CS 360: Machine Learning

Sara Mathieson, Sorelle Friedler

Spring 2024

HAVERFORD COLLEGE

Sit somewhere new!

Admin

- **EVERYONE:** Sign in again
- Sorelle office hours **Thursday: 4-5pm in H110**
- **Lab 1** was due last night
- **Lab 2** due Thursday Feb 8
  - Don't wait til the last minute!
- **TA hour schedule** on Piazza

Outline for Jan 30

- Python style and implementation notes
- Overfitting
- K-nearest neighbors
- KD Trees

Logistic Regression and Gradient Descent Review:
Moved to discussion of softmax

Outline for Jan 30

- Python style and implementation notes
- Overfitting
- K-nearest neighbors
- KD Trees

Python style

- Decompose code into natural functions
- Avoid global variables (sometimes useful)
- Include a file header with purpose, author, and date
- Include headers for each function
- No lines over 80 chars
- Variable names implicitly show type
- Include line breaks and comments!

Python style

- "Snake-case" not "camel-case"
  - ~~linearSearch~~
  - **linear_search**

- Alphabetize imports and don't use "*"
  - ~~from numpy import *~~
  - **import numpy as np**

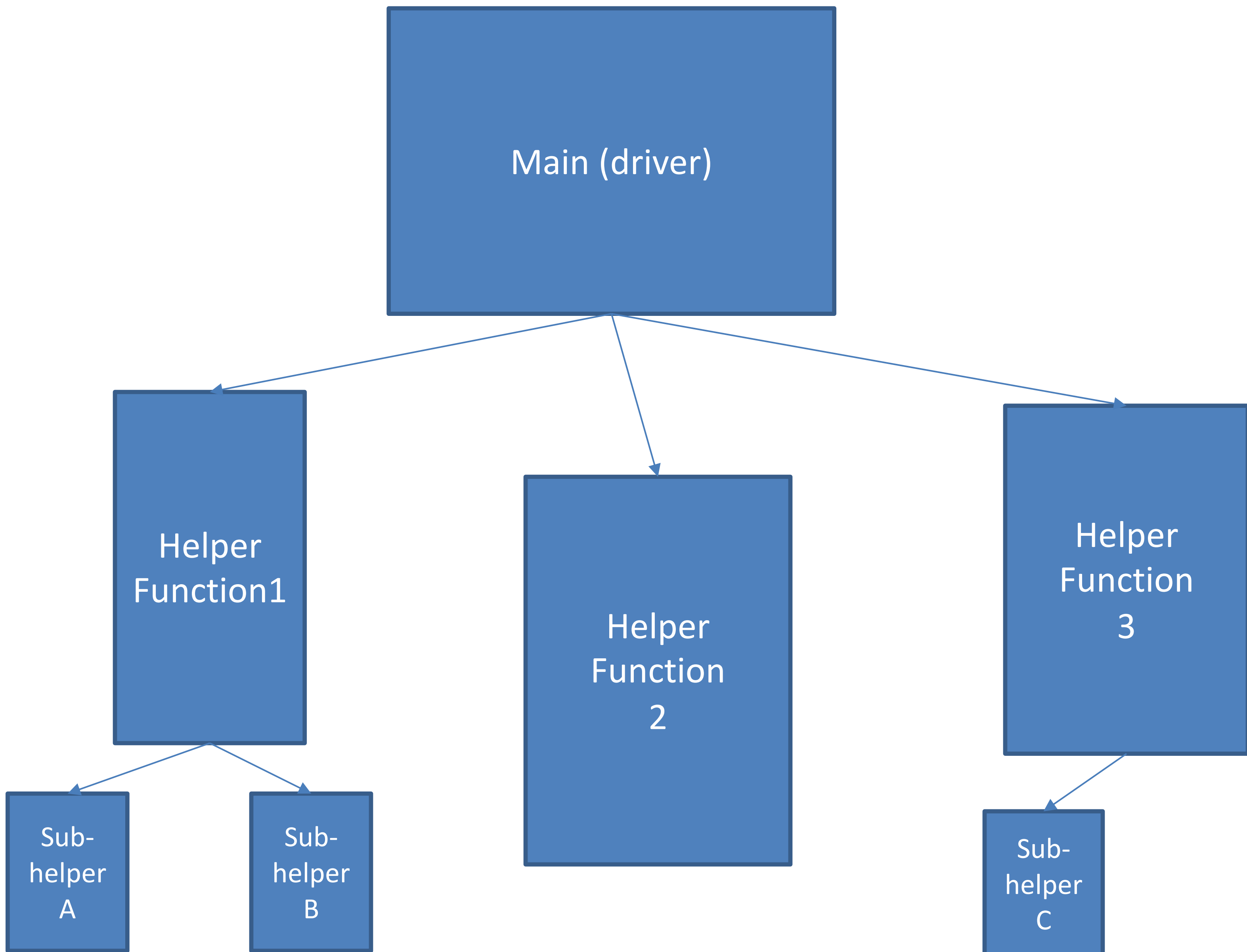
Python style examples

```
'''
Author      : Allison Gong
Class       : CS260
Date        : 7/30/21
Description : This is the main driver program for the Introduction to Python
portion of Lab 1. This intro consists of a series of coding exercises that cover
some of the core syntax, data structures, and functionality of Python.
'''
```

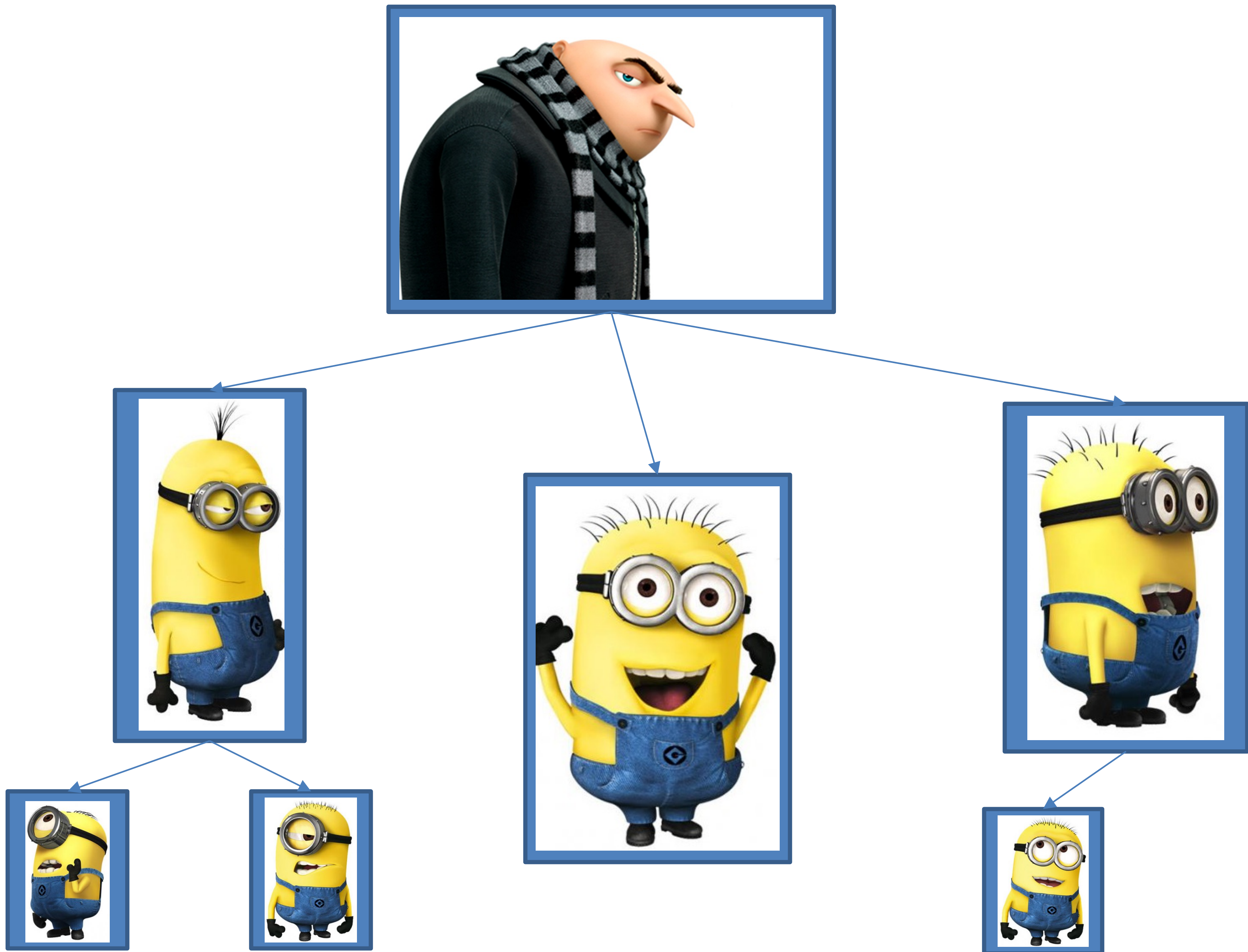
```
# add up 10 random numbers
total = 0
for i in range(10):
    total += random.randrange(6) # includes 0, excludes 6
print("sum:", total)
```

```
def fib(n):
  '''
  Compute and return the nth Fibonacci number.
  n: non-negative integer
  return: nth Fibonacci number
  '''
  # code here
```

