Unit 4 Review

15-110 – Wednesday 12/02

Agenda

- Unit 4 Overview
- Matplotlib
- Heuristics
- **Lecture 1:** Monte Carlo methods and Reading/Writing files
- **Lecture 2:** Time-based Simulation and Minimax

Unit 4 Overview

Unit 4 Goals

Our fourth unit explored how computer science could be used as a **tool** in other domains. We discussed this in two stages: how to **model** domain-specific data, and how to **answer questions** about that data.

How did the topics we discussed fit into these themes?

Modeling Data

In data analysis, we discussed how you can **read and write files** and **interpret data with protocols** to load real data into a program. We also learned how to **reformat** data to meet the task's needs.

In simulation, we learned about the **model-view-controller** framework, where we store data in a shared structure, represent it graphically with a view, and update it **over time** or based on **events**.

In machine learning, we talked about how to choose different **categories of reasoning** based on the **features** being analyzed. We also used **game decision trees** to model data for AI agents.

Answering Questions

In data analysis, we discussed a variety of **analysis methods**, and used **Matplotlib** to visualize data as charts.

In simulation, we used **randomness** and **Monte Carlo methods** to run **experiments** over simulations and find the expected results.

In machine learning, we discussed how we use data to **train, validate, and test** a reasoning model, and how an AI can **perceive, reason, and act** to accomplish a goal. We also used the **Minimax** algorithm and **heuristics** to help an AI find a good next action quickly.

Upcoming Topics

Our final unit will address how computer science affects the world by diving into **history**, exploring questions regarding **ethics** in the present day, and looking forward at the **future**.

We'll wrap up the course with a single lecture exploring different opportunities in the **School of Computer Science**.

Matplotlib

Coding with Matplotlib

Writing code with Matplotlib isn't like writing code to solve a homework problem.

Instead of starting from scratch, it's best to **start from an example**, then modify the example's code to work for the data you're trying to chart.

Why is this different? The Matplotlib library is **huge**. It isn't efficient to memorize every possible function- it's better to look up functions when you need them.

Example: Making a Scatter Plot

Goal: we want to visually compare the most popular ice cream flavors in our dataset to determine if there are any interesting trends in which flavors were chosen most often as the 1st vs 2nd favorite flavor.

Start with the code from the Data Analysis II lecture that creates a dictionary from the ice cream flavors, but change it to map each flavor to a new dictionary that separates the counts of 1st favorite, 2nd favorite, and 3rd favorite for each flavor.

Scatter Plot Demo

If we go to the Matplotlib examples page (https://matplotlib.org/gallery/index.html) and search for Scatter Plot, we'll find an example: https://matplotlib.org/gallery/shapes_and_collections/scatter.html#sphx-glr-gallery-shapes-and-collections-scatter-py

This shows that the function we need is `plt.scatter()`. We can read about its arguments here: https://matplotlib.org/api/_as_gen/matplotlib.pyplot.scatter.html#matplotlib.pyplot.scatter

Reading Documentation

Documentation provides descriptions of what the functions in a library do. They also provide information about which arguments are **required** vs **optional**.

An **optional parameter** is one that can be included in a function call, or left alone. If they aren't included, they're set to **default values** instead.

Matplotlib uses a lot of optional parameters to allow for customization of charts!

