAMP,
    Text VARCHAR(140)
    PRIMARY KEY (ID)
);
```

This automatically enforces a non-null, unique value for that attribute. We may also auto assign primary keys (see this article). Alternatively, we can use existing values (this is not as recommended):

```
PRIMARY KEY (Tweet, Person)
```

1.5 Foreign Keys

A foreign key is a key that acts as a pointer to another table.

```
create table TWEET(
    ID INT,
    Time TIMESTAMP,
    Text VARCHAR(140)
);
create table PERSON(
    Handle VARCHAR(100),
    Name VARCHAR(1000)
);
create table AUTHOR(
    Tweet INT,
    Person VARCHAR(100),
    PRIMARY KEY (Tweet, Person),
    FOREIGN KEY (Tweet) REFERENCES TWEET(ID),
    FOREIGN KEY (Person) REFERENCES PERSON(Handle)
);
```

Foreign keys do not have to be primary keys, but they have to be unique and they cannot be null (nulls are distinct, so `NULL == NULL` is false). Generally you will want to make foreign keys primary keys.

Referential integrity: we have to make sure that foreign keys actually refer to rows that exist. One could handle this with

```
FOREIGN KEY (Tweet) REFERENCES TWEET(ID) ON DELETE CASCADE
```

Deletes all the references to that row.

```
FOREIGN KEY (Tweet) REFERENCES TWEET(ID) ON DELETE SET NULL
```

Sets to null when the row is deleted.

```
FOREIGN KEY (Tweet) REFERENCES TWEET(ID) ON DELETE RESTRICT
```

Prevents deletion of rows.

1.6 Queries

The query syntax is as follows:

```
SELECT <attribute list or * for all attributes>
FROM <table list>
[ WHERE <condition> ]
[ GROUP BY <attribute list> ]
[ HAVING <condition> ]
[ ORDER BY <attribute list> ];
```

SQL Fiddle is a good way to test out this syntax.

EXAMPLE: : Return to our Tweet database. If we want to show every attribute (note the asterisk) for the tweets where the text is "hey":

```
SELECT *
FROM TWEET
WHERE Text = "hey"
```

If we want the ID and the Person from the Tweet and Author databases, where we line up the tweet and the author:

```
SELECT ID, Person
FROM Tweet,
    Author
WHERE ID = Tweet
```

This `WHERE ID = Tweet` is called the **join condition**. If we omit the join condition, we will get the Cartesian product of the two tables. For every entry of one table, we combine it with every row of the second table.

Aliasing: To make things more readable and avoid ambiguity, one can use aliasing:

```
SELECT ID, Text
FROM Tweet AS t,
    Author AS a
WHERE t.ID = a.ID
```

To include further conditions, one can use `AND` and `OR`.

```
SELECT ID, Text
FROM Tweet AS t,
```

```
    Author AS a
WHERE t.ID = a.Tweet
    AND a.Person = "d"
```

1.7 Joins

The above scenarios where we queried two databases are called **joins**. Here we use the explicit `join` keyword:

```
SELECT ID, Text
FROM (TWEET JOIN AUTHOR ON ID = Tweet)
WHERE Person = "d"
```

Recall if we omit the join condition and just list two tables in the `FROM` clause, we will get the Cartesian product of the two tables. For every entry of one table, we combine it with every row of the second table.

Sometimes not every key in one table will have an entry in the other table, and you will get a table with some values missing (these values will be `NULL`s). SQL handles this in three different ways:

1. **Inner join** (default): we will only return the rows that do not have any missing values.
2. **Left outer join**: keep everything that occurred in the left table, throw away things that occur on the right table but not on the left.

   ```
   ...
   FROM (TWEET LEFT OUTER JOIN AUTHOR ON ID = Tweet)
   ```

3. **Right outer join**: the above but with the right.
4. **Full outer join**: returns everything.
5. **Natural join**: assumes condition is *all pairs* of attributes with the same name, then does an inner join. If two columns have the same name, it tries to use those for a join. With no matches, it forms a cross product.

   If we don't have matching columns, we can alias each column, and natural join will join on those columns:

   ```
   SELECT ID, Text
   FROM (TWEET AS t(tweetid, text) JOIN AUTHOR AS a(person, tweetid))
   ```

   Although if we are aliasing the columns, we may rename two columns to the same name when they don't hold the same type of data, we will get an empty join.

1.8 Useful Keywords

- `ORDER BY`: Must be at the very end of the query. Specifies the attribute using which SQL should order things.
- `GROUP BY`: produces one row for each value of the attribute

  ```
  SELECT Text,
  COUNT(*), AVG(Likes)
  FROM Tweet
  GROUP BY Text
  ```

  Where we have attributes `COUNT`, `AVG`, `SUM`, `MIN`, `MAX` which only work when there is a `GROUP BY` statement. These will produce that specified value (e.g., the average) of the given attribute or attributes of all the rows that were grouped together.

- `HAVING`: only used with attributes specific to `GROUP BY`s, analogue of the `WHERE` keyword:

```
SELECT Text,
COUNT(*), AVG(Likes)
FROM Tweet
GROUP BY Text
HAVING COUNT(*) > 1
```

  You should use this for things like `AVG` but `WHERE` on other attributes. This keyword exists because the `GROUP BY` columns exist in memory and not on disk.

- `LIKE`: RegEx-like searches; for example,

```
SELECT Text
FROM Tweet
WHERE Text LIKE '%1951A%'
```

  (If you're finding yourself constructing really complex text queries, it's best to switch to something other than SQL: perhaps preprocess your data with Python.)

- `IN`: Set-like operation. This runs on a subquery; a query inside another. Lets you check if an attribute is in the return of another query:

```
SELECT Name
FROM STUDENT
WHERE ID IN
    (SELECT Student
    FROM GRADES
    WHERE Course = 1941A)
```

  To return the names of students in the table grade with course 1951A.

- `ALL/ANY`: All elements in a set.

```
SELECT Name
FROM STUDENT
WHERE Grade >= ALL
    (SELECT Grade
    FROM STUDENT
    WHERE Course = 1941A)
```

  This will return true if a particular student's grade is greater than or equal to all of the grades returned by that subquery.

- `DISTINCT`: Removes duplicates by default.

```
SELECT DISTINCT Name
FROM STUDENT
WHERE Grade >= ALL
    (SELECT Student
    FROM GRADES
    WHERE Course = 1941A)
```

- `EXISTS`: Checks if the output of a query exists at all:

```
SELECT NAME
FROM STUDENT s
WHERE NOT EXISTS
    (...)
```

1.9 Dealing with NULLs

NULLs are effectively black holes. They annihilate all data, and we can't make any assumptions about the values:

```
NULL + 1 = NULL
NULL * 0 = NULL
NULL = TRUE => UNKNOWN
NULL = NULL => UNKNOWN
```

All logical operations with NULL ends in NULL except for AND with a false and OR with a true.

Often SQL ignores NULLs. It treats UNKNOWN and FALSE equally, so usually queries will not return any missing values.

For GROUP BY: If NULL exists, there will be a group for NULL.

For predicates with null: use IS:

```
SELECT Text, ID
FROM TWEET
WHERE TEXT IS NULL
```

Aggregators (SUM, AVG, etc.): simply ignore NULL unless it is the *only* value.

1.10 Execution Order

There are multiple ways we could execute each of the keywords (SELECT, WHERE, FROM) while still obtaining the same results. The best execution order is

```
SELECT(WHERE(FROM))
```

If we do, say WHERE(SELECT(FROM)), then the WHERE clause might need an attribute that we threw away with the SELECT clause. The canonical execution order is exactly the above. Notice that this is not exactly how one would write this in natural language.

1.11 Complexity and Efficiency

Suppose a query where each Ri has m tuples

```
SELECT A1, ..., An
FROM R1, ..., Rk,
WHERE P
```

this query runs in $O(m^k)$. We have to take each tuple in R1 and multiply it by each tuple in R2, leading to $m \cdot m$ tuples, then once again by R3, yielding m^3 tuples... so on until m^k tuples, which we must filter (runs in linear time), yielding a complexity in $O(m^k)$.

EXAMPLE: We want to search the tweets that Barack Obama made on a certain date. It is inefficient to search all of the tweets (i.e., after having joined both tables naively), looking for ones that satisfy being Obama *and* the timestamp. So we first find tweets made in that date, then combine that with the author table, then query:

```
SELECT TWEET.Time
FROM TWEET, AUTHOR
WHERE AUTHOR.TWEET = TWEET.ID
```

```
    and TWEET.Date == '01/01/2019'
    and AUTHOR.Person = "Barack Obama"
```

1.12 Nested Queries

Nested queries are queries within queries, e.g.,

```
SELECT s.Name
FROM STUDENT AS s
WHERE NOT EXISTS(
    SELECT *
    FROM GRADES AS g
    WHERE s.ID = g.Student
)
```

Correlated query: Inner query relies on outer query. Inefficient. Try to write such that the inner query does not change based on the current value of the outer query (for then we have to execute the subquery once for every row of the outer query).

The above example is a correlated query. We can rewrite it as

```
SELECT s.Name
FROM STUDENT AS s
WHERE s.ID not in (
    SELECT Student
    FROM GRADES
)
```

1.13 NoSQL

Just key-value stores; dictionaries instead of tables, basically jsons. It is good for fast development, flexibility, and messy data; it is bad for data integrity, and safety guarantees. You don't have to actually design tables with certain attributes and whatnot.

1.14 SQLite

SQLite "implements a small, fast, self-contained, high-reliability, full-featured, SQL database engine." One can install SQLite using this this guide. SQLite allows you to query databases from terminal. Run `sqlite3 database.db` to launch an interactive database environment. Documentation can be found here, and command-line interface information can be found here.

From the command line you can simply run queries such as

```
>>> SELECT * FROM TABLE WHERE <COND>
```

as you would normally.

1.15 SQL in Python

You can import SQLite into Python using the package `sqlite3`. The API usage is as follows: one first creates a connection object:

```
conn = sqlite3.connect('PATH/TO/DATABASE.db')
```

Then one uses a `cursor` object to execute commands and queries. Commands are taken as strings as arguments to the `execute` method:

```
command = ```
          SELECT *
          FROM TABLE
          WHERE <COND>
          ```
c = conn.cursor()
c.execute(command)
```

If, beyond queries, one makes changes to the database (e.g., adds a table) then they must commit the result using `conn.commit()`.
```

Chapter 2

Web Crawling

2.1 Packages Used

Commonly used packages for web crawling are

- `urllib`: built-in Python package for working with URLs.
- `json`: built-in Python package for parsing JSON code.
- Requests: send HTTP requests.
- BeautifulSoup: parse raw HTML.
- Selenium: mimic human interaction.

2.2 Obtaining Data

Some packages that will be used:

```
import os
import urllib
import re
import json
from json import JSONDecodeError
import requests
from datetime import timedelta, date
from time import sleep
from bs4 import BeautifulSoup
from selenium import webdriver
```

We need to obtain the text of the webpage—the raw HTML.

```
req = urllib.request.Request(
    "https://..." )
```

This returns a request object. To obtain the response,

```
response = urllib.request.urlopen(req)
```

which produces a response object. To access the text:

```
html_doc = response.read()
```

Calling `print` on this will return unformatted HTML.

2.3 BeautifulSoup Demos

2.3.1 Demo 1

`BeautifulSoup` will actually parse this messy code.

```
html_dump = BeautifulSoup(html_doc, 'html.parser')
```

where the second argument specifies that we are parsing HTML.