Structure of main and "helper" functions



Structure of main and “helper” functions



```python
"""
Given an input phrase and a letter, count how many times that letter appears
in the phrase. For example:

phrase: creative code
letter: e
Number of e's: 3

Author: Jeff Knerr & Sara Mathieson
Date: 9/21/18
"""

def main():

    # ask the user for a phrase and a letter
    phrase = input("phrase: ")
    letter = input("letter: ")
    num_chars = len(phrase)

    # set up accumulator variable count
    count = 0
    for i in range(num_chars):
        # add on 1 each time we see the desired letter
        if phrase[i] == letter:
            count = count + 1

    # example of string formatting (%s for str, %i for int)
    print("Number of %s's: %i" % (letter, count))

main()
```

Reminder: steps of top-down-design (TDD)

Reminder: steps of top-down-design (TDD)

1) Design a **high-level main function** that captures the basic idea of the program.

Reminder: steps of top-down-design (TDD)

1) Design a high-level main function that captures the basic idea of the program.

2) As you're writing/designing main, think about which details can be abstracted into small tasks. Make names for these functions and write their signatures below main.

Reminder: steps of top-down-design (TDD)

1) Design a high-level main function that captures the basic idea of the program.

2) As you're writing/designing main, think about which details can be abstracted into small tasks. Make names for these functions and write their signatures below main.

3) "Stub" out the functions. This means that they should work and return the correct type so that your code runs, but they don't do the correct task yet. For example, if a function should return a list, you can return []. Or if it returns a boolean, you can return False.

Reminder: steps of top-down-design (TDD)

1) Design a high-level main function that captures the basic idea of the program.

2) As you're writing/designing main, think about which details can be abstracted into small tasks. Make names for these functions and write their signatures below main.

3) "Stub" out the functions. This means that they should work and return the correct type so that your code runs, but they don't do the correct task yet. For example, if a function should return a list, you can return []. Or if it returns a boolean, you can return False.

4) Iterate on your design until you have a working main and stubbed out functions. Then start implementing the functions, starting from the "bottom up".

Outline for Jan 30

- Python style and implementation notes
- Overfitting
- K-nearest neighbors
- KD Trees

Overfitting with a high-degree polynomial

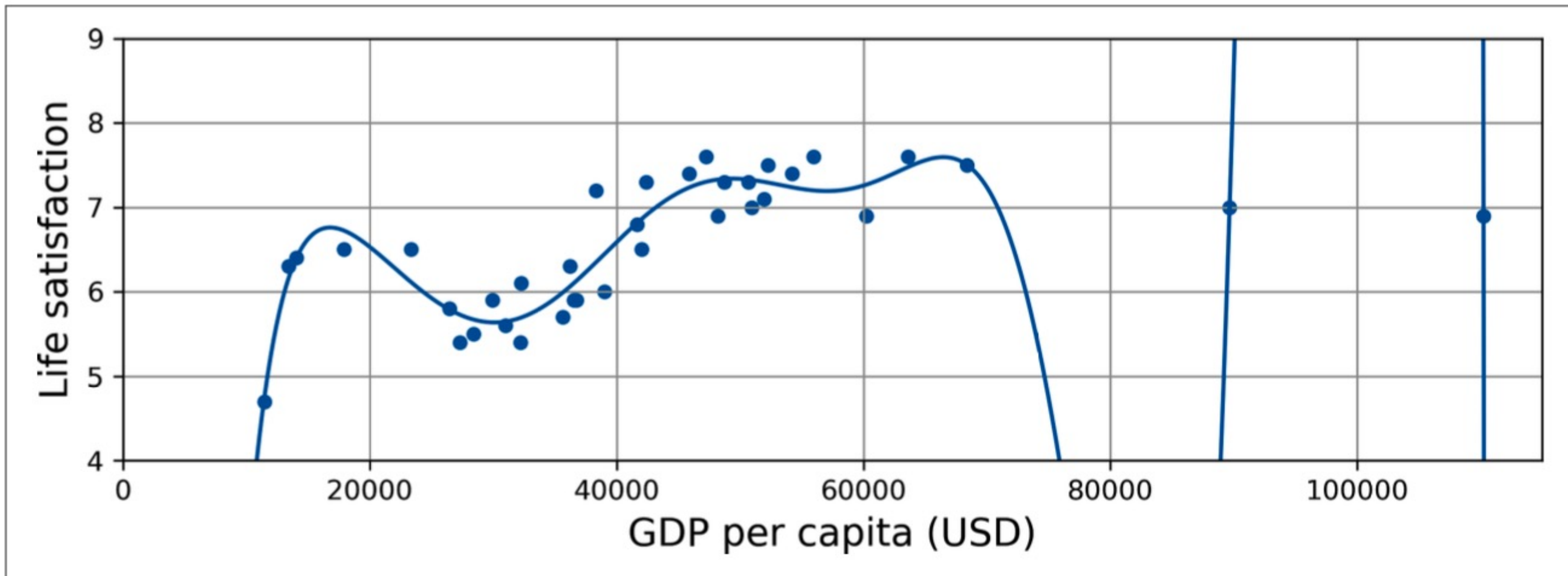


Figure 1-23. Overfitting the training data

Geron: Figure 1-23

Overfitting

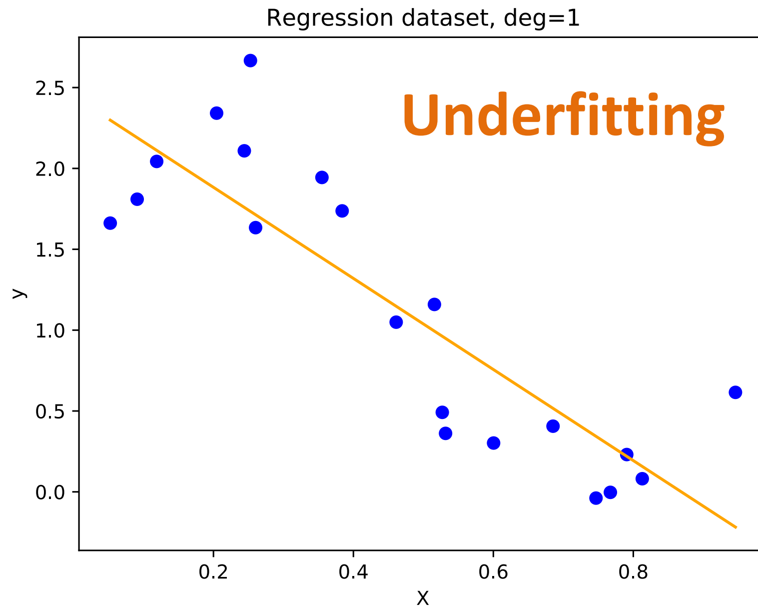
- "***Overfitting*** happens when the model is too complex relative to the amount and noisiness of the training data." (Geron Chap 1)