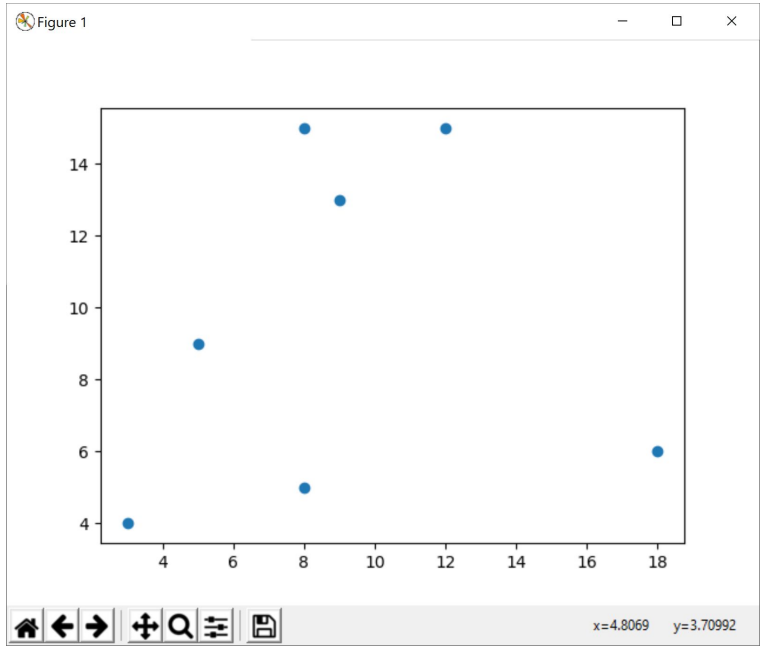
Setting up a Basic Chart

The core `plt.scatter()` function requires two lists - one of x values, one of y values. We'll need to **reformat** the data to make those lists.

```
flavors = []
firstRanks = []
secondRanks = []
for flavor in d:
    if d[flavor][1] + d[flavor][2] + d[flavor][3] >= 10:
        flavors.append(flavor)
        firstRanks.append(d[flavor][1])
        secondRanks.append(d[flavor][2])
```

Once we've done that, we can create a simple scatter plot!

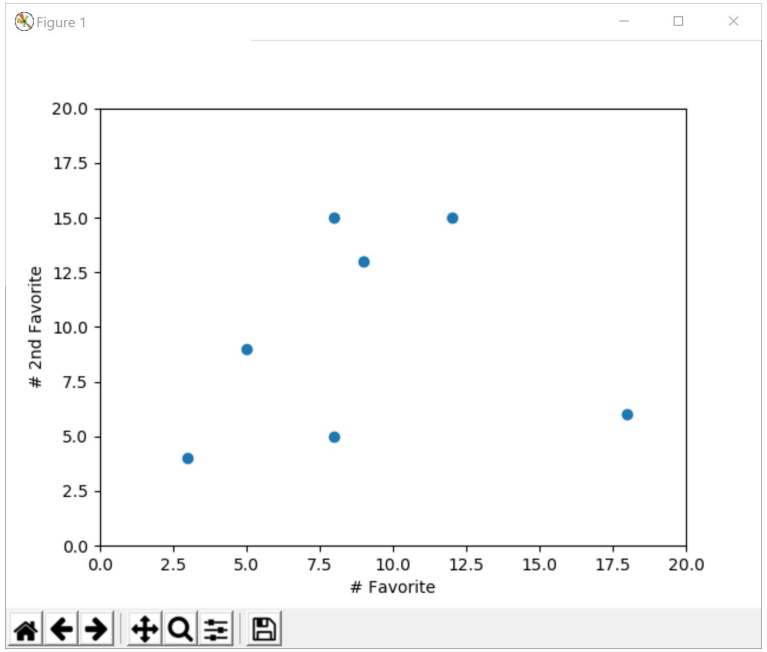
```
import numpy as np
import matplotlib.pyplot as plt
plt.scatter(firstRanks, secondRanks)
plt.show()
```



Modifying the Axes

If we want to make the chart look nicer, we can use the **figure** and **axes** components of the plot to add more information.

Let's add axis labels and set the ranges from 0 to 10 each. Reading the documentation shows how: https://matplotlib.org/api/axes_api.html#matplotlib.axes.Axes




```
fig, ax = plt.subplots()

plt.scatter(firstRanks, secondRanks)

ax.set_xlabel("# Favorite")
ax.set_ylabel("# 2nd Favorite")
ax.set_xlim(0, 20)
ax.set_ylim(0, 20)

plt.show()
```