When writing the crawler, we want to be able to crawl through the HTML and find the actual data we need. Let's tell BS to find everything inside table tags:

```
html_tables = html_dump.find_all('table')
```

Let's narrow down our search to the table with the class list:

```
class_list_html = None
for table in html_tables:
    # if the table has a caption and the text is "Summary Class List"
    if table.caption and table.caption.string == "Summary Class List":
        class_list_html = table
```

Find all the rows:

```
rows = class_list_html.find_all('tr')
header = rows[0]
```

Let's obtain all the column names from the header

```
col_names = [c.string for c in header.find_all('th')]
```

Return the index where the name column appears:

```
idx = col_names.index("Student Name")
```

Now go through every other row and pull out the column at `idx`.

```
for row in rows[1:]: # first row is header
    cols = row.find_all('td')
    student_name_col = cols[idx]
```

2.3.2 Demo 2

Let's go through a more complicated example: Pitchfork. We would have to, manually, copy the text from each album review; so we'll have to scrape the home page for each link, then scrape each album review for the contents. For links, they will be under an `a` tag as an `href` attribute. In the case of Pitchfork, we additionally have a `class` of `href`, `review_link`.

```
def main(MAX_PAGES=2)
    base_url = "https://pitchfork.com"

    # announce that you're using Firefox to prevent changes depending on
    # device.
```

```
    hdr = {'User-Agent' : 'Mozilla/5.0 (X11; Linux x84_64)'}

    all_data = []

    # The reviews are separated by pages.
    for i in range(1,MAX_PAGES):
        top_url = base_url+"reviews/albums/?page=%d"%i

        # Attempt a request, handle error.
        try:
            req = urllib.request.Request(top_url, headers=hdr)
            response = urllib.request.urlopen(req)
        except(UnicodeEncodeError, urllib.error.HTTPError, urllib.error.URLError):
            print("Error on request: %s"%top_url)
            continue

        # Initialize BeautifulSoup object to find all links in the page.
        txt = response.read()
        doc = BeautifulSoup(txt, 'html.parser')

        for item in doc.find_all('a'):
            # Get href attribute of this particular link
            url = item.get('href')

            # Check if it's a link to a review and append the scraped review
            in the case that it is.
            cls = item.get('class')
            if "review__link" in cls:
                print(base_url+url)
                all_data.append(scrape_review(base_url+url))
    return all_data
```

Now we actually define the function that we will run on each review page.

```
def scrape_review(url):
    # Again provide the client information for consistency.
    hdr = {'User-Agent': 'Mozilla/5.0 (X11; Linux x86_64)'}

    # Request this page.
    try:
        req = urllib.request.Request(url, headers=hdr)
        response = urllib.request.urlopen(req)
    except(UnicodeEncodeError, urllib.error.HTTPError, urllib.error.URLError):
        print("Error on request: %s"%url)
        return None
    txt = response.read()
    doc = BeautifulSoup(txt, 'html.parser')

    review_data = {} # Artist, Album, Review, Score

    # get the artist name
    for a in doc.find_all('a'):
        href = a.get("href")
        if href and href.startswith("/artists"):
            review_data["Artist"] = a.text
```

```
    # get the album name
    for h1 in doc.find_all('h1'):
        cls = h1.get("class")
        if h1 and 'single-album-tombstone__review-title' in cls:
            review_data["Album"] = h1.text

    # get score
    for span in doc.find_all('span'):
        cls = span.get("class")
        if cls and 'score' in cls:
            review_data["Rating"] = span.text

    # get all of the review text on the page
    all_text = []
    for p in doc.find_all('p'):
        all_text.append(p.text.strip())
    review_data["Review"] = ' '.join(all_text)
    return review_data
```

Now we write to an external JSON file

```
scraped_data = main()
print(json.dumps(scraped_data, indent=4, sort_keys=True))

outfile = open("scraped_pitchfork_data.json", "w")
outfile.write(json.dumps(scraped_data, indent=4, sort_keys=True))
outfile.close()
```

In general, you will always have to look at the raw HTML to determine where the important information is.

2.4 Basic API Calls

Some sites will provide you an API to search that specific site. In this case we will use the New York Times API, but APIs are a very common tool.

```
# need to create an account to obtain this key.
KEY = "..."

base_url = "https://api.nytimes.com/svc/"
```

The call to the API is an URL which you could put in a browser yourself and read the raw data. But we want it in Python, so

```
base_url = "https://api.nytimes.com/svc/search/v2/articlesearch.json"
query = "providence"
start = "20210101"
end = "20210525"

# Put the query parameters into a dictionary
params = { 'api-key': KEY,
           'q': query,
           'begin_date': start,
           'end_date': end}
```

```
# Use requests to convert the params dictionary into url format.
response = requests.get(base_url, params=params)
response.raise_for_status() # will throw an error if status is not OK

# Response will be a raw json output as a string, convert this to a Python
# json object.
data = response.json()

for article in data['response']['docs']:
    print('%s\n%s\n'%(article['web_url'], article['snippet']))
```

This script is, of course, specific to the NY Times API; however, most APIs are quite well documented, and the calls will not differ wildly.

2.5 Mimicking Human Interaction

`selenium` allows you to, if the website dislikes bots, mimic human interaction with the webpage. If some part of the site is dynamic—requiring checking a box or whatnot—`selenium` is a great tool for this.

Chapter 3

Data Cleaning

3.1 Dirty Data

Dirty data can manifest in many ways:

- Parsing input data: issues with separators like tabs, commas, spaces
- Naming conventions: NYC vs New York City
- Formatting issues, especially with dates
- Missing values and required fields
- Different representations: 2 vs two
- Fields that are too long and get truncated
- Primary key violations from merging
- Redundant records from merging

3.2 What's to be Done?

There isn't one thing you can do to clean your data, the process is dependent on the data set. You just need to look at your data.

- Maybe set sensible defaults if anything is missing.
- Remove outliers. Maybe set them all to a default, maybe set them all to zero. Or we could assume a maximum value and throw out anything above that. Or just entirely delete the entry.
- Maybe machine learn some of the things you have to do.
- When you issue a query, don't take the answer as gospel.

3.3 String Matching/String Similarity

Sometimes we ant to treat similar words as the same word; this constitutes data cleaning as it will facilitate analysis later down the line. Here are some measures of string similarity:

- **Edit Distance**: what is the minimum number of edits needed to transform string A into string B? An edit constitutes an insertion, substitution, or deletion. E.g., "cat" and "cut" have an edit distance of 1. "Cat" and "cater" have an edit distance of 2.
- **Jaccard similarity**:

$$J(A, B) = \frac{|A \cap B|}{|A \cup B|}$$

  where A and B are sets of words. One can also weigh the words of the the set when finding the Jaccard similarity in order to consider certain words above others.
- **Cosine similarity**: consider words that occur before and after a word. Treat the number of times these words occur as vectors. Find the angle between the two vectors. Good for semantically related words but not for structurally similar words.

Chapter 4

MapReduce

4.1 Introduction

MapReduce is a framework for parallelizing (making processing large datasets much larger). It is a functional-programming paradigm. You will use two types of functions:

- **Map**: `(in-key, in-value) -> list-of(intermediate-key, intermediate-value)`
- **Reduce**: `(intermediate-key, list-of(intermediate-value)) -> (out-key, out-value)`

EXAMPLE: Consider a word count program. We suppose four documents:

```
doc1 = "hello world"
doc2 = "oh hi there world"
doc3 = "why hell there , world"
doc4 = "world ! how the hell are ya ?"
```

We want to produce a list of each unique word and how many times it appears. First we a mapper; call this `mapper1`. The mapper will split the document string by spaces and return a tuple with that word and a count of one:

```
mapper1(doc1) -> [(hello, 1), (world, 1)]
mapper1(doc2) -> [(oh, 1), (hi, 1), (there, 1), (world, 1)]
mapper1(doc3) -> [(why, 1), (hell, 1), (there, 1) (world, 1)]
mapper1(doc4) -> [(world, 1), (how, 1), (the, 1), (hell, 1), (are, 1), (ya, 1)]
```

Now *reduce*. We group by key—in this case the key (the first element of each tuple) is the word itself, then combine all of the elements with the same key using some function we have defined:

```
reduce-by-key([(hello, 1), (world, 1), ...], add)
    -> (hello, add([1, 1])),
       (world, add([1, 1, 1, 1])),
       (oh, add([1])),
       (hi, add([1])),
       (there, add([1, 1]))
    -> (hello, 2),
       (world, 4)
       (oh, 1),
       (hi, 1),
       (there, 2)
```

So, there are four phases:

1. Input
2. Map phase
3. Shuffle phase (group by)
4. Reduce phase

The programmer needs only specify the mappers and reducers (sometimes several of each!), whereas the remainder is implemented by the framework.

4.2 Pseudocode Implementation

Code for a MapReduce program will look something like this

```
def mapper: (String, String) -> (String, Int) {
    ...
}

def reducer: (String, List(Int)) -> (String, Int) {
    ...
}

// Pipeline
Table<String, String> table = read(table_path) // presumably read parallelizes
Table<String, Int> output = table.mapper().reducer();
write(output)
```

The pipeline might also look something like this (a more complex chain)

```
Table<String, Int> output = table.mapper1().reducer1().mapper2().reducer1();
```

The bulk of the work is actually defining the mappers and reducers. Mappers will usually convert the input into a *stream* of key/value pairs (use `yield` in Python!):

```
def mapper: (String, String) -> (String, Int) {
    for w in input.value().split() {
        emit(w, 1) // a stream!
    }
}
```

We output as a stream so that we need not finish running the loop in the function for the remainder of the machines to start working: as soon as this key-value pair is produced, the reducer that needs it can start processing it, and we don't lose any efficiency to waiting on the mapper.

The reducer will look something like

```
def reducer: (String, Stream(Int)) -> (String, Int) {
    sum = 0;
    for c in input.value() {
        sum += c;
    }
    emit(input.key(), sum)
}
```

Note that you don't always have to use keys, nor do the keys actually have to contain information you will use later down the pipeline.

4.3 Behind-the-scenes Details

The shuffle phase does *not* sort, it simply groups by key. So there is no guarantee that the result be sorted in any way.

The reduce phase does guarantee that elements with the same key will appear in the same machine so that we actually account for all elements with that particular key.

There are constraints on the mapper and reducer:

- The mapper must be equivalent to applying a deterministic "pure" function (a mathematical function; does not depend on external context, does not affect anything external like global variables) to each input *independently*.
- The reducer must be equivalent to applying a deterministic pure function to *the sequence of values for each key*.

Benefits:

- When a program contains only pure functions, expressions can be evaluated in any order, lazily, and in parallel.
- Consistent results, but computation partitioned.
- Referential transparency: a call expression can be replaced by its value without changing the program. Allows for caching.
- Re-computation and caching of results as needed—can naively replace the computation with a cached result.

Coordination:

- There is one master scheduler that oversees the pipeline. It is responsible for making sure machines finish their assigned jobs. The machines cannot talk to each other, they must communicate to the master.
- The master assigns tasks to the available mappers and reducers.
- It also keeps track of metadata:
  - Task status: idle, in-progress, completed (if idle will delegate task to another machine)
  - Idle tasks get scheduled as workers become available
  - When a map task completes, it sends the master the location and sizes of its intermediate files, one for each reducer
  - Master incrementally pushes this info to reducers
- Input, final output are stored on distributed file system.
- Scheduler tries to schedule map tasks "close" to physical storage location of input data. The less far you have to send the data, the faster.
- Intermediate results are stored on local file system of map and reduce workers.
- Output is often input to another map-reduce task.

4.4 Other MapReduce Functions

Beyond mappers and reducers, there are also other functions provided by MapReduce frameworks (in particular PySpark):

- **Sort**
- **Unique**
- **Sample**: take a random sample of inputs
- **First**: Take the first n inputs
- **Join**: Join two sets of inputs!

4.4.1 Joins

Suppose we have two tables with columns `facts: [subject, predicate, object]` and columns `categories: [category, entity]`, and we want to perform the following join (in SQL):

```
SELECT * FROM facts, categories
WHERE subject == entity
GROUP BY subject
```

A MapReduce join will produce the following, assuming the proper columns are set as keys:

```
Key: String // the subject in question
Value: (stream-of((String, String, String)), stream-of((String, String)))
       // first stream is all of the rows in the facts table wherein that key
       // appears, and the second stream is all of the rows in the categories
       // table wherein that key appears.
```

So, to prepare the data for a join, you will need to re-key the tables using mappers, making sure that the columns you want to join on are indeed the keys. E.g., for the `categories` table, something like:

```
def mapper(input) {
    emit (input[1], input)
}
```

And then we can emulate the SQL `GROUP BY` using a reducer!

4.5 MapReduce in Python

PySpark is an implementation of Spark, a MapReduce-like parallelization framework.