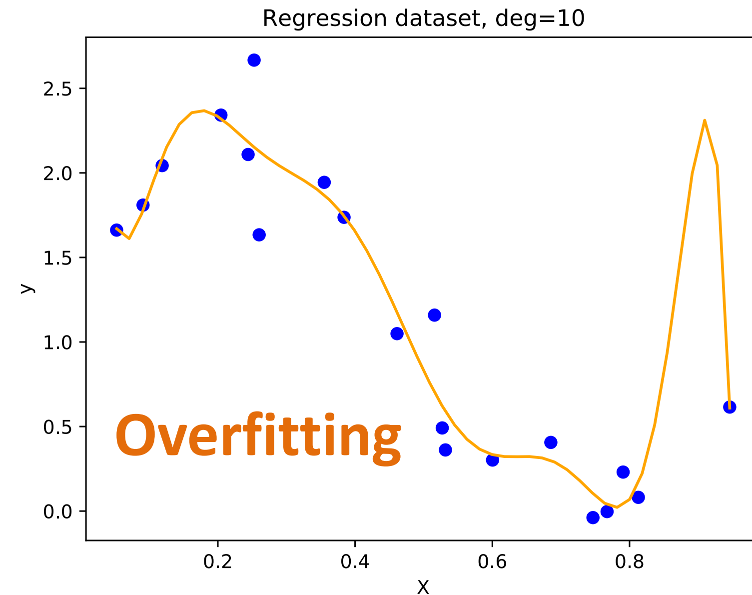
- **Solutions**
  - Reduce the complexity of the model
  - Get more training data
  - Reduce noise in the training data

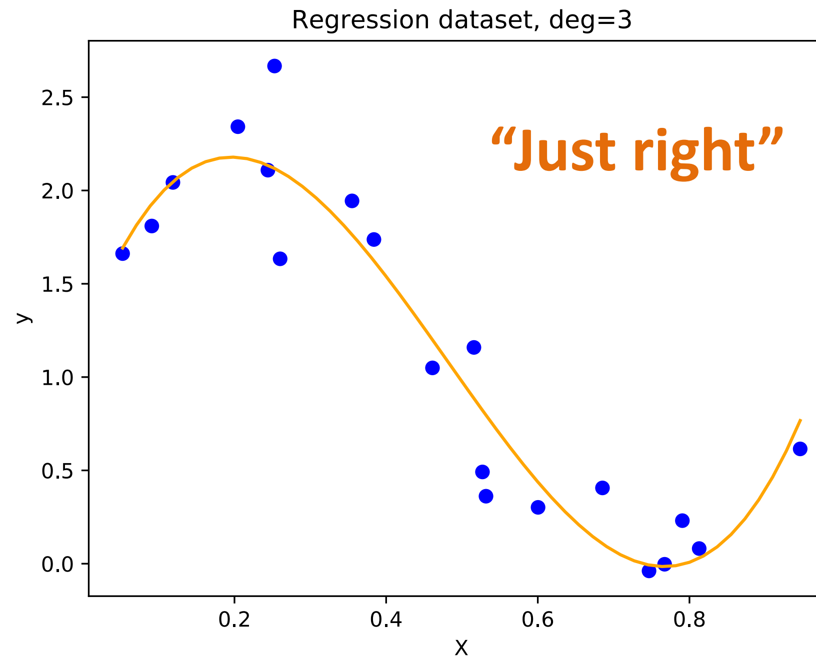
Terminology

- *Underfitting*: "had the opportunity to learn something but didn't" (Duame)
- *Overfitting*: memorized individual training examples (fit to noise) and can't generalize

Under and over-fitting (CS260 example)