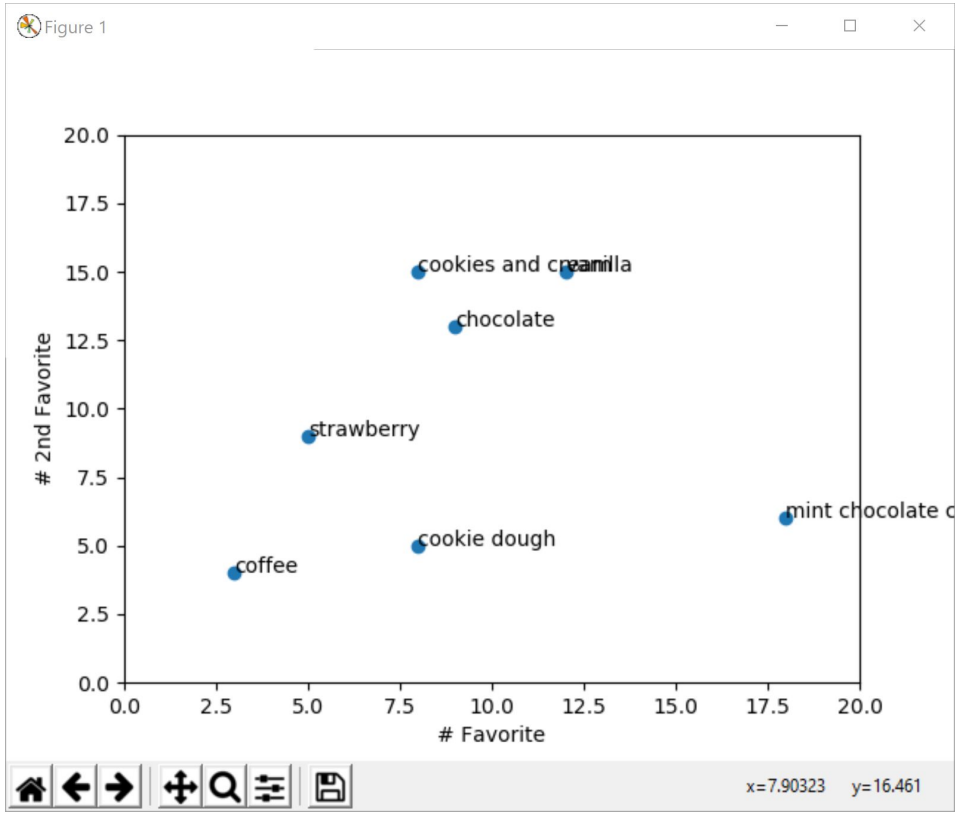
Adding annotations

Finally, how can we add text labels to the data points?

```
for i in range(len(flavors)):
    ax.annotate(flavors[i], [firstRanks[i], secondRanks[i]])
```

Searching 'annotate' on the page reveals the function `annotate()`. Go to that function's page to learn more, and implement it!

Note- we'll have to call the function for each individual label.



Heuristics

Heuristics Support Good-Enough Search

Recall that we defined **heuristics** as algorithmic techniques that let us search a large set of data quickly by searching for a **good enough** answer instead of the **best** answer.

In Travelling Salesperson, we do this by always choosing the shortest edge that leads to an unvisited node. This means we only need to generate one path, instead of O(n!) paths.

In Tic-Tac-Toe, we used heuristics to **cut off** the game tree at a certain level and evaluate each state immediately instead of continuing all the way down to the end states.

Design Heuristics to Score Possibilities

In both of our previous examples, the heuristic let us **score** the possible choices so that we could compare them directly.

This approach only works if we **design the heuristic well**. The score that the algorithm assigns must be a good representation of the probability that the state is the best choice to make.

How can we design heuristics well? Try to map all of the information contained in the state to a number- the larger, the better!

Example: AI for Connect Four

Consider the game Connect Four. Players alternate in placing discs in the bottom-most open position in one of the columns. The first player to get four in a row- horizontally, vertically, or diagonally- wins.

If we developed an AI to play Connect Four, it might create a game tree to decide its next move. It will have 7 choices for which column to place a disc in each turn (or fewer if one of the columns is full), and there are 7*6 = 42 total moves. That's approximately 7^42 different end states- too big!! A heuristic will help here.

Activity: Heuristic for Connect Four

Let's design a heuristic to assess a game state of Connect Four together.

A state where the AI has four in a row scores a 1. A state where the user has four in a row scores a -1. A state where every slot has been filled and no one won scores a 0.

You do: what other features should we assess?