Usage is as follows: first import `SparkContext`:

```
from pyspark import SparkContext
```

Then instantiate a `SparkContext` object:

```
sc = SparkContext()
```

And open, say, a text file using the object:

```
txt = sc.textFile("path/to/file").collect()
```

Which we can then use to view the file itself.

Now say we have an array of data, we can right away run

```
data = sc.parallelize(data, 128)
```

where we've parallelized the data and requested 128 slices.

Then we can call the pipeline:

```
data = data.map(mapper1).reduceByKey(add).flatMap(mapper2).filter(filterer)
```

Where `flatMap` simply flattens the result of the mapper, and `filter`, well, filters. Finally we collect the result, and we are free to print it or manipulate it however we wish:

```
result = data.collect()
```

And we stop the current `SparkContext`:

```
sc.stop()
```

Chapter 5

Hypothesis Testing

5.1 Data Analysis

One will likely formulate a hypothesis once they've collected their data. There are a handful of approaches to formulating a hypothesis upon analyzing data:

- **Heuristic analysis**:
  - Make observations on the available data
  - No (or very weak) guarantee on generalizability of the results
  - Still can be useful
  - Many ML, database analysis, and big data analytics techniques are heuristic in nature
  - Simply exploring the data and making intuitive guesses
- **Probabilistic method**:
  - Assume the existence of an underlying (ground truth) randomized phenomenon that generates the observed data according to some unknown probability distribution.
  - Assume observed data to be obtained by sampling this distribution (often independent sampling).
  - Analyze the data to infer conclusions that are valid for the entire underlying model (generalize to the unknown distribution)
  - Want guarantees on the accuracy of our insights. E.g., want to be able to claim that our estimate of some parameter is close to the true value (within, say ϵ) with *high probability*.
  - Statistical machine learning, probabilistic data analysis, etc.
- **Hypothesis testing**:
  - Formulate an a priori "null" hypothesis on the "world" or some phenomena (think of this as a widely held belief).
  - Come up with a new "alternative" hypothesis which contradicts the null. This is the hypothesis that will be tested.
  - Obtain data to test our hypothesis. We will argue whether it is it possible to reject the null hypothesis as extremely unlikely based on the observed data. Rather consider how likely it is that the data would be generated by a phenomenon that follows the null hypothesis.
  - Want guarantees on the accuracy of our insights. Asymptotic guarantees on the accuracy of our rejections.

- Very different from probabilistic approach, wherein we obtain finite guarantees.
- Many sub-branches: "frequentist" statistical tests, Bayesian approach.

The differences between statistics and probabilistic analysis is subtle: both make data about how the data was generated; both seek guarantees on estimates. Although there is some historical difference: earlier there was less available data, and the focus was on hypothesis testing. In the era of Big Data, other methods are viable.

Depending on the available data one method may be more desirable: how much data is available? How much priory information do I have about the model? What kind of guarantees on the results do I want?

Statistically testing yields asymptotic guarantees, while probabilistic analysis yields stronger finite sample guarantees.

5.2 What is a Hypothesis?

A hypothesis is a statement about the properties of some observed phenomenon. The most important aspect of a hypothesis is it must be falsifiable. But a hypothesis should also be a disputed, interesting claim. It should also be a specific claim: avoid vague terminology.

5.3 Probability Theory

The term **probability space** is the formal notation used to treat the domains or problems that we want to study probabilistically. It's defined as the tuple

$$\langle \Omega, F, P \rangle$$

- Ω: set of all possible outcomes. E.g., for rolling a die, $\Omega = \{1, 2, 3, 4, 5, 6\}$.
- F: the family of sets representing the allowable events, where each item set in F is a subset of the sample space Ω. E.g., for rolling a die, $\{\{\}, \{1\}, \ldots, \{1, 2, 3, 4, 5, 6\}\}$.
- P: probability function which assigns a real number to each event in F. It is a function $P : F \to \mathbb{R}$:

$$0 \leq P(E) \leq 1 \forall E \in F$$
$$P(\Omega) = 1$$
$$P\left(\bigcup_i E_i\right) = \sum_i P(E_i)$$

- Random variable X assigns a number to each outcome. $X : \Omega \to \mathbb{R}$. Use $X = a$ to mean the event $\omega | X(\omega) = a$
- Probability mass function gives probability that X takes the value a: $p(a) = Pr(X = a)$
- Cumulative distribution function gives probability that X takes any value *up to* a: $F(a) = Pr(X \leq a)$.

Two events are **independent** if and only if

$$P((X = x) \cap (Y = y)) = P(X = x)P(Y = y)$$

This definition extends to multiple random values as well.

Two random variables X and Y are identically distributed if and only if

$$P(X = x) = P(Y = x) \forall x \text{ in the range of } X, \text{ and } Y$$

Sometimes we say two or more variables are **IID** or **iid** as a shorthand to say that the variables are all identically distributed and pairwise independent.

Expected value: the expected value that a random variable will take on, i.e. the value it will converge to on average after many samples.

$$E(X) = \sum_i x_i P(x_i) \iff E(X) = \int_i x P(x)\,\mathrm{d}x$$

Law of large numbers: if you perform the same experiment a large number of times the average will converge to the expected value.

Central limit theorem: the distribution likewise approaches the normal distribution for an experiment that is repeated several times.

Variance: the variance is defined by

$$\mathrm{Var}(X) = E((X - E(X))^2)$$

And the **standard deviation** is the square root of the variance, characterizing how much deviation from the expectation we are likely to observe when evaluating a random variable.

Binomial distribution: this is the probability distribution of the number of heads in n coin flips (it generalizes to many other phenomena!). It assumes all tosses are iid, and disregards the order in which the heads and tails appear:

$$P_x = \binom{n}{x} p^k (1-p)^{n-k}$$

where x is the number of heads and n is the number of flips.

Normal/Gaussian Distribution: this is the continuous distirbution for a real-valued random variable, parameterized by μ, the mean, and σ, the standard deviation. When $\sigma = 1$, $\mu = 0$, this is a standard normal distribution. The pmf is given by

$$\varphi(x) = \frac{1}{\sqrt{2\pi}} e^{-x^2/2}$$

and the cmf is complex to compute by hand, and is usually evaluated numerically using tables or libraries.

5.4 Hypothesis Testing at Large

We will formulate, as described above, a null hypothesis H_0 and an alternative hypothesis H_a. Some example null hypotheses:

- The current standard of care is as good as the new one
- INflation was the expected 2% over the last quarter
- There is no difference in the types of movies streamed during vs before COVID.

The procedure will be as follows: we assume the null hypothesis is true. If there is enough evidence to suggest that H_0 is highly unlikely, we say we "reject the null hypothesis." If there is not enough evidence, we "fail to reject it." Note that we don't "accept" or "prove" either H_0 or H_a.

In detail, we will

- Formulate hypothesis

- Set up an experiment: select appropriate statistical test (more on this later) and a test statistic, acquire data, and come up with a priori theoretical distribution for the test statistic (often alraedy given by the definition of the statistical test).
- Select a threshold α for what you will consider surprising or significant. If the null hypothesis turns out to have a probability of α or lower of occurring we will reject the null.
- Compute the likelihood of observing the test statistic under the null hypothesis (p-values).
- Then compare the p-value with α and determine whether the null is rejected.

5.5 Statistical Tests (Which to Use?)

- *Is my sample value different than a hypothetical value?* One sample t-test (normally distributed), or Wilcoxon test (not normally distributed)
- *Is this sample's value different than some other sample's value?* If samples are dependent: paired wo sample t-test (normally distributed) or Wilcoxon test (not normally distributed) If samples are independent: unpaired two-sample t-test (normally distributed) or Mann-Whitney test (not normally distributed).
- *Is this sample's distribution different than that sample's distribution?* Chi-squared test (discrete) or Kolmogorov-Smirnoff test (continuous).
- *Does variable a affect variable b?* Correlations (not controlling for confounds) or linear regression (controlling for confounds).

5.6 One-Sample Z-Test

One-sample z-tests are actually not commonly used, but they lay the foundation for t-test, which are used much more commonly.

Compute the mean of the sample and compare to expected mean, the hypothesized mean. We can invoke the central limit theorem and use the normal distribution. Our test statistic is

$$z = \frac{\bar{x} = \mu_0}{\sqrt{\sigma^2/n}}$$

which gives us how many standard deviations away from the population mean $\bar{x}$, the same mean, is. We use this z-score to find the value of the cmf of this normal distribution at that point.

I.e., compute probability that, given the null-hypothesis is true, we get a the value we obtained or lower (or higher, depending on alternative hypothesis). We choose the level of significance α, typically 0.10 or 0.05.

There is unidirectional (one-tailed) hypothesis, which means with $\alpha = 0.05$, the mean will be higher (or lower) by chance 5% of the time. There is bidirectional (two-tailed) hypothesis, with mean being lower 2.5% of the time and higher 2.5% of the time.

Then, with unidirectional test, you use the α to compute the number of standard deviations away from the mean (the z-score) that the sample mean must be for there to be a 5% chance or less of obtaining that value. With that, if the number of standard deviations away from the mean the sample mean lies is greater than (or less than, or greater than or less than) that z-score, we reject the null hypothesis.

Naturally, for a bidirectional test, we compute the z-score for 2.5% probability on the lower tail and 2.5% probability for the higher tail such that the total probability of such a deviation is $\alpha = 0.05$. When in doubt use the two-tailed test unless you have good reason.

However, notice that our z-test formula has the population variance, which we do not know unless we know the population distribution!

5.7 T-Tests

5.7.1 One-Sample T-Test

When the population variance is not known (as it most commonly the case), we use a one-sample t-test. We instead use the sample variance and standard deviation to estimate the population distribution. This t-distribution will be *slightly different* than a normal distribution—the tails of the distribution have more probability mass, so in general, we are *less* surprised by values farther away from the mean. As the sample gets larger and larger, the t-distribution approaches a normal distribution.

The t-score is

$$\boxed{t = \frac{\bar{x} - \mu_0}{s/\sqrt{n}}}$$

5.7.2 Two-Sample T-Test

Two-sample t-tests are used to compare two values—say, means or proportions.

For a two-sample t-test: we use a similar formula, except now we are computing the distribution of difference of means

$$\boxed{z = \frac{\bar{x} = \mu_0}{s/\sqrt{n}} \text{ with } \mu_0 = 0,\ \bar{x} = \bar{x}_1 - \bar{x}_2}$$

You will have the choice between a paired two-sample t-test and an unpaired test:

- **Paired**: same sample in two different conditions.
- **Unpaired**: two independent samples. Individuals are not necessarily the same between two samples.

5.8 Chi-Squared Test

Chi-squared tests are used to determine the difference between (discrete) distributions. Tip: if you're using bar charts, that probably means you'll want to use a chi-squared test!

One models the difference between two distributions. Each category (discrete variable of the distribution) has an observed and expected value. We wil consider their difference

$$X_i - E(X_i)$$

We want to capture the for each category in one number, so we sum the squares of differences then normalize

$$\sum_i \frac{(X_i - E(X_i))^2}{E(X_i)}$$

It turns out that this above test-statistic follows what is called a chi-squared distribution, which is what will be used to determine significance. The cdf of this distribution is

$$\frac{1}{\Gamma(k/2)} \cdot \gamma\left(\frac{k}{2}, \frac{x}{2}\right)$$

The value against which we will compare will depend on the chosen α value as well as the "degrees of freedom" of the problem:

$$df = (r-1)(c-1)$$

where r is the number of rows (categories) and c is the number of columns (different distributions).

The expected values will depend on the null hypothesis as well as the size of the sample. For example, if we hypothesize that students are equally likely to be absent on every day of the week, and one collects a sample of size 120, then each expected value will be 20.

For a two-sample chi-squared test: just treat one of the two samples to be the "expected" value and use the same procedure.

5.9 Correlations

Does variable XYZ affect variable ABC? First thing to do—plot your data. Dependent variable on y axis and independent variable on x axis.