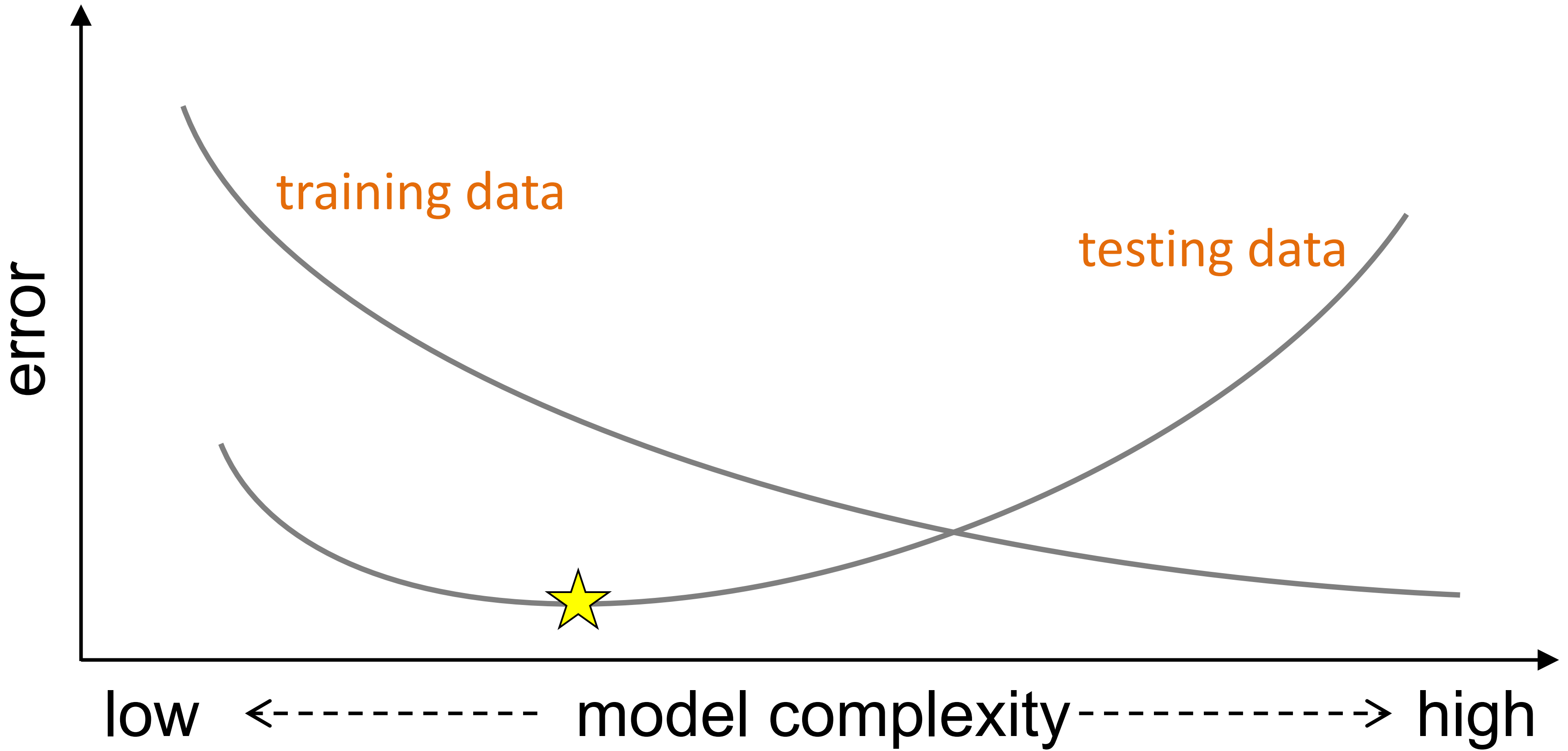






Geron: Figure 1-23

Common pattern observed in machine learning



Validation data

- Is it wrong to use the test data to determine the model complexity?
- **Yes!**

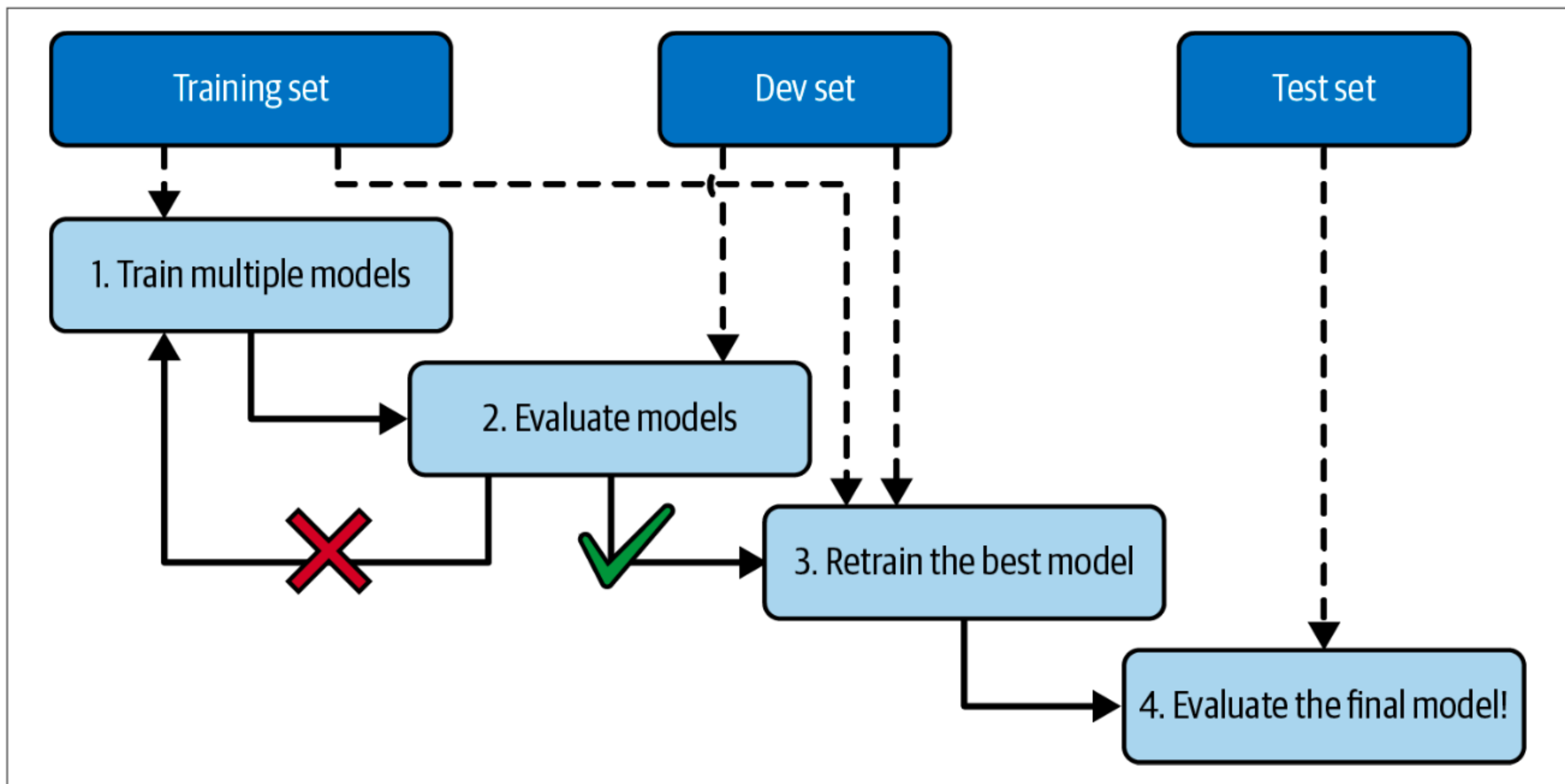


Figure 1-25. Model selection using holdout validation

Geron: Figure 1-25

Outline for Jan 30

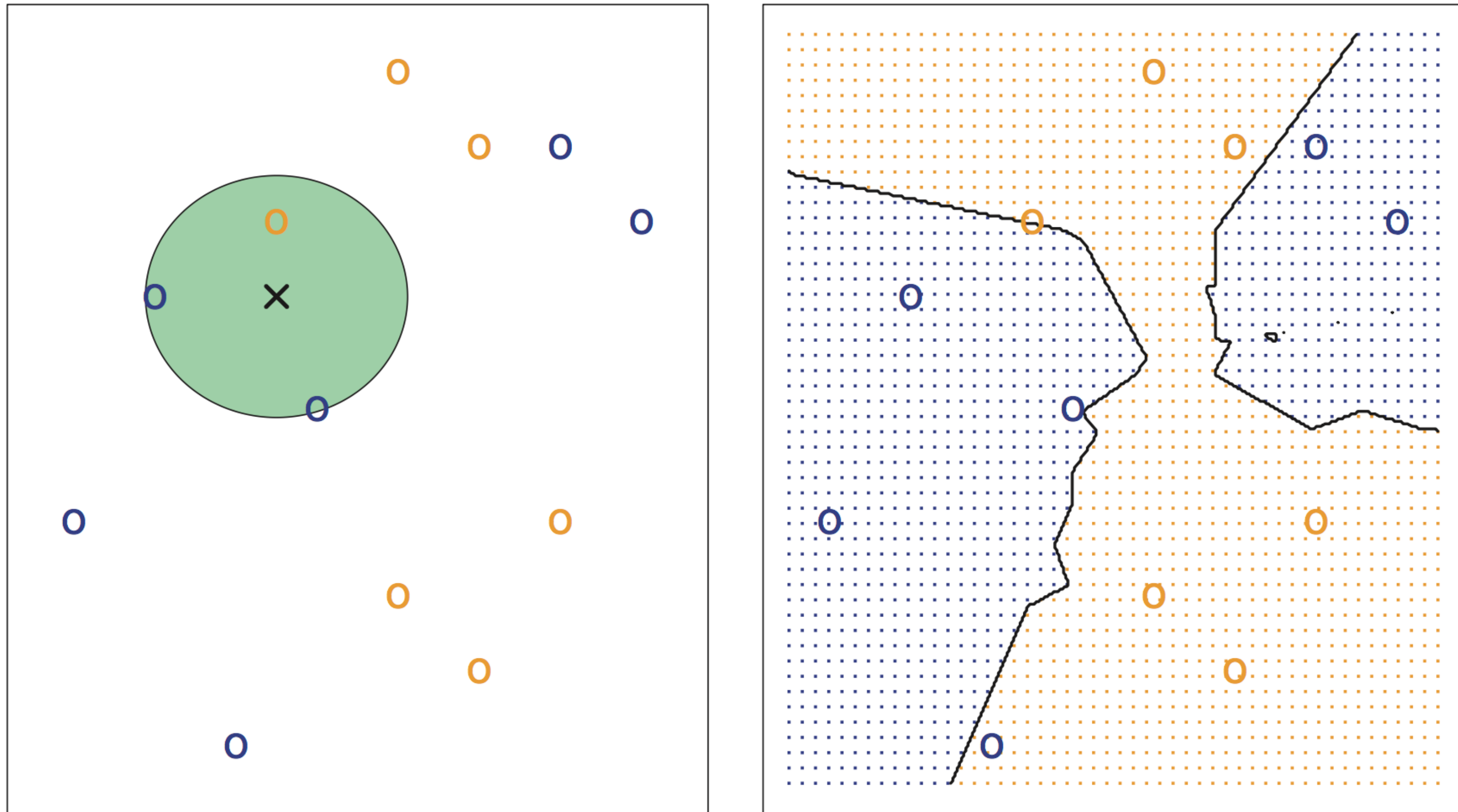
- Python style and implementation notes
- Overfitting
- K-nearest neighbors
- KD Trees

Nearest Neighbors

Why would we be interested in finding a point's nearest neighbor in a set of points?