Reading/Writing Files

Opening Files

To read a file from your computer into a string, you first need to call the `open()` built-in function with the **filepath** of the file.

If the file is in the same directory as the Python file, the filepath is just its name. If it's in a folder, use `"folder/name"` instead, with `"/"` separating the folder's name from the file's name.

This can be repeated for nested folders - for example, `"project/data/icecream.csv"`.

Reading Files

Once you've opened a file properly, you'll have a **file object** stored in a variable. The simplest way to get data from that file object is to call the `read()` method on it:

```
text = f.read()
```

This reads *all* the text from the file into a string, and stores that string in the variable. You can then parse the variable to separate lines or parts of lines as needed.

Monte Carlo Methods

Monte Carlo Methods Find Probable Results

We use **Monte Carlo methods** to find the probable/average result of an event that involves randomness.

We do this by running the event over and over again across many **trials** and averaging the results.

If we run enough trials, we should get a reasonable degree of accuracy in the answer.

Example: Poker Odds

Let's build a simple Monte Carlo experiment to calculate the odds of getting certain types of hands in Poker. We need to generate a deck of cards. Represent each card as a two-element list - a suit (string) and value (number) - with Jack/Queen/King represented as 11/12/13. To keep things simple, we'll rank Aces as high, as a score of 14, and ignore the cases where we'd want Aces to be 1.

```
def generateDeck():
    deck = []
    for suit in ["Club", "Diamond", "Heart", "Spade"]:
        for value in range(2, 15): # 2 to Ace (14)
            deck.append([suit, value])
    return deck
```

Example: Poker Odds

For each trial, shuffle the deck, then draw the first five cards as the hand.

```
def calculateOdds(trials):
    count = 0
    deck = generateDeck()
    for trial in range(trials):
        random.shuffle(deck)
        hand = deck[:5] # first five cards
        if isFlush(hand):
            count += 1
    return count / trials
```

Example: Poker Odds - Flush

How do we know if the five cards form a flush? Check if each card's suit matches the first card's suit.

```
def isFlush(hand):
    suits = []
    firstSuit = hand[0][0]
    for card in hand:
        if card[0] != firstSuit:
            return False
    return True
```

Example: Poker Odds - Straight

How do we know if the five cards form a straight? If we sort the five values, they should be continuous- each one larger than the previous one.

```
def isStraight(hand):
    values = []
    for card in hand:
        values.append(card[1])
    values.sort()
    for i in range(len(values)-1):
        if values[i] != values[i+1] - 1:
            return False
    return True
```

Time-based Simulation

Model, View, Controller

Recall that our simulation framework separates the simulation into three parts.

Model: represents the current state of the simulation (as a dictionary mapping key-names to value-values).

View: draws the state of the world graphically

Controller: modifies the state of the world based on **time passing** or **events**.

Time-Based Simulation

For **time passing**, we use a function that is repeatedly called by the framework every time a certain amount of time has passed.

By adding code to this function (`runRules`), we can make the simulation smoothly update over time.

The function will need to change **the model's dictionary** for the effects to stick over time.

Example: Generating Bubbles

Let's program a basic simulation that makes bubbles appear in random locations as time passes.

At any given moment, the **state** of the simulation is the number of bubbles, and where they're located. So the bubbles will need to be stored in the **model**. We start with no bubbles, so start with an empty list.

```
def makeModel(data):
    data["bubbles"] = []
```

Time Controller

Every time a certain amount of time passes, we want to add a single bubble to the screen. Let's represent a bubble as an x,y position and a color, three values in a list.

To do this, **add a single bubble to the model**. The graphical part will be handled separately, in the view. Note that we **must** update the `data` variable to save the changes.

```
def runRules(data, calls):
    x = random.randint(0, 400)
    y = random.randint(0, 400)
    color = random.choice(["red", "green", "blue"])
    bubble = [x, y, color]
    data["bubbles"].append(bubble)
```

View

To see the model, we'll need to implement the **view**. The canvas is constantly erased and re-painted with the current state of the model, so that it is always up-to-date.

```
def makeView(data, canvas):
    for bubble in data["bubbles"]:
        x = bubble[0]
        y = bubble[1]
        color = bubble[2]
        r = 20
        canvas.create_oval(x - r, y - r, x + r, y + r, fill=color)
```