Correlations answer the question: what proportion of variation in y can be explained by variation in x? They are not a measure of causality. Only captures linear relationship. Three main variants (usually similar results):

- Pearson: `scipy.stats.pearsonr`. When x and y are continuous, and the *distance* between points matters. It is computed using the normalized covariance,

  $$\rho(X, Y) = \frac{COV(X, Y)}{\sigma_x \sigma_y}$$

  where

  $$COV(X, Y) = E[(X - E[X])(Y - E[Y])]$$

- Spearman: `scipy.stats.spearmanr`. Compute correlation between list of *ranks* rather than values themselves: e.g., for $[3.4, 5.7, 1.9] \to [1, 2, 0]$. We here consider the rank the order in ascending direction, (1.9 then has the lowest rank of 0), and we obtain the Pearson correlations of the ranks in this manner (effectively discarding the distance between points).

  Better for discrete "ordinal" variables (star ratings, survey responses, etc).

  $$\rho(X, Y) = \frac{COV(\text{ranks}_X, \text{ranks}_Y)}{\sigma_{\text{ranks}_X} \sigma_{\text{ranks}_Y}}$$

- Kendall-Tau: `scipy.stats.kendalltau`. Also uses *ranks*, but doesn't use Pearson under the hood. Can use similarly to Spearman, makes fewer assumptions. Instead use concordant and discordant pairs:
  - **Concordant pairs**: (x_i, y_i) and (x_j, y_j) s.t. $i < j$ and $\text{sign}(x_i - x_j) = \text{sign}(y_i, y_j)$.
  - **Discordant**: anything that does not satisfy the above.

  Then

  $$\tau = \frac{\#\text{concordant} - \#\text{discordant}}{\binom{n}{2}}$$

Overall, correlations are

- Great for exploratory analysis
- Easy first thing to check (if you think X causes Y, there should be at least a correlation between X ad Y; not always, but a good sanity check).
- But! Hard to draw strong conclusions—not a measure of causality.

Why don't we draw stronger conclusions from correlations? The **lurking variable problem:** E.g., does getting more sleep cause better grades? Many alternatives: sleep causes better grades, or less stress causes more sleep and better grades, living in a quiet dorm, where you can study better and get better grades, and also sleep more. Many different manners in which there could be correlation.

5.10 Linear Regression

Linear regressions are used to model a phenomenon in more detail than correlations, accounting for multiple variables. We will start by exploring the single-variable case for illustrative purposes.

The model is the familiar

$$y = mx + b + e$$

With

$$\begin{aligned} y &\equiv \text{dependent variable} \\ x &\equiv \text{independent variable} \\ m &\equiv \text{slope (coefficient)} \\ b &\equiv \text{intercept} \\ e &\equiv \text{random (hopefully) error} \end{aligned}$$

With a finite amount of data, the model will actually look something like this, where x_i and y_i are the given observations:

$$\begin{bmatrix} y_1 \\ y_1 \\ \vdots \\ y_n \end{bmatrix} = m \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix} + b + \begin{bmatrix} e_1 \\ e_2 \\ \vdots \\ e_n \end{bmatrix}$$

We will want to use x_i and y_i to find m and b (which we assume to be shared across the population), and we will do this by minimizing the error vector.

The **residuals** will be the difference between the predicted y value and the actual y_i value for a certain data point. (These are actually what go into the error vector $\mathbf{e}$ to ensure that the outputs are the correct y_i!) We will minimize the sum of the squared errors by choosing the proper m and b.

Once we've determined m and b, we can say that an increase of 1 unit of the independent variable correlates to an increase of m unit of the dependent variable.

To make a stronger claim, we want to control for other confounding variables; we want to avoid the **omitted variable bias**.

5.10.1 Multiple Linear Regression

Usually we instead run a multiple linear regression—this will include multiple explanatory variables x_i:

$$y = m_1 x_1 + m_2 x_2 + \cdots + m_n x_n \tag{5.1}$$

Where $m_n x_n$ is a constant term, the intercept.

One can derive that the slope m_1 is a function of all of the variables, for

$$Q = \sum_{i=1}^{n} (Y_i - (m_1 x_{1i} + \cdots = m_n x_{ni})^2$$

and thus

$$\frac{\partial Q}{\partial m_1} = f(x_1, x_2, x_3, \ldots, x_n, y)$$

I.e., we compute a slope m_1 assuming all other variables x_i with $i \neq 1$ are held *constant*, which is what allows us to control for the other variables!

We can then treat (5.1) as a matrix equation:

$$Y = X\beta$$

where β is a vector of coefficients. We would like to find the β which, once applied to X, returns the vector in the range of X *closest* to Y. This invites a projection matrix (read more here):

$$\hat{\beta} = (X^T X)^{-1} X^T Y$$

For some matrices it is not possible to compute the inverse $(X^T X)^{-1}$ (if X has linearly dependent columns). In that case use the pseudo-inverse (libraries do this for you). Ideally we throw out variables that are linearly dependent.

Dummy variables: Also called Boolean variables, one-hot variables, sparse variables. Say we have a non-numerical value like, say, a Boolean or a category. We can encode these types of variables in our model by assigning 0 or 1 depending on whether an observation satisfied that category.

$$\begin{bmatrix} 20 & 31 & 0 & 1 & 1 \\ 20 & 5 & 0 & 1 & 1 \\ 20 & 40 & 0 & 1 & 1 \\ 25 & 18 & 1 & 0 & 1 \end{bmatrix}$$

Where the third column is true (say, "had breakfest"), the fourth column is false, and the last column are the intercepts.

In this case, the intercept can than be interpreted as the predicted shift in the dependent variable depending on whether or not the Boolean is satisfied. We include a column for each category in order to study the effect due to each those categories separately (think, say, the effect of living in Rhode Island as compared to *not* living in Rhode Island).

Except the above matrix has an issue—the last column is linearly dependent on third and fourth columns, it is their sum! We've fallen for the **dummy variable trap**. We fix this by dropping one of the columns or categories (so that if the value is zero for all other columns, we assume it to be one for that category). (We *could* drop the constant term, but this makes for a difficult interpretation).

$$\begin{bmatrix} 20 & 31 & 0 & 1 \\ 20 & 5 & 0 & 1 \\ 20 & 40 & 0 & 1 \\ 25 & 18 & 1 & 1 \end{bmatrix}$$

Nonlinear relationships: include nonlinear terms. Add another variable which is the dependent variable squared, or cubed, or whatever.

Interaction terms: multiply two of the variables together.

5.11 Miscellaneous Definitions

Effect size: measures the size of the difference between two groups.

$$\frac{\text{mean of experimental group} - \text{mean of control}}{\text{std dev}}$$

The higher, the stronger the apparent impact of the different method/conditions/ etc. Unit of effect size will be application specific—e.g., number of smokers who quit, number of T-type cells.

As the study size increases, the effect size decreases. Lower effect size leads to weaker statistical evidence supporting the result.

- **Type I error**: null is true, and reject it. We really want to avoid this type of error in particular—say, we don't want to push a drug that does not work better than others. Our α value is the probability of a type one error.
- **Type II error**: null is false and we fail to reject it. Obtaining a guarantee that this error won't occur is actually quite difficult. This would require major assumptions on the data and randomness of the observed distribution.

5.12 Non-Parametric Methods

5.12.1 Parametric vs Non-parametric Methods

In parametric methods we make some assumptions about the form of the model and thus limit ourselves to the parameters of that form—e.g., linear regressions are parametric, with the parameters being the slope and intercept.

But let's say we are not comfortable claiming that this model is linear, or perhaps we do not know anything about the form that the data will take (maybe there's too little data). This is where one would use a non-parametric method. One example is *nearest neighbors*.

Note that with non-parametric models, one would need to keep track of every existing data point in order to make new predictions, whereas with parametric models we could just keep track of the parameters.

- **Pros:** Can work well with small data, or when you have very complex distributions and you aren't sure what assumptions can be made.
- **Cons:** Size of model can increase with size of data. Can be slow to compute. Fewer assumptions leads to weaker conclusions.

5.12.2 Non-Parametric Hypothesis Testing

Say we have some data and a test-statistic, and we would like to determine the probability of a certain test statistic to test a hypothesis. However, sometimes we do not have an analytic solution; this often occurs for one of two reasons:

- The assumptions about the analytic solution are inappropriate (e.g., sample size too small).
- The theoretical distribution is unknown, complex, or hard to write down.

Consider the former case. There are non-parametric versions of every important hypothesis test we have covered; their analogues are

Parametric	**Non-parametric**
Unpaired 2-sample t-test	Mann-Whitney (Rank Sum) or permutation test
One sample or paired 2-sample t-test	Wilcoxon Test
Chi-squared test	Fisher's exact test
Pearson correlation	Kendall-Tau and Spearman correlation

- Mann-Whitney U-test
  - The t-test assumes that samples are normally distributed. This assumption is made when we take the sample variance to be the population variance.

  - For large sample sizes, can invoke the central limit theorem. But in some cases the data are not clearly normal and we cannot make this assumption.
  - Consider the samples x_1, x_2. Rank the combined list, remembering which list every item came from (just put the values together in one big list).
  - Replace the values with their ranks (i.e., 1, 2, so on). Average the ties—if two ranks correspond ot the same value, say 1 and 2, we replace that with 1.5.
  - Compute R_1 be summing the ranks of the entries of the smaller sample and R_2 by summing the ranks of the entries of the larger sample.
  - The null hypothesis is that the mean of the two distributions are the same. The U-test statistic is computed as
$$U = \min\left\{R_1 - \frac{n_1(n_1+1)}{2}, R_2 - \frac{n_2(n_2+1)}{2}\right\}$$
These quotients that we are subtracting are the sum $1 + 2 + \cdots + n_1$ where n_1 is the size of x_1. If the values of x_1 are all ranked lower than the values of x_2, then we'd expect the difference to be zero. With complete separation between populations $U = 0$. If values are well interweaved, observe higher values of U.

- Wilcoxon Signed-Rank Test
  - Compute absolute difference of each pair of the two samples, and compute the sign of the differences.
  - Sort by increasing absolute differences. Ignore differences of zero. Again compute ranks and average ties.
  - The test statistic is
$$W = \sum_{i=1}^{n} \text{sign}(x_{2_i} - x_{1_i}) \cdot \text{rank}_i$$
Look up the p-value for this number of data points. Do not count zero-differences when counting number of data.

In practice the case that there is insufficient data is not very common (at least in the domain of data science) do the availability of very many data.

Now let us consider the case that the data is *complex, unknown, or hard to write down.* In some cases we are not interested in simple metrics like means or proportions—we could want to measure the median, the 90th percentile, so on. In this case, we would use **bootstrapping/resampling methods**.

The assumption behind bootstrapping is that the observed sample is representative of the population (i.e., it is IID and sufficiently large). Then we can *resample* from our sample in order to simulate the original sampling procedure. Make sure to sample with replacement—need each sample from the bootstrap to be IID.

We resample in order to approximate the distribution of the test statistic. Compute the tes statistic over each sample, repeat some large number of times, and view the distribution of the computed test statistics.

Application: deep learning.

- People are competing for the best model.
- Comparisons are not usually rigorous. Train a single model and compare the results, differences are often very small.
- Typical set up: Pretrain a very large neural network (typically only done once). Add intervention—i.e., new modeling ideas that you think will improve the model. Finetune model on a task of interest (done many times by many different researchers). Then evaluate on some pre-defined test to see how good your model is.

- Problem is there is a lot of noise in this process—random state of the machine when we pretrain the model, or when we are finetuning. Random sample of data used to test the model.
- Model without intervention M and model with intervention M'.
- Use boottrap sampling over N random states times K test examples to estimate difference between performance of M and performance of M'.
- Estimates of M and M' performance are paired in two ways: share pretraining states and

5.13 P-Hacking

Interpreting P-Values:

- **False interpretations:** P-value is the probability the null hypothesis is true, or p-value is the probability of the alternative being false. If we observe a non-significant difference between two groups, this means there is no difference between the groups (effectively a p-value interpretation again, except a two-sample test).
- **Correct interpretations:** $p = 0.05$ means that the probability of data we have observed, plus anything more extreme, would only occur 5% of the time assuming the null hypothesis is true.

"You can find almost anything if you look hard enough." If we run the same study under the same hypothesis 100 times, we expect at least $\alpha \cdot 100$ to be significant, even if the null hypothesis is true! This is *by definition*.