- Fill in missing data
- Prediction unknown values (labels, output, etc)
- subroutine for some clustering methods

K-nearest neighbors creates implicit decision boundaries



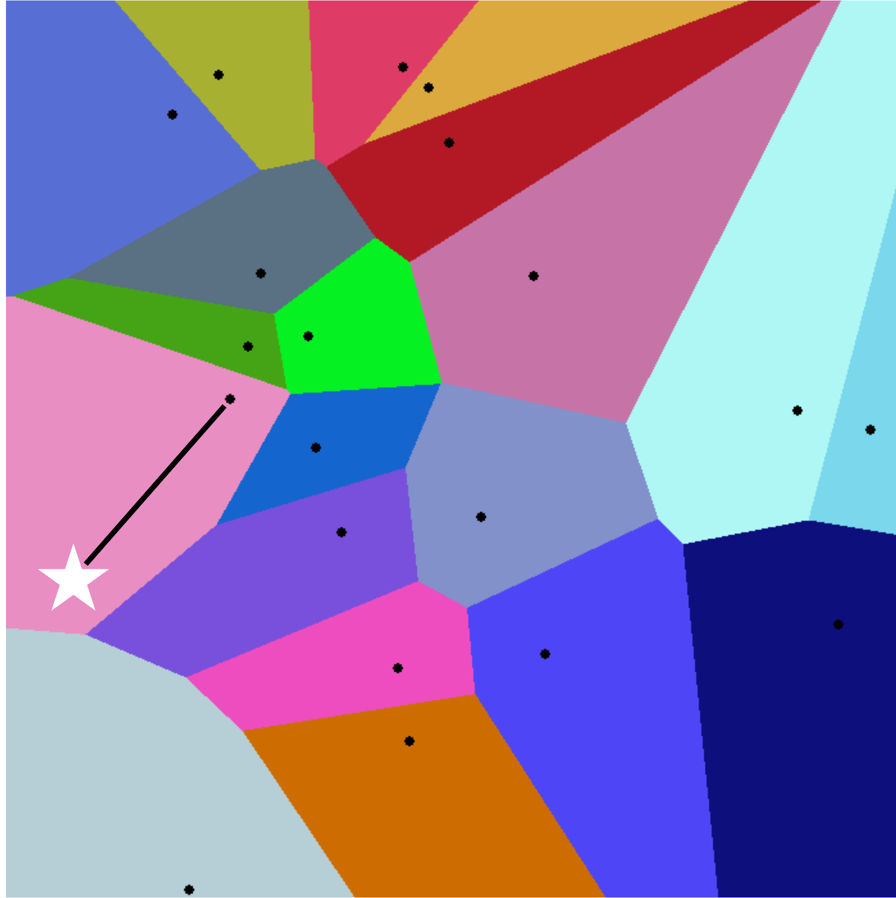
Decision boundary: separates regions of the feature space that would be classified as positive or negative (or multiclass)

Figure 2.14 from ISL book, KNN with two classes (C=2), and K=3

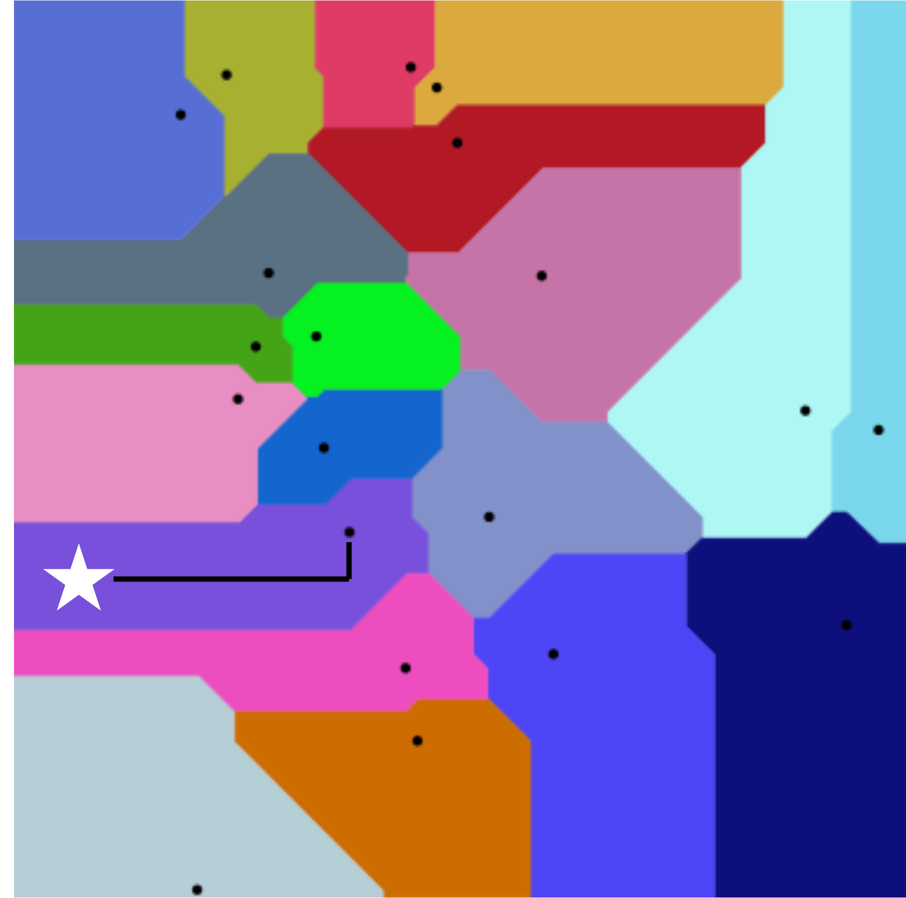
Voronoi Diagrams

Nearest neighbor queries in 2D

☆ query point



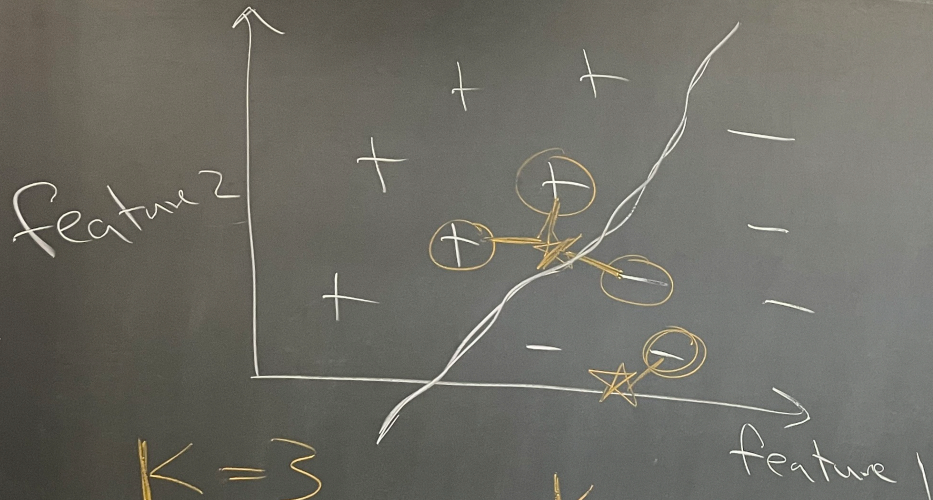
Euclidean distance



Manhattan distance

Images: wikipedia, Slide: modified from Sorelle Friedler

KNN for binary classification



K=3

label = (+)

K=1

label => (−)

Naive KNN

knn(points, q)

training data

query

```
for p in points:
    if d(p, q) < min
        min = d(p, q)
        nn = p
```

① runtime if len(points) = n

$\Rightarrow O(n)$ ignoring # dims

② what about higher K?