Minimax

Minimax Searches a Game Tree

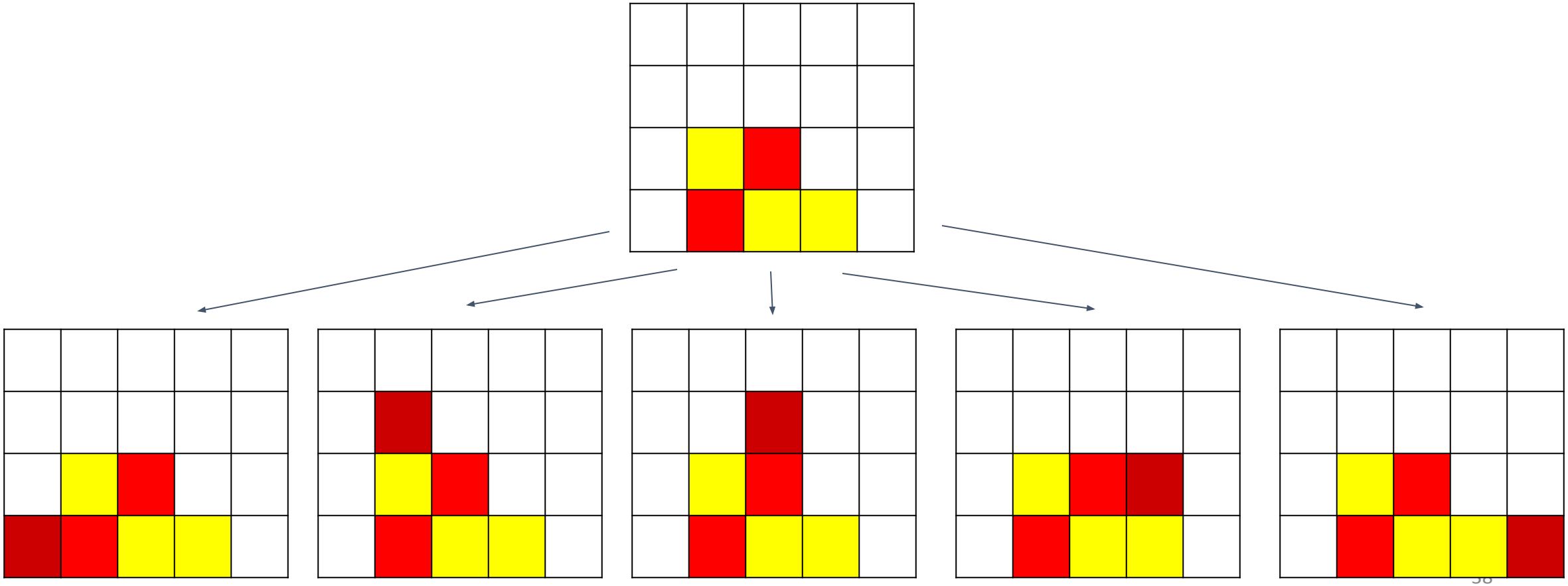
Minimax is a type of search algorithm that finds the **best next move** in a game tree.

The game tree represents all the possible routes the game could take. Minimax tries to find the best one by picking the best option at each point for the AI, and assuming that the user will always pick the best option for themselves.

It accomplishes this by choosing the **maximum** of all possible outcomes when it's the AI's turn, and the **minimum** of all possible outcomes when it's the user's turn. Leaves are labeled with a score based on the outcome (AI win = 1, User win = -1, Tie = 0).

Connect Four Minimax

In a simplified Connect Four board (5 columns x 4 rows), a game tree might start like:



Applying Minimax

To apply minimax, you would need to **complete** the Connect Four tree, or cut it off at a certain point and use a heuristic.

Once the leaves / cut-off nodes have been scored, you can use Minimax to find the scores of the **parents** of the scored leaves.

This process can be repeated until the original root node's children have scores. Then the AI can choose to take an action that leads to the optimal child.

Choosing an Action

For example, if the children were scored as shown below, the AI should choose the 2nd or 3rd option.