Bonferroni correction: Assume we are testing N hypotheses at the same time, e.g., for each state is there a significant difference between Republicans and Democrats on x (50 hypotheses), or is there a difference between CS majors and non-majors on x during any year of college (4 hypotheses). We should expect a significant effect $N \cdot \alpha$ times by chance. Then one computes the FWER: family-wise error rate, the probability of having at least one false positive. We use that to compute the Bonferroni correction: control the FWER by dividing α by N, and using that as the new significance level (i.e., p must be, say, greater than α/N).

When am I at risk of this so-called multiple comparison error?

- You are literally running the same test multiple times.
- Running several experiments and looking for significant results after the fact.

To avoid such errors:

- Pre-register your hypothesis/methods.
- Try to do *one* test—e.g., count total number of subjective words in each population and do a single test for the population proportion.

Researcher Degrees of Freedom: There can be a large number of potential comparisons when the details of the data analysis are highly contingent on data. In short the fallacy is forming a hypothesis that the data will help you confirm because the data informed your hypothesis.

There are enough categories that one of them will, as a subset, be significant. Avoid refining experimental design mid-study and especially in response to data, unless the study is *explicitly* exploratory.

This could look like changing the data preprocessing, aggregating differently, or using different tests. Again, these are fine things to do when you are conducting an *exploratory* analysis.

In general:

- Define your hypothesis ahead of time, based on independent data (not the data you will analyse).

- When possible, pre-register your methods.
- Recall that significance testing indicates the level of uncertainty.
- Stay curious, "recognize the actual open-ended aspect of your projects... and analyze your data with this generality in mind."

5.14 Text Processing

We will not cover machine learning techniques in this section. We will use tools like NLTK and Spacy.

Ways one might use NLP:

- Use text as a feature for some prediction task.
  - Say, classify sentiment on twitter, predict popularity of posts, track spread of articles/ideas.
- Wnat to make predictions/hypotheses about language *itself*.
  - Model changes in word use over time/across location, find words that cause articles to be shared.
- Clustering of text data—classify documents in some way,
  - In either of the above use cases .
  - Are these words similar, is this document similar to this query, are these documents similar to each other, etc.

Units of analysis:

- Characters `("s", "w", "i", "m", "m", "i", "n", "g", "l", "y")`
- Morphemes `("swim", "ing", "ly)`
- Words `("swimmingly)`
- Sentences
- Documents

Linguistic Preprocessing: language is ambiguous... take the sentence they "freaked out when they found the **bug** in their apartment." They could be referring to an actual bug, a computer bug, by "freaking out" one could mean they were excited, and so on.

Language is also redundant. There is a constant tradeoff between treating more words as though they were the same (redundancy) and preserving as much difference as possible (ambiguity).

One of the first processing steps is **tokenization** (phrasal collocations/morphological analysis?):

- Get rid of punctuation. `"okay..."` versus `"okay!"`
- Lowercase everything, or normalization."trump" versus "Trump".
- Get rid of numbers, replace with, say `<NUM>`
- Stop words, such as and, or, so on. `"pb and jelly"` versus `"pb or jelly"`.
- Tagging. Label the type of word: noun, adjective, so on, like in `"fish fish fish fish fish"`.
- Choosing a vocabulary:

- Remove "out-of-vocabulary".
- Remove rare words.
- Remove uninteresting words (tf-idf, pmi).
- Add a little syntax (POS tags, ngrams, pmi).

Finding Key Terms:

- Term-Frequency Inverse-Document-Frequency (tf-idf)
  - Assigns higher weights to words that differentiate this document from other documents:

$$\text{tf-idf} = \frac{\text{frequency of word in doc}}{\text{number of times it appears across all docs}}$$

  - Can filter out low tf-idf words or else just reweight the term-document matrix accordingly.
- Pointwise Mutual Information (pmi)
  - Again higher weights to words that differentiate this document from other documents.

$$\text{PMI}(word, doc) = \log P(\text{word}|\text{doc}) \dots$$

  - Used more fore finding word-label relationships or word-word collocations.

N-Grams are used to provide some context—we treat a sequence of N words (unigrams, bigrams, trigrams, 4-grams) as one word: "cute" and "dog" versus "cute dog." Blows up size of vocabulary, and increases sparsity.

Collocations Try to find just the interesting phrases by finding words that occur together more often than chance. Often use PMI for this.

Bag of words model: represent sentences/documents as just an unordered set of words. This is the basis of most modern NLP. It is used for information retrieval or search, clustering and recommending, and as input to most ML models.

Recall that the vocabulary is every unique word across all documents. Then we consider the document as a row vector with row value 0 if the document does not contain that word and a value 1 if it does. With multiple documents, these rows together comprise a **term-document matrix**.

5.15 Stats in Python

- **Multiple linear regression**: a commonly used toolkit is `statsmodels`, which is used as follows:

```
import statsmodels.api as sm

# read the data from a csv or whatnot
y, X = read_data()

# include a constant term column
X = sm.add_constant(X)

# use, say, ordinary least squares
model = sm.OLS(y, X)

# fit the model to the data
results = model.fit()
print(results.summary())
```

Chapter 6

Machine Learning

6.1 ML Preliminaries

Oversimplified ML: three main components: data, model, goal/task.

1. **Goal/task**: prediction of some kind. Price of stock, object in an image, strategy for a video game, parse tree of a sentence. Having a task is the first phase of a machine learning project.
2. **Data**: can be anything. Usually data size and/or representation is limiting factor.
3. **Model**: decisions about how the problem is structured *and* how to estimate the parameters. Linear/logistic regression, SVMs, naive Bayes, Bayesian networks, neural networks, etc.

Output features need to have an objective/loss function to quantify how well we're doing—i.e., how well we are predicting using the model.

EXAMPLE: Consider the task of increasing consumption, and reading habit data. Input features need to be concrete and representable. This quantification of our goal will be an **objective/loss function**. Our prediction target (measurement of increased consumption) could be clicks. Our loss function can then be the difference between the predicted and true values, or the squared difference between predicted and the true value, or the predicted probability of the true value, etc. Let us try to minimize the difference between the predicted total number of clicks and the actual total number of clicks.

The data will have **features**—things we will train on and make predictions off of (all will eventually have to be reduced to quantities, so think carefully about the encoding you will use). These will be stored in a **features/feature matrix** which will only contain numeric data. The feature matrix is what the machine learning will be conducted on. Sparse features are features that obtain a value 0 for most rows—consider, for example, a diverse vocabulary.

With the task and data defined, one can turn to considering what model to use. Models are, in essence, function approximators. They fit lines or curves in several-dimensional space. The fitting will occur on your training data.

6.2 Supervised/Unsupervised Learning

Unsupervised: common for exploratory analysis wherein there is little known labelling.

- No labeled data, no examples of what the "real" labels are.
- Good for exploratory analysis or preprocessing

- common for modeling natural phenomena, when you can make strong parametric assumptions.
- Currently very common for representation learning or "pretraining"
- Clustering: KMeans, Mixture models
- Dimensionality reduction: PCA, LDA
- Visualization methods: TSNE

Supervised: you are trying to make a prediction or label more data based on data you already know is labelled, etc.

- Labelled data.
- Label some data and want to train a model to label more, unlabelled data.
- Regression: linear/ridge/lasso, neural networks
- Classification: KNN, decision trees, logistic regression, SVMs, Naive naive Bayes, neural networks.

In addition, there are semi-supervised models which combine large amounts of unlabelled with smaller amounts of labelled, weakly/distantly supervised models which have noisy or partial labels, and reinforcement learning which label on the result of a sequence of actions but not on each action (think of a robot which plays a game).

There are also two different types of tasks, **regressions**, where there is a continuum of possible outputs, and **classifications**, where there is a discrete output, based on thresholds.

6.3 Train-Test Splits

When we train our machine learning model, we only have a measure of how well we've done with our dataset, we don't know how well we'll do with actual predictions. To ameliorate this we split our dataset into a "train" set and a "test" set. We fit our model to the train set, and when we test how good our model is, we apply it to the *test* set. We expect, for example, the MSE to go up, for we optimized the MSE on the train set not the test set.

The more powerful/flexible our model is, the more specific it'll be to our training set, and the worse it'll be for the test set. The main problem one will have to consider is picking a model that performs well on the test set.

One commonly assumes that the train and test sets are iid: they are drawn from the same distribution. Standard practice is to shuffle your dataset, shuffle it, split into 80% train and 20% test. In practice, the test isn't always iid—e.g., time-scale data is often difficult to generalize into the future.

6.4 Linear Regressions (as ML Models)

In the above example, we might have a linear regression of the form

```
clicks = m(reading_level) + b
m = cov(rl, c) / var(rl)
```

in this case the parameters are the slope and intercept and the training set is whatever data you give the model to fit on.

6.4.1 Two Faces of Linear Regression

Regression as it appears in ML is different than as it appears in statistics. They are the same model, but the context is not the same.

Regression in Stats	**Regression in ML**
Make claims about whether there is a meaningful relationship between X and Y	Given X, predict Y; deploy as model to make predictions for new inputs
Often interested in causation; focus on controls and removing colinearity	Focused on prediction accuracy; exploiting correlation is totally fine
A "result" is typically in the form of a significant relationship and/or practically relevant effect size	A "result" is typically in the form of an improvement in prediction performance on a held out test set
Avoid overfitting by preferring simple models; avoid overclaiming by accounting for "degrees of freedom" when computing p values.	Avoid overfitting through regularization; avoid overclaiming by maintaining tain/test splits and reporting test performance.

6.5 Unsupervised Learning

Unsupervised learning is used for "finding structure" or "finding patterns" as opposed to predicting or labelling. In data science this is typically for exploratory analysis, preprocessing or deciding what features to use (pretraining).

Unsupervised learning: finding structure or finding patterns (as opposed to labelling or predicting). IN data science this is typically for exploratory analysis or preprocessing and decising what features to use.

6.5.1 Clustering - K Means

Clustering concerns finding groups. Say: find groups of customers with similar tastes, find topics within a set of news articles, find genres within a library of music, extrapolate—make predictions about your new business based on behavior of similar old businesses.

K Means is an intuitive clustering algorithm that is commonly used. First, one defined the algorithm's **hyperparameters**, parameters that are not obtained by the algorithm itself and must be set by the user:

- K: how many clusters we want to find
- Max iterations: when to stop
- Minimum difference: the difference you consider to be sufficient

The algorithm then runs as follows: we randomly guess the means of the clusters, assume those means are correct. Now we assign data points to each cluster depending on which mean they are closest to. Then we recompute the means for each we've partitioned and plot those new means, then repeat the process. We stop when the difference between two means computed is less than the minimum difference. Alternatively, we might hit iteration threshold if a point is iterating between two means.

```
hyperparameters: K, max_iter, min_diff

iter = 0
change = if
means = [random() for _ in range(K)]
```

```
while iter < max_iter and change > min_diff:
    update_assignments()
    compute_new_means()
    change = max_i(dist(new_means_i, old_mean_i))
    iter += 1
```

The "loss" we are trying to minimize is distance of points to their respective clusters.

But *how many clusters do we use*? The more clusters you have, the lower loss, but the less meaningful our result (too many clusters!). It's not obvious a priori. To choose the right number of clusters, one may run the algorithm with several different numbers of clusters, plot the number of clusters on the horizontal axis and the mean distance to each data point's center on the vertical axis, and look for an "elbow point".

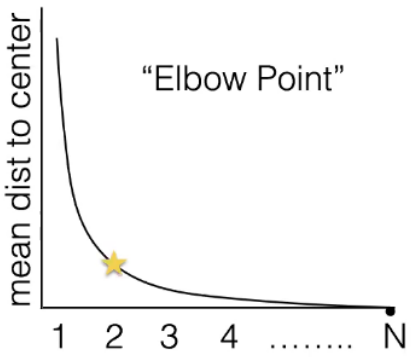

Figure 6.1: Note the labelled elbow point.

Other techniques include silhouette score, intuition/divine intervention, LGTM.

6.6 Dimensionality Reduction

Often our feature matrix may have tons of columns for a given piece of data. A lot of these columns won't be that useful. This wll lead to slower training, overfitting, and hard-to-visualize results.