Euclidean dist

$a = (a_1, a_2 \cdots a_d)$

$b = (b_1, b_2 \cdots b_d)$

$d_e(a,b) = \sqrt{(a_1 - b_1)^2 + (a_2 - b_2)^2 \cdots\cdots}$

↙ Manhattan

$d_m(a,b) = |a_1 - b_1| + |a_2 - b_2| \cdots\cdots$

Comparison of decision boundaries

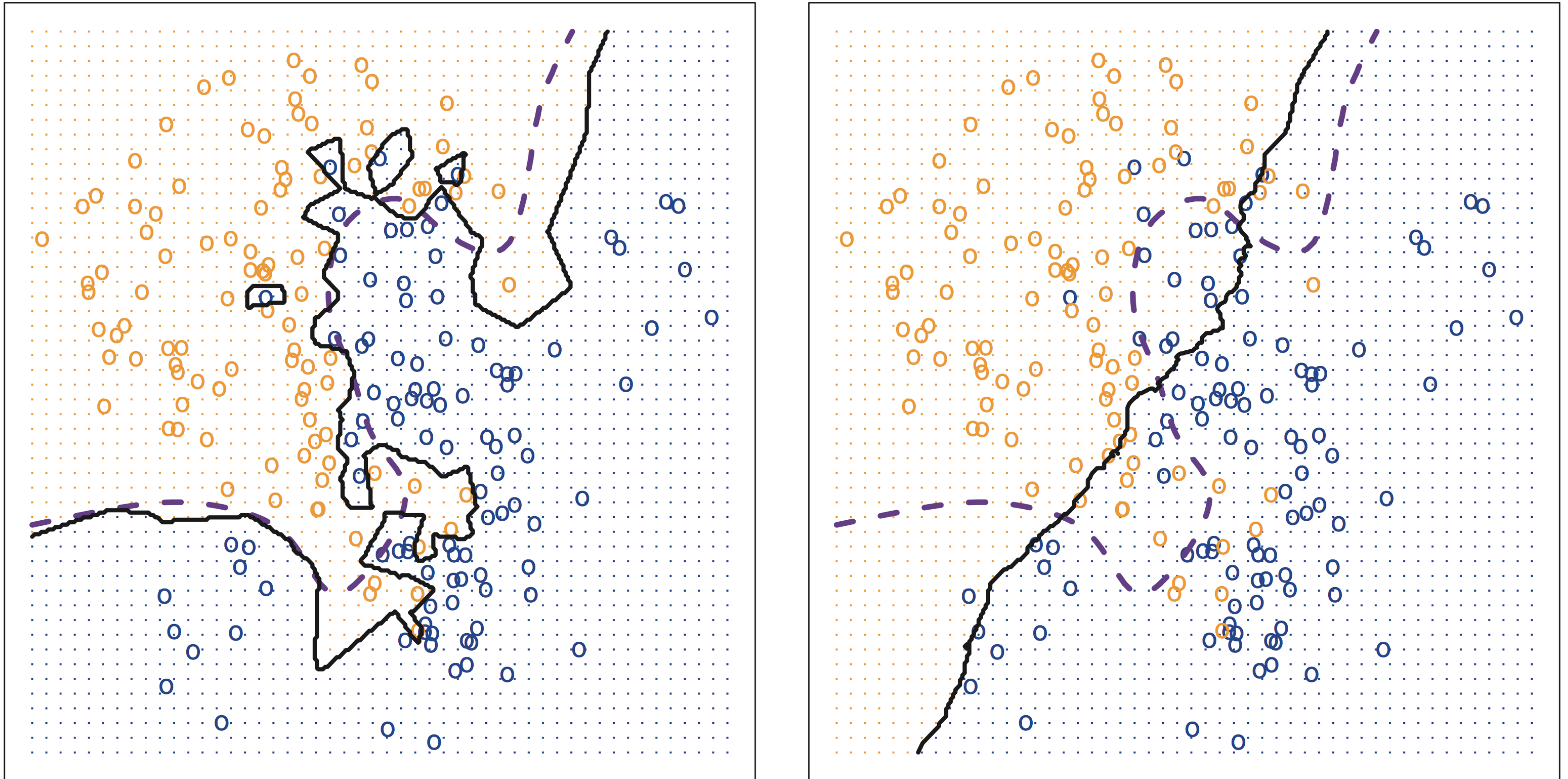
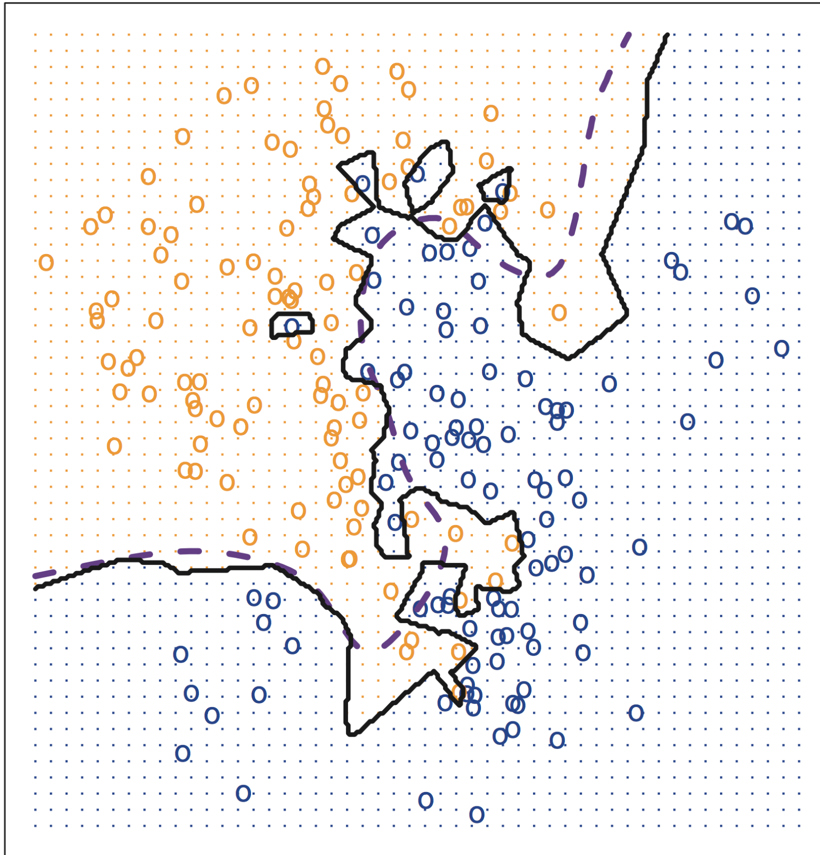


Figure 2.16 from ISL book (dashed line is “ideal” boundary)

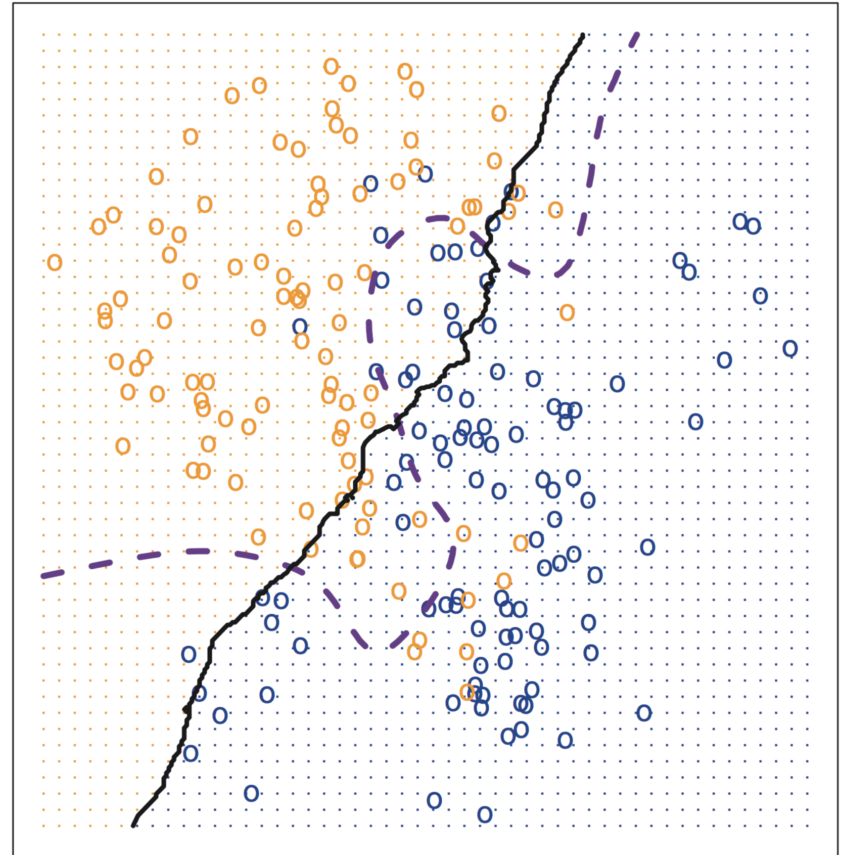
Comparison of decision boundaries

KNN: K=1



Overfitting

KNN: K=100



Underfitting

Figure 2.16 from ISL book (dashed line is “ideal” boundary)