We use **dimensionality reduction**—a linear algebraic method to transform the feature matrix—to ameliorate this We represent each data point in a *new feature space*. The new feature space is ideally: *more informative for machine learning*, since it consolidates redundant features and makes patterns clearer, and *less interpretable to humans*, since the columns of the feature matrix no longer refer to features that we have defined.

6.6.1 Principal Component Analysis

If we want to get away with fewer features, which feature is best to use? **Principal component analysis** (PCA) allows us to choose. Plot a `(feature 1, feature 2)` graph. Project the points onto feature 1 axis, see how close together the data is, how much overlap. then do the same with 2. Sometimes neither feature can reasonably be dismissed!

Well, perhaps let us create new orthogonal axes, (call these principal components) Choose a line, trying to preserve as much variation and distance between points as possible (upon projecting the data points onto that line). That first line will comprise a new axis, the first principal component. The second principal component will be an axis orthogonal to this, the third as well, and so on. We are effectively performing a *change of basis*.

Now we have an arbitrary number of features (the n principal components) we can choose from while feeling alright about throwing away the other components, for we have preserved variance in data.

This technique is really common when using large datasets, or datasets that are very sparse, e.g. image processing. A Python API to use is `sklearn.decomposition.PCA`.

Note that we are allowed to employ these dimensionality reduction techniques by making a "low rank assumption": we assume that our features contain a large amount of redundant information.

6.6.2 Singular Value Decomposition

Alternatively, we can use **singular value decomposition**; PCA and SVD are functionally equivalent for most purposes.

The linear algebraic theory of singular value decomposition is covered in detail here.

Start with an $m \times n$ matrix M. Each of your columns is a feature of that data point. We want to factorize our M into three matrices: U is $m \times m$, D is $m \times n$, V is $n \times n$. We obtain

$$M = UDV$$

where D contains the "singular values" of M, U represents the rows of M in the new feature space—it is exactly M but in basis of the principal components, and V is the change of basis matrix into the principal component basis.

We then yield as many columns as we had before; to actually reduce the dimensionality of the matrix, we take the first k columns of M, the upper-left-most k by k matrix inside of D, and the first k rows of V. This is called a truncated singular value decomposition.

A Python API to use is `sklearn.decomposition.TruncatedSVD`.

6.6.3 Uses of PCA/SVD

- Use the U matrix as input to a clustering algorithm or (supervised) classifier. This will give us fewer but better features to learn from.
- Use the V matrix to analyze the features in your data. This will give you a sense of what types of things explain differences between points in the data as a whole.
- For a given datapoint/set of datapoints, look at the U and V matrices to understand which features describe that datapoint.
- Use just the first two components (columns of U) to plot your data in 2D so you can visualize it.

6.7 Classification

The main goal of classification can be summarized by $P(X|Y)$ i.e., we want to find the probability of a label given certain features. For example: P(email is spam | words in the message), P(genre of song | tempo, harmony, lyrics), P(article clicked | title, font, photo).

For example, there is a supervised analogue to K Means, **K Nearest Neighbors**. This algorithm already contains labelled clusters, and, upon given a new, unlabelled data point, and set it to the cluster that the majority of its K nearest neighbors belong to.

This is a non-parametric model, no assumptions are made about the form of the model. Additionally, all work is done at classification time, and as such we need to store every existing training data point. *But* it does work with very small amounts of data.

There are two approaches to classification models:

Generative model	**Discriminative model**
Estimate $P(x, y)$ first, explicit probability model	Estimate $P(Y\|X)$ not an explicit probability model
We can assign probability to new observations, generate new observations	Supports only classification, less flexible, made to discriminate between different classes
Often more parameters, but more flexible	Often fewer parameters, better performance on small data
Naive Bayes, Bayes Nets, VAEs, GANs	Logistic Regression, SVMs, Perceptrons, KNN

6.7.1 Naive Bayes Classifier

Consider wine reviews; we would like to classify new reviews as either positive or negative. We will use a term-document matrix with one-hot columns for each word, and a column for the label—1 for positive, 0 for negative.

The Naive Bayes Classifier, a generative model, uses Bayes's rule:

$$P(Y|X) = \frac{P(X|Y)P(Y)}{P(X)}$$

Here $P(Y|X)$ is the posterior, $P(X|Y)$ is the likelihood, $P(Y)$ is the prior, and $P(X)$ is the marginal. we actually don't care about the denominator (plus, it's hard to find because we have a sample, not the population) because we're just trying the maximize the left hand side.

We are interested, in, say, $P(\text{label} = 1 \mid \text{lovely, good, ...})$, which, using Bayes's rule:

$$\begin{aligned} P(\text{label} = 1 \mid \text{lovely, good, ...}) &= P(\text{lovely, good, ... } \mid \text{label} = 1) \\ &= P(\text{label} = 1, \text{lovely, good, ...}) \\ &= P(\text{lovely} \mid \text{label} = 1,\ \text{good, ...})P(\text{label} = 1,\ \text{good, ...}) \end{aligned}$$

Symbolically,

$$P(C|x_1, x_2, \ldots, x_k) = P(x_1|x_2, \ldots, x_k, C)P(x_2|x_3, \ldots, x_k, C) \ldots P(x_k|C)P(X)$$

But this is hard for us to estimate in practice; there are lots of parameters, we need counts for every combination of features, and likely most combinations of features are never observed. Instead assume features are independent.

$$\begin{aligned} P(C|x_1, x_2, \ldots, x_k) &= P(x_1|x_2, \ldots, x_k, C)P(x_2|x_3, \ldots, x_k, C) \ldots P(x_k|C)P(X) \\ &= P(x_1|C)P(x_2|C) \ldots P(x_k|C)P(C) \end{aligned}$$

i.e., assume probability of x_2 is the same regardless of whether x_3, x_4, etc. are true.

Now we can populate a matrix with the probabilities of a feature X_i given the label is 1 or 0, and subsequently use these probabilities to compute $P(C|x_i) = P(x_i|C)P(C)$ (What are the probabilities of the classes—$P(C)$? We could say the classes are equally probable, or use the training data, or use real-world frequencies.)

Then we compute the probabilities $P(C = 1)$ and $P(C = 0)$ by multiplying the probabilities $P(x_i|C = 1)$ and $P(x_i|C = 0)$ for each word x_i that appears in the review we are classifying. If a word does not appear we omit that word from the multiplication. We classify the new review as whichever probability is highest.

Bayes isn't great for anomalies, for it is really hard to overcome the probability $P(C)$. But it is a good, simple, intuitive classification model, and it is resistant to overfitting.

6.7.2 Logistic Regression

Linear regressions have weight vectors and feature vectors:

$$y = w_1x_1 + w_2x_1 + \ldots + w_kx_k = \mathbf{w} \cdot \mathbf{x}$$

We're going to turn this from a linear regression to a logistic regression to make it a classification—not prediction—algorithm. **Logistic regressions** push high values higher, low values lower:

$$y = \frac{1}{1 + e^{-(\mathbf{w} \cdot \mathbf{x})}}$$

Where we've plugged the linear regression into this sigmoid function, turning a line into the curve pictured in figure 6.2.

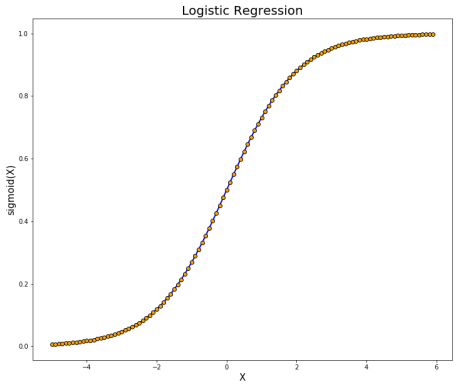

Figure 6.2: A sigmoid function, used for logistic regressions.

Logistic regression is effectively the classification equivalent of linear regression.

We train this model with **gradient descent**. In a linear regression we find the weight by minimizing the sum of the squares. Now we do the same thing: minimize the difference between the predictions and the actual values. But this time we minimize on a logistic function (which makes it difficult to solve analytically for a minimum):

$$\text{loss} = -Y \log \hat{Y} + (1 - Y) \log\left(1 - \hat{Y}\right)$$

The idea of gradient descent is we'll pick a random value for these weights, then take the gradient of the loss function with respect to the weights, and "head" in the negative of that direction to decrease the value by the most we can. Then repeat this iteratively until we reach a minimum.

Once we find the weight vector of this gradient descent procedure, $\mathbf{w}$, then apply it to the above sigmoid equation.

Logistic regressions will also yield weights, but they're not interpretable as *probabilities*. Instead, if we get a weight of one, we should say: 1 is the coefficient on the blank variable in linear regression (i.e., the $\mathbf{w}$ vector) that minimizes the log loss.

6.7.3 Decision Boundaries

This is the boundary one "draws" to, say, separate clusters ("if this dot is to the left of the line, it's part of this cluster, if to the right, it's this other cluster"). These boundaries can often be quite complex, and hard to visualize with many dimensions.

There are linear models such as linear regressions, logistic regressions, Naive Bayes, decision trees. There are also non-linear models: support vector machines (with nonlinear kernels), neural networks (with activation

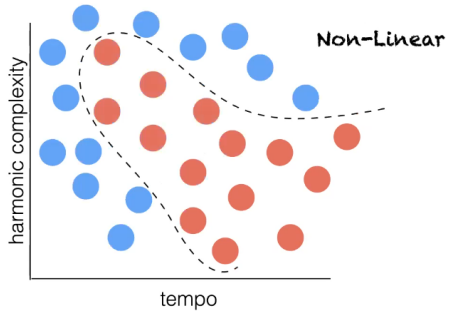

Figure 6.3: A non-linear decision boundary.

functions). Non-linear models use "tricks" (kernels, activations) to transform feature vectors (e.g., applying sigmoid functions to the vector). Then they can learn linear combinations of non-linear features.

In general, start with linear models first. They are simpler (less likely to overfit), easier to interpret (features are the same as the ones you put in), and surprisingly powerful. Try nonlinear ones too, but only use if there is good reason, such as large improvements on the heldout dev set.

6.8 More Train/Test Splits

By definition, trained models are minimizing their objective for data they see, but not for the data they don't see. We are really more interested in how the model performs on the data it does not see. This is the purpose of splitting the data into training and tesst sets—to assess performance on test data using parameters set on train data.

To make our machine learning development rigorous, we actually split into three sets—train, **development** or **validation**, and test.

- Never peek at the test data so that you don't make biased choices.
- Instead use dev (or cross-validation on train) to make decisions about how to design your model.
- Only when research is done you evaluate on the test data.
- Use dev to make meta-decisions:
  - Which model is better (generalizes better to held out data)?
  - What regularization to use?
  - How many training iterations?

Alternatively, you can use cross validations to get a stable estimate of test performance. This involves iteratively shuffling into an 80-20 split and obtaining an accuracy on the test data for this split, then taking the average of these accuracies.

```
accs = []
for i in range(num_folds):
    train, test = random.split(data)
    clf.fit(train)
    accs.append(clf.score(test))
```

6.9 Regularization

Overfitting: Models are likely to overfit when the model is more complex than is needed to explain the variation we care about. By complex we mean that the number of parameters (features) is high. When the number of parameters is greater than or equal to the number of observations, we can trivially memorize the training data (set weights such that it exactly memorizes the training set) without learning anything generalizable to the test data set.

This can mitigated by regularization. Here are some types of regularization:

- Penalty term—incur a cost for models that are more complex: Adds an extra hyperparameter (λ) which controls how much you penalize:

$$\min_{\theta}(\text{loss}(x, \theta) + \lambda \cdot \text{cost}(\theta))$$

- Early stopping—for iterative training procedures such as gradient descent, stop before the model has fully converged (assume final steps are spent memorizing noise).

We effectively make the training on the training set worse, but this should bring an improvement to the performance on the test set.

6.9.1 Norms

There are two types of penalty terms that one will encounter (the second term in the minimum functon above):

- L1 norm:

$$l_1 = \sum_i |x_i|$$

  Penalizes non-zero weights, encourages sparsity (using fewer features). More interpretable.

- L2 norm:

$$l_2 = \sqrt{\sum_i x_i^2}$$

  Penalizes large weights (positive or negative), encourages models with smaller weights. Usually more stable.