Calculating the nearest neighbor

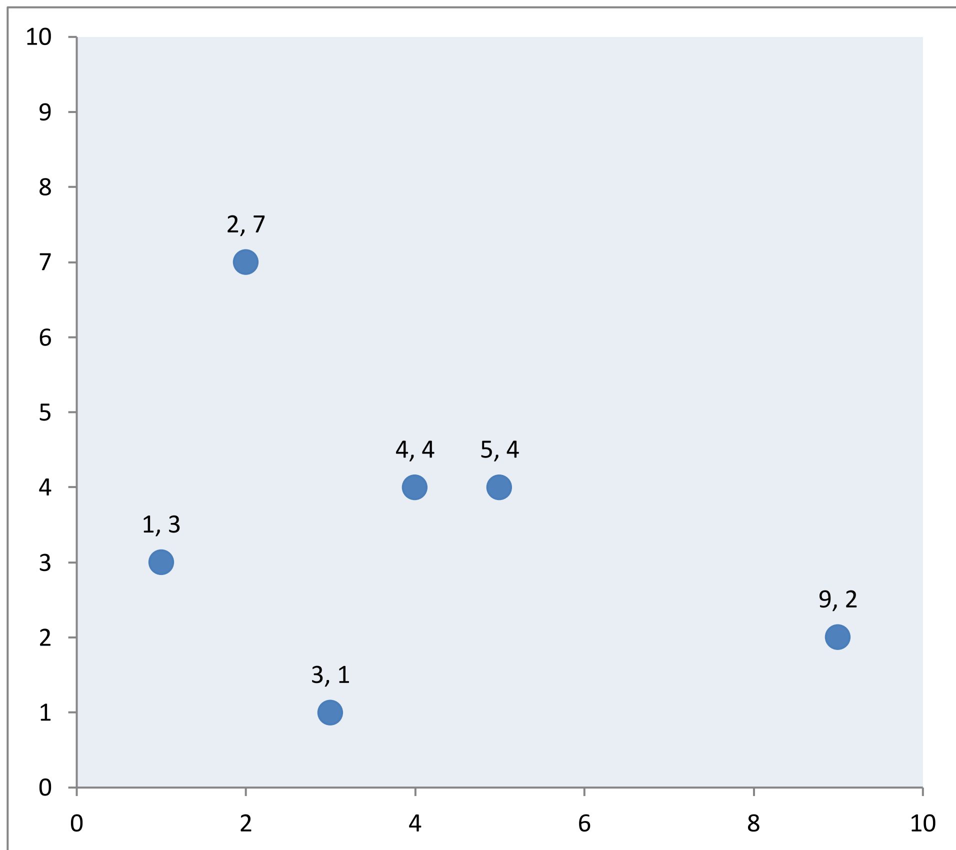
- What is the "naïve" approach?
- How long does it take to find the nearest neighbor of a point? In 2D? In d-dimensions?
- How could we do better?

Slide: modified from Sorelle Friedler

Outline for Jan 30

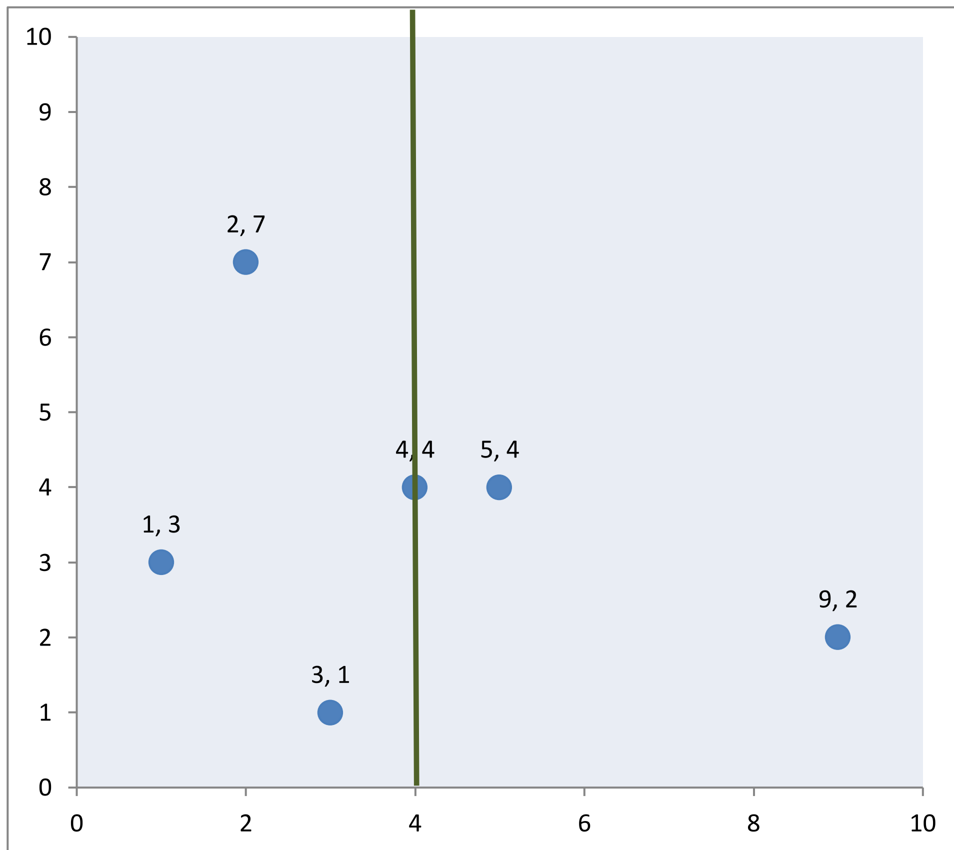
- Python style and implementation notes
- Overfitting
- K-nearest neighbors
- KD Trees

Making a kd-tree



Slide: modified from Sorelle Friedler

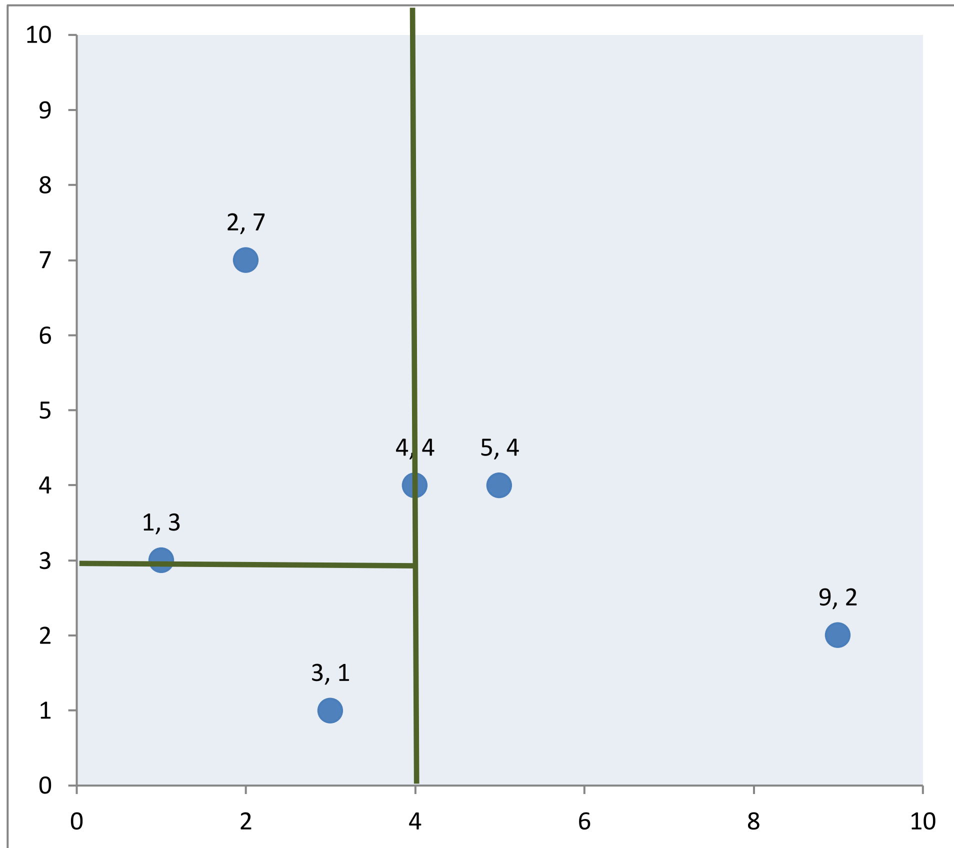
Making a kd-tree

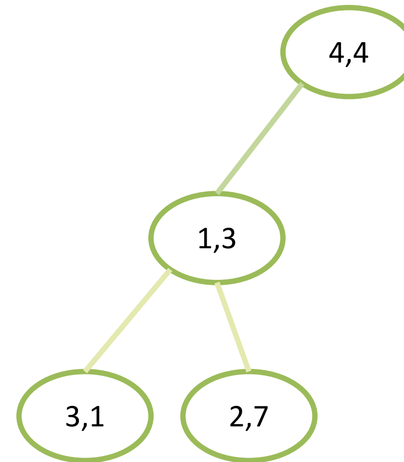


4,4

Slide: modified from Sorelle Friedler

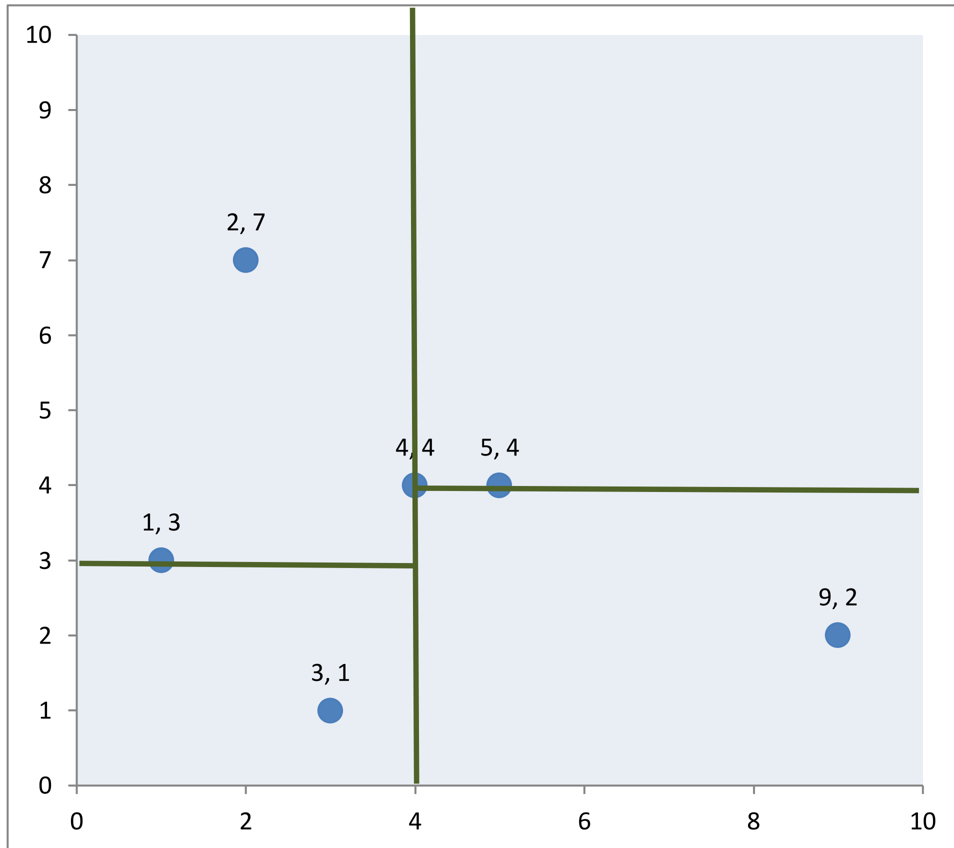
Making a kd-tree

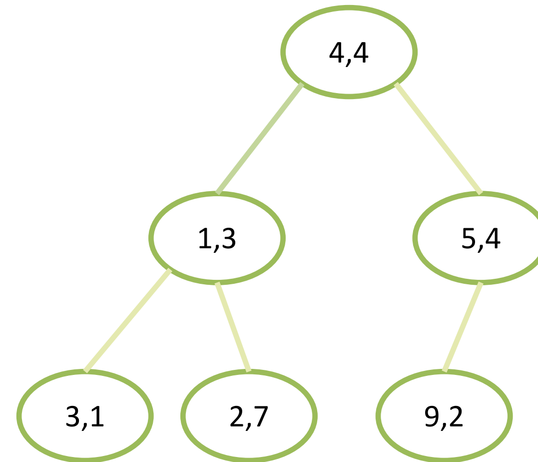




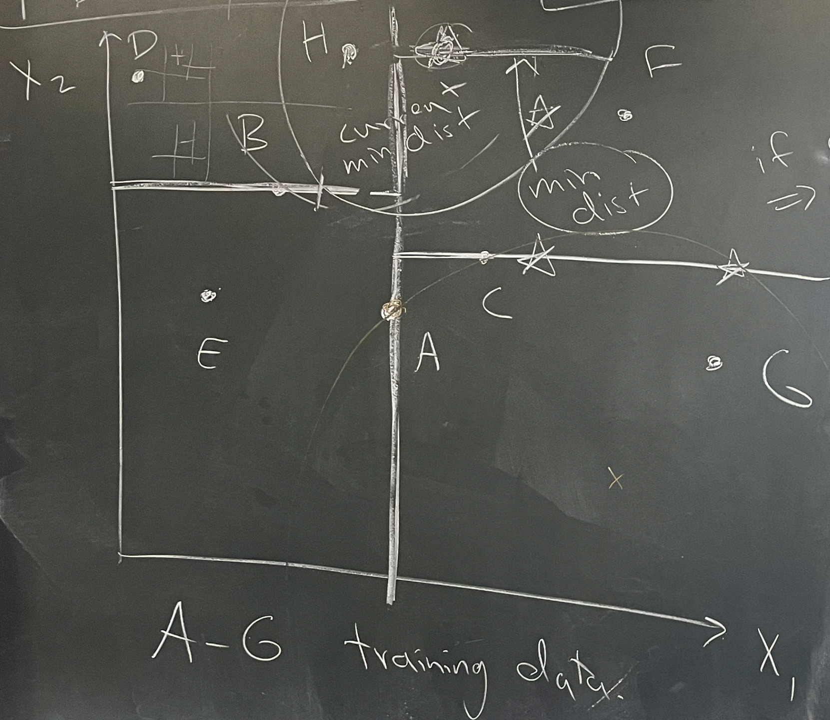
Slide: modified from Sorelle Friedler

Making a kd-tree





Slide: modified from Sorelle Friedler



kdtree(points, depth)

dim = depth % d

sort points & take median

make node at median

node.point = median

node.left = kdtree(points on the left, depth+1)

node.right = kdtree(points on right, depth+1)

return ?

kdtree()

split

kd-tree

dim=0 D E B A C F G (sort in dim=0)

dim=1 E B D

dim=1 G C F