6.9.2 Regularized Linear Regression

- Non-regularized linear regression:

$$\min_{w}((y - \mathbf{w} \cdot \mathbf{x})^2)$$

- Lasso regression—linear regression with $L1$ penalty on the loss

$$\min_{w}((r - \mathbf{w} \cdot \mathbf{x})^2 + \lambda l_1(\mathbf{w}))$$

- Ridge regression—linear regression with $L2$ penalty on the loss:

$$\min_{w}((y - \mathbf{w} \cdot \mathbf{x})^2 + \lambda l_2(\mathbf{w}))$$

Logistic regression usually uses L1 or L2 regularization by default (e.g., `sklearn`).

6.10 Feature Selection

Why perform "feature Selection"? You have too many features (way more features than data points), including all of them slows down training and leads to overfitting. Feature selection is more of an art than a science, trial and error will teach you. Look at your features (e.g., plot correlations between them), and use intuition—if a feature doesn't seem relevant, don't include it.

Some strategies:

- Remove features with low variance.
- Remove features that are highly correlated with existing features
- Iteratively add/remove features with highest/lowest weight or information gain (see here).
- Use dimensionality reduction.

6.11 Deep Learning

6.11.1 TLDR

Deep learning is a type of machiune learning: deep learning $\subset$ machine learning $\subset$ AI. The main benefit of deep learning is it learns feature representations as a part of training. Allegedly there is less time spent feature engineering (though in practice, just as much guess-and-check if required). These feature representations are then optimized for the test—leading to better performance.

Feature learning in DL is essentially a type of dimentionality reduction—a similar idea to SVD. In SVD or PCA, the new features are linear combinations of the olde features; however, in deep learning, the new features can be quite complicated, for you can have many, many layers of modeling. On each iteration you obtain features that are further and further removed from your input features.

6.11.2 Linear Regression (as Perceptrons)

Again, we have a model of the form $y = wX + b$. We can write this in matrix form as well:

$$\begin{bmatrix} x_{11} & \cdots & x_{1m} \\ \vdots & \ddots & \vdots \\ x_{n1} & \cdots & x_{nm} \end{bmatrix} \begin{bmatrix} w_1 \\ \vdots \\ w_m \end{bmatrix} = \begin{bmatrix} y_1 \\ \vdots \\ y_n \end{bmatrix}$$

Or, if focusing one one data point,

$$\begin{bmatrix} x_1 & \cdots & x_m \end{bmatrix} \begin{bmatrix} w_1 \\ \vdots \\ w_m \end{bmatrix} = \begin{bmatrix} y \end{bmatrix}$$

In a logistic regression, this output y will be between zero or one; for a classifier we use an activation function τ such that if y is greater than τ we round to one ("true") and otherwise we produce zero.

$$y = 1 \text{ if } \mathbf{w} \cdot \mathbf{x} > \tau \text{ else } 0$$

This is called a perceptron; it is the most basic neural network. A one-layer perceptron is called an **on-line** perceptron, as it trains one data point at a time. But we have not learned any features yet, and we are not gleaning anything beyond a normal linear regression.

6.11.3 Hidden Units

We will use this basic network (perceptron), but rather than predicting y, predict an arbitrary "latent state" h first, then use h to predict y. h is your new feature vector (recall we are stil considering one row of the feature matrix only). Essentially, we chain two features together:

1. Given x predict features.
2. Given features predict y.

the middle step between one and two replaces the part where you choose the features.

Train entire thing "end to end" so that features are chosen such that they are whatever they need to be for the model to perform as well as possible at predicting y.

Mathematically, we can write this as such:

$$\begin{bmatrix} w_{11} & w_{12} & w_{13} \\ w_{21} & w_{22} & w_{23} \\ w_{31} & w_{32} & w_{33} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} h_1 \\ h_2 \\ h_3 \end{bmatrix} \begin{bmatrix} w_{2_1} & w_{2_1} & w_{2_3} \end{bmatrix} = y$$

The first vector $\mathbf{x}$ times the top row $\begin{bmatrix} w_{11} & w_{12} & w_{13} \end{bmatrix}$ is just a logistic regression that ends up being the first entry of $\mathbf{h}$, h_1. Then we do the same thing but with the second row of the matrix w, and the third. Then $\mathbf{h}$ will itself be used as a new feature vector in a fourth and final logistic regression. Each of the entries of $\mathbf{h}$ is called a **neuron** (stronger weights for that particular entry of $\mathbf{h}$ can be thought of as stronger connections).

In practice, the size of the "hidden state" $\mathbf{h}$ is a hyperparameter you can set. You might want a smaller hidden state in order to get rid of redundancies.

In reality, we don't actually output $\mathbf{h}$. We instead use

$$\begin{bmatrix} w_{2_1} & w_{2_2} & w_{2_3} \end{bmatrix} \begin{bmatrix} w_{11} & w_{12} & w_{13} \\ w_{21} & w_{22} & w_{23} \\ w_{31} & w_{32} & w_{33} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = y \implies W_2 \cdot (W_1 \cdot \mathbf{x}) = y$$

We compute the gradient with respect to the parameter; that said, there are a lot of parameters, we're not sure which parameters to update and by how much. One sees our loss function is

$$f = \mathcal{L}(W_2 \cdot g(\mathbf{x})) \text{ where } g = W_1 \cdot \mathbf{x}$$

Then the gradient becomes

$$\frac{\partial f}{\partial x_i} = \frac{\partial f}{\partial g} \frac{\partial g}{\partial x_i}$$

This procedure of partial differentiating with the chain rule on every layer at once is called **backpropagation**.

We *could* add more and more hidden feature vectors until you have some $\mathbf{h}_1, \mathbf{h}_2, \ldots, \mathbf{h}_k$, so k layers. This is what we mean by "deep" learning. But the issue is we become highly specialized to our training data, and risk overfitting.

The loss function can be any differentiable function of the form $f(\text{pred}, \text{true})$. Commonly MSE if continuous, cross entropy if true is categorical.

6.11.4 Gradient Descent

We don't know what the optimal $\mathbf{w}$ is (we are trying to minimize some loss function $\mathcal{L}(\mathbf{w}, \text{data})$ which is a function of the data and $\mathbf{w}$), and there is no analytical solution that yields $\mathbf{w}$.

Start by guessing what the parameters should be—randomly generate $\mathbf{w}$. Then compute the loss. We then take the gradient of the loss with respect to the parameter coordinates:

$$\nabla \mathcal{L} = \left(\frac{\partial \mathcal{L}}{\partial w_1}, \frac{\partial \mathcal{L}}{\partial w_2}, \ldots, \frac{\partial \mathcal{L}}{\partial w_n} \right)$$

Take a small step, set $\mathbf{w}$ equal to that location, and reevaluate the gradient, and repeat until you either converge, or the steps increase the loss in all directions, or you time out. Keep in mind you might get stuck in a local minimum.

There are some optimized neural network classifiers:

- Multilayer perceptrons
- Convolutional Neural Networks: specific for image processing; considers "clusters" of two-dimensional data.
- Recurrent Neural Network: temporal or linguistic data; order of inputs matters.

6.11.5 Transfer Learning

The idea is to train a model to do some task T1 where there is a lot of data. Let the model converge. Now your hidden states contain whatever features were good for T1. Maybe these features are good for some other task T2 as well. Now perhaps you can do T2 with less training.

6.11.6 Should I Deep Learn It?

Do you have a ton of data? If not, no (unless the task can be accomplished using some pretrained model). If you have a decent amount (1000s), then sure, but start with simpler models first like MLP, or even a basic logistic regression. If you have a *lot* of data, then yeah, probably.

6.12 More Natural Language Processing

6.12.1 Topic Models

A topic model is a procedure for describing a document in terms of topics as opposed to individual words. Usually involves unsupervised learning algorithms. Use for exploratory analysis and as a preprocessing step for ML.

In a bag of words representation, documents are a set of unrelated words. In your topic model, each topic is a set of words that are *related*:

```
topic1 = ['stencil', 'instructions', 'part', 'step', 'rubric', ... ]
topic2 = ['html', 'javascript', 'debug', 'display', 'elements', ... ]
topic3 = ['mac', 'windows', 'linux', 'firefox', ... ]
topic4 = ['I', 'you', 'when', 'the', 'and', ... ]
```

There are two main types of topic models:

LDA Topic Models: yields a more thorough model of where the document came form, "the generative story."

1. Sample a topic
2. Sample a word from that topic

LDA	LSA
Latent Dirichlet Allocation (latent means not directly observed; Dirichlet means prior follows a Dirichlet distribution)	Latent Semantic Analysis
Generative topic model	Discriminative model
Set parameters using EM or MCMC	Set parameters by factorizing the term-document matrix.

The probability of seeing a word w_i is

$$P(w_i) = \sum_{j=1}^{T} P(w_i|z_i = j)P(z_i = j)$$

where z_i is some topic. We assume that words are conditionally independent of one another.

We then set the parameters to maximize the probability of obtaining the words we are actually observing. This means finding some distribution of topics in a document and within each topic a distribution of words.

LSA Topic Models: Recall SVD; we decompose the term-document matrix. It's like principal component analysis (PCA) where each component is a "topic": a distribution over topics. In practice, usually use non-negative MF not SVD to facilitate interpretability.

6.12.2 Word Embeddings

Bag-of-words models are fine but we lose some information that language conveys. We would like our model to know that, say "cat" and "kittens" are words with similar meanings. How do you represent the meaning of word?

- The word's definition?
- Set of things the word refers to? I.e., cat $= c$: c is a cat
- The distributional hypothesis: the meaning of a word is defined by the contexts in which that word can be used.

We can actually use the distributional hypothesis to build a model. We build a vector for the word, say, "cat" which captures how often each of the words that appears along with "cat" appears. Repeating this process for several words, we now can construct a **word-context** matrix. Each cell a_{ij} in this matrix is the number of times the word i occurs with word j.

Now we can easily take a dot product between two words to see how similar they are.

Some other models to represent word meaning are:

- **Vector Space Model** (VSM): anything that represents a word as a vector.
- **Distributional Semantics Model** (DSM): Representation is derived from context in which the word is used (usually a VSM, so components of the vector are derived from contexts. So you can have a DSM that's also VSM, but you can also have a VSM without a DSM.)
- **Word Embedding**: dense, low-dimensional vector representation of the word, as opposed to space BOW models we discussed earlier.

Word Embeddings: Where do they come from? Derived from same data used to build a vanilla BOW data (word-context matrix), but it is lower-dimensional and often non-linear. They are built using matrix factorization or deep learning.

Word embeddings from deep learning will try to predict what words come before and after for example,"cat." The hidden states are the embeddings. So you can train a NN to try to predict what word comes after a specific word, then use that network's hidden states as embeddings.

Word embeddings are used just as you would use a bag of words model: instead of bag of words model, it is a bag of vectors model. The representation of a document is then the sum of the vectors of the words in that document.

6.13 Time Series

It is common to deal with temporal data but it is difficult to deal with this type of data because you are modelling several things at the same time. You could be trying to model cyclic trends while at the same time predicting future data, especially extreme events.

The main model used in time series is **autoregression**. It is just another linear regression, but with past values as explanatory values.

You will be dealing with "lag": how far back to look. Usual tradeoffs: longer lag leads to more powerful but also sparser signal, more overfitting. For example, AR(2), a second-order autoregression, autoregression with lag length of 2. The model is fit using OLS just like linear regression, although other procedures exist:

$$x_t = mx_{t-1} + b + e$$

We can also have a p-th order AR, which goes back to as far as $t - p$, and capture more complex trends:

$$x_t = m_1 x_{t-1} + m_2 x_{t-2} + \cdots + m_p x_{t-p} + b + e$$

where an increase of 1 in the value at time $t - p$ leads to an increase in m_p at time t, all else fixed.

Vector autoregression: we will have lags of multiple variables, i.e., multiple time series. E.g., let p be the price and d be the demand:

$$p_t = m_1 p_{t-1} + m_2 d_{t-1} + b + e$$
$$d_t = m_1 p_{t-1} + m_2 d_{t-1} + b + e$$

`statsmodels` has packages for vector autoregression.

These models can be used for hypothesis testing, but also for **forecasting:** we can use the models to predict at a time $t+2$, by feeding in prediction for $t+1$ as if it's a past variable. Though beware errors can compound further out, because you're creating a "feedback loop."

Some AR Variants:

- Autoregressive-moving-average: incorporates notion that error can be a function of past errors.
- Autoregressive-integrative-moving-average: incorporates additional normalization by modeling changes since last step.

Some useful packages are

- **Hypotehsis testing or prediction**: use `stats_models`.
  - `statsmodels.tsa.vector_ar.var_model.VAR`
  - `statsmodels.tsa.arima_model.ARMA`
  - `statsmodels.tsa.arima_model.ARIMA`
- **Prediction with lots of data**: consider deep learning, using Pytorch or TensorFlow.

6.14 Fixed Effects Model

Panel data: you have grouped data, and repeated measurements on the same group. Also called longitudinal data, cross-sectional data. Cons: observations not independent, and you need to deal with that. Pros: observations not independent! So they can help yu control for "effects otherwise not specified."

Omitted variable bias—we assume that the dependent variable can be predicted from the explanatory variables only. Assume changes in the dependent variable are correlated with the explanatory variable *because* of the explanatory variable.

Controlling for unobserved effects: your explanatory variables do not need to cover all possible causes. But with panel data, you can make the assumption that these things you did not control for are constant within a group. Then you can divorce their effect from the primary explanatory variable of interest. We say that there are group-specific idiosyncracies.

Is my data panel data? do you have some kind of repeated measuremenet on an individual or group? Is there anything you can group by that would be meaningful: state, country, person, product category, genre, etc. Also, if you have any reason to believe these features are correlated.

Random Effects Model: Assume that all of unobserved effects are not correlated with other explanatory variables, and assume that they don't vary over time. Pro: can estimate coefficients for time-invariant effects. Con: usually assumptions don't hold. It is usually safer to use fixed effects as a model.

The only assumption of **fixed effects models** is that the effects don't vary over time; we allow the possibility that they are correlated with other explanatory variables. This means we can control for the effects (they will become part of the intercept) but we can't actually estimate the coefficients.

FE Models are estimated by preprocessing, e.g., subtracting the within-subject mean from each variable, or adding dummy variables for each person, or other more advanced techniques.

`statsmodels` implements this model; for example:

```
import statsmodels.api as sm
import statsmodels.formula.api as smf

data = read_data()
model = smf.mixedlm("GPA ~ Course_Load", # regression you are interested in.
                    data,
                    groups=data["Person"]) \ # group effects you want to
                .fit()                       control.
```

`statsmodels` will output something like this:

```
Mixed Linear MOdel Regression Results
=========================================================
    ...
---------------------------------------------------------
            Coef. Std.Err.   z   P>|z| [0.025 0.975]
---------------------------------------------------------
Intercept   15.72    0.788  19.95 0.00  14.312 17.25
Course_Load  6.95    0.174  207.5 0.00   6.441  7.02
^ coefficient and p-value for the variable you are
  interested in
Group Var   40.45    2.149
^ Variance explained by group difference--not
  directly interpretable the way the other coefs are
  (no individual coefs or p-values)
=========================================================
```

Random Effects Model Assume: Unobserved effects are not correlated with other explanatory variables and they don't vary over time. Pro: can estimate coefficients for time-invariant effects. Con: usually assumptions don't hold.

6.15 ML Fairness

In some applications, there is a moral and/or legal obligation to value something other than prediction accuracy. We should have the ability to specify what our models learn. Modeling the world is not the same as making predictions based on past events.

EXAMPLE: Consider the task of, say, filtering rec letters. The model could pick up on non-explicit cues about a person's demographic without any explicit mention. The whole point of building word-embeddings was that we wanted to treat semantically similar words similarly. Now we see this backfires—we can further associate words that have gendered correlations with that gender. Some similarities from a study:

Similar to 'she'	**Similar to 'he'**
homemaker	maestro
nurse	skipper
receptionest	protege
librarian	philosopher
socialite	captain
hairdresser	architect
nanny	financier
bookkeeper	warror
stylist	broadcaster
housekeeper	magician

6.15.1 Bias in Input Data

Take the example of designing a navigation app. Some sources are:

- **Poorly selected data:** designers decided that certain data are important to the decision but not others. E.g., only include roads, not public transit, etc.
- **Incomplete, incorrect, outdated data**: lack of technical rigor and comprehensiveness to data collection. E.g., bus or train routes not updated as quickly as road traffic.
- **Selection bias**: set of data inputs to a model is not representative of a population. E.g., data collected from smartphone users.
- **Unintentional perpetuation and promotion of historical biases:** Feedback loop causes bias in results of the past to replicate for the future. E.g., hiring for "culture fit." Challenging because it means you cannot simply collect data naively.

6.15.2 What is "Fair"?

- **Fairness through unawareness**: simply ignoring those variables.
- **Demographic parity**: Your distribution should be the same as the population distribution. But there's no guarantee the model will perform equally well for both groups. The minority group might be practically randomized, prioritizing the majority group. No guarantee the actual classification will be fair, only that it will be distributed properly.

- **Equal opportunity:** Make sure that the harmful classification has an equal chance for both groups. But beneficial classification need not be equally distributed.
- **Equalized odds:** the above, but we also ensure that both groups have equal odds of obtaining the beneficial outcome.
- **Counterfactual fairness**: look inside the model and investigate the part that made it make that demographic decision. Then take that part and change it to benefit another group, and see what happens.

There are several challenges that we face:

- **Feedback loops**: people change their behavior to influence outcomes. Think college applications. People learn what the model looks for.
- **Transparency**: not just about the outcome, it's about understanding the process, understanding why the model chose what it chose.
- **Decision makers vs. stakeholders**: who makes the decisions about what's fair versus who is affected by such decisions?
- **Labeled data**: Research on fairness requires data about potentially sensitive and/or subjective attributes. Collecting such data raises concerns about privacy violation and other ethical issues.

Chapter 7

Data Visualization

7.1 When to Visualize?

- At the very start of analysis to find out what the hell is going on in your data.
- Periodically throughout to vet the quantitative trends you are seeing.
- At the very end of a project, to showcase the results.

In the first three steps, you are the main audience, so it is not necessary to make the graph look pretty and super presentable; these are concerns that arise in the last scenario.

7.2 The Three Pillars of Data Visualization

1. **Clarity**—your figures should speak for themselves. The analysis should be understandable and conclusions should be obviously supported, without too much effort.
    - Labels should be present and descriptive. No abbreviations.
    - All data should be visible, skewed data shouldn't dictate the axis— e.g. don't hide bins in a histogram because of outliers. Can fix this by using logs (but make sure to be clear you did so) or removing outliers (but say, like,"fifteen subjects in the age range 80-100 are not shown"). Also, overlapping data shouldn't obscure trends: just plot everything separately, or, for a scatterplot, add jitter (but be clear you did so, specify amount of jitter), or make points transparent `plt.scatter(...,alpha=0.4)`.
    - Chart type matches data you want to display. E.g., phenomena that evolve over time are not well visualizable by pie charts.
    - The point you want to make should be the most visually salient trend. Regroup so that colors/layout reflect the trend you want to highlight. Use words, explicitly label the trend you want people to see, perhaps in the title.
    - Axes in charts/rows should be sorted in a meaningful way.
    - Colors/sizes/fonts should be legible and accessible (think colorblindness). Blue and orange look good in grayscale and are colorblind. Use at a minimum a 24 point font. friendly.
2. **Honesty**—don't obfuscate your data or hide the process used to come to the conclusions. Give people enough data so they can disagree with you if they want.
    - Charts should include clear baselines or points of comparison.

- Present more than just summary statistics. Tradeoff between overwhelming the audience with too much data and not hiding anything.
- Axes should be scaled in a way that is informative, not to over/understate trends.
- Honesty and clarity can compete with each other.

3. **Minimalism**—substance over style. Make your point concisely, without redundant or distracting information or ornamentation.
   - Use color as a communicative tool, not just self expression. For example, if you're comparing two bar charts, their positions and the axis labelling should communicate all you need to communicate.
   - Simpler charts are better. If a boring plot does the job, use the boring plot. No reason to have shading or gradients or drop shadows.
   - Don't use superfluous animations or decorations. No three- dimensional charts.
   - Present all and only the numbers needed to make a conclusion. Choose reasonable significant figures. Don't let rounding overstate trends, but don't bloat the chart with extra digits.

7.3 Data Viz Cheat Sheet

Type of Hypothesis	**Recommended Plots**
Group A differs from group B according to metric C	Side-by-side histograms with means and CIs
X affects Y	Scatterplot with correlation
Prediction tasks, recommendations	Dimensionally reduce feature matrix to 2D, scatter and color by label/group
Any of the above	Correlation matrices between all features
Any of the above	Counts of all features (broken down by groups/labels if relevant)

7.4 Libraries

- **Matplotlib**: Oldie but goldie. Not super streamlined but gives the user much control.
- **Seaborn**: Plays well with numpy, streamlines process for making complex charts, harder to tweak little things.
- **Plotly**: Good for quick interactive charts.
- **D3** (Javascript): good for very flashy plots.

Chapter 8

Miscellaneous

8.1 Recommender Systems

Matrix factorization can be used for **matrix completion**. These are known as recommender systems, and widely applicable. Some examples include: suggesting movies based on past movies, suggesting songs on a playlist, suggesting products based on past purchases, suggesting friends on social media.

You will have some matrix where the rows represent users and the columns represent the movies they watched. There are unobserved values, the blank columns:

$$\begin{bmatrix} 1 & 0 & 1 & & \\ 0 & & & 0 & 1 \\ 1 & 0 & 1 & & 1 \end{bmatrix}$$

We will try to exploit similarities between rows. We see the third row has the same three entries as the first row, so we would like something like this:

$$\begin{bmatrix} 1 & 0 & 1 & & 1 \\ 0 & & & 0 & 1 \\ 1 & 0 & 1 & & 1 \end{bmatrix}$$

Recall singular value decomposition. We make the "low rank assumption" and assume we have redundant information, assume many of the singular values are zero. In real life, the matrix is likely full-rank, it is very unlikely that one person is predictable based on another person; i.e., very unlikely two people have seen the same movies. But we assume that the matrix is *close* to a low rank matrix, and we approximate the matrix it is close to; after the factorization we get a "de-noised" version of this matrix.

However, the data is often incomplete: it might be missing values or have new observations. We can still use SVD for this. We use a truncated SVD where we

We would like to minimize the difference between the matrix M and the product UV, except ignore the values that are missing in M—just don't compute their differences. Our loss function is then:

$$\min_{U,V} \sum_{i,j} (M_{ij} - u_i \cdot v_j)^2$$

Upon making the minimization we get a completed matrix M':

$$M \approx UV = M'$$

Our matrix might then be something like this, a matrix that is completely filled in:

$$M' = \begin{bmatrix} 0.9 & 0.1 & 0.89 & 0.25 & 0.98 \\ 0 & & & 0 & 1 \\ 1 & 0 & 1 & & 1 \end{bmatrix}$$

Once we have computed this matrix, there are some additional applications: U has row embeddings we can use to cluster/find similar users. The completed matrix M' can be used to make actual recommendations. V can be used as feature embeddings/"topics" to analyze patterns in your content (content beings movies, songs, etc.).

A Challenge: say we have added a new film or a new user to our platform SVD does not handle this well. With all-empty rows for that user, it will never populate that row and they will obtain all zeroes. One thing we can do is make a default set of recommendations and wait until you've gathered some data on the user. Or we can ask new users to provide initial interests or likes.

For new columns—new content— we can recommend to some people and see who views it ("new/featured" category). Or we can manually group this new column with similar items.

For both: we can use data other than view data for populate column entries for, say, a new user—e.g., geographical location, friends, so on.

Matrix Factorization Train/Test Splits: in most cases we will not hold out users or movies because we cannot generalize from there. Typically what is done is hold out some observed cells, pretend they are unobserved. The test is then effectively: how well do we predict something a user actually liked?

8.2 What to Take Next?

- Databases: CSCI1270
- MapReduce: CSCI1380, CSCI1570
- Stats: APMA1650, ECON1620, CLPS0900, SOC2010
- Machine Learning: CSCI1420, CSCI1470, CSCI1410, CSCI